Trace theory and VLSI design

TRACE THEORY AND VLSI DESIGN

Jan L.A. van de Snepscheut

TRACE THEORY AND VLSI DESIGN

TRACE THEORY

AND

VLSI DESIGN

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR. S. T. M. ACKERMANS, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP VRIJDAG 14 OKTOBER 1983 TE 16.00 UUR

door

JOHANNES LAMBERTUS ADRIANA VAN DE SNEPSCHEUT
GEBOREN TE OOSTERHOUT, NOORD-BRABANT

dit proefschrift is goedgekeurd
door de promotoren
Prof. dr. M. Rem
en
Prof. dr. E. W. Dijkstra

Aan Trees,
Suzanne
en Marieke

Contents

0. Introduction

We are interested in programs suggesting implementations with a high degree of concurrency. This interest is dictated by three observations. First, the design of such programs is very difficult. By trying to deal with them we may hope to become better programmers in general. Second, the study of implementation methods is often based on sequential programs only. The problems arising from concurrency are, however, bound to be interesting from a mathematical point of view. Third, the advent of techniques for constructing machinery operating with a high degree of concurrency clamors for the study of suitable implementation methods.

In this monograph we discuss a method to implement programs as semiconductor chips. We propose a notation for programs that prescribe the cooperation of components arranged in a hierarchical structure. Hierarchical structure assists in bridling the complexity of program design. The components and the way in which they cooperate are defined such that an implementation potentially exhibits a high degree of concurrency.

VLSI is a technique of constructing semiconductor chips containing a large number of active, electronic elements. Since these elements operate concurrently, VLSI nicely matches the concurrency suggested by the program notation. One of the problems in chip design is that increasing the scale factor, i.e. decreasing the feature size, leads to a decrease of the propagation speed of electrical signals relative to the switching speed. Employing suitable communication protocols, the implementation method proposed yields chip designs whose correct operation is independent of the propagation speed.

In the case of sequential programming, the specification of programs, and of the statements from which they are formed, is conveniently given in terms of pre- and postconditions describing initial and final states. The convenience is caused by the sequential composition of statements: the final state of a statement is the initial state of the next statement. With concurrent programming this is no longer appropriate. Though concurrent composition may be viewed as "sequential composition in any order" in the case of atomic statements, this view is not applicable to composite statements. In the latter case, the phrase "any suitably synchronized interleaving" is more appropriate. Consequently, we choose to give the specification of programs, and of the components from which they are formed, in terms of finite-length sequences of atomic statements. Such a sequence is called a trace. The atomic statements are called symbols. The specification of a component then consists of a trace set (a set of traces) and an alphabet (a set of symbols). The symbols in the traces are chosen from the alphabet. The combination of a trace set and an alphabet is called a trace structure. When forming a component from subcomponents, the trace structure of the component is a function of the trace structures of the subcomponents.

In the first chapter a theory of trace structures and composition functions is discussed. In chapters two and three a program notation is proposed and the relation with finite state machines is discussed. The fourth chapter contains a discussion of some aspects of VLSI design and in the fifth chapter a method of deriving semiconductor chips from programs is proposed.

In this monograph a slightly unconventional notation for variable-binding constructs is used, which will be explained

here informally. Universal quantification, also called continued conjunction, is denoted by

$$\underline{A}(l : D : E)$$

where $\underline{A}$ is a quantifier, l is a list of bound variables, D is a predicate, and E is the quantified expression. Both D and E will, in general, contain variables from l. D delineates the domain of the bound variables. Existential quantification, also called continued disjunction, is denoted by

$$\underline{E}(l : D : E) \quad .$$

Continued summation is denoted by

$$\underline{S}(l : D : E)$$

where E is an arithmetic expression. In the case of set formation we write

$$\{l : D : E\}$$

for the set of all values of E obtained by substituting for all variables in l values such that D holds. The empty set is denoted by $\{\}$. In each case, domain D may be omitted if D is obvious from the context.

A property of the form $E \leq G$ will often be proved in two or more steps by the introduction of one or more intermediate expressions. So may the proof of $E \leq G$ take the form of establishing both $E = F$ and $F \leq G$, for some judiciously chosen F. It is recorded as follows.

$$\begin{array}{ll} & E \\ = & \{ \text{hint why } E = F \} \\ & F \\ \leq & \{ \text{hint why } F \leq G \} \\ & G \end{array}$$

Remember that, according to [9], a hint is a "slight indication, covert or indirect suggestion" or a "small piece of practical information", and no more than that! The frequently occurring hint

{ calculus }

indicates that an appeal to everyday mathematics, like predicate calculus or arithmetic, is meant.

The above notations have been adapted from [5].

1. Trace theory

1.0. Traces and trace structures

An alphabet is a finite set of symbols. Symbols will be denoted by identifiers. For each alphabet B, B^* denotes, as usual, the set of all finite-length sequences of elements of B, including the empty sequence which is denoted by ε. Finite-length sequences of symbols are called traces. A trace structure T is a pair $< \underline{t}T, \underline{a}T >$, in which $\underline{a}T$ is an alphabet and $\underline{t}T$ is a set of traces satisfying $\underline{t}T \subseteq (\underline{a}T)^*$. $\underline{t}T$ is called the trace set of T, and $\underline{a}T$ is called the alphabet of T. The elements of $\underline{t}T$ are called traces of T, and the elements of $\underline{a}T$ are called symbols of T. We shall use $\underline{t}$ and $\underline{a}$ as operators on trace structures.

Example 1.0

A binary semaphore [3], initialized at 0, is a component specified by the trace structure SEM_1. The alphabet of SEM_1 is the set $\{p, v\}$ and the trace set is, using regular expressions [14], the set generated by the expression $(v\,p)^* + (v\,p)^*v$. The latter is the set of all finite-length alternations of v and p that do not start with a p.
(End of example)

Sometimes we appeal to a mechanistic appreciation of formally defined notions. A trace structure may then be viewed as the specification of a mechanism. We interpret each symbol from the trace structure's alphabet as the type of a communication action possible between the mechanism and its environment. Each trace may be interpreted as a sequence of communication actions that may take place between the mechanism and its environment. The trace set describes all patterns of communication actions possible. The operation of

the mechanism corresponds to the selection of symbols. The selection is such that, at any moment, the trace of symbols selected thus far is an element of the trace set. Initially the trace selected thus far is the empty trace. Which symbol is selected depends, in general, on the "behaviour" of the environment. Each selection of a symbol corresponds to an act of communication. The order of the symbols in the trace is the order in time of the corresponding acts of communication.

We would like to stress that we do not intend to model the physical properties of existing mechanisms. If one wants to relate trace structures and physical realizations, a mechanistic appreciation of trace structures must be agreed upon. A possible appreciation is given above. A combination of a physical realization and a mechanistic appreciation is called an implementation. If the realization does not match with the mechanistic appreciation, we have an erroneous implementation. In chapter five we discuss an implementation method using a mechanistic appreciation that differs slightly from the one given above.

Note

Unless noted otherwise, small and capital letters near the beginning of the Latin alphabet denote symbols and alphabets respectively; small and capital letters near the end of the Latin alphabet denote traces and trace structures respectively.
(End of note)

The projection of a trace t on an alphabet B, denoted by $t \restriction B$, is defined as follows.

$\varepsilon \restriction B = \varepsilon$;
$(t\ b) \restriction B = (t \restriction B)\ b$ for all t, b : $b \in B$;
$(t\ b) \restriction B = t \restriction B$ for all t, b : $b \notin B$.

(Concatenation is denoted by juxtaposition.)

Property 1.1

For all traces t and u, and alphabets B we have

$(t\,u)\restriction B = (t\restriction B)(u\restriction B)$.

(End of property)

Property 1.2

For all traces t and alphabets A and B we have

$t\restriction A\restriction B = t\restriction(A\cap B) = t\restriction B\restriction A$.

(End of property)

We shall often apply property 1.2 in the form

$A\supseteq B \;\Rightarrow\; t\restriction A\restriction B = t\restriction B$.

Property 1.3

For each trace t and alphabet A we have

$t\in A^* \;\equiv\; t = t\restriction A$.

(End of property)

Property 1.4

For all s, t, A, and B, such that

$s\in A^* \wedge t\in B^*$,

we have

$s\restriction B = t\restriction A \;\equiv\; \underline{E}(u:\ u\in(A\cup B)^*:\ u\restriction A = s \wedge u\restriction B = t)$.

Proof

$s\restriction B = t\restriction A$

$\Rightarrow$ { see below }

$\underline{E}(u:\ u\in(A\cup B)^*:\ u\restriction A = s \wedge u\restriction B = t)$

$\Rightarrow$ { calculus }

$\underline{E}(u:\ u\in(A\cup B)^*:\ u\restriction A\restriction B = s\restriction B \wedge u\restriction B\restriction A = t\restriction A)$

$\equiv$ { property 1.2 }

$\underline{E}(u:\ u\in(A\cup B)^*:\ s\restriction B = u\restriction(A\cap B) = t\restriction A)$

$\Rightarrow$ { calculus }

$s\restriction B = t\restriction A$

We prove

$s \restriction B = t \restriction A \;\Rightarrow\; \underline{E}(u:\ u \in (A \cup B)^*:\ u \restriction A = s \;\wedge\; u \restriction B = t)$

by induction on the length of s.

$\underline{s = \varepsilon}$

$\varepsilon \restriction B = t \restriction A$

$\equiv$ { def. of $\restriction$, property 1.3 }

$t \restriction A = \varepsilon \;\wedge\; t \restriction B = t$

$\Rightarrow$ { $t \in B^*$ hence $t \in (A \cup B)^*$ }

$\underline{E}(u:\ u \in (A \cup B)^*:\ u \restriction A = \varepsilon \;\wedge\; u \restriction B = t)$

$\underline{s = v\,b \;\wedge\; b \in A \setminus B}$

$(v\,b) \restriction B = t \restriction A$

$\equiv$ { $b \in A \setminus B$ }

$v \restriction B = t \restriction A$

$\Rightarrow$ { ind. hyp. }

$\underline{E}(u:\ u \in (A \cup B)^*:\ u \restriction A = v \;\wedge\; u \restriction B = t)$

$\equiv$ { $b \in A \setminus B$ }

$\underline{E}(u:\ u \in (A \cup B)^*:\ (u\,b) \restriction A = v\,b \;\wedge\; (u\,b) \restriction B = t)$

$\Rightarrow$ { renaming of the dummy }

$\underline{E}(u:\ u \in (A \cup B)^*:\ u \restriction A = v\,b \;\wedge\; u \restriction B = t)$

$\underline{s = v\,b \;\wedge\; b \in A \cap B}$

$(v\,b) \restriction B = t \restriction A$

$\equiv$ { $b \in A \cap B$ }

$(v \restriction B)\,b = t \restriction A$

$\Rightarrow$ { calculus }

$\underline{E}(w, x:\ t = w\,b\,x \;\wedge\; x \restriction A = \varepsilon:\ v \restriction B = w \restriction A)$

$\Rightarrow$ { ind. hyp. }

$\underline{E}(w, x:\ t = w\,b\,x \;\wedge\; x \restriction A = \varepsilon:\ \underline{E}(u:\ u \in (A \cup B)^*:\ u \restriction A = v \;\wedge\; u \restriction B = w))$

$\equiv$ { calculus }

$\underline{E}(u, w, x:\ u \in (A \cup B)^* \wedge\ t = w\,b\,x \;\wedge\; x \restriction A = \varepsilon:\ u \restriction A = v \;\wedge\; u \restriction B = w)$

$\equiv$ { $(b\,x) \restriction A = b$, $(b\,x) \restriction B = b\,x$ }

$\underline{E}(u, w, x:\ u \in (A \cup B)^* \wedge\ t = w\,b\,x \;\wedge\; x \restriction A = \varepsilon$
$\qquad :\ (u\,b\,x) \restriction A = v\,b \;\wedge\; (u\,b\,x) \restriction B = w\,b\,x$
$\qquad)$

$\equiv$ { calculus }

$\underline{E}(u, x:\ u \in (A \cup B)^* \wedge\ x \restriction A = \varepsilon:\ (u\,b\,x) \restriction A = v\,b \;\wedge\; (u\,b\,x) \restriction B = t)$

$\Rightarrow$ { renaming of the dummy }

$\underline{E}(u : u \in (A \cup B)^* : u \restriction A = v\, b \;\wedge\; u \restriction B = t)$

(End of property and proof)

The next property may be viewed as a generalization of the previous one. This can be seen by letting A coincide with either B or C. Since it is not an obvious property to look for, it derives its right of existence only from its being applied in the proof of property 1.16.

Property 1.5

For all t, u, x, A, B, and C, such that

$t \in B^* \;\wedge\; u \in C^* \;\wedge\; x \in (A \cap (B \cup C))^* \;\wedge\; B \cap C \subseteq A$,

we have

$x \restriction B = t \restriction A \;\wedge\; x \restriction C = u \restriction A$

$\equiv$

$\underline{E}(y : y \in (B \cup C)^* : y \restriction A = x \;\wedge\; y \restriction B = t \;\wedge\; y \restriction C = u)$.

Proof

Since the property is trivial in the case $B \cup C \subseteq A$ one might try to give a proof by mathematical induction on shrinking A. This turned out to be rather cumbersome and we present a different proof. Since it is a lengthy proof we factor out some intermediate results, viz.

(0) $t \restriction A = t \restriction (A \cap (B \cup C))$,

(1) $u \restriction A = u \restriction ((A \cap C) \cup B)$,

for all y : $y \in (B \cup C)^*$,

(2) $y \restriction ((A \cap C) \cup B) \restriction A = y \restriction A$,

(3) $y \restriction ((A \cap C) \cup B) \restriction B = y \restriction B$,

for all z : $z \in ((A \cap C) \cup B)^*$,

(4) $z \restriction (A \cap (B \cup C)) = z \restriction A$, and

(5) $z \restriction A \restriction C = z \restriction C$.

We shall first prove these six properties.

(0): $t \restriction A$

$=$ { property 1.3 }

$t \restriction B \restriction A$

$=$ { property 1.2 }

$t \restriction (B \cap A)$

$=$ { $B = B \cap (B \cup C)$ }

$t \restriction (B \cap A \cap (B \cup C))$

$=$ { property 1.2 }

$t \restriction B \restriction (A \cap (B \cup C))$

$=$ { property 1.3 }

$t \restriction (A \cap (B \cup C))$

(1): $u \restriction A$

$=$ { properties 1.3, 1.2 }

$u \restriction (C \cap A)$

$=$ { $B \cap C \subseteq A$ hence $C \cap A = C \cap ((A \cap C) \cup B)$ }

$u \restriction (C \cap ((A \cap C) \cup B))$

$=$ { properties 1.2, 1.3 }

$u \restriction ((A \cap C) \cup B)$

(2): $y \restriction ((A \cap C) \cup B) \restriction A$

$=$ { property 1.2 }

$y \restriction (((A \cap C) \cup B) \cap A)$

$=$ { $((A \cap C) \cup B) \cap A = (B \cup C) \cap A$ }

$y \restriction ((B \cup C) \cap A)$

$=$ { properties 1.2, 1.3 }

$y \restriction A$

(3): $y \restriction ((A \cap C) \cup B) \restriction B$

$=$ { property 1.2 }

$y \restriction (((A \cap C) \cup B) \cap B)$

$=$ { $((A \cap C) \cup B) \cap B = B$ }

$y \restriction B$

(4): $z \upharpoonright (A \cap (B \cup C))$

$= \{ A \cap (B \cup C) = ((A \cap C) \cup B) \cap A \}$

$z \upharpoonright (((A \cap C) \cup B) \cap A)$

$= \{$ properties 1.2, 1.3 $\}$

$z \upharpoonright A$

(5): $z \upharpoonright A \upharpoonright C$

$= \{$ properties 1.3, 1.2 $\}$

$z \upharpoonright (((A \cap C) \cup B) \cap A \cap C)$

$= \{ B \cap C \subseteq A$ hence $((A \cap C) \cup B) \cap A \cap C = ((A \cap C) \cup B) \cap C \}$

$z \upharpoonright (((A \cap C) \cup B) \cap C)$

$= \{$ properties 1.2, 1.3 $\}$

$z \upharpoonright C$

In the proof of property 1.5 we omit the domains of the bound variables, viz. $y \in (B \cup C)^*$ and $z \in ((A \cap C) \cup B)^*$.

$x \upharpoonright B = t \upharpoonright A \ \wedge \ x \upharpoonright C = u \upharpoonright A$

$\equiv \{ (0) \}$

$x \upharpoonright B = t \upharpoonright (A \cap (B \cup C)) \ \wedge \ x \upharpoonright C = u \upharpoonright A$

$\equiv \{$ property 1.4, $(A \cap (B \cup C)) \cup B = (A \cap C) \cup B \}$

$\underline{E}(z :: z \upharpoonright (A \cap (B \cup C)) = x \ \wedge \ z \upharpoonright B = t) \ \wedge \ x \upharpoonright C = u \upharpoonright A$

$\equiv \{ (4) \}$

$\underline{E}(z :: z \upharpoonright A = x \ \wedge \ z \upharpoonright B = t) \ \wedge \ x \upharpoonright C = u \upharpoonright A$

$\equiv \{$ calculus $\}$

$\underline{E}(z :: z \upharpoonright A = x \ \wedge \ z \upharpoonright B = t \ \wedge \ x \upharpoonright C = u \upharpoonright A)$

$\equiv \{$ calculus $\}$

$\underline{E}(z :: z \upharpoonright A = x \ \wedge \ z \upharpoonright B = t \ \wedge \ z \upharpoonright A \upharpoonright C = u \upharpoonright A)$

$\equiv \{ (5) \}$

$\underline{E}(z :: z \upharpoonright A = x \ \wedge \ z \upharpoonright B = t \ \wedge \ z \upharpoonright C = u \upharpoonright A)$

$\equiv \{ (1) \}$

$\underline{E}(z :: z \upharpoonright A = x \ \wedge \ z \upharpoonright B = t \ \wedge \ z \upharpoonright C = u \upharpoonright ((A \cap C) \cup B))$

$\equiv \{$ property 1.4, $(A \cap C) \cup B \cup C = B \cup C \}$

$\underline{E}(z :: z \upharpoonright A = x \ \wedge \ z \upharpoonright B = t \ \wedge \ \underline{E}(y :: y \upharpoonright ((A \cap C) \cup B) = z \ \wedge \ y \upharpoonright C = u))$

$\equiv \{$calculus$\}$

$$\underline{E}(y, z :: z \restriction A = x \;\wedge\; z \restriction B = t \;\wedge\; y \restriction ((A \cap C) \cup B) = z \;\wedge\; y \restriction C = u)$$
$$\equiv \quad \{ \text{ calculus } \}$$
$$\underline{E}(y :: y \restriction ((A \cap C) \cup B) \restriction A = x \;\wedge\; y \restriction ((A \cap C) \cup B) \restriction B = t \;\wedge\; y \restriction C = u)$$
$$\equiv \quad \{ (2), (3) \}$$
$$\underline{E}(y :: y \restriction A = x \;\wedge\; y \restriction B = t \;\wedge\; y \restriction C = u)$$

(End of property and proof)

Next we define a partial ordering on trace structures. It is denoted by $\subseteq$ and it is, for trace structures T and U, defined by

$$T \subseteq U \;\equiv\; \underline{t}T \subseteq \underline{t}U \;\wedge\; \underline{a}T = \underline{a}U \;.$$

In later sections it will become clear why we have chosen $\underline{a}T = \underline{a}U$ instead of the obvious alternative $\underline{a}T \subseteq \underline{a}U$.

Finally, we extend the definition of projection to operate on trace structures as well. It is defined by

$$T \restriction B = \langle \{ x : x \in \underline{t}T : x \restriction B \}, \underline{a}T \cap B \rangle \;.$$

Property 1.6

Projection of trace structures is idempotent and monotonic.

Proof of monotonicity

For all T, U, and B we have

$$T \subseteq U$$
$$\equiv \quad \{ \text{def. of } \subseteq \}$$
$$\underline{t}T \subseteq \underline{t}U \;\wedge\; \underline{a}T = \underline{a}U$$
$$\equiv \quad \{ \text{calculus} \}$$
$$\{x : x \in \underline{t}T : x\} \subseteq \{x : x \in \underline{t}U : x\} \;\wedge\; \underline{a}T = \underline{a}U$$
$$\Rightarrow \quad \{ \text{calculus} \}$$
$$\{x : x \in \underline{t}T : x \restriction B\} \subseteq \{x : x \in \underline{t}U : x \restriction B\} \;\wedge\; \underline{a}T \cap B = \underline{a}U \cap B$$
$$\equiv \quad \{ \text{def. of } \restriction \}$$
$$\underline{t}(T \restriction B) \subseteq \underline{t}(U \restriction B) \;\wedge\; \underline{a}(T \restriction B) = \underline{a}(U \restriction B)$$
$$\equiv \quad \{ \text{def. of } \subseteq \}$$
$$T \restriction B \subseteq U \restriction B \;.$$

(End of property and proof)

1.1. Prefix closure

In our mechanistic appreciation a trace structure describes, for some mechanism, all sequences of communication actions the mechanism may be involved in. The order of the symbols in a trace is the order of the communication actions in time. In the classic theory of finite state machines, a trace may be fed into such a machine, thereby leading the machine through a sequence of states. States are divided in two classes : final and non-final states. In general, one is not interested in the actual state of the machine that a trace leads to, but merely in the class to which that state belongs. Hence, some two-valued indicator is an essential part of the mechanism and its specification. Since we want to give the entire specification of a mechanism in terms of a trace structure, we abolish the indicator and the division of states in classes. The trace set consists of those traces that, formerly, led to a final state. A trace is fed into the machine one symbol after the other and, hence, each initial segment of a trace is an equally acceptable trace. Our mechanistic appreciation thus requires that each trace structure used to specify a mechanism is such that all initial segments of a trace are elements of the trace set too. An initial segment of a trace is called a prefix, i.e. we call s a prefix of t if $\underline{E}(u :: t = su)$. For trace structure T, the trace structure that contains all traces of T and prefixes thereof as well is called the prefix closure of T, and is denoted by $PREF(T)$. We have

$$PREF(T) = \langle \{s, u : su \in \underline{t}T : s\}, \underline{a}T \rangle .$$

Trace structure T is called prefix-closed if $T = PREF(T)$.

Property 1.7

For all trace structures T we have $T \subseteq PREF(T)$.

(End of property)

Property 1.8

Prefix closure is idempotent and monotonic.

(End of property)

1.2. Weaving

We shall now define our first composition function, called weaving. Composition functions express what we have informally referred to as "any suitably synchronized interleaving". In the case of two mechanisms that do not communicate, i.e. in the case of two trace structures with disjoint alphabets, weaving amounts to interleaving or shuffle as it is called in [10]. Otherwise, we stipulate that each communication in the intersection of the two alphabets requires simultaneous participation of both mechanisms. This leads to the following definition. The weave of two trace structures T and U, denoted by $T \underline{w} U$, is the trace structure

$$\langle \{x : x \in (\underline{\alpha}T \cup \underline{\alpha}U)^* \wedge x \lceil \underline{\alpha}T \in \underline{t}T \wedge x \lceil \underline{\alpha}U \in \underline{t}U : x\} , \underline{\alpha}T \cup \underline{\alpha}U \rangle .$$

Property 1.9

Weaving is symmetric, idempotent, and associative.

Proof of associativity

For all T, U, and V we have

$$\begin{aligned}
&(T \underline{w} U) \underline{w} V \\
&= \{ \text{def. of } \underline{w} \} \\
&\langle \{x : x \in (\underline{\alpha}T \cup \underline{\alpha}U)^* \wedge x \lceil \underline{\alpha}T \in \underline{t}T \wedge x \lceil \underline{\alpha}U \in \underline{t}U : x\} , \underline{\alpha}T \cup \underline{\alpha}U \rangle \ \underline{w} \ V \\
&= \{ \text{def. of } \underline{w} \} \\
&\langle \{y : y \in ((\underline{\alpha}T \cup \underline{\alpha}U) \cup \underline{\alpha}V)^* \\
&\quad \wedge y \lceil (\underline{\alpha}T \cup \underline{\alpha}U) \lceil \underline{\alpha}T \in \underline{t}T \wedge y \lceil (\underline{\alpha}T \cup \underline{\alpha}U) \lceil \underline{\alpha}U \in \underline{t}U \wedge y \lceil \underline{\alpha}V \in \underline{t}V \\
&\quad : y \} , (\underline{\alpha}T \cup \underline{\alpha}U) \cup \underline{\alpha}V \rangle
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ calculus, property 1.2} \} \\
&\quad \langle \{ y : y \in (\underline{a}T \cup \underline{a}U \cup \underline{a}V)^* \wedge y \restriction \underline{a}T \in \underline{t}T \wedge y \restriction \underline{a}U \in \underline{t}U \wedge y \restriction \underline{a}V \in \underline{t}V \\
&\qquad : y \\
&\qquad \} \\
&\quad , \underline{a}T \cup \underline{a}U \cup \underline{a}V \\
&\quad \rangle \ .
\end{aligned}
$$

The latter formula is symmetric in T, U, and V, which, due to the symmetry of weaving, proves the associativity.
(End of property and proof)

Property 1.10

For all T we have $T \ \underline{w} \ \langle \{\varepsilon\}, \{\} \rangle = T$.
(End of property)

Example 1.11

$$
\begin{aligned}
&\langle \{ab, de\}, \{a, b, d, e\} \rangle \ \underline{w} \ \langle \{bc, ef\}, \{b, c, e, f\} \rangle \\
&= \langle \{abc, def\}, \{a, b, c, d, e, f\} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle \{ab, de\}, \{a, b, d, e\} \rangle \ \underline{w} \ \langle \{cb, fe\}, \{b, c, e, f\} \rangle \\
&= \langle \{acb, cab, dfe, fde\}, \{a, b, c, d, e, f\} \rangle
\end{aligned}
$$

(End of example)

Note

One might be tempted to introduce the convention that the alphabet of a trace structure contains exactly those symbols that occur in the trace set of the trace structure. Such a convention would, however, destroy the associativity of the weaving operation, as the following example demonstrates.

$$
\begin{aligned}
&(\{ba, \varepsilon\} \ \underline{w} \ \{ab, \varepsilon\}) \ \underline{w} \ \{ab, \varepsilon\} \\
&= \{\varepsilon\} \ \underline{w} \ \{ab, \varepsilon\} \\
&= \{ab, \varepsilon\} \\
&\neq \{\varepsilon\} \\
&= \{ba, \varepsilon\} \ \underline{w} \ \{ab, \varepsilon\} \\
&= \{ba, \varepsilon\} \ \underline{w} \ (\{ab, \varepsilon\} \ \underline{w} \ \{ab, \varepsilon\})
\end{aligned}
$$

(End of note)

Property 1.12

For all T and U, such that $\underline{a}T \subseteq \underline{a}U$, we have

$T \underline{w} U = \langle \{x : x \in \underline{t}U \wedge x \upharpoonright \underline{a}T \in \underline{t}T : x\}, \underline{a}U \rangle$.

Proof

$T \underline{w} U$

$= \{$ def. of $\underline{w}\}$

$\langle \{x : x \in (\underline{a}T \cup \underline{a}U)^* \wedge x \upharpoonright \underline{a}T \in \underline{t}T \wedge x \upharpoonright \underline{a}U \in \underline{t}U : x\}, \underline{a}T \cup \underline{a}U \rangle$

$= \{\underline{a}T \subseteq \underline{a}U\}$

$\langle \{x : x \in (\underline{a}U)^* \wedge x \upharpoonright \underline{a}T \in \underline{t}T \wedge x \upharpoonright \underline{a}U \in \underline{t}U : x\}, \underline{a}U \rangle$

$= \{ x \in (\underline{a}U)^* \wedge x \upharpoonright \underline{a}U \in \underline{t}U \equiv x \in \underline{t}U \}$

$\langle \{x : x \in \underline{t}U \wedge x \upharpoonright \underline{a}T \in \underline{t}T : x\}, \underline{a}U \rangle$

(End of property and proof)

Property 1.13

Weaving is monotonic.

