The Enabling Model

A Tool for Performance Analysis

of Concurrent Mechanisms

Wim Kloosterhuis

The Enabling Model

A Tool for Performance Analysis of Concurrent Mechanisms

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op maandag 18 november 1991 om 16.00 uur

door

Willem Egbert Hendrik Kloosterhuis

geboren te Onstwedde

druk: wibro dissertatiedrukkerij, helmond.

Dit proefschrift is goedgekeurd door

de promotor
prof.dr. M. Rem

en

de copromotor
dr. P.A.J. Hilbers.

Preface

A program is correct if it satisfies its *specification*. For sequential programs, the specification usually captures data only: a program must produce output values that are in a specified relation to its input. In addition to the specification of data, there may be performance specifications, such as the specification of speed, size, and power-consumption of VLSI realizations. In this thesis we confine ourselves to real-time behaviour. We distinguish two types of real-time specifications. The first is the 'critical specification': such a specification states that output has to be produced within a restricted amount of time after the input. The second is the 'intentional specification', which is not really a specification. In the case of an intentional specification, the *quality* of a program is expressed in terms of its speed: a fast program has a high quality, a program without progress has zero quality (and is rejected).

For programs that cooperate with other programs by means of exchange of data (communications), the real-time aspect becomes more complicated. It is not just a matter of being fast enough, the real-time behaviour of a program must also be in accordance with the behaviour of its environment: for example, a program and its environment should not impose conflicting restrictions on the order of their communications. Such behaviours are also called *reactive systems* [10, 22]. In contrast to [10, 22], however, we are especially interested in those members of this species that have a rather regular behaviour: we do not intend to describe the behaviours of micro-wave ovens, nuclear power plants, and TV-sets, nor the effect of power failure and nuclear melt down. Furthermore, again in contrast to [10, 22], we intend to describe the *progress* of implementations, and their speed relative to (real-time) specifications.

In the sequel the term *mechanism* is used for programs, specifications, and implementations. We assume that the mechanisms operate under *true concurrency* (also known as maximal parallelism), without synchronization upon a global clock. The only interaction between cooperating mechanisms is by means of synchronization upon shared actions via *rendez vous* communication. So, we have an *asynchronous* mode of operation, and *synchronous* communication. The described mechanisms may be VLSI programs [2], or their realizations on the level of handshake protocols [3]. This is a higher level of abstraction than in [4], in which the performance of mechanisms is analysed on the level of individual transitions. Furthermore, we are especially interested in external behaviour, whereas in [4] only *closed* mechanisms are analysed.

Two aspects of the behaviour of mechanisms are *safety* and *liveness* properties: a mechanism is *safe* when it cannot get into an undesirable state; it is *live* when it will eventually perform some desired behaviour. By their nature, safety properties are easier to cope with than liveness properties (it suffices to check all possible behaviours, whereas for liveness properties it is important in which way the 'actual' behaviour is selected from this set). Up to a reasonable extent, liveness properties can be formalized without the introduction of a 'time-axis': for example with temporal logic [18, 22], or with a trace formalism [13, 16, 21]. We are interested in a more detailed analysis of the behaviour of mechanisms: typical questions we want to answer are: *'when is a mechanism at least as fast as another mechanism?'* and *'how much faster (slower) is a mechanism than another one?'*. In order to support such an analysis, a dense time-axis is indispensable.

In the enabling model, possible behaviours of mechanisms are given by *schedules*. In contrast to traces, which give an ordering of actions only, schedules assign 'time-slots' to actions. Schedules are similar to the sequence functions that are used in [23, 27]. In these works, however, the underlying model is a trace-formalism, which is inadequate to support the use of schedules in a satisfactory way. Within the enabling model, we introduce a comparison relation for performance, such that parallel composition and alphabet restriction are monotonic (with respect to this relation). Such a relation is suitable for compositional design of mechanisms, i.e. for design by means of 'divide and conquer'.

The enabling model is a tool for performance analysis that is useful for (at least) a restricted class of mechanisms (see the characterization above). In the use of a time-axis and time-slots, the model is similar to the real-time extension of process algebra as given in [1]. In contrast to [1], our aim is *not* to give a general theory covering all types of mechanisms, *nor* to support a specific programming language.

Contents

Chapter 1

Introduction

1.1 Informal introduction to the model

The enabling model is a mathematical model that can be used to analyze the real-time behaviour of mechanisms. This can be done in abstract time-units, as well as in physical time units. Among others, it supports parallel composition and alphabet restriction.

Enabling structures describe the real-time behaviour, to be called *behaviour* in the sequel, of mechanisms that cooperate in a truly concurrent way with synchronization upon shared actions only. The behaviour of a mechanism is described in terms of point actions: (atomic) actions of which the performance has zero duration. The enabling of an action depends on the moments that other actions have been performed in a deterministic way. *Internal* actions of a mechanism are performed as soon as they are enabled by this mechanism. *External* actions —communications— of a mechanism may be delayed by its *environment*: they are performed as soon as they are enabled by *both*, the mechanism *and* its environment. There is no disabling: once enabled, an (external) action remains possible until it is performed.

The set of the behaviours (schedules) in which a mechanism can be involved is called its *process*. A mechanism is deterministic, so it cannot choose which schedule of its process is performed; this depends entirely on the moments at which the environment enables the mutual communications.

Evidently, not all 'real world' devices in the satisfy these conditions. However, some discrepancies between the model and the 'real world' can be bridged:

- Events in devices, such as communications, usually have a non-zero duration. Within the model, such events can be modeled by an action for initiation and an action for completion. When the duration of an event is known, it suffices to use an action for completion only; this convention is used in this thesis.

- In the model, timing of actions does depend on the scheduling of previous actions in a deterministic way. In devices, timing may also depend on other influences such as temperature, complexity of data, and the like. This problem can be solved by describing a device with a whole range of enabling structures, rather than with just one enabling structure. In some cases, such as the buffer with bypassing in Section 5.5, an analysis of the real-time behaviour of the device requires an analysis over this whole range. For devices in which timing is bounded by fixed delays between actions, or combinations of actions, it suffices to consider the extreme delays only (Section 3.6).

The informal programming notation we use has some resemblance with Pascal. It is akin to the language CP-0 in [2] and to the programming notation that is used in [23]. The use of question-marks for input, and exclamation-marks for output is as in [2, 11, 23]. The question-mark and the exclamation-mark give the direction of data only: communication with the environment happens simultaneously in the program and in the environment, as soon as *both* the program *and* its environment are ready for it.

Consider for example the following, very simple, program that inputs one value and outputs the square of this value:

Example 1.1 A disposable square element.

```
program square ( input in : integer , output out : integer ) :
var x : integer ;
begin
   in?x ; out!x^2
end.
```

This program performs three actions:

- first a value is received along *channel in*, it is stored in variable x,
- second, the square of x is computed,
- at last, the result of this computation, x^2, is sent along channel *out*.

Assume that the program is initiated at moment zero, that a communication takes one time unit, and that the computation of x^2 takes half a time unit. Let action a symbolize the completion of the input, b the completion of the computation, and c the completion of the output.

Obviously a and c are external actions, and b is an internal action.

The behaviour of the program is given by:

- a is enabled (by the program) at moment 1,
- b is enabled 0.5 time unit after performing a,
- c is enabled one time unit after performing b.

The program imposes restrictions on the moments that a, b, and c can be performed. A schedule s satisfies these restrictions if (and only if):

- $s.a \geqslant 1$: a is external so it may be delayed by the environment,
- $s.b = s.a + 0.5$: b is internal so it cannot be delayed by the environment,
- $s.c \geqslant s.b + 1$: c is external.

Whether a schedule s happens or not, does depend on the environment of the program; it depends in particular on the moments at which this environment enables the communications a and c. It may well be that the environment prohibits one of the actions a or c; in case it prohibits a, none of the actions a, b, and c will happen!

When duration 0.5 is only an upper bound of the duration of the computation of x^2, the enabling as described above gives an upper bound of the actual enabling.

□

1.2 Overview

In Chapter 2 we introduce the enabling model. In Section 2.1 we introduce among others the time-axis, schedules, and processes. In Section 2.2, enabling structures are introduced as a way to describe mechanisms; enabling functions describe the external behaviour of mechanisms. In the same section parallel composition is used to define the process of enabling structures. Renaming and scaling are introduced in Section 2.3. In Section 2.4 we introduce choice-free commands as a way to describe (a restricted type of) enabling functions. The behaviours of choice-free commands can be classified as having fixed delays, in Section 2.5 we discuss fixed delays in general. Alphabet restriction is introduced in Section 2.6. We consider enabling structures with the same behaviour to be equivalent (Section 2.7).

In Chapter 3 we are in quest of a relation that can be used for comparing performance. First we discuss what type of relation we want to use to express that one enabling function implements another (Section 3.1), in particular when the correctness concern is speed (Section 3.2). In Section 3.3 we introduce the implementation relation that suits us most. Section 3.4 is an interlude in which we discuss the existence of 'most liberal implementations' and 'most severe specifications'. In Section 3.5, we introduce a quantitative way to compare enabling functions. In particular, we introduce the relative speed of an 'implementation' with respect to a 'specification'. In Section 3.6 we discuss how variation of behaviour of an implementation affects its speed with respect to its specification. In Section 3.7 we conclude this chapter with some remarks on the utility of the introduced theory.

In Chapter 4 we use systolic arrays to illustrate the use of 'divide and conquer' during the computation, or estimation, of the external behaviour of mechanisms. Certain claims about infinite arrays in Section 4.2 are supported by the theory in Appendix C that deals with metric spaces. In Section 4.3 we give some examples of the application of systolic arrays. All examples are based on the same problem: computing the maximum of a fixed number of subsequent inputs.

In Chapter 5 we perform a case-study on distributed implementations of FIFO-buffers. In Section 5.2 we give theoretical bounds on the speed of such implementations, dependent on their 'shape'. In Sections 5.1 and 5.4 we present implementations that meet these bounds. These implementations have (at least) one drawback in common: they have a poor performance when they are almost empty. In Section 5.5 we introduce implementations that bypass empty parts, in order to improve this performance. These implementations are especially interesting because (drastic) choices about future behaviour are made dependent on the order in which communications happen. The chapter is concluded, in Section 5.6, with a brief discussion of computations in which 'buf-like' behaviour comes as a side-effect.

In Chapter 6 we conclude this thesis with a brief discussion on the scope of the enabling model, and of the type of comparison that is introduced in Chapter 3. In Sections 6.1 and 6.2 we discuss possible extensions of the expressive power and the manipulative power of the model respectively. Among others, we introduce serial composition of mechanisms. In Section 5.4 we metion some other points of interest.

In Appendix A we explain some notational conventions.

In Appendix B we discuss process description by means of {AND,OR}-causal compositions of elementary dependencies. It turns out that this gives equally powerful descriptions as are obtained by enabling structures. Some results of this appendix are used in Section 2.5.

In Appendix C we briefly discuss metric spaces. We mention a generalized version of Banach's contraction theorem. Furthermore, we apply the theory on some cases considering the enabling model.

Chapter 2

The Model

We use *enabling structures* to describe mechanisms. In an enabling structure, the enabling of subsequent actions depends on actions that were performed in the past. In this dependence relation we assume a positive *delay* between cause and effect. We distinguish between *internal* and *external* actions. Internal actions are strictly private, they cannot be shared with other mechanisms; we assume they are performed as soon as they are enabled. External actions can be shared with other mechanisms, thus scheduling of these actions may be delayed by the environment of the mechanism.

In Section 2.1 we introduce the time-domain, alphabets, schedules, and processes. For processes we define the operations of *parallel composition* and *projection*, which are defined for enabling structures in Sections 2.2 and 2.6 respectively.

In Section 2.2 we introduce dependence functions to describe the dependence of *one* action upon other actions; enabling structures to describe the behaviour of mechanisms; and enabling functions to describe the *external* behaviour of mechanisms. Furthermore we introduce the *history* of an enabling structure as its behaviour in a greedy environment, and the *process* of an enabling structure as the set of all behaviours it may exhibit, dependent of the environment. Parallel composition is introduced to describe concurrent cooperation; *masking* is introduced to hide external actions from the environment.

Renaming and scaling are introduced in Section 2.3.

In Section 2.4 we introduce *generic actions* as a way to describe sequences of actions 'of the same type'; for example communications along the same channel. Furthermore, choice-free commands are introduced as a way to describe (a certain type of) enabling functions over generic actions. A lot of programs have behaviours that can be described by choice-free commands.

The behaviours that are described by choice-free commands exhibit fixed delays between cause and effect. In Section 2.5 we discuss behaviours with fixed delays in general.

In Section 2.6 we introduce projection of enabling structures as abstraction from internal actions. Projection and masking are combined into *restriction*.

In Section 2.7 we discuss equivalence of enabling structures; enabling structures are considered equivalent when they exhibit the same behaviour. We introduce a normal form with respect to equivalence of external behaviour.

2.1 Preliminaries

An *alphabet* is a set of *actions*. In the sequel a through d range over actions, A, B, and C over alphabets. These are all 'variables', so it may well be that, for example, $a = b$. In concrete examples, actions with different names are implicitly assumed to be different. When no confusion is possible, we tend to abbreviate singleton alphabets by using a rather than $\{a\}$.

We use the real numbers as a reflection of the 'real-world' time-axis. The *time-domain* we use also captures the value ∞ (infinite); when an action is scheduled on moment ∞ this expresses that it is *not* scheduled. ∞ is chosen because $\infty > x$ for all real numbers x, so an expression like $s.c \geqslant s.b + 1$ (see Example 1.1) also captures the possibility of $s.c$ being infinite, which reflects that c is not scheduled at all.

Definition 2.1 The time-domain: T.

$$\mathsf{T} \;=\; \langle -\infty, \infty]$$

□

In the sequel M, N, and O range over the time-domain.

An advantage of the choice for real numbers plus ∞, above for example rational numbers, is the existence of least upper bounds of non-empty subsets of the time-domain and the existence of greatest lower bounds of a lot of sets. Furthermore, with the distance that is given in Appendix C, it is a complete metric space. The integers enjoy similar properties, but these are considered too course grained to reflect the 'real world' time axis.

The arithmetic of the real numbers extended with ∞ and $-\infty$ is defined straightforward in the cases it is obvious: $x + \infty = \infty$ for $x > -\infty$, $x * \infty = \infty$ for $x > 0$, $x * \infty = -\infty$ for $x < 0$, etc. .

A *schedule* is a mapping of actions in the time-domain. The only restriction we impose upon such a mapping is that is has a 'beginning', other than $-\infty$: mapping s given by $s.a_i = i$ (for $i \geqslant 0$) is a schedule with *base*, $\mathbf{b}s$, zero, but mapping t given by $t.a_i = -i$ (for $i \geqslant 0$) is not a schedule because it models an activity that has been going on forever ($\mathbf{b}t = -\infty$).

Definition 2.2 Schedule, base: $\mathbf{b}$.

For a function s in $A \to \mathsf{T}$ we define its *base* $\mathbf{b}s$ by:

$$\mathbf{b}s \;=\; (\,\mathbf{glb}\, a : a \in A : s.a\,)$$

A schedule over A is a function s as above with $\mathbf{b}s > -\infty$. The set of schedules over alphabet A is denoted by $\mathcal{S}.A$.

□

In the sequel, s through v range over schedules.

A schedule that models 'doing nothing' is called an *empty* schedule.

Definition 2.3 Empty schedules: ε_A, ε.

An *empty schedule* is a schedule that schedules all actions on ∞. The empty schedule over alphabet A is denoted by ε_A. When alphabet A is understood we just write ε instead of ε_A.

□

Mechanisms impose restrictions upon the scheduling of the actions they are involved with; they can be described by giving all schedules they allow to happen. As for schedules, we demand that the activity of a mechanism has a beginning (other than $-\infty$): for example, the process of the square element in Example 1.1 begins at 1. The proper way to describe the mechanism with alphabet A that does not allow scheduling of any actions on its alphabet, is with process $\{\varepsilon_A\}$, and *not* with $\varnothing$.

Definition 2.4 Schedule-set, process.

A *schedule-set* is a set of schedules over the same alphabet. The *base* of a schedule-set $\mathcal{P}$, denoted by $\mathbf{b}\mathcal{P}$, is defined by:

$$\mathbf{b}\mathcal{P} \;=\; (\,\mathbf{glb}\, s : s \in \mathcal{P} : \mathbf{b}s\,)$$

A *process* is a non-empty schedule-set $\mathcal{P}$ with base $\mathbf{b}\mathcal{P} > -\infty$.

□

In the sequel, $\mathcal{P}$ and $\mathcal{Q}$ range over processes, occasionally they are also used for schedule-sets.

Example 2.5 Processes.

$$\mathcal{P} \;=\; \{\{(a,2),(c,3)\},\{(a,5),(c,7)\},\{(a,11),(c,13)\}\}$$

$$\mathcal{Q} \;=\; (\,\mathbf{set}\, s : s \in \mathcal{S}.\{a,b,c\} \wedge s.a \geqslant 1 \wedge s.b = s.a + 0.5 \wedge s.c \geqslant s.b + 1 \\ : s\,)$$

Both $\mathcal{P}$ and $\mathcal{Q}$ are processes; $\mathbf{b}\mathcal{P} = 2$ and $\mathbf{b}\mathcal{Q} = 1$. $\mathcal{Q}$ is the process of the square element in Example 1.1.

□

As a general concept we introduce *the alphabet of* . For any 'X' it is denoted by $\mathbf{a}X$ and it is the set of actions X is involved in. For example $\mathbf{a}a = \{a\}$, $\mathbf{a}A = A$, and the alphabet of a process is the domain of its schedules.

We define restriction to actions as well as restriction in the time-domain.

Projection is just a domain-restriction of schedules and processes, it can be used to abstract from internal actions.

Definition 2.6 Projection: $\restriction$, $\backslash$.

For schedule s and alphabet A, the projection of s on A, $s \restriction A$, is the domain restriction of s to A: $\mathbf{a}(s \restriction A) = \mathbf{a}s \cap A$ and $(s \restriction A).a = s.a$.

Hiding is the complement of projection: $s \backslash A = s \restriction (\mathbf{a}s \backslash A)$.

For process $\mathcal{P}$ and alphabet A, the process $\mathcal{P} \restriction A$ is defined by:

$$\mathcal{P} \restriction A \;=\; (\,\mathbf{set}\; s : s \in \mathcal{P} : s \restriction A\,)$$

The convention for hiding is also used for projection of processes, and enabling structures.

□

Observe that alphabet restriction of a schedule (process) yields a schedule (process) and that (for X a schedule or a process) $\mathbf{b}(X \restriction A) \geqslant \mathbf{b}X$.

In the time-domain we define a kind of *prefixing*. s *until* M, denoted by $s \downharpoonleft M$, is in a way the *prefix of* s *until* M : actions that are not scheduled by s until moment M are scheduled on ∞ by $s \downharpoonleft M$, and scheduling on ∞ has been introduced as a mathematical trick to model 'not scheduling'.

Definition 2.7 Until: $\downharpoonleft$.

M *until* N, denoted by $M \downharpoonleft N$, is defined by:

$$\begin{array}{lcl} M \downharpoonleft N & = & \mathbf{if}\;\; M < N \;\rightarrow\; M \\ & & [\!]\;\; M \geqslant N \;\rightarrow\; \infty \\ & & \mathbf{fi} \end{array}$$

For schedules we define $s \downharpoonleft M$ as the schedule over $\mathbf{a}s$ that satisfies $(s \downharpoonleft M).a = s.a \downharpoonleft M$ (for $a \in \mathbf{a}s$). For processes we define

$$\mathcal{P} \downharpoonleft M \;=\; (\,\mathbf{set}\; s : s \in \mathcal{P} : s \downharpoonleft M\,)$$

□

The choice of the alternatives $M < N$ and $M \geqslant N$ (rather than $M \leqslant N$ and $M > N$), in the definition of $\downharpoonleft$, is *not* immaterial: Propositions 2.9.2 (first line), 2.10.2, 2.16, and 2.24, depend critically on this choice.

Example 2.8

For process $\mathcal{Q}$ in Example 2.5, $\mathcal{Q} \restriction \{a,c\} \downharpoonleft 2$ is equal to

$$(\,\mathbf{set}\; s : \mathbf{a}s = \{a,c\} \wedge (1 \leqslant s.a < 2 \vee s.a = \infty) \wedge s.c = \infty : s\,)$$

□

We mention the following (evident) properties of projection and prefixing:

Proposition 2.9 (without proof)

1 $s \restriction \varnothing \;=\; \varepsilon_\varnothing$, $\mathcal{P} \restriction \varnothing \;=\; \{\varepsilon_\varnothing\}$

And for X a schedule or a process:

$$\begin{array}{lcl} X \restriction \mathbf{a}X & = & X \\ X \restriction A \restriction B & = & X \restriction (A \cap B) \\ X \restriction A \downharpoonleft M & = & X \downharpoonleft M \restriction A \end{array}$$

2 $M \downharpoonright M \;=\; \infty \;,\quad s \downharpoonright \mathbf{b}s \;=\; \varepsilon \;,\quad \mathcal{P} \downharpoonright \mathbf{b}\mathcal{P} \;=\; \{\varepsilon\}$

and for X a moment in time, or a schedule, or a process:

$$\begin{aligned} X \downharpoonright \infty &= X \\ X \downharpoonright M \downharpoonright N &= X \downharpoonright (M \text{ min } N) \end{aligned}$$

□

Schedules over the same alphabet can be compared by using the partial order $\leqslant$ that is defined for functions. $s \leqslant t$ states that s schedules all actions at least as early as t. Projection and prefixing are monotonic with respect to this relation.

Proposition 2.10 (without proof)

1 (monotonicity of $\upharpoonright$ and $\downharpoonright$)

$s \leqslant t \;\Rightarrow\; s \upharpoonright A \leqslant t \upharpoonright A \;,\quad s \leqslant t \;\Rightarrow\; s \downharpoonright M \leqslant t \downharpoonright M$

2 (generalization of 1 for $\downharpoonright$)

$s \downharpoonright M \leqslant t \downharpoonright M \;\Leftrightarrow\; (\forall N : N < M : s \downharpoonright N \leqslant t \downharpoonright N)$

3 $M > (\mathbf{b}s \text{ min } \mathbf{b}t) \wedge s \downharpoonright M \leqslant t \downharpoonright M \;\Rightarrow\; \mathbf{b}s \leqslant \mathbf{b}t$

□

We introduce *parallel composition* of processes as a way to model mechanisms that synchronize via shared actions. Two mechanisms share an action when this action is a member of the alphabet of both.

In order to improve readability, we phrase most definitions, propositions, and proofs concerning parallel composition in terms of $\|$ as a binary operator. They can all be generalized to quantified expressions over $\|$ in a straightforward way. This includes a generalization to parallel composition of an infinite number of objects.

Definition 2.11 Parallel composition of processes: $\|$.

For processes $\mathcal{P}$ and $\mathcal{Q}$ we define their parallel composition $\mathcal{P} \| \mathcal{Q}$ by:

$\mathcal{P} \| \mathcal{Q} \;=\; (\mathbf{set}\, s : s \in \mathcal{S}(\mathbf{a}\mathcal{P} \cup \mathbf{a}\mathcal{Q}) \wedge s \upharpoonright \mathbf{a}\mathcal{P} \in \mathcal{P} \wedge s \upharpoonright \mathbf{a}\mathcal{Q} \in \mathcal{Q} : s)$

$\mathcal{P}$ and $\mathcal{Q}$ are *composable* if $\mathcal{P} \| \mathcal{Q}$ is a process.

□

The parallel composition of two processes is a schedule-set. For processes $\mathcal{P}$ and $\mathcal{Q}$ we have

$\mathbf{b}(\mathcal{P} \| \mathcal{Q}) \;\geqslant\; \mathbf{b}\mathcal{P} \text{ min } \mathbf{b}\mathcal{Q}$

(the composite cannot begin before one of the participants is ready to begin).

We infer that processes $\mathcal{P}$ and $\mathcal{Q}$ are composable if (and only if) $\mathcal{P} \| \mathcal{Q} \neq \varnothing$.

Parallel composition is generalized to an infinite number of processes in a straightforward way. However, when these processes are pairwise composable, this does not guarantee that they are composable as a set: the composite may be empty, or its base may be $-\infty$. An example of the latter phenomenon is the parallel composition of the processes $\{\{(a_i,-i)\}\}$ for $i \geqslant 0$.

In the sequel when giving a parallel composition, composability is implicitly assumed.

The unit element of parallel composition of processes is process $\{\varepsilon_\emptyset\}$: for any process $\mathcal{P}$, $\{\varepsilon_\emptyset\} \parallel \mathcal{P} = \mathcal{P}$. If it were allowed as a process, $\emptyset$ would be the zero element: $\emptyset \parallel \mathcal{P} = \emptyset$ for any process $\mathcal{P}$.

Example 2.12 Parallel composition of processes.

Let $\mathcal{P}$ and $\mathcal{Q}$ be as in Example 2.5. Their parallel composition is given by:

$$\mathcal{P} \parallel \mathcal{Q} \;=\; \{\{(a,5),(b,5.5),(c,7)\},\{(a,11),(b,11.5),(c,13)\}\}$$

□

Proposition 2.13 Parallel composition (without proof).

1 $(\mathcal{P} \parallel \mathcal{Q}) \restriction A \;=\; \mathcal{P} \restriction A \parallel \mathcal{Q} \restriction A \qquad \text{for } A \supseteq (\mathbf{a}\mathcal{P} \cap \mathbf{a}\mathcal{Q})$

$(\mathcal{P} \parallel \mathcal{Q}) \downharpoonleft M \;\subseteq\; \mathcal{P} \downharpoonleft M \parallel \mathcal{Q} \downharpoonleft M$

2 $\mathcal{P} \parallel \mathcal{Q} \;=\; \mathcal{P} \cap \mathcal{Q}$, if $\mathbf{a}\mathcal{P} = \mathbf{a}\mathcal{Q}$.

□

The $\subseteq$ in the comparison of prefixes, in the previous proposition, should intuitively be an equality. The following is a counter example:

Example 2.14

Define processes $\mathcal{P}$ and $\mathcal{Q}$ by:

$$\begin{aligned}\mathcal{P} &= \{\{(a,1),(b,3)\},\{(a,2),(b,3)\}\}\\ \mathcal{Q} &= \{\{(a,1),(b,3)\},\{(a,2),(b,4)\}\}\end{aligned}$$

and observe that

$$\begin{aligned}(\mathcal{P} \parallel \mathcal{Q}) \downharpoonleft 3 &= \{\{(a,1),(b,\infty)\}\}\\ \mathcal{P} \downharpoonleft 3 \parallel \mathcal{Q} \downharpoonleft 3 &= \{\{(a,1),(b,\infty)\},\{(a,2),(b,\infty)\}\}\end{aligned}$$

□

However, when composing enabling structures in parallel, the process of the resulting enabling structure does satisfy our intuition.

2.2 Enabling structures

Enabling structures describe mechanisms in terms of cause and effect. Moreover, they discriminate between external actions that can be shared with other mechanisms and internal actions that cannot. *Initiated* enabling structures are used to describe mechanisms relative to the moment of initiation, that is: the moment of initiation is the moment zero on the time-axis. To keep the mathematics simple, we assume a (universal) positive delay between cause and effect. When the environment of an enabling structure is *greedy*, each action is performed as soon as it is enabled; the schedule that is performed in this situation is called the *history* of the enabling structure. The process of an enabling structure is the set of schedules it may be engaged in when placed in any environment.

The concept of *similarity* is used to define the delay between cause and effect. The similarity of two objects (of the same type) is the maximal moment in time up to which they cannot be distinguished.

Definition 2.15 Similarity: sim.

For X and Y both in $\mathbf{T}$, both schedules, or both processes (over the same alphabet) we define:

$$\mathrm{sim}(X,Y) \;=\; (\,\mathbf{lub}\, M : X \downharpoonright M = Y \downharpoonright M : M\,)$$

furthermore for schedule s and process $\mathcal{P}$ over the same alphabet:

$$\mathrm{sim}(s,\mathcal{P}) \;=\; (\,\mathbf{lub}\, M : s \downharpoonright M \in \mathcal{P} \downharpoonright M : M\,)$$

□

In fact, the least upper bounds in the previous definition are maxima:

Proposition 2.16 (without proof)

For X, Y, s, and $\mathcal{P}$ as in Definition 2.15

$$X \downharpoonright \mathrm{sim}(X,Y) \;=\; Y \downharpoonright \mathrm{sim}(X,Y)$$

$$s \downharpoonright \mathrm{sim}(s,\mathcal{P}) \;\in\; \mathcal{P} \downharpoonright \mathrm{sim}(s,\mathcal{P})$$

□

Other properties of similarity are given by:

Proposition 2.17 (without proof)

1 $\mathrm{sim}(N,O) \;=\; \mathbf{if}\ N = O \rightarrow \infty \ [\!]\ N \neq O \rightarrow N \min O\ \mathbf{fi}$

2 For X and Y both schedules, or both processes, or a schedule and a process over the same alphabet:

$$\mathrm{sim}(X,Y) \;\geqslant\; \mathbf{b}X \min \mathbf{b}Y$$

3 For X and Y both schedules or both processes over the same alphabet with $\mathbf{b}X \neq \mathbf{b}Y$:

$$\mathrm{sim}(X,Y) \;=\; \mathbf{b}X \min \mathbf{b}Y$$

4 $\mathrm{sim}(s_0 \mathrm{\ lub\ } t_0, s_1 \mathrm{\ lub\ } t_1) \;\geqslant\; \mathrm{sim}(s_0,s_1) \min \mathrm{sim}(t_0,t_1)$

5 $\mathrm{sim}(s,t) \;=\; (\,\mathbf{glb}\, a : : \mathrm{sim}(s.a,t.a)\,)$

□

Let us first observe the enabling of one action. Assume we have a mechanism that 'enables action b half a time unit after performing a (Example 1.1). Apparently the enabling of b depends on a schedule. When function ϕ gives this dependence we have $\phi.s = s.a + \frac{1}{2}$ for the enabling of b. After scheduling of the 'cause' a, there elapses a 'causal' delay of $\frac{1}{2}$ before the 'effect' b is enabled. Since we assume a positive delay, the function $\phi.s = s.a - \frac{1}{2}$ is obviously not allowed. But what about the delay of the following functions?

$$\begin{array}{lcl}
\phi_0.s & = & s.a + 1 \text{ max } s.b + 1.5 \text{ min } s.c + 3 \\
\phi_1.s & = & \textbf{if} \;\; s.a < 1 \;\rightarrow\; s.a + 1 \\
 & & [\!]\;\; s.a = 1 \;\rightarrow\; \infty \\
 & & [\!]\;\; s.a > 1 \;\rightarrow\; 2 \\
 & & \textbf{fi} \\
\phi_2.s & = & s.a + s.b + 1 \\
\phi_3.s & = & s.a + s.a^{-2}
\end{array}$$

A function ϕ has a delay between cause and effect of (at least) Δ, when for any two schedules that are identical until moment M, their ϕ-image is identical until moment $M + \Delta$. The maximal value of Δ that satisfies for ϕ_0 and ϕ_1 is 1; the maximal values for ϕ_2 and ϕ_3 are $-\infty$ and zero respectively: ϕ_2 and ϕ_3 are no dependence functions.

Definition 2.18 Dependence function.