Proof

For all T, T', and U we have

$T \subseteq T'$

$\equiv \{$ def. of $\subseteq\}$

$\underline{t}T \subseteq \underline{t}T' \wedge \underline{a}T = \underline{a}T'$

$\equiv \{$ calculus $\}$

$\underline{A}(x : x \in \underline{t}T : x \in \underline{t}T') \wedge \underline{a}T = \underline{a}T'$

$\equiv \{$ calculus $\}$

$\underline{A}(y : y \in (\underline{a}T \cup \underline{a}U)^* \wedge y \upharpoonright \underline{a}T \in \underline{t}T : y \upharpoonright \underline{a}T \in \underline{t}T')$
$\wedge \underline{a}T = \underline{a}T'$

$\Rightarrow \{$ calculus $\}$

$\underline{A}(y : y \in (\underline{a}T \cup \underline{a}U)^* \wedge y \upharpoonright \underline{a}U \in \underline{t}U \wedge y \upharpoonright \underline{a}T \in \underline{t}T : y \upharpoonright \underline{a}T \in \underline{t}T')$
$\wedge \underline{a}T = \underline{a}T'$

$\equiv \{$ calculus $\}$

$\underline{A}(y : y \in (\underline{a}T \cup \underline{a}U)^* \wedge y \upharpoonright \underline{a}U \in \underline{t}U \wedge y \upharpoonright \underline{a}T \in \underline{t}T$
$: y \in (\underline{a}T' \cup \underline{a}U)^* \wedge y \upharpoonright \underline{a}U \in \underline{t}U \wedge y \upharpoonright \underline{a}T' \in \underline{t}T'$
$) \wedge \underline{a}T = \underline{a}T'$

$\equiv \{$ def. of $\underline{w}\}$

$\underline{A}(y : y \in \underline{t}(T \underline{w} U) : y \in \underline{t}(T' \underline{w} U)) \wedge \underline{a}T = \underline{a}T'$

$\equiv$ { calculus }

$\underline{t}(T \underline{w} U) \subseteq \underline{t}(T' \underline{w} U) \;\wedge\; \underline{a}T = \underline{a}T'$

$\Rightarrow$ { def. of $\underline{w}$ }

$\underline{t}(T \underline{w} U) \subseteq \underline{t}(T' \underline{w} U) \;\wedge\; \underline{a}(T \underline{w} U) = \underline{a}(T' \underline{w} U)$

$\equiv$ { def. of $\subseteq$ }

$T \underline{w} U \subseteq T' \underline{w} U$.

(End of property and proof)

Example 1.14

On page 11 we have defined a partial ordering on trace structures by

$T \sqsubseteq U \;\equiv\; \underline{t}T \subseteq \underline{t}U \;\wedge\; \underline{a}T = \underline{a}U$.

When removing the last conjunct, or replacing it by $\underline{a}T \subseteq \underline{a}U$, weaving is no longer a monotonic operation, as shown by the following example.

Let T, T', and U be such that

$T = \langle \{a\}, \{a\} \rangle$,

$T' = \langle \{a, b\}, \{a, b\} \rangle$, and

$U = \langle \{ab\}, \{a, b\} \rangle$.

We then have

$T \underline{w} U = \langle \{ab\}, \{a, b\} \rangle$ and

$T' \underline{w} U = \langle \{\}, \{a, b\} \rangle$.

Hence

$\underline{t}T \subseteq \underline{t}T' \;\wedge\; \neg(\underline{t}(T \underline{w} U) \subseteq \underline{t}(T' \underline{w} U))$.

(End of example)

The next two properties describe the distribution of projection through weaving.

Property 1.15

For all T, U, and A we have

$(T \underline{w} U) \restriction A \subseteq T \restriction A \;\underline{w}\; U \restriction A$.

Proof

Observe that both the alphabet of $(T \underline{w} U) \restriction A$ and of

$T \restriction A \;\underline{w}\; U \restriction A$ equals $(\underline{a}T \cup \underline{a}U) \cap A$. Hence it suffices to prove inclusion of the trace sets. For all x, such that $x \in ((\underline{a}T \cup \underline{a}U) \cap A)^*$, we have

$$
\begin{array}{ll}
 & x \in \underline{t}((T \;\underline{w}\; U) \restriction A) \\
\equiv & \{ \text{calculus} \} \\
 & \underline{E}(y : y \in \underline{t}(T \;\underline{w}\; U) : x = y \restriction A) \\
\equiv & \{ \text{def. of } \underline{w} \} \\
 & \underline{E}(y : y \in (\underline{a}T \cup \underline{a}U)^* \wedge y \restriction \underline{a}T \in \underline{t}T \wedge y \restriction \underline{a}U \in \underline{t}U : x = y \restriction A) \\
\Rightarrow & \{ \text{calculus} \} \\
 & \underline{E}(y : y \restriction \underline{a}T \restriction A \in \underline{t}(T \restriction A) \wedge y \restriction \underline{a}U \restriction A \in \underline{t}(U \restriction A) : x = y \restriction A) \\
\equiv & \{ \text{property 1.2} \} \\
 & \underline{E}(y : y \restriction A \restriction \underline{a}T \in \underline{t}(T \restriction A) \wedge y \restriction A \restriction \underline{a}U \in \underline{t}(U \restriction A) : x = y \restriction A) \\
\equiv & \{ \text{calculus} \} \\
 & x \restriction \underline{a}T \in \underline{t}(T \restriction A) \wedge x \restriction \underline{a}U \in \underline{t}(U \restriction A) \\
\equiv & \{ x = x \restriction A \} \\
 & x \restriction (\underline{a}T \cap A) \in \underline{t}(T \restriction A) \wedge x \restriction (\underline{a}U \cap A) \in \underline{t}(U \restriction A) \\
\equiv & \{ \text{def. of } \underline{w} \} \\
 & x \in \underline{t}(T \restriction A \;\underline{w}\; U \restriction A) \; .
\end{array}
$$

(End of property and proof)

Property 1.16

For all T, U, and A, such that $\underline{a}T \cap \underline{a}U \subseteq A$, we have $(T \;\underline{w}\; U) \restriction A = T \restriction A \;\underline{w}\; U \restriction A$.

Proof

Since the two alphabets of interest are obviously equal, we only prove the equality of the trace sets. For all x, such that $x \in ((\underline{a}T \cup \underline{a}U) \cap A)^*$, we have

$$
\begin{array}{ll}
 & x \in \underline{t}(T \restriction A \;\underline{w}\; U \restriction A) \\
\equiv & \{ \text{def. of } \underline{w} \} \\
 & x \restriction (\underline{a}T \cap A) \in \underline{t}(T \restriction A) \wedge x \restriction (\underline{a}U \cap A) \in \underline{t}(U \restriction A) \\
\equiv & \{ x = x \restriction A \} \\
 & x \restriction \underline{a}T \in \underline{t}(T \restriction A) \wedge x \restriction \underline{a}U \in \underline{t}(U \restriction A) \\
\equiv & \{ \text{calculus} \} \\
 & \underline{E}(t, u : t \in \underline{t}T \wedge u \in \underline{t}U : x \restriction \underline{a}T = t \restriction A \wedge x \restriction \underline{a}U = u \restriction A)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \text{ property 1.5, } \underline{a}T \cap \underline{a}U \subseteq A \} \\
& E(t,u,y : t \in \underline{t}T \wedge u \in \underline{t}U \wedge y \in (\underline{a}T \cup \underline{a}U)^* \\
& \qquad : y \restriction A = x \wedge y \restriction \underline{a}T = t \wedge y \restriction \underline{a}U = u \\
& \qquad) \\
\equiv & \{ \text{ calculus } \} \\
& \underline{E}(y : y \in (\underline{a}T \cup \underline{a}U)^* \wedge y \restriction \underline{a}T \in \underline{t}T \wedge y \restriction \underline{a}U \in \underline{t}U \; : \; y \restriction A = x) \\
\equiv & \{ \text{ def. of } \underline{w} \} \\
& \underline{E}(y : y \in \underline{t}(T \,\underline{w}\, U) \; : \; y \restriction A = x) \\
\equiv & \{ \text{ calculus } \} \\
& x \in \underline{t}((T \,\underline{w}\, U) \restriction A) \; .
\end{array}
$$

(End of property and proof)

We conclude this section with three properties related to weaving and prefix closure.

Property 1.17

The weave of prefix-closed trace structures is prefix-closed.

Proof

For all prefix-closed trace structures T and U, and all traces x and y, such that $xy \in (\underline{a}T \cup \underline{a}U)^*$, we have

$$
\begin{array}{ll}
& xy \in \underline{t}(T \,\underline{w}\, U) \\
\equiv & \{ \text{ def. of } \underline{w} \} \\
& (xy) \restriction \underline{a}T \in \underline{t}T \wedge (xy) \restriction \underline{a}U \in \underline{t}U \\
\equiv & \{ \text{ property 1.1 } \} \\
& (x \restriction \underline{a}T)(y \restriction \underline{a}T) \in \underline{t}T \wedge (x \restriction \underline{a}U)(y \restriction \underline{a}U) \in \underline{t}U \\
\Rightarrow & \{ T \text{ and } U \text{ are prefix-closed } \} \\
& x \restriction \underline{a}T \in \underline{t}T \wedge x \restriction \underline{a}U \in \underline{t}U \\
\equiv & \{ \text{ def. of } \underline{w} \} \\
& x \in \underline{t}(T \,\underline{w}\, U) \; ,
\end{array}
$$

which proves that $T \,\underline{w}\, U$ is prefix-closed.

(End of property and proof)

Property 1.18

For all T and U we have

$$PREF(T \underline{w} U) \subseteq PREF(T) \underline{w} PREF(U) .$$

Proof

true

$\equiv$ { property 1.7 }

$T \subseteq PREF(T) \wedge U \subseteq PREF(U)$

$\Rightarrow$ { weaving is monotonic }

$T \underline{w} U \subseteq PREF(T) \underline{w} U \wedge PREF(T) \underline{w} U \subseteq PREF(T) \underline{w} PREF(U)$

$\Rightarrow$ { transitivity of inclusion }

$T \underline{w} U \subseteq PREF(T) \underline{w} PREF(U)$

$\Rightarrow$ { PREF is monotonic }

$PREF(T \underline{w} U) \subseteq PREF(PREF(T) \underline{w} PREF(U))$

$\equiv$ { property 1.17 }

$PREF(T \underline{w} U) \subseteq PREF(T) \underline{w} PREF(U)$

(End of property and proof)

Property 1.19

For all T and U, such that $\underline{a}T \cap \underline{a}U = \{\}$, we have

$$PREF(T \underline{w} U) = PREF(T) \underline{w} PREF(U) .$$

Proof

Again, we are concerned with equality of the trace sets only. For all traces x, such that $x \in (\underline{a}T \cup \underline{a}U)^*$, we have

$x \in \underline{t}(PREF(T \underline{w} U))$

$\Rightarrow$ { property 1.18 }

$x \in \underline{t}(PREF(T) \underline{w} PREF(U))$

$\equiv$ { def. of $\underline{w}$ }

$x \restriction \underline{a}T \in \underline{t}(PREF(T)) \wedge x \restriction \underline{a}U \in \underline{t}(PREF(U))$

$\equiv$ { def. of PREF }

$\underline{E}(y, z : y \in (\underline{a}T)^* \wedge z \in (\underline{a}U)^* : (x \restriction \underline{a}T)y \in \underline{t}T \wedge (x \restriction \underline{a}U)z \in \underline{t}U)$

$\equiv$ { $\underline{a}T \cap \underline{a}U = \{\}$ }

$\underline{E}(y, z : y \in (\underline{a}T)^* \wedge z \in (\underline{a}U)^* : (xyz) \restriction \underline{a}T \in \underline{t}T \wedge (xyz) \restriction \underline{a}U \in \underline{t}U)$

$\equiv$ { def. of $\underline{w}$ }

$$
\begin{array}{ll}
 & \underline{\exists}(y,z:\ y \in (\underline{a}T)^* \wedge z \in (\underline{a}\cup)^* :\ x\,y\,z \in \underline{t}(T \underline{w} \cup)) \\
\Rightarrow & \{\ \textit{calculus}\ \} \\
 & \underline{\exists}(w:\ w \in (\underline{a}T \cup \underline{a}\cup)^* :\ x\,w \in \underline{t}(T \underline{w} \cup)) \\
\equiv & \{\ \textit{def. of PREF}\ \} \\
 & x \in \underline{t}(\mathrm{PREF}(T \underline{w} \cup))\ .
\end{array}
$$

(End of property and proof)

This concludes our investigation of weaving and its relations to projection and prefix closure.

1.3. Junctivity of weaving

In this section we define the operations intersection and union on trace structures and we investigate how weaving distributes through them. First, we give the definitions. For all trace structures T and U we define

$$T \sqcap U = \langle \underline{t}T \cap \underline{t}U,\ \underline{a}T \cap \underline{a}U \rangle \quad \text{and}$$
$$T \sqcup U = \langle \underline{t}T \cup \underline{t}U,\ \underline{a}T \cup \underline{a}U \rangle \ .$$

Property 1.20

For all T and U, such that $\underline{a}T = \underline{a}U$, we have $T \underline{w} U = T \sqcap U$.

Proof

$$\begin{aligned}
& T \underline{w} U \\
= & \quad \{ \text{property 1.12} \} \\
& \langle \{x : x \in \underline{t}U \wedge x \restriction \underline{a}T \in \underline{t}T : x\},\ \underline{a}U \rangle \\
= & \quad \{ \underline{a}T = \underline{a}U \ ,\ \text{property 1.3} \} \\
& \langle \{x : x \in \underline{t}U \wedge x \in \underline{t}T : x\},\ \underline{a}T \cap \underline{a}U \rangle \\
= & \quad \{ \text{calculus} \} \\
& \langle \underline{t}T \cap \underline{t}U,\ \underline{a}T \cap \underline{a}U \rangle \\
= & \quad \{ \text{def. of } \sqcap \} \\
& T \sqcap U
\end{aligned}$$

(End of property and proof)

Property 1.21

Weaving distributes through union and intersection of trace structures with equal alphabets.

Proof

For all T, T', and U, such that $\underline{a}T = \underline{a}T'$, we have

$$\begin{aligned}
& U \underline{w} (T \sqcup T') \\
= & \quad \{ \text{def. of } \sqcup,\ \underline{a}T = \underline{a}T' \} \\
& U \underline{w} \langle \underline{t}T \cup \underline{t}T',\ \underline{a}T \rangle \\
= & \quad \{ \text{def. of } \underline{w} \} \\
& \langle \{x : x \in (\underline{a}T \cup \underline{a}U)^* \wedge x \restriction \underline{a}U \in \underline{t}U \wedge x \restriction \underline{a}T \in \underline{t}T \cup \underline{t}T' : x\},\ \underline{a}T \cup \underline{a}U \rangle \\
= & \quad \{ \text{calculus} \}
\end{aligned}$$

$$
\begin{aligned}
&\langle \{x:\ x \in (\underline{a}T \cup \underline{a}U)^* \wedge x \restriction \underline{a}U \in \underline{t}U \wedge x \restriction \underline{a}T \in \underline{t}T\ :\ x\} \\
&\cup \{x:\ x \in (\underline{a}T \cup \underline{a}U)^* \wedge x \restriction \underline{a}U \in \underline{t}U \wedge x \restriction \underline{a}T \in \underline{t}T'\ :\ x\} \\
&,\ \underline{a}T \cup \underline{a}U \\
&\rangle \\
=\ &\{\ \text{def. of } \cup,\ \underline{a}T = \underline{a}T'\ \} \\
&\langle \{x:\ x \in (\underline{a}T \cup \underline{a}U)^* \wedge x \restriction \underline{a}U \in \underline{t}U \wedge x \restriction \underline{a}T \in \underline{t}T\ :\ x\},\ \underline{a}T \cup \underline{a}U \rangle \\
&\cup \langle \{x:\ x \in (\underline{a}T' \cup \underline{a}U)^* \wedge x \restriction \underline{a}U \in \underline{t}U \wedge x \restriction \underline{a}T' \in \underline{t}T'\ :\ x\},\ \underline{a}T' \cup \underline{a}U \rangle \\
=\ &\{\ \text{def. of } \underline{w}\ \} \\
&(U \mathbin{\underline{w}} T) \cup (U \mathbin{\underline{w}} T')\ ,
\end{aligned}
$$

which proves that weaving distributes through union of trace structures with equal alphabets. Distribution through intersection may be proven similarly. A proof that is specific to intersection, however, is

$$
\begin{aligned}
&U \mathbin{\underline{w}} (T \cap T') \\
=\ &\{\ \text{property } 1.20\ \} \\
&U \mathbin{\underline{w}} (T \mathbin{\underline{w}} T') \\
=\ &\{\ \text{weaving is idempotent, symmetric, and associative}\ \} \\
&(U \mathbin{\underline{w}} T) \mathbin{\underline{w}} (U \mathbin{\underline{w}} T') \\
=\ &\{\ \text{property } 1.20\ \} \\
&(U \mathbin{\underline{w}} T) \cap (U \mathbin{\underline{w}} T')\ .
\end{aligned}
$$

(End of property and proof)

Example 1.22

We have shown that weaving distributes through union and intersection of trace structures with equal alphabets. The following example shows that, when removing the latter restriction, we lose the distribution properties.

Let trace structures U, V, W, X, Y, and Z be such that

$U = \langle \{a\}, \{a\} \rangle$,
$V = \langle \{b\}, \{b\} \rangle$,
$W = \langle \{a\ b\}, \{a, b\} \rangle$,
$X = \langle \{a, b\}, \{a, b\} \rangle$,
$Y = \langle \{\}, \{a, b\} \rangle$, and
$Z = \langle \{\}, \{\} \rangle$.

Absence of distribution through union follows from:

$$U \cup V = X \;\wedge\; X \underline{w} W = Y \;\wedge\; U \underline{w} W = W \;\wedge\; V \underline{w} W = W \;\wedge\; W \cup W = W$$

hence

$$(U \cup V) \underline{w} W = Y \neq W = (U \underline{w} W) \cup (V \underline{w} W) .$$

Absence of distribution through intersection follows from:

$$U \cap V = Z \;\wedge\; Z \underline{w} W = Y \;\wedge\; U \underline{w} W = W \wedge V \underline{w} W = W \;\wedge\; W \cap W = W$$

hence

$$(U \cap V) \underline{w} W = Y \neq W = (U \underline{w} W) \cap (V \underline{w} W) .$$

(End of example)

Union of two trace structures is a special case of continued union: the union of a bag of zero or more trace structures. From example 1.22 and property 1.21 we know that we may hope for distribution of weaving through a bag of trace structures with equal alphabets only. Let B be a bag of trace structures and let A be an alphabet, such that

$$\underline{A}(T : T \in B : \underline{a} T = A) .$$

Given A and B, continued union of the trace structures in B is denoted by

$$U(T : T \in B : T)$$

and it is for non-empty bag B defined as usual. For empty bag B it equals the trace structure

$$< \{\}, A >$$

in order to guarantee

$$\underline{a}(U(T : T \in B : T)) = A$$

for any bag B. As in [5], an operation is called universally disjunctive if it distributes through the union of any bag of trace structures with equal alphabets. Similarly, an operation is called universally conjunctive if it distributes through the intersection of such bags.

Property 1.23

Weaving is universally disjunctive and universally conjunctive.
(End of property)

1.4. Blending

Designing programs that suggest implementations with a high degree of concurrency requires a very careful design technique, for, as we know, uncontrolled concurrency results in uncontrollable complexity. Hierarchical design is an effective technique for controlling complexity. Using this technique, the design of a component amounts to the choice of subcomponents and relations between them. The relations express how the parts (the subcomponents) constitute the whole (the component). Designing the subcomponents in a similar fashion, we obtain hierarchical components. In order to bridle the complexity of the design task it is required that the specification of a component does not reflect the component's internal structure. Consequently, the specification comprises the component's net effect only, i.e. it consists of all possible communication patterns between the component and its environment. We represent such a specification by a trace structure. Relations between subcomponents will be expressed as equations between symbols in the alphabets of the trace structures of the subcomponents.

We have proposed that the weave of two trace structures describe the joint operation of the two mechanisms to which they correspond. This includes their mutual communication, embodied by the common symbols. Omitting these symbols from the weave we obtain the trace structure specifying the net effect of the compound mechanism. We introduce a second composition function on trace structures, called blending, which is weaving followed by the elimination of common symbols. The blend of two trace structures T and U is denoted by $T \underline{b} U$, and is the trace structure

$$(T \underline{w} U) \upharpoonright (\underline{a}T \div \underline{a}U)$$

where $\div$ denotes symmetric set difference, i.e. $A \div B = (A \cup B) \setminus (A \cap B)$.

(Symmetric set difference is symmetric and associative.) This composition operation differs from the one in [21] in that the latter operation replaces eliminated symbols by "silent moves".

Property 1.24

Blending is symmetric.

(End of property)

Property 1.25

For all T we have

$$\underline{t}T = \{\} \quad \vee \quad T \mathbin{\underline{b}} T = \langle \{\varepsilon\}, \{\} \rangle \ .$$

(End of property)

Property 1.26

For all T and U, such that $\underline{a}T \cap \underline{a}U = \{\}$, we have

$$T \mathbin{\underline{w}} U = T \mathbin{\underline{b}} U \ .$$

(End of property)

Example 1.27

$$\begin{aligned}
&(\langle \{a\,b\}, \{a, b\} \rangle \mathbin{\underline{b}} \langle \{a\,c\}, \{a, c\} \rangle) \mathbin{\underline{b}} \langle \{a\,c\}, \{a, c\} \rangle \\
&= \langle \{b\,c, c\,b\}, \{b, c\} \rangle \mathbin{\underline{b}} \langle \{a\,c\}, \{a, c\} \rangle \\
&= \langle \{a\,b, b\,a\}, \{a, b\} \rangle \\
&\neq \langle \{a\,b\}, \{a, b\} \rangle \\
&= \langle \{a\,b\}, \{a, b\} \rangle \mathbin{\underline{b}} \langle \{\varepsilon\}, \{\} \rangle \\
&= \langle \{a\,b\}, \{a, b\} \rangle \mathbin{\underline{b}} (\langle \{a\,c\}, \{a, c\} \rangle \mathbin{\underline{b}} \langle \{a\,c\}, \{a, c\} \rangle)
\end{aligned}$$

(End of example)

As the above example shows, blending is not associative. We have, however, the following property.

Property 1.28

For all T, U, and V, such that

$$\underline{a}T \cap \underline{a}U \cap \underline{a}V = \{\} \ ,$$

we have

$$(T \mathbin{\underline{b}} U) \mathbin{\underline{b}} V = T \mathbin{\underline{b}} (U \mathbin{\underline{b}} V) \ .$$

Proof

Observe that, since $\underline{a}T \cap \underline{a}U \cap \underline{a}V = \{\}$, we have
$\underline{a}T \div \underline{a}U = (\underline{a}T \cup \underline{a}U) \cap ((\underline{a}T \div \underline{a}U) \cup \underline{a}V)$.

$$
\begin{aligned}
& (T \mathbin{\underline{b}} U) \mathbin{\underline{b}} V \\
= & \quad \{ \text{def. of } \underline{b} \} \\
& ((T \mathbin{\underline{w}} U) \restriction (\underline{a}T \div \underline{a}U) \mathbin{\underline{w}} V) \restriction ((\underline{a}T \div \underline{a}U) \div \underline{a}V) \\
= & \quad \{ \text{observation above, property 1.3} \} \\
& ((T \mathbin{\underline{w}} U) \restriction ((\underline{a}T \div \underline{a}U) \cup \underline{a}V) \mathbin{\underline{w}} V) \restriction (\underline{a}T \div \underline{a}U \div \underline{a}V) \\
= & \quad \{ \underline{a}V \subseteq (\underline{a}T \div \underline{a}U) \cup \underline{a}V, \text{ properties 1.2, 1.3} \} \\
& ((T \mathbin{\underline{w}} U) \restriction ((\underline{a}T \div \underline{a}U) \cup \underline{a}V) \mathbin{\underline{w}} V \restriction ((\underline{a}T \div \underline{a}U) \cup \underline{a}V)) \restriction (\underline{a}T \div \underline{a}U \div \underline{a}V) \\
= & \quad \{ \text{property 1.16, } (\underline{a}T \cup \underline{a}U) \cap \underline{a}V \subseteq (\underline{a}T \div \underline{a}U) \cup \underline{a}V \} \\
& ((T \mathbin{\underline{w}} U) \mathbin{\underline{w}} V) \restriction ((\underline{a}T \div \underline{a}U) \cup \underline{a}V) \restriction (\underline{a}T \div \underline{a}U \div \underline{a}V) \\
= & \quad \{ \text{property 1.2, } (\underline{a}T \div \underline{a}U) \cup \underline{a}V \supseteq \underline{a}T \div \underline{a}U \div \underline{a}V \} \\
& (T \mathbin{\underline{w}} U \mathbin{\underline{w}} V) \restriction (\underline{a}T \div \underline{a}U \div \underline{a}V)
\end{aligned}
$$

The latter formula is symmetric in T, U, and V, which, due to the symmetry of blending, proves the property.
(End of property and proof)

Whenever employing the blending operation, we will see to it that each symbol occurs in at most two alphabets of the constituting trace structures. Under this restriction, blending is associative.

A number of properties of blending can immediately be derived from those of weaving. We list them without proof.

Property 1.29
For all T we have $T \mathbin{\underline{b}} \langle \{\varepsilon\}, \{\} \rangle = T$.

Property 1.30
Blending is monotonic.

Property 1.31
For all T, U, and A we have
$(T \mathbin{\underline{b}} U) \restriction A \subseteq T \restriction A \mathbin{\underline{b}} U \restriction A$.
If $\underline{a}T \cap \underline{a}U \subseteq A$ then
$(T \mathbin{\underline{b}} U) \restriction A = T \restriction A \mathbin{\underline{b}} U \restriction A$.

Property 1.32

The blend of prefix-closed trace structures is prefix-closed.

Property 1.33

For all T and U we have

$$PREF(T \underline{b} U) \subseteq PREF(T) \underline{b} PREF(U) .$$

If $\underline{a}T \cap \underline{a}U = \{\}$ then

$$PREF(T \underline{b} U) = PREF(T) \underline{b} PREF(U) .$$

Property 1.34

For all T, T', and U, such that $\underline{a}T = \underline{a}T'$, we have

$$U \underline{b} (T \cup T') = (U \underline{b} T) \cup (U \underline{b} T') \quad \text{and}$$

$$U \underline{b} (T \cap T') \subseteq (U \underline{b} T) \cap (U \underline{b} T') .$$

Property 1.35

Blending is universally disjunctive.

(End of list of properties)

Distribution through intersection, and, hence, universal conjunctivity, does not hold for blending as it does for weaving. The necessity of weakening equality to inclusion in property 1.34 is illustrated by the following example.

Example 1.36

Let T, T', and U be such that

$$T = \langle \{a\ b\}, \{a, b\} \rangle ,$$

$$T' = \langle \{b\ a\}, \{a, b\} \rangle , \text{ and}$$

$$U = \langle \{a\ b, b\ a\}, \{a, b\} \rangle .$$

We then have

$$\begin{aligned} & U \underline{b} (T \cap T') \\ &= U \underline{b} \langle \{\}, \{a, b\} \rangle \\ &= \langle \{\}, \{\} \rangle \\ &\subset \langle \{\varepsilon\}, \{\} \rangle \\ &= \langle \{\varepsilon\}, \{\} \rangle \cap \langle \{\varepsilon\}, \{\} \rangle \\ &= (U \underline{b} T) \cap (U \underline{b} T') \end{aligned}$$

where $\subset$ denotes proper inclusion.

(End of example)

We conclude this section with the introduction of continued blending. Let Z be a finite set of trace structures, such that each symbol occurs in at most two alphabets of trace structures in Z. The continued blend of the trace structures in Z is denoted by

$$\underline{B}(T:\ T \in Z :\ T)$$

and is defined by

$$\underline{B}(T:\ T \in \{\} :\ T) = \langle \{\varepsilon\},\ \{\} \rangle ,$$

$$\underline{B}(T:\ T \in Z \cup \{U\} :\ T) = \underline{B}(T:\ T \in Z :\ T)\ \underline{b}\ U .$$

Property 1.37

For all finite, disjoint sets of trace structures Y and Z, such that each symbol occurs in at most two alphabets of trace structures in $Y \cup Z$, we have

$$\underline{B}(T:\ T \in Y \cup Z :\ T) = \underline{B}(T:\ T \in Y :\ T)\ \underline{b}\ \underline{B}(T:\ T \in Z :\ T) .$$

(End of property)

1.5. Regular trace structures

As mentioned before, we are interested in the implementation of programs as chip. Programs specify trace structures. Since chips are finite mechanisms, an implementation as chip is feasible only if the trace structure is regular, where a trace structure is called regular if its trace set is a regular set [14].

Let T be a trace structure. Consider relation E defined on $\underline{t}(PREF(T))$ by

$$(x \; E \; y) \equiv \underline{A}(z: z \in (\underline{a}T)^*: xz \in \underline{t}T \equiv yz \in \underline{t}T) .$$

Relation E is an equivalence relation. Its equivalence classes, i.e. the elements of $\underline{t}(PREF(T))/E$, are called the states of T. The state of which x is a member is denoted by $[x]$. A well-known property is

T is regular $\equiv$ T has a finite number of states .

Example 1.38

Let T be the trace structure $\langle \{k: k \geq 0: (v\ p)^k\}, \{v, p\} \rangle$. Then $PREF(T) = SEM_1$, SEM_1 as defined in example 1.0. Relation E has two equivalence classes, viz. the elements of SEM_1 of even length and those of odd length. T has, consequently, two states: $[\varepsilon]$ and $[v]$.
(End of example)

In chapter three we shall prove that the class of regular trace structures is closed under weaving and blending. We do so by constructing finite state machines accepting the weave or blend of two trace structures, given the finite state machines accepting the latter two.