For functions $\phi \in S.A \rightarrow \mathbf{T}$, for any non-empty alphabet A, we define the *delay*, $\mathbf{d}\phi$, of ϕ as follows:

$$\mathbf{d}\phi \;=\; (\,\mathbf{glb}\, s,t : s,t \in S.A \wedge s \neq t : \text{sim}(\phi.s, \phi.t) - \text{sim}(s,t)\,)$$

A *dependence function* is such a function ϕ with $\mathbf{d}\phi > 0$.

□

A mechanism is described by giving its external alphabet, its internal alphabet, and the dependence functions for all actions. We choose to collect the dependence relations in one function ($\mathbf{f}E$).

Definition 2.19 Structure, $\mathbf{e}E$, $\mathbf{i}E$, $\mathbf{f}E$.

A *structure* is a triple $E = \langle A_e, A_i, X \rangle$ with $A_e \cap A_i = \emptyset$ and $X \in S.A \rightarrow S.A$, where $A = A_e \cup A_i$. The *external alphabet*, *internal alphabet*, and the *(functional) behaviour* of a structure E as above are defined respectively: $\mathbf{e}E \;=\; A_e$, $\mathbf{i}E \;=\; A_i$, $\mathbf{f}E \;=\; X$.

By convention, the alphabet of E is $\mathbf{a}E = \mathbf{e}E \cup \mathbf{i}E$. Though structures are not functions, in the sequel we will use them as such in two different ways. The first is with the convention $E.s.a = \mathbf{f}E.s.a$, the second with $E.a.s = \mathbf{f}E.s.a$ (both for $s \in S.\mathbf{a}E$ and $a \in \mathbf{a}E$). $E.s$ is the function that gives the enabling of all actions given a schedule s, $E.a$ gives the enabling of one action — a — only, as a function of the schedule.

□

Example 2.20 The square element revisited.

The dependence functions for a, b, and c for the square element of Example 1.1 are given by:

$$\phi_a.s \;=\; 1 \qquad \phi_b.s \;=\; s.a + 0.5 \qquad \phi_c.s \;=\; s.b + 1 \ .$$

Structure E, with external alphabet $\{a, c\}$, internal alphabet $\{b\}$, and functional behaviour as given below, can be used to describe the square element.

$$E.s.a \;=\; 1 \qquad E.s.b \;=\; s.a + 0.5 \qquad E.s.c \;=\; s.b + 1 \ .$$

In the definition of this functional behaviour, s is an arbitrary schedule over $\{a, b, c\}$. $E.s$ gives the moments at which actions are enabled, provided that the schedule is s. In Example 1.1 we already observed that not all schedules s can be performed by the square element: since internal actions happen as soon as they are enabled, and external actions may be delayed by the environment, only those schedules s that satisfy $s.b = E.s.b$, $s.a \geqslant E.s.a$, and $s.c \geqslant E.s.c$, can be performed. The process of the square element is the set of schedules that satisfy these restrictions (see Proposition 2.33). In the table below, we give three schedules, s, that are member of this process, and three schedules that are not. Which of the members of the process is actually performed depends on the moments at which the environment enables a and c.

s			$E.s$			Member of
$.a$	$.b$	$.c$	$.a$	$.b$	$.c$	Process
1	1.5	2.5	1	1.5	2.5	
2	2.5	4	1	2.5	3.5	Yes
∞	∞	∞	1	∞	∞	
$\underline{0}$	0.5	2	$\underline{1}$	0.5	1.5	
1	$\underline{1}$	4	1	$\underline{1.5}$	2	No [a]
1	$\underline{2}$	∞	1	$\underline{1.5}$	3	

Table Process of the square element.

[a] The underlined values are in conflict.

□

We do not consider all structures to be valid descriptions of mechanisms. As for dependence functions, we assume a positive delay between cause and effect (see also Proposition 2.22). Furthermore, we still assume that the activity of a mechanism has a beginning (other than $-\infty$). The actual way in which we define this beginning ($\mathbf{b}E$) is justified by Proposition 2.34.1 that states the equality of the base of an enabling structure and the base of its process.

Definition 2.21 Enabling structure, $\mathbf{b}E$, $\mathbf{d}E$.

The *base* and *delay* of a structure E are given by:

$$\begin{array}{lcl} \mathbf{b}E & \doteq & \mathbf{b}(E.\varepsilon) \\ \mathbf{d}E & = & (\,\mathbf{glb}\, s,t : s,t \in \mathcal{S}.\mathbf{a}E \wedge s \neq t : \mathrm{sim}(\mathbf{f}E.s, \mathbf{f}E.t) - \mathrm{sim}(s,t)\,) \end{array}$$

In case $\mathbf{a}E = \emptyset$, its delay is ∞. An *enabling structure* is a structure E with $\mathbf{b}E > -\infty$ and $\mathbf{d}E > 0$; a *closed* enabling structure is an enabling structure with an empty external alphabet. An *enabling function*, on the contrary, is an enabling structure with an empty internal alphabet. An *initiated* enabling structure (function), is an enabling structure (function) E with $\mathbf{b}E > 0$.

The set of enabling structures over (A_e, A_i) is denoted by $\mathcal{ES}(A_e, A_i)$, the set of all enabling structures is denoted by $\mathcal{ES}$. The set of enabling functions over A is denoted by $\mathcal{EF}.A$, the set of all enabling functions is denoted by $\mathcal{EF}$.

□

In the sequel E, F, and G range over enabling structures and e through h over enabling functions. $\mathcal{E}$ is used for sets of enabling structures, or enabling functions.

Proposition 2.22 (without proof)

For a structure E:

$$\mathbf{b}E \;=\; (\,\mathbf{glb}\, a : \; : E.a.\varepsilon\,) \quad \text{and} \quad \mathbf{d}E \;=\; (\,\mathbf{glb}\, a : \; : \mathbf{d}(E.a)\,)\,.$$

So for an enabling structure E, the functions $E.a$ are dependence functions that satisfy $\mathbf{d}(E.a) \geqslant \mathbf{d}E$.

□

The definition of similarity is extended to enabling structures.

Definition 2.23 Similarity of enabling structures.

For E and F enabling structures over the same alphabets we define:

$$\mathrm{sim}(E,F) \;=\; (\,\mathbf{lub}\, M : (\forall s : \; : E.s \downharpoonright M = F.s \downharpoonright M\,) : M\,)$$

□

As for the similarity of other objects (see Definition 2.15), the least upper bound turns out to be a maximum (compare to Proposition 2.16):

Proposition 2.24 (without proof)

For E and F as in Definition 2.23, and for s a schedule over their alphabet:

$$E.s \downharpoonright \mathrm{sim}(E,F) \;=\; F.s \downharpoonright \mathrm{sim}(E,F)$$

□

The most elementary enabling structures are those in which the enabling is independent of what actually happens: the *constant* enabling structures.

Definition 2.25 Constant, *Const*.

A function x is *constant on* X if (and only if)

$(\forall s,t : s,t \in X : x.s = x.t)$

A function is called *constant* if it is constant on its entire domain. The set of constant enabling functions is denoted by *Const*.

□

A constant enabling function is fully described by a schedule s that gives the moments at which actions are enabled. We introduce the following notation for such enabling functions.

Definition 2.26 Enabling function of a schedule: **n**.

The enabling function of schedule s, $\mathbf{n}s$, is the enabling function over $\mathbf{a}s$ defined by $\mathbf{n}s.t \;=\; s$, for all t over $\mathbf{a}s$.

For schedules over a singleton domain we use the shorthand $\mathbf{n}(a,M)$ rather than $\mathbf{n}\{(a,M)\}$.

□

The *history* of an enabling structure is the schedule that coincides with its enabling, it is the schedule performed by a mechanism that is not delayed by its environment. For example, the history of $\mathbf{n}s$ is s. The history of an enabling structure is uniquely defined because the behaviour of an enabling structure is a contraction on a complete metric space of schedules (see Appendix C). From a more operational point of view, the history is uniquely defined since enabling of actions does depend on the scheduling of actions in the past in a deterministic way.

Definition 2.27 The history of an enabling structure: **h**.

For an enabling structure E, the history $\mathbf{h}E$ of E is defined as the unique schedule $s \in \mathcal{S}.\mathbf{a}E$ that satisfies the 'history equation' $s = E.s$.

□

Proposition 2.28

1 For $s \in \mathcal{S}.\mathbf{a}E$: $\mathbf{b}s \geqslant \mathbf{b}E \;\Rightarrow\; \mathbf{b}(E.s) = \mathbf{b}E$

2 $\mathbf{b}\,\mathbf{h}E \;=\; \mathbf{b}E$

3 $\mathbf{h}\,\mathbf{n}s \;=\; s$

Proof of 1 and 2 (3 is evident).

1 Instead of 1 we prove, for $s \in S.\mathbf{a}E$:

$\mathbf{b}s \geqslant \mathbf{bh}E \;\Rightarrow\; \mathbf{b}(E.s) = \mathbf{bh}E$

Together with 2 this implies 1.

Let $s \in S.\mathbf{a}E$ with $\mathbf{b}s \geqslant \mathbf{b\,h}E$, we derive:

$$\begin{array}{ll}
 & \mathbf{true} \\
\Leftrightarrow & \quad \{\text{Proposition 2.17.2, } \mathbf{b}s \geqslant \mathbf{b\,h}E\ \} \\
 & \mathrm{sim}(\mathbf{h}E, s) \;\geqslant\; \mathbf{b\,h}E \\
\Rightarrow & \quad \{\text{definition of } \mathbf{d}\ (2.21)\} \\
 & \mathrm{sim}(E.\mathbf{h}E, E.s) \;\geqslant\; \mathbf{b\,h}E + \mathbf{d}E \\
\Leftrightarrow & \quad \{\text{definition of } \mathbf{h}\ (2.27)\} \\
 & \mathrm{sim}(\mathbf{h}E, E.s) \;\geqslant\; \mathbf{b\,h}E + \mathbf{d}E \\
\Leftrightarrow & \quad \{\text{definition of sim } (2.15)\} \\
 & \mathbf{h}E \downharpoonleft (\mathbf{b\,h}E + \mathbf{d}E) \;=\; E.s \downharpoonleft (\mathbf{b\,h}E + \mathbf{d}E) \\
\Rightarrow & \quad \{\text{Proposition 2.10.3, } \mathbf{d}E > 0\ \} \\
 & \mathbf{b\,h}E \;=\; \mathbf{b}(E.s)
\end{array}$$

2 Use the previous result and $\mathbf{b}E = \mathbf{b}(E.\varepsilon)$ (see definition of $\mathbf{b}$ (2.21)).

□

Before we introduce processes of enabling structures, we define parallel composition and *masking* of enabling structures. Parallel composition is used to describe concurrent mechanisms; masking is used to hide external actions from the environment. Masking is particularly useful when enabling structures are composed, and (shared) actions have to be hidden from the common environment.

When mechanisms are composed in parallel, shared actions are enabled in the composition as soon as they are enabled by all mechanisms that share them. In order to hide the case-analysis in the definition of parallel composition, we first introduce a generalization of lub for schedules.

Definition 2.29 Lub .

For schedules s and t, schedule s Lub t is the schedule over $\mathbf{a}s \cup \mathbf{a}t$ that is defined by:

$$\begin{array}{lcll}
(s \text{ Lub } t).a & = & \mathbf{if} & a \in \mathbf{a}s \setminus \mathbf{a}t \;\rightarrow\; s.a \\
 & & [\!] & a \in \mathbf{a}s \cap \mathbf{a}t \;\rightarrow\; s.a \max t.a \\
 & & [\!] & a \in \mathbf{a}t \setminus \mathbf{a}s \;\rightarrow\; t.a \\
 & & \mathbf{fi} &
\end{array}$$

□

Definition 2.30 Parallel composition of enabling structures: $\|$.

Two enabling structures E and F are *composable* if their internal alphabets are mutually private. That is, if $\mathbf{i}E \cap \mathbf{a}F = \varnothing$ and $\mathbf{i}F \cap \mathbf{a}E = \varnothing$. For composable E and F the parallel composition $E \parallel F$ is the enabling structure in $\mathcal{ES}(\mathbf{e}E \cup \mathbf{e}F, \mathbf{i}E \cup \mathbf{i}F)$ defined by:

$$(E \parallel F).s \;=\; E.(s \upharpoonright \mathbf{a}E) \text{ Lub } F.(s \upharpoonright \mathbf{a}F)$$

□

Observe that parallel composition of two (composable) enabling structures indeed yields an enabling structure and that

$$\mathbf{b}(E \parallel F) \geqslant \mathbf{b}E \min \mathbf{b}F \quad\text{and}\quad \mathbf{d}(E \parallel F) \geqslant \mathbf{d}E \min \mathbf{d}F .$$

Parallel composition of an infinite number of pairwise composable enabling structures (in a quantified expression) does not always yield an enabling structure: the resulting structure may have a base $-\infty$ or a delay of 0; this will also be treated as a case of non-composability. An example of both phenomena is the parallel composition of enabling functions e_i for $i > 0$ with $\mathbf{a}e_i = \{a_i, b_i\}$ and:

$$e_i.s.a_i \;=\; -i \qquad e_i.s.b_i \;=\; s.a_i + 1/i \qquad (\text{for } i > 0).$$

In the sequel, when performing parallel composition, composability is implicitly assumed.

Masking is very easy to define: it simply consists of moving actions from the external alphabet of an enabling structure to its internal alphabet.

Definition 2.31 Masking: $\shortparallel$.

$$E \shortparallel A \;=\; \langle \mathbf{e}E \cap A, \mathbf{i}E \cup (\mathbf{e}E \setminus A), \mathbf{f}E \rangle$$

□

The process $\mathbf{P}E$ of enabling structure E is the set of schedules it may engage in when placed in a closed connection to an environment:

$$\mathbf{P}E \;=\; (\cup F : \,:\, \mathbf{P}((E \parallel F) \shortparallel \varnothing) \upharpoonright \mathbf{a}E)$$

Because a closed enabling structure cannot be delayed by its environment, this can be rephrased into:

Definition 2.32 Process of an enabling structure: $\mathbf{P}$.

$$\mathbf{P}E \;=\; (\mathbf{set}\, F : \,:\, \mathbf{h}(E \parallel F) \upharpoonright \mathbf{a}E)$$

□

Members of the process of an enabling structure are also called the *(possible) behaviours* of this enabling structure.

The process of an enabling structure is non-empty because $\mathbf{h}E \in \mathbf{P}E$. Due to Proposition 2.34.1 the base of the process of an enabling structure is not $-\infty$. We conclude that the process of an enabling structure is (indeed) a process.

The following proposition reflects that the environment can delay the performance of external actions only.

Proposition 2.33

$$\mathbf{P}E \;=\; (\,\mathbf{set}\, s : \mathbf{a}s = \mathbf{a}E \;\wedge\; s \geqslant E.s \;\wedge\; s \upharpoonright \mathbf{i}E = E.s \upharpoonright \mathbf{i}E : s\,)$$

Proof We prove this proposition by mutual inclusion.

$\subseteq$ inclusion:

Let $s \in \mathbf{P}E$ and let F be composable with E such that $s = \mathbf{h}(E \parallel F) \upharpoonright \mathbf{a}E$ (see definition of $\mathbf{P}$ (2.32)). We derive:

$$
\begin{array}{ll}
 & s = \mathbf{h}(E \parallel F) \upharpoonright \mathbf{a}E \\
\Leftrightarrow & \quad \{\text{definitions of } \mathbf{h} \text{ (2.27) and } \parallel \text{ (2.30)}\} \\
 & (\forall a : a \in \mathbf{a}E \\
 & \quad : s.a \;=\; \begin{array}[t]{llll} \mathbf{if} & a \notin \mathbf{a}F & \rightarrow & E.s.a \\ [\!] & a \in \mathbf{a}F & \rightarrow & E.s.a \max F(\mathbf{h}(E \parallel F) \upharpoonright \mathbf{a}F).a \\ \mathbf{fi} & & & \end{array} \\
 &) \\
\Rightarrow & \\
 & s \geqslant E.s \;\wedge\; s \setminus \mathbf{a}F = E.s \setminus \mathbf{a}F \\
\Rightarrow & \quad \{\, \mathbf{a}F \cap \mathbf{i}E = \varnothing \,\} \\
 & s \geqslant E.s \;\wedge\; s \upharpoonright \mathbf{i}E = E.s \upharpoonright \mathbf{i}E
\end{array}
$$

$\supseteq$ inclusion:

Let $\mathbf{a}s = \mathbf{a}E$ and $s \geqslant E.s$ and $s \upharpoonright \mathbf{i}E = (E.s) \upharpoonright \mathbf{i}E$, define $F = \mathbf{n}(s \upharpoonright \mathbf{e}E)$.

From the definitions of $\mathbf{h}$ (2.27) and $\parallel$ (2.30) we infer that $\mathbf{h}(E \parallel F)$ is the unique schedule t satisfying (for all $a \in \mathbf{a}E$):

$$
t.a \;=\; \begin{array}[t]{llll} \mathbf{if} & a \in \mathbf{e}E & \rightarrow & s.a \max E.t.a \\ [\!] & a \in \mathbf{i}E & \rightarrow & E.t.a \\ \mathbf{fi} & & & \end{array}
$$

But s is also a solution of this equation. So $s = \mathbf{h}(E \parallel F)$. From $s = s \upharpoonright \mathbf{a}E$ and the definition of $\mathbf{P}$ (2.32) we infer $s \in \mathbf{P}E$.

□

Some other properties of processes of enabling structures are given by:

Proposition 2.34

1 $\mathbf{bP}E \;=\; \mathbf{b}E$

2 $(\mathbf{Pe}) \downharpoonright M \;\subseteq\; \mathbf{Pe}$

3 $\mathbf{P}\,\mathbf{n}s \;=\; (\,\mathbf{set}\, t : t \geqslant s : t\,)$

Proof of 1 and 2 (3 is evident).

1 $\mathbf{b}\,\mathbf{P}E \leqslant \mathbf{b}E$ follows from Proposition 2.28.2 and $\mathbf{h}E \in \mathbf{P}E$.

Remains to prove the $\geqslant$ part. Let $s \in \mathbf{P}E$; we derive:

$$\begin{array}{ll}
 & \mathbf{true} \\
\Rightarrow & \quad \{\ \mathbf{d}E > 0\ \} \\
 & \mathrm{sim}(E.s, E.\varepsilon) > \mathrm{sim}(s,\varepsilon) \\
\Leftrightarrow & \\
 & \mathrm{sim}(E.s, E.\varepsilon) > \mathbf{b}s \\
\Rightarrow & \quad \{\text{Proposition 2.17.3}\} \\
 & \mathbf{b}(E.s) = \mathbf{b}(E.\varepsilon) \ \ \vee\ \ \mathbf{b}(E.s) \min \mathbf{b}(E.e) > \mathbf{b}s \\
\Leftrightarrow & \quad \{\ s \in \mathbf{P}E \text{ and Proposition 2.33 give } \mathbf{b}s \geqslant \mathbf{b}(E.s)\ \} \\
 & \mathbf{b}(E.s) = \mathbf{b}(E.\varepsilon) \\
\Rightarrow & \quad \{\text{previous hint and definition of } \mathbf{b}\ (2.21)\} \\
 & \mathbf{b}s \geqslant \mathbf{b}E
\end{array}$$

2 We derive:

$$\begin{array}{ll}
 & s \in \mathbf{P}e \\
\Leftrightarrow & \quad \{\text{Proposition 2.33}\} \\
 & s \geqslant e.s \\
\Rightarrow & \quad \{\text{Proposition 2.10.1}\} \\
 & s \downharpoonleft M \ \geqslant\ e.s \downharpoonleft M \\
\Leftrightarrow & \quad \{\ \mathrm{sim}(s, s \downharpoonleft M) \geqslant M\ ,\ \mathbf{d}e > 0\ \} \\
 & s \downharpoonleft M \ \geqslant\ e(s \downharpoonleft M) \downharpoonleft M \\
\Rightarrow & \quad \{\text{definition of } \downharpoonleft (2.7)\} \\
 & s \downharpoonleft M \ \geqslant\ e.(s \downharpoonleft M) \\
\Leftrightarrow & \quad \{\text{Proposition 2.33}\} \\
 & s \downharpoonleft M \in \mathbf{P}e
\end{array}$$

□

The relation between parallel composition of enabling structures and parallel composition of processes is given in the following proposition:

Proposition 2.35

For composable enabling structures E and F: $\mathbf{P}(E \parallel F) \ = \ \mathbf{P}E \parallel \mathbf{P}F$

Proof

Let E and F as in the proposition, and let $s \in S(\mathbf{a}E \cup \mathbf{a}F)$. We derive:

$$\begin{array}{ll}
 & s \in \mathbf{P}(E \parallel F) \\
\Leftrightarrow & \quad \{\text{Proposition 2.33}\} \\
 & s \geqslant (E \parallel F).s \ \ \wedge\ \ s \restriction \mathbf{i}(E \parallel F) = (E \parallel F).s \restriction \mathbf{i}(E \parallel F) \\
\Leftrightarrow & \quad \{\text{definition of } \parallel\ (2.30)\} \\
 & s \restriction \mathbf{a}E \geqslant E.(s \restriction \mathbf{a}E) \restriction \mathbf{a}E \ \ \wedge\ \ s \restriction \mathbf{i}E = E.(s \restriction \mathbf{a}E) \restriction \mathbf{i}E \ \ \wedge \\
 & s \restriction \mathbf{a}F \geqslant F.(s \restriction \mathbf{a}F) \restriction \mathbf{a}F \ \ \wedge\ \ s \restriction \mathbf{i}F = F.(s \restriction \mathbf{a}F) \restriction \mathbf{i}F
\end{array}$$

$\Leftrightarrow$ { Proposition 2.33 }

$s \restriction \mathbf{a}E \in \mathbf{P}E \;\wedge\; s \restriction \mathbf{a}F \in \mathbf{P}F$

$\Leftrightarrow$ { definition of $\|$ (2.11) }

$s \in (\mathbf{P}E \,\|\, \mathbf{P}F)$

□

Unlike parallel composition, we have no direct relation between masking and an operation on processes. We mention the following (evident) properties:

Proposition 2.36 (without proof)

$$\mathbf{P}(E \shortparallel A) \;\subseteq\; \mathbf{P}E \qquad \mathbf{P}(E \shortparallel \varnothing) \;=\; \{\,\mathbf{h}E\,\}$$

□

The following example shows that when composing mechanisms, it is important that their behaviours fit.

Example 2.37 Deadlock.

In this example we compose the disposable square element of Example 2.20, with a disposable one-place buffer in a very inconvenient way.

The behaviour of the square element is given by enabling structure E in Example 2.20. The history of E is $\{(a,1),(b,1.5),(c,2.5)\}$, and its process is process $\mathcal{Q}$ of Example 2.5. There does not exist an enabling structure that has process $\mathcal{P}$ of the same example.

Consider the following program for a disposable one-place buffer:

```
program buffer( input in : integer , output out : integer ) :
var x : integer
begin
   in?x ; out!x
end.
```

Name the input-action of this buffer c, and the output-action a. Under the same assumptions as in Example 1.1, enabling function e over $\{a,c\}$ can be used to describe the real-time behaviour of this program.

$$e.s.c \;=\; 1 \qquad e.s.a \;=\; s.c+1$$

The process and the history of this buffer are given by:

$$\mathbf{Pe} \;=\; (\,\mathbf{set}\, s : \mathbf{a}s = \{a,c\} \wedge s.c \geqslant 1 \wedge s.a \geqslant s.c+1 : s\,)$$

$$\mathbf{he} \;=\; \{(c,1),(a,2)\}$$

We compose the buffer and the square element under hiding of both communications to the common environment: in the composition action a symbolizes the output of a value from the buffer to the square element, and action c symbolizes the output of a value from the square element to the buffer. The composite is given by enabling structure $(e \,\|\, E) \shortparallel \varnothing$.

$$\begin{aligned}((\mathrm{e} \parallel E) \shortparallel \varnothing).s.a &= s.c + 1 \max 1 \\ ((\mathrm{e} \parallel E) \shortparallel \varnothing).s.b &= s.a + 0.5 \\ ((\mathrm{e} \parallel E) \shortparallel \varnothing).s.c &= s.b + 1 \max 1\end{aligned}$$

It is left as an exercise for the reader to verify that $\mathbf{P}((\mathrm{e} \parallel E) \shortparallel \varnothing) = \{\varepsilon\}$: though both the square element and the buffer are willing to perform actions, the composite is not capable of doing anything. In [13] this is called lock, in [27] deadlock.

□

2.3 Renaming and scaling

The actions that are used in the description of mechanisms are just arbitrary names; when mechanisms are composed in parallel, *renaming* may be necessary to avoid name-clashes or to enforce name-clashes (communications). Where renaming is a transformation in the 'action domain', *scaling* is a linear transformation in the time domain. Scaling can be applied for two reasons. The first is to rewrite mechanisms into descriptions in an other time unit; for example μ-seconds instead of seconds. The other reason is that it allows quantitative comparison of mechanisms such as the statement 'this mechanism is twice as fast as that'.

In general, when composing predefined mechanisms the names of actions have to be adjusted in order to get the proper connections. Consider for example the square element and the buffer in Example 2.37. In order to avoid that the output of the buffer is fed back into the square element it suffices to replace all occurrences of a in the definition of e by d. A formal definition of renaming is given by:

Definition 2.38 Renaming.

A *renaming* $\mathcal{R}$ of A in B is a bijection in $A \to B$.

Renaming is extended to alphabets, schedules, and processes in a straightforward way: e.g. for $\mathcal{R} \in A \to B$ and $\mathbf{a}s \subseteq A$, renaming $\mathcal{R}.s$ is the schedule over $\mathcal{R}.\mathbf{a}s$ that satisfies: $\mathcal{R}.s.a = \mathcal{R}(s.a)$.

For a renaming $\mathcal{R} \in A \to B$ and an enabling structure E with $\mathbf{a}E \subseteq A$, the renaming $\mathcal{R}.E$ of E is the enabling structure in $\mathcal{ES}(\mathcal{R}.\mathbf{e}E, \mathcal{R}.\mathbf{i}E)$ defined by $\mathcal{R}.E.s = \mathcal{R}(E(\mathcal{R}^{-1}.s))$.

□

In the sequel $\mathcal{R}$ is a renaming. We also denote renaming by a subscript that indicates the renaming of individual actions. All actions in the domain that are not mentioned in the subscript are left unchanged: for example $a_{a\to b} = b$ and $c_{a\to b} = c$ for $c \neq a$. This notation leaves the domain of the renaming implicit.

By the way, renaming of enabling structures is what one would expect from renaming: $\mathbf{P}(\mathcal{R}.E) = \mathcal{R}.\mathbf{P}E$.

Example 2.39 Cascade of square element and buffer.

The square element and the one-place buffer of Example 2.37 can be composed by connecting the output of the square element to the input of the buffer. This connection can be described by renaming the output of the buffer from a into d.

The behaviour of the connection is described by $F = (E \parallel \mathrm{e}_{a \to d}) \shortparallel \{a, d\}$. It is left as an exercise to the reader to verify that

$$\begin{aligned} F.s.a &= 1 & F.s.b &= s.a + 0.5 \\ F.s.c &= s.b + 1 \max 1 & F.s.d &= s.c + 1 \end{aligned}$$

□

We introduce scaling as a linear transformation of the time-domain. For the behaviours in the previous example it is very simple to apply a scaling with, say, a factor 2; such a scaling just consists of multiplying all delays with a factor 2. For e this gives $(2 \odot \mathrm{e}).s.a = 2$ and $(2 \odot \mathrm{e}).s.b = s.a + 2$.

The formal definition of renaming seems not to contribute to the understanding of renaming (in practice, renaming simply consists of replacing occurrences of actions by occurrences of other actions). With scaling things are more complicated. We invite the reader to verify the following 'counter intuitive' example of scaling without consulting the formal definition of scaling.

Example 2.40 'Reverse scaling'.

Let e be the enabling function over $\{a, b\}$, defined by:

$$\mathrm{e}.s.a = 1 \qquad \mathrm{e}.s.b = 2 * s.a^2 + 1$$

Scaling of e with a factor 2 gives the following behaviour:

$$(2 \odot \mathrm{e}).s.a = 2 \qquad (2 \odot \mathrm{e}).s.b = s.a^2 + 2$$

This seems odd: *slowing down* of e by a factor 2, gives an enabling function with smaller delays for b. Observe though, that e is 'growing old' in the sense that the later a happens, the longer the delay until b is enabled. Since $2 \odot \mathrm{e}$ is half as fast as e, it does grow old at half speed.

□

Definition 2.41 Scaling.

A *scale* ρ is a linear function in $\mathbf{T} \to \mathbf{T}$. It is given by its *magnification* $\lambda : 0 < \lambda < \infty$ and its *translation* $\mu : -\infty < \mu < \infty$ as follows

$$\rho.M = \lambda * M + \mu$$

A scale ρ is called a *speeding up from* M when it has magnification at most 1 and $\rho.M \leqslant M$; it is called a *slowing down from* M when it has magnification at least 1 and $\rho.M \geqslant M$.

Scaling is extended to scales, schedules, and processes by

$$\begin{aligned} \rho.\sigma.M &= \rho(\sigma.M) \\ \rho.s.a &= \rho(s.a) \\ \rho.\mathcal{P} &= (\mathbf{set}\ s : s \in \mathcal{P} : \rho.s) \end{aligned}$$

For scale ρ and enabling structure E, the scaling $\rho.E$ of E is the enabling structure in $\mathcal{ES}(\mathbf{e}E, \mathbf{i}E)$ defined by $\rho.E.s = \rho(E(\rho^{-1}.s))$.

For x a schedule, a process, or an enabling structure, a scale is called a speeding up (slowing down) of x if (and only if) it is a speeding up (slowing down) from its base $\mathbf{b}x$.

The set of scales is denoted by SC. The set of slowing downs of (from) x is denoted by $SD.x$, and its set of speeding ups by $SU.x$.

We also denote scaling with the operators $\odot$ and $\oplus$ for magnification and translation respectively: e.g. $\lambda \odot s \oplus \mu$ instead of $\rho.s$ for scale ρ with magnification λ and translation μ.

□

In the sequel ρ and σ range over scales.

Similar to renaming, scaling of enabling structures is what one would expect from scaling: $\mathbf{P}(\rho.E) \;=\; \rho.\mathbf{P}E$.

The following proposition states that speeding up a speeding up gives a speeding up, and that slowing down a slowing down gives a slowing down.

Proposition 2.42 (without proof)

$$\rho \in SU.x \wedge \sigma \in SU(\rho.x) \;\Rightarrow\; (\rho.\sigma) \in SU.x$$

$$\rho \in SD.x \wedge \sigma \in SD(\rho.x) \;\Rightarrow\; (\rho.\sigma) \in SD.x$$

□

2.4 Generic actions and choice-free commands

The programming notation we use is partly based on *choice-free* commands (similar to the restricted commands in [27]). The primitive operations are catenation (;), parallel composition (,), and repetition ($*$). For example, the formula $(a\ ;\ b)^*$ describes a mechanism that alternately performs a actions and b actions; a and b may for example be names of channels along which the mechanism communicates with its environment. Since in the enabling model actions may happen at most once, we have to describe such behaviours in terms of *occurrences* of the *generic actions* that occur in the choice-free commands. In the sequel we frequently use enabling structures over (occurrences of) generic actions. We introduce the following

Convention 2.43 Enabling structures over generic actions.