1.6. Synchronization mechanisms

In this section we discuss some properties of three classes of trace structures that may be appreciated as the specification of a synchronization mechanism [16]. In the chapters that follow they will play an important role. An interesting class of such trace structures, called SYNC, is defined as follows. For natural k and l, such that $k \geq 0 \wedge l \geq 0 \wedge k+l > 0$, and distinct symbols b and c, the trace structure $(k, l)\ \mathrm{SYNC}\ (b, c)$ equals

$$\langle \{x : x \in \{b, c\}^* \wedge \underline{A}(y, z : x = yz : -l \leq y\underline{N}b - y\underline{N}c \leq k) : x\},\ \{b, c\} \rangle$$

where $y\underline{N}b$ denotes the number of occurrences of b in y. Notice that this trace structure is prefix-closed. It contains all traces in which the lead of b's over c's is bounded by k, and the lead of c's over b's is bounded by l. The number $k+l$ is called the slack. Obviously,

$$(k, l)\ \mathrm{SYNC}\ (b, c) = (l, k)\ \mathrm{SYNC}\ (c, b) .$$

An example of a trace structure of this class is SEM_1, the trace structure from example 1.0, since

$$SEM_1 = (1, 0)\ \mathrm{SYNC}\ (v, p) .$$

Property 1.39

$(k, l)\ \mathrm{SYNC}\ (b, c)$ has $k+l+1$ states, viz. $[b^i c^j]$ for all i and j such that $(j = 0 \wedge 0 \leq i \leq k) \vee (i = 0 \wedge 0 \leq j \leq l)$.
(End of property)

An important property is the following one, called the rule of addition.

Property 1.40

For distinct symbols a, b, and c, and natural numbers k, l, m, and n, such that $k+l > 0 \wedge m+n > 0$, we have

$$(k, l)\mathrm{SYNC}(a, b)\ \underline{b}\ (m, n)\mathrm{SYNC}(b, c) = (k+m, l+n)\ \mathrm{SYNC}\ (a, c).$$

Proof

Let AC and ABC be two trace structures defined by

$ABC = (k, l)\,SYNC(a, b)\ \underline{w}\ (m, n)\,SYNC(b, c)$ and

$AC = (k+m, l+n)\,SYNC(a, c)$.

We prove the property by first showing $ABC\restriction\{a, c\} \subseteq AC$ and then $AC \subseteq ABC\restriction\{a, c\}$. Notice that their alphabets are equal.

For all t, such that $t \in \{a, b, c\}^*$, we have

$t \in \underline{t}\,ABC$

$\equiv$ { def. of ABC and $\underline{w}$ }

$t\restriction\{a, b\} \in \underline{t}((k, l)SYNC(a, b)) \wedge t\restriction\{b, c\} \in \underline{t}((m, n)SYNC(b, c))$

$\equiv$ { def. of SYNC }

$\underline{A}(x, y: t\restriction\{a, b\} = x\,y : -l \leq x\,\underline{N}\,a - x\,\underline{N}\,b \leq k)$

$\wedge\ \underline{A}(x, y: t\restriction\{b, c\} = x\,y : -n \leq x\,\underline{N}\,b - x\,\underline{N}\,c \leq m)$

$\Rightarrow$ { calculus }

$\underline{A}(x, y: t = x\,y : -l-n \leq x\,\underline{N}\,a - x\,\underline{N}\,c \leq k+m)$

$\equiv$ { def. of SYNC }

$t\restriction\{a, c\} \in \underline{t}\,AC$

from which we conclude $ABC\restriction\{a, c\} \subseteq AC$.

In order to prove $AC \subseteq ABC\restriction\{a, c\}$ we demonstrate for all t, such that $t \in \underline{t}\,AC$, the existence of a trace t' satisfying both

$t'\restriction\{a, c\} = t$

and

$\underline{A}(x, y: t' = x\,y : -l \leq x\,\underline{N}\,a - x\,\underline{N}\,b \leq k \ \wedge\ -n \leq x\,\underline{N}\,b - x\,\underline{N}\,c \leq m)$.

The latter is equivalent to $t' \in \underline{t}\,ABC$, which implies $t'\restriction\{a, c\} \in \underline{t}(ABC\restriction\{a, c\})$. Using the first equation we conclude $t \in \underline{t}(ABC\restriction\{a, c\})$, which completes the proof of our property. We supply a proof of the existence of t' by mathematical induction on the length of t .

$\underline{t = \varepsilon}$

We choose $t' = \varepsilon$ and verify both

$\varepsilon\restriction\{a, c\} = \varepsilon$

$\equiv$ { def. of $\restriction$ }

true

and

$$\underline{A}(x,y: \varepsilon = x\,y: -l \le x\,\underline{N}\,a - x\,\underline{N}\,b \le k \wedge -n \le x\,\underline{N}\,b - x\,\underline{N}\,c \le m)$$

$\equiv$ { calculus }

$$-l \le 0 \le k \wedge -n \le 0 \le m$$

$\equiv$ { k, l, m, and n are natural }

true .

$\underline{t = s\,a}$

Since AC is prefix-closed, $t \in \underline{t}AC$ implies $s \in \underline{t}AC$. By induction hypothesis there exists a trace s' satisfying

(0) $s' \restriction \{a, c\} = s$

and

(1) $\underline{A}(x,y: s' = x\,y: -l \le x\,\underline{N}\,a - x\,\underline{N}\,b \le k \wedge -n \le x\,\underline{N}\,b - x\,\underline{N}\,c \le m)$,

which implies

(2) $-l \le s'\,\underline{N}\,a - s'\,\underline{N}\,b \le k \wedge -n \le s'\,\underline{N}\,b - s'\,\underline{N}\,c \le m$.

From $s \in \underline{t}AC$ and $s\,a \in \underline{t}AC$ we conclude

$$s\,\underline{N}\,a - s\,\underline{N}\,c \le k + m - 1$$

and, hence, due to (0),

(3) $s'\,\underline{N}\,a - s'\,\underline{N}\,c \le k + m - 1$.

Given any such trace s', we choose for t' either $s'a$ or $s'ba$. As we only want to demonstrate the existence of a suitable t' we need not specify the actual choice. We verify both

$$(s'a) \restriction \{a,c\} = s\,a \wedge (s'\,b\,a) \restriction \{a,c\} = s\,a$$

$\equiv$ { def. of $\restriction$ }

$$s' \restriction \{a,c\} = s \wedge s' \restriction \{a,c\} = s$$

$\equiv$ { (0) }

true

and

$$\underline{A}(x,y: s'a = x\,y: -l \le x\,\underline{N}\,a - x\,\underline{N}\,b \le k \wedge -n \le x\,\underline{N}\,b - x\,\underline{N}\,c \le m)$$
$$\vee\ \underline{A}(x,y: s'ba = x\,y: -l \le x\,\underline{N}\,a - x\,\underline{N}\,b \le k \wedge -n \le x\,\underline{N}\,b - x\,\underline{N}\,c \le m)$$

$\equiv$ { calculus }

$$\underline{A}(x,y: s^l = x\,y : -l \le x\underline{N}a - x\underline{N}b \le k \;\wedge\; -n \le x\underline{N}b - x\underline{N}c \le m)$$
$$\wedge\ ((-l \le (s^l a)\underline{N}a - (s^l a)\underline{N}b \le k \;\wedge\; -n \le (s^l a)\underline{N}b - (s^l a)\underline{N}c \le m)$$
$$\vee(-l \le (s^l b)\underline{N}a - (s^l b)\underline{N}b \le k \;\wedge\; -n \le (s^l b)\underline{N}b - (s^l b)\underline{N}c \le m$$
$$\wedge -l \le (s^l ba)\underline{N}a - (s^l ba)\underline{N}b \le k \;\wedge\; -n \le (s^l ba)\underline{N}b - (s^l ba)\underline{N}c \le m$$
$$))$$

$\equiv$ { calculus , (1) }

$$(-l-1 \le s^l\underline{N}a - s^l\underline{N}b \le k-1 \;\wedge\; -n \le s^l\underline{N}b - s^l\underline{N}c \le m)$$
$$\vee(-l+1 \le s^l\underline{N}a - s^l\underline{N}b \le k+1 \;\wedge\; -n-1 \le s^l\underline{N}b - s^l\underline{N}c \le m-1$$
$$\wedge\ -l \le s^l\underline{N}a - s^l\underline{N}b \le k \;\wedge\; -n-1 \le s^l\underline{N}b - s^l\underline{N}c \le m-1$$
$$)$$

$\equiv$ {calculus}

$$(-l-1 \le s^l\underline{N}a - s^l\underline{N}b \le k-1 \;\wedge\; -n \le s^l\underline{N}b - s^l\underline{N}c \le m)$$
$$\vee(-l+1 \le s^l\underline{N}a - s^l\underline{N}b \le k \;\wedge\; -n-1 \le s^l\underline{N}b - s^l\underline{N}c \le m-1)$$

$\equiv$ { (2) }

$$s^l\underline{N}a - s^l\underline{N}b \le k-1 \;\vee\; (-l+1 \le s^l\underline{N}a - s^l\underline{N}b \;\wedge\; s^l\underline{N}b - s^l\underline{N}c \le m-1)$$

$\equiv$ { $k+l>0$, hence $k \ge -l+1$ }

$$s^l\underline{N}a - s^l\underline{N}b \le k-1 \;\vee\; s^l\underline{N}b - s^l\underline{N}c \le m-1$$

$\equiv$ { (2), (3) }

true .

$\underline{t = sc}$

Similarly .

(End of property and proof)

Next we define a class of trace structures, called DEL, which may be appreciated as synchronization mechanisms with unbounded slack. For distinct symbols b and c we define

$$DEL(b,c) = \bigcup(k: k \ge 1 : (k,0)\ SYNC(b,c)) .$$

Since for all k, such that $k \ge 1$,

$$(k,0)\ SYNC(b,c) \subseteq (k+1,0)\ SYNC(b,c) ,$$

DEL(b,c) may be viewed as the limit of the monotonic sequence $(k,0)\ SYNC(b,c)$, $k \ge 1$. Since DEL has an infinite number of states, it is not a regular trace structure.

Property 1.41

For distinct symbols b and c we have

$$\begin{aligned} &DEL(b, c) \\ &= \langle \{ x : x \in \{b, c\}^* \wedge \underline{A}(y, z : x = yz : y \underline{N} b \geq y \underline{N} c) : x \}, \{b, c\} \rangle . \end{aligned}$$

(End of property)

Notice the asymmetry of DEL(b, c) : it does not bound the lead of b's over c's , whereas the lead of c's over b's is bound by 0.

Property 1.42

For distinct symbols b, c, and d we have

$$\begin{aligned} &DEL(b, c) \ \underline{b} \ (1, 0) \ SYNC \ (c, d) \\ &= DEL(b, d) \\ &= (1, 0) SYNC(b, c) \ \underline{b} \ DEL(c, d) \ . \end{aligned}$$

Proof

$$\begin{aligned} &DEL(b, c) \ \underline{b} \ (1,0) \ SYNC(c, d) \\ &= \quad \{ \text{def. of DEL} \} \\ &\quad \mathbin{\mathbf{U}}(k : k \geq 1 : (k,0) SYNC(b, c)) \ \underline{b} \ (1, 0) SYNC(c, d) \\ &= \quad \{ \text{blending is universally disjunctive} \} \\ &\quad \mathbin{\mathbf{U}}(k : k \geq 1 : (k, 0) SYNC(b, c) \ \underline{b} \ (1,0) SYNC(c, d)) \\ &= \quad \{ \text{property 1.40} \} \\ &\quad \mathbin{\mathbf{U}}(k : k \geq 1 : (k+1, 0) SYNC(b, d)) \\ &= \quad \{ (1, 0) SYNC(b, d) \subseteq (2, 0) SYNC(b, d) \} \\ &\quad \mathbin{\mathbf{U}}(k : k \geq 0 : (k+1, 0) SYNC(b, d)) \\ &= \quad \{ \text{renaming the dummy} \} \\ &\quad \mathbin{\mathbf{U}}(k : k \geq 1 : (k, 0) SYNC(b, d)) \\ &= \quad \{ \text{def. of DEL} \} \\ &\quad DEL(b, d) \end{aligned}$$

(End of property and proof)

We now introduce a class of trace structures with an alphabet of four symbols that, similar to SYNC, may be appreciated as the specification of a synchronization mechanism. This new class is called QSYNC. For

natural k and l, such that $k \geq 0 \wedge l \geq 0 \wedge k+l > 0$, and distinct symbols a, b, c, and d, the trace structure $(k, l)QSYNC(a, b, c, d)$ equals

$$\begin{aligned} &\langle \{ x : x \in \{a, b, c, d\}^* \\ &\quad \wedge \underline{A}(y, z : x = y\,z : -l \leq (y \underline{N} a + y \underline{N} b) - (y \underline{N} c + y \underline{N} d) \leq k) \\ &\quad : x \\ &\quad \} \\ &, \{a, b, c, d\} \\ &\rangle . \end{aligned}$$

Without proofs we mention two more rules of addition.

Property 1.43

For natural numbers k, l, m, and n, such that $k+l > 0 \wedge m+n > 0$, and distinct symbols a, b, c, d, e, and f, we have

$$(k, l)\,QSYNC(a, b, c, d) \;\underline{b}\; (m, n)\,QSYNC(c, d, e, f) = (k+m, l+n)\,QSYNC(a, b, e, f) .$$

(End of property)

Property 1.44

For natural numbers k, l, m, and n, such that $k+l > 0 \wedge m+n > 0$, and distinct symbols a, b, c, and d, we have

$$(k, l)\,QSYNC(a, b, c, d) \;\underline{b}\; (m, n)\,SYNC(c, b) = (k+m, l+n)\,SYNC(a, d) .$$

(End of property)

This concludes the chapter on trace theory. Using the results obtained thus far, we develop in the next chapter a concise notation for (prefix-closed) trace structures. Such a notation is called a program notation.

2. A program notation

In this chapter we discuss a program notation for the representation of components. We introduce this notation in three steps. First we consider components without subcomponents. Next we discuss components that consist of subcomponents and relations between them. Finally we turn to the most general form a component can have, which is a combination of the two previous forms.

2.0. Commands as components

A component without subcomponents derives its trace structure from a so-called command. It is represented by a program text of the form

com C (A) : S moc

where C is the name of the component, A is an alphabet, and S is a command. With command S a trace structure TR(S) is associated. A command has one of five possible forms:

- A symbol is a so-called atomic command. Command b has trace structure TR(b) satisfying
  $TR(b) = \langle \{b\}, \{b\} \rangle$.
- If S and T are commands then S|T is a command.
  $TR(S|T) = TR(S) \cup TR(T)$.
- If S and T are commands then S;T is a command.
  $TR(S;T) = \langle \{x,y: x \in \underline{t}TR(S) \wedge y \in \underline{t}TR(T): xy\}, \underline{a}S \cup \underline{a}T \rangle$.
- If S is a command then S^* is a command.
  $TR(S^*) = \langle (\underline{t}\, TR(S))^*, \underline{a}\, TR(S) \rangle$,
  where $(\underline{t}\, TR(S))^*$ denotes the set of all finite concatenations of zero or more traces in $\underline{t}\, TR(S)$. This is called the Kleene closure.
- If S and T are commands then S,T is a command.
  $TR(S,T) = TR(S) \underline{w} TR(T)$.

Furthermore, parentheses are used in the usual way. The first four forms of commands make up the regular expressions [14]. In chapter three we show that the class of regular trace structures is closed under weaving. Consequently, the class of regular trace structures represented by commands is the class of regular trace structures.

Having defined the trace structure associated with a command, we may now define the trace structure of a component without subcomponents,

com C (A) : S moc .

Component C has trace structure TR(C) defined by

TR(C) = PREF(TR(S)) .

We impose the syntactic restriction

⊆ TR(S) = A .

Since the class of regular trace structures is, obviously, closed under PREF we have the following property.

Property 2.0

A component without subcomponents has a regular trace structure.

(End of property)

Our mechanistic appreciation of trace structures induces a mechanistic appreciation of commands. This can be seen as follows. The operation of a mechanism specified by a trace structure corresponds to the selection of symbols, one after the other, in such a way that, at any moment, the trace selected thus far is an element of the trace set. If the trace structure is given by a component then, due to the definition of TR(C), the selection of symbols is such that the trace selected thus far is a (prefix of a) trace in TR(S), the trace structure associated with the command. In the case of an atomic command the selection is straightforward since the trace structure

contains only one trace. If the command is of the form S|T then either a trace from TR(S) or one from TR(T) is selected. We may also say that either S or T is selected. If the command is of the form S;T first S is selected and then T is selected. Similarly, S^* is the finitely-repeated selection of S. Finally, if the command is of the form S,T then the trace selected is a weave of a trace from TR(S) and a trace from TR(T). We may say that both S and T are selected and that they are synchronized with respect to the atomic statements they have in common.

The trace structure of a component is defined as the prefix closure of the trace structure of its command and contains, therefore, two types of traces, viz. the "complete" ones specified by the command, and the "incomplete" ones introduced by the prefix closure. Concatenation of two traces is only of interest if the leading trace is complete. Since we have defined concatenation of commands, we have seen to it that a command's trace structure consists of complete traces only. Furthermore, since a component's trace structure may contain incomplete traces, we shall not introduce concatenation of components. In [12] no distinction is drawn between commands and components. As a consequence, traces have to be recognizable as either complete or incomplete traces to cater for concatenation. To that end, [12] introduces a termination symbol, denoted by ✓, marking the complete traces. Following our approach, no such symbol is required.

Example 2.1

com $sem_1(v, p)$: $(v; p)^*$ moc

The trace structure of sem_1 is SEM_1 as defined in example 1.0.
(End of example)

We introduce the rule that, in commands, the star has the highest binding power. Of the binary operators, the comma has the highest binding, followed by the semicolon, and then the bar, i.e. the larger symbol separates more. The bar and comma are symmetric. The bar, comma, and semicolon are associative. The semicolon distributes both at the left and at the right through bar and comma. As indicated by example 1.22, the comma does in general not distribute through the bar. According to property 1.21 the comma distributes through the bar if the commands connected by the bar have trace structures with equal alphabets.

Example 2.2

com buf_1 $(x0, x1, y0, y1)$:
$(x0; y0 \mid x1; y1)^*$
moc

The trace structure of buf_1 may be described in a way similar to SYNC. Every trace t of buf_1 and, since the trace structure of a component is prefix-closed, every prefix t of every trace of buf_1 satisfies

$$\begin{aligned} & 0 \le t.x0 - t.y0 \le 1 \\ \wedge\ & 0 \le t.x1 - t.y1 \le 1 \\ \wedge\ & 0 \le (t.x0 + t.x1) - (t.y0 + t.y1) \le 1 \ . \end{aligned}$$

The component is called a buffer since, for every trace t of buf_1, $t \upharpoonright \{y0, y1\}$ is, apart from renaming, a prefix

of $t \restriction \{x0, x1\}$. Since there are two distinct x's and two distinct y's, they encode one bit of information. Since the lead of x's over y's is at most one, we also say that the component is a one-place one-bit buffer.

The three inequalities listed above reveal that this component may also be viewed as a split binary semaphore [4]. The last inequality shows that the four symbols together form a binary semaphore. According to the other two inequalities x0 and y0 constitute a binary semaphore, and x1 and y1 too.
(End of example)

In the above example, a bit is encoded as one of two symbols. We shall often encounter this encoding in the sequel.

Example 2.3

The parity of a sequence of bits is the sum modulo 2 of the bits. A component that produces the parity of a sequence of bits has traces of the form

t e p0 for all t such that $t \in \{b0, b1\}^* \wedge$ t contains an even number of b1's ,

or t e p1 for all t such that $t \in \{b0, b1\}^* \wedge$ t contains an odd number of b1's ,

and prefixes thereof. The symbol e may be viewed as indicating the end of a sequence of bits, whereas p0 and p1 denote even and odd parity respectively.

If we want to derive a program text for this component we massage the above specifications into a command. This directly leads to

S0 ; e ; p0 | S1 ; e ; p1

where $\underline{a}TR(S0) = \{b0, b1\}$ and for each t, $t \in \{b0, b1\}^*$,

$t \in \underline{t}TR(S0) \equiv t \restriction \{b1\}$ has even length.

S_1 satisfies similar conditions. Next we calculate $TR(S_0)$ as follows.

$$TR(S_0)\restriction\{b_0\} = TR(b_0^*) \;\wedge\; TR(S_0)\restriction\{b_1\} = TR((b_1; b_1)^*)$$
$$\equiv \quad \{ \text{ def. of } \underline{w} \}$$
$$TR(S_0) = TR(b_0^*) \;\underline{w}\; TR((b_1; b_1)^*)$$
$$\equiv \quad \{ \text{ def. of } , \}$$
$$TR(S_0) = TR(b_0^*, (b_1; b_1)^*)$$

For S_1 we find

$$TR(S_1)\restriction\{b_0\} = TR(b_0^*) \;\wedge\; TR(S_1)\restriction\{b_1\} = TR((b_1; b_1)^*; b_1)$$
$$\equiv \quad \{ \text{ def. of } \underline{w} \}$$
$$TR(S_1) = TR(b_0^*) \;\underline{w}\; TR((b_1; b_1)^*; b_1)$$
$$\equiv \quad \{ \text{ def. of } , \}$$
$$TR(S_1) = TR(b_0^*, ((b_1; b_1)^*; b_1)) \;.$$

As a result the program text becomes

```
com parity ( b0, b1, e, p0, p1) :
      b0*, (b1; b1)* ; e; p0 | b0*, ((b1; b1)*; b1); e; p1
moc
```

(End of example)

Example 2.4

A binary variable is a component with an alphabet of four symbols, x0, x1, y0, and y1 say, and every trace t satisfies

$$\underline{A}(s:\ t = s\ y0 :\ \underline{E}(u::\ s\restriction\{x0, x1, y1\} = u\ x0))$$
$$\wedge\ \underline{A}(s:\ t = s\ y1 :\ \underline{E}(u::\ s\restriction\{x1, x0, y0\} = u\ x1)) \;,$$

and the trace structure is prefix-closed. The condition on t is equivalent to

$$\underline{E}(s::\ (t = s\ y0 \wedge \underline{E}(u::\ s\restriction\{x0, x1, y1\} = u\ x0))$$
$$\vee\ (t = s\ y1 \wedge \underline{E}(u::\ s\restriction\{x1, x0, y0\} = u\ x1))$$
$$\vee\ t = s\ x0$$
$$\vee\ t = s\ x1$$
$$)$$
$$\vee\ t = \varepsilon \;.$$

The condition $s \upharpoonright \{x0, x1, y1\} = u\,x0$ may be written as

$$s \in \underline{t}\,TR(U;\ x0;\ y0^*) \ \wedge\ u \in \underline{t}\,TR(U) \ .$$

Hence, the condition on t may be written as

$$\begin{aligned} t \in \underline{t}\,TR\,(&U;\ x0;\ y0^*;\ y0 \\ &\mid U;\ x1;\ y1^*;\ y1 \\ &\mid U;\ x0 \\ &\mid U;\ x1 \\ &) \end{aligned}$$

$$\vee\ t = \varepsilon$$

which is equivalent to

$$t \in \underline{t}\,TR(U;\ x0;\ y0^* \mid U;\ x1;\ y1^*) \ \vee\ t = \varepsilon \ .$$

Due to the prefix-closedness of the trace structure, the condition on u, viz. $u \in \underline{t}\,TR(U)$, is equivalent to the one on t, and, therefore the program text becomes

```
com var(x0, x1, y0, y1):
      (x0; y0* | x1; y1*)*
moc .
```

The trace structure of this component has three states : $[\varepsilon]$, $[x0]$, and $[x1]$. From a mechanistic point of view, the three states stand for "variable uninitialized", "variable has value 0", and "variable has value 1". The symbols x0 and x1 then stand for "assignment of value 0 and 1" respectively, and the symbols y0 and y1 for "inspection of value 0 and 1" respectively. Notice that the variable is constructed in such a way that it must be initialized before it can be inspected.
(End of example)

Example 2.5

A full adder is a component that repeatedly accepts three one-bit numbers and generates one two-bit number that is the sum of the other three. Let the numbers to be added be a, b, and c, satisfying

$0 \le a < 2 \;\wedge\; 0 \le b < 2 \;\wedge\; 0 \le c < 2$

and let the sum be represented by d and e such that

$0 \le d < 2 \;\wedge\; 0 \le e < 2 \;\wedge\; a+b+c = 2*d + e$.

The literature abounds with programs obeying this specification and sharing the property of disturbing the symmetry among a, b, and c. This is achieved by calling c the carry-in, d the carry-out, and by distinguishing the carry-kill, carry-propagate, and carry-generate conditions, i.e. $a+b=0$, $a+b=1$, and $a+b=2$ respectively. Merely observing that d equals the majority and e the oddity of a, b, and c, is more attractive and yields the following, symmetric program text.

```
com full adder (a0, a1, b0, b1, c0, c1, d0, d1, e0, e1):
    ( a0, b0;  d0, (c0;  e0|c1;  e1)
    | b0, c0;  d0, (a0;  e0|a1;  e1)
    | c0, a0;  d0, (b0;  e0|b1;  e1)
    | a1, b1;  d1, (c0;  e0|c1;  e1)
    | b1, c1;  d1, (a0;  e0|a1;  e1)
    | c1, a1;  d1, (b0;  e0|b1;  e1)
    )*
moc
```

(End of example)

Example 2.6

Numbers may be represented in various ways. In this example we consider the conversion between the binary code and the Gray code. In both codes a sequence of bits is used to represent a number. With the binary code, sequence $b(i:\ 0 \le i < n)$, $n>0$, represents the number

$$\underline{S}(i:\ 0 \le i < n:\ b(i) * 2^i) \ ,$$

whereas the Gray code uses $g(i:\ 0 \le i < n)$, $n>0$, to represent

$$\underline{S}(i:\ 0 \le i < n:\ (\underline{S}(j:\ i \le j < n:\ g(j)) \ \underline{\text{mod}}\ 2) * 2^i) \ .$$

For sequences $b(i: 0 \le i < n)$ and $g(i: 0 \le i < n)$, $n > 0$, representing the same number we derive from the above formulae

$$\underline{A}(i: 0 \le i < n: b(i) = \underline{S}(j: i \le j < n: g(j)) \ \underline{\text{mod}}\ 2)$$

and

$$g(n-1) = b(n-1)$$
$$\wedge\ \underline{A}(i: 0 \le i < n-1: g(i) = (b(i) + b(i+1)) \ \underline{\text{mod}}\ 2) \ .$$

A distinguishing feature of the Gray code is that two numbers whose difference is 1 are represented by two sequences of bits that differ in 1 bit. This makes the Gray code particularly useful in minimizing errors in analog-digital conversions.

Using b, c, and d for sequence b, and g, h, and i for sequence g, programs for conversion between 3-bit sequences are

```
com binary to Gray (b0, b1, c0, c1, d0, d1, g0, g1, h0, h1, i0, i1):
    (( b0, c0; g0 | b0, c1; g1 | b1, c0; g1 | b1, c1; g0 )
    ,( c0, d0; h0 | c0, d1; h1 | c1, d0; h1 | c1, d1; h0 )
    ,( d0; i0 | d1; i1)
    )*
moc
```

and

```
com Gray to binary(b0, b1, c0, c1, d0, d1, g0, g1, h0, h1, i0, i1):
    (( i0; d0 | i1; d1 )
    ,( h0, d0; c0 | h1, d1; c0 | h1, d0; c1 | h0, d1; c1)
    ,( g0, c0; b0 | g1, c1; b0 | g1, c0; b1 | g0, c1; b1)
    )*
moc .
```

(End of example)

2.1. Components and subcomponents

We now turn to components that consist of a number of subcomponents and relations between them. Such a component is represented by a program text of the form

com C (A) :
    sub $s_0 : C_0, \ldots, s_{n-1} : C_{n-1}$
    $a_0 = b_0, \ldots, a_{m-1} = b_{m-1}$
moc .

Component C has n subcomponents, named s_i, $0 \le i < n$. The n names s_i are distinct. We say that subcomponent s_i is of type C_i, where C_i is a component with a given trace structure $TR(C_i)$. To distinguish between A and the alphabets of the C_i we introduce compound symbols. A compound symbol is a symbol s.b in which s is the name of a subcomponent and b is a symbol in the alphabet of that subcomponent's type. Symbols that are not compound are called simple. If A is an alphabet of simple symbols then s.A denotes the alphabet of compound symbols obtained by changing each symbol b, $b \in A$, into s.b. A trace, a trace set, and a trace structure can be changed similarly. The alphabet of a component contains simple symbols only.

For component C as defined above we define alphabet B as follows.

$$B = U(i : 0 \le i < n : s_i.(\underline{\alpha} TR(C_i))) \cup A$$

We impose three syntactic restrictions on the m equalities occurring in C, viz.

- $a_j \in B \wedge b_j \in B$ for all $j : 0 \le j < m$;
- a_j and b_j belong to two different (of the $n+1$) alphabets, for all $j : 0 \le j < m$;
- each symbol in B occurs exactly once in the m equalities.

The trace structure of component C satisfying the above restrictions is defined to be the blend of the trace structures of the subcomponents. Two symbols occurring in one equality are considered to be the same symbol. This is realized by systematically replacing, in the trace structures, one of them by the other. If a simple symbol is equated to a compound symbol, the compound symbol is to be replaced by the simple symbol. If two compound symbols are equated it is irrelevant which one replaces the other. Due to the last syntactic restriction each symbol in $B \setminus A$ is, then, either equated to a symbol in A or common to exactly two alphabets of the trace structures of the subcomponents. According to property 1.28 blending is then associative, and yields A as the alphabet, i.e.

$$TR(C) = \underline{B}(i : 0 \leq i < n : s_i.TR(C_i))$$

where

$$\underline{a}\, TR(C) = A \ .$$

Due to the associativity, the order in which the subcomponents occur in the component is irrelevant. The order of the equations is irrelevant too. By property 1.32 this kind of components preserve prefix-closedness of their trace structures.

Notice that $TR(C)$ depends on the specification of the subcomponents, i.e. the $TR(C_i)$'s, and not on the way in which those subcomponents are constructed. This is essential in admitting hierarchical design.

Example 2.7

Let sem_1 be given as in example 2.1.

```
com  sem2 (v, p) :
     sub  s0, s1 :  sem1
     v = s0.v,   s0.p = s1.v,   s1.p = p
moc
```

The text $s0, s1 : sem_1$ is short for $s0: sem_1$, $s1: sem_1$.

The trace structures of the two subcomponents are $(1, 0)$ SYNC $(s0.v,\ s0.p)$ and $(1, 0)$ SYNC $(s1.v,\ s1.p)$ as defined on page 31. The trace structure of sem_2 is, due to the equalities, the blend of $(1, 0)$ SYNC $(v,\ s0.p)$ and $(1, 0)$ SYNC $(s0.p,\ p)$, which is, according to property 1.40, $(2, 0)$ SYNC $(v,\ p)$. This component is called a ternary semaphore.
(End of example)

Example 2.8

We consider a k-place one-bit buffer. For $k=1$ we use the component from example 2.2. For $k>1$ it consists of a concatenation of two smaller buffers.

com $buf_k(x0, x1, y0, y1)$:
sub $l: buf_1$, $r: buf_{k-1}$
$x0 = l.x0$, $x1 = l.x1$,
$l.y0 = r.x0$, $l.y1 = r.x1$,
$r.y0 = y0$, $r.y1 = y1$
moc

(End of example)

Example 2.9

A divide-by-n component is a component with an alphabet of two symbols, x and y say, whose trace structure is
PREF $(\ <\{i:\ i \geq 0:\ (x^n\ y)^i\},\ \{x, y\}>\ )$.
For $n=1$ this component is a binary semaphore. Programs for all greater powers of 2, i.e. $n=2^k \wedge k \geq 1$, are given below.
For $k=1$

com $div_k(x, y)$:
$(x; x; y)^*$
moc

and for $k>1$

```
com  div_k (x, y):
     sub d: div_{k-1} , h: halve
     h.lx = x, h.rx = d.x, h.ly = d.y, h.ry = y
moc

com  halve (lx, rx, ly, ry):
     ((rx ; lx)* ; ly ; (rx ; lx)* ; ly ; ry)*
moc  .
```

(End of example)

2.2. General components

The most general form a component can have is a combination of the two forms discussed, viz. a component with subcomponents, equalities, and a command. Such a component is represented by a program text of the form

com C (A) :
    sub $s_0 : C_0$, ... , $s_{n-1} : C_{n-1}$
    $a_0 = b_0$, ... , $a_{m-1} = b_{m-1}$
    S
moc .

Alphabet B is defined as in the case of a component without command, viz.

$$B = U(i : 0 \leq i < n : s_i.(\underline{a}\, TR(C_i))) \cup A .$$

We impose three syntactic restrictions on the m equalities occurring in C , viz.

- $a_j \in B \wedge b_j \in B$ for all $j : 0 \leq j < m$;
- a_j and b_j belong to two different (of the $n+1$) alphabets , for all $j : 0 \leq j < m$;
- each symbol in B occurs at most once in the m equalities.

Of these only the last restriction differs from the restrictions in the case of a component without command. Similar to the case of a component without subcomponents we impose the syntactic restriction

$$\underline{a}\, TR(S) = B \setminus U(j : 0 \leq j < m : \{ a_j, b_j \}) .$$

The trace structure of component C is defined to be

$$TR(C) = \underline{B}(i : 0 \leq i < n : s_i.TR(C_i)) \ \underline{b}\ PREF(TR(S)) .$$

Due to the restrictions we have again

$$\underline{a}\, TR(C) = A$$

and the associativity of the blending operation in the definition of TR(C). By property 1.32 this kind of components preserve prefix-closedness of their trace structures. Consequently, the trace structures of all components are

prefix - closed.

Example 2.10

Let sem_1 be as in example 2.1.

com sem_4 (v, p) :
  sub s0, s1 : sem_1
  s0.p = s1.v
  $(v;\ s0.v)^*,\ (s1.p;\ p)^*$
moc

The trace structure can be calculated as follows. First we derive

$$
\begin{aligned}
& PREF(TR((v;\ s0.v)^*,\ (s1.p;\ p)^*)) \\
= \ & \{ \text{def. of ,} \} \\
& PREF(TR((v;\ s0.v)^*)\ \ \underline{w}\ \ TR((s1.p;\ p)^*)) \\
= \ & \{ \text{property 1.26} \} \\
& PREF(TR((v;\ s0.v)^*)\ \ \underline{b}\ \ TR((s1.p;\ p)^*)) \\
= \ & \{ \text{property 1.33} \} \\
& PREF(TR((v;\ s0.v)^*))\ \ \underline{b}\ \ PREF(TR((s1.p;\ p)^*)) \\
= \ & \{ \text{example 2.1} \} \\
& (1,0)\ SYNC(v,\ s0.v)\ \ \underline{b}\ \ (1,0)\ SYNC(s1.p,\ p)\ \ ,
\end{aligned}
$$

and then

$$
\begin{aligned}
& TR(sem_4) \\
= \ & \{ \text{def. of TR, } s0.p = s1.v \} \\
& PREF(TR((v;\ s0.v)^*,\ (s1.p;\ p)^*)) \\
& \quad \underline{b}\ (1,0)\ SYNC(s0.v,\ s0.p)\ \ \underline{b}\ \ (1,0)\ SYNC(s0.p,\ s1.p) \\
= \ & \{ \text{see above} \} \\
& \quad (1,0)\ SYNC(v\ \ ,\ \ s0.v) \\
& \underline{b}\ (1,0)\ SYNC(s1.p,\ p\ \ ) \\
& \underline{b}\ (1,0)\ SYNC(s0.v,\ s0.p) \\
& \underline{b}\ (1,0)\ SYNC(s0.p,\ s1.p) \\
= \ & \{ \text{property 1.40} \} \\
& (4,0)\ SYNC(v,\ p)\ \ .
\end{aligned}
$$

(End of example)

Strictly speaking, this third form a component can have is superfluous. We can change any component into one that has either no subcomponents or no command. We shall, however, retain the general form since it allows more compact and elegant programs.

Example 2.11

Let sem_1 be as in example 2.1 and let k be a natural number satisfying $k \geq 2$.

com sem_k (v, p) :
sub s : sem_{k-1}
((v | s.p) ; (s.v | p))*
moc

We prove that sem_k has $(k, 0)\,SYNC(v, p)$ as its trace structure, by mathematical induction on k. For $k = 1$ this is clear from example 2.1. For $k > 1$ it follows from
$PREF(TR(((v | s.p) ; (s.v | p))^*)) = (1, 0)\,QSYNC(v, s.p, s.v, p)$,
the induction hypothesis, and property 1.44.

From a mechanistic point of view, the above component is a bit suspect. The trace structure of the component contains traces of bounded length that may be the result of combining a trace from the subcomponent and a trace from the component's command that are both of unbounded length. This implies that for a bounded amount of external communication the amount of internal communication may be unbounded. This phenomenon is called unbounded chatter. In general, unbounded chatter makes the implementation of programs much more difficult. In chapters four and five we shall discuss implementation issues.

In order to remove the unbounded chatter we might first rewrite the command as follows, without changing its

trace structure. Distribution of semicolon through bar yields

$$(v;\ s.v \mid v;\ p \mid s.p;\ s.v \mid s.p;\ p)^* \ .$$

Observing that selection of the command

$$s.p;\ s.v$$

changes neither the state of the component nor the state of the subcomponent, we may omit it from the component's command. This does affect the command's trace structure but it does not affect the component's trace structure. The resulting command is

$$(v;\ s.v \mid v;\ p \mid s.p;\ p)^* \ .$$

As can easily be seen, this version does not exhibit unbounded chatter. Internal communication can even be reduced further by observing that selection of

$$v;\ s.v$$

followed by

$$s.p;\ p$$

is, in effect, equivalent to the selection of

$$v;\ p$$

while the latter induces no internal communication. To exclude the former selection we change the command into

$$((v;\ s.v)^*;\ v;\ p \mid s.p;\ p)^* \ .$$

We may, furthermore, exclude all traces starting with $s.p$ from the command's trace structure without affecting the component's trace structure since those traces can not be combined with a trace from the subcomponent. This leads to the command

$$((v;\ s.v)^*;\ v;\ p;\ (s.p;\ p)^*)^* \ .$$

(End of example)

Example 2.12

This example is a binary stack of bounded depth. A binary stack has an alphabet of four symbols: two opening parentheses, denoted by x0 and x1, and two closing parentheses, denoted by y0 and y1. The following is a

syntactic definition of all so-called well-nested sequences of these parentheses.

$\langle seq \rangle ::= (x0 \ \langle seq \rangle \ y0 \mid x1 \ \langle seq \rangle \ y1)^*$

where $(\ldots)^*$ denotes zero or more instances of the enclosed. The trace set of a stack is the set of all prefixes of well-nested sequences. Our example is not a general stack but a stack of certain depth. Its trace set is a subset of the trace set of a general stack and contains all those traces in which the lead of x's over y's is at most the stack's depth.

Each trace t of the trace set leads to a certain contents of the stack, defined as follows. From t symbols are removed in pairs as long as $t \restriction \{y0, y1\} \neq \epsilon$. The two symbols removed per step are the leftmost $y0$ or $y1$, and the symbol directly preceding it. From the definition of the trace set we conclude that a pair of symbols removed is either $x0\ y0$ or $x1\ y1$. The y-free sequence thus obtained is the contents of the stack.

From a mechanistic point of view, we may say that $x0$ and $x1$ correspond to "push 0" and "push 1" respectively. The symbols $y0$ and $y1$ correspond to "pop 0" and "pop 1" respectively.

A stack of depth one is a one-place one-bit buffer.

```
com stack1 (x0, x1, y0, y1) :
    (x0; y0 | x1; y1)*
moc
```

Observe that a unary stack, i.e. a stack in which no distinction is made between 0 and 1, is a semaphore. Inspired by the discussion of the semaphore in the previous example, we propose for a stack of depth $k+1$, for all k: $k \geq 1$, the program text

```
com  stack_{k+1} (x0, x1, y0, y1) :
     sub  s : stack_k
     (( x0  ; s.x0 | x1  ; s.x1 )*
     ;( x0  ; y0   | x1  ; y1   )
     ;( s.y0 ; y0  | s.y1 ; y1  )*
     )*
moc  .
```

An invariant relation of each of the repetitions of this command is

for all traces t, such that t ∈ {x0, x1, y0, y1}*,
the trace selected thus far, followed by t, is a well-nested sequence
≡ the contents of the stack, followed by t, is a well-nested sequence .

(End of example)

Example 2.13

In this example we discuss a program implementing mutual exclusion in a ring of subcomponents. The example is taken from [6] and [17], where it is called "perpetuum mobile".

A ring of n subcomponents, n>1, consists of n-1 subcomponents of type

```
com  node (l, r, u, v) :
     (l; r | l; u; v; r)*
moc
```

and one of type

```
com  enode (l, r, u, v):
     r; (l; r | l; u; v; r)*
moc  .
```

Apart from initialization, these two types of subcomponents are equivalent. The n subcomponents in the ring are

numbered 0 through n-1. In this example all subscripts are to be taken modulo n. The component is

com ring $(u_0, v_0, \ldots, u_{n-1}, v_{n-1})$:
sub $s_0, \ldots, s_{n-2}$: node, s_{n-1} : enode
$s_i.r = s_{i+1}.l$, $u_i = s_i.u$, $v_i = s_i.v$ for all $i : 0 \le i < n$
moc .

We shall show that the above component has trace structure

$$PREF(TR((u_0 ; v_0 \mid \ldots \mid u_{n-1} ; v_{n-1})^*)) ,$$

i.e. the ring provides mutual exclusion.

First we prove, for distinct symbols x_i and commands y_i, $0 \le i < n$, such that

$$\underline{A}(j : 0 \le j < n : x_j \notin \underline{a}TR(y_i) \wedge (i=j \vee \underline{a}TR(y_i) \cap \underline{a}TR(y_j) = \{\})) ,$$

that we have

(0) $PREF(TR(S_{n-1})) = PREF(TR((y_0 ; \ldots ; y_{n-1})^*))$

where S_j, $0 \le j < n$, is a command that satisfies

$$PREF(TR(S_j))$$
$$= \underline{B}(i : 0 \le i < n \wedge i \ne j : PREF(TR((x_i ; y_i ; x_{i+1})^*)))$$
$$\underline{b}\ PREF(TR((x_{j+1} ; x_j ; y_j)^*)) .$$

Remember that $\underline{B}$ denotes continued blending. Notice that we have for all j, $0 \le j < n$,

(1) $PREF(TR((x_{j+1} ; y_{j+1} ; x_{j+2})^*))$
$\underline{b}\ PREF(TR((x_{j+1} ; x_j ; y_j)^*))$
$= PREF(TR(x_{j+1} ; y_{j+1} ; x_{j+2} ; (x_{j+1} ; y_{j+1} ; x_{j+2})^*))$
$\underline{b}\ PREF(TR(x_{j+1} ; x_j ; y_j ; (x_{j+1} ; x_j ; y_j)^*))$
$= PREF(TR(y_{j+1} ; x_{j+2} ; (x_{j+1} ; y_{j+1} ; x_{j+2})^*))$
$\underline{b}\ PREF(TR(x_j ; y_j ; (x_{j+1} ; x_j ; y_j)^*))$
$= PREF(TR(y_{j+1} ; (x_{j+2} ; x_{j+1} ; y_{j+1})^*))$
$\underline{b}\ PREF(TR((x_j ; y_j ; x_{j+1})^*))$.

As a consequence, we have for all j, $0 \le j < n$,

$$
\begin{array}{ll}
 & \mathrm{PREF}(\mathrm{TR}(S_j)) \\
= & \quad \{ \text{ def. of } S_j \} \\
 & \quad \underline{\mathrm{B}}(i: 0 \le i < n \wedge i \neq j: \mathrm{PREF}(\mathrm{TR}((x_i ; y_i ; x_{i+1})^*))) \\
 & \underline{\mathrm{b}}\ \mathrm{PREF}(\mathrm{TR}((x_{j+1} ; x_j ; y_j)^*)) \\
= & \quad \{ \ n>1, \text{ then } (1) \ \} \\
 & \quad \underline{\mathrm{B}}(i: 0 \le i < n \wedge i \neq j+1: \mathrm{PREF}(\mathrm{TR}((x_i ; y_i ; x_{i+1})^*))) \\
 & \underline{\mathrm{b}}\ \mathrm{PREF}(\mathrm{TR}(y_{j+1} ; (x_{j+2} ; x_{j+1} ; y_{j+1})^*)) \\
= & \quad \{ \ n>1, \text{ hence all } x_i \text{ are common to two trace structures} \ \} \\
 & \quad \mathrm{PREF}(\mathrm{TR}(y_{j+1} ; S_{j+1})) \ .
\end{array}
$$

Applying the above equality n times we find

$$\mathrm{PREF}(\mathrm{TR}(S_{n-1})) = \mathrm{PREF}(\mathrm{TR}(y_0 ; \ldots ; y_{n-1} ; S_{n-1}))$$

from which (0) follows.

Substituting $s_i.l$ for x_i, and $(\varepsilon \mid u_i ; v_i)$ for y_i, $0 \le i < n$, equality (0) shows

$$
\begin{array}{ll}
 & \mathrm{TR}(\mathrm{ring}) \\
= & \quad \{ \text{ def. of } \mathrm{TR}(\mathrm{ring}) \ \} \\
 & \quad \underline{\mathrm{B}}(i: 0 \le i < n-1: s_i.\mathrm{TR}(\mathrm{node})) \\
 & \underline{\mathrm{b}}\ s_{n-1}.\mathrm{TR}(\mathrm{enode}) \\
= & \quad \{ \text{ def. of } \mathrm{TR}(\mathrm{node}) \text{ and } \mathrm{TR}(\mathrm{enode}) \ \} \\
 & \quad \underline{\mathrm{B}}(i: 0 \le i < n-1: \mathrm{PREF}(\mathrm{TR}(s_i.l ; (\varepsilon \mid u_i ; v_i) ; s_{i+1}.l)^*))) \\
 & \underline{\mathrm{b}}\ \mathrm{PREF}(\mathrm{TR}((s_0.l ; s_{n-1}.l ; (\varepsilon \mid u_{n-1} ; v_{n-1}))^*)) \\
= & \quad \{ \ (0) \text{ with substitutions} \ \} \\
 & \quad \mathrm{PREF}(\mathrm{TR}(((\varepsilon \mid u_0 ; v_0) ; \ldots ; (\varepsilon \mid u_{n-1} ; v_{n-1}))^*)) \\
= & \quad \{ \text{ calculus } \} \\
 & \quad \mathrm{PREF}(\mathrm{TR}((u_0 ; v_0 \mid \ldots \mid u_{n-1} ; v_{n-1})^*)) \ .
\end{array}
$$

(End of example)

2.3. Recursion

We say that a component C has component D as a composing part if C has a subcomponent that either is of type D or has D as a composing part. A component is called recursive if it has itself as a composing part. The way in which we have defined components excludes them to be recursive. We could have allowed recursion but in that case one has to be more careful with the formula defining $TR(C)$, which is then a recursive equation. It may have several solutions. Such a solution is not necessarily a regular trace structure and it need not be prefix-closed. In this monograph we restrict ourselves to nonrecursive components and merely present, in this section, a few examples of recursion. For a discussion of recursion we refer to [27], and we follow the suggestion that, in the ordering of page 11, the least solution containing the empty trace be the trace structure of C. We call this the least, nontrivial solution.

Example 2.14

```
com  sem (v, p) :
     sub s : sem
     ((v; s.v)*; v; p; (s.p; p)* )*
moc
```

According to [27] the trace structure of this component satisfies

$$TR(sem) = DEL(v, p) \quad ,$$

where DEL is as defined on page 34. This is not surprising if we compare the above program text with the ultimate program text given for sem_k in example 2.11 on page 53, and if we remember that $DEL(v, p)$ is the limit of $TR(sem_k)$ for increasing k, $k \geq 1$.

For each prefix t of a trace of the command we have that the length of $t \upharpoonright \{s.v, s.p\}$ is at most the length of $t \upharpoonright \{v, p\}$. As shown in [27] this implies that the least, nontrivial solution of the recursive equation defining TR(sem) equals the greatest solution. Had we chosen to write sem as

```
com  sem(v, p) :
     sub  s : sem
     ((v | s.p) ; (s.v | p))*
moc
```

then the least, nontrivial solution would again be DEL(v, p) but it would differ from the greatest solution, which in this case is $< \{v, p\}^*, \{v, p\} >$.
(End of example)

Example 2.15

In this example we discuss a binary sorter. A binary sorter is a component that maintains a bag of binary values. At any time a value may be added to the bag. A value may be removed only if it occurs in the bag. Furthermore, in order to remove a 1 from the bag, the bag should not contain a 0. We use symbols x0 and x1 for adding a 0 or 1 to the bag respectively, and symbols y0 and y1 to remove a 0 or 1 respectively. The trace structure of the binary sorter is prefix-closed, and each trace t satisfies

$$\begin{aligned} & t \underline{N}\, x0 \geq t \underline{N}\, y0 \\ \wedge\ & t \underline{N}\, x1 \geq t \underline{N}\, y1 \\ \wedge\ & \underline{A}(s:\ t = s\, y1 :\ s \underline{N}\, x0 = s \underline{N}\, y0) \ . \end{aligned}$$

In order to derive a program text for the sorter we first concentrate on the first two inequalities. Each of these specifies an unbounded semaphore, together they specify an unbounded binary bag. The program text for the bag is

```
com  bag (x0, x1, y0, y1) :
     sub  b0, b1 : sem
     x0 = b0.v,   x1 = b1.v,   y0 = b0.p,   y1 = b1.p
moc  .
```

If we want to add the restriction

$$\underline{A}(s:\ t = s\,y1\ :\ s\ \underline{N}\ x0 = s\ \underline{N}\ y0)$$

to the bag in order to obtain the sorter, then we have to disable the selection of $y1$ if, in the trace selected thus far, the number of $x0$'s exceeds the number of $y0$'s, i.e. if in the trace of $b0$ selected thus far, the number of v's exceeds the number of p's. In order to control the selection of $x0$ and $y1$ we rewrite bag as follows.

```
com  bag (x0, x1, y0, y1) :
     sub  b0, b1 : sem
     x1 = b1.v,   y0 = b0.p
     (x0; b0.v)* , (b1.p; y1)*
moc
```

We have now replaced two equalities by blendings with trace structures $(1,0)SYNC(x0,\ b0.v)$ and $(1,0)SYNC(b1.p, y1)$. According to property 1.42 this does not change $TR(bag)$. Notice that we are not interested in restricting selection of $x1$ since it does not occur in the requirement we are about to add. The symbol $y0$, however, does occur in it. Yet we need not restrict selection of $y0$ since our first inequality

$$t\ \underline{N}\ x0\ \geq\ t\ \underline{N}\ y0$$

allows the requirement we are dealing with to be written as

$$\underline{A}(s:\ t = s\,y1\ :\ s\ \underline{N}\ x0\ \leq\ s\ \underline{N}\ y0)\ ,$$

which is not falsified by selection of $y0$.

Next we slightly modify $b0$ by adding to it a symbol, e say, that occurs in a trace of $b0$ only if it is preceded by an equal number of v's and p's. We do so

by changing the type of b0 from sem to sam .

```
com sam(v, p, e):
    sub s : sam
    e*; ((v|s.p); (s.v| p); (s.e; e*)*)*
moc
```

An invariant of the outer repetition of sam is that for the trace thus far selected, t say,

$$t\underline{N}v + t\underline{N}\,s.p = t\underline{N}\,s.v + t\underline{N}\,p \ ,$$

i.e.

$$t\underline{N}v - t\underline{N}\,p = t\underline{N}\,s.v - t\underline{N}\,s.p \ ,$$

i.e.

$$(t\underline{N}v = t\underline{N}\,p) \equiv (t\underline{N}\,s.v = t\underline{N}\,s.p) \ ,$$

which justifies the possibility of selecting e if s.e can be selected. After selection of the command

$$(v \mid s.p)$$

in the outer repetition we have, similarly,

$$t\underline{N}v - t\underline{N}\,p = 1 + t\underline{N}\,s.v - t\underline{N}\,s.p$$

which, on account of

$$t\underline{N}\,s.v \geq t\underline{N}\,s.p \ ,$$

implies

$$t\underline{N}v > t\underline{N}\,p \ ,$$

which justifies the impossibility of selecting e . This concludes the changes of b0 .

Finally we transform bag into sorter . In order to guarantee that, in the trace thus far selected, the number of x0's equals the number of y0's , the selection of y1 is preceded by selection of b0.e . Selection of b0.e implies equality of the number of b0.v's and b0.p's , i.e. equality of the number of b0.v's and y0's . In order to guarantee equality of the number of x0's and b0.v's the selections of x0; b0.v and b0.e; b1.p; y1 are made mutually exclusive. We thus obtain

```
com sorter (x0, x1, y0, y1):
    sub  b0 : sam,   b1 : sem
    x1 = b1.v ,   y0 = b0.p
    (x0;  b0.v | b0.e ;  b1.p;  y1)*
moc .
```

(End of example)

This concludes our discussion of the program notation. It is often stated that, although regular expressions may be used to specify any finite process, they are not always as concise as one might want (e.g. [7]). Examples of the latter kind are code conversion, buffer, and divide-by-n components. In examples 2.6, 2.8, and 2.9 we have shown that weaving and blending, expressed by comma and sub-components, admit concise programs for the specification of these components.

3. Finite state machines

In this chapter two independent subjects are discussed. Both of them are related to finite state machines. First, we show that the class of regular trace structures is closed under weaving and blending. We do so by constructing appropriate finite state machines. Second, we discuss an algorithm for the minimization of the number of states of finite state machines. In these two sections our manipulative needs are slightly different and, hence, we shall use two slightly different notations for finite state machines.

3.0. The relation of commands to finite state machines

It is well-known that regular expressions and finite state machines define the same set of trace structures: the regular trace structures. Rules exist for deriving a, possibly non-deterministic, finite state machine from a regular expression built up from symbols of a certain alphabet and the operations concatenation, union, and Kleene closure, such that expression and machine define the same trace structure. We shall show that the class of regular trace structures is closed under weaving and blending by constructing a finite state machine accepting the weave or blend of two trace structures defined by regular expressions. We use the commands from section 2.0 to denote regular expressions.

We describe a finite state machine by a quintuple

$< B, Q, iq, fq, T >$

where

B is an alphabet,
Q is a finite set of states,
iq is an element of Q, called the initial state,
fq is an element of Q, called the final state, and

T is a finite set of state transitions $p \xrightarrow{s} q$, where
$p \in Q \wedge q \in Q \wedge (s \in B \vee s = \varepsilon)$.

A trace t, $t \in B^*$, is a trace from state q to state r if $s(i: 0 \le i < n)$ and $p(i: 0 \le i < n+1)$, $n \ge 0$, exist such that

$$\underline{A}(i: 0 \le i < n : p(i) \xrightarrow{s(i)} p(i+1) \in T)$$
$$\wedge \ \underline{CONCAT}\ (i: 0 \le i < n : s(i)) = t$$
$$\wedge \ p(0) = q$$
$$\wedge \ p(n) = r \ .$$

The trace structure accepted by a finite state machine contains precisely all traces from initial state to final state, and its alphabet is the alphabet of the finite state machine. Hence, the trace structure is

$$\langle \{t: t \in B^* \wedge t \text{ is a trace from } iq \text{ to } fq : t\},\ B \rangle \ .$$

Next we recall the classic construction rules for a finite state machine, given a command. The construction is defined recursively over the command's structure.

A command consisting of a single symbol

b

corresponds to the finite state machine

$$\langle \{b\},\ \{i, f\},\ i,\ f,\ \{ i \xrightarrow{\varepsilon} i,\ i \xrightarrow{b} f,\ f \xrightarrow{\varepsilon} f \} \rangle$$

where i and f are fresh names. Notice that we have included ε-transitions from each state to itself. All finite state machines in this section will have this property.

Concatenation of commands

$E ; F$

where E corresponds to the finite state machine

$$\langle A, P, ip, fp, S \rangle$$

and F corresponds to

$$\langle B, Q, iq, fq, T \rangle \ ,$$

and P and Q are disjoint, corresponds to

$$\langle A \cup B,\ P \cup Q,\ ip,\ fq,\ S \cup T \cup \{ fp \xrightarrow{\varepsilon} iq \} \rangle \ .$$

In the same vein, union of commands

$$E \mid F$$

corresponds to

$$\langle A \cup B,\ P \cup Q \cup \{i, f\},\ i,\ f,\ S \cup T \cup \{ i \xrightarrow{\varepsilon} i,\ i \xrightarrow{\varepsilon} ip,\ i \xrightarrow{\varepsilon} iq,\ fq \xrightarrow{\varepsilon} f,\ fp \xrightarrow{\varepsilon} f,\ f \xrightarrow{\varepsilon} f \} \rangle$$

where i and f are fresh names.

Kleene closure

$$E^*$$

corresponds to

$$\langle A,\ P,\ ip,\ fp,\ S \cup \{ ip \xrightarrow{\varepsilon} fp,\ fp \xrightarrow{\varepsilon} ip \} \rangle .$$

The weave of commands

$$E, F$$

is slightly more elaborate. It corresponds to

$$\langle A \cup B,\ P \times Q,\ \langle ip, iq \rangle,\ \langle fp, fq \rangle,\ \{ pi, pj, qk, ql, s : pi \in P \wedge pj \in P \wedge qk \in Q \wedge ql \in Q \wedge (s \in A \cup B \vee s = \varepsilon) \wedge pi \xrightarrow{s \uparrow A} pj \in S \wedge qk \xrightarrow{s \uparrow B} ql \in T : \langle pi, qk \rangle \xrightarrow{s} \langle pj, ql \rangle \} \rangle .$$

The correctness of this construction follows directly from the definitions of weaving and of the trace structure accepted by a finite state machine.

The inclusion of transitions $p \xrightarrow{\varepsilon} p$ for all states p in the finite state machines as described above is essential for the concise characterization of a weave's finite state machine. Thanks to these ε-transitions we could avoid the distinction between three cases, viz. $s \in A \cap B \vee s = \varepsilon$, $s \in A \setminus B$, and $s \in B \setminus A$. Removing all ε-transitions from a finite state machine is straightforward; removing them

from the above descriptions is not straightforward.