- The set of occurrences of a generic action a is $(\,\mathbf{set}\, i : i \geqslant 0 : a_i\,)$.
- An alphabet must contain all occurrences of a generic action or none.
- An enabling structure must respect the order of occurrences of the same generic action. That is, for $s \in \mathbf{P}E$: $s.a_{i+1} > s.a_i$ or $s.a_{i+1} = \infty$.

- The notation of alphabets is abbreviated by just mentioning the generic actions. For example, the notation $\{a, b\}$ is used instead of $(\mathbf{set}\ i : i \geqslant 0 : a_i) \cup (\mathbf{set}\ i : i \geqslant 0 : b_i)$.

□

For the sake of completeness, we mention the 'independence condition':

$a_i = b_j \;\Leftrightarrow\; a = b \wedge i = j$

In choice-free commands, $S\,;T$ stands for S followed by T, S,T for S parallel T, S^* for an infinite repetition of S, and S^n for a repetition of n times S.

Definition 2.44 Choice-free command.

- ε is a choice-free command.
- a is a choice-free command.
- For S a choice-free command that has no infinite repetition, and T a choice-free command: $S\,;T$ is a choice-free command.
- For S and T choice-free commands that share no actions: S,T is a choice-free command.
- For S a choice-free command that has no infinite repetition: S^* and S^n are choice-free commands, for any natural n.

□

We assign the highest priority to the unary operators * and ^{n}, and the lowest priority to the semi-colon.

In the enabling function of a choice-free command we assume a unit delay of 1 between consecutive causally dependent actions, and we assume the first actions to be enabled at moment 1 on the time axis. This corresponds to the description of devices relative to the moment of initiation, where actions model the completion of events that have a duration of 1 time unit. Occasionally we assume deviating delays, instead of one time unit. Furthermore we sometimes use τ to denote an internal action. We do not give a formal definition of the enabling function $\mathbf{n}\,S$ of a choice-free command S, but we explain it informally by means of some examples.

Example 2.45 Enabling functions of choice-free commands.

- The enabling function of ε is the enabling function over the empty alphabet, $\mathbf{n}\,\varepsilon_\varnothing$.
- $\mathbf{n}\,a.s.a_i \;=\; \mathbf{if}\;\; i=0 \;\rightarrow\; 1 \;\;[]\;\; i>0 \;\rightarrow\; \infty \;\;\mathbf{fi}$
- $\mathbf{n}\,a^*.s.a_i \;=\; \mathbf{if}\;\; i=0 \;\rightarrow\; 1 \;\;[]\;\; i>0 \;\rightarrow\; s.a_{i-1}+1 \;\;\mathbf{fi}$

- $\mathrm{n}(a\ ;\ \tau)^* .s.a_i \ = \ \textbf{if}\ \ i=0 \ \rightarrow\ 1 \ \ [\!]\ \ i>0 \ \rightarrow\ s.a_{i-1}+2 \ \ \textbf{fi}$
- Under the assumption that a actions take ρ time units, the following enabling function corresponds to a^* :

  $e.s.a_i \ = \ \textbf{if}\ \ i=0 \ \rightarrow\ \rho \ \ [\!]\ \ i>0 \ \rightarrow\ s.a_{i-1}+\rho \ \ \textbf{fi}$
- $$\begin{array}{lll} \mathrm{n}\,a^n .s.a_i \ = & \textbf{if}\ \ i=0 \wedge i<n & \rightarrow\ 1 \\ & [\!]\ \ 0<i<n & \rightarrow\ s.a_{i-1}+1 \\ & [\!]\ \ i \geqslant n & \rightarrow\ \infty \\ & \textbf{fi} & \end{array}$$
- $$\begin{array}{lll} \mathrm{n}(a\ ;\ (a\,,b)^*).s.a_i \ = & \textbf{if}\ \ i=0 & \rightarrow\ 1 \\ & [\!]\ \ i=1 & \rightarrow\ s.a_0+1 \\ & [\!]\ \ i>1 & \rightarrow\ s.a_{i-1}\ \max\ s.b_{i-2}+1 \\ & \textbf{fi} & \end{array}$$

  $\mathrm{n}(a\ ;\ (a\,,b)^*).s.b_i \ = \ \mathrm{n}(a\ ;\ (a\,,b)^*).s.a_{i+1}$

□

2.5 Enabling with fixed delays

The behaviours that are described by choice-free commands have fixed delays between cause and effect. In this section we discuss enabling with fixed delays in general. At the end of the section we introduce *conservative* enabling structures. From the theory in Appendix B follows that these are exactly the enabling structures with fixed delays.

The first generalization we discuss is the description of mechanisms by means of *dependence relations*. Partial orders can be used to describe the 'trace processes' that are called *cubic* in [27], and *{AND}-causal* in [8]. Mazurkiewicz traces, [20], are partial orders. We introduce a slightly more general description of enabling functions by using *dependence relations*.

Definition 2.46 Dependence relations.

A binary relation R over an alphabet is the dependence relation of an enabling structure E over the same alphabet if

$E.s.a \ = \ (\,\mathbf{lub}\, b : b\,R\,a : s.b+1\,)\ ;\ \text{where}\ \ \mathrm{lub}\,\varnothing = 1\,.$

□

Example 2.47 Two dependence relations.

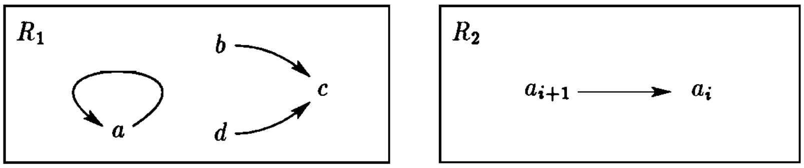


Figure Graphical representations of relations R_1 and R_2

1 Relation R_1 over $\{a,b,c,d\}$ is given by $a\,R_1\,a$, $b\,R_1\,c$, $d\,R_1\,c$, and **false** otherwise. The corresponding enabling function is given by

$$\begin{array}{lll} e.s.b = 1 & & e.s.a = s.a+1 \\ e.s.d = 1 & & e.s.c = s.b+1 \max s.d+1 \end{array}$$

2 Relation R_2 is given by $a_{i+1}\,R_2\,a_i$ for $i \geqslant 0$, and **false** otherwise. The corresponding enabling function is given by $e.s.a_i = s.a_{i+1}+1$.

Both enabling functions use a rather baroque way to state that a actions cannot happen.

□

All behaviours that can be described with choice-free commands can also be described with dependence relations. However, even for a relation over occurrences of generic actions there does not always exist a choice-free command with the same process.

Example 2.48 Dependence relations versus choice-free commands.

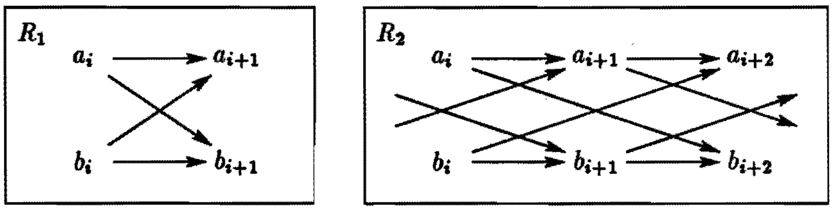


Figure Graphical representations of relations R_1 and R_2

1 Relation R_1 over $\{a,b\}$ is given by:

$a_i\,R_1\,a_{i+1}$ $b_i\,R_1\,b_{i+1}$ $a_i\,R_1\,b_{i+1}$ $b_i\,R_1\,a_{i+1}$ and **false** otherwise.

R_1 describes the same process as $(a,b)^*$.

2 Relation R_2 over $\{a,b\}$ is given by:

$a_i\,R_2\,a_{i+1}$ $b_i\,R_2\,b_{i+1}$ $a_i\,R_2\,b_{i+2}$ $b_i\,R_2\,a_{i+2}$ and **false** otherwise.

There is no choice-free command that describes the same process as R_2 .

□

Similar to choice-free commands, dependence relations can be generalized to describe behaviours with arbitrary (but fixed) delays. These generalized dependence relations correspond to the *event-rule systems* of [4]. We use this generalization in graphical representations only. As an example we give the dependencies for the square element in Figure 2.49. In this figure, $\perp$ symbolizes the moment *zero* on the time-axis.

$$\begin{tikzcd} \perp \arrow[r, "1"] & a \arrow[r, "0.5"] & b \arrow[r, "1"] & c \end{tikzcd}$$

Figure 2.49 Dependencies in the square element (see Example 2.20).

The behaviours that are described by (generalized) dependence relations are *AND-causal* behaviours: actions are enabled as soon as *all* preconditions are fulfilled. Take for example relation R_1 of Example 2.47; action c is enabled 1 time unit after *both* b and d are performed. In the dual type of dependency an action is enabled as soon as *at least one* precondition is fulfilled. Such an *OR-causal* dependency cannot be expressed with choice-free commands nor with dependence relations. The merge is a typical example of a specification (of data) with OR-causal dependencies.

Example 2.50 Synchronized merge of two input streams.

Consider the following specification for a program with input channels a and b, and output channel c:

$$\{\, c(2i), c(2i+1) \,\} \quad = \quad \{\, a(i), b(i) \,\} \quad \text{for all } i \geqslant 0 \,,$$

where $a(i)$ is the value that is received during communication a_i (similar for b) and $c(j)$ is the value that is sent to the environment during communication c_j.

A possible implementation of this specification is given by the following program:

```
program Merge₁(input a, output b) :
var va , vb ;
begin
    (a?va , b?vb ; c!va ; c!vb)*
end.
```

where $a?va$ denotes receipt of an input-value along channel a and storage of this value in variable va, and $c!va$ denotes output of the value of va along channel c (etc.). The program has communication behaviour $(a\,, b\ ;\ c\ ;\ c)^*$.

$$\begin{tikzcd}
 & a_i \arrow[dr] & & \\
c_{2i-1} \arrow[ur] \arrow[dr] & & c_{2i} \arrow[r] & c_{2i+1} \\
 & b_i \arrow[ur] & &
\end{tikzcd}$$

Figure Dependencies in $(a\,, b\ ;\ c\ ;\ c)^*$.

In fact, the previous program implements the following more restrictive specification:

$$c(2i) \;=\; a(i) \quad \wedge \quad c(2i+1) \;=\; b(i) \,.$$

However, according to the original specification, $c(2i)$ is allowed to be the first received of the values $a(i)$ and $b(i)$. This OR-causal dependence is reflected in the following enabling function.

$$\begin{aligned} e.s.a_i &= \textbf{if}\ \ i=0 \ \rightarrow\ 1 \ \ [] \ \ i>0 \ \rightarrow\ s.c_{2i-1}+1 \ \ \textbf{fi} \\ e.s.b_i &= e.s.a_i \end{aligned}$$

$$\begin{aligned} e.s.c_{2i} &= s.a_i \min s.b_i \\ e.s.c_{2i+1} &= s.a_i \max s.b_i \max s.c_{2i} \end{aligned}$$

In the graphical representation below, the arrows that give OR-causal dependence are labeled with a $\vee$.

$$\begin{tikzcd} & a_i \arrow[dr, "\vee"] \arrow[drr, bend left] & & \\ c_{2i-1} \arrow[ur] \arrow[dr] & & c_{2i} \arrow[r] & c_{2i+1} \\ & b_i \arrow[ur, "\vee"'] \arrow[urr, bend right] & & \end{tikzcd}$$

Figure Dependencies in e.

The following program notation for this version of the synchronized merge somewhat obscures the fact that a and b happen independently. When the a communication happens first, the first alternative is chosen; when the b communication happens first, the second alternative is chosen. When a and b happen simultaneously, either alternative can be chosen.

```
program Merge₂( input a, output b) :
var va , vb ;
begin
   (  a?va  →  b?vb , c!va ; c!vb
   [] b?vb  →  a?va , c!vb ; c!va
   )*
end.
```

□

In its most general form, an enabling structure with fixed delays may capture AND-causal, as well as OR-causal dependencies with arbitrary delays. The initial delay may even be negative.

Having fixed delays turns out to be equivalent with being *conservative* (Definition 2.53, Corollary 2.54). Before defining conservatism, we first consider the related, but simpler, notion of being *ascending*.

Definition 2.51 Ascending, *Asc*.

For x an enabling structure or a dependence function, and $\mathcal{P}$ a set of schedules over the same alphabet, x is *ascending on* $\mathcal{P}$ if (and only if)

$$(\forall s,t : s,t \in \mathcal{P} \wedge s \leqslant t : x.s \leqslant x.t)$$

An enabling structure, or dependence function, is called *ascending* if it is ascending on its entire domain. The set of ascending enabling functions is denoted by Asc. The set of enabling functions that are ascending on $\mathcal{P}$ is denoted by $Asc.\mathcal{P}$.

□

In the following example we show that not all enabling structures are ascending, and that not all ascending enabling functions have fixed delays.

Example 2.52

Enabling functions e and f are very similar. Since the delay between a and b may be 1 as well as 2, dependent on the moment at which a is performed, none of them has fixed delays. Enabling function e is not even ascending because postponing a may result in a faster enabling of b.

$$\begin{array}{lcl}
e.s.a & = & 1 \\
e.s.b & = & \textbf{if}\ s.a < 2 \rightarrow s.a + 2 \\
& & [\!]\ \ s.a \geqslant 2 \rightarrow s.a + 1 \\
& & \textbf{fi}
\end{array}
\qquad
\begin{array}{lcl}
f.s.a & = & 1 \\
f.s.b & = & \textbf{if}\ s.a < 2 \rightarrow s.a + 1 \\
& & [\!]\ \ s.a \geqslant 2 \rightarrow s.a + 2 \\
& & \textbf{fi}
\end{array}$$

□

Definition 2.53 Conservative, *Con*.

For x an enabling structure or a dependence function, and $\mathcal{P}$ a set of schedules over the same alphabet, x is *conservative on* $\mathcal{P}$ if (and only if)

$$(\forall s,t,\mu : s,t \in \mathcal{P} \wedge 0 \leqslant \mu < \infty \wedge s \leqslant t \oplus \mu : x.s \leqslant x.t \oplus \mu)$$

An enabling structure, or dependence function, is called *conservative* if it is conservative on its entire domain. The set of conservative enabling functions is denoted by *Con*.

□

From the definition we infer that being conservative implies being ascending (take $\mu = 0$).

From Theorem B.7.5 we infer the following corollary.

Corollary 2.54

An enabling structure has fixed delays if (and only if) it is conservative.

□

Apart from being a nice theoretical result, this corollary has also practical applications. The behaviours of programs without choice, for example those that are described by choice-free commands, have fixed delays. Furthermore, it will turn out in the next chapter that conservatism is useful property when comparing performance.

2.6 Alphabet restriction

Using parallel composition and masking, we are capable of describing parallel cooperation of mechanisms, including hiding of actions to the (common) environment. Every act of masking, however, adds new internal actions to the description of a mechanism. Since we are, usually, not interested in the internal behaviour of mechanisms, it is a good idea to have an operation that allows us to abstract from internal actions. For this purpose we introduce the *projection* of enabling structures. We even tend to describe mechanisms by their external behaviour only, by means of enabling functions. In this case, every act of masking must immediately be followed by an act of projection. We introduce *restriction* as the combination of both operations.

Removing internal actions from an enabling structure is possible because the environment has no direct influence on their performance, the moments they are performed depend on the moments other actions are performed only: for $A \supseteq \mathbf{e}E$ and $s \in \mathbf{P}E \restriction A$ there exists a unique $t \in \mathbf{P}E$ with $t \restriction A = s$. The projection $(E \restriction A).s$ is defined in terms of this t by $(E \restriction A).s \;=\; E.t \restriction A$.

Let, for example, s be a schedule over the external alphabet of enabling structure E of Example 2.37. The moment at which b is performed by E depends on the moment at which a is performed only: if schedule s is performed, then b is performed at moment $s.a + 0.5$. This knowledge can be used to compute the moment at which c is enabled, which is $s.b + 1 = s.a + 1.5$.

In order to obtain schedule t we define the *extrapolation of s with E*.

Definition 2.55 Extrapolation of schedules: $\uparrow$.

For s and E such that $\mathbf{a}s \subseteq \mathbf{a}E$, the *extrapolation of s with E*, $s \uparrow E$, is defined as the unique schedule t over $\mathbf{a}E$ that satisfies

$$t.a \;=\; \textbf{if}\;\; a \in \mathbf{a}s \;\rightarrow\; s.a \;\;[]\;\; a \notin \mathbf{a}s \;\rightarrow\; E.t.a \;\;\textbf{fi}$$

□

The argument for uniqueness of the extrapolation of a schedule is similar to the argument for uniqueness of the history of an enabling structure.

The first item of the following proposition states that $s \uparrow E$ is indeed the intended extrapolation. The second item gives an alternative characterization.

Proposition 2.56

For E and A such that $\mathbf{e}E \subseteq A$:

1 $s \in \mathbf{P}E \restriction A \;\Leftrightarrow\; s \uparrow E \in \mathbf{P}E$

2 $s \in \mathbf{P}E \restriction A \;\Leftrightarrow\; s \uparrow E = \mathbf{h}(\mathbf{n}(s \restriction \mathbf{e}E) \parallel E)$

Proof

1 The $\Leftarrow$ implication is evident. Remains to prove the $\Rightarrow$ implication.

When $s \in \mathbf{P}E \restriction A$ then there exists a $v \in \mathbf{P}E$ such that $v \restriction A = s$. This v satisfies trivially as a solution for t in the definition of $\uparrow$ (for $a \in \mathbf{a}s$ holds $v.a = s.a$, and for $a \notin \mathbf{a}s$ a is internal so $v.a = E.v.a$). Since this equation has a unique solution, this yields $v = s \uparrow E$, and thus $s \uparrow E \in \mathbf{P}E$.

2 The $\Leftarrow$ follows from the previous item. Remains to prove the $\Rightarrow$ implication.

$\mathbf{h}(\mathbf{n}(s \restriction \mathbf{e}E) \parallel E)$ is the unique schedule u over $\mathbf{a}E$ satisfying:

$u.a \;=\; \mathbf{if}\;\; a \in \mathbf{e}E \;\rightarrow\; E.u.a \max s.a \;\;[]\;\; a \in \mathbf{i}E \;\rightarrow\; E.u.a \;\;\mathbf{fi}$

Proceed with the same v as in the previous item.

□

Definition 2.57 Projection of enabling structures: $\restriction$.

For E and A such that $\mathbf{e}E \subseteq A$, the *projection of* E *on* A, $E \restriction A$, is the enabling structure in $\mathcal{ES}(\mathbf{e}E, \mathbf{i}E \cap A)$ defined by:

$(E \restriction A).s \;=\; E.(s \uparrow E) \restriction A$

□

From Proposition 2.58 we conclude that $\mathbf{b}(E \restriction A) \geqslant \mathbf{b}E$ and $\mathbf{d}(E \restriction A) \geqslant \mathbf{d}E$. Consequently, any projection of an enabling structure is indeed an enabling structure.

Proposition 2.58

For s, u, and ε over $A \cap \mathbf{a}E$:

1 $\mathbf{b}(E.(\varepsilon \uparrow E)) \;=\; \mathbf{b}E$

2 $\mathsf{sim}(s \uparrow E, u \uparrow E) \;\geqslant\; \mathsf{sim}(s, u)$

Proof of 1 (2 is evident).

Recall that $\mathbf{a}\varepsilon = A \cap \mathbf{a}E$.

Let $F = \langle A \cap \mathbf{a}E, \mathbf{i}E \setminus A, \mathbf{f}E \rangle$ (a kind of inverse masking). We derive:

$\mathbf{b}(\varepsilon \uparrow E)$

$=$

$\mathbf{b}(\varepsilon \uparrow F)$

$\geqslant$ $\{\; \varepsilon \in \mathbf{P}F \restriction \mathbf{e}F$ so $\varepsilon \uparrow F \in \mathbf{P}F$, Proposition 2.34.1 $\}$

$\mathbf{b}F$

$=$ $\{$ Proposition 2.28.2, $\mathbf{h}E = \mathbf{h}F$ $\}$

$\mathbf{b}E$

From Proposition 2.28.1 we infer $\mathbf{b}(E.(\varepsilon \uparrow E)) \;=\; \mathbf{b}E$.

□

The projection of enabling structures is closely related to the projection of processes:

Proposition 2.59

$\mathbf{P}(E \restriction A) \;=\; \mathbf{P}E \restriction A \qquad$ (for $A \supseteq \mathbf{e}E$)

Proof

Let E and A as above, and let $s \in \mathcal{S}.(A \cap \mathbf{a}E)$, we derive:

$s \in \mathbf{P}E \restriction A$

$\Leftrightarrow$ { Proposition 2.56.1 }
$s \uparrow E \in \mathbf{P}E$
$\Leftrightarrow$ { Proposition 2.33 }
$s \uparrow E \geqslant E.(s \uparrow E) \ \wedge \ s \uparrow E \restriction \mathrm{i}E = E.(s \uparrow E) \restriction \mathrm{i}E$
$\Leftrightarrow$ { definition of $\uparrow$ (2.55) }
$s \geqslant E.(s \uparrow E) \restriction (\mathbf{a}E \cap A) \ \wedge \ s \restriction \mathrm{i}E = E.(s \uparrow E) \restriction (\mathrm{i}E \cap A)$
$\Leftrightarrow$ { definition of $\restriction$ (2.57) }
$s \geqslant (E \restriction A).s \ \wedge \ s \restriction \mathrm{i}(E \restriction A) = (E \restriction A).s \restriction \mathrm{i}(E \restriction A)$
$\Leftrightarrow$ { Proposition 2.33 }
$s \in \mathbf{P}(E \restriction A)$

□

In the following example we give a formal derivation of the projection of enabling structure E from Example 2.37 on its external alphabet.

Example 2.60 External behaviour of the square element.

Let E be as in Example 2.20. For $s \in \mathcal{S}.\mathbf{e}E$ we derive:

$(E \restriction \mathbf{e}E).s$
$=$ { definition of $\restriction$ }
$E.(s \uparrow E) \restriction \mathbf{e}E$
$=$ { definitions of $\uparrow$ and E }
$E.\{ (a, s.a), (b, s.a + 0.5), (c, s.c) \} \restriction \mathbf{e}E$
$=$ { definition of E }
$\{ (a, 1), (b, s.a + 0.5), (c, s.a + 0.5 + 1) \} \restriction \mathbf{e}E$
$=$
$\{ (a, 1), (c, s.a + 1.5) \}$

□

Masking and projection are combined in restriction.

Definition 2.61 Restriction of enabling structures: $\Vert\!\restriction$.

For enabling structure E and alphabet A, the *restriction* of E to A, $E \Vert\!\restriction A$, is defined by $E \Vert\!\restriction A \ = \ (E \Vert A) \restriction A$.

□

We immediately draw the following conclusion (see Definitions 2.31 and 2.57):

Proposition 2.62 (without proof)

For any enabling structure E and alphabet A, $E \Vert\!\restriction A$, is the enabling structure in $\mathcal{ES}(\mathbf{e}E \cap A, \mathrm{i}E \cap A)$ given by:

$(E \Vert\!\restriction A).s \ = \ E.(s \uparrow E) \restriction A$

□

From Propositions 2.36 and 2.59 we infer

Proposition 2.63 (without proof)

$$\mathbf{P}(E \parallel\!\restriction A) \;\subseteq\; \mathbf{P}E \restriction A$$

□

The $\subseteq$ in this property may, in general, not be replaced by an equality; this is shown in the example below.

Example 2.64

Let enabling function e over $\{a,b\}$ be defined by:

$$\begin{array}{lcl} e.s.a & = & 1 \\ e.s.b & = & \textbf{if } s.a < 2 \;\rightarrow\; 4 \;\;[]\;\; s.a \geqslant 2 \;\rightarrow\; 3 \;\;\textbf{fi} \end{array}$$

It is left as an exercise to the reader to verify that

$$\begin{array}{lcl} \mathbf{P}(\mathrm{e} \parallel\!\restriction b) & = & (\,\textbf{set}\; s : s.b \geqslant 4 : s\,) \\ \mathbf{P}\mathrm{e} \restriction b & = & (\,\textbf{set}\; s : s.b \geqslant 3 : s\,) \end{array}$$

□

In Proposition 2.65 we give some mutual relations between parallel composition, masking, and restriction. The relations for parallel composition and masking are evident. It turns out that restriction behaves similar to masking. Keep in mind that the properties for $\parallel\!\restriction$ also hold for $\restriction$.

Proposition 2.65

1
$$\begin{array}{lcl} E \shortparallel A \shortparallel B & = & E \shortparallel (A \cap B) \\ E \shortparallel A \parallel\!\restriction B & = & E \parallel\!\restriction B \shortparallel A \\ E \parallel\!\restriction A \parallel\!\restriction B & = & E \parallel\!\restriction (A \cap B) \end{array}$$

2 (the history is a special case of masking)
$$\begin{array}{lcl} \mathbf{h}(E \shortparallel A) & = & \mathbf{h}E \\ \mathbf{h}(E \parallel\!\restriction A) & = & \mathbf{h}E \restriction A \end{array}$$

3 For E and F composable, and for $A \supseteq (\mathbf{e}E \cap \mathbf{e}F)$:
$$\begin{array}{lcl} (E \parallel F) \shortparallel A & = & E \shortparallel A \parallel F \shortparallel A \\ (E \parallel F) \parallel\!\restriction A & = & E \parallel\!\restriction A \parallel F \parallel\!\restriction A \end{array}$$

Proof

1 The first equality follows from the definition of masking (Definition 2.31); the second equality follows from Proposition 2.62. The third equality is proven below.

The alphabets of $E \shortparallel A \parallel\!\restriction B$ and $E \parallel\!\restriction B \shortparallel A$ are the same. Remains to prove equality of the behaviours.

Let s be a schedule over $\mathbf{a}E \cap A \cap B$.

First we derive $s \uparrow (E \Vert\!\!\upharpoonright A) \uparrow E \;=\; s \uparrow E$.

Let $t = s \uparrow (E \Vert\!\!\upharpoonright A)$ and $u = t \uparrow E$. We derive for $a \in \mathbf{a}E$:

$(s \uparrow (E \Vert\!\!\upharpoonright A) \uparrow E).a$

$=$

$u.a$

$=$ { definition of $\uparrow$ (2.55) }

if $a \in \mathbf{a}t \;\rightarrow\; t.a \;[\!]\; a \notin \mathbf{a}t \;\rightarrow\; E.u.a$ **fi**

$=$ { idem }

if $a \in \mathbf{a}t \;\rightarrow$ **if** $a \in \mathbf{a}s \;\rightarrow\; s.a \;[\!]\; a \notin \mathbf{a}s \;\rightarrow\; (E \Vert\!\!\upharpoonright A).t.a$ **fi**
$[\!]\; a \notin \mathbf{a}t \;\rightarrow\; E.u.a$
fi

$=$ { Proposition 2.62 }

if $a \in \mathbf{a}t \;\rightarrow$ **if** $a \in \mathbf{a}s \;\rightarrow\; s.a \;[\!]\; a \notin \mathbf{a}s \;\rightarrow\; E.u.a$ **fi**
$[\!]\; a \notin \mathbf{a}t \;\rightarrow\; E.u.a$
fi

$=$

if $a \in \mathbf{a}s \;\rightarrow\; s.a \;[\!]\; a \notin \mathbf{a}s \;\rightarrow\; E.u.a$ **fi**

$=$ { definition of $\uparrow$ }

$(s \uparrow E).a$

Next we derive:

$E \Vert\!\!\upharpoonright (A \cap B).s$

$=$ { Proposition 2.62 }

$E.(s \uparrow E) \upharpoonright (A \cap B)$

$=$ { result of previous derivation }

$E.(s \uparrow (E \Vert\!\!\upharpoonright A) \uparrow E) \upharpoonright A \upharpoonright B)$

$=$ { Proposition 2.62 }

$(E \Vert\!\!\upharpoonright A).(s \uparrow (E \Vert\!\!\upharpoonright A) \upharpoonright B)$

$=$ { idem }

$(E \Vert\!\!\upharpoonright A \Vert\!\!\upharpoonright B).s$

2 The first equality is evident; the second follows from Propositions 2.36, 2.59, and the second formula in the previous item.

3 The first formula is evident; the second one is proven below.

Let E , F , and A be as in the proposition. The alphabets of $(E \parallel F) \Vert\!\!\upharpoonright A$ and $E \Vert\!\!\upharpoonright A \parallel F \Vert\!\!\upharpoonright A$ are the same.
Remains to prove equality of the behaviours.

Let s be a schedule over $A \cap (\mathbf{a}E \cup \mathbf{a}F)$.

First we derive $s \uparrow (E \parallel F) \upharpoonright \mathbf{a}E \;=\; s \upharpoonright \mathbf{a}E \uparrow E$.

Let $t = s \uparrow (E \parallel F)$; we derive for $a \in \mathbf{a}E$:

$t.a$

$=$ { definition of $\uparrow$ (2.55) }

if $a \in \mathbf{a}s \;\rightarrow\; s.a \;[\!]\; a \notin \mathbf{a}s \;\rightarrow\; (E \parallel F).t.a$ **fi**

$=$ { definition of $\parallel$, $(\mathbf{a}E \setminus \mathbf{a}s) \cap \mathbf{a}F = \varnothing$ }

if $a \in \mathbf{a}s \;\rightarrow\; s.a \;[\!]\; a \notin \mathbf{a}s \;\rightarrow\; E.(t \restriction \mathbf{a}E).a$ **fi**

Conclude (from the definition of $\uparrow$) that

$s \uparrow (E \parallel F) \restriction \mathbf{a}E \;=\; s \restriction \mathbf{a}E \uparrow E$

On account of symmetry the same conclusion is drawn for F .

Next we derive:

$((E \parallel F) \Vert\!\restriction A).s$

$=$ { Proposition 2.62 }

$(E \parallel F).(s \uparrow (E \parallel F)) \restriction A$

$=$ { definition of of $\parallel$ }

$(\; E.(s \uparrow (E \parallel F) \restriction \mathbf{a}E) \;\;\text{Lub}\;\; F.(s \uparrow (E \parallel F) \restriction \mathbf{a}F)\;) \restriction A$

$=$ { see above }

$(\; E.(s \restriction \mathbf{a}E \uparrow E) \;\;\text{Lub}\;\; F.(s \restriction \mathbf{a}F \uparrow F)\;) \restriction A$

$=$

$E.(s \restriction \mathbf{a}E \uparrow E) \restriction A \;\;\text{Lub}\;\; F.(s \restriction \mathbf{a}F \uparrow F) \restriction A$

$=$ { Proposition 2.62 }

$(E \Vert\!\restriction A).(s \restriction \mathbf{a}E) \;\;\text{Lub}\;\; (F \Vert\!\restriction A).(s \restriction \mathbf{a}F)$

$=$

$(E \Vert\!\restriction A).(s \restriction \mathbf{a}(E \Vert\!\restriction A)) \;\;\text{Lub}\;\; (F \Vert\!\restriction A).(s \restriction \mathbf{a}(F \Vert\!\restriction A))$

$=$ { definition of $\parallel$ }

$(E \Vert\!\restriction A \parallel F \Vert\!\restriction A).s$

□

The projection of enabling structures does not always exactly yield the description one expects. In particular, the projection of an enabling structure may contain dependencies that seem redundant. Apart from being an exercise in computing projections, the following example serves as an illustration for this phenomenon. The notion of equivalence, that is introduced in the next section, provides a way to prune such redundant dependencies.

Example 2.66

Consider the dependence relations that are given in Example 2.48. Let e be the enabling function of $(a,c)^*$ ($R_{1\ b\rightarrow c}$), and let f be the enabling function of $(c,b)^*$ ($R_{1\ a\rightarrow c}$).

In this example we discuss the composition of e and f in which the communications along c are hidden to the common environment. The behaviour of this composition is given by $g = (e \parallel f) \Vert\!\upharpoonright \{a,b\}$. By intuition one can already tell that the behaviour of g is similar to R_2 (the difference between the number of a 's and the number of b 's that is performed is at most 2). Computation of this behaviour, however, gives a far more complicated result.

e $\parallel$ f is given by:

$$\begin{array}{lcl}
(e \parallel f).s.a_i & = & \textbf{if } i=0 \rightarrow 1 \\
 & & [\,]\ \ i>0 \rightarrow s.a_{i-1}+1 \text{ max } s.c_{i-1}+1 \\
 & & \textbf{fi} \\
(e \parallel f).s.b_i & = & \textbf{if } i=0 \rightarrow 1 \\
 & & [\,]\ \ i>0 \rightarrow s.b_{i-1}+1 \text{ max } s.c_{i-1}+1 \\
 & & \textbf{fi} \\
(e \parallel f).s.c_i & = & \textbf{if } i=0 \rightarrow 1 \\
 & & [\,]\ \ i>0 \rightarrow s.a_{i-1}+1 \text{ max } s.b_{i-1}+1 \\
 & & \qquad\qquad\ \ \text{max } s.c_{i-1}+1 \\
 & & \textbf{fi}
\end{array}$$

For s a schedule over $\{a,b\}$, the extrapolation $t = s \uparrow (e \parallel f)$ is given by:

$$\begin{array}{lcl}
t.a_i & = & s.a_i \\
t.b_i & = & s.b_i \\
t.c_i & = & (\,\textbf{max}\, j : 0 \leqslant j < i : s.a_j + i - j \text{ max } s.b_j + i - j\,) \text{ max } i+1
\end{array}$$

Applying Proposition 2.62 results in

$$\begin{array}{lcl}
g.s.a_i & = & (\,\textbf{max}\, j : 0 \leqslant j < i : s.a_j + i - j\,) \text{ max} \\
 & & (\,\textbf{max}\, j : 0 \leqslant j < i-1 : s.b_j + i - j\,) \text{ max } i+1
\end{array}$$

For b_i the enabling is similar (interchange a and b).

This is quite a baroque description of the behaviour of $(e \parallel f) \Vert\!\upharpoonright \{a,b\}$. Careful rewriting gives the following formula:

$$\begin{array}{lcl}
g.s.a_i & = & \textbf{if } i=0 \rightarrow 1 \\
 & & [\,]\ \ i=1 \rightarrow s.a_{i-1}+1\ \underline{\text{max } 1} \\
 & & [\,]\ \ i\geqslant 2 \rightarrow s.a_{i-1}+1 \text{ max } s.b_{i-2}+2 \\
 & & \qquad\qquad \underline{\text{max } (\,\textbf{max}\, j : 0 \leqslant j < i-1 : s.a_j + i - j\,)} \\
 & & \qquad\qquad \underline{\text{max } (\,\textbf{max}\, j : 0 \leqslant j < i-2 : s.b_j + i - j\,)} \\
 & & \qquad\qquad \underline{\text{max } i+1} \\
 & & \textbf{fi}
\end{array}$$

The dependency $s.a_{i-1}+1$ in the enabling of a_i imposes a delay of 1 time unit between successive a actions. Consequently, for any schedule in the process of g the number of time units between a_j and a_i, for $j < i$, is at least $i-j$. This observation makes the first underlined quantification redundant. For similar reasons the other underlined dependencies are redundant. Pruning the redundant dependencies gives the following alternative description of the behaviour of $(e \parallel f) \Vert\!\upharpoonright \{a,b\}$:

$$\begin{array}{rcll} h.s.a_i & = & \textbf{if} & i=0 \rightarrow 1 \\ & & [\!] & i=1 \rightarrow s.a_0+1 \\ & & [\!] & i \geqslant 2 \rightarrow s.a_{i-1}+1 \text{ max } s.b_{i-2}+2 \\ & & \textbf{fi} & \end{array}$$

(similar for b_i).

Except for a 'latency' of 2 time units between a and b actions and vice versa (instead of 1) this is the behaviour given by dependence relation R_2 (of Example 2.48).

□

The composition that is given in this example is similar to the cascade of FIFO buffers that is discussed in Chapter 5.

2.7 Equivalence of enabling structures

We consider enabling structures to be equivalent when they exhibit the same behaviour, in any environment. Equivalence is not only important because it allows to compare distinct mechanisms, it also allows to simplify descriptions of mechanisms. In Example 2.66, for example, we computed an enabling function with a lot of redundant dependencies, and mentioned the possibility of pruning them (without changing the actual behaviour). In the second part of this section we introduce a normal form with respect to equivalence of external behaviour.

The behaviour of an enabling structure is given by its process; the external behaviour is given by its process projected on the external alphabet. We introduce equivalence relations that state the equality of behaviour, and of external behaviour, up to a moment in time.

Definition 2.67 Equivalence of enabling structures: $\sim_M$ and $\approx_M$.

For $M \in \mathrm{T}$, we define equivalence relations $\sim_M$ and $\approx_M$ by:

$E \sim_M F \;\Leftrightarrow\; \mathbf{e}E = \mathbf{e}F \wedge \mathbf{i}E = \mathbf{i}F \wedge \mathbf{P}E \downharpoonright M = \mathbf{P}F \downharpoonright M$

$E \approx_M F \;\Leftrightarrow\; \mathbf{P}E \upharpoonright \mathbf{e}E \downharpoonright M = \mathbf{P}F \upharpoonright \mathbf{e}F \downharpoonright M$

$\sim$ and $\approx$ are abbreviations for $\sim_\infty$ and $\approx_\infty$ respectively.

□

The following relations between these equivalences are evident.

Proposition 2.68 (without proof)

$$E \sim_M F \;\Rightarrow\; E \approx_M F \qquad e \sim_M f \;\Leftrightarrow\; e \approx_M f$$

and for $M \geqslant N$:

$$E \sim_M F \;\Rightarrow\; E \sim_N F \qquad E \approx_M F \;\Rightarrow\; E \approx_N F$$

□

Enabling structures do not have to be identical to be $\sim$ equivalent. Take for example enabling structure E in Example 2.20. Because for all schedules s in its process $s.b = s.a + 0.5$, it is rather evident that the term $s.b + 1$ in the enabling of c may be replaced by $s.a + 1.5$. A generalization of this result is given in Corollary 2.71. This corollary is a consequence of the extension property. The extension property allows us to reason about the process of an enabling structure in terms of prefixes: the second part states when the prefix of a schedule is a member of the prefix of the process, it is a generalization of Proposition 2.33.

Proposition 2.69 Extension property.

1 Let $s \in \mathcal{S}.\mathbf{a}E$ and M such that:

$$s \downharpoonright M \geqslant E.s \downharpoonright M \;\wedge\; s \upharpoonright \mathrm{i}E \downharpoonright M = E.s \upharpoonright \mathrm{i}E \downharpoonright M$$

Define t over alphabet $\mathbf{a}E$ as the unique solution of

$$t.a \;=\; \mathbf{if}\;\; s.a < M \;\rightarrow\; s.a \;\;[]\;\; s.a \geqslant M \;\rightarrow\; M \max E.t.a \;\;\mathbf{fi}$$

Then t satisfies $t \downharpoonright M = s \downharpoonright M$ and $t \in \mathbf{P}E$. Furthermore t satisfies:

$$\begin{array}{lll} t.a \downharpoonright (M + \mathbf{d}E) & = & \mathbf{if}\;\; s.a < M \;\rightarrow\; s.a \\ & & []\;\; s.a \geqslant M \;\rightarrow\; M \max E.s.a \downharpoonright (M + \mathbf{d}E) \\ & & \mathbf{fi} \end{array}$$

2 $$\mathbf{P}E \downharpoonright M \;=\; (\,\mathbf{set}\, s : s \downharpoonright M \geqslant E.s \downharpoonright M \;\wedge\; s \upharpoonright \mathrm{i}E \downharpoonright M = E.s \upharpoonright \mathrm{i}E \downharpoonright M \;:\; s \downharpoonright M\,)$$

Proof

1 Let s , E and M as in the proposition.

In order to show uniqueness and existence of t we define e over $\mathbf{a}E$ by:

$$\begin{array}{lll} e.u.a & = & \mathbf{if}\;\; s.a < M \;\rightarrow\; s.a \\ & & []\;\; s.a \geqslant M \;\rightarrow\; M \max E.u.a \\ & & \mathbf{fi} \end{array}$$

e is an enabling function with $\mathbf{b}e \geqslant \mathbf{b}E$ and $\mathbf{d}e \geqslant \mathbf{d}E$. Furthermore, t is the unique fixed point, $\mathbf{h}e$, of the history equation, $e.u = u$.

$t \downharpoonright M = s \downharpoonright M$ is evident.

Furthermore we derive:

$t.a$

$\geqslant \quad \{\, s \downharpoonright M \geqslant E.s \downharpoonright M \,\}$

$\quad$ **if** $s.a < M \;\rightarrow\; E.s.a \;[\!]\; s.a \geqslant M \;\rightarrow\; M \max E.t.a$ **fi**

$\geqslant \quad \{\, E.s \downharpoonright M \;=\; E.t \downharpoonright M \,\}$

$\quad E.t.a$

and for $a \in \mathbf{i}E$:

$\quad t.a$

$= \quad \{\, s \upharpoonright \mathbf{i}E \downharpoonright M \;=\; E.s \upharpoonright \mathbf{i}E \downharpoonright M \,\}$

$\quad$ **if** $E.s.a < M \;\rightarrow\; E.s.a \;[\!]\; E.s.a \geqslant M \;\rightarrow\; M \max E.t.a$ **fi**

$= \quad \{\, E.s \downharpoonright M \;=\; E.t \downharpoonright M \,\}$

$\quad E.t.a$

From Proposition 2.33 we conclude $t \in \mathbf{P}E$.

The formula for $t \downharpoonright (M + \mathrm{d}E)$ is evident because $s \downharpoonright M = t \downharpoonright M$.

2 The $\supseteq$ inclusion is a direct result of 1. Remains to prove the $\subseteq$ part.
Let $s \in \mathbf{P}E \downharpoonright M$, let $t \in \mathbf{P}E$ with $t \downharpoonright M = s \downharpoonright M$, we derive:

$\quad t \in \mathbf{P}E$

$\Leftrightarrow \quad \{\text{Proposition 2.33}\}$

$\quad t \geqslant E.t \;\wedge\; t \upharpoonright \mathbf{i}E = E.t \upharpoonright \mathbf{i}E$

$\Rightarrow \quad \{\text{Proposition 2.10.1}\}$

$\quad t \downharpoonright M \geqslant E.t \downharpoonright M \;\wedge\; t \upharpoonright \mathbf{i}E \downharpoonright M = E.t \upharpoonright \mathbf{i}E \downharpoonright M$

$\Leftrightarrow \quad \{\, \mathrm{d}E > 0 ,\; s \downharpoonright M = t \downharpoonright M \,\}$

$\quad s \downharpoonright M \geqslant E.s \downharpoonright M \;\wedge\; s \upharpoonright \mathbf{i}E \downharpoonright M = E.s \upharpoonright \mathbf{i}E \downharpoonright M$

□

In Corollary 2.71 we use the following relations that compare the behaviours of enabling structures over a set of schedules. Relation $\leqslant_{\mathcal{P}}$ is a generalization of relation $\leqslant$ for functions.

Definition 2.70 $\leqslant_{\mathcal{P}}$ and $=_{\mathcal{P}}$.

For alphabets A and B such that $A \cap B = \varnothing$, and for $\mathcal{P}$ a set of schedules over $A \cup B$, we define relation $\leqslant_{\mathcal{P}}$ on $\mathcal{ES}(A, B)$ by:

$E \leqslant_{\mathcal{P}} F \;\Leftrightarrow\; (\forall s : s \in \mathcal{P} : E.s \leqslant F.s)$

$=_{\mathcal{P}}$ is defined as $\leqslant_{\mathcal{P}} \wedge \geqslant_{\mathcal{P}}$. For $\mathcal{P} = \mathcal{S}.(A \cup B)$ we write $\leqslant$ instead of $\leqslant_{\mathcal{P}}$, the corresponding $=_{\mathcal{P}}$ is exactly the equality.

□

Corollary 2.71

1 For E and F over the same alphabets: $E \sim F \;\Leftrightarrow\; E =_{\mathbf{P}E} F$

2 For e and f over the same alphabet: $\mathbf{P}e \supseteq \mathbf{P}f \;\Leftrightarrow\; e \leqslant_{\mathbf{P}e \cap \mathbf{P}f} f$

□

This corollary follows from the extension property, via the following proposition.

Proposition 2.72 Generalization of Corollary 2.71.

1 For E and F over the same alphabets:
$E \sim_M F \;\Leftrightarrow\; (\forall s : s \in \mathbf{P}E : E.s \downharpoonright M = F.s \downharpoonright M)$

2 $\mathbf{Pe} \downharpoonright M \supseteq \mathbf{Pf} \downharpoonright M \;\Leftrightarrow\; (\forall s : s \in (\mathbf{Pe} \cap \mathbf{Pf}) : e.s \downharpoonright M \leqslant f.s \downharpoonright M)$

Proof

1 Let E and F be enabling structures over the same alphabets.

1 sub $\Rightarrow$ implication.

It suffices to prove:

$\mathbf{P}E \downharpoonright N = \mathbf{P}F \downharpoonright N \;\Rightarrow\; (\forall s : s \downharpoonright N \in \mathbf{P}E \downharpoonright N : E.s \downharpoonright N \leqslant F.s \downharpoonright N)$

Assume the left-hand side, and let $s \downharpoonright N \in \mathbf{P}E \downharpoonright N$ and $b \in \mathbf{a}s$.

If $E.s.b \geqslant N$, the right-hand side is evident. Assume $E.s.b < N$ and let $M = E.s.b$. Observe that s and M satisfy in the premise in Proposition 2.69.1. Let t be the 'unique schedule' as given in Proposition 2.69.1, we derive:

$t \in \mathbf{P}E$

$\Rightarrow$ { assumption }

$t \downharpoonright N \in \mathbf{P}F \downharpoonright N$

$\Rightarrow$ { Proposition 2.69.2 }

$F.t.b \downharpoonright N \leqslant t.b \downharpoonright N$

$\Leftrightarrow$ { $t.b = M < N$ }

$F.t.b \leqslant M$

$\Leftrightarrow$ { $s \downharpoonright M = t \downharpoonright M$ }

$F.s.b \leqslant M$

$\Leftrightarrow$ { $M = E.s.b$ }

$F.s.b \leqslant E.s.b$

1 sub $\Leftarrow$ implication. We give a proof by induction.

base For $M \leqslant \mathbf{b}E \min \mathbf{b}F$ the left-hand side holds trivially.

step Let $M < \infty$, assume the $\Leftarrow$ implication holds for M.

Let $\Delta : 0 < \Delta \leqslant (\mathbf{d}E \min \mathbf{d}F)$, we prove the $\Leftarrow$ implication for $M + \Delta$.

Assume the right-hand side for $M + \Delta$; first observe:

$(\forall s : s \in \mathbf{P}E : E.s \downharpoonright (M + \Delta) = F.s \downharpoonright (M + \Delta))$

$\Rightarrow$

$(\forall s : s \in \mathbf{P}E : E.s \downharpoonright M = F.s \downharpoonright M)$

$\Rightarrow$ { assumption for M }

$\mathbf{P}E \downharpoonright M = \mathbf{P}F \downharpoonright M$

For $s:\ s \downharpoonright M \in \mathbf{P}E \downharpoonright M$ we derive $E.s \downharpoonright (M+\Delta) = F.s \downharpoonright (M+\Delta)$. From Proposition 2.69.2 follows $\mathbf{P}E \downharpoonright (M+\Delta) = \mathbf{P}F \downharpoonright (M+\Delta)$.

Let s as above, and let $t \in \mathbf{P}E$ such that $s \downharpoonright M = t \downharpoonright M$, we derive:

$$
\begin{array}{ll}
 & E.s \downharpoonright (M+\Delta) \\
= & \quad \{\ s \downharpoonright M = t \downharpoonright M \text{ and } \mathbf{d}E \geqslant \Delta\ \} \\
 & E.t \downharpoonright (M+\Delta) \\
= & \quad \{\ t \in \mathbf{P}E, \text{ right-hand side for } M+\Delta\ \} \\
 & F.t \downharpoonright (M+\Delta) \\
= & \quad \{\ s \downharpoonright M = t \downharpoonright M \text{ and } \mathbf{d}F \geqslant \Delta\ \} \\
 & F.s \downharpoonright (M+\Delta)
\end{array}
$$

2 Let e and f be enabling functions over the same alphabet, we derive:

$$
\begin{array}{ll}
 & (\forall s : s \in \mathbf{P}e \cap \mathbf{P}f : e.s \downharpoonright M \leqslant f.s \downharpoonright M\ ) \\
\Leftrightarrow & \quad \{\text{Proposition 2.13.2}\} \\
 & (\forall s : s \in \mathbf{P}e \parallel \mathbf{P}f : e.s \downharpoonright M \leqslant f.s \downharpoonright M\ ) \\
\Leftrightarrow & \quad \{\text{Proposition 2.35}\} \\
 & (\forall s : s \in \mathbf{P}(e \parallel f) : e.s \downharpoonright M \leqslant f.s \downharpoonright M\ ) \\
\Leftrightarrow & \quad \{\text{definition of } \parallel \ (2.30)\} \\
 & (\forall s : s \in \mathbf{P}(e \parallel f) : (e \parallel f).s \downharpoonright M = f.s \downharpoonright M\ ) \\
\Leftrightarrow & \quad \{\text{previous item}\} \\
 & \mathbf{P}(e \parallel f) \downharpoonright M \ = \ \mathbf{P}f \downharpoonright M \\
\Leftrightarrow & \quad \{\text{as above}\} \\
 & (\mathbf{P}e \cap \mathbf{P}f) \downharpoonright M \ = \ \mathbf{P}f \downharpoonright M \\
\Leftrightarrow & \\
 & \mathbf{P}e \downharpoonright M \ \supseteq \ \mathbf{P}f \downharpoonright M
\end{array}
$$

□

Corollary 2.71 provides a way to prune redundant dependencies in enabling structures. Example 2.73 (see below) illustrates that not all dependencies that seem redundant may (formally) be pruned. In Section 6.1 we show that one should be careful, when formalizing *liberal delay conditions* that allow to prune such dependencies. In the mean time, we will occasionally use them in order to get more concise descriptions.

Example 2.73 Adaptive ordering.

We consider a program that communicates with the environment along channels a and b. The first communications are performed independently; for all other communications the same order is enforced as for the first communications:

```
$a, b$ ; if    $a_0$ before $b_0$   $\rightarrow$  $(a\ ;\ b)^*$
         []  $\neg(a_0$ before $b_0)$  $\rightarrow$  $(b\ ;\ a)^*$
         fi
```

Under the assumption of unit delays, enabling function e describes the behaviour of this program.

```
$e.s.a_i$  =  if  $i = 0$ $\rightarrow$ 1
              []  $i > 0$ $\rightarrow$ if  $s.a_0 < s.b_0$   $\rightarrow$  $s.b_{i-1} + 1$
                               []  $s.a_0 \geqslant s.b_0$   $\rightarrow$  $s.b_i + 1$
                               fi
              fi  $\underline{\max\ s.a_0 + 1}$
$e.s.b_i$  =  if  $i = 0$ $\rightarrow$ 1
              []  $i > 0$ $\rightarrow$ if  $s.a_0 < s.b_0$   $\rightarrow$  $s.a_i + 1$
                               []  $s.a_0 \geqslant s.b_0$   $\rightarrow$  $s.a_{i-1} + 1$
                               fi
              fi  $\underline{\max\ s.b_0 + 1}$
```

Except for the underlined dependencies, this enabling function follows from the program in a straightforward way. Without these redundant dependencies, however, e would not be an enabling function. This phenomenon is illustrated as follows: let f be the result of pruning the underlined dependencies in e, let $M < \infty$, and let s and t be defined by:

$$\begin{array}{lllll} s.a_1 & = & t.a_1 & = & 1 \\ s.a_0 & = & t.b_0 & = & M \\ s.b_0 & = & t.a_0 & = & M+1 \quad \text{and all other actions scheduled on } \infty . \end{array}$$

The similarity of s and t is M. Since $f.s.b_1 = 2$ and $f.t.b_1 = M + 2$, the similarity of $f.s$ and $f.t$ is at most two.

f, however, uniquely describes the behaviour of the program: for all schedules in **P**e it is identical to e. f can be considered an enabling function under 'liberal delay conditions'. The advantage of f above e is the absence of redundant, and rather arbitrary, dependencies.

□

Proposition 2.74 states that the equivalence relations behave as they should behave with respect to the operations on enabling structures that are given in the previous sections.

Proposition 2.74

1 Relations $\sim_M$ are congruences with respect to $\|$, $\shortparallel$, $\restriction$, and $\|\!\restriction$.

2 Relations $\approx_M$ are congruences with respect to $\shortparallel$, $\restriction$, and $\|\!\restriction$.

□

The proof of this proposition is postponed until after Proposition 2.77.

The relation between $\approx_M$ and parallel composition is more subtle: for composable enabling structures E and F and enabling structures E' and F' such that $E \approx E'$ and $F \approx F'$, the latter two may share internal actions, in which case they are not composable. Observe though, that the names of internal actions do not affect the external behaviour of an enabling structure. Therefore, a solution is to define parallel composition of equivalence classes.

Definition 2.75 Equivalence class modulo internal renaming: $[E]_R$.

$$[E]_R \quad = \quad (\textbf{set}\, \mathcal{R} : \mathcal{R} \text{ is an internal renaming of } E : \mathcal{R}.E)$$

where an internal renaming of E is a renaming $\mathcal{R}$ on $\mathbf{a}E$ such that $\mathcal{R}.a = a$ for $a \in \mathbf{e}E$.

□

Definition 2.76 Parallel composition of $[\;]_R$ classes.

$$[E]_R \parallel [F]_R \quad = \quad [\mathcal{R}_E.E \parallel \mathcal{R}_F.F]_R$$

for internal renamings $\mathcal{R}_X$ of X such that $\mathcal{R}_E.E$ and $\mathcal{R}_F.F$ are composable.

□

It is easily verified that this is a proper definition (that is: independent of which internal renamings are chosen).

Proposition 2.74 can be extended with:

Proposition 2.74.2a

$\approx_M$ is a congruence with respect to parallel composition of $[\;]_R$ classes.

□

The proof of this proposition is postponed until after Proposition 2.77.

A consequence of $\sim$ and $\approx$ being congruences with respect to masking, is the following behaviour of the equivalences with respect to the history of enabling structures.

Proposition 2.77 Equivalence and $\mathbf{h}$ (without proof).

$$E \sim_M F \quad \Rightarrow \quad \mathbf{h}E \downharpoonright M = \mathbf{h}F \downharpoonright M$$

$$E \approx_M F \quad \Rightarrow \quad \mathbf{h}E \downharpoonright M \upharpoonright \mathbf{e}E = \mathbf{h}F \downharpoonright M \upharpoonright \mathbf{e}E$$

□

Proof of Proposition 2.74, including 2.74.2a .

1 sub parallel composition.

We derive for E_i and F_i composable:

$$E_0 \sim_M E_1 \quad \wedge \quad F_0 \sim_M F_1$$

$\Leftrightarrow$ $\quad$ { Proposition 2.72.1 }

$$\begin{aligned}
&(\forall s : s \in \mathbf{P}E_0 : E_0.s \downharpoonright M \; = \; E_1.s \downharpoonright M) \;\wedge \\
&(\forall s : s \in \mathbf{P}F_0 : F_0.s \downharpoonright M \; = \; F_1.s \downharpoonright M) \\
\Rightarrow &\quad \{ (\mathcal{P} \parallel \mathcal{Q}) \upharpoonright \mathbf{a}\mathcal{P} \subseteq \mathcal{P} \} \\
&(\forall s : s \in (\mathbf{P}E_0 \parallel \mathbf{P}F_0) : E_0(s \upharpoonright \mathbf{a}E_0) \downharpoonright M \; = \; E_1(s \upharpoonright \mathbf{a}E_1) \downharpoonright M \;\wedge \\
&\qquad\qquad\qquad\qquad F_0(s \upharpoonright \mathbf{a}F_0) \downharpoonright M \; = \; F_1(s \upharpoonright \mathbf{a}F_1) \downharpoonright M) \\
\Rightarrow &\quad \{ \text{Definition 2.30, Proposition 2.35} \} \\
&(\forall s : s \in \mathbf{P}(E_0 \parallel F_0) : (E_0 \parallel F_0).s \downharpoonright M \; = \; (E_1 \parallel F_1).s \downharpoonright M) \\
\Leftrightarrow &\quad \{ \text{Proposition 2.72.1} \} \\
&E_0 \parallel F_0 \;\; \sim_M \;\; E_1 \parallel F_1
\end{aligned}$$

1 sub masking.

The proof is similar as for parallel composition, use $\mathbf{P}(E \shortparallel A) \subseteq \mathbf{P}E$ (Proposition 2.36).

1 sub projection.

Let E and F be enabling structures over the same alphabets, let A be an alphabet such that $\mathbf{e}E \subseteq A$.

From $\sim_M$ being a congruence for parallel composition and masking, and from Proposition 2.56.2, and Proposition 2.36 (second formula), we conclude the first step in the following derivation.

$$\begin{aligned}
&E \sim_M F \\
\Leftrightarrow &\quad \{ \text{see above} \} \\
&E \sim_M F \;\;\wedge\;\; (\forall s : s \in \mathbf{P}E \upharpoonright A : s \uparrow E \downharpoonright M \; = \; s \uparrow F \downharpoonright M) \\
\Rightarrow &\quad \{ \text{Propositions 2.72.1 and 2.56.1} \} \\
&(\forall s : s \in \mathbf{P}E \upharpoonright A : E(s \uparrow E) \downharpoonright M \; = \; F(s \uparrow F) \downharpoonright M) \\
\Rightarrow &\quad \{ \text{definition of } \upharpoonright \text{ (2.57)} \} \\
&(\forall s : s \in \mathbf{P}E \upharpoonright A : (E \upharpoonright A).s \downharpoonright M \; = \; (F \upharpoonright A).s \downharpoonright M) \\
\Leftrightarrow &\quad \{ \text{Proposition 2.72.1} \} \\
&E \upharpoonright A \sim_M F \upharpoonright A
\end{aligned}$$

1 sub restriction: use masking and projection.

2 We give the proof for parallel composition (2.74.2a) only; the other proofs are similar. It suffices to give the proof for composable enabling structures E_i and F_i.

We derive:

$$\begin{aligned}
&E_0 \approx_M F_0 \;\;\wedge\;\; E_1 \approx_M F_1 \\
\Leftrightarrow &\quad \{ \text{Definition 2.67, Proposition 2.59} \} \\
&E_0 \upharpoonright \mathbf{e}E_0 \sim_M F_0 \upharpoonright \mathbf{e}F_0 \;\;\wedge\;\; E_1 \upharpoonright \mathbf{e}E_1 \sim_M F_1 \upharpoonright \mathbf{e}F_1
\end{aligned}$$

$\Leftrightarrow$ $\{\sim_M$ is a congruence for parallel composition $\}$

$(E_0 \restriction \mathbf{e}E_0 \parallel E_1 \restriction \mathbf{e}E_1) \;\sim_M\; (F_0 \restriction \mathbf{e}F_0 \parallel F_1 \restriction \mathbf{e}F_1)$

$\Leftrightarrow$ $\{$Proposition 2.65.3$\}$

$(E_0 \parallel E_1) \restriction \mathbf{e}(E_0 \parallel E_1) \;\sim_M\; (F_0 \parallel F_1) \restriction \mathbf{e}(F_0 \parallel F_1)$

$\Leftrightarrow$ $\{$Definition 2.67, Proposition 2.59$\}$

$(E_0 \parallel E_1) \;\approx_M\; (F_0 \parallel F_1)$

□

Normal form

We introduce $\mathcal{N}$ as a *normal form* of enabling structures with respect to $\approx$. The normal form of an enabling structure is an enabling function over its external alphabet that captures the 'unfolding' of all dependencies between actions. For example, the normal form of enabling structure E of Example 2.20, is given by $\mathcal{N}.E.s.a = 1$ and $\mathcal{N}.E.s.c = s.a + 1.5 \max 2.5$. The term $\max 2.5$ in the enabling of c is an unfolding of $s.a + 1.5$ with the enabling of a.

Definition 2.78 Normal form: $\mathcal{N}$.

The *normal form* $\mathcal{N}.e$ of enabling function e is the enabling function over alphabet $\mathbf{a}e$ defined by $\mathcal{N}.e.s = e.\mathbf{h}(\mathbf{n}s \parallel e)$. The normal form is extended to enabling structures in general by $\mathcal{N}.E = \mathcal{N}(E \restriction \mathbf{e}E)$, and to sets of enabling structures by $\mathcal{N}.\mathcal{E} = (\,\mathbf{set}\, E : E \in \mathcal{E} : \mathcal{N}.E\,)$.

The set of normal enabling functions, $\mathcal{N}$, is defined by

$\mathcal{N} \;=\; (\,\mathbf{set}\, e : \;: \mathcal{N}.e\,)$.

□

Observe that the normal form of an enabling structure is an enabling function and that $\mathbf{b}(\mathcal{N}.e) = \mathbf{b}e$ and $\mathbf{d}(\mathcal{N}.e) \geqslant \mathbf{d}e$.

Proposition 2.79 $\mathcal{N}$ is a normal form w.r.t. $\approx$:

1 $E \approx \mathcal{N}.E$

2 $E \approx F \;\Leftrightarrow\; \mathcal{N}.E = \mathcal{N}.F$

Proof

1 Observe that for $s \in \mathbf{P}e$, $\mathcal{N}.e.s = e.s$ and conclude from Corollary 2.71.1 that $\mathbf{P}(\mathcal{N}.e) = \mathbf{P}e$. Observe furthermore (Proposition 2.59) that $E \restriction \mathbf{e}E \approx E$.

2 The $\Leftarrow$ part of the implication follows from the previous item of this proposition. In order to prove the $\Rightarrow$ implication it suffices to observe enabling functions only (see Proposition 2.59).

For $e \approx f$ we derive:

$$
\begin{array}{ll}
 & \mathcal{N}.e.s \\
= & \\
 & e.\mathbf{h}(\mathbf{n}s \parallel e) \\
= & \quad \{\, e \approx f \,, \mathbf{h}(\mathbf{n}s \parallel e) \in \mathbf{Pe} \,, \text{Corollary 2.71.1} \,\} \\
 & f.\mathbf{h}(\mathbf{n}s \parallel e) \\
= & \quad \{\, e \approx f \,, \text{Proposition 2.77} \,\} \\
 & f.\mathbf{h}(\mathbf{n}s \parallel f) \\
= & \\
 & \mathcal{N}.f.s
\end{array}
$$

□

Alternative characterizations of $\mathcal{N}$ are given in the next proposition.