The above construction shows that the class of regular trace structures is closed under weaving. Blending can be dealt with in a similar way. In order to obtain a finite state machine that accepts the blend, the only change required in the above construction is the change of the state transitions into

$$\langle p_i, q_k \rangle \xrightarrow{sl(A \div B)} \langle p_j, q_l \rangle$$

and of the alphabet into

$$A \div B \ .$$

From the definition on page 25 it follows that this change yields a finite state machine accepting the blend. Hence, regular trace structures are also closed under blending.

3.1. Minimization of finite state machines

In this section we present a beautiful algorithm for the minimization of finite state machines, and a proof of its correctness. Minimization of a (possibly nondeterministic) finite state machine is the construction of a deterministic finite state machine accepting the same trace structure while having the minimum number of states. The algorithm is little known despite its overwhelming simplicity. We have seen it in the literature only for the case of a deterministic finite state machine and without proof of correctness [0]. The proof that we give here is essentially due to [22].

We now present the algorithm. The reverse R M of a finite state machine M is obtained from M by first reversing all state transitions and interchanging final and initial states, by next applying the well-known subset construction to make it deterministic [14], and by finally omitting all unreachable states. We shall show that R (R M) is the minimal finite state machine accepting the same trace structure as M. We do so by showing that the traces accepted by R M are the reverse of traces accepted by M from which we conclude that R(R M) and M accept the same traces. Furthermore, we show that, for deterministic M, R M is minimal. Since for any M, R M is deterministic, R (R M) is minimal.

First we recall some standard definitions. A deterministic finite state machine is a quintuple

< B, Q, q0, F, d >

where

B is an alphabet,
Q is a finite set of states,
q0 is an element of Q, called the initial state,
F is a subset of Q, called the set of final states, and

d is the state transition function, $d: Q \times B \to Q$.
From d a function, called the closure of d and also denoted by d, $d: Q \times B^* \to Q$, is derived that satisfies

$$d(q, \varepsilon) = q \qquad \text{for all } q: q \in Q$$
$$d(q, x\,a) = d(d(q, x), a) \qquad \text{for all } q, x, a: q \in Q \wedge x \in B^* \wedge a \in B.$$

Obviously

$$d(q, x\,y) = d(d(q, x), y)$$

for all q, x, y: $q \in Q \wedge x \in B^* \wedge y \in B^*$.
Trace structure $T\,M$ accepted by M is

$$\langle \{x: x \in B^* \wedge d(q0, x) \in F: x\},\ B \rangle .$$

In the sequel it is assumed that M has reachable states only, i.e.

$$\underline{A}(q: q \in Q: \underline{E}(x: x \in B^*: q = d(q0, x))) .$$

Equivalence relation $E\,T$ on $(\underline{a}\,T)^*$ induced by a regular trace structure T is

$$x\,(E\,T)\,y \equiv \underline{A}(z: z \in (\underline{a}\,T)^*: x\,z \in \underline{t}\,T \equiv y\,z \in \underline{t}\,T)$$

for all x, y: $x \in (\underline{a}\,T)^* \wedge y \in (\underline{a}\,T)^*$. $E\,T$ is right invariant, i.e.

$$x\,(E\,T)\,y \Rightarrow x\,z\ (E\,T)\ y\,z$$

for all x, y, z: $x\,y\,z \in (\underline{a}\,T)^*$. The equivalence class containing x is denoted by $[x]$.

Note

Equivalence relation $E\,T$ differs only slightly from the one introduced in chapter one, page 30. $E\,T$ is defined on $(\underline{a}\,T)^*$ and the relation from chapter one on $\underline{t}\,PREF(T)$. IF $\underline{t}\,PREF(T) = (\underline{a}\,T)^*$ the two equivalence relations are the same, and otherwise $E\,T$ has one equivalence class more; it consists of $(\underline{a}\,T)^* \setminus \underline{t}\,PREF(T)$, while all other classes of the two equivalence relations are the same.
(End of note)

Deterministic finite state machine $D\,T$ is, for any trace structure T,

$$\langle \underline{a}T,\ (\underline{a}T)^*/(E\,T),\ [\varepsilon],\ \{x:\ x \in \underline{t}T:\ [x]\},\ d' \rangle$$

where

$$d'([x], a) = [x\,a] \qquad \text{for all } x, a:\ x \in (\underline{a}T)^* \wedge a \in \underline{a}T\ .$$

Notice that d' is well-defined since $E\,T$ is right invariant. For regular trace structure T, $(\underline{a}T)^*/(E\,T)$ is known to be finite.

Property 3.0

The minimal, deterministic finite state machine accepting T is, apart from renaming of the states, unique and is given by $D\,T$.
(End of property)

So much for the standard definitions.

Next we introduce the notions of the reverse of a trace, of a trace structure, and of a finite state machine. We warn the reader that, since they are closely related, we shall indicate each of these three with the letter R.

The reverse of a trace x will be denoted by $R\,x$ and satisfies

$$R\,a = a \qquad \text{for all symbols } a,$$
$$R\,(x\,y) = (R\,y)(R\,x) \qquad \text{for all traces } x \text{ and } y.$$

Obviously $R\,(R\,x) = x$.

The reverse of a trace structure T will be denoted by $R\,T$ and equals

$$\langle \{x:\ x \in \underline{t}T:\ R\,x\},\ \underline{a}T \rangle\ .$$

Obviously $R\,(R\,T) = T$.

Combining these two notions leads to the following property.

Property 3.1

For all traces x and trace structures T we have

$$x \in \underline{t}(R\,T) \quad \equiv \quad (R\,x) \in \underline{t}T\ .$$

(End of property)

For a deterministic finite state machine M,

$M = \langle B, Q, q0, F, d \rangle$,

the reverse is denoted by $R\,M$ and equals

$\langle B, R\,Q, F, R\,q0, b \rangle$

where b is defined such that

$q \in b(V, a) \equiv d(q, a) \in V$ for all q, a, V : $q \in Q \wedge a \in B \wedge V \subseteq Q$,

and $R\,Q$ is defined such that

$V \in R\,Q \equiv \underline{E}(x : x \in B^* : b\,(F, x) = V)$ for all V : $V \subseteq Q$,

and $R\,q0$ is defined such that

$V \in R\,q0 \equiv q0 \in V$ for all V : $V \in R\,Q$.

An important property relating the state transition functions of M and $R\,M$ is the following.

Property 3.2

For all q, V, x : $q \in Q \wedge V \in R\,Q \wedge x \in B^*$ we have

$q \in b(V, x) \equiv d(q, R\,x) \in V$.

Proof by mathematical induction on the length of x .

$q \in b(V, \varepsilon)$

$\equiv$ { def. of closure }

$q \in V$

$\equiv$ { def. of closure }

$d(q, \varepsilon) \in V$

$\equiv$ { $\varepsilon = R\,\varepsilon$ }

$d(q, R\,\varepsilon) \in V$

For all a : $a \in B$ we have

$q \in b(V, a\,x)$

$\equiv$ { property of closure }

$q \in b(b(V, a), x)$

$\equiv$ { induction hypothesis }

$d(q, R\,x) \in b(V, a)$

$\equiv$ { def. of b }

$d(d(q, R\ x), a) \in V$

$\equiv$ { def. of closure }

$d(q, (R\ x)a) \in V$

$\equiv$ { def. of R }

$d(q, R(a\ x)) \in V$.

(End of property and proof)

The next property shows that the trace structure accepted by R M is the reverse of the trace structure accepted by M .

Property 3.3

For any deterministic finite state machine M we have $T(R\ M) = R(T\ M)$.

Proof

For all x : $x \in B^*$, we have

$x \in \underline{t}(T(R\ M))$

$\equiv$ { def. of $T(R\ M)$ }

$b\ (F, x) \in R\ q0$

$\equiv$ { def. of $R\ q0$ }

$q0 \in b\ (F, x)$

$\equiv$ { property 3.2 }

$d(q0, R\ x) \in F$

$\equiv$ { def. of T M }

$(R\ x) \in \underline{t}(T\ M)$

$\equiv$ { property 3.1 }

$x \in \underline{t}(R(T\ M))$.

(End of property and proof)

According to our plan it remains to show that, for a deterministic finite state machine M, R M is minimal. According to property 3.3 we know that R M accepts trace structure $R(T\ M)$, and from property 3.0 we know that N, $N = D(R(T\ M))$, is the minimal finite state

machine accepting that trace structure. Hence, we are finished if we show that N and $R\,M$ are equal up to an isomorphism, i.e. a renaming of the states.

Substituting the definition of D in $N = D(R(T\,M))$ leads to

$$N = \langle B,\ B^*/(E(R(T\,M))),\ [\varepsilon],\ G,\ c \rangle ,$$

where

$$G = \{ x : x \in \underline{t}(R(T\,M)) : [x] \} ,$$

$$c([x], a) = [x\,a] \quad \text{for all } x, a : x \in B^* \wedge a \in B ,$$

and $[x]$ is the equivalence class of $E(R(T\,M))$ containing x.

Property 3.4

$$[x] = [y] \equiv b(F, x) = b(F, y) \quad \text{for all } x, y : xy \in B^* .$$

Proof

$$\begin{aligned}
& [x] = [y] \\
\equiv\ & \{ \text{def. of } E(R(T\,M)) \} \\
& \underline{A}(z :: x(R\,z) \in \underline{t}(R(T\,M)) \equiv y(R\,z) \in \underline{t}(R(T\,M))) \\
\equiv\ & \{ \text{property 3.1} \} \\
& \underline{A}(z :: z(R\,x) \in \underline{t}(T\,M) \equiv z(R\,y) \in \underline{t}(T\,M)) \\
\equiv\ & \{ \text{def. of } T\,M \} \\
& \underline{A}(z :: d(q0, z(R\,x)) \in F \equiv d(q0, z(R\,y)) \in F) \\
\equiv\ & \{ \text{property of closure} \} \\
& \underline{A}(z :: d(d(q0, z), R\,x) \in F \equiv d(d(q0, z), R\,y) \in F) \\
\equiv\ & \{ \text{calculus} \} \\
& \underline{A}(q, z : q = d(q0, z) : d(q, R\,x) \in F \equiv d(q, R\,y) \in F) \\
\equiv\ & \{ \text{property 3.2} \} \\
& \underline{A}(q, z : q = d(q0, z) : q \in b(F, x) \equiv q \in b(F, y)) \\
\equiv\ & \{ M \text{ has reachable states only} \} \\
& \underline{A}(q :: q \in b(F, x) \equiv q \in b(F, y)) \\
\equiv\ & \{ \text{calculus} \} \\
& b(F, x) = b(F, y)
\end{aligned}$$

(End of property and proof)

This property permits the introduction of a function g satisfying

$$g([x]) = b(F, x) \qquad \text{for all } x : x \in B^* .$$

We shall show that g is an isomorphism mapping N into $R\,M$. Since g renames the states of N into states of $R\,M$, we have to show that it renames the initial and final states of N into the initial and final states of $R\,M$ respectively, and that it preserves the state transition function.

Property 3.5

$$g([\varepsilon]) = F$$

Proof

$$\begin{aligned} & g([\varepsilon]) \\ = & \quad \{ \text{def. of } g \} \\ & b(F, \varepsilon) \\ = & \quad \{ \text{def. of closure} \} \\ & F \end{aligned}$$

(End of property and proof)

Property 3.6

For all $x : x \in B^*$, we have

$$[x] \in G \equiv g([x]) \in R\ q0 .$$

Proof

$$\begin{aligned} & [x] \in G \\ \equiv & \quad \{ \text{def. of } G \} \\ & x \in \underline{t}(R(T\,M)) \\ \equiv & \quad \{ \text{property 3.1} \} \\ & (R\ x) \in \underline{t}(T\,M) \\ \equiv & \quad \{ \text{def. of } T\,M \} \\ & d(q0, R\ x) \in F \\ \equiv & \quad \{ \text{property 3.2} \} \\ & q0 \in b(F, x) \\ \equiv & \quad \{ \text{def. of } R\ q0 \} \\ & b(F, x) \in R\ q0 \end{aligned}$$

$\equiv$ { def. of g }
$g([x]) \in R\ q0$
(End of property and proof)

Property 3.7

For all $x, a : x \in B^* \wedge a \in B$, we have
$g(c([x], a)) = b(g([x]), a)$.

Proof

$g(c([x], a))$
$=$ { def. of c }
$g([x\ a])$
$=$ { def. of g }
$b(F, x\ a)$
$=$ { def. of closure }
$b(b(F, x), a)$
$=$ { def. of g }
$b(g([x]), a)$
(End of property and proof)

Property 3.8

N and R M are isomorphic.

Proof

From the definition it follows that g maps $B^*/(E\ (R\ (T\ M)))$ onto $R\ Q$.
From property 3.4 it follows that g is a one-to-one mapping.
From property 3.5 it follows that g maps $[\varepsilon]$ onto F.
From property 3.6 it follows that g maps G onto $R\ q0$.
From property 3.7 it follows that g preserves the state transition function.
(End of property and proof)

This property concludes our proof obligations and, hence, this chapter.

4. VLSI design

4.0. Introduction

VLSI is a technique of constructing semiconductor chips containing a large number of active, electronic elements. Since these elements operate concurrently, VLSI nicely matches the concurrency suggested by the program notation introduced in chapter two. A chip consists of pads, transistors, and wires connecting them. The pads are points via which the interaction with the circuit's environment takes place, the wires propagate electrical signals, and the transistors either block or propagate electrical signals. The pads, transistors, and wires are laid out in the plane. Designing a VLSI circuit thus consists of designing both an interconnection scheme, which is called a schematic, and a layout thereof.

Over the years, integrated circuits have become smaller in size. It is interesting to study a circuit's operation when it is scaled down. The time required for electrical signals to propagate via wires is controlled by two distinct processes. One process is drift in an electric field. Drift is the net effect that an electric field has on the thermal motion of charge carriers. The other process is called diffusion. Diffusion determines the rate at which the voltage driven onto a wire at one point equalizes across the length of the wire; it is controlled by the product of the resistivity of the wire and its parasitic capacitance. In short wires the delay induced by drift dominates and in long wires diffusion dominates.

Scaling down the size of a VLSI circuit affects the delay of wires of different length in a different way. This can be seen as follows. As in [18], we examine

the effects of scaling down all dimensions by dividing them by a factor α. In order to keep all electric fields constant, the voltage is likewise divided by α.
The delay in a short wire is determined by the drift of the charge carriers in an electric field. For a wire of length l, and electric field E, the delay is proportional to

$$l / E .$$

When scaling down, l is divided by α, and E is not affected. Hence, the delay of a short wire is divided by α. An example of such a wire is the channel of an MOS transistor. The transit time thereof is, therefore, also divided by α.
The delay in a long wire is determined by diffusion. For a wire of length l, with resistance R and capacitance C per unit length, the delay is proportional to

$$R * C * l^2 .$$

When scaling down, R is multiplied by α^2, C is not affected, and l is divided by α. Consequently, the delay is unaffected. We may also say that the delay in long wires increases relative to the delay in short wires. If a circuit's operation depends on those relative delays, the scaled-down circuit may not function anymore.

From the above we conclude that it is attractive to design so-called delay-insensitive circuits, i.e. circuits whose correct operation does not depend on any relation between delays, not even between delays in wires of equal length or between delays of successive transmissions via one wire. Although for a schematic various layouts may exist that differ in relative wire lengths, these differences are irrelevant in the case of delay-insensitive circuits: delay-insensitive circuits enable the separation of the two design tasks. A consequence is that scaling down does not affect

the correct functioning of a chip.

Due to the possibility of separating the two design tasks, it will be simpler to design delay-insensitive circuits than other circuits. Since there are no assumptions on relative speeds, all synchronization must be done explicitly, instead of implicitly by some clocking mechanism. This leads to a reduction of proof obligations and to a clearer demarcation of layout freedom.

However attractive delay-insensitive circuits may be, there is a fundamental problem : it is not known how to construct purely delay-insensitive circuits from wires and transistors alone. It may well be impossible. In [25], C. L. Seitz proposes the following method of circumventing this problem. The chip is divided into so-called isochronic or equipotential regions. The regions are so small that, within a region, the diffusion delay in wires is negligible compared to the delay in transistors. Exploiting this knowledge about relative delays one may construct a number of small building blocks. These building blocks are interconnected by wires with delays about which no assumptions are made. The circuits thus obtained are still called delay-insensitive circuits. Since there is no known fixed set of building blocks from which a sufficiently large class of circuits can be constructed, there is a tendency to introduce new building blocks whenever convenient. Since there is no known design method this is attended with a tendency to choose large isochronic regions covering these building blocks. Scaling down the size of a chip has repercussions on the maximum number of transistors per isochronic region. As we have seen before, dividing all dimensions by α also divides the transit time of transistors by α. Since diffusion delay in a wire of length l is proportional to

$R * C * l^2$

and R is multiplied by α^2, the length of a wire, whose diffusion delay equals the transit time, is divided by $\alpha^{3/2}$. Hence, the area of an isochronic region is divided by α^3. Since the area of all transistors is divided by α^2, the maximum number of transistors per isochronic region is divided by α. For that reason we would prefer to choose the isochronic regions as small as possible. In this monograph, however, we shall make a compromise and choose the isochronic regions "reasonably small". Their sizes and contents will depend on the program being implemented.

As we have seen in the previous chapter, it is possible to derive from a program, written in the notation of chapter two, a deterministic finite state machine whose trace structure equals the program's trace structure. Hence, if we have a method of implementing such machines then we also have a method of implementing programs. Such an implementation has the attractive property that it is, potentially, extremely fast. From the "current state" and the "current symbol" the "new state" follows without any computations, merely by table-lookup, so to say. Another attractive property of such an implementation is the absence of any problem related to unbounded chatter (cf. example 2.11). Unbounded chatter manifests itself in the finite state machines constructed in section 3.0 as a cyclic path of ε-transitions. Since all ε-transitions can easily be removed, unbounded chatter is easily removed. The reverse of the medal is the potentially gigantic size of an implementation. The chip area required to implement a finite state machine is, in the worst case, proportional to the number of states of the trace structure [7]. The number of states of the blend of

some trace structures is, in the worst case, the product of the numbers of states of the composing trace structures. Hence, the chip area grows exponentially with the number of components in the program, which is not very attractive. We, therefore, pursue the following strategy: a program gives rise to a finite state machine per component, plus interconnections between these machines by which they communicate. The layout of a circuit implementing a program then consists of two parts: one part for the finite state machines which exhibits linear growth with the number of components, and one part for the interconnections which is estimated to exhibit quadratic growth, in the worst case. With this implementation method the reverse of the medal is, of course, that a single external communication may lead to a lot of internal communication: in the case of unbounded chatter even to an unbounded amount of internal communication. In the sequel we shall neglect the problem of unbounded chatter by and large, and we shall introduce concurrency to reduce the time required by internal communications. The isochronic regions referred to above, will be chosen such that each circuit implementing a finite state machine is embedded within an isochronic region. The circuits and the communications between them will be such that they yield a delay-insensitive circuit.

We have defined the trace structures described by programs, using blending, a composition operation based on the simultaneous participation of two communicating mechanisms. Such a form of communication cannot be implemented directly by physical mechanisms since there is always a delay between sending and reception. In the case of electrical circuits this means that voltage transitions propagate at some finite speed along a wire.

If we strive for delay-insensitive circuits we may not assume that all voltage transitions propagate at equal speed. As a consequence, we must see to it that at most one voltage transition is under way per wire in order to prevent a voltage transition from interfering with another one propagating along the same wire. Interference might lead to the absorption of one or both transitions or to the introduction of new ones. Absorption of transitions equals an infinite delay and may cause a grinding halt, whereas extra transitions may cause malfunctioning. In the sequel we shall refer to the bounding of the number of transitions on a wire as the exclusion of transmission interference. When constructing electrical circuits we shall see to it that transmission interference is excluded. We may then assume that a voltage transition, transmitted by one circuit, will eventually arrive at another circuit. If, again, we make no assumptions on the speed of circuits and transmissions, a voltage transitions may arrive at a circuit before that circuit is ready to receive it. We shall see to it that such a premature input signal does not interfere with the computation that goes on before the circuit is ready for the signal's reception. In the sequel we shall refer to this as the exclusion of computation interference.

As we have seen above, the design of a VLSI circuit consists of the design of a schematic and the design of a layout. We have stated that in the case of delay-insensitive circuits the two design tasks may be separated, with the exception of some restrictions on the layout within isochronic regions. In this monograph we only discuss the design of schematics and the isochronic regions required for their correctness, and we shall ignore the layout problem. Since the schematics will prescribe delay-insensitive circuits, any

layout satisfying the constraints of the isochronic regions will lead to a correctly functioning chip. Since we do not intend to go into the aspects of VLSI device physics here, we boldly state that any schematic, that is guaranteed to exclude both transmission and computation interference without making assumptions on delays (with the exception of clearly demarcated isochronic regions), prescribes a delay-insensitive circuit. By doing so we do not intend to suggest that the problem of making a layout is, in any way, a trivial problem. Part of designing a layout is the placement of the pads and transistors, and the routing of the wires. Usually the size of the pads is fixed, but the size of the transistors and the width of the wires depends on the layout and, hence, determining those sizes and widths is part of the layout design task. Also the number of amplifiers ("relays") inserted in wires depends on the layout. This number depends, for example, on the wire's length. Insertion of amplifiers in long wires or, even better, in front of long wires [19], does not only reduce the delay in such a wire, it also reduces the noise sensitivity. The latter can be seen as follows. If a voltage is driven onto a wire at one point, the voltage will, due to the diffusion process, change only slowly on a distant point on that wire. Such a voltage might be used to control the switching of a transistor between the conducting and the blocking state. Since a transistor is an analogue device, the state is not a discrete but a continuous, though steep, function of the control voltage. If that voltage changes too slowly it may stay too long in the steep region, and small fluctuations of the voltage, as in the case of noise, may then cause a transistor to switch to and fro between the conducting and the blocking state. This will, in general, lead to malfunctioning. Hence, if a layout specifies a long wire, amplifiers need to be inserted. Though this will change the

wire's delay, it does not affect the correct functioning of the circuit.

We have thus extended the layout task slightly beyond what is customary. We have done so deliberately since we have now succeeded in separating the digital and analogue concerns. Such a separation is vital in reducing the complexity of any VLSI design. As a result, the communications between isochronic regions may be thought of as discrete signals.

We have proposed exclusion of transmission and computation interference as a means of realizing delay-insensitive circuits. Delay-insensitive circuits are (too) little known in the literature. They are sometimes referred to by one of the following adjectives, although each of these has also been used for a wider class of circuits : self-timed, delay-insensitive, speed-independent, and asynchronous.

4.1. Directed trace structures

In order to reason about mechanisms whose communications take a nonnegligible amount of time, we first expand our trace theory. We do so by introducing a third composition function, called agglutination. Agglutination is similar to blending, but it expresses unbounded, finite delay and overtaking. (Overtaking is the phenomenon that order in time between signals is not preserved by their transmission; it is a direct consequence of the independence of delays.) There is a direction in delay: a signal's sending precedes its reception. To express such an asymmetry, a trace structure's alphabet is partitioned into an input and an output alphabet, both of which may be empty. The result is called a directed trace structure. A directed trace structure T is a triple

$$< \underline{t}T,\ \underline{i}T,\ \underline{o}T >$$

where

$\underline{i}T$ is the input alphabet,

$\underline{o}T$ is the output alphabet, $\underline{i}T$ and $\underline{o}T$ are disjoint, and

$\underline{t}T$ is the trace set, $\underline{t}T \subseteq (\underline{i}T \cup \underline{o}T)^*$.

We shall use $\underline{a}T$ to denote $\underline{i}T \cup \underline{o}T$. Symbols in $\underline{i}T$ are called input symbols, or symbols of type input. Symbols in $\underline{o}T$ are called output symbols, or symbols of type output. When appropriate, we refer to the trace structures defined in chapter one as undirected trace structures. If T is a directed trace structure then $< \underline{t}T, \underline{a}T >$ is an undirected trace structure; it will be denoted by $\underline{u}T$. Sometimes we shall not explicitly distinguish between directed and undirected trace structures.

Two directed trace structures, T and U say, will be composed only (using the blending or agglutination operation) when both $\underline{i}T \cap \underline{i}U$ and $\underline{o}T \cap \underline{o}U$ are empty, i.e. for each symbol they have in common, its type in T differs

from its type in U. The blend of T and U, denoted by $T \underline{b}\, U$, is the directed trace structure

$$\begin{aligned} &< \underline{t}(\underline{u}T \ \underline{b} \ \underline{u}U) \\ &, (\underline{i}T \setminus \underline{o}U) \cup (\underline{i}U \setminus \underline{o}T) \\ &, (\underline{o}T \setminus \underline{i}U) \cup (\underline{o}U \setminus \underline{i}T) \\ &> . \end{aligned}$$

Notice that

$$(\underline{i}T \setminus \underline{o}U) \cup (\underline{i}U \setminus \underline{o}T) = (\underline{i}T \cup \underline{i}U) \setminus (\underline{o}T \cup \underline{o}U)$$

since $\underline{i}T \cap \underline{o}T = \{\} = \underline{i}U \cap \underline{o}U$.

Property 4.0

For all directed trace structures T and U, such that $\underline{i}T \cap \underline{i}U = \{\} = \underline{o}T \cap \underline{o}U$, we have

$$\underline{u}(T \ \underline{b} \ U) = \underline{u}T \ \underline{b} \ \underline{u}U .$$

(End of property)

In order to distinguish between symbol b as an output symbol and symbol b as an input symbol we may postfix the symbol with ! or ? , yielding b! or b? respectively. The symbols b, b! , and b? are three distinct symbols. We shall use ! for postfixing output symbols and ? for postfixing input symbols.

Let $A0$ and $A1$ be disjoint alphabets. If B is an alphabet then $B?!(A0, A1)$ is the alphabet obtained from B by replacing each symbol b, $b \in B \cap A0$, by the postfixed symbol $b?$, and each symbol b, $b \in B \cap A1$, by the postfixed symbol $b!$. Trace set or trace structure $T?!(A0, A1)$ is similarly obtained from trace set or trace structure T.

If A is an alphabet then the directed trace structure $\underline{G}(A)$ is defined such that

$$G(A) = \underline{B}(b: b \in A : < \underline{t}\, DEL(b!, b?), \{b!\}, \{b?\} >)$$

where $\underline{B}$ denotes continued blending (cf. page 29) . Notice

that this blending operation is applied to trace structures that have no symbols in common and, hence, amounts to interleaving. In the sequel $G(A)$ will, from a mechanistic point of view, correspond to the communication channel along which the symbols from A are transmitted. It is, therefore, not amazing that for all b, $b \in A$, $b!$ is an input and $b?$ an output symbol of this trace structure.

Using the above definitions we may now introduce the agglutinate of two directed trace structures, T and U say, that satisfy $\underline{i}T \cap \underline{i}U = \{\} = \underline{o}T \cap \underline{o}U$. (We shall not repeat this restriction anymore; it is assumed to be satisfied for all directed trace structures being composed.) The agglutinate of T and U, denoted by $T \underline{g} U$, is the directed trace structure

$$T?!(\underline{i}T \cap \underline{o}U,\ \underline{o}T \cap \underline{i}U)\ \underline{b}\ G(\underline{g}T \cap \underline{g}U)\ \underline{b}\ U?!(\underline{i}U \cap \underline{o}T,\ \underline{o}U \cap \underline{i}T)\ .$$

The alphabets of T and U are made disjoint by post-fixing all common symbols. They are then connected by $G(\underline{g}T \cap \underline{g}U)$, the glue, so to say. In our mechanistic appreciation, the trace structures $DEL(b!, b?)$ constituting the glue, express the delay between the sending $b!$ of symbol b and the reception $b?$ of b, i.e. they form the synchronization mechanisms of unbounded slack controlling the communications. Remember that DEL is asymmetric: it does not bound the lead of $b!$ over $b?$, whereas the lead of $b?$ over $b!$ is at most 0. The disjointness of the alphabets of the various DELs constituting the glue expresses the independence of the various delays, thereby admitting the possibility of overtaking.

Obviously, all essential properties of blending, as discussed in chapter one, are retained in the case of directed trace structures. In general, agglutination has

similar properties. We mention that the agglutinate of prefix-closed trace structures is prefix-closed. Analogous to blending, we shall agglutinate a number of trace structures only if each symbol occurs in at most two of them. Under this restriction, agglutination is associative.

Example 4.1

Let T and U be directed trace structures, such that

$T = \langle \{a\,b\,c\},\ \{a\},\ \{b, c\} \rangle$ and

$U = \langle \{d\,b\,e\},\ \{b, d\},\ \{e\} \rangle$.

We then have

$T \underline{b}\ U = \langle \{adce, adec, dace, daec\},\ \{a, d\},\ \{c, e\} \rangle$,

$T \underline{g}\ U = \langle \{adce, adec, dace, daec, acde\},\ \{a,d\},\ \{c,e\} \rangle$.

Notice that, due to the asymmetry, deac is not a trace of the agglutinate.