Proposition 2.80

1 $e \in \mathcal{N} \;\Leftrightarrow\; (\forall s : : e.s = e.\mathbf{h}(\mathbf{n}s \parallel e))$

2 $e \in \mathcal{N} \;\Leftrightarrow\; (\forall s : : e.s = e(s \text{ lub } e.s))$

3 $e \in \mathcal{N} \;\Rightarrow\; (\forall s : : (s \text{ lub } e.s) \in \mathbf{Pe})$

Proof

1 Follows from the definition of $\mathcal{N}$, and the previous proposition.

2 sub $\Rightarrow$ implication.

Assume $e \in \mathcal{N}$, due to the previous item, it suffices to establish that $\mathbf{h}(\mathbf{n}s \parallel e) = s \text{ lub } e.s$; we derive:

$$
\begin{array}{ll}
 & \mathbf{h}(\mathbf{n}s \parallel e) \\
= & \quad \{\, \text{definition of } \mathbf{h} \text{ (2.27)} \,\} \\
 & (\mathbf{n}s \parallel e).\mathbf{h}(\mathbf{n}s \parallel e) \\
= & \quad \{\, \mathbf{a}e = \mathbf{an}s \,, \text{definition of } \parallel \text{ (2.30)} \,\} \\
 & \mathbf{n}s.\mathbf{h}(\mathbf{n}s \parallel e) \text{ lub } e.\mathbf{h}(\mathbf{n}s \parallel e) \\
= & \quad \{\, \text{definition of } \mathbf{n} \text{ (2.26)}, \; e \in \mathcal{N} \,\} \\
 & s \text{ lub } e.s
\end{array}
$$

2 sub $\Leftarrow$ implication.

Assume the right-hand side, again it suffices to establish that

$\mathbf{h}(\mathbf{n}s \parallel e) = s \text{ lub } e.s$.

Due to the choice of e we infer for schedule s:

$s \text{ lub } e.s \;=\; s \text{ lub } e.(s \text{ lub } e.s)$

So $s \text{ lub } e.s$ is a solution of t in the following equation:

$t \;=\; s \text{ lub } e.t$

This equation, however, has as a unique solution $\mathbf{h}(\mathbf{n}s \parallel e)$.

3 We derive for $e \in \mathcal{N}$:

$$\begin{array}{ll} & e(s \text{ lub } e.s) \\ = & \quad \{\text{previous item, } e \in \mathcal{N}\} \\ & e.s \\ \leqslant & \\ & s \text{ lub } e.s \end{array}$$

On account of Proposition 2.33 we conclude s lub $e.s \in \mathbf{Pe}$.

□

The interested reader is invited to verify that the normal form is preserved under restriction ($e \in \mathcal{N} \Rightarrow e \,\|\!\upharpoonright A \in \mathcal{N}$) but in general not under parallel composition.

The presence of all unfoldings of dependencies in enabling functions that are in normal form, may result in rather cumbersome appearances. Compare for example enabling function g in Example 2.66, which is in normal form, with the equivalent enabling function h (in the same example). In turns out, though, that we have employment for the normal form: some properties of enabling functions that depend on the process of enabling functions only, are easier proven (or phrased) in terms of their normal form. The proof of the first half of Proposition 2.82 (see below) is already an example of the usage of the normal form.

In the next chapter we compare the behaviours of enabling functions. In fact, we compare the behaviours of equivalence classes over $\sim$. In the sequel such classes are denoted with [ ] brackets:

Definition 2.81 Equivalence class, closure: [ ].

$[\,e\,] \;=\; (\,\mathbf{set}\, f : f \sim e : f\,)$

The notation is extended to a *closure* operation on sets of enabling functions as follows:

$[\,\mathcal{E}\,] \;=\; (\,\cup\, e : e \in \mathcal{E} : [\,e\,])$

□

We remind the reader that for enabling functions both relations $\approx$ and $\sim$ denote the same equivalence.

For the three types of enabling functions we introduced thus far (*Const*, *Asc*, and *Con*) the closures and an estimation of their normal forms are given by:

Proposition 2.82

1 $[\,\mathit{Const}\,] \;=\; (\,\mathbf{set}\; e : e \text{ is constant on } \mathbf{Pe} : e\,)$

$[\,\mathit{Asc}\,] \;=\; (\,\mathbf{set}\; e : e \text{ is ascending on } \mathbf{Pe} : e\,)$

$[\,\mathit{Con}\,] \;=\; (\,\mathbf{set}\; e : e \text{ is conservative on } \mathbf{Pe} : e\,)$

2 $\mathcal{N}.[\,Const\,] \;=\; Const$

$\mathcal{N}.[\,Asc\,] \;\subset\; Asc$

$\mathcal{N}.[\,Con\,] \;\subset\; Con$

Proof

1 Let $Const'$, Asc', and Con' be the sets at the right-hand sides. The equalities are a direct consequence of the following observations, for $\mathcal{E}$ is $Const$, Asc, and Con respectively:

- $\mathcal{E}' \;=\; [\mathcal{E}']$
- $\mathcal{E} \;\subseteq\; \mathcal{E}'$
- $\mathcal{N} \cap \mathcal{E}' \;\subseteq\; \mathcal{E}$

The first observation follows from Corollary 2.71.1, the second one is evident. Consequently it suffices to prove the last observation. This last observation also implies the $\subseteq$ inclusions in the second part of this proposition.

1 sub $\mathcal{E} = Const$

Let $\mathrm{e} \in \mathcal{N} \cap Const'$, and let s be a schedule over $\mathbf{ae}$. We derive:

$\mathrm{e}.s$

$=$ { Proposition 2.80.2 }

$\mathrm{e}.(s \text{ lub } \mathrm{e}.s)$

$=$ { Proposition 2.80.3, e is constant on $\mathbf{Pe}$ }

$\mathrm{e}.\varepsilon$

Which implies $\mathrm{e} \in Const$.

1 sub $\mathcal{E} = Asc$

Let $\mathrm{e} \in \mathcal{N} \cap Asc'$, we prove by induction that $\mathrm{e} \in Asc$.

Let s and t be schedules over $\mathbf{ae}$ such that $s \leqslant t$.

base $\mathrm{e}.s \downharpoonright \mathbf{b}(\mathrm{e}.t) \;\leqslant\; \mathrm{e}.t \downharpoonright \mathbf{b}(\mathrm{e}.t)$

step

$\mathrm{e}.s \downharpoonright M \;\leqslant\; \mathrm{e}.t \downharpoonright M$

$\Rightarrow$ { $s \leqslant t$ }

$(s \text{ lub } \mathrm{e}.s) \downharpoonright M \;\leqslant\; (t \text{ lub } \mathrm{e}.t) \downharpoonright M$

$\Rightarrow$ { $\mathrm{e} \in \mathcal{N} \cap Asc'$, Propositions 2.80.3, 2.34.2 }

$\mathrm{e}((s \text{ lub } \mathrm{e}.s) \downharpoonright M) \;\leqslant\; \mathrm{e}((t \text{ lub } \mathrm{e}.t) \downharpoonright M)$

$\Rightarrow$ { definition of $\mathbf{d}$ (2.21) }

$\mathrm{e}(s \text{ lub } \mathrm{e}.s) \downharpoonright (M + \mathbf{de}) \;\leqslant\; \mathrm{e}(t \text{ lub } \mathrm{e}.t) \downharpoonright (M + \mathbf{de})$

$\Rightarrow$ { $\mathrm{e} \in \mathcal{N}$, Proposition 2.80.2 }

$\quad \mathrm{e}.s \downharpoonright (M + \mathbf{de}) \;\leqslant\; \mathrm{e}.t \downharpoonright (M + \mathbf{de})$

(end of step)

1 sub $\mathcal{E} = Con$ (this proof is more of the same)

Let $e \in (\mathcal{N} \cap Con')$, we prove by induction that $e \in Con$.

Let s, t (over **ae**), and $\mu : 0 \leqslant \mu < \infty$ such that $s \leqslant t \oplus \mu$.

base $\mathrm{e}.s \downharpoonright \mathbf{b}(\mathrm{e}.t \oplus \mu) \;\leqslant\; (\mathrm{e}.t \oplus \mu) \downharpoonright \mathbf{b}(\mathrm{e}.t \oplus \mu)$

step

$\quad \mathrm{e}.s \downharpoonright M \;\leqslant\; (\mathrm{e}.t \oplus \mu) \downharpoonright M$

$\Rightarrow \quad \{\, s \downharpoonright M \leqslant (t \oplus \mu) \downharpoonright M \,\}$

$\quad s \downharpoonright M \text{ lub } (\mathrm{e}.s) \downharpoonright M \;\leqslant\; (t \oplus \mu) \downharpoonright M \text{ lub } (\mathrm{e}.t \oplus \mu) \downharpoonright M$

$\Leftrightarrow$

$\quad (s \text{ lub } \mathrm{e}.s) \downharpoonright M \;\leqslant\; (t \text{ lub } \mathrm{e}.t) \downharpoonright (M - \mu) \oplus \mu$

$\Rightarrow \quad \{\, \mathrm{e} \in \mathcal{N} \cap Con'$, Propositions 2.80.3, 2.34.2 $\}$

$\quad \mathrm{e}((s \text{ lub } \mathrm{e}.s) \downharpoonright M) \;\leqslant\; \mathrm{e}((t \text{ lub } \mathrm{e}.s) \downharpoonright (M - \mu)) \oplus \mu$

$\Rightarrow \quad \{$ definition of $\mathbf{d}$ (2.21) $\}$

$\quad e(s \text{ lub } \mathrm{e}.s) \downharpoonright (M + \mathbf{de}) \;\leqslant\; e(t \text{ lub } \mathrm{e}.s) \downharpoonright (M + \mathbf{de} - \mu) \oplus \mu$

$\Rightarrow \quad \{\, \mathrm{e} \in \mathcal{N}$, Proposition 2.80.2 $\}$

$\quad \mathrm{e}.s \downharpoonright (M + \mathbf{de}) \;\leqslant\; \mathrm{e}.t \downharpoonright (M + \mathbf{de} - \mu) \oplus \mu$

$\Leftrightarrow$

$\quad \mathrm{e}.s \downharpoonright (M + \mathbf{de}) \;\leqslant\; (\mathrm{e}.t \oplus \mu) \downharpoonright (M + \mathbf{de})$

(end of step)

2 The $\subseteq$ inclusions follow from the 'last observation' in the previous item. It is evident that all constant enabling functions are normal. Finally, the inequalities for ascending and conservative enabling functions follow from the existence of enabling function e (see below) that is a member of Asc as well as Con, but that is not a member of $\mathcal{N}$:

$\quad \mathbf{ae} = \{a, b\} \qquad \mathrm{e}.s = \{(a, 1), (b, s.a + 1)\}$

(the normal form of e enables b at $2 \max s.a + 1$)

□

In the sequel we refer to $[\,Const\,]$, $[\,Asc\,]$, and $[\,Con\,]$ as the *class of* constant, ascending, and conservative enabling functions respectively. The behaviours of these enabling functions are the same as the behaviours of the constant, ascending, and conservative enabling functions; the behaviour of each enabling function being equivalent to the behaviour of its normal form in particular.

In the next chapter we will encounter the three above-mentioned classes again: a lot of proofs for comparison relations are based on their behaviours on $[\,Const\,]$, and both $[\,Asc\,]$ and $[\,Con\,]$ turn out to be the reflexive domain of important comparison relations.

Chapter 3

Comparing Performance

The mechanisms we consider are designed to satisfy timing conditions on the external actions, where satisfaction means being 'fast enough'. Since we are interested in external behaviour only, both timing conditions (specifications) and mechanisms (implementations) are expressed with enabling functions, rather than with enabling structures in general.

In Section 3.2 we figure out for what relation **imp** between enabling functions, $e\,\mathbf{imp}\,f$ expresses that 'e is an implementation of f', where the correctness concern is being 'fast enough'. In the search for a suitable relation we distinguish two problems:

- what *kind* of relation do we aim at (Section 3.1), and
- what is being *fast enough*, or being *at least as fast as* (Section 3.2).

It turns out, in Section 3.3, that after imposing some restrictions we are left with only one relation, $\lhd\!\shortmid$, that is convenient for our purposes. Major choices that lead to this relation are the following:

- The relation has to *imply* being 'at least as fast as' only.

  That is: $e\,\mathbf{imp}\,f \;\Rightarrow\;$ 'e is at least as fast as f' . An extreme, and useless, relation that satisfies this condition, is the relation that does not allow any implementation of a specification: $e\,\mathbf{imp}\,f \Leftrightarrow \mathbf{false}$. We strive for a relation that allows as many implementations as possible.

- The relation must be transitive, and parallel composition as well as restriction must be monotonic with respect to it.

  This condition supports compositional design of mechanisms (design by means of 'divide and conquer'). Consider for example a specification e. The first step in the derivation of an implementation may be to implement e with two parallel components: $(f \parallel g) \Vert\!\!\upharpoonright A\ \mathbf{imp}\ e$. A second step may be to find (realizable) implementations for both f and g. The only condition for these implementations, f' and g', is that $f'\,\mathbf{imp}\,f$ and $g'\,\mathbf{imp}\,g$. Monotonicity of parallel composition and restriction, and transitivity of **imp** , guarantee that the composition of f' and g' is an implementation of the original specification: $(f' \parallel g') \Vert\!\upharpoonright A$ **imp** e .

- The relation does not have to be reflexive for all enabling functions.

  Reflexivity and monotonicity of $\parallel$ and $\Vert\!\upharpoonright$ are conflicting demands. This is due to the fact that there are enabling functions that have a 'speed twisting effect', for example enabling function e that is used in Examples 3.12 and 3.16:

  $$\begin{array}{lcl} e.s.a & = & 0.5 \\ e.s.b & = & \textbf{if } s.a < 1 \rightarrow 100 \\ & & [\!]\ \ s.a \geqslant 1 \rightarrow 3 \\ & & \textbf{fi} \end{array}$$

  The usage of an implementation relation that is not reflexive on its entire domain may be surprising, it is not new. In [5], for example, DI decomposition is an implementation relation that is reflexive for DI components only (Theorem 3.2.1.3, page 59 of [5]).

- The relation should be *robust*. That is, when scaling up an allowed implementation, the result must also be an implementation:

  for any speeding up ρ of e $\quad e\,\mathbf{imp}\,f \ \Rightarrow\ \rho.e\,\mathbf{imp}\,f$.

  Observe that we do *not* demand (for any speeding up ρ of e) $\rho.e\,\mathbf{imp}\,e$. This demand would imply reflexivity because the identity is a speeding up.

Given a specification, the question arises whether or not there exists a *most liberal implementation* that satisfies this specification. More general even, given a set of specifications the question arises whether or not there exists a most liberal implementation that satisfies all specifications. The dual problem is that of *most severe specifications*. In Section 3.4 we establish the existence of most liberal implementations and most severe specifications (under rather weak conditions).

In Section 3.5 we introduce *angelic* and *demonic response time* as a first way to describe the speed of mechanisms. We are, however, more interested in the speed of a mechanism (implementation) with respect to a specification. Therefore we use relation $\lhd\!\!\mid$ to give the *quality* of an implementation relative to a specification. The idea is that the amount of scaling, ρ, that is necessary for an enabling function e in order to satisfy $\rho.e \lhd\!\!\mid f$ is a measure of the quality of implementation e relative to specification f. As a derived concept we introduce *relative response time*.

In Section 3.6 we discuss the description of devices in which timing may vary dependent on causes that are not captured in the enabling model; such as temperature, voltage, and complexity of data. The description of such a device may consist of a enabling structure in which all timing information is parameterized. The major result in this section, is that when the timing of a device can be described with —relatively— fixed delays between cause and effect, the influence of variation of these delays is no more than proportional.

We conclude this chapter, in Section 3.7, with a brief discussion of what (not) to expect from the type of performance analysis that is introduced in this chapter. It turns out that the way in which we compare behaviours is not useful for specifications with 'choice'. Furthermore it turns out that though we do not compare delays between individual actions, there is a simple way to keep a check on such delays.

3.1 How to compare

We introduce *general implementation relations* as the type of relation we use for comparing enabling functions. In Section 3.2 we tailor these to relations that compare speed.

Definition 3.1 General implementation relation.

A *general implementation relation* is a *uniform*, transitive relation on enabling functions such that $\|$ and $\|\!\upharpoonright$ are monotonic with respect to it, and such that only enabling functions over the same alphabet are comparable.

- A relation **imp** on enabling functions is uniform if it is preserved under $\approx$ equivalence, renaming, and scaling. That is: for $e_0 \,\mathbf{imp}\, e_1$ and $f_i \approx e_i$, or $f_i = \mathcal{R}.e_i$, or $f_i = \rho.e_i$, also $f_0 \,\mathbf{imp}\, f_1$.
- $\|$ is monotonic with respect to a relation **imp** if for enabling functions e_x and f_x: $(\forall x : : e_x \,\mathbf{imp}\, f_x) \;\Rightarrow\; (\,\|\, x : : e_x) \,\mathbf{imp}\, (\,\|\, x : : f_x)$.
- $\|\!\upharpoonright$ is monotonic with respect to **imp** if $e \,\mathbf{imp}\, f \;\Rightarrow\; e \|\!\upharpoonright A \,\mathbf{imp}\, f \|\!\upharpoonright A$.

Application of general implementation relations can be extended to enabling structures by hiding the internal symbols first:

$E \,\mathbf{imp}\, F \;\Leftrightarrow\; E \upharpoonright \mathbf{e}E \;\mathbf{imp}\; F \upharpoonright \mathbf{e}F$

□

In the sequel, **imp** is a general implementation relation.

Examples of general implementation relations are the relation that is identical **false**, equivalence relation $\approx$, and the relation that is identical **true**.

Enabling functions for which a general implementation relation is reflexive, are considered to be 'smooth' with respect to this relation: they can be used to implement themselves.

Definition 3.2 The reflexive domain of a relation: $\mathcal{RD}$.

The *reflexive domain* of a binary relation R, denoted by $\mathcal{RD}\,.\,R$, is defined by: $\mathcal{RD}\,.\,R \;=\; (\,\mathbf{set}\, x : x\,R\,x : x\,)$.

□

Definition 3.3 Smoothness-class.

A *smoothness-class*, or *class* for short, is a collection $\mathcal{C}$ of enabling functions, such that there exists a general implementation relation **imp** for which $\mathcal{C} = \mathcal{RD}.\mathbf{imp}$.

A class $\mathcal{C}$ is called *non-trivial* if for some $e \in \mathcal{C}$, $\mathbf{P}e \neq \{\varepsilon\}$.

□

Another characterization of (smoothness) classes is given by:

Proposition 3.4 (without proof)

A class is a *uniform*, *compositional* set of enabling functions.

- Set $\mathcal{C}$ of enabling functions is *uniform* when it is closed under $\approx$ equivalence, renaming, and scaling. That is: for $e \in \mathcal{C}$ and $f \approx e$, or $f = \mathcal{R}.e$, or $f = \rho.e$, also $f \in \mathcal{C}$.
- Set $\mathcal{C}$ of enabling functions is *compositional* when it is closed under (possibly infinite) parallel composition and restriction.

□

In the sequel, $\mathcal{C}$ ranges over classes.

The empty set, and the set of all enabling functions are classes. Furthermore we have

Proposition 3.5 (without proof)

$[\,Const\,]$ is a class, it is even the minimal non-trivial class:

$(\forall \mathcal{C} : \mathcal{C} \text{ is non-trivial} : [\,Const\,] \subseteq \mathcal{C})$

□

Is is easily verified that $[\,Asc\,]$ and $[\,Con\,]$ are also classes. In fact, this even follows from Propositions 3.30 and 3.40 respectively.

In order to give the reader some feeling about general implementation relations and classes, we mention the following properties.

Proposition 3.6 (without proof)

1 For general implementation relations $\mathbf{imp}_0$ and $\mathbf{imp}_1$, the relation **imp** as defined below is also a general implementation relation.

$$e\,\mathbf{imp}\,f \;=\; e\,\mathbf{imp}_0\,f \wedge e\,\mathbf{imp}_1\,f$$

The counterpart with $\vee$ instead of $\wedge$ does in general not hold.

2 For classes $\mathcal{C}_0$ and $\mathcal{C}_1$, $\mathcal{C}_0 \cap \mathcal{C}_1$ is also a class.

3 For any general implementation relation $\mathbf{imp}_0$, and class C , the relation $\mathbf{imp}_1$ as defined below is a general implementation relation.

$$\begin{array}{lcll} e\,\mathbf{imp}_1\, f & = & \mathbf{if} & e \in C \wedge f \in C \rightarrow e\,\mathbf{imp}_0\, f \\ & & [] & e \notin C \vee f \notin C \rightarrow \mathbf{false} \\ & & \mathbf{fi} & \end{array}$$

□

We conclude this section with two interesting 'maximality' properties of $[Asc]$ (compare Proposition 3.7.2 with Proposition 2.63).

Proposition 3.7

1 $[Asc]$ is the maximal class C that satisfies
$(\forall e, s : e \in C \wedge s \in \mathbf{P}e : s \geqslant \mathbf{h}e)$

2 $[Asc]$ is the maximal class C that satisfies
$(\forall e, A : e \in C : \mathbf{P}(e \Vert A) = \mathbf{P}e \restriction A)$

□

The following lemma is used in the proof of maximality of $[Asc]$ in Proposition 3.7.1.

Lemma 3.8

For enabling structure E and schedule v over the same alphabet:
$(\forall s : s \in \mathbf{P}E : s \geqslant v) \;\Leftrightarrow\; (\forall s : s \in \mathbf{P}E : E.s \geqslant v)$

Proof

The $\Leftarrow$ implication is evident. Remains to prove the $\Rightarrow$ part.

Assume the left-hand side. Let $s \in \mathbf{P}E$, $a \in \mathbf{a}E$ and let $M = E.s.a$.

From Proposition 2.33 we infer that s and M satisfy the premise in Proposition 2.69.1. So there exists a t in PE with $E.s.a = t.a \geqslant v.a$.

□

Proof of Proposition 3.7.

1 : $[Asc]$ satisfies $(\forall e, s : e \in [Asc] \wedge s \in \mathbf{P}e : s \geqslant \mathbf{h}e)$.

The reader is invited that for e and f in $[Asc]$, $e \leqslant f \Rightarrow \mathbf{h}e \leqslant \mathbf{h}f$ (this is a special case of the first property that is proved in the proof of Theorem 3.28).

Using this result, we derive for $s \in \mathbf{P}e$:

$$\begin{array}{ll} & e \in [Asc] \\ \Rightarrow & \quad \{\text{see above}\} \\ & \mathbf{h}e \leqslant \mathbf{h}(e \Vert \mathbf{n}s) \\ \Leftrightarrow & \quad \{\, s \in \mathbf{P}e \,\} \\ & \mathbf{h}e \leqslant s \end{array}$$

1 : $[Asc]$ is maximal.

Assume $\mathcal{C}$ satisfies in $(\forall e,s : e \in \mathcal{C} \wedge s \in \mathbf{P}e : s \geqslant \mathbf{h}e)$.

Let $e \in \mathcal{C}$, $s,t \in \mathbf{P}e$, and $a \in \mathbf{a}e$ such that $s \leqslant t$: to prove $e.s.a \leqslant e.t.a$.

When $e.t.a = \infty$, this is evident.

Assume $e.t.a < \infty$. Conclude that $\mathcal{C}$ is not trivial, hence $[Const] \subseteq \mathcal{C}$.

Define e' over alphabet $\mathbf{a}e$ by:

$e'.u.b \;=\; \mathbf{if}\; b \neq a \;\rightarrow\; s.b \;[]\; b = a \;\rightarrow\; e.t.a \;\mathbf{fi}$

Because $e' \in [Const]$, we may conclude $e' \in \mathcal{C}$ and $(e \parallel e') \in \mathcal{C}$.

We derive:

$$\begin{array}{ll} & e.t.a \\ = & \\ & (e \parallel e').t.a \\ \geqslant & \quad \{ (e \parallel e') \in \mathcal{C},\ t \in \mathbf{P}(e \parallel e') \text{ so } t \geqslant \mathbf{h}(e \parallel e'),\ \text{Lemma 3.8} \} \\ & \mathbf{h}(e \parallel e').a \\ = & \\ & e.s.a \;\max\; e.t.a \end{array}$$

which implies $e.s.a \leqslant e.t.a$.

2 : $[Asc]$ satisfies $(\forall e, A : e \in [Asc] : \mathbf{P}(e \Vert\!\restriction A) = \mathbf{P}e \restriction A)$.

Since $\mathbf{P}(e \Vert\!\restriction A) \subseteq \mathbf{P}e \restriction A$, we only have to prove the $\supseteq$ part.

Let $e \in [Asc]$, $s \in \mathbf{P}e$, and A an alphabet. Let $f \in Asc$ such that $e \approx f$ (for example $f = \mathcal{N}.e$) .

By induction it is easily proven that $s \restriction A \uparrow f \leqslant s$. Furthermore, we derive:

$$\begin{array}{ll} & (f \Vert\!\restriction A)(s \restriction A) \\ = & \quad \{ \text{Proposition 2.62} \} \\ & f(s \restriction A \uparrow f) \restriction A \\ \leqslant & \quad \{ f \in Asc,\ s \restriction A \uparrow f \leqslant s \} \\ & f.s \restriction A \\ \leqslant & \quad \{ s \in \mathbf{P}f \} \\ & s \restriction A \end{array}$$

So $s \restriction A \in \mathbf{P}(f \Vert\!\restriction A) = \mathbf{P}(e \Vert\!\restriction A)$.

2 : $[Asc]$ is maximal.

Let $\mathcal{C}$ such that $\mathcal{C} \setminus [Asc] \neq \varnothing$. Choose e, s, and a such that $e \in \mathcal{C}$ and $s \in \mathbf{P}e$ and $\mathbf{h}e.a > s.a$ (see previous item).

Observe that $s \restriction a \in \mathbf{P}e \restriction a$ but $s \restriction a \notin \mathbf{P}(e \Vert\!\restriction a)$.

□

3.2 How to compare speed

In the previous section we discussed what kind of relation we want to use to compare enabling functions. Remains the other question, what is *fast enough* or *at least as fast as*. We discuss three points of view:

- speed in a greedy environment,
- speed in any possible situation, and
- the effect on an 'observing' environment.

Each of these points of view can be justified as a criterion for comparing speed. In general, such —competing— criteria may lead to distinct 'types' of comparison relations. It turns out, however, that it is immaterial which of these criteria we use: they all give rise to the same type of comparison relation.

Speed in a greedy environment

When a mechanism is composed with a greedy environment, all its actions are performed as soon as they are enabled. This gives rise to relation $\sqsubseteq_h$ for comparing enabling functions:

Definition 3.9 $\sqsubseteq_h$.

$$e \sqsubseteq_h f \quad \Leftrightarrow \quad \mathbf{h}e \leqslant \mathbf{h}f$$

□

This relation, however, is no general implementation relation: it satisfies all demands except for monotonicity of $\|$.

Example 3.10 $\sqsubseteq_h$ and $\|$.

Consider the following enabling functions e and f over $\{a, b\}$:

$$\begin{aligned} e.s.a &= 1 & \qquad f.s.a &= 1 \\ e.s.b &= s.a + 1 & \qquad f.s.b &= 2 \end{aligned}$$

Observe that $e \sqsubseteq_h f$ and $\mathrm{n}(a,2) \sqsubseteq_h \mathrm{n}(a,2)$,

but that $\neg(e \;\|\; \mathrm{n}(a,2) \sqsubseteq_h f \;\|\; \mathrm{n}(a,2))$.

□

Speed in any possible situation

One obtains a more sophisticated comparison relation by comparing the behaviour of two enabling functions for all situations in which both can be engaged. With relation $\sqsubseteq_P$ we express that in any situation, the (intended) implementation is at least as fast as the specification.

Definition 3.11 $\sqsubseteq_P$.

$$e \sqsubseteq_P f \;\Leftrightarrow\; \mathbf{a}e = \mathbf{a}f \wedge e \leqslant_{\mathbf{P}e \cap \mathbf{P}f} f$$

□

From Corollary 2.71.2 we conclude that this coincides with process inclusion:

$e \sqsubseteq_P f \;\Leftrightarrow\; \mathbf{P}e \supseteq \mathbf{P}f$.

Alas, relation $\sqsubseteq_P$ is no general implementation relation: it satisfies all demands except for monotonicity of $\Vert\!\upharpoonright$.

Example 3.12 Speed Twisting by an internal trap: $\sqsubseteq_P$ & $\Vert\!\upharpoonright$.

We compare two enabling functions over $\{a, b\}$:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
e.s.a & = & 0.5 & f.s.a & = & 1 \\
e.s.b & = & \textbf{if}\;\; s.a < 1 \;\rightarrow\; 100 & f.s.b & = & e.s.b \\
 & & [\!]\;\; s.a \geqslant 1 \;\rightarrow\; 3 & & & \\
 & & \textbf{fi} & & &
\end{array}$$

Observe that e is indeed an enabling function: for any s and t holds $\mathrm{sim}(e.s, e.t) \geqslant 3$, and if $\mathrm{sim}(s,t) \geqslant 1$ then $\mathrm{sim}(e.s, e.t) = \infty$ (similar for f). Observe furthermore that $e \sqsubseteq_P f$, but that

$e \Vert\!\upharpoonright b \approx \mathbf{n}(b, 100)$ and $f \Vert\!\upharpoonright b \approx \mathbf{n}(b, 3)$, so $\neg(e \Vert\!\upharpoonright b \sqsubseteq_P f \Vert\!\upharpoonright b)$.

□

This example also shows the relative merits of being at least as fast as at any moment: by being fast for one particular action, a mechanism can be 'trapped' because other actions suffer from (excessive) delays.

One may wonder, 'Why bother about those 'weird' enabling functions?' (like e in the previous example) They only cause problems. Can we not just discuss 'smooth' enabling functions with process inclusion as a —reflexive— comparison relation?'. The answer is, yes we could, but we have two critical observations to this approach.

- Apart from the arbitrary choice of process inclusion, the question remains: what is the (or a?) maximal class that can be given the predicate 'smooth'? This maximal class comes as a spin-off in the analysis of Section 3.2 (in the non-robust case it is $[\,Asc\,]$, in the robust case $[\,Con\,]$).

- More importantly still, what when someone comes up with a mechanism with a 'weird' external behaviour, for example enabling function e of Example 3.12? Then the comparison relation cannot be used to verify whether it may be plugged into some design, at some place where a behaviour is assumed at least as fast as $\mathbf{n}\{(a, 0.5), (b, 100)\}$, or perhaps $\mathbf{n}\{(a, 1), (b, 3)\}$. One may have some intuition about it, but a comparison relation that excludes e is useless in this case.

In fact, we use 'weird' enabling functions when analyzing distributed FIFO buffers with bypassing (Section 5.5).

The effect on an 'observing' environment

Finally we try an observational relation. We consider environments (G) that are interested in scheduling their own actions as fast as possible.

Definition 3.13 $\preceq$.

$$e \preceq f \;\Leftrightarrow\; \mathbf{a}e = \mathbf{a}f \wedge (\forall G : : \mathbf{h}(G \parallel e) \restriction \mathbf{a}G \leqslant \mathbf{h}(G \parallel f) \restriction \mathbf{a}G)$$

□

Other characterizations of $\preceq$ are given by:

Proposition 3.14 (without proof)

$$\begin{aligned} e \preceq f \;&\Leftrightarrow\; \mathbf{a}e = \mathbf{a}f \wedge (\forall g : : \mathbf{h}(g \parallel e) \leqslant \mathbf{h}(g \parallel f)) \\ &\Leftrightarrow\; \mathbf{a}e = \mathbf{a}f \wedge (\forall g : \mathbf{a}g \supseteq \mathbf{a}e : \mathbf{h}(g \parallel e) \leqslant \mathbf{h}(g \parallel f)) \end{aligned}$$

□

In Section 2.1 of [9], relations between mechanisms are expressed in terms of passing or failing tests. Relation $\preceq$ can also be expressed in such a way. For enabling functions over alphabet A we introduce tests $T(s,g)$, with $\mathbf{a}s = A \cup \mathbf{a}g$. An enabling function e passes test $T(s,g)$ if (and only if) $\mathbf{h}(e \parallel g) \leqslant s$. Relation $\preceq$ can then be expressed as follows:

$$\begin{aligned} e \preceq f \;\Leftrightarrow\; &\mathbf{a}e = \mathbf{a}f \wedge \\ &(\forall s,g : \mathbf{a}s = \mathbf{a}e \cup \mathbf{a}g \wedge f \text{ passes } T(s,g) : e \text{ passes } T(s,g)) \end{aligned}$$

The good news is that relation $\preceq$ is an implementation relation, it has, however, one major drawback:

Proposition 3.15

$$e \preceq f \;\Leftrightarrow\; e \approx f$$

□

This is not surprising, since an observing environment may contain a 'trap':

Example 3.16 Speed Twisting by a trap in the environment: $\preceq$ and $\parallel$.

We compare two simple enabling functions, $\mathbf{n}(a,1)$ and $\mathbf{n}(a,0.5)$. Clearly the second is a speeding up of the first, it seems reasonable to consider it as an implementation of the first.

Environment e of Example 3.12, however, is trapped when a is enabled before moment 1. The histories of the compositions are given by:

$$\begin{aligned} \mathbf{h}(\mathbf{n}(a,1) \parallel e) &= \{(a,1),(b,3)\} \\ \mathbf{h}(\mathbf{n}(a,0.5) \parallel e) &= \{(a,0.5),(b,100)\} \end{aligned}$$

Conclude that $\neg(\mathbf{n}(a,0.5) \preceq \mathbf{n}(a,1))$.

□

Proof of Proposition 3.15

The non-trivial part of this proposition is the $\Rightarrow$ part of the equivalence. We subsequently prove $e \preceq f \;\Rightarrow\; \mathbf{P}e \supseteq \mathbf{P}f$ and $e \preceq f \;\Rightarrow\; \mathbf{P}e \subseteq \mathbf{P}f$.

- $e \preceq f \;\Rightarrow\; \mathbf{P}e \supseteq \mathbf{P}f$

  Assume $e \preceq f$, we derive:

  $$\begin{array}{l} \quad s \in \mathbf{P}f \\ \Leftrightarrow \\ \quad \mathbf{h}(f \parallel \mathbf{n}s) = s \\ \Rightarrow \qquad \{ e \preceq f ,\ \mathbf{n}s \preceq \mathbf{n}s \} \\ \quad \mathbf{h}(e \parallel \mathbf{n}s) \leqslant s \\ \Leftrightarrow \\ \quad \mathbf{h}(e \parallel \mathbf{n}s) = s \\ \Leftrightarrow \\ \quad s \in \mathbf{P}e \end{array}$$

- $e \preceq f \;\Rightarrow\; \mathbf{P}e \subseteq \mathbf{P}f$

  Let e and f be enabling functions over A.

  Assume the negation of the right-hand side, we prove $\neg(e \preceq f)$.

  Let $s \in \mathbf{P}e \setminus \mathbf{P}f$, $\Delta = \mathbf{d}e \min \mathbf{d}f$, and $M = \mathrm{sim}(s, \mathbf{P}f)$.

  Observe that $\neg(f.s \downharpoonright (M+\Delta) \leqslant s \downharpoonright (M+\Delta))$ and let $b \in A$ such that $s.b < M + \Delta$ and $s.b < f.s.b$.

  Now define enabling function g over $A \cup \{c\}$, for some c, $c \notin A$, by:

  $$\begin{array}{lcl} g.t.a & = & s.a \quad (a \neq c) \\ g.t.c & = & \textbf{if } t.b \leqslant s.b \;\rightarrow\; \infty \\ & & [\!]\;\; t.b > s.b \;\rightarrow\; s.b + \Delta \\ & & \textbf{fi} \end{array}$$

  Observe that $\mathbf{h}g = s \cup \{(c, \infty)\}$ so $\mathbf{h}g \upharpoonright A \in \mathbf{P}e$ and conclude

  $$\mathbf{h}(g \parallel e) \;=\; s \cup \{(c,\infty)\} \;.$$

  Furthermore, $\mathbf{h}(g \parallel f) \downharpoonright M \upharpoonright A \;=\; s \downharpoonright M$.
  Since $f.s.b > s.b$ and $s.b < M + \mathbf{d}f$, we may conclude $\mathbf{h}(g \parallel f).b > s.b$, which implies $\mathbf{h}(f \parallel g).c = s.b + \Delta$.

  We have to conclude $\neg(\mathbf{h}(g \parallel e) \leqslant \mathbf{h}(g \parallel f))$, and thus $\neg(e \preceq f)$.

□

Apparently, we cannot make every environment happy; we have to consider a subset of 'smooth' environments only:

Definition 3.17 $\preceq_{\mathcal{E}}$.

For $\mathcal{E}$ a subset of the enabling functions, the relation $\preceq_{\mathcal{E}}$ is defined by:

$$e \preceq_{\mathcal{E}} f \;\Leftrightarrow\; \mathbf{a}e = \mathbf{a}f \wedge (\forall g : g \in \mathcal{E} : \mathbf{h}(g \parallel e) \leqslant \mathbf{h}(g \parallel f))$$

□

Relation $\preceq_{\mathcal{E}}$ can be expressed using tests $T(s,g)$ with $g \in \mathcal{E}$.
Remain the questions: what (kind of) $\mathcal{E}$ to choose, and is the resulting relation an implementation relation? The answers are: we suggest a smoothness-class, and in general not.

All roads lead to Rome

Instead of trying to find an implementation relation that *exactly captures* some —intuitive— criterion for being at least as fast as, we concentrate on finding implementation relations that *imply* being 'at least as fast as'. Theorem 3.19 shows that it is immaterial which of the previously introduced criteria is chosen.

Definition 3.18 $\Rightarrow_{\mathcal{E}}$.

For $\mathcal{E}$ a set of enabling functions, the pre-order $\Rightarrow_{\mathcal{E}}$ is defined by:

$(R_0 \Rightarrow_{\mathcal{E}} R_1) \quad \Leftrightarrow \quad (\forall e,f : e,f \in \mathcal{E} \wedge eR_0f : eR_1f)$

for relations R_0 and R_1 on enabling functions. $\Leftarrow_{\mathcal{E}}$ and $\Leftrightarrow_{\mathcal{E}}$ are used, with the obvious meaning, and when $\mathcal{E}$ is the class of all enabling functions it may be omitted as a subscript: e.g. $\Rightarrow$ instead of $\Rightarrow_{\mathcal{EF}}$.

□

Theorem 3.19

$$\begin{array}{cl} & \mathbf{imp} \;\Rightarrow\; \sqsubseteq_h \\ \Leftrightarrow & \\ & \mathbf{imp} \;\Rightarrow\; \sqsubseteq_P \\ \Leftrightarrow & \\ & \mathbf{imp} \;\Rightarrow\; \preceq_{\mathcal{RD}}.\mathbf{imp} \end{array}$$

□

The proof of this theorem is postponed until after Definition 3.20.

Definition 3.20 Implementation relation, robustness:

An *implementation relation*, or IR for short, is a general implementation relation $\mathbf{imp}$ that satisfies $\mathbf{imp} \Rightarrow \sqsubseteq_h$.

An IR $\mathbf{imp}$ is *robust* if $e\,\mathbf{imp}\,f \;\Rightarrow\; (\forall \rho : \rho \in SU.e : \rho.e\,\mathbf{imp}\,f)$.

□

In the sequel $\mathbf{imp}$ ranges, by default, over implementation relations.

Observe that the condition for robustness is equivalent with

$e\,\mathbf{imp}\,f \;\Rightarrow\; (\forall \rho : \rho \in SD.f : e\,\mathbf{imp}\,\rho.f)$

A reason for demanding robustness is that when —for example by some new technique— an implementation is scaled up, it should still be fast enough. Another reason is that in general the designer has no exact knowledge of 'real world' delays: usually the order of magnitude is known only.

The remainder of this section is used to prove Theorem 3.19 and some additional properties. For further use we mention the following

Remark 3.21

In none of the proofs that lead to Theorem 3.19 we use transitivity of general implementation relations, and of implementation relations in particular.

□

In the proof of Theorem 3.19 we use some knowledge about the behaviour of implementation relations on $[\,Const\,]$. It turns out that in more proofs such knowledge is valuable. Therefore we first analyze the behaviour of implementation relations on $[\,Const\,]$. It turns out that only six behaviours are possible, of which four are robust.

Definition 3.22 R_i

The relations R_i $(1 \leqslant i \leqslant 6)$ are defined by:

- $\mathrm{e} \notin [\,Const\,] \vee f \notin [\,Const\,] \vee \mathbf{ae} \neq \mathbf{af} \;\Rightarrow\; \neg(\,\mathrm{e}\, R_i\, f\,)$
- otherwise $(\,\mathrm{e} \in [\,Const\,] \wedge f \in [\,Const\,] \wedge \mathbf{ae} = \mathbf{af}\,)$:

$$\begin{array}{lcl}
\mathrm{e}\, R_1\, f & \Leftrightarrow & \mathbf{false} \\
\mathrm{e}\, R_2\, f & \Leftrightarrow & \mathbf{he} = \mathbf{hf} = \varepsilon \\
\mathrm{e}\, R_3\, f & \Leftrightarrow & \mathbf{hf} = \varepsilon \\
\mathrm{e}\, R_4\, f & \Leftrightarrow & \mathbf{he} \leqslant \mathbf{hf} \\
\mathrm{e}\, R_5\, f & \Leftrightarrow & \mathbf{he} = \mathbf{hf} \\
\mathrm{e}\, R_6\, f & \Leftrightarrow & (\,\exists A : \mathbf{he} \restriction A = \mathbf{hf} \restriction A : \mathbf{hf} \setminus A = \varepsilon\,)
\end{array}$$

□

The mutual relationships between relations R_i are given in the following diagram:

$$\begin{tikzcd}
 & & R_3 \arrow[dr, Rightarrow] & & \\
R_1 \arrow[r, Rightarrow] & R_2 \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & & R_6 \arrow[r, Rightarrow] & R_4 \\
 & & R_5 \arrow[ur, Rightarrow] & &
\end{tikzcd}$$

Figure 3.23 The partial ordering of relations R_i .

Theorem 3.24 Behaviour of IR's over $[\,Const\,]$.

1 For any IR **imp** there is an i, $1 \leqslant i \leqslant 6$, such that $\mathbf{imp} \Leftrightarrow_{[\,Const\,]} R_i$.

2 For any robust IR **imp** this i satisfies $1 \leqslant i \leqslant 4$.

□

In the proof of this theorem we use the following

Lemma 3.25

Let q be an action.
For $M \in \mathbf{T}$ we define enabling function e_M by $e_M = \mathbf{n}(\mathrm{q}, M)$.
Let furthermore $\mathcal{E}$ be defined by $\mathcal{E} = (\,\mathbf{set}\ M : \ : e_M\,)$.

For general implementation relations $\mathbf{imp}_0$ and $\mathbf{imp}_1$ holds:

$(\,\mathbf{imp}_0 \Rightarrow_{[\,Const\,]} \mathbf{imp}_1\,) \;\Leftrightarrow\; (\,\mathbf{imp}_0 \Rightarrow_{\mathcal{E}} \mathbf{imp}_1\,)$

Proof

The $\Rightarrow$ implication is trivial since $[\,Const\,] \supseteq \mathcal{E}$. Remains to prove the $\Leftarrow$ part.

Assume $\mathbf{imp}_0 \Rightarrow_{\mathcal{E}} \mathbf{imp}_1$ and let e and f be constant enabling functions over the same alphabet, say A, choose s and t over A such that $e \approx \mathbf{n}s$ and $f \approx \mathbf{n}t$.

We derive:

$\{$ the hints with uniformity and monotonicity refer to Definition 3.1 $\}$

$e\,\mathbf{imp}_0\,f$

$\Leftrightarrow$ $\{$ uniformity sub $\approx$, choice of s and t $\}$

$\mathbf{n}s\,\mathbf{imp}_0\,\mathbf{n}t$

$\Rightarrow$ $\{ \Vert\!\upharpoonright$ is monotonic $\}$

$(\forall a : a \in A : \mathbf{n}s \Vert\!\upharpoonright a\,\mathbf{imp}_0\,\mathbf{n}t \Vert\!\upharpoonright a)$

$\Leftrightarrow$ $\{ \mathbf{n}s \Vert\!\upharpoonright a = \mathbf{n}(s \upharpoonright a) \}$

$(\forall a : a \in A : \mathbf{n}(s \upharpoonright a)\,\mathbf{imp}_0\,\mathbf{n}(t \upharpoonright a))$

$\Leftrightarrow$ $\{$ uniformity sub renaming $\}$

$(\forall a : a \in A : \mathbf{n}(s \upharpoonright a)_{a \to \mathbf{q}}\,\mathbf{imp}_0\,\mathbf{n}(s \upharpoonright a)_{a \to \mathbf{q}})$

$\Rightarrow$ $\{$ renamings are members of $\mathcal{E}$, assumption: $\mathbf{imp}_0 \Rightarrow_{\mathcal{E}} \mathbf{imp}_1$ $\}$

$(\forall a : a \in A : \mathbf{n}(s \upharpoonright a)_{a \to \mathbf{q}}\,\mathbf{imp}_1\,\mathbf{n}(s \upharpoonright a)_{a \to \mathbf{q}})$

$\Leftrightarrow$ $\{$ uniformity sub renaming $\}$

$(\forall a : a \in A : \mathbf{n}(s \upharpoonright a)\,\mathbf{imp}_1\,\mathbf{n}(t \upharpoonright a))$

$\Rightarrow$ $\{ \Vert$ is monotonic $\}$

$(\Vert\, a : a \in A : \mathbf{n}(t \upharpoonright a))\,\mathbf{imp}_1\,(\Vert\, a : a \in A : \mathbf{n}(s \upharpoonright a))$

$\Leftrightarrow$ $\{ \mathbf{n}s = (\Vert\, a : a \in \mathbf{a}s : \mathbf{n}(s \upharpoonright a)) \}$

$\mathbf{n}s\,\mathbf{imp}_1\,\mathbf{n}t$

$\Leftrightarrow$ $\{$ uniformity sub $\approx$, choice of s and t $\}$

$e\,\mathbf{imp}_1\,f$

□

Proof of Theorem 3.24

We only show how to prove the first statement. Verification of the second one is left to the reader

First we observe that relations R_i are general implementation relations. From Lemma 3.25 we infer that it suffices to prove that for any IR $\mathbf{imp}$ there is a i, $1 \leqslant i \leqslant 6$, such that $\mathbf{imp} \Leftrightarrow_{\mathcal{E}} R_i$. Let us first rewrite R_i for members of $\mathcal{E}$:

$$\begin{array}{lcl}
e_M\, R_1\, e_N & \Leftrightarrow & \textbf{false} \\
e_M\, R_2\, e_N & \Leftrightarrow & M = N \ \wedge \ N = \infty \\
e_M\, R_3\, e_N & \Leftrightarrow & N = \infty \\
e_M\, R_4\, e_N & \Leftrightarrow & M \leqslant N \\
e_M\, R_5\, e_N & \Leftrightarrow & M = N \\
e_M\, R_6\, e_N & \Leftrightarrow & M = N \ \vee \ N = \infty
\end{array}$$

In the remainder of this proof, **imp** is an implementation relation.

- Assume $R_1 \Rightarrow_{\mathcal{E}}$ **imp** and $\neg(R_1 \Leftarrow_{\mathcal{E}}$ **imp**$)$. We prove $R_2 \Rightarrow_{\mathcal{E}}$ **imp** .

  Let M and N such that e_M **imp** e_N , we derive:

  $$\begin{array}{ll}
  & e_M \ \mathbf{imp}\ e_N \\
  \Rightarrow & \quad \{\text{uniformity sub scaling}\} \\
  & (\forall \mu : \mu \geqslant 0 : e_M \oplus \mu \ \mathbf{imp}\ e_N \oplus \mu) \\
  \Rightarrow & \quad \{\ \| \text{ is monotonic}\} \\
  & (\ \| \ \mu : \mu \geqslant 0 : e_M \oplus \mu)\,\mathbf{imp}\ (\ \| \ \mu : \mu \geqslant 0 : e_N \oplus \mu) \\
  \Leftrightarrow & \\
  & (\ \| \ \mu : \mu \geqslant 0 : e_{M+\mu})\,\mathbf{imp}\ (\ \| \ \mu : \mu \geqslant 0 : e_{N+\mu}) \\
  \Leftrightarrow & \\
  & e_\infty \ \mathbf{imp}\ e_\infty
  \end{array}$$

  Conclude $R_2 \Rightarrow_{\mathcal{E}}$ **imp** .

- Assume $R_2 \Rightarrow_{\mathcal{E}}$ **imp** and $\neg(R_2 \Leftarrow_{\mathcal{E}}$ **imp**$)$.

  We prove $(R_3 \Rightarrow_{\mathcal{E}}$ **imp**$) \vee (R_5 \Rightarrow_{\mathcal{E}}$ **imp**$)$.

  We distinguish three cases:

  - $(\exists M, N : M > N : e_M \ \mathbf{imp}\ e_N)$: since **imp** $\Rightarrow \sqsubseteq_h$ this case cannot happen.
  - $(\exists M : M < \infty : e_M \ \mathbf{imp}\ e_\infty)$: conclude, with scaling, that $R_3 \Rightarrow_{\mathcal{E}}$ **imp** ,
  - $(\exists M, N : M \leqslant N < \infty : e_M \ \mathbf{imp}\ e_N)$: see below.

  Let M and N as in the quantification and define for $i : 1 \leqslant i < \infty$ scale ρ_i by $\rho_i.O \ = \ N + \frac{O-N}{i}$.

  We derive:

  $$\begin{array}{ll}
  & e_M \ \mathbf{imp}\ e_N \\
  \Rightarrow & \\
  & (\forall i : \ : \rho_i.e_M \ \mathbf{imp}\ \rho_i.e_N) \\
  \Rightarrow & \\
  & (\ \| \ i : \ : \rho_i.e_M)\,\mathbf{imp}\,(\ \| \ i : \ : \rho_i.e_N) \\
  \Leftrightarrow & \quad \{\text{definition of } \rho_i\ \} \\
  & (\ \| \ i : \ : e_{N+(M-N)/i})\,\mathbf{imp}\,(\ \| \ i : \ : e_N) \\
  \Leftrightarrow & \quad \{\ M \leqslant N\ \} \\
  & e_N \ \mathbf{imp}\ e_N
  \end{array}$$

Conclude, with scaling, that $R_5 \Rightarrow_{\mathcal{E}}$ **imp** .

- The remaining part is proven in the style of both previous items, this exercise is left to the reader.

□

The following lemma is used in the proof of the $\Rightarrow$ part of the first equivalence in Theorem 3.19. It is also used in the proof of Theorem 3.32.2 and Theorem 3.37.

Lemma 3.26 Behaviour of IR's over $[\,Const\,]$.

For any IR **imp**

$$(\exists\, \mathrm{e}, f : \mathrm{e}\,\mathbf{imp}\,f : \mathbf{P}f \neq \{\varepsilon\}) \;\Rightarrow\; (R_5 \Rightarrow_{[Const]} \mathbf{imp})$$

Proof

Assume the left-hand side and let e and f be as in the left-hand side. Observe that $\mathbf{P}f \neq \{\varepsilon\}$ is equivalent with $\mathbf{h}f \neq \varepsilon$. Let a such that $\mathbf{h}\mathrm{e}.a < \infty$. Observe that $\mathrm{e} \Vert\!\!\upharpoonright a$ and $f \Vert\!\!\upharpoonright a$ are constant enabling functions, that (since $\Vert\!\!\upharpoonright$ is monotonic) $\mathrm{e} \Vert\!\!\upharpoonright a\,\mathbf{imp}\,f \Vert\!\!\upharpoonright a$, and that $\mathbf{h}(f \Vert\!\!\upharpoonright a) \neq \varepsilon$.

This rules out relations R_1 , R_2 , and R_3 . Together with the mutual relationships (Figure 3.23) and Theorem 3.24.1 this implies $R_5 \Rightarrow_{[Const]}$ **imp** .

□

Proof of Theorem 3.19

In this proof **imp** is a *general* implementation relation.

- $(\mathbf{imp} \Rightarrow \sqsubseteq_h) \;\Rightarrow\; (\mathbf{imp} \Rightarrow \sqsubseteq_P)$.

Assume the left-hand side. Let e and f such that $\mathrm{e}\,\mathbf{imp}\,f$.

When $\mathbf{P}f = \{\varepsilon\}$, $\mathbf{P}\mathrm{e} \supseteq \mathbf{P}f$ is obvious.

Otherwise we conclude from Lemma 3.26 that $R_5 \Rightarrow_{[Const]}$ **imp** . Now we can proceed as in the first part of the proof of Proposition 3.15, with **imp** instead of $\preceq$.

- $(\mathbf{imp} \Rightarrow \sqsubseteq_P) \;\Rightarrow\; (\mathbf{imp} \Rightarrow \sqsubseteq_h)$.

Assume the left-hand side; we derive:

$$\begin{array}{ll}
 & \mathrm{e}\,\mathbf{imp}\,f \\
\Rightarrow & \quad \{\ \Vert\!\!\upharpoonright \text{ is monotonic}\ \} \\
 & (\forall a : : \mathrm{e} \Vert\!\!\upharpoonright a\,\mathbf{imp}\,f \Vert\!\!\upharpoonright a) \\
\Rightarrow & \quad \{\ \mathbf{imp} \Rightarrow \sqsubseteq_P\ \} \\
 & (\forall a : : \mathbf{P}(\mathrm{e} \Vert\!\!\upharpoonright a) \supseteq \mathbf{P}(f \Vert\!\!\upharpoonright a)) \\
\Leftrightarrow & \quad \{\ \mathrm{e} \Vert\!\!\upharpoonright a,\ f \Vert\!\!\upharpoonright a \in [\,Const\,]\ \} \\
 & (\forall a : : \mathbf{h}(\mathrm{e} \Vert\!\!\upharpoonright a) \leqslant \mathbf{h}(f \Vert\!\!\upharpoonright a)) \\
\Leftrightarrow &
\end{array}$$

$(\forall a : : \mathbf{h}e \restriction a \leqslant \mathbf{h}f \restriction a))$

$\Leftrightarrow$

$\mathbf{h}e \leqslant \mathbf{h}f$

$\Leftrightarrow$ $\{$Definition 3.9$\}$

$e \sqsubseteq_h f$

- $(\mathbf{imp} \Rightarrow \sqsubseteq_h) \;\Rightarrow\; (\mathbf{imp} \Rightarrow \preceq_{\mathcal{RD}.\mathbf{imp}})$.

  Assume the left-hand side; we derive:

  $e\,\mathbf{imp}\,f$

  $\Rightarrow$ $\{\ \|\ $ is monotonic$\}$

  $(\forall g : g\,\mathbf{imp}\,g : e \parallel g\ \mathbf{imp}\ f \parallel g)$

  $\Rightarrow$ $\{$definition of $\mathcal{RD}$ (3.2), $\mathbf{imp} \Rightarrow \sqsubseteq_h$ $\}$

  $(\forall g : g \in \mathcal{RD}.\mathbf{imp} : \mathbf{h}(e \parallel g) \leqslant \mathbf{h}(f \parallel g))$

  $\Leftrightarrow$ $\{$Definition 3.17$\}$

  $e \preceq_{\mathcal{RD}.\mathbf{imp}} f$

- $(\mathbf{imp} \Rightarrow \preceq_{\mathcal{RD}.\mathbf{imp}}) \;\Rightarrow\; (\mathbf{imp} \Rightarrow \sqsubseteq_h)$.

  Assume the left-hand side; we derive:

  $e\,\mathbf{imp}\,f$

  $\Rightarrow$ $\{\ \Vert\!\restriction$ is monotonic$\}$

  $e\,\mathbf{imp}\,f \;\;\wedge\;\; e \Vert\!\restriction \varnothing\ \mathbf{imp}\ f \Vert\!\restriction \varnothing$

  $\Rightarrow$ $\{$assumption, $e \Vert\!\restriction \varnothing = f \Vert\!\restriction \varnothing$ $\}$

  $\mathbf{h}(e \parallel e \Vert\!\restriction \varnothing) \;\leqslant\; \mathbf{h}(f \parallel f \Vert\!\restriction \varnothing)$

  $\Leftrightarrow$

  $\mathbf{h}e \leqslant \mathbf{h}f$

  $\Leftrightarrow$

  $e \sqsubseteq_h f$

$\square$

3.3 Maximal implementation relations

We are interested in those implementation relations that allow for as many implementations as possible; these turn out to be the non-robust relation $\lhd$ and its robust derivative $\lhd\!\shortmid$. Though we are not particularly interested in non-robust IR's, we use $\lhd$ when defining and analyzing the robust IR $\lhd\!\shortmid$. The latter is used in the sequel for comparing enabling functions. Theorem 3.37 states that it is the best robust implementation relation: when $e\,\mathbf{imp}\,f$, for any robust implementation relation $\mathbf{imp}$, then also $e \lhd\!\shortmid f$. At the end of this section we give properties that can be used to simplify the computation of $\lhd\!\shortmid$ in special cases.

A maximal (non-robust) implementation relation

Before giving the definition of $\lhd$, we rewrite the definition of $\sqsubseteq_P$ (Definition 3.11):

$e \sqsubseteq_P f \quad \Leftrightarrow \quad (\forall s,t : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge s = t : e.s \leqslant f.t)$

e is only demanded to enable at least as fast as f for identical schedules. In the definition of $\lhd$, e must also enable at least as fast as f for faster schedules:

Definition 3.27 $\lhd$.

$$e \lhd f \quad \Leftrightarrow \quad (\forall s,t : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge s \leqslant t : e.s \leqslant f.t)$$

□

Theorem 3.28

Relation $\lhd$ is an implementation relation.

□

In the proof of transitivity of relation $\lhd$ we use the following lemma:

Lemma 3.29

$$e \lhd f \wedge \mathbf{P}f \supseteq \mathbf{P}g \quad \Rightarrow \quad e \lhd g$$

Proof

We derive:

$e \lhd f \quad \wedge \quad \mathbf{P}f \supseteq \mathbf{P}g$

$\Leftrightarrow \qquad \{\text{definition of } \lhd, \text{ Corollary 2.71.2}\}$

$(\forall s,t : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge s \leqslant t : e.s \leqslant f.t) \wedge$

$(\forall t : t \in \mathbf{P}g : t \in \mathbf{P}f \wedge f.t \leqslant g.t)$

$\Rightarrow$

$(\forall s,t : s \in \mathbf{P}e \wedge t \in \mathbf{P}g \wedge s \leqslant t : e.s \leqslant f.t \leqslant g.t)$

$\Rightarrow$

$e \lhd g$

□

Proof of Theorem 3.28

Let e and f be enabling functions over the same alphabet.

- First we prove $\lhd \Rightarrow \sqsubseteq_h$. That is: $e \lhd f \Rightarrow \mathbf{h}e \leqslant \mathbf{h}f$.

  The proof is by induction, assume $e \lhd f$.

  base $\mathbf{h}e \downharpoonright \mathbf{b}f \quad \leqslant \quad \mathbf{h}f \downharpoonright \mathbf{b}f$

  step

  $\mathbf{h}e \downharpoonright M \quad \leqslant \quad \mathbf{h}f \downharpoonright M$

  $\Rightarrow \qquad \{ e \lhd f, \text{ Proposition 2.34.2}\}$

  $e(\mathbf{h}e \downharpoonright M) \quad \leqslant \quad f(\mathbf{h}f \downharpoonright M)$

$\Rightarrow$ { definition of Δ }

$e.\mathbf{h}e \downharpoonright (M+\Delta) \leqslant f.\mathbf{h}f \downharpoonright (M+\Delta)$

$\Leftrightarrow$ { definition of **h** (2.27) }

$\mathbf{h}e \downharpoonright (M+\Delta) \leqslant \mathbf{h}f \downharpoonright (M+\Delta)$

(end of step)

Remains to prove that $\lhd$ is a general implementation relation.

- **uniformity**

  Due to Corollary 2.71.1 it is evident that $\lhd$ is uniform with respect to $\approx$. Uniformity with respect to renaming and scaling is clearly perceptible.

- **monotonicity of** $\|$

  Let $A = (\cup x : : A_x)$ and let $e = (\| x : : e_x)$ and $f = (\| x : : f_x)$ such that $\mathbf{a}e_x = \mathbf{a}f_x = A_x$, we derive:

  $(\forall x : : e_x \lhd f_x)$

  $\Leftrightarrow$ { definition of $\lhd$ (3.27) }

  $(\forall x, s', t' : s' \in \mathbf{P}e_x \wedge t' \in \mathbf{P}f_x \wedge s' \leqslant t' : e_x.s' \leqslant f_x.t')$

  $\Rightarrow$ { $s \upharpoonright \mathbf{a}e_x = s'$, $t \upharpoonright \mathbf{a}e_x = t'$ }

  $(\forall s,t : \mathbf{a}s = A \wedge (\forall x : : s \upharpoonright A_x \in \mathbf{P}e_x) \wedge$
  $\mathbf{a}t = A \wedge (\forall x : : t \upharpoonright A_x \in \mathbf{P}f_x) \wedge$
  $(\forall x : : s \upharpoonright A_x \leqslant t \upharpoonright A_x)$
  $: (\forall x : : e_x(s \upharpoonright A_x) \leqslant f_x(t \upharpoonright A_y)))$

  $\Rightarrow$ { definition of $\|$ (2.30) }

  $(\forall s,t : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge s \leqslant t : e.s \leqslant f.t)$

  $\Leftrightarrow$
  $e \lhd f$

- **monotonicity of** $\|\!\upharpoonright$

  Let e and f be enabling functions such that $e \lhd f$, and let $s \in \mathbf{P}(e \|\!\upharpoonright A)$ and $t \in \mathbf{P}(f \|\!\upharpoonright A)$ such that $s \leqslant t$. We derive:

  true

  $\Rightarrow$ { $e \lhd f$, $\mathbf{n}s \lhd \mathbf{n}t$, $\|$ is monotonic }

  $e \| \mathbf{n}s \lhd f \| \mathbf{n}t$

  $\Rightarrow$ { $\lhd \Rightarrow \sqsubseteq_h$ }

  $\mathbf{h}(e \| \mathbf{n}s) \leqslant \mathbf{h}(f \| \mathbf{n}t)$

  $\Leftrightarrow$
  $\mathbf{h}((e \shortparallel A) \| \mathbf{n}s) \leqslant \mathbf{h}((f \shortparallel A) \| \mathbf{n}t)$

  $\Leftrightarrow$ { Proposition 2.56.2 }

  $s \uparrow (e \shortparallel A) \leqslant t \uparrow (f \shortparallel A)$

  $\Rightarrow$ { $e \lhd f$, Proposition 2.56.1, $\mathbf{P}(e \shortparallel A) \subseteq \mathbf{P}e$, $\mathbf{P}(f \shortparallel A) \subseteq \mathbf{P}f$ }

$$
\begin{array}{ll}
 & e(s \uparrow (e \shortparallel A)) \;\leqslant\; f(t \uparrow (f \shortparallel A)) \\
\Leftrightarrow & \\
 & e(s \uparrow e) \;\leqslant\; f(t \uparrow f) \\
\Rightarrow & \quad \{\text{Proposition } 2.62\} \\
 & (e \mathbin{\|\!\upharpoonright} A).s \;\leqslant\; (f \mathbin{\|\!\upharpoonright} A).t)
\end{array}
$$

- **transitivity**

  Transitivity is the only property that is not yet proven. Together with Remark 3.21 and Theorem 3.19, this justifies the first step in the derivation below.

$$
\begin{array}{ll}
 & e \lhd f \;\;\wedge\;\; f \lhd g \\
\Rightarrow & \quad \{\text{see above}\} \\
 & e \lhd f \;\;\wedge\;\; \mathbf{P}f \supseteq \mathbf{P}g \\
\Rightarrow & \quad \{\text{Lemma } 3.29\} \\
 & e \lhd g
\end{array}
$$

□

It is rather evident that $\lhd$ is reflexive for the enabling functions that are ascending on their processes.