(End of example)

Example 4.2

In this example SYNC and DEL stand for directed trace structures. The left argument is then an input symbol and the right argument is an output symbol. For three distinct symbols a, b, and c we have

$(1, 0)\ SYNC(a, b)\ \underline{g}\ (1, 0)\ SYNC(b, c)$

$=$ { def. of $\underline{g}$ and G }

$(1, 0)\ SYNC(a, b!)\ \underline{b}\ DEL(b!, b?)\ \underline{b}\ (1, 0)\ SYNC(b?, c)$

$=$ { property 1.42 }

$DEL(a, c)$

and

$(1, 0)\ SYNC(a, b)\ \underline{b}\ (1, 0)\ SYNC(b, c)$

$=$ { property 1.40 }

$(2, 0)\ SYNC(a, c)$.

(End of example)

Agglutination is a composition function expressing synchronization with unbounded slack. A special case

thereof is synchronization with slack 0. The latter corresponds to blending. The following property is, therefore, not entirely unexpected.

Property 4.3

For all directed trace structures T and U we have

$$T \underline{b} U \subseteq T \underline{g} U .$$

Proof

Let T' and U' be such that

$$T' = T ?! (\underline{i}T \cap \underline{o}U , \ \underline{o}T \cap \underline{i}U) \quad \text{and}$$
$$U' = U ?! (\underline{i}U \cap \underline{o}T , \ \underline{o}U \cap \underline{i}T) .$$

For all x, such that $x \in (\underline{a}T \div \underline{a}U)^*$, we have

$$\begin{aligned}
& x \in \underline{t}(T \underline{b} U) \\
\equiv & \quad \{ \text{def. of } \underline{b} \} \\
& \underline{E}(y : y \in \underline{t}(\underline{u}T \ \underline{w} \ \underline{u}U) : y \restriction (\underline{a}T \div \underline{a}U) = x) \\
\equiv & \quad \{ \text{def. of } \underline{w} \} \\
& \underline{E}(y : y \in (\underline{a}T \cup \underline{a}U)^* \wedge y \restriction \underline{a}T \in \underline{t}\underline{u}T \wedge y \restriction \underline{a}U \in \underline{t}\underline{u}U : y \restriction (\underline{a}T \div \underline{a}U) = x) \\
\Rrightarrow & \quad \{ \text{replace each occurrence of } b,\ b \in \underline{a}T \cap \underline{a}U, \text{ in } y \text{ by } b!\, b? \} \\
& \underline{E}(y : y \in (\underline{a}T' \cup \underline{a}U')^* \wedge y \restriction \underline{a}T' \in \underline{t}T' \wedge y \restriction \underline{a}U' \in \underline{t}U' \\
& \qquad \wedge \underline{A}(b : b \in \underline{a}T \cap \underline{a}U : y \restriction \{b!, b?\} \in \underline{t}\, DEL(b!, b?)) \\
& \qquad : y \restriction (\underline{a}T \div \underline{a}U) = x \\
& \qquad) \\
\equiv & \quad \{ \text{def. of } G \} \\
& \underline{E}(y : y \in (\underline{a}T' \cup \underline{a}U')^* \wedge y \restriction \underline{a}T' \in \underline{t}T' \wedge y \restriction \underline{a}U' \in \underline{t}U' \\
& \qquad \wedge y \restriction (\underline{a}T \cap \underline{a}U) \in \underline{t}\, G(\underline{a}T \cap \underline{a}U) \\
& \qquad : y \restriction (\underline{a}T \div \underline{a}U) = x \\
& \qquad) \\
\equiv & \quad \{ \text{def. of } \underline{g} \} \\
& x \in \underline{t}(T \underline{g} U) .
\end{aligned}$$

(End of property and proof)

Example 4.2 shows that the class of regular trace structures is not closed under agglutination. This should not bother us now. In the next two sections we shall develop restrictions that, en passant, imply regularity.

4.2. Transmission interference

In the introduction of this chapter we have discussed both transmission and computation interference. In this section we investigate compositions of trace structures that are free from transmission interference. Computation interference is not discussed any further in this chapter; it will only be encountered when dealing with electrical circuits in the next chapter.

A well-known technique of excluding transmission interference is called hand-shaking. With this technique a transmission via one wire is acknowledged by a transmission via another wire in the opposite direction. These acknowledgements need not pair individual wires; it is more common to have the transmission via one wire from a set of wires acknowledged by the transmission via one wire from another set. Such sets of wires are called ports. In the case of two communicating components, both having one input and one output port, where the input port of each component is connected with the other component's output port, transmission interference along those connections is excluded by letting the transmissions via one connection and via the other alternate. Observe the asymmetry between the two components caused by the fact that only one of them is allowed to do the first transmission. In the case of more ports various hand-shaking protocols may be envisaged, of which we shall discuss one. We do so in our terminology of (directed) trace structures.

We define an input port to be a nonempty set of input symbols; an output port is a nonempty set of output symbols. We shall assume of directed trace structures that their alphabets be partitioned into disjoint input and output ports. Henceforth, we also make the important assumption

that the ports are the finest grain of interconnection, i.e. if two trace structures are composed they have for every port either all or none of the port's symbols in common. A port of the composite is also a port of one of the composing trace structures. Consequently, ports are neither "split" nor "merged". We have, in fact, defined a new kind of trace structure. It is, however, not worth while to formalize it as another tuple with various properties.

Let T and U be two directed trace structures. Then trace structure TU,

$$TU = \underline{\cup} T?!(\underline{\cdot} T \cap \underline{\circ} U,\ \underline{\circ} T \cap \underline{\cdot} U)\ \underline{w}\ \underline{\cup} U?!(\underline{\cdot} U \cap \underline{\circ} T,\ \underline{\circ} U \cap \underline{\cdot} T),$$

corresponds to the "independent, unsynchronized operation" of T and U. Notice that the weaving operation is applied to trace structures with disjoint alphabets. Trace structure TGU,

$$TGU = TU\ \underline{w}\ \underline{\cup} G(\underline{\circ} T \cap \underline{\circ} U),$$

corresponds to the agglutinate of T and U in which their internal communications are retained. We have

$$TGU \upharpoonright (\underline{\circ} T \div \underline{\circ} U) = \underline{\cup}(T\ \underline{g}\ U).$$

The composition of trace structures T and U is defined to be free from transmission interference if for every port P they have in common

$$TGU \upharpoonright (P! \cup P?) \subseteq (1, 0)\,SYNC(P!, P?)$$

where

$$(k, l)\,SYNC(B, C) =$$
$$\langle \{x: x \in (B \cup C)^* \wedge \underline{A}(y, z: x = yz: -l \le y\underline{N}B - y\underline{N}C \le k): x\},\ B \cup C \rangle,$$
$$y\underline{N}B = \underline{S}(b: b \in B: y\underline{N}b), \quad \text{and}$$

$P!$ and $P?$ are obtained from P by postfixing each symbol in P with $!$ and $?$ respectively.

The above is generalized to the case of more trace structures by replacing TGU by the weave of all these trace structures and their glue (with postfixed symbols), and by letting P range over all ports in common to two of the trace structures.

Example 4.4

Let T and U be such that

$T = < \underline{t}\, TR((a; c; d; b)^*), \{a, d\}, \{b, c\} >$,

$U = < \underline{t}\, TR((c; e; f; d)^*), \{c, f\}, \{d, e\} >$,

and each symbol constitutes a port. Following the above nomenclature we have

$$TGU = TR((a; c!; d?; b)^*) \; \underline{w}\; TR((c?; e; f; d!)^*) \; \underline{w}\; DEL(c!, c?) \; \underline{w}\; DEL(d!, d?)$$
$$= TR((a; c!; c?; e; f; d!; d?; b)^*)$$

and

$$TGU \upharpoonright \{c!, c?\} = TR((c!; c?)^*) = (1, 0)\, SYNC(c!, c?)$$
$$TGU \upharpoonright \{d!, d?\} = TR((d!; d?)^*) = (1, 0)\, SYNC(d!, d?)$$

from which we conclude that the composition of T and U is free from transmission interference.
(End of example)

If a composition is free from transmission interference then their "glue" is used in a limited way, i.e. for each symbol b common to two of the trace structures, only a subset of $DEL(b!, b?)$ is used, viz. a subset of $(1, 0)\, SYNC(b!, b?)$. We may, therefore, replace $DEL(b!, b?)$ by $(1, 0)\, SYNC(b!, b?)$ without affecting the agglutinate. Since $(1, 0)\, SYNC(b!, b?)$ is regular, this implies that the agglutinate of regular trace structures, whose composition is free from transmission interference, is regular.

We define a signalling set to be a set of ports, containing at least one input and one output port. We shall assume of trace structures that their set of ports be partitioned into disjoint signalling sets. If V is a signalling set and s is a sequence of symbols, then s is called a signalling sequence w.r.t. V if s contains exactly one symbol of every port in V and no other symbols, and the first

symbol and the last symbol are of different type. A signalling sequence is called active if its first symbol is of type output, and passive otherwise. Next we extend these definitions to trace structures: a trace structure T is called passive w.r.t. signalling set V if every trace in T↑V is a prefix of a concatenation of passive signalling sequences w.r.t. V ; active is defined similarly. Notice that in the case of a signalling set with two ports we obtain the form of hand-shaking informally discussed above. The asymmetry between the hand-shaking components is expressed by their being active or passive. The notion of signalling sets was, in a restricted form, first introduced in [28].

Example 4.5

Consider the full adder of example 2.5 and partition its alphabet into three input ports, viz.

{a0, a1} , {b0, b1}, {c0, c1} ,

and two output ports, viz.

{d0, d1}, {e0, e1},

and let these five ports form one signalling set. Each selection of the repetitive command yields a passive signalling sequence. Hence, the component is passive w.r.t. this signalling set.
(End of example)

Inspired by our informal discussion of hand-shaking we may expect that the composition of two trace structures is free from transmission interference if their common ports form a signalling set and one of them is active and the other passive w.r.t. this signalling set. This is indeed the case as will be shown below. This class of compositions may be generalized as follows. Imagine that the passive component is "split" into two passive components, and that it can be shown that the agglutinate of one of them and

the active component looks like a single active component, and that the latter composition is free from transmission interference. We may then conclude that the composition of the three trace structures is also free from transmission interference. Hence, any composition of passive trace structures with an active one is free from transmission interference if the passive trace structures can be ordered to admit mathematical induction. Although matters are not that simple, this is the quintessence of the class CL compositions discussed below. All compositions that we consider in the sequel are of this class. For an informal discussion of class CL compositions we refer to [25].

Let T and S_i, $0 \leq i < n$, be $n+1$ directed trace structures, $n \geq 1$. We say that the composition of these $n+1$ trace structures is of class CL if the following conditions are satisfied:

- the n trace structures S_i are such that their connections form no cyclic paths (there is a connection from S_i to S_j if an output port of S_i is an input port of S_j);
- for each S_i the ports that are also a port of T or of some S_j form a signalling set, and S_i is passive w.r.t. it;
- the ports of T that are also a port of some S_i form a signalling set, and T is active w.r.t. it.

Property 4.6

A composition of class CL is free from transmission interference.

Proof

We first present some notions that are instrumental in this proof. Let t be a trace (t is fixed throughout this paragraph), x a set of ports, and i a natural number. Then $min_i(x)$ denotes the minimum length of any prefix of t that contains i occurrences of symbols of some port of x,

$max_i(x)$ denotes the minimum length of any prefix of t that contains i occurrences of symbols of each port of x, i.e.

$$min_i(x) = \underline{MIN}(r,s: t=rs \wedge \underline{E}(P: P \in x: |r \restriction P| = i) : |r|),$$
$$max_i(x) = \underline{MIN}(r,s: t=rs \wedge \underline{A}(P: P \in x: |r \restriction P| = i) : |r|),$$

where $|r|$ denotes the length of trace r. For some values of i and x, $min_i(x)$ and $max_i(x)$ are defined as the minimum length of any trace in the empty set. We stipulate that this is a single, large number, exceeding all other numbers in question. We shall use the following properties (of which we do not supply a proof):

for all sets of ports x, y, and z, and all natural i,

(0) $min_i(x) \leq min_i(y) \Rightarrow min_i(x \cup z) \leq min_i(y \cup z)$,

(1) $max_i(x) \leq max_i(y) \Rightarrow max_i(x \cup z) \leq max_i(y \cup z)$,

(2) $min_i(x \cup y) \leq min_i(x)$,

(3) $max_i(x) \leq max_i(x \cup y)$.

Notice that inequalities like these are invariant under weaving, i.e. if they hold for trace t then they also hold for any trace u satisfying $u \restriction A = t$ for some alphabet A that contains all symbols from ports occurring in the inequalities.

The notions min and max derive their significance from the fact that, for any port P and trace t, the condition

$$t \restriction \{P! \cup P?\} \in \underline{t}((1,0)\ SYNC\ (P!, P?)),$$

which expresses absence of transmission interference along port P, is equivalent to

$$max_i(\{P?\}) \leq min_{i+1}(\{P!\}) \quad \text{for all natural } i.$$

If x is a set of ports and t a trace of $G(x)$, we have, for all natural i,

(+) $min_i(x!) \leq min_i(x?) \wedge max_i(x!) \leq max_i(x?)$.

In order to show the absence of transmission interference on all ports of x, it then suffices, on account of (2) and (3), to show, for all natural i,

$max_i(x?) \leq min_{i+1}(x!)$.

Let x and y be nonempty sets of input and output ports respectively, and let t be a signalling sequence that is passive w.r.t. $x? \cup y!$. From the definition it follows that of each port in $x? \cup y!$ one symbol occurs in t, that the first symbol of t is an input symbol, hence

$min_1(x?) \leq min_1(y!)$,

and that the last symbol of t is an output symbol, hence

$max_1(x?) \leq max_1(y!)$.

If t is a concatenation of two passive signalling sequences then the last symbol of the first sequence precedes the first symbol of the last sequence, hence

$max_1(y!) \leq min_2(x?)$.

If t is a prefix of a concatenation of passive signalling sequences, the above generalizes to

(*) $min_i(x?) \leq min_i(y!) \wedge max_i(x?) \leq max_i(y!) \leq min_{i+1}(x?)$

for all natural i .

We shall now prove property 4.6 for the case $n = 2$. Let T, U, and V be three trace structures, and v, w, x, y, and z five disjoint sets of ports, such that

$\underline{i}T \subseteq x \cup z$, $\underline{o}T \subseteq v \cup y$,

$\underline{i}U \subseteq v$, $\underline{o}U \subseteq w \cup x$,

$\underline{i}V \subseteq w \cup y$, $\underline{o}V \subseteq z$,

$\underline{o}T \cap \underline{o}U = v \cup x$,

$\underline{o}U \cap \underline{o}V = w$,

$\underline{o}V \cap \underline{o}T = y \cup z$,

T is active w.r.t. $v \cup x \cup y \cup z$,

U is passive w.r.t. $v \cup w \cup x$,

V is passive w.r.t. $w \cup y \cup z$.

From $\underline{o}U \cap \underline{o}V = w$ we may conclude that the connections of U and V form no cyclic paths. We obtain a composition of class CL if we choose $S_0 = U$ and $S_1 = V$.

By appropriate substitutions in (+) and (*) we may conclude that for each trace t of

$$T?!(x \cup z,\ v \cup y) \ \underline{w}\ U?!(v,\ w \cup x) \ \underline{w}\ V?!(w \cup y,\ z) \ \underline{w}\ G(v \cup w \cup x \cup y \cup z)$$

the following conditions hold for all natural i :

(T) $\max_i(x? \cup z?) \leq \min_{i+1}(v! \cup y!)$,

(U) $\min_i(v?) \leq \min_i(w! \cup x!) \wedge \max_i(v?) \leq \max_i(w! \cup x!)$,

(V) $\min_i(w? \cup y?) \leq \min_i(z!) \wedge \max_i(w? \cup y?) \leq \max_i(z!)$,

(G) $\min_i(u!) \leq \min_i(u?) \wedge \max_i(u!) \leq \max_i(u?)$
for all u : $u \subseteq v \cup w \cup x \cup y \cup z$.

Observe that we have taken only the last inequality of (*) in (T) and that we have omitted that inequality in (U) and (V). We have no use for the additional terms.
First we derive for all natural i

$\max_i(v? \cup y?)$ (M0)
$\leq$ { (U), (1) }
$\max_i(w! \cup x! \cup y?)$ (M1)
$\leq$ { (G), (1) }
$\max_i(w? \cup x! \cup y?)$ (M2)
$\leq$ { (V), (1) }
$\max_i(x! \cup z!)$ (M3)
$\leq$ { (G) }
$\max_i(x? \cup z?)$ (M4)
$\leq$ { (T) }
$\min_{i+1}(v! \cup y!)$ (M5)
$\leq$ { (G) }
$\min_{i+1}(v? \cup y?)$ (M6)
$\leq$ { (U), (0) }
$\min_{i+1}(w! \cup x! \cup y?)$ (M7)
$\leq$ { (G), (2), (0) }
$\min_{i+1}(w? \cup y?)$ (M8)
$\leq$ { (V) }
$\min_{i+1}(z!)$ (M9) .

Using (2) and (3) we conclude the absence of transmission interference on all ports of

v from (M0) and (M5) ,
w from (M2) and (M7) ,
x from (M4) and (M7) ,
y from (M0) and (M5) ,
z from (M4) and (M9) .

As a consequence the composition of T, U, and V is free from transmission interference. Since the above holds if $v = w = x = \{\}$, this property also holds for the case $n = 1$. In order to show that it holds for the case $n > 2$ as well we show that, independent of V, $T \mathbin{⨾} U$ satisfies a property similar to the only property of T that we have used in the above calculations, viz. (T). Since the arrangement of the passive trace structures is acyclic, property 4.6 then follows by mathematical induction. The set of ports that $T \mathbin{⨾} U$ has in common with V being $w \cup y \cup z$, w,y output and z input, we show for all natural i

$$
\begin{array}{ll}
 & \max_i(z?) \\
\leq & \{\ (3)\ \} \\
 & \max_i(x? \cup z?) \\
\leq & \{\ (T)\ \} \\
 & \min_{i+1}(v! \cup y!) \\
\leq & \{\ (G),\ (0)\ \} \\
 & \min_{i+1}(v? \cup y!) \\
\leq & \{\ (U),\ (0)\ \} \\
 & \min_{i+1}(w! \cup x! \cup y!) \\
\leq & \{\ (2)\ \} \\
 & \min_{i+1}(w! \cup y!) \quad ,
\end{array}
$$

which concludes our proof obligations.
(End of property and proof)

4.3. Slack-independence

We have defined the trace structures described by programs using blending, a composition function that corresponds to a synchronization mechanism with slack zero. This makes blending attractive from a programmer's point of view, and hard to implement. Agglutination has been introduced as a composition function that recognizes delay problems. Agglutination, especially of trace structures whose composition is free from transmission interference, seems to be simpler to implement than blending. It is, therefore, attractive to strive for compositions of trace structures whose blend and agglutinate are equal: programs defined with blending may then be implemented with agglutination without affecting the result. The composition of a number of trace structures whose blend and agglutinate are equal is called slack-independent.

Example 4.7

Let T, U, and V be such that

$T = \langle \{a\ b\ c\ f\},\ \{a, f\},\ \{b, c\} \rangle$,

$U = \langle \{d\ b\ e\ f\},\ \{b, d\},\ \{e, f\} \rangle$, and

$V = \langle \{d\ c\},\ \{c\},\ \{d\} \rangle$.

We then have (cf. example 4.1)

$T \underline{b} U = \langle \{adce, adec, dace, daec\},\ \{a, d\},\ \{c, e\} \rangle$,

$T \underline{g} U = \langle \{adce, adec, dace, daec, acde\},\ \{a,d\},\ \{c,e\} \rangle$,

$T \underline{b} U \underline{b} V = \langle \{a\ e\},\ \{a\},\ \{e\} \rangle$, and

$T \underline{g} U \underline{g} V = \langle \{a\ e\},\ \{a\},\ \{e\} \rangle$.

Observe that

$T \underline{b} U \neq T \underline{g} U$ and $T \underline{b} U \underline{b} V = T \underline{g} U \underline{g} V$

and

$T \underline{b} V = T \underline{g} V$, $V \underline{b} U = V \underline{g} U$, and $T \underline{b} U \neq T \underline{g} U$.

(End of example)

The above example shows that pairwise slack-independence is not necessary to guarantee that the composition of a number of trace structures is slack-independent. It also shows that slack-independence is not transitive. We have, however, the following property.

Property 4.8

Pairwise slack-independence implies the slack-independence of the composition of a number of directed trace structures.

Proof

Let T, U, and V be three directed trace structures satisfying

$$T \underline{b} U = T \underline{g} U \;\wedge\; U \underline{b} V = U \underline{g} V \;\wedge\; V \underline{b} T = V \underline{g} T .$$

In the following proof we frequently use the assumption $\underline{o}T \cap \underline{o}U \cap \underline{o}V = \{\}$ which guarantees the associativity of both the agglutination and the blending operation. It implies that, while agglutinating, each symbol is postfixed at most once in each trace structure. In the proof thir will be referred to as "distribution of postfixing". Using TU, $TU = T \underline{b} U$, we have

$$\begin{aligned}
& T \underline{g} U \underline{g} V \\
= \;& \{ \text{ associativity, } T \underline{g} U = T \underline{b} U \} \\
& (T \underline{b} U) \underline{g} V \\
= \;& \{ \text{ def. of } \underline{g} \} \\
& (T \underline{b} U) ?! (\underline{i}TU \cap \underline{o}V,\ \underline{o}TU \cap \underline{i}V) \\
& \underline{b}\, G(\underline{o}TU \cap \underline{o}V) \\
& \underline{b}\, V ?! (\underline{i}V \cap \underline{o}TU,\ \underline{o}V \cap \underline{i}TU) \\
= \;& \{ \text{ distribution of postfixing } \} \\
& T ?! (\underline{i}T \cap \underline{o}V,\ \underline{o}T \cap \underline{i}V) \\
& \underline{b}\, U ?! (\underline{i}U \cap \underline{o}V,\ \underline{o}U \cap \underline{i}V) \\
& \underline{b}\, V ?! (\underline{i}V \cap \underline{o}TU,\ \underline{o}V \cap \underline{i}TU) \\
& \underline{b}\, G(\underline{o}TU \cap \underline{o}V) \\
= \;& \{ \; G(A \cup B) = G(A) \;\underline{b}\; G(B) \text{ for disjoint } A \text{ and } B, \\
& \quad \underline{o}T \cap \underline{o}U \cap \underline{o}V = \{\} \; \}
\end{aligned}$$

$$
\begin{aligned}
& T \mathbin{?!} (\underline{\iota} T \cap \underline{o} V \ , \ \underline{o} T \cap \underline{\iota} V) \\
& \underline{b}\, U \mathbin{?!} (\underline{\iota} U \cap \underline{o} V \ , \ \underline{o} U \cap \underline{\iota} V) \\
& \underline{b}\, V \mathbin{?!} (\underline{\iota} V \cap \underline{o} T U \ , \ \underline{o} V \cap \underline{\iota} T U) \\
& \underline{b}\, G\, (\underline{a} T \cap \underline{a} V) \\
& \underline{b}\, G\, (\underline{a} U \cap \underline{a} V) \\
= \ & \{ \text{ distribution of postfixing } \} \\
& (T \mathbin{?!} (\underline{\iota} T \cap \underline{o} V, \ \underline{o} T \cap \underline{\iota} V) \\
& \underline{b}\ G(\underline{a} T \cap \underline{a} V) \\
& \underline{b}\, V \mathbin{?!} (\underline{\iota} V \cap \underline{o} T, \ \underline{o} V \cap \underline{\iota} T) \\
&) \mathbin{?!} (\underline{\iota} V \cap \underline{o} U, \ \underline{o} V \cap \underline{\iota} U) \\
& \underline{b}\, U \mathbin{?!} (\underline{\iota} U \cap \underline{o} V, \ \underline{o} U \cap \underline{\iota} V) \\
& \underline{b}\ G(\underline{a} U \cap \underline{a} V) \\
= \ & \{ \text{ def. of } \underline{g} \} \\
& (T \underline{g} V) \mathbin{?!} (\underline{\iota} V \cap \underline{o} U, \ \underline{o} V \cap \underline{\iota} U) \ \underline{b}\ U \mathbin{?!} (\underline{\iota} U \cap \underline{o} V, \ \underline{o} U \cap \underline{\iota} V) \ \underline{b}\ G(\underline{a} U \cap \underline{a} V) \\
= \ & \{ \ T \underline{g} V = T \underline{b} V \ \} \\
& (T \underline{b} V) \mathbin{?!} (\underline{\iota} V \cap \underline{o} U, \ \underline{o} V \cap \underline{\iota} U) \ \underline{b}\ U \mathbin{?!} (\underline{\iota} U \cap \underline{o} V, \ \underline{o} U \cap \underline{\iota} V) \ \underline{b}\ G(\underline{a} U \cap \underline{a} V) \\
= \ & \{ \text{ distribution of postfixing } \} \\
& T \underline{b}\, V \mathbin{?!} (\underline{\iota} V \cap \underline{o} U, \ \underline{o} V \cap \underline{\iota} U) \ \underline{b}\ U \mathbin{?!} (\underline{\iota} U \cap \underline{o} V, \ \underline{o} U \cap \underline{\iota} V) \ \underline{b}\ G(\underline{a} U \cap \underline{a} V) \\
= \ & \{ \text{ def. of } \underline{g} \} \\
& T \underline{b}\ (U \underline{g} V) \\
= \ & \{ \ U \underline{g} V = U \underline{b} V \ , \ \text{associativity} \ \} \\
& T \underline{b}\ U \ \underline{b}\ V \ .
\end{aligned}
$$

(End of property and proof)

Reconsider example 4.7 and let each symbol constitute a port. The ports that T and U have in common form one signalling set, viz. $\{\{b\}, \{f\}\}$, w.r.t. which T is active and U is passive. The composition of T and U is, therefore, of class CL but, as the example shows, it is not slack-independent. Compositions that are both free from transmission interference and slack-independent are, however, easily derived from arbitrary compositions by the introduction of fresh symbols.

Let T and U be two directed trace structures, and let b and c be two fresh symbols. T' and U' are two directed trace structures,

$T' = < \underline{t}T',\ \underline{i}T \cup \{b\},\ \underline{o}T \cup \{c\} >$,

$U' = < \underline{t}U',\ \underline{i}U \cup \{c\},\ \underline{o}U \cup \{b\} >$,

where $\underline{t}T'$ and $\underline{t}U'$ are obtained from $\underline{t}T$ and $\underline{t}U$ respectively, by replacing in every trace each occurrence of the symbol d by the sequence bd for all d such that $d \in \underline{o}T \cap \underline{i}U$, and by the sequence dc for all d such that $d \in \underline{i}T \cap \underline{o}U$. All common symbols of T' and U' (including b and c) are grouped together in one input and one output port. These two ports form one signalling set, viz. $\{\underline{i}T' \cap \underline{o}U', \underline{o}T' \cap \underline{i}U'\}$. The transformation of T and U is called the introduction of alternation; T' is called the passivated T, and U' is the activated U. Notice that the introduction of alternation doubles the amount of internal communication between T and U in such a way that transmissions in one direction and the other alternate.

Property 4.9

For T' and U' as defined above, the composition is free from transmission interference.

Proof

T' is passive and U' is active w.r.t. their common signalling set. Hence, the composition is of class CL and, on account of property 4.6, free from transmission interference.

(End of property and proof)

Property 4.10

For trace structures T, U, T', and U' as defined above we have

$T \underline{b} U = T' \underline{b} U' = T' \underline{g} U'$ and

$PREF(T) \underline{b} PREF(U) = PREF(T') \underline{b} PREF(U') = PREF(T') \underline{g} PREF(U')$.

Remark

The proof of this property is omitted, not on account of non-existence but because its length does not contribute to the reliability of this property.
(End of property and remark)

Notice that the introduction of alternation preserves regularity of trace structures.

From properties 4.9 and 4.10 we derive that arbitrary trace structures can be transformed in such a way that their blend is unaffected and that the composition of the transformed trace structures is slack-independent and free from transmission interference.

Example 4.11

Let T and U be as in example 4.7. Introducing alternation by fresh symbols p and q yields

$T' = \langle \{apbcfq\}, \{a, f, p\}, \{b, c, q\} \rangle$ and

$U' = \langle \{dpbefq\}, \{b, d, q\}, \{e, f, p\} \rangle$.

We then have

$$T \underline{b} U = T' \underline{b} U' = T' \underline{g} U'$$
$$= \langle \{adce, adec, dace, daec\}, \{a, d\}, \{c, e\} \rangle .$$

(End of example)

With this example we conclude our explorations of the problems associated with delays. In the next chapter we reap the fruits hereof in the form of a strategy for implementing programs as delay-insensitive VLSI circuits.

5. An implementation strategy

5.0. Global strategy

Now we turn to the implementation of programs as silicon chips. A program, as defined in chapter two, consists of a number of components, each having its own prefix-closed trace structure. We assume that each trace structure's alphabet has been partitioned into an input and an output alphabet, these two alphabets into ports, and these ports into signalling sets. Furthermore, we assume that none of the trace sets is empty. The latter assumption follows from our mechanistic appreciation which states that, initially, the trace selected thus far is the empty trace. We propose to follow the structure of a program to derive an implementation. A component consists of a command and a number of subcomponents, each having its own trace structure. If the composition of these trace structures is not of class CL or if it is slack-dependent, then all these trace structures are pairwise transformed by the introduction of alternation, and by adding all prefixes of traces in order to make the transformed trace structures prefix-closed again. According to property 4.10 the transformation does not affect the trace structure of the composite. The program thus leads to a tree of trace structures, viz. one per command, whose composition is both slack-independent (properties 4.8, 4.10) and free from transmission interference (properties 4.6, 4.9). For each of these trace structures, or commands, a circuit may be constructed. Since each of the program's constituting commands is expected to be small, each of the corresponding circuits is expected to be small. Therefore, we allow ourselves the freedom of making circuits under the assumption

that each circuit for a command is embedded in an isochronic region. If we see to it that computation interference is excluded, the tree of circuits is a delay-insensitive circuit for the complete program (cf. section 4.0). There are then, for example, no restrictions on the relative placement of these circuits and the lengths of wires that connect them.

If we want to incorporate the above implementation strategy in an algorithm for deriving circuits from programs, a so-called silicon compiler, we have to circumvent the need for manipulating trace structures since they may have an infinite trace set. As we have seen in chapter three, for every command a finite state machine can be constructed. The transformations mentioned above, viz. introduction of alternation and addition of prefixes of traces, can be defined on finite state machines as well. (Defining them on the program text is much more cumbersome.) A program thus leads to a tree of finite state machines. In the next section a strategy for deriving a circuit from a finite state machine is discussed.

We have to decide how the symbols and traces are related to electrical signals and their ordering in time, i.e. we have to decide on a mechanistic appreciation. We shall first discuss a rather straightforward mechanistic appreciation and then we shall refine it.

Let M be a mechanism, an electrical circuit, implementing a command or finite state machine with trace structure T as follows. M has a number of electrical terminals to which wires can be connected; there is one input terminal per input symbol of T and one output terminal per output symbol of T. (We shall ignore connections with the "power supply".) A symbol's

occurrence is represented by a voltage transition at its corresponding terminal, which for an input or output symbol implies that M receives or generates respectively that voltage transition. The relation between M and T is such that a sequence in time of voltage transitions at the terminals of M is, for the corresponding symbols, a trace of T.

We make two remarks on the above mechanistic appreciation. First, if the terminals of M are, by wires, connected to other terminals then the delays of the electrical signals in the wires may be such that the voltage transitions at those other terminals occur in a sequence that differs from the one at the terminals of M, i.e. the ordering in time of electrical signals is not preserved by wires. It is, therefore, not necessary to require of M that it realizes exactly the ordering prescribed by T, it may be more liberal. Second, the terminals of M are located at distinct points in space and, hence, we must be careful when ordering in time the events occurring at those terminals. We, therefore, refine our mechanistic appreciation as follows. In any complete circuit the terminals of M are, by wires, connected to other terminals: there is for each symbol one wire connecting the two terminals corresponding to that symbol. In order to describe the relation between M and T we bend these wires such that they all pass through a very small region, so small that we may order in time the events occurring in it. We say that M is an implementation of T if a sequence in time of voltage transitions at the wires in this small region is, for the corresponding symbols, a trace of F(T). F(T) is a trace structure that, in agglutinations, has the same effect as T. In the next section we shall propose such a function F. Notice that, since the

circuit is delay-insensitive, in an actual layout this bending of the wires is superfluous. It only serves to describe the mechanistic appreciation.

5.1. Local strategy

In this section a method is discussed for deriving a circuit from a finite state machine. We shall first restrict ourselves to trace structures with only one signalling set and they are assumed to be passive w.r.t. this set.

Since the trace structure we consider is prefix-closed, the unique, minimal finite state machine corresponding to this trace structure has at most one state which is not a final state. We shall derive a circuit from the final states and we, therefore, assume that non-final state to have been removed. The remaining states correspond to the equivalence classes of traces as defined in section 1.5. Among these states are so-called signalling states, states that are equivalence classes of those traces that are concatenations of signalling sequences. All other states correspond to proper prefixes of such concatenations. A transition from one signalling state to another corresponds to a signalling sequence and is called a signalling state transition. We derive a circuit from the signalling states and the signalling state transitions without considering the other states.

In exceptional cases a state may exist that contains both traces that are concatenations of signalling sequences and traces that are proper prefixes of such concatenations. We assume that such a state is represented by two states in the finite state machine in such a way that the notion of signalling states is well-defined.

We can only hope to construct a physical realization for deterministic finite state machines. A finite state machine is called deterministic if in each state the input symbols uniquely determine both the next state and the output symbols. In

order to describe this requirement we introduce a function called $\underline{f}$. For trace structure T the boolean function f is, for any signalling state $[s]$, set of input symbols V, and output symbol c, defined by

$$f([s], V, c)$$
$$\equiv \underline{E}(t: stc \in \underline{\leq} T \wedge tc \text{ is a prefix of a signalling sequence} : t \restriction_{\underline{i}} T \in V^*)$$

Informally we may say that $f([s], V, c)$ is equivalent to: in state $[s]$ inputs V admit output c. For signalling state $[s]$ and trace t we call t an f-trace in $[s]$ if t contains distinct symbols only and

$$\underline{A}(c, V: c \in \underline{\subseteq} T \wedge c \text{ occurs in } t \wedge V \text{ is the set of input symbols preceding } c \text{ in } t : V \text{ contains at most one symbol of each input port} \wedge f([s], V, c)) .$$

Notice that f is a monotonic function, i.e. if $V_0 \subseteq V_1$ then $f([s], V_0, c) \Rightarrow f([s], V_1, c)$.

Trace structure T, or the minimal finite state machine with trace structure T, is called deterministic if the following two conditions hold for each signalling state $[s]$ and each trace t that is an f-trace in $[s]$:

- t can be obtained from a trace u, such that $su \in \underline{\leq} T$, by shifting in u input symbols to the left and output symbols to the right, and
- all such u lead to the same state $[su]$ of T.

Notice that the first of these two conditions implies that

- all symbols in t are from distinct ports,
- if t contains a symbol from each output port then t also contains a symbol from each input port, and
- if t contains a symbol from each port then t is a passive signalling sequence.

Let $F(T)$ be defined as the trace structure

$$
\begin{aligned}
&< \{\, s, t \\
&\quad : [s] \text{ is a signalling state of } T \ \wedge\ t \text{ is an f-trace in } [s] \\
&\quad : st \\
&\quad \} \\
&, \underline{i}T \\
&, \underline{o}T \\
&> \quad ;
\end{aligned}
$$

$F(T)$ and T have the same ports and signalling set. Notice that, for deterministic T, $F(T)$ is passive w.r.t. this signalling set, and T and $F(T)$ have the same signalling states. For all T we have $T \leq F(T)$.

Example 5.0

For trace structures T and U, such that

$T = \mathrm{PREF}(<\{abcd, adcb\}, \{a, c\}, \{b, d\}>)$,

$U = \mathrm{PREF}(<\{acbd\}, \{a, c\}, \{b, d\}>)$,

and each symbol constitutes a port, we have

T is not deterministic,

U is deterministic,

$$
\begin{aligned}
F(T) = \mathrm{PREF}(<&\{abcd, abdc, acbd, acdb \\
&, adbc, adcb, cabd, cadb \\
&\} \\
&, \{a, c\} \\
&, \{b, d\} \\
&>) \quad . \quad \text{and}
\end{aligned}
$$

$F(U) = \mathrm{PREF}(<\{acbd, acdb, cabd, cadb\}, \{a,c\}, \{b,d\}>)$.

(End of example)

Property 5.1

Let U be a trace structure that has, among its signalling sets, the signalling set of T. Let U be active w.r.t. this signalling set. For deterministic T we have

$U \underline{g} T = U \underline{g} F(T)$.

Proof

Since $T \subseteq F(T)$ it remains to show $\sqcup_{\underline{g}} F(T) \subseteq \sqcup_{\underline{g}} T$. Notice that $\underline{o}T \subseteq \underline{o}\sqcup$. For all x: $x \in (\underline{o}\sqcup \setminus \underline{o}T)^*$, we have, informally,

$x \in \underline{t}(\sqcup_{\underline{g}} F(T))$

$\equiv$ { def. of $\underline{g}$, $\underline{o}F(T) = \underline{o}T \subseteq \underline{o}\sqcup$ }

$x \in \underline{t}(\sqcup?!(\underline{o}T, \underline{i}T)\ \underline{b}\ G(\underline{o}T)\ \underline{b}\ F(T)?!(\underline{i}T, \underline{o}T))$

$\equiv$ { def. of $\underline{b}$ }

$\underline{E}(t, y, z:\ y \in \underline{t}\sqcup?!(\underline{o}T, \underline{i}T) \wedge y\restriction(\underline{o}T? \cup \underline{i}T!) = z\restriction(\underline{o}T? \cup \underline{i}T!)$
$\wedge\ t \in \underline{t}F(T)?!(\underline{i}T, \underline{o}T) \wedge t = z\restriction(\underline{i}T? \cup \underline{o}T!) \wedge z \in \underline{t}G(\underline{o}T)$
$:\ y\restriction(\underline{o}\sqcup \setminus \underline{o}T) = x)$

$\Rrightarrow$ { def. of $F(T)$, T is deterministic }

$\underline{E}(t, u, y, z:\ y \in \underline{t}\sqcup?!(\underline{o}T, \underline{i}T) \wedge y\restriction(\underline{o}T? \cup \underline{i}T!) = z\restriction(\underline{o}T? \cup \underline{i}T!)$
$\wedge\ u \in \underline{t}T?!(\underline{i}T, \underline{o}T) \wedge t = z\restriction(\underline{i}T? \cup \underline{o}T!) \wedge z \in \underline{t}G(\underline{o}T)$
$\wedge\ t$ is obtained from u by shifting in u symbols from $\underline{i}T?$ to the left and symbols from $\underline{o}T!$ to the right
$:\ y\restriction(\underline{o}\sqcup \setminus \underline{o}T) = x)$

$\equiv$ { calculus }

$\underline{E}(t, u, y, z:\ y \in \underline{t}\sqcup?!(\underline{o}T, \underline{i}T) \wedge y\restriction(\underline{o}T? \cup \underline{i}T!) = z\restriction(\underline{o}T? \cup \underline{i}T!)$
$\wedge\ u \in \underline{t}T?!(\underline{i}T, \underline{o}T) \wedge t = z\restriction(\underline{i}T? \cup \underline{o}T!) \wedge z \in \underline{t}G(\underline{o}T)$
$\wedge\ u$ is obtained from t by shifting in t symbols from $\underline{i}T?$ to the right and symbols from $\underline{o}T!$ to the left
$:\ y\restriction(\underline{o}\sqcup \setminus \underline{o}T) = x)$

$\equiv$ { informal def. of G }

$\underline{E}(t, u, y, z:\ y \in \underline{t}\sqcup?!(\underline{o}T, \underline{i}T) \wedge y\restriction(\underline{o}T? \cup \underline{i}T!) = z\restriction(\underline{o}T? \cup \underline{i}T!)$
$\wedge\ u \in \underline{t}T?!(\underline{i}T, \underline{o}T) \wedge u = z\restriction(\underline{i}T? \cup \underline{o}T!) \wedge z \in \underline{t}G(\underline{o}T)$
$:\ y\restriction(\underline{o}\sqcup \setminus \underline{o}T) = x)$

$\equiv$ { def. of $\underline{b}$ }

$x \in \underline{t}(\sqcup?!(\underline{o}T, \underline{i}T)\ \underline{b}\ G(\underline{o}T)\ \underline{b}\ T?!(\underline{i}T, \underline{o}T))$

$\equiv$ { def. of $\underline{g}$ }

$x \in \underline{t}(\sqcup_{\underline{g}} T)$.

(End of property and proof)

We have said that a symbol's occurrence is represented by a voltage transition. Such a voltage transition may arrive at a circuit before the circuit is ready for its reception, a phenomenon that we have called computation interference (cf. section 4.0). We exclude computation interference by holding up voltage transitions at the receiving circuit's input until the circuit is ready for their reception. This holdup can, from the outside so to say, not be distinguished from wire delay and, hence, the protocol that guarantees the absence of transmission interference will delay the transmission of subsequent voltage transitions at the sending circuits.

Voltage transitions are alternately high-going and low-going transitions. If a circuit has held up a voltage transition at its input, it may, upon becoming ready for reception, reconstruct that voltage transition from the voltage level at its input since in the mean time no more voltage transitions have been sent. This provides us with another reason for not allowing multiple transitions on a wire. Reconstructing that voltage transition may be done by comparing the present voltage level with the voltage level as it was since the last preceding transition. This requires a storage element per input terminal and is often called 2-cycle signalling. In order to economize on the amount of circuitry, however, 4-cycle signalling is preferred. In that case only a high-going voltage transition represents a symbol's occurrence. The low-going transitions are then irrelevant as far as the symbols are concerned. With respect to interference, however, they are not irrelevant. We shall denote a high-going or low-going voltage transition related to symbol b by $b\uparrow$ and $b\downarrow$ respectively. Similarly, a trace may be postfixed with $\uparrow$ or $\downarrow$ to express that each of the symbols in the trace is to be

postfixed accordingly. We propose that a passive signalling sequence s leads to a voltage transition sequence $(s\uparrow)(t\downarrow)$ where t is a passive signalling sequence and a permutation of s. A concatenation of signalling sequences leads to a concatenation of voltage transition sequences in which high-going and low-going subsequences alternate. Since each symbol b occurs at most once in each signalling sequence, $b\uparrow$ and $b\downarrow$ alternate.

We shall now discuss how a circuit M may be derived from trace structure T. With the mechanistic appreciation discussed in section 5.0, understood with due observance of the rules stated above for high-going and low-going voltage transitions, M will be an implementation of T. It is assumed that M is embedded in an isochronic region, which allows us to order in time the events occurring in M.

Circuit M consists of a number of blocks, one block per signalling state, plus some additional circuitry. Block $[s]$ corresponds to signalling state $[s]$ and has a number of input and output terminals, viz.

one input terminal $[s].b$ per input symbol b,
one output terminal $[s].c$ per output symbol c,
one input terminal $[s].N$,
one input terminal $[s].P$, and
one output terminal $[s].X([t])$ per signalling state $[t]$ for which a signalling state transition from $[s]$ to $[t]$ exists.

Blocks can have a privilege. At any moment one block has the privilege, indicating that the circuit is in the corresponding signalling state, or that it is making a signalling state transition from that state. An unprivileged block generates no voltage transitions. A privileged block $[s]$ is a partial implementation of T in the sense that its traces

are the f-traces in [s]. This must be understood with due observance of the rules stated above for high-going and low-going voltage transitions, with the following exception. A signalling sequence u leads to a voltage transition sequence $(u\uparrow)(v\downarrow)$ as before, with the exception that v is not necessarily a signalling sequence: v is a permutation of u, v starts with an input symbol, but v need not end with an output symbol. The additional circuitry that combines the blocks into circuit M deals with this exception, and it routes the privilege such that the partial implementationr of T together form a complete implementation of T. We shall first describe the construction of the blocks and then how they are combined.

Presence or absence of the privilege is, per block [s], recorded in a flip-flop [s].p. If a block, [s] say, has the privilege, it inspects the inputs for high-going transitions, generates high-going output transitions, keeps them high until the inputs go low, generates low-going outputs, and finally if all inputs are low as indicated by [s].N, it inverts its privilege flip-flop and generates a voltage transition at terminal [s].X([t]) for that [t] to which a signalling state transition has been completed. This state [t] depends on which inputs go high (and on [s], of course) and, hence, is to be computed as those inputs are received, and to be stored until the state transition is completed. One flip-flop [s].x([t]) per pair of states [s] and [t], for which a signalling state transition from [s] to [t] exists, is used for this purpose. Also the voltages to be generated for output symbols depend on the inputs. They are to be kept high by the block only as long as the inputs are high and, therefore, no storage elements are needed. Finally, a block [s] not having the privilege

generates no voltage transitions and doesn't change state until it receives the privilege by a voltage transition at its input terminal $[s].P$.

In order to describe electrical circuits we write boolean expressions and assignment statements. High and low voltages correspond to the boolean values true and false respectively. An assignment $v := e$ denotes that v is constantly assigned a value such that $v \equiv e$ is true. Existential and universal quantifications are computed with a number of OR and AND elements respectively. A flip-flop ff has two input terminals, $ff.R$ and $ff.S$, and two output terminals, $ff.Q$ and $ff.\overline{Q}$. Their values will be such that $\neg ff.R \vee \neg ff.S$ and $ff.\overline{Q} \equiv \neg ff.Q$. Assigning false to both $ff.R$ and $ff.S$ does not affect the flip-flop's outputs whereas

$ff.R := true$ yields $\neg ff.Q \wedge ff.\overline{Q}$

and

$ff.S := true$ yields $ff.Q \wedge \neg ff.\overline{Q}$.

The absence or presence of the privilege is recorded such that

$[s].p.Q \equiv$ block $[s]$ is privileged .

For each output symbol c , $c \in \underline{\underline{\subseteq}} T$, the value of output terminal $[s].c$ is computed by

$[s].c := [s].p.Q \wedge f([s], V, c)$

where V is the set of input symbols whose input terminals have value true. Substituting the definition of f and V, and distributing the conjunction through existential quantification we may rewrite this assignment as

$$[s].c := \underline{E}(t: stc \in \underline{\underline{\subseteq}} T \wedge tc \text{ is a prefix of a signalling sequence} : [s].p.Q \wedge \underline{A}(b: b \text{ occurs in } t \mid \underline{\underline{\subseteq}} T: [s].b))$$

The state transition flip-flops are driven similarly, viz.

$$[s].x([t]).S := \underline{E}(u:\ s\,u \in \underline{\iota}T \wedge [s\,u]=[t] \wedge u \text{ is a signalling sequence} : [s].p.Q \wedge \underline{A}(b:\ b \text{ occurs in } u\restriction\underline{\iota}T : [s].b))\ .$$

Their other input is controlled by

$$[s].x([t]).R := [s].p.\overline{Q}\ .$$

The privilege flip-flop $[s].p$ is controlled by

$$[s].p.S := [s].P$$

and

$$[s].p.R := [s].N \wedge \underline{E}(t:: [s].x([t]).Q)\ .$$

The remaining output terminals are controlled by

$$[s].X([t]) := [s].p.\overline{Q} \wedge [s].x([t]).Q\ .$$

From an operational point of view we may conclude from the above that, if block $[s]$ does not have the privilege, we have

$$\neg [s].p.Q$$

and, consequently,

$\neg [s].c$ for all c ,

$\neg [s].x([t]).S$ for all t .

We shall see to it that

$\neg [s].x([t]).Q$ for all t

from which we conclude

$\neg [s].X([t])$ for all t ,

$\neg [s].p.R$.

Thus an unprivileged block $[s]$ has outputs with value false only and it is ready for receiving the privilege via $[s].P$. Upon reception of the privilege $[s].P$ becomes true and, hence,

$$[s].p.S := \text{true}$$

which, thanks to $\neg [s].p.R$, yields

$$[s].p.Q \wedge \neg [s].p.\overline{Q}\ .$$

Next the output terminals $[s].c$ are assigned values in accordance with the values that the input terminals $[s].b$ receive. Due to the determinism of the trace structure being implemented, at most one symbol's terminal of each output

port becomes true. If the input values are such that they represent the input symbols of a signalling sequence u then

$$[s].\ x([s\,u]).\ S := \text{true}$$

which yields

$$[s].\ x([s\,u]).\ Q \quad .$$

Due to the determinism at most one state transition flip-flop changes value. Furthermore, of each output port one symbol's terminal has then been assigned the value true. From the signalling set protocol it follows that thereafter the inputs $[s].b$ become false again. As soon as they are false we have

$$[s].N \quad ,$$

$$\neg\,[s].\ c \qquad \text{for all } c \ ,$$

$$\neg\,[s].\ x([t]).\ S \qquad \text{for all } t \ .$$

($[s].N$ is an input of block $[s]$ and its becoming true cannot be concluded from the above text; we return to this later.) Notice that signalling sequence u leads, by the above scheme, to a voltage transition sequence $(u\uparrow)(v\downarrow)$ where v is a permutation of u that starts with an input symbol but need not end on an output symbol. The method of computing the outputs in $u\uparrow$ gives $v\downarrow$ for free.

Next we have

$$[s].\ p.\ R := \text{true}$$

which yields

$$\neg\,[s].\ p.\ Q \ \wedge\ [s].\ p.\ \overline{Q}$$

followed by

$$[s].\ X([s\,u]) := \text{true} \quad ,$$

$$[s].\ x([s\,u]).\ R := \text{true} \quad .$$

Next follows

$$\neg\,[s].\ x([s\,u]).\ Q$$

and then

$$[s].\ X([s\,u]) := \text{false} \quad ,$$

$$[s].\ p.\ R := \text{false} \quad .$$

The fact that

$[s].X([s\,u]) := \text{true}$

precedes

$\neg\,[s].x([s\,u]).Q$

is not a consequence of the above assignments and must be controlled by, for example, making the x flip-flops relatively slow.

The blocks thus obtained are combined to yield the circuit that implements a trace structure. The input terminals of the circuit are connected to the corresponding input terminals of the blocks by

$[s].b := b$.

Controlling the circuit's outputs is more complex since, as we have said before, the low-going voltage transition sequence starts with an input but need not end with an output. It does end with an output if there is only one input port. If there is more than one input port, some output must be delayed in becoming low until all inputs are low. To realise this delay, Muller C elements are used [20]. (A Muller C element is an element with a number of inputs and one output. If all inputs have the same value the output will also receive that value, and otherwise the output doesn't change value.) There are two ways of computing the circuit's outputs: the immediate output

$c := \underline{E}(s::\ [s].c)$

and the delayed output

$c := \underline{E}(s::\ [s].c)\ \ \underline{C}\ \ \underline{E}(b:\ b \in \underline{i}T:\ b)$

where $x\ \underline{C}\ y$ denotes the output of a Muller C element with inputs x and y. There are two cases: if there is only one input port, all outputs are immediate. If there is more than one input port, arbitrarily one of the output ports is selected to have delayed outputs for all symbols in that port, and all other outputs are immediate.

For every block [s] input terminal [s].N indicates that all

terminals corresponding to input symbols have value false. Therefore we include

$$[s].N := \neg \underline{E}(b: b \in \underline{i}T: b) \quad .$$

The privilege passing is, for each block $[t]$, controlled by

$$[t].P := \underline{E}(s:: [s].X([t])) \quad .$$

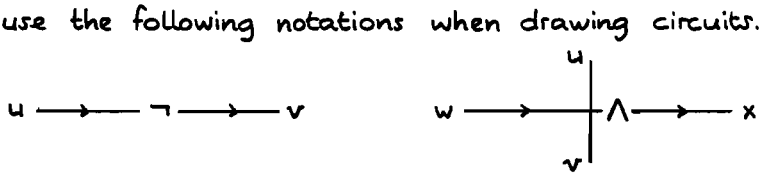
The initialization of the circuit must be such that all input and output terminals have value false, that all flip-flops have outputs $\overline{Q} \wedge \neg Q$, with exception of the privilege flip-flop of the initial state, whose output must be $[\varepsilon].p.Q \wedge \neg [\varepsilon].p.\overline{Q}$. This completes the circuit's construction.

With respect to the layout we recall the assumption that the circuit implementing a finite state machine is embedded in an isochronic region. This assumption is essential in order to express requirements on relative speeds. We have already said that the x flip-flops must be relatively slow. The other requirements are that the blocks receive their inputs at about the same time, and that the passing of privileges is not excessively slow compared to the handling of inputs and outputs. The latter requirement boils down to the requirement that high-going inputs of the next signalling sequence arrive after the currently privileged block has given up the privilege, though not necessarily after the successor block has received the privilege. Notice that, in the latter case, the conjunct $[s].p.Q$ in the computation of $[s].c$ and $[s].x([t]).S$ is false and represents the holdup of input transitions until the circuit, more specifically block $[s]$, is ready for their reception. Thus, the privileges are used to exclude computation interference.

Notice that the values at inputs of OR and AND elements computing the circuit's outputs do not change in opposite directions. As a consequence, the circuit does not momentarily generate erroneous values: the circuit is free from hazards.

We mention a few optimizations that will be applied to the examples we present. Privilege passing from a block to itself, i.e. along $[s].X([s])$, can be omitted if in that case also the assignment $[s].p.R := true$ is omitted. If a circuit consists of one block only and that block always has the privilege, the presence thereof need not be recorded (in a flip-flop). The circuits often contain OR and AND elements with either only one input or two inputs of which one is constant. They can be removed but we have not done so in every case. The circuits often compute boolean values described by an expression of the form $E0 \vee (E0 \wedge E1)$. This expression may be reduced to $E0$.

We use the following notations when drawing circuits.



These pictures represent an inverter, satisfying

$$v := \neg u \ ,$$

and an AND element, satisfying

$$u \equiv v \quad \text{and} \quad x := u \wedge v \wedge w \ .$$

An OR or C element is drawn similarly.

Example 5.2

Consider the full adder of example 2.5 and the partitioning of its alphabet of example 4.5. The full adder is passive w.r.t. the signalling set that consists of all its ports. The trace structure is deterministic and, hence, without transformations the strategy for deriving a circuit as described above is applicable.

In the program text that follows we indicate the partitioning of the alphabet but we make no attempt at formalizing the syntax.

```
com full adder ( in {a0, a1}, {b0, b1}, {c0, c1},
                 out {d0, d1}, {e0, e1} ) :
    (a0, b0;  d0,  (c0; e0 | c1; e1)
    | b0, c0;  d0,  (a0; e0 | a1; e1)
    | c0, a0;  d0,  (b0; e0 | b1; e1)
    | a1, b1;  d1,  (c0; e0 | c1; e1)
    | b1, c1;  d1,  (a0; e0 | a1; e1)
    | c1, a1;  d1,  (b0; e0 | b1; e1)
    )*
moc
```

The finite state machine has one signalling state, viz. [ε], and, hence, the circuit consists of one block which is permanently privileged. Therefore no privilege is recorded. We shall avoid to draw a diagram of the finite state machine, and we merely observe that, for example, in the 36 signalling state transitions in which symbol d0 occurs d0 is preceded by either

a0 and b0 (plus, possibly, c0 or c1)

or

b0 and c0 (plus, possibly, a0 or a1)

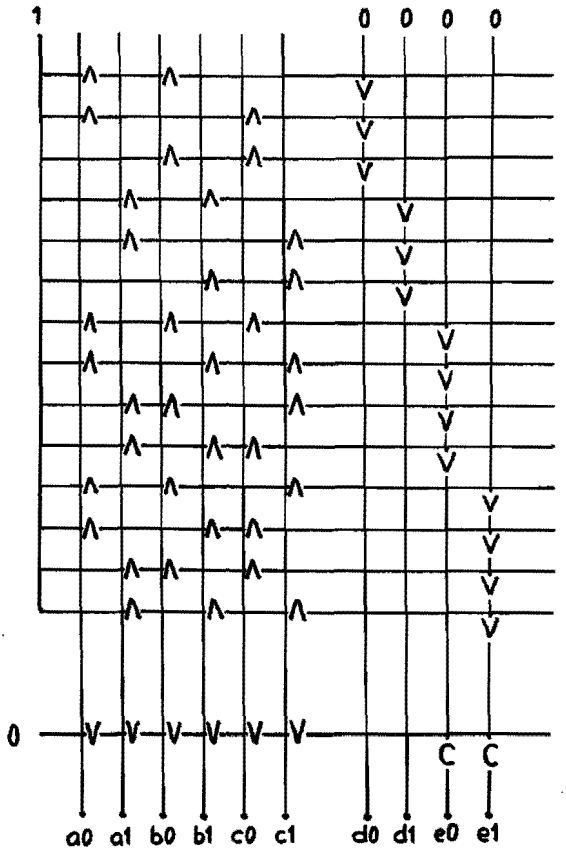
or

c0 and a0 (plus, possibly, b0 or b1) .

According to the construction method, output d0 is controlled by the assignment

$d0 := (a0 \wedge b0) \vee (b0 \wedge c0) \vee (c0 \wedge a0)$.

In general, the outputs are controlled by such an AND-OR circuit. This type of a circuit is often called a PLA and can be implemented efficiently [18].



The port {e0, e1} is chosen to have Muller C elements. With respect to correct operation this choice is irrelevant. With respect to speed, however, this choice is the preferred one: it allows maximum concurrency in the undoing of carry-propagation in the low-going voltage transition sequence if full adders are cascaded.