Proposition 3.30 (see Proposition 2.82.1)

The reflexive domain of $\lhd$ is $[\mathit{Asc}]$.

□

Theorem 3.32 states why $\lhd$ is such a special implementation relation. In this theorem occurs the following —weird— relation on enabling functions:

Definition 3.31 $\preceq^{\infty}$.

$$e \preceq^{\infty} f \;\;\Leftrightarrow\;\; \mathbf{a}e = \mathbf{a}f \wedge (\forall s, a : s \in \mathbf{P}f : e.s.a = f.s.a \vee f.s.a = \infty)$$

□

The second item of the theorem can be read as 'when an implementation relation does not imply $\lhd$, it is almost as useless as $\approx$ '.

Theorem 3.32

1 $(\mathbf{imp} \Leftrightarrow_{[\mathit{Const}]} \lhd) \;\Rightarrow\; (\mathbf{imp} \Rightarrow \lhd)$

2 $(\mathbf{imp} \Rightarrow \preceq^{\infty}) \;\vee\; (\mathbf{imp} \Rightarrow \lhd)$

3 $\mathbf{imp} \Rightarrow_{[\mathit{Asc}]} \lhd$

Proof

1 Assume the left-hand side.

Conclude that for e' and f' in $[\mathit{Const}]$: $e' \,\mathbf{imp}\, f' \;=\; \mathbf{h}e' \leqslant \mathbf{h}f'$.

Now let e and f such that $e \,\mathbf{imp}\, f$ and let $s \in \mathbf{P}e$, $t \in \mathbf{P}f$ with $s \leqslant t$ and let a be a member of their alphabet. Define the constant enabling functions e' and f' over the same alphabet as e and f by:

$e'.u.b \;=\; \textbf{if}\;\; b \neq a \;\rightarrow\; s.b \;\;[]\;\; b = a \;\rightarrow\; f.t.a \;\;\textbf{fi}$

$f'.u.b \;=\; \textbf{if}\;\; b \neq a \;\rightarrow\; t.b \;\;[]\;\; b = a \;\rightarrow\; f.t.a \;\;\textbf{fi}$

Observe $\mathbf{h}e' \leqslant \mathbf{h}f'$ and thus $e' \,\mathbf{imp}\, f'$.

Observe furthermore that:

$\mathbf{h}(e \parallel e').a \;=\; e.s.a$ lub $f.t.a$

$\mathbf{h}(f \parallel f').a \;=\; f.t.a$

Conclude from $(e \parallel e')\;\mathbf{imp}\;(f \parallel f')$ (and $\lhd \Rightarrow \sqsubseteq_h$) that $e.s.a \leqslant f.t.a$.

2 Due to the previous item, and to $\lhd \Leftrightarrow_{[\mathit{Const}]} R_4$, it suffices to prove:

$\neg(\mathbf{imp} \Rightarrow \preceq^{\infty}) \;\Rightarrow\; \mathbf{imp} \Leftrightarrow_{[\mathit{Const}]} R_4$ (see Definition 3.22 for R_4).

Assume the left-hand side and let e and f such that $e \,\mathbf{imp}\, f$ but not $e \preceq^{\infty} f$.

Let s and a such that $s \in \mathbf{P}f$ and $e.s.a \neq f.s.a$ and $f.s.a < \infty$.

Observe that $\mathbf{P}f \neq \{\varepsilon\}$ and conclude from Lemma 3.26 that $R_5 \Rightarrow_{[\mathit{Const}]} \mathbf{imp}$ and thus that $\mathbf{imp}$ is reflexive on $[\mathit{Const}]$.

We derive:

$e \,\mathbf{imp}\, f$

$\Rightarrow$ { $\parallel$ is monotonic, $\mathbf{imp}$ is reflexive on $[\mathit{Const}]$ }

$e \parallel \mathbf{n}(s \setminus a) \;\;\mathbf{imp}\;\; f \parallel \mathbf{n}(s \setminus a)$

$\Rightarrow$ { $\Vert\!\!\upharpoonright$ is monotonic }

$(e \parallel \mathbf{n}(s \setminus a)) \Vert\!\!\upharpoonright a \;\;\mathbf{imp}\;\; (f \parallel \mathbf{n}(s \setminus a)) \Vert\!\!\upharpoonright a$

From Theorem 3.19 we infer $s \in \mathbf{P}e$; we conclude that

$\mathbf{h}((e \parallel \mathbf{n}(s \setminus a)) \Vert\!\!\upharpoonright a) = (a, e.s.a)$ and $\mathbf{h}((f \parallel \mathbf{n}(s \setminus a)) \Vert\!\!\upharpoonright a) = (a, f.s.a)$.

So, we have two constant enabling functions e' and f' such that $e' \,\mathbf{imp}\, f'$ and $\mathbf{h}e' < \mathbf{h}f'$ and $\mathbf{h}f' \neq \varepsilon$.

From Theorem 3.24 we conclude $\mathbf{imp} \Leftrightarrow_{[\mathit{Const}]} R_4$.

3 Due to Theorem 3.19 it suffices to observe $\sqsubseteq_P \Rightarrow_{[\mathit{Asc}]} \lhd$.

This follows from Lemma 3.29 and Proposition 3.30.

□

The following example shows that $\lhd$ is not robust.

Example 3.33 Speed Twisting by speeding up: $\lhd$ and SU.

We exhibit an enabling function e and a speeding up ρ of e such that e $\lhd$ e but $\neg(\rho.e \lhd e)$.

Define enabling function e over $\{a,b\}$ by:

$$\begin{array}{lcl} e.s.a & = & 1 \\ e.s.b & = & \textbf{if}\ \ s.a \leqslant 2 \ \rightarrow\ s.a+1 \\ & & [\!]\ \ s.a > 2 \ \rightarrow\ s.a+3 \\ & & \textbf{fi} \end{array}$$

Observe (with Definition 2.41) that the speeding up $(\frac{1}{2} \odot e)$ of e is given by:

$$\begin{array}{lcl} (\frac{1}{2}\odot e).s.a & = & 0.5 \\ (\frac{1}{2}\odot e).s.b & = & \textbf{if}\ \ s.a \leqslant 1 \ \rightarrow\ s.a+0.5 \\ & & [\!]\ \ s.a > 1 \ \rightarrow\ s.a+1.5 \\ & & \textbf{fi} \end{array}$$

We compose these enabling functions with $\mathbf{n}(a,1.5)$. The histories of the compositions are given by:

$$\begin{array}{rcl} \mathbf{h}(e \parallel \mathbf{n}(a,1.5)) & = & \{(a,1.5),(b,2.5)\} \\ \mathbf{h}((\frac{1}{2}\odot e) \parallel \mathbf{n}(a,1.5)) & = & \{(a,1.5),(b,3)\} \end{array}$$

Because $\mathbf{n}(a,1.5) \lhd \mathbf{n}(a,1.5)$, we must conclude $\neg(\frac{1}{2}\odot e \lhd e)$.

□

The maximal robust implementation relation

Now it is time for the 'real thing', the maximal robust IR.

Definition 3.34 $\lhd\!\!\mid$.

$$e \lhd\!\!\mid f \ = \ (\forall \rho : \rho \in SU.e : \rho.e \lhd f)$$

which is equivalent to $e \lhd\!\!\mid f \ = \ (\forall \rho : \rho \in SD.f : e \lhd \rho.f)$.

□

Theorem 3.35

Relation $\lhd\!\!\mid$ is a robust implementation relation.

□

In the proof of transitivity of relation $\lhd\!\!\mid$ we use the following lemma:

Lemma 3.36 (compare with Lemma 3.29)

$$e \lhd\!\!\mid f \wedge \mathbf{P}f \supseteq \mathbf{P}g \ \Rightarrow\ e \lhd\!\!\mid g$$

Proof

We derive:

$$e \lhd\!\!\mid f \wedge \mathbf{P}f \supseteq \mathbf{P}g$$

$\Rightarrow$ $\{$definition of $\lhd\!\!\mid$ (3.34)$\}$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd f) \wedge \mathbf{P}f \supseteq \mathbf{P}g$

$\Rightarrow$ $\{$Lemma 3.29$\}$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd g)$

$\Leftrightarrow$

$e \lhd\!\!\mid g$

□

Proof of Theorem 3.35

The implication $\lhd\!\!\mid \;\Rightarrow\; \sqsubseteq_h$ follows from $\lhd\!\!\mid \;\Rightarrow\; \lhd$. Uniformity holds trivially because of uniformity of $\lhd$, and so does monotonicity of $\|$, and monotonicity of $\|\!\!\upharpoonright$.

For transitivity we derive:

$e \lhd\!\!\mid f \wedge f \lhd\!\!\mid g$

$\Rightarrow$ $\{$Theorem 3.19, $\lhd\!\!\mid \;\Rightarrow \lhd$ $\}$

$e \lhd\!\!\mid f \wedge \mathbf{P}f \supseteq \mathbf{P}g$

$\Rightarrow$ $\{$Lemma 3.36$\}$

$e \lhd\!\!\mid g$

Remains to prove robustness; we derive:

$e \lhd\!\!\mid f$

$\Leftrightarrow$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd f)$

$\Leftrightarrow$ $\{$Proposition 2.42$\}$

$(\forall \rho,\sigma : \rho \in SU.e \wedge \sigma \in SU(\rho.e) : \sigma.\rho.e \lhd f)$

$\Leftrightarrow$ $\{\sigma.\rho.e = \sigma(\rho.e)\}$

$(\forall \rho : \rho \in SU.e : (\forall \sigma : \sigma \in SU(\rho.e) : \sigma(\rho.e) \lhd f))$

$\Leftrightarrow$ $\{$definition of $\lhd\!\!\mid$ $\}$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd\!\!\mid f)$

□

Theorem 3.37 Maximality of $\lhd\!\!\mid$.

For any robust implementation relation **imp** : **imp** $\Rightarrow$ $\lhd\!\!\mid$.

Proof

Let **imp** be a robust IR.

If, for all e and f, $e\,\mathbf{imp}\,f \Rightarrow \mathbf{P}f = \{\varepsilon\}$, **imp** $\Rightarrow \lhd\!\!\mid$ trivially holds.

Furthermore, we derive:

$(\exists e,f : e\,\mathbf{imp}\,f : \mathbf{P}f \neq \{\varepsilon\})$

$\Rightarrow$ $\{$Lemma 3.26$\}$

$R_5 \Rightarrow_{[\mathit{Const}]} \textbf{imp}$

$\Leftrightarrow \quad \{ \textbf{imp} \text{ is robust, Theorem 3.24.2} \}$

$R_4 \Leftrightarrow_{[\mathit{Const}]} \textbf{imp}$

$\Leftrightarrow \quad \{ \lhd \Leftrightarrow_{[\mathit{Const}]} R_4 \text{ , Theorem 3.32.1} \}$

$\textbf{imp} \Rightarrow \lhd$

and under the assumption of $\textbf{imp} \Rightarrow \lhd$ we derive:

$e\,\textbf{imp}\,f$

$\Rightarrow \quad \{\text{robustness}\}$

$(\forall \rho : \rho \in SU.e : \rho.e\,\textbf{imp}\,f)$

$\Rightarrow \quad \{ \textbf{imp} \Rightarrow \lhd \}$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd f)$

$\Leftrightarrow \quad \{\text{definition of } \lhd\!| \}$

$e \lhd\!| f$

□

The definition of $\lhd\!|$ is 'derived' from the definition of robustness (see Definition 3.20), in that all possible speeding ups (respectively slowing downs) are considered. It turns out to be sufficient to consider translations only:

Proposition 3.38

$$e \lhd\!| f \quad \Leftrightarrow \quad (\forall \mu : \mu \geqslant 0 : e \lhd f \oplus \mu)$$

which can be rewritten into:

$$e \lhd\!| f \quad \Leftrightarrow \quad (\forall s,t,\mu : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge \mu \geqslant 0 \wedge s \leqslant t \oplus \mu : e.s \leqslant f.t \oplus \mu)$$

Proof

The $\Rightarrow$ part of the equivalence is evident. Remains to prove the $\Leftarrow$ implication. This implication can be rewritten into:

$$(\forall s,t,\mu : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge \mu \geqslant 0 \wedge s \leqslant t \oplus \mu : e.s \leqslant f.t \oplus \mu) \Rightarrow$$
$$(\forall s,t,\rho : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge \rho \in SD.f \wedge s \leqslant \rho.t : e.s \leqslant \rho(f.t))$$

Assume the left-hand side. Let $s \in \mathbf{P}e$, $t \in \mathbf{P}f$, and $\rho \in SD.f$. Let furthermore $\Delta = \mathbf{d}e \min \mathbf{d}f$.

For $M \geqslant \mathbf{b}f$ we derive:

$s \leqslant \rho.t$

$\Rightarrow \quad \{ \rho \text{ has magnification at least one} \}$

$s \downharpoonright (\rho.M) \;\leqslant\; t \downharpoonright M \oplus (\rho.M - M)$

$\Rightarrow \quad \{\text{left-hand side, } \rho.M \geqslant M \}$

$e(s \downharpoonright (\rho.M)) \;\leqslant\; f(t \downharpoonright M) \oplus (\rho.M - M)$

$\Rightarrow$

$e.s \downharpoonright (\rho.M + \Delta) \;\leqslant\; f.t \downharpoonright (M + \Delta) \oplus (\rho.M - M)$

For all a in the alphabet of e we conclude, with $M = f.t.a$:

$$\begin{array}{ll} & s \leqslant \rho.t \\ \Rightarrow & \quad \{\text{previous derivation}\} \\ & e.s.a \downarrow (\rho(f.t.a) + \Delta) \;\leqslant\; f.t.a \downarrow (f.t.a + \Delta) + (\rho(f.t.a) - f.t.a) \\ \Leftrightarrow & \\ & e.s.a \downarrow (\rho(f.t.a) + \Delta) \;\leqslant\; \rho(f.t.a) \\ \Leftrightarrow & \\ & e.s.a \leqslant \rho(f.t.a) \end{array}$$

□

By means of an example we show that it is not sufficient to consider magnifications only.

Example 3.39 $\lhd\!\!\shortmid$ and magnifications.

Let enabling function e be defined by:

$e.s.a \;=\; 1$ and $e.s.b \;=\; 2 * s.a \max 1$

This function is not a member of the reflexive domain of $\lhd\!\!\shortmid$ because in general, for $s \in \mathbf{Pe}$ and $\mu \geqslant 0$, $e.(s \oplus \mu) \leqslant e.s \oplus \mu$ does not hold:

$$e.(s \oplus \mu).b \;=\; 2 * s.a + 2 * \mu \max 1$$

$$(e.s \oplus \mu).b \;=\; 2 * s.a + \mu \max 1 + \mu$$

However, the following formula, which is in terms of magnifications only, does hold:

$$(\forall s, t, \lambda : s \in \mathbf{Pe} \wedge t \in \mathbf{Pe} \wedge \lambda \geqslant 1 \wedge s \leqslant t \odot \lambda : e.s \leqslant e.t \odot \lambda)$$

Verification of this result is left to the reader.

□

The reflexive domain of $\lhd\!\!\shortmid$ is the set of enabling functions that are conservative on their processes:

Proposition 3.40 (follows from Proposition 3.38 and Proposition 2.82)

The reflexive domain of $\lhd\!\!\shortmid$ is $[\,Con\,]$.

□

Computing $\lhd$ and $\lhd\!\!\shortmid$

We give two propositions that help to simplify the computation of $\lhd$ and $\lhd\!\!\shortmid$ respectively. Several of the properties that are given have already been mentioned before.

Proposition 3.41

1 $e \lhd f \;\Rightarrow\; \mathbf{Pe} \supseteq \mathbf{P}f$
$e \lhd f \;\Leftrightarrow\; \mathbf{Pe} \supseteq \mathbf{P}f$ if $e \in [\,Asc\,]$.
(remark: $\mathbf{Pe} \supseteq \mathbf{P}f$ is equivalent to $e \leqslant_{\mathbf{P}f} f$)

2 $e \lhd f \;\Leftarrow\; e \leqslant_{\mathbf{Pe}} f$ if $f \in Asc.\mathbf{Pe}$,
$e \lhd f \;\Rightarrow\; e \leqslant_{\mathbf{Pe}} f$ if $f \in \mathcal{N}$, and thus
$e \lhd f \;\Leftrightarrow\; e \leqslant_{\mathbf{Pe}} f$ if $f \in \mathcal{N}.Asc$.

3 $e \lhd f \;\Leftrightarrow\; e \leqslant f$ if $e, f \in \mathcal{N}$ and at least one of them in Asc.

Proof

1 The first result is a part of Theorem 3.19; it implies the $\Rightarrow$ implication in the second result. For the $\Leftarrow$ implication in the second result we refer to Lemma 3.29 and Proposition 3.30.

2 First formula:
Let $f \in Asc.\mathbf{Pe}$; we derive:

$$\begin{array}{ll}
 & e \leqslant_{\mathbf{Pe}} f \\
\Leftrightarrow & \\
 & (\forall s : s \in \mathbf{Pe} : e.s \leqslant f.s) \\
\Leftrightarrow & \quad \{\, f \in Asc_{\mathbf{Pe}} \text{ and } \mathbf{Pe} \supseteq \mathbf{P}f \text{ (Corollary 2.71.2)} \,\} \\
 & (\forall s,t : s \in \mathbf{Pe} \wedge t \in \mathbf{P}f \wedge s \leqslant t : e.s \leqslant f.s \leqslant f.t) \\
\Rightarrow & \\
 & e \lhd f
\end{array}$$

2 Second formula:
Let $f \in \mathcal{N}$; we derive:

$$\begin{array}{ll}
 & e \lhd f \\
\Leftrightarrow & \\
 & (\forall s,t : s \in \mathbf{Pe} \wedge t \in \mathbf{P}f \wedge s \leqslant t : e.s \leqslant f.t) \\
\Rightarrow & \quad \{\text{Proposition 2.80.3}\} \\
 & (\forall s : s \in \mathbf{Pe} : e.s \leqslant f(s \text{ lub } f.s)) \\
\Leftrightarrow & \quad \{\text{Proposition 2.80.2}\} \\
 & (\forall s : s \in \mathbf{Pe} : e.s \leqslant f.s)
\end{array}$$

3 For the $\Leftarrow$ implication we refer to both previous items. Remains to prove the $\Rightarrow$ implication.
Assume $e, f \in \mathcal{N}$ and $e \in [\,Asc\,]$, and assume $e \lhd f$. We derive:

$$\begin{array}{ll}
 & e.s \\
= & \quad \{\text{Definition 2.78}\} \\
 & e(\mathbf{h}(\mathbf{n}s \parallel e))
\end{array}$$

$\leqslant$ $\{\, e \in Asc\,,\ \mathbf{n}s \parallel e \lhd \mathbf{n}s \parallel f \,\}$

$e(\mathbf{h}(\mathbf{n}s \parallel f))$

$\leqslant$ $\{\, e \lhd f\,,\ \mathbf{h}(\mathbf{n}s \parallel f) \in \mathbf{P}f\,,$ Proposition 3.41.1 $\}$

$f(\mathbf{h}(\mathbf{n}s \parallel f))$

$=$ $\{$Definition 2.78$\}$

$f.s$

For $f \in [\,Asc\,]$, instead of $e \in [\,Asc\,]$, the derivation is similar (with usage of Proposition 3.41.2 instead of 3.41.1).

□

Proposition 3.42

$$\lhd\!| \quad \Rightarrow \quad \lhd$$
$$e \lhd\!| f \quad \Leftrightarrow \quad e \lhd f \qquad \text{if } e \in [\,Con\,] \text{ or } f \in [\,Con\,]\,.$$

Proof

The first implication is evident; remains to prove the second one.

Let e, f be enabling functions over the same alphabet, and let $e \in [\,Con\,]$. We derive:

$e \lhd\!| f$

$\Leftrightarrow$ $\{$definition of $\lhd\!|$ (3.34)$\}$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd f\,)$

$\Leftrightarrow$ $\{$the identity is a speeding up$\}$

$(\forall \rho : \rho \in SU.e : \rho.e \lhd f \wedge e \lhd f\,)$

$\Leftrightarrow$ $\{\, \rho.e \lhd e\,,\ \lhd$ is transitive$\}$

$e \lhd f$

For $f \in [\,Con\,]$ the derivation is similar (with a slowing down at the right-hand side instead of a speeding up at the left-hand side).

□

From these propositions, and from Lemmata 3.29, and 3.36, we conclude that process inclusion (the second criterion: speed in any possible situation) was not such a bad guess for comparing enabling functions.

We conclude this section with an example of the usage of $\lhd\!|$.

Example 3.43 Comparison of four enabling functions.

We compare the following enabling functions over generic actions a and b.

- e is the enabling function of $(a\,,b)^*$:

$$\begin{array}{lcl} e.s.a_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ 1 \\ & & [\!]\ \ i > 0 \ \rightarrow\ s.a_{i-1} + 1 \ \max\ s.b_{i-1} + 1 \\ & & \textbf{fi} \\ e.s.b_i & = & e.s.a_i \end{array}$$

- g is similar to e, but with additional delay between occurrences of the same generic action:

```
g.s.a_i  =  if  i = 0  →  1
            []  i > 0  →  s.a_{i-1} + 2  max  s.b_{i-1} + 1
            fi
g.s.b_i  =  if  i = 0  →  1
            []  i > 0  →  s.a_{i-1} + 1  max  s.b_{i-1} + 2
            fi
```

- f is a hybrid of e and g:

```
f.s.a_i  =  if  i = 0  →  1
            []  i > 0  →  if  s.a_{i-1} < s.b_{i-1}  →  s.a_{i-1} + 2
                          []  s.a_{i-1} ≥ s.b_{i-1}  →  s.a_{i-1} + 1
                          fi  max  s.b_{i-1} + 1
            fi
f.s.b_i  =  if  i = 0  →  1
            []  i > 0  →  if  s.b_{i-1} < s.a_{i-1}  →  s.b_{i-1} + 2
                          []  s.b_{i-1} ≥ s.a_{i-1}  →  s.b_{i-1} + 1
                          fi  max  s.a_{i-1} + 1
            fi
```

- h is the enabling function of $(a\ ;\ b)^*$:

```
h.s.a_i  =  if  i = 0  →  1
            []  i > 0  →  s.b_{i-1} + 1
            fi
h.s.b_i  =  s.a_i + 1
```

All these enabling functions are conservative, except for f which is not even ascending. The mutual relations are given by $e \lhd f \lhd g \lhd h$.

Observe that $g \lhd h$, but the histories of both are given by:

$$\begin{array}{lll} \mathrm{h}g.a_i = 2i+1 & & \mathrm{h}h.a_i = 2i+1 \\ \mathrm{h}g.b_i = 2i+1 & & \mathrm{h}h.b_i = 2i+2 \end{array}$$

So when placed in a greedy environment, the delays between successive actions are 2 for g and 1 for h. With scheduling, for example, b_0 at 1, g takes an advance with respect to h; this advance is reclaimed by not enabling a_1 until 3. This already illustrates the 'amortized' or 'linear time' aspect of $\lhd$ that is discussed in the second part of Section 3.7.

□

3.4 Liberal implementations and severe specifications

Given a specification f, it is an interesting question whether or not there exists a unique (modulo $\approx$) *most liberal implementation* of f. That is, an enabling function $\triangleq f$ that satisfies:

$\triangleq f \lhd\!\mid f$ and $e \lhd\!\mid \triangleq f \;\Leftrightarrow\; e \lhd\!\mid f$.

More general, when given a set of specifications, $\mathcal{F}$, the question arises whether or not there exists a unique most liberal implementation $\triangleq \mathcal{F}$ that implements all these specifications:

$(\forall f : f \in \mathcal{F} : \triangleq \mathcal{F} \lhd\!\mid f)$ and $e \lhd\!\mid \triangleq \mathcal{F} \;\Leftrightarrow\; (\forall f : f \in \mathcal{F} : e \lhd\!\mid f)$.

Most liberal implementations are in fact greatest lower bounds with respect to $\lhd\!\mid$.

The same questions arise considering the existence of *most severe specifications* (least upper bounds) of (sets of) implementations.

In this section we show the existence of most liberal implementations and most severe specifications, under rather weak conditions. The results of this section are not used in the remainder of this thesis.

Example 3.44 Most severe specifications and most liberal implementations.

Enabling function e is inspired on the behaviour of bypass buffers, as given in Section 5.5.

$$\begin{array}{lcl} e.s.a &=& 1 \\ e.s.b &=& 1 \\ e.s.c &=& \textbf{if}\ \ s.a < s.b \ \rightarrow\ s.a + 4 \\ && [\!]\ \ s.a \geqslant s.b \ \rightarrow\ s.a + 2 \\ && \textbf{fi} \end{array}$$

e implements both f and g :

$$\begin{array}{lcl@{\qquad}lcl} f.s.a &=& 1 & g.s.a &=& 1 \\ f.s.b &=& 1 & g.s.b &=& 1 \\ f.s.c &=& s.a + 4 & g.s.c &=& s.a + 2 \ \max\ s.b + 4 \end{array}$$

Enabling function h , as given below, is not only the most severe specification of e , it is also the most liberal implementation of $\{f,g\}$:

$$\begin{array}{lcl} h.s.a &=& 1 \\ h.s.b &=& 1 \\ h.s.c &=& s.a + 2 \ \max\ (s.a + 4 \ \min\ s.b + 4) \end{array}$$

□

In general, however, the most liberal implementations and the most severe specifications of sets of enabling functions do not exist:

Example 3.45

Consider the set of enabling functions $e_j : j \geqslant 0$ for:

$$\begin{array}{lcl} e_j.a_i &=& \textbf{if}\ \ i = 0 \ \rightarrow\ -j \\ && [\!]\ \ i > 0 \wedge i = j \ \rightarrow\ s.a_{i-1} + 1/j \\ && [\!]\ \ i > 0 \wedge i \neq j \ \rightarrow\ -j \\ && \textbf{fi} \end{array}$$

This set of enabling functions has no 'lower bounds', and thus no most liberal implementation. The most severe specification would be given by:

```
e.$a_i$  =  if  $i = 0$  →  $0$
            []  $i > 0$  →  $s.a_{i-1} + 1/i$
            fi
```

which is *no* enabling function because it has delay ($\mathbf{de}$) zero.

□

Apart from boundary cases with problematic delays or problematic bases, as in the example, all sets of enabling functions over the same alphabet, have (modulo $\approx$) a unique most liberal implementation and a unique most severe specification. For enabling functions, the most liberal implementations are given by:

Definition 3.46 $\trianglelefteq$ for enabling functions.

$$\trianglelefteq f.s \;=\; (\,\mathbf{glb}\, t,\mu : t \in \mathbf{P}f \wedge \mu \geqslant 0 \wedge s \leqslant t \oplus \mu : f.t \oplus \mu\,)$$

□

We leave it to the reader to verify that $\trianglelefteq f$ is a conservative enabling function with $\mathbf{b} \trianglelefteq f = \mathbf{b}f$ and $\mathbf{d} \trianglelefteq f \geqslant \mathbf{d}f$.

Proposition 3.47

$\trianglelefteq f$ is a most liberal implementation of f:

$$\trianglelefteq f \;\lhd\!\!\shortmid\; f \quad \text{and} \quad e \lhd\!\!\shortmid f \;\Leftrightarrow\; e \lhd\!\!\shortmid \trianglelefteq f$$

Proof

Because $\trianglelefteq f$ is conservative, it suffices to prove the second formula only. We derive:

$$
\begin{array}{ll}
 & e \lhd\!\!\shortmid f \\
\Leftrightarrow & \quad \{\,\text{Proposition 3.38}\,\} \\
 & (\forall s,t,\mu : s \in \mathbf{P}e \wedge t \in \mathbf{P}f \wedge \mu \geqslant 0 \wedge s \leqslant t \oplus \mu : e.s \leqslant f.t \oplus \mu\,) \\
\Leftrightarrow & \\
 & (\forall s : s \in \mathbf{P}e \\
 & \qquad : e.s \leqslant (\,\mathbf{glb}\, t,\mu : t \in \mathbf{P}f \wedge \mu \geqslant 0 \wedge s \leqslant t \oplus \mu : f.t \oplus \mu\,)) \\
\Leftrightarrow & \\
 & (\forall s : s \in \mathbf{P}e : e.s \leqslant \trianglelefteq f.s\,) \\
\Leftrightarrow & \quad \{\,\text{Propositions 3.42 and 3.41.2, } \trianglelefteq f \in \mathit{Asc}\,\} \\
 & e \lhd\!\!\shortmid \trianglelefteq f
\end{array}
$$

□

The definition of $\trianglelefteq$ can be extended to sets of enabling functions as follows:

Definition 3.48 $\trianglelefteq$ for sets.

For $\mathcal{F}$ a set of enabling functions over the same alphabet we define:

$$\trianglelefteq \mathcal{F}.s \;=\; (\,\mathbf{glb}\, f,t,\mu : f \in \mathcal{F} \wedge t \in \mathbf{P}f \wedge \mu \geqslant 0 \wedge s \leqslant t \oplus \mu \;\; : f.t \oplus \mu\,)$$

□

$\triangle \mathcal{F}$ is a structure that satisfies:

$$\mathbf{b} \triangle \mathcal{F} \geqslant (\mathbf{glb}\, f : f \in \mathcal{F} : \mathbf{b}f) \quad \text{and} \quad \mathbf{d} \triangle \mathcal{F} \geqslant (\mathbf{glb}\, f : f \in \mathcal{F} : \mathbf{d}f) \ .$$

In case $\mathbf{b} \triangle \mathcal{F} > -\infty$ and $\mathbf{d} \triangle \mathcal{F} > 0$, $\triangle \mathcal{F}$ is an enabling function and it is the most liberal implementation of $\mathcal{F}$. The proof of this statement is similar to the proof of Proposition 3.47. From this result we infer that sets of enabling functions of which base and delay have uniform lower bounds ($> -\infty$ and > 0 respectively) have a most liberal implementation.

For the dual concept of most strict specifications, we only give a definition. The details are left to the reader.

Definition 3.49 $\bar{\nabla}$ for sets.

For a non-empty set of enabling functions, $\mathcal{E}$, over the same alphabet, we define:

$$\begin{aligned}\bar{\nabla}\, \mathcal{E}.t \;=\; & (\,\mathbf{lub}\, e, s, \mu : e \in \mathcal{E} \wedge s \in \mathbf{P}e \wedge \mu \leqslant 0 \wedge s \oplus \mu \leqslant t \\ & \qquad\qquad : e.s \oplus \mu\,) \\ & \mathrm{lub}\, u_b\end{aligned}$$

where u_b is given by $u_b.a = (\,\mathbf{lub}\, e : e \in \mathcal{E} : \mathbf{b}e\,)$

□

The additional term ' $\mathrm{lub}\, u_b$ ' in this definition is included to assure that $\bar{\nabla}\, \mathcal{E}.t.a$ cannot become $-\infty$ (in case of a **lub** over an empty range).

3.5 The quality of implementations

In this section we assume mechanisms to be described by initiated enabling functions. The moment at which the first action is enabled is considered to be the initial delay of the mechanism, it is included in the performance analysis.

We introduce *angelic-* and *demonic response time*, both giving an extreme view on the speed of mechanisms. Both give, however, very little information about the actual behaviours of mechanisms. We are more interested in comparing the behaviour of an implementation to that of a specification. Therefore we also introduce the *quality* of an implementation relative to a specification, as well as the derived notion of *relative response time*.

In [23, 27], *constant response time* is introduced as a characterization of the progress of cubic trace-theory systems; a system has constant response time, if there exists a possible behaviour for which there exists a finite upper bound of the elapsed time between consecutive external actions. In this definition of constant response time, *sequence functions* (schedules in the time-domain of natural numbers) are used to describe possible behaviours. In terms of enabling functions we define the related concept of *angelic response time*. In the definition we use the response time of a schedule, which is the maximal delay between consecutive actions, including the initial delay. The angelic response time of an enabling function is the lower bound of the response time of all schedules in its process.

Definition 3.50 Angelic response time: $\mathrm{R^a}$.

The response time $\mathrm{R}.s$ of a schedule with $\mathbf{b}s > 0$ is defined by:

$$\begin{aligned}\mathrm{R}.s \;=\; \mathbf{b}s\ \max\ (\,\mathbf{lub}\, a : (\,\exists b : \;: s.b < s.a\,) \\ : (\,\mathbf{glb}\, b : s.b < s.a : s.a - s.b\,)\,)\end{aligned}$$

where $\mathbf{lub}\varnothing = 0$.

The *angelic response time* of initiated enabling function e is defined by:

$$\mathrm{R^a}.e \;=\; (\,\mathbf{glb}\, s : s \in \mathbf{P}e : \mathrm{R^a}.s\,)$$

Angelic response time may be extended to initiated enabling structures by

$$\mathrm{R^a}.E \;=\; \mathrm{R^a}.(E \restriction \mathbf{e}E)$$

Similar extensions are implicitly assumed in subsequent definitions.

□

A nice property of $\mathrm{R^a}$ is that $e \lhd f \Rightarrow \mathrm{R^a}.e \leqslant \mathrm{R^a}.f$. This follows from the observation that $\mathbf{P}e \supseteq \mathbf{P}f \Rightarrow \mathrm{R^a}.e \leqslant \mathrm{R^a}.f$. The problem, of course, is that there is no guarantee whatsoever that this response time will be met in a concrete situation: the actual time between consecutive external actions depends on the environment (so the term angelic does not refer to some 'angel' in the mechanism itself, but to the behaviour of the environment).

A solution seems to be the introduction of demonic response time. Observe that it does not make sense to take the upper bound of the response time of individual schedules in stead of the lower bound: ε is a member of the process, and the response time of ε is ∞! We have to use a more sophisticated view on response time. Demonic response time is the maximum time an environment can be forced to wait until it can perform the next communication. For example, enabling function g of Example 3.43, has demonic response time 2: when a_i and b_i are performed simultaneously, the environment has to wait two time units before it can perform the next communication (a_{i+1} or b_{i+1}). Enabling function h of the same example, has demonic response time 1. The demonic response time of an enabling function is the maximal delay between performing an action (b) and enabling of the next action (a); when this next action is not scheduled at the moment of enabling, this is due to the environment and the enabling function itself cannot be blamed for the additional delay.

Definition 3.51 Demonic response time: $\mathrm{R^d}$.

The *demonic response time* of an initiated enabling function is defined by

$$\begin{aligned}\mathrm{R^d}.e \;=\; \mathbf{b}e\ \max\ (\,\mathbf{lub}\, s,a : s \in \mathbf{P}e \wedge (\,\exists b : \;: s.b < e.s.a\,) \\ : (\,\mathbf{glb}\, b : s.b < e.s.a : e.s.a - s.b\,)\,)\end{aligned}$$

□

For $\mathrm{R^d}$, however, the nice property of $\mathrm{R^a}$ does not hold: there exist enabling functions g and h such that $g \lhd\!\shortmid\, h$ but $\mathrm{R^d}.g > \mathrm{R^d}.h$. Take for example enabling functions g and h from Example 3.43 (see also Table 3.55).

The quality of an 'implementation' relative to a 'specification' is defined in terms of the robust implementation relation $\lhd\!\shortmid$ and scaling. An implementation has high quality with respect to a specification if it is fast with respect to this specification. It has, on the other hand, zero quality if it has a totally unfit behaviour, or if it becomes unbounded slow (with respect to the specification).

Definition 3.52 Quality, relative response time: Q, R.

For e and f initiated enabling functions over the same alphabet, the *quality* of e relative to f is given by:

$$\mathrm{Q}(\mathrm{e},f) \;=\; (\,\mathbf{lub}\,\lambda : \lambda > 0 \wedge \lambda \odot \mathrm{e} \lhd\!\shortmid\, f : \lambda\,) \qquad \text{where } \mathbf{lub}.\varnothing = 0\,.$$

The response time of e *relative to* f is defined as

$$\mathrm{R}(\mathrm{e},f) \;=\; 1\,/\,\mathrm{Q}(\mathrm{e},f)$$

□

In general, the lub in the definition of quality is a maximum:

Proposition 3.53

If $0 < \mathrm{Q}(\mathrm{e},f) < \infty$ then $\mathrm{Q}(\mathrm{e},f) \odot \mathrm{e} \lhd\!\shortmid\, f$.

Proof

It suffices to prove:

$$(\forall \lambda : 0 < \lambda < 1 : \lambda \odot e \lhd\!\shortmid\, f\,) \;\Rightarrow\; e \lhd\!\shortmid\, f$$

And in order to prove this, it suffices to prove:

$$(\forall \lambda : 0 < \lambda < 1 : \lambda \odot e \lhd f\,) \;\Rightarrow\; e \lhd f$$

We derive:

$$
\begin{array}{ll}
 & (\forall \lambda : 0 < \lambda < 1 : \lambda \odot e \lhd f\,) \\
\Leftrightarrow & \{\text{definition of } \lhd \text{ (3.27)}\} \\
 & (\forall \lambda, s, t : 0 < \lambda < 1 \wedge s \in \mathbf{P}(\lambda \odot \mathrm{e}) \wedge t \in \mathbf{P}f \wedge s \leqslant t \\
 & \qquad : (\lambda \odot \mathrm{e}).s \leqslant f.t\,) \\
\Leftrightarrow & \{\text{definition of scaling, (2.41)}\} \\
 & (\forall \lambda, s, t : 0 < \lambda < 1 \wedge s \in \lambda \odot \mathbf{P}\mathrm{e} \wedge t \in \mathbf{P}f \wedge s \leqslant t \\
 & \qquad : \lambda \odot \mathrm{e}.(\tfrac{1}{\lambda} \odot s) \leqslant f.t\,) \\
\Leftrightarrow & \{\text{dummy change}\} \\
 & (\forall \lambda, s, t : 0 < \lambda < 1 \wedge s \in \mathbf{P}\mathrm{e} \wedge t \in \mathbf{P}f \wedge \lambda \odot s \leqslant t : \lambda \odot (\mathrm{e}.s) \leqslant f.t\,) \\
\Rightarrow & \\
 & (\forall \lambda, s, t : 0 < \lambda < 1 \wedge s \in \mathbf{P}\mathrm{e} \wedge t \in \mathbf{P}f \wedge s \leqslant t : \lambda \odot (\mathrm{e}.s) \leqslant f.t\,) \\
\Rightarrow & \{\text{ lub over } \lambda\text{, continuity of multiplication}\} \\
 & (\forall s, t : s \in \mathbf{P}\mathrm{e} \wedge t \in \mathbf{P}f \wedge s \leqslant t : \mathrm{e}.s \leqslant f.t\,)
\end{array}
$$

$\Leftrightarrow$ { definition of $\lhd$ }

$e \lhd f$

□

In contrast with angelic and demonic response time, quality (and thus relative response time) does not deal with the delays between individual actions. An implementation, however, can be forced to exhibit delays as expected from its quality relative to a specification, just by adding a controller (see Section 3.7). Furthermore, quality can be used in compositional design: it has a kind of transitivity, in a way $\|$ and $\|\!\!\upharpoonright$ are monotonic, and it is robust:

Proposition 3.54 (without proof)

1 'transitivity': $Q(e,g) \geqslant Q(e,f) * Q(f,g)$

2 'monotonicity of $\|$ and $\|\!\!\upharpoonright$':

$$Q(e_0 \| e_1, f_0 \| f_1) \geqslant Q(e_0, f_0) \min Q(e_1, f_1)$$
$$Q(e \|\!\!\upharpoonright A, f \|\!\!\upharpoonright A) \geqslant Q(e,f)$$

3 robustness: for any scaling up ρ of e: $Q(\rho.e,f) \geqslant Q(e,f)$

4 'reflexive domain':

$$Q(e,e) \leqslant 1 \quad \vee \quad Pe = \{\varepsilon\}$$
$$Q(e,e) = 1 \quad \Leftrightarrow \quad e \in [Con] \wedge Pe \neq \{\varepsilon\}$$

□

.	R^a.	R^d.	R(.,e)	R(.,f)	R(.,g)	R(.,h)
e	1	1	1	1	1	1
f	1	2	2	2	1	1
g	1	2	2	2	1	1
h	1	1	∞	∞	∞	1

Table 3.55 Response times for Example 3.43.

3.6 Variation of delays

Enabling structures can be used to describe (abstractions of) behaviours of 'real world' devices. However, the timing in these devices may vary dependent on temperature, voltage, complexity of data to be processed, and the like. So when having calculated the speed of some device, under the assumption of some particular timing (enabling structure), it is important to know what happens if timing in the device deviates.

The following example shows that a tiny deviation of timing may have a disastrous impact on the behaviour.

Example 3.56 Livelock.

Assume someone has designed a device that shares a resource between communications over its a channel and communications over its b channel. The device enables a actions when during a sufficiently large interval of time no b actions have occurred; similar for b actions. The device is characterized by delays α_i, $\bar{\alpha}_i$, β_i, and $\bar{\beta}_i$ (for $i \geqslant 0$) as follows:

$$\begin{array}{lcl} e.s.a_i & = & \textbf{if } i = 0 \rightarrow \alpha_i \\ & & [\!]\ i > 0 \rightarrow \mathcal{F}(s.a_{i-1}, b, \bar{\beta}, s) + \alpha_i \\ & & \textbf{fi} \\ e.s.b_i & = & \textbf{if } i = 0 \rightarrow \beta_i \\ & & [\!]\ i > 0 \rightarrow \mathcal{F}(s.b_{i-1}, a, \bar{\alpha}, s) + \beta_i \\ & & \textbf{fi} \end{array}$$

where function $\mathcal{F}$ is given by:

$$\mathcal{F}(M, c, \bar{\gamma}, s) \ = \ (\, \textbf{glb}\, N : N \geqslant M \wedge \neg(\exists\, i : \ : N - \bar{\gamma}_i < s.c_i < N\,) : N\,)$$

The designer claims it is a 'good' device in that it has a finite response time with respect to specification (a^*, b^*). This is due to the fact, he says, that all delays are equally Δ: $\alpha_i = \bar{\alpha}_i = \beta_i = \bar{\beta}_i = \Delta$ ($\Delta > 0$).

And indeed, this assumption gives the following estimation of the behaviour of the device:

$$\begin{array}{lcl} e.s.a_i & \leqslant & \textbf{if } i = 0 \rightarrow \Delta \\ & & [\!]\ i > 0 \rightarrow s.a_{i-1} + 2\Delta \\ & & \textbf{fi} \\ e.s.b_i & \leqslant & \textbf{if } i = 0 \rightarrow \Delta \\ & & [\!]\ i > 0 \rightarrow s.b_{i-1} + 2\Delta \\ & & \textbf{fi} \end{array}$$

for $s \in \mathbf{Pe}$.

So the response time of the device with respect to (a^*, b^*) is at most 2Δ.

However, this result depends critically on the assumption that all delays are exactly the same. Assume for example that the delays $\bar{\beta}_i$ are a tiny little bit more than Δ: that is, a actions are enabled when during a time interval longer than Δ no b actions have occurred. In this case the history of e is given by:

$$\begin{array}{lllll} \mathrm{he}.a_i & = & \mathbf{if}\ i \leqslant 1 \rightarrow (i+1)*\Delta & \qquad \mathrm{he}.b_i & = \ (i+1)*\Delta\ . \\ & & [\!]\ i > 1 \rightarrow \infty & & \\ & & \mathbf{fi} & & \end{array}$$

This looks more like an implementation of (a^2, b^*); the response time with respect to (a^*, b^*) is ∞.

In general, when the delays in the device vary around the intended value Δ, it may postpone communication along the a channel unbounded by performing communications along the b channel (and vice versa). This phenomenon is called livelock in [13, 16]. We suggest as an exercise to the reader to verify the following response times:

	$k = l = 0$	$k * l = 1$	$k \leqslant 1 < l$	$k > 1$
first version	0	Δ	2Δ	2Δ
second version	0	Δ	2Δ	∞

Table $\mathrm{R}(\mathrm{e}, \mathbf{n}(a^k, b^l))$, for $0 \leqslant k, l \leqslant \infty$ (where $\infty = *$).

□

If the timing of a device is dependent on, or can be estimated with, —relatively— fixed delays ($\bar{\pi}$ and $\hat{\pi}$ in Appendix B), the influence of variation of these delays is no more than proportional. Furthermore it suffices to know an upper bound (lower bound) of these delays in order to compute an upper bound (lower bound) of the performance of the device (with respect to a specification). More precisely, if, apart from timing, E and F have the same dependencies then:

1. if the delays in E are at most λ times the delays in F: $\mathrm{R}(E, F) \leqslant \lambda$;
2. if the delays in E are at least λ times the delays in F: $\mathrm{R}(E, F) \geqslant \lambda$.

Both claims can be divided into two parts. The first parts are rather evident:

1. if the delays in E are at most λ times the delays in F: $E \leqslant \lambda \odot F$;
2. if the delays in E are at least λ times the delays in F: $E \geqslant \lambda \odot F$.

The second parts are stated in the following proposition.

Proposition 3.57

Let E and F be enabling structures over the same alphabet, such that at least one of them has fixed delays.

1 $E \leqslant \lambda \odot F \ \Rightarrow\ \mathrm{R}(E, F) \leqslant \lambda$

2 $E \geqslant \lambda \odot F \ \Rightarrow\ \mathrm{R}(E, F) \geqslant \lambda$, if $\mathbf{P}F \restriction \mathbf{e}F \neq \{\varepsilon\}$.

Proof

Let e and f be the 'unmaskings' of E and F respectively:
$e = \langle \mathbf{a}E, \varnothing, \mathbf{f}E \rangle$ and $f = \langle \mathbf{a}F, \varnothing, \mathbf{f}F \rangle$.
Let A be the external alphabet of both enabling structures. The implications can be rewritten into:

$$\begin{array}{lll} 1 & e \leqslant \lambda \odot f & \Rightarrow \quad \mathrm{R}(e \mathbin{\|\!\!\upharpoonright} A, f \mathbin{\|\!\!\upharpoonright} A) \leqslant \lambda \\ 2 & e \geqslant \lambda \odot f & \Rightarrow \quad \mathrm{R}(e \mathbin{\|\!\!\upharpoonright} A, f \mathbin{\|\!\!\upharpoonright} A) \geqslant \lambda \end{array}$$

1 From Corollary 2.54 we infer that e or f is an element of *Con*. We derive:

$e \leqslant \lambda \odot f$

$\Rightarrow$ $\{$ e or f in *Asc*, since $Con \subseteq Asc$, Proposition 3.41.1 & .2 $\}$

$e \lhd \lambda \odot f$

$\Rightarrow$ $\{$ e or f in *Con*, Proposition 3.42 $\}$

$e \lhd\!\!\mid \lambda \odot f$

$\Rightarrow$

$e \mathbin{\|\!\!\upharpoonright} A \lhd\!\!\mid \lambda \odot f \mathbin{\|\!\!\upharpoonright} A$

$\Rightarrow$

$\mathrm{R}(e \mathbin{\|\!\!\upharpoonright} A, f \mathbin{\|\!\!\upharpoonright} A) \leqslant \lambda$

2 Assume $e \mathbin{\|\!\!\upharpoonright} A \lhd\!\!\mid \lambda' \odot f \mathbin{\|\!\!\upharpoonright} A$. It suffices to prove that the left-hand side implies $\lambda' \geqslant \lambda$. We derive:

$e \geqslant \lambda \odot f$

$\Leftrightarrow$

$f \leqslant \frac{1}{\lambda} \odot e$

$\Rightarrow$ $\{$ first implication $\}$

$f \mathbin{\|\!\!\upharpoonright} A \lhd\!\!\mid \frac{1}{\lambda} \odot e \mathbin{\|\!\!\upharpoonright} A$

$\Rightarrow$ $\{$ assumption $\}$

$f \mathbin{\|\!\!\upharpoonright} A \lhd\!\!\mid \frac{1}{\lambda} \odot e \mathbin{\|\!\!\upharpoonright} A \lhd\!\!\mid \frac{\lambda'}{\lambda} \odot f \mathbin{\|\!\!\upharpoonright} A$

$\Rightarrow$ $\{$ transitivity of $\lhd\!\!\mid$ $\}$

$f \mathbin{\|\!\!\upharpoonright} A \lhd\!\!\mid \frac{\lambda'}{\lambda} \odot f \mathbin{\|\!\!\upharpoonright} A$

$\Rightarrow$ $\{$ $\mathbf{P}(f \mathbin{\|\!\!\upharpoonright} A) \neq \{\varepsilon\}$, Proposition 3.54.4 $\}$

$\lambda' \geqslant \lambda$

□

The following example illustrates that taking the upper (lower) bounds for individual delays is a convenient way to determine an upper (lower) bound of the behaviour of a device (which is described in terms of fixed delays). In case of coupled delays, however, tighter bounds can be achieved by a more penetrating analysis.

Example 3.58 Variation of delays.

Consider a device with external alphabet a and internal alphabet b that behaves as:

$$\begin{array}{lcllll}
E_\delta.s.a_i & = & \textbf{if} & i=0 & \rightarrow & s.b_i+\delta(ba,i) \\
 & & [\!] & i>0 & \rightarrow & s.b_i+\delta(ba,i) \max s.a_{i-1}+\delta(aa,i) \\
 & & \textbf{fi} & & & \\
E_\delta.s.b_i & = & \textbf{if} & i=0 & \rightarrow & \delta(ab,i) \\
 & & [\!] & i>0 & \rightarrow & s.a_{i-1}+\delta(ab,i) \max s.b_{i-1}+\delta(bb,i) \\
 & & \textbf{fi} & & &
\end{array}$$

for some —unknown— delays $\delta(xy,i)$ of which only an upper bound, Δ, is known: $\delta(xy,i) \leqslant \Delta$.

$$\begin{tikzcd}
\bot \arrow[dr, "\delta(ab,0)"'] & & a_0 \arrow[rr, "\delta(aa,1)"] \arrow[dr, "\delta(ab,1)"'] & & a_1 \arrow[rr, "\delta(aa,2)"] \arrow[dr, "\delta(ab,2)"'] & & a_2 & \\
& b_0 \arrow[ur, "\delta(ba,0)"] \arrow[rr, "\delta(bb,1)"'] & & b_1 \arrow[ur, "\delta(ba,1)"] \arrow[rr, "\delta(bb,2)"'] & & b_2 \arrow[ur, "\delta(ba,2)"] \arrow[r] & \cdots &
\end{tikzcd}$$

Figure Delays in E.

We only have to fill in the upper bounds Δ instead of the real delays, in order to get an upper bound E_Δ, $E_\Delta \geqslant E_\delta$, for any δ:

$$\begin{array}{lcllll}
E_\Delta.s.a_i & = & \textbf{if} & i=0 & \rightarrow & s.b_i+\Delta \\
 & & [\!] & i>0 & \rightarrow & s.b_i+\Delta \max s.a_{i-1}+\Delta \\
 & & \textbf{fi} & & & \\
E_\Delta.s.b_i & = & \textbf{if} & i=0 & \rightarrow & \Delta \\
 & & [\!] & i>0 & \rightarrow & s.a_{i-1}+\Delta \max s.b_{i-1}+\Delta \\
 & & \textbf{fi} & & &
\end{array}$$

Consider furthermore the specification $e = \mathrm{n}(a^*)$ for the external behaviour of the device. Since $E_\Delta \approx 2\Delta \odot e$, we conclude $\mathrm{R}(E_\delta, e) \leqslant 2\Delta$ for all δ; which is an upper bound for the external behaviour of the device that is proportional to the upper bound of the individual delays (Δ). In order to draw this conclusion we did not need to analyze the general case E_δ, we needed to observe the upper bound E_Δ only.

The upper bound we gave for the external behaviour still holds when we assume a coupling of delays, for example: $\delta(ab,i)+\delta(ba,i) \leqslant \Delta$. In this case a tight upper bound of the external behaviour is given by $\mathrm{R}(E_\delta, e) \leqslant 1\frac{1}{2}\Delta$. In order to observe that this is an upper bound, it suffices to observe

$E_\delta \parallel 1\frac{1}{2}\Delta \odot e \;\approx\; 1\frac{1}{2}\Delta \odot e$, which is left as an exercise to the reader.

If Δ is a lower bound of the individual delays, instead of an upper bound, we conclude $\mathrm{R}(E_\delta, e) \geqslant 2\Delta$ (for all δ).

□

3.7 Conclusion

We have succeeded in finding a relation, $\lhd\!\mid$, that can be used to compare the speed of mechanisms. The statement '$e \lhd\!\mid f$' can be interpreted as 'e implements f', or 'e is as least as fast as f'. The comparison is one of *global* behaviour, rather than a comparison of delays between individual actions. An important feature of relation $\lhd\!\mid$ is that it can be used in compositional design: $\lhd\!\mid$ is transitive, and parallel composition and restriction are monotonic with respect to $\lhd\!\mid$. The price that is paid to achieve this result, is that $\lhd\!\mid$ is reflexive on the class of conservative enabling functions only. In Chapters 4 and 5 we illustrate the usage of $\lhd\!\mid$ in the design of linear systolic arrays, and of distributed FIFO buffers.

In this section we explain why relation $\lhd\!\mid$ is not suitable for specifications of behaviours that capture 'choice'. Furthermore we show that $\lhd\!\mid$ *can* be used when the delays between individual actions are a correctness concern.

Choice

The approach we have chosen to compare behaviours is unfit for specifications (of external behaviour) in which choice is essential. Such specifications do not fit in a philosophy in which *faster* is *better*, simply because a faster environment may mess up the choice.

Example 3.59 Adaptive ordering revisited.

Consider for example the adaptive ordering, specified by enabling function e of Example 2.73. Observe that e is not conservative, consequently $\neg(e \lhd\!\mid e)$. This is due to the fact that the environment does not always enforce the 'proper' choice. Take for example environment $f = \mathbf{n}(a\ ;\ b)^*$. In cooperation with this environment the behaviour is given by $e \parallel f = f$. But when the environment is implemented by $g = \mathbf{n}(a, b\ ;\ (a\ ;\ b)^*)$ (verify that $g \lhd\!\mid f$), the cooperation performs $e \parallel g$ which is equivalent to:

```
$h.s.a_i$ = if $i = 0$ → 1
            [] $i > 0$ → if $s.a_0 < s.b_0$ → $s.b_{i-1} + 1$
                         [] $s.a_0 \geqslant s.b_0$ → $\infty$
                         fi
            fi
$h.s.b_i$ = if $i = 0$ → 1
            [] $i > 0$ → if $s.a_0 < s.b_0$ → $s.a_i + 1$
                         [] $s.a_0 \geqslant s.b_0$ → $\infty$
                         fi
            fi
```

(description under liberal delay conditions)

Consequently $e \parallel g$ cannot be considered an implementation of $e \parallel f$. Because $g \lhd\!\!\shortmid f$, e cannot be used to implement e. In fact, any implementation of e must, for each i, allow a_i and b_i in any order. For example, $\mathrm{n}\,(a\,,b)^*$ is an implementation of e.

For similar reasons it is (also) not a good idea to use the adaptive ordering, e, as an implementation of $(a\ ;\ b)^*$ or $(b\ ;\ a)^*$: it implements neither of these behaviours.

□

This does not mean that problems in which 'choice' is essential are taboo. Take for example the buffer with bypassing that is discussed in Section 5.5. By using other criteria (than $\lhd\!\!\shortmid$) for being 'an implementation of' on a local level, we are able to give estimations of the quality of such buffers. These local criteria capture, for example, the precision at which a (real world) device is able to perform a choice.

Another problem with choice is *meta-stability*: if a device is not sure what alternative to choose, it may take arbitrarily long before the decision is made. This is a phenomenon that cannot be described within the enabling model in a satisfactory way.

Delays between individual actions

The way in which we compare behaviours does *not* consider the delays between individual actions. It may well be that, in some environment, an implementation exhibits larger delays than its specification. If this is the case, however, it is always due to the fact that the implementation has 'taken an advance' on the specification. An example of this phenomenon is already given in Example 3.43, where enabling function g implements h, in spite of the fact that h exhibits delays of one time unit only, while g also exhibits delays of two time units. However, a simple interference (addition of a controller) can prevent an implementation from taking liberties.

Example 3.60 A number generator.

In this example we assume that communications take one time unit. We consider a number generator that is to produce results of computations that are increasingly complex, along channel a. The real-time constraint is that is produces at least one value every two time units.

$$
\begin{array}{lcl}
f.s.a_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ 1 \\
 & & [\!]\ \ i > 0 \ \rightarrow\ s.a_{i-1} + 2 \\
 & & \textbf{fi}
\end{array}
$$

A parallel implementation that takes $i * \kappa_i$ time units to compute the value that is send during communication a_i is described by:

$$e.s.a_i \;=\; \begin{array}{ll} \textbf{if} & i=0 \rightarrow 1 \\ [\!] & i>0 \rightarrow s.a_{i-1}+1 \text{ max } i*\kappa_i+1 \\ \textbf{fi} & \end{array}$$

The reader is invited to verify that e $\lhd_1$ f if (and only if) for all i $\kappa_i \leqslant 2$. But, does this implementation indeed produce one value, every two time units? Assume some clever programmer finds a a way to output the first N values at double rate:

$$e.s.a_i \;=\; \begin{array}{lll} \textbf{if} & i=0 & \rightarrow 1 \\ [\!] & 0<i<N & \rightarrow s.a_{i-1}+1 \text{ max } i*\kappa/2+1 \\ [\!] & N \leqslant i & \rightarrow s.a_{i-1}+1 \text{ max } i*\kappa+1 \\ \textbf{fi} & & \end{array}$$

for some κ, $1<\kappa\leqslant 2$.

In a greedy environment the outputs a_i for i up to $N-1$ are produced at a rate of 1 per time unit, at moments $i+1$. For $i \geqslant N$ the outputs are produced at a rate of 1 per κ time units, at moments $i*\kappa+1$. In particular, output a_{N-1} is produced at moment N and output a_N at moment $N*\kappa+1$. The delay between these outputs is $N*(\kappa-1)+1$. This delay may exceed 2. In fact, the delay is proportional to N. It seems that the programmer should better not be too clever.

□

Considering this example, one may argue that $\lhd_1$ has to do with delays since initiation, rather than with delays between individual actions. From this point of view ' e $\lhd_1$ f ' means that the *amortized* time complexity of e is at most that of f, or that e is a *linear time* implementation of f (terminology of [25] and [6] respectively).

However, when it is really important to control the delays between successive actions, there is a very simple solution. Just add a controller (timer) that behaves as f. Because e $\lhd_1$ f implies e $\|$ $f \approx f$, the combination e $\|$ f behaves exactly as specified. In the example, this controller guarantees that the environment is offered exactly one communication in every two time units, no more, no less.

Chapter 4

Divide And Conquer

Systolic arrays[1] are, usually large, regular composites of, usually small, building blocks, also called *cells*. There are two ways to compute, or to estimate, the external behaviour of such 'compound' structures. Consider for example $F = (E_0 \parallel E_1 \parallel E_2) \mathbin{\|\!\upharpoonright} A$.

One method is to compute, or estimate, the behaviour of the *entire* structure F, and to use this behaviour in order to compute, or estimate, the external behaviour $F \upharpoonright A$. This method is not recommended for large structures, since this involves manipulation of complicated enabling structures with a large alphabet.

The other method consists of using a divide and conquer scheme. For example for F as above:

- compute enabling functions e_i that satisfy: $\mathrm{e}_0 \approx E_0 \mathbin{\|\!\upharpoonright} (\mathbf{e}E_1 \cup \mathbf{e}E_2 \cup A)$ etc.;
- compute $f \approx (\mathrm{e}_0 \parallel \mathrm{e}_1) \mathbin{\|\!\upharpoonright} (\mathbf{a}\mathrm{e}_2 \cup A)$;
- compute $g \approx (f \parallel \mathrm{e}_2) \mathbin{\|\!\upharpoonright} A$;

and conclude (from the theory of Section 2.7) that $g \approx F$.

When an estimation of the external behaviour is required only, the following scheme suffices:

- determine enabling functions e_i that satisfy: $E_0 \mathbin{\|\!\upharpoonright} (\mathbf{e}E_1 \cup \mathbf{e}E_2 \cup A) \mathrel{\lhd\!|} \mathrm{e}_0$ etc.;
- determine f such that $(\mathrm{e}_0 \parallel \mathrm{e}_1) \mathbin{\|\!\upharpoonright} (\mathbf{a}\mathrm{e}_2 \cup A) \mathrel{\lhd\!|} f$;
- determine g such that $(f \parallel \mathrm{e}_2) \mathbin{\|\!\upharpoonright} A \mathrel{\lhd\!|} g$;

and conclude (from the theory of Section 3.3) that $F \mathrel{\lhd\!|} g$. A similar scheme can be used for an estimation of the type $g \mathrel{\lhd\!|} F$.

[1] In contrast to some authors, we use the term 'systolic array' for implementations in which the cells are *not* synchronized by a global clock.

In Section 4.1 we introduce linear systolic arrays, in Section 4.2 we discuss how to compute, or estimate, their external behaviour by means of divide and conquer. In Section 4.3 we discuss systolic implementations of a 'segment problem': determining the maximum of successive input values. This problem is both simple enough to go through the data-part relatively easily, and interesting enough to illustrate the effect of implementation decisions on the performance.

4.1 Linear systolic arrays

In this chapter we address linear systolic arrays that consist of identical cells, possibly with a deviating tail. We consider arrays with external communication at one side only.

An infinite (direct recursive) array consists of a regular, one sided infinite, linear arrangement of identical cells: that is, identical modulo a renaming p. The general form of such an infinite array is given by:

(4.1) $F \;=\; (\,\|\; n : 0 \leqslant n : p^n.E\,) \