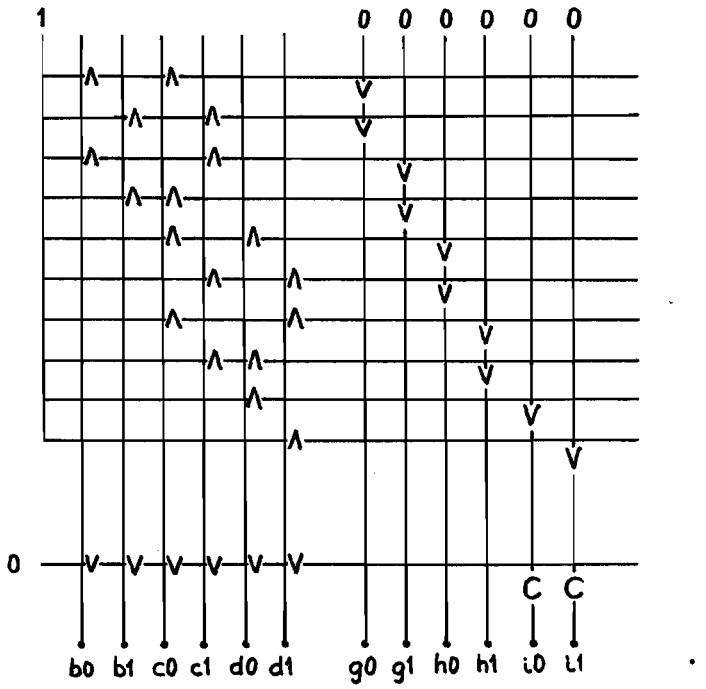
(End of example)

Example 5.3

The programs from example 2.6 can be dealt with in a way similar to the previous example. Therefore

```
com  binary to Gray ( in  {b0, b1}, {c0, c1}, {d0, d1},
                      out {g0, g1}, {h0, h1}, {i0, i1} ) :
     (( b0, c0; g0 | b0, c1; g1 | b1, c0; g1 | b1, c1; g0)
     ,( c0, d0; h0 | c0, d1; h1 | c1, d0; h1 | c1, d1; h0)
     ,( d0; i0 | d1; i1 )
     )*
moc
```

yields



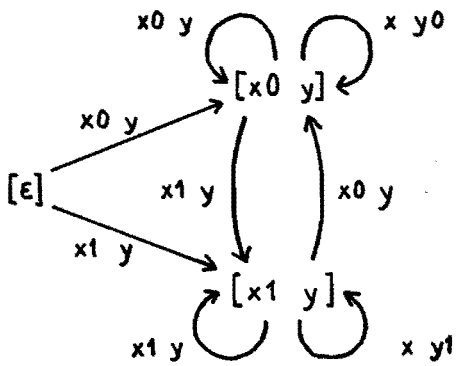
(End of example)

Example 5.4

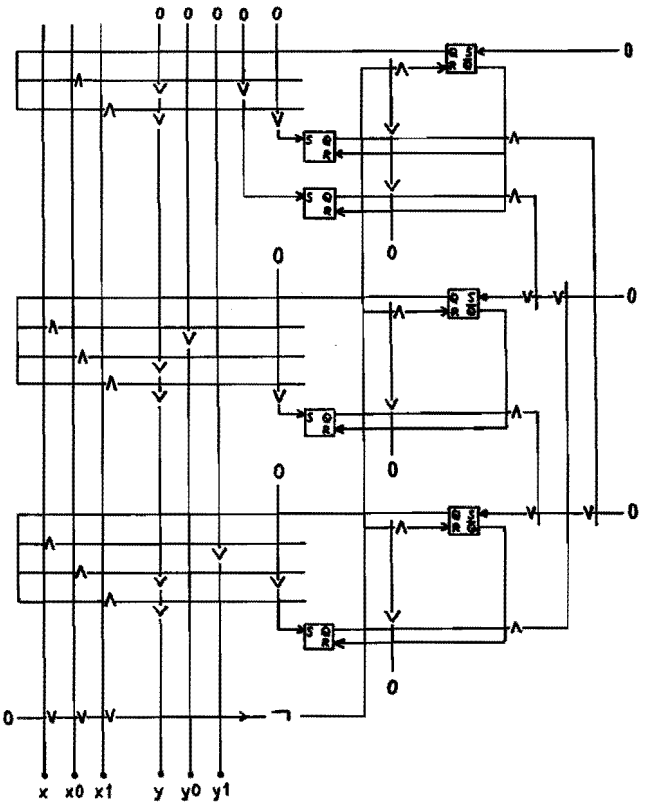
In example 2.4 a binary variable is discussed. If x0 and x1 are inputs, and y0 and y1 outputs, no partitioning of the alphabets makes the variable passive or active. Hence, it is to be transformed by the introduction of alternation. We assume it to be passivated. Using fresh symbols x and y the result of the transformation might be described by the program text

```
com   var ( in {x, x0, x1}, out {y, y0, y1} ) :
      (x0; y; (x; y0)* | x1; y; (x; y1)* )*
moc   .
```

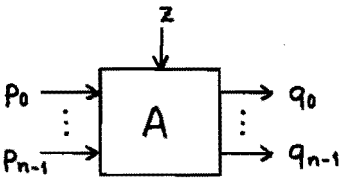
The trace structure has three signalling states as may be illustrated by the following diagram.



Consequently, the circuit has three blocks. By optimization, two privilege passing flip-flops are omitted, thus leading to the circuit on the next page. The blocks [ε], [x0 y], and [x1 y] are drawn, in that order, from top to bottom. At the bottom the computation of N is drawn, and to the right of each block [s] the computation of [s].P is drawn. Within each block we find, from left to right, the inputs, the outputs, the x flip-flops, N, and the p flip-flop.



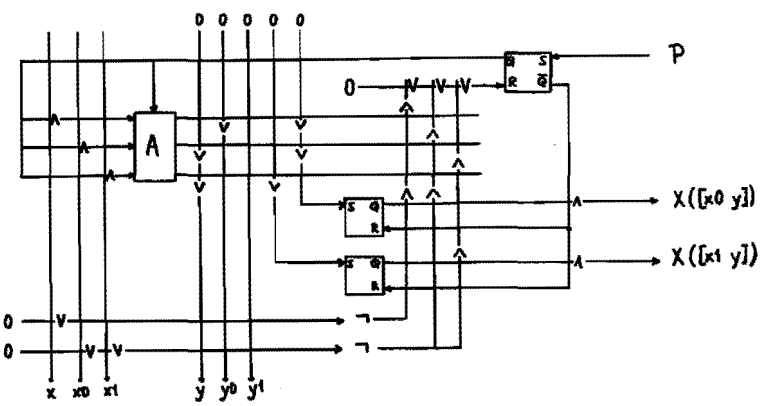
Let us, by way of experiment, construct a circuit for the same variable in the case of a different partitioning of its alphabets, viz. four ports and two signalling sets $\{\{x\}, \{y0, y1\}\}$ and $\{\{x0, x1\}, \{y\}\}$. This is a variable that allows its value to be inspected by another component than the one that assigns it a value. We shall make no attempt at defending the circuit's correctness. The variable is passive w.r.t. both signalling sets. In the states [x0 y] and [x1 y] it is possible, due to the <u>two</u> signalling sets, that both x and one of x0 and x1 become true, in which case a selection mechanism is required. We postulate the availability of an <u>arbitration circuit</u> with n+1 inputs and n outputs, $n \geq 1$, which we draw as follows.



The values of the inputs and outputs are such that

$$(z \wedge \underline{N}(i : 0 \le i < n : q_i) \le 1)$$
$$\vee\ (\neg z \wedge \underline{N}(i : 0 \le i < n : q_i) = 0)$$

where $\underline{N}(i : 0 \le i < n : q_i)$ denotes the number of values of i in the range $0 \le i < n$ for which q_i is true. Furthermore, if q_i is true then p_i has been true at some moment since z lastly became true. No p input changes from true to false before some q has become true. No q output changes from true to false as long as z remains true. Notice that if z and at least two p inputs are true a choice must be made regarding to which q is to be true. It is not specified which one is chosen, although the choice does not change as long as z holds. Notice also that we do not require the arbiter to "immediately" make a choice: it may take an arbitrary amount of time to do so. Drawing only one of the three blocks, [x0 y] say, including the computation of the two N inputs (one per signalling set), we obtain the following picture.



Notice that there are three modifications. First, the arbiter has been added. Second, the state transition flip-flop that corresponds to the transition from [x0 y] to [x0 y] has not been removed since the falsity of [x0 y]. p. Q is used to "reset" the arbiter. Third, there is an input [x0 y].N per signalling set.

In this circuit, designed for the case of two signalling sets, the possibility of computation interference is much more pronounced than in the case of only one signalling set. It is in this example especially the arbiter that is used for excluding this form of interference.
(End of example)

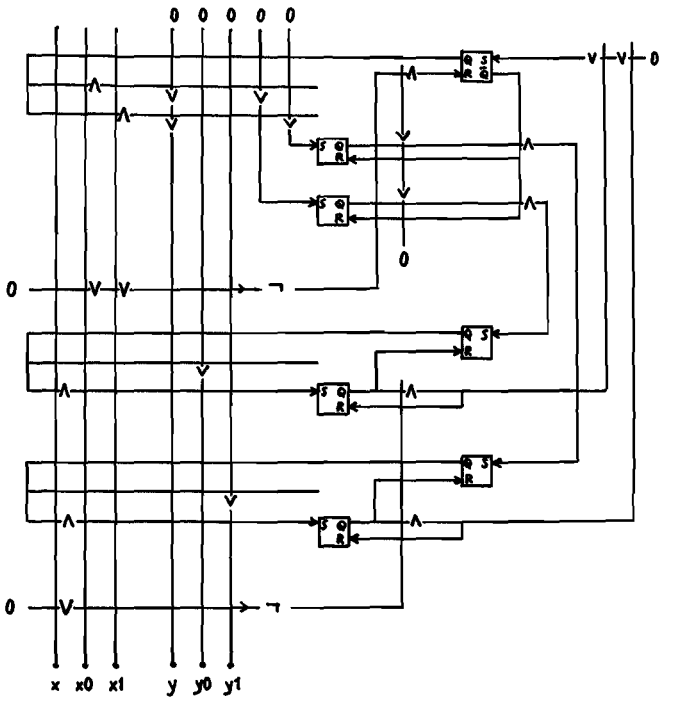
Next we mention the changes in the case of an active trace structure. Only the following assignments are different from the passive case:

$$[s].\ x([t]).\ R := [s].\ N \wedge [s].\ x([t]).\ Q \quad ,$$
$$[s].\ p.\ R := \underline{E}(t:: [s].\ x([t]).\ Q) \quad , \text{ and}$$
$$[s].\ X([t]) := [s].\ N \wedge [s].\ x([t]).\ Q \quad .$$

Example 5.5

Consider component buf_1 from example 2.2 used in the component buf_k from example 2.6. The latter component is in fact a sequence of k components buf_1. Following our strategy each of these is transformed twice by the introduction of alternation: once together with the component "to the left" and once together with the component "to the right". The result is a component with two signalling sets; it is passive w.r.t. {{ x0, x1}, {y}} and active w.r.t. {{y0, y1}, {x}}. The program text is

```
com  buf1 ( in  {x0, x1} ,  out {y} ,
            out {y0, y1} ,  in  {x} ) :
     (x0; y; y0; x | x1; y; y1; x)*
moc  .
```



The above circuit consists of one block, viz. [ε], constructed according to the rules for a passive trace structure, and two blocks, viz. [x0 y] and [x1 y], constructed according to the rules for an active trace structure. The blocks are drawn in the order mentioned from top to bottom. Since there are two signalling sets, there are two different N's.

In this circuit the use of privileges to exclude computation interference is more pronounced than in the case of only one signalling set.

Notice that we may increase the amount of concurrency in the following way. A privileged block that has com-

pleted the high-going voltage transition sequence might already give a privilege to the successor block and keep its own privilege until it has completed the low-going voltage transition sequence. Since the successor block deals with a different signalling set no transmission interference results, and since low-going transitions do not represent a symbol's occurrence it does not affect the mechanistic appreciation.
(End of example)

We have illustrated the construction of a circuit for a trace structure with more than one signalling set by examples. These circuits are more complex than for the case of one signalling set. Proceeding along the lines sketched above, we obtain circuits in which, in general, more than one block can be privileged, viz. upto the number of signalling sets. Arbiters may be required in some of those blocks. Though arbiters may require an unbounded amount of time for arbitration [2, 24, 25] this does, due to the delay-insensitivity, not affect the correct operation of the circuits.

One may wonder why we have chosen all encodings as "wasteful" as possible, viz. a wire per symbol, a flip-flop per signalling state, and a flip-flop per signalling state transition. This is the so-called "one-hot" assignment. It was chosen here in order not to complicate the description unnecessarily, to ease the avoidance of hazards, and to allow generalization to trace structures with more than one signalling set. In [13] it is argued that the one-hot assignment is not inefficient for the encoding of states and state transitions. Due to our way of excluding transmission interference, viz. per port and not per symbol, we might reduce the number of wires per port and use a k-hot assignment, $k > 1$.

In this chapter an implementation method for deterministic trace structures has been proposed. In the previous chapter a transformation has been described, viz. the introduction of alternation, that does not consider determinism. In [26] a protocol is proposed that exploits the hierarchical structure to resolve the non-determinism.

The circuits discussed in this chapter display a modest amount of concurrency, which increases in the case of more than one signalling set. Large-scale concurrency results from combining these circuits into the implementation of a program. Since no additional circuitry is needed all these circuits operate concurrently. One might say that the elimination of symbols as prescribed by the blending operation is performed concurrently.

6. On what we have rejected

The preceding chapters are the result of the research we have done. Of course, they reflect in no way the history of the considerations leading to the results reported. In this chapter we shortly discuss some of the considerations leading to the rejection of proposals not mentioned in the preceding chapters.

The early phase of our research was mainly inspired by the wish of writing programs such that they express sequential ordering of statements only when required, i.e. programs should not be overly specific and express a total ordering of their statements but should express a partial ordering. To that end we introduced the comma as a statement connective, where execution of S0, S1 stands for the execution of S0 and S1 "in either order". This connective turned out to be rather inconvenient, for example, it is not associative, and it was replaced by "any interleaving". Attempts at characterizing interleaving led us to the use of traces.

Another important source of inspiration was the program notation CSP [11], especially the input and output commands. When designing CSP programs we often encountered the need for signal communication, i.e. communication of a structured value with no components. In this case communication reduces to synchronization and it is then a nuisance if one of the processes must be specified as the sender and the other process as the receiver: transmitting no value is a symmetric relation between the two processes. When we noticed that signals could be used to transmit information, viz. by selecting one out of several signals, we abandoned the other form of input and output commands, and concentrated on the properties of undirected

signals as the sole communication mechanism. As a consequence, we could ignore such ill-understood notions as value and type.

Weaving and blending were originally defined with recursive predicates. For example, the predicate $x(t \underline{b} u)$, expressing that trace x can be blended from traces t and u, $x \in (\underline{a}T \div \underline{a}U)^* \wedge t \in (\underline{a}T)^* \wedge u \in (\underline{a}U)^*$ was defined by

$$
\begin{aligned}
& x(t \underline{b} u) \\
\equiv\ & (x = \varepsilon \wedge t = \varepsilon \wedge u = \varepsilon) \\
& \vee \underline{E}(a, t0, x0 :: t = a\ t0 \wedge x = a\ x0 \wedge a \in \underline{a}T \setminus \underline{a}U \wedge x0(t0 \underline{b} u)) \\
& \vee \underline{E}(a, u0, x0 :: u = a\ u0 \wedge x = a\ x0 \wedge a \in \underline{a}U \setminus \underline{a}T \wedge x0(t \underline{b} u0)) \\
& \vee \underline{E}(a, t0, u0 :: t = a\ t0 \wedge u = a\ u0 \wedge a \in \underline{a}T \cap \underline{a}U \wedge x(t0 \underline{b} u0)).
\end{aligned}
$$

The blend of trace structures T and U is

$$
\begin{aligned}
& \langle \{ x, t, u : x \in (\underline{a}T \div \underline{a}U)^* \wedge t \in \underline{t}T \wedge u \in \underline{t}U \wedge x(t \underline{b} u) : x \} \\
& , \underline{a}T \div \underline{a}U \\
& \rangle \ .
\end{aligned}
$$

A definition of this kind is not very amenable to formal manipulation. Besides being too long, it invites case-analysis and proofs by mathematical induction over a trace's construction. The use of projection has greatly reduced the need for such proofs.

A forerunner of agglutination was defined with a similar predicate, viz.

$$
\begin{aligned}
& x(t \underline{g} u) \\
\equiv\ & (x = \varepsilon \wedge t = \varepsilon \wedge u = \varepsilon) \\
& \vee \underline{E}(a, t0, x0 :: t = a\ t0 \wedge x = a\ x0 \wedge a \in \underline{a}T \setminus \underline{a}U \wedge x0(t0 \underline{g} u)) \\
& \vee \underline{E}(a, u0, x0 :: u = a\ u0 \wedge x = a\ x0 \wedge a \in \underline{a}U \setminus \underline{a}T \wedge x0(t \underline{g} u0)) \\
& \vee \underline{E}(a, t0, u0, u1 :: t = a\ t0 \wedge u = u0\ a\ u1 \wedge a \in \underline{a}T \cap \underline{i}U \wedge x(t0 \underline{g} u0\ u1)) \\
& \vee \underline{E}(a, t0, t1, u0 :: t = t0\ a\ t1 \wedge u = a\ u0 \wedge a \in \underline{i}T \cap \underline{a}U \wedge x(t0\ t1 \underline{g} u0))
\end{aligned}
$$

which yields a composition function that does not preserve prefix-closedness. It was, therefore, rejected in favour of agglutination defined with DEL.

We have done some experiments with the comma in commands representing blending instead of weaving. Though some nice programs could then be devised [23] we have rejected this choice on account of the weaving's being slightly simpler than blending, as exemplified by the associativity and conjunctivity. Since the time of writing chapter two we have become more fluent in deriving programs from their specifications. Reconsidering example 2.2 we would now derive the program by joining (with commas) three commands, one command per conjunct of the specification. This yields

```
com  buf1 (x0, x1, y0, y1) :
       (x0; y0)*
     , (x1; y1)*
     , ((x0 | x1); (y0 | y1))*
moc  .
```

This beautiful and-comma-rule reinforces the preference for weaving. This rule is the "dual" of the or-bar-rule that we have used without mention.

In the literature various methods of translating regular expressions into electrical circuits are known. A very elegant method uses so-called programmable building-blocks [8]. We were, however, unable to extend it with a building-block for the weaving operation. We seriously doubt the existence of such a building-block. We found it also difficult to extend the circuits such that they produce outputs instead of merely accepting or rejecting a trace. As a result we resorted to the use of deterministic finite state machines as an intermediary in the translation.

7. Epilogue

We have proposed a program notation that suggests an implementation with a high degree of concurrency. A theory of traces and trace structures has been developed to facilitate a definition of the program notation and to assist in the design of programs. We have studied the problems that may be encountered when implementing programs as silicon chips.

No attempt has been made to incorporate concurrency in the formalism used to define the program notation since we consider concurrency as belonging to an implementation. Others have suggested to replaces traces by vectors of traces [15]. Concurrency then gives rise to symbols occurring in different elements (traces) of the same vector. The length of the vectors is the maximum degree of concurrency with which the component may be executed. There are two reasons why we have not adopted the vector approach. First, the vectors reflect the internal structure of a component. Second, the vectors do not lend themselves well to the definition of recursive components. Our experience is that our traces and trace structures can be elegantly used in the definition of recursive components [27].

Brock and Ackerman [1] demonstrate that components should not be defined with functions from input traces to sets of output traces. The reason is that such a formalism does not express order between occurrences of input and output symbols. Our approach does not have this defect.

The notation for directed trace structures and their operations is not yet very satisfactory. For example, the

proof of property 4.8 is more a struggle with the notations than with the property.

Designing electrical circuits without making assumptions on wire delays turned out to be very challenging. We have proposed circuits that consist of smaller circuits embedded in isochronic regions. No global or local clock is used, not even a pausable clock. A fundamental phenomenon of arbiter and synchronizer circuits, viz. synchronization failure, does not affect correct circuit operation. In the long run this class of circuits will, therefore, be considered simpler than the now-popular synchronous circuits.

8. References

[0] F. S. Beckman,
Mathematical Foundations of Programming,
The systems programming series,
Addison - Wesley, 1980.

[1] J. Dean Brock, William B. Ackerman,
Scenarios: a model of non-determinate computation,
in Proc. Int. Coll. on Formalization of Programming Concepts,
Lecture Notes in Computer Science, 107,
Springer-Verlag, 1981, pp. 252 - 259.

[2] T. J. Chaney, C. E. Molnar,
Anomalous Behavior of Synchronizer and Arbiter Circuits,
IEEE Transactions on Computers, vol. C-22, 1973, pp. 421-422.

[3] Edsger W. Dijkstra,
Cooperating Sequential Processes,
in Programming Languages (F. Genuys, ed.),
Academic Press, 1968, pp. 43-112.

[4] Edsger W. Dijkstra,
A tutorial on the split binary semaphore,
EWD 703, 1979.

[5] Edsger W. Dijkstra,
Lecture notes "Predicate transformers" (Draft),
EWD 835, 1982.

[6] Edsger W. Dijkstra,
Reducing control traffic in a distributed
implementation of mutual exclusion,
EWD 851, 1983.

[7] Robert W. Floyd, Jeffrey D. Ullman,
The compilation of regular expressions into integrated circuits,
Journal of the ACM, vol. 29, 1982, pp. 603-622.

[8] M. J. Foster, H. T. Kung,
Recognize regular languages with programmable building-blocks,
in VLSI 81 (John P. Gray, ed.),
Academic Press, 1981, pp. 75-84.

[9] H. W. Fowler, F. G. Fowler,
The Concise Oxford Dictionary of Current English,
Seventh Edition, Oxford University Press, 1982.

[10] Seymour Ginsburg,
The Mathematical Theory of Context-free Languages,
McGraw-Hill, 1966.

[11] C. A. R. Hoare,
Communicating Sequential Processes,
Communications of the ACM, vol. 21, 1978, pp. 666-677.

[12] C. A. R. Hoare,
A Model for Communicating Sequential Processes,
Technical Monograph PRG-22, 1981.

[13] Lee A. Hollaar,
Direct Implementation of Asynchronous Control Units,
IEEE Transactions on Computers, vol. C-31, 1982, pp. 1133-1141.

[14] J. E. Hopcroft, J. D. Ullman,
Formal Languages and their Relation to Automata,
Addison-Wesley, 1969.

[15] P. E. Lauer,
Synchronization of concurrent processes without globality assumptions,
ACM Sigplan Notices, vol. 16, 1981, pp. 66-80.

[16] Alain J. Martin,
An axiomatic definition of synchronization primitives,
Acta Informatica, vol. 16, 1981, pp. 219-235.

[17] Alain J. Martin,
Distributed mutual exclusion on a ring of processes,
Caltech technical report 5080:TR:83, 1983.

[18] Carver Mead, Lynn Conway,
Introduction to VLSI systems,
Addison-Wesley, 1980.

[19] Carver Mead, Martin Rem,
Minimum Propagation Delays in VLSI,
IEEE Journal of Solid-State Circuits, vol. SC-17,
1982, pp. 773-775.

[20] Raymond E. Miller,
Switching Theory,
Wiley, 1965, vol. 2, chapter 10.

[21] Robin Milner,
A Calculus of Communicating Systems,
Lecture Notes in Computer Science, 92,
Springer-Verlag, 1980.

[22] W. Peremans,
private communication, 1982.

[23] M. Rem,
Partially ordered computations, with applications to
VLSI design,
in Foundations of Computer Science IV, part 2,
(J.W. de Bakker, J. van Leeuwen, eds.),
MC-Tract 159, Mathematical Centre, 1983, pp. 1-44.

[24] Science and the citizen,
Scientific American, vol. 228, 1973, pp. 43-44.

[25] C. L. Seitz,
System Timing,
in [18], pp. 218-262.

[26] Jan L. A. van de Snepscheut,
Synchronous communication between asynchronous components,
Information Processing Letters, vol. 13, 1981, pp. 127-130.

[27] Jan Tijmen Udding,
On recursively defined sets of traces,
THE memorandum, JTU 0a, 1983.

[28] Jan Tijmen Udding,
On self-timed and delay-insensitive compositions,
THE memorandum, JTU 1, 1982.

Index

Samenvatting

In dit proefschrift wordt een methode besproken om programma's als chip te implementeren. Er wordt een programmanotatie voorgesteld waarbij programma's samenwerking voorschrijven van componenten die in een hiërarchische structuur gerangschikt zijn. Deze hiërarchische structuur helpt bij het beteugelen van de complexiteit van programmaontwerp en suggereert een implementatie met een grote mate van gelijktijdigheid.

VLSI is een techniek die het mogelijk maakt om een groot aantal actieve elementen in een chip onder te brengen. Daardoor sluit VLSI goed aan bij de grote mate van gelijktijdigheid die door de programmanotatie gesuggereerd wordt. Een probleem bij het ontwerpen van chips is dat bij afnemende afmetingen de transportsnelheid van elektrische signalen relatief afneemt. Door geschikte communicatieprotocollen af te spreken leidt de voorgestelde implementatiemethode tot chips waarvan de correcte werking onafhankelijk is van de transportsnelheid.

Om de implementatie van programma's als chip te beschrijven wordt een theorie van symboolrijen en verzamelingen symboolrijen ingevoerd. De betekenis van programma's, de vertraging die optreedt bij transporten, en de communicatieprotocollen worden met behulp van deze theorie gedefinieerd. Er worden operaties ingevoerd die bij twee verzamelingen symboolrijen een derde bepalen. Hiermee wordt de samenwerking van componenten beschreven. Een belangrijk deel van het proefschrift is gewijd aan eigenschappen van deze operaties, o.a. in combinatie met vereniging, doorsnijding en projectie. Reguliere talen blijken

gesloten te zijn onder deze operaties. In dit verband wordt aandacht geschonken aan eindige automaten en, en passant, aan het minimaliseren ervan. Bij de vertaling van programma's naar chips worden die automaten als tussenstap gebruikt.

Curriculum vitae

De schrijver van dit proefschrift werd op 12 september 1953 geboren te Oosterhout, Noord-Brabant. Na het eindexamen HBS-B in 1970 aan het Van der Putt Lyceum te Eindhoven volgde de studie elektrotechniek aan de Technische Hogeschool Eindhoven. Na het afstudeerwerk onder leiding van prof. dr. F. E. J. Kruseman Aretz werd in april 1977 het diploma elektrotechnisch ingenieur behaald. Van 1 november 1974 tot 1 mei 1977 werden werkzaamheden verricht binnen het rekencentrum van de hogeschool. Daarna werd onder leiding van drs. C. Bron onderzoek verricht in de vakgroep Informatica van de Technische Hogeschool Twente. Sinds 15 september 1978 werd als wetenschappelijk assistent en later als wetenschappelijk medewerker aan de Onderafdeling der Wiskunde en Informatica van de Technische Hogeschool Eindhoven gewerkt in de vakgroep Informatica onder leiding van prof. dr. M. Rem.

Stellingen

0. In hoofdstuk 2 van dit proefschrift wordt een programmanotatie ingevoerd. Wanneer recursieve componenten worden toegelaten zijn alle berekenbare functies te programmeren.

1. Het vergelijken van de prestaties van rekenautomaten op grond van het gemiddelde van genormeerde rekentijden leidt tot anomalieën.

2. Het verschil tussen de begrippen "macro expanded code", "threaded code" en "interpreted code" is zo gering dat zij nauwelijks aparte namen verdienen.

3. Type-controle van Pascal programma's wordt bemoeilijkt door de expressies [] en nil.

4. Een voordeel van applicatieve programmanotaties ten opzichte van imperatieve is dat ze zich nog minder lenen voor handsimulatie.

5. Voor auto's met een conventionele LPG-installatie is met betrekkelijk eenvoudige middelen een veel betere verbranding te realiseren. Daardoor kan het brandstofverbruik met een derdedeel gereduceerd worden.

6. Zolang het verschil tussen static en dynamic hazards niet anders geformuleerd kan worden dan met voorbeelden, verdient het aanbeveling om het onderscheid niet aan te brengen.

7. Het vierkleurenprobleem is niet opgelost.

   Lit. K. Appel, W. Haken,
   Every planar map is four colorable.
   Illinois J. Math., vol. 21, 1977, pp. 429 - 567.

8. Het lijkt onverstandig om te proberen het vierkleuren-probleem op te lossen door vier groepen van 150 wiskundigen te formeren en elke groep één kleur toe te wijzen. Het ontwerpen van grote programma's gebeurt vaak wel op zo'n manier, al is het even uitzichtloos.

9. Het verschil in effectiviteit tussen wetenschappelijk getrainde programmeurs en door de praktijk gevormde programmeurs wordt met de dag groter. Dat komt zowel door de snelle kwalitatieve groei in de informatica als door de snelle kwantitatieve groei in onvoldoend geschoolde programmeurs.

10. De ontwikkeling van calculi wordt op grote schaal veronachtzaamd.

11. Programmeren is meer dan het aanwenden van standaardtechnieken; het meerdere is karakteristiek voor het vak. Didactiek van de informatica hoort daarom bij de informatica.

Jan L. A. van de Snepscheut
Eindhoven 19831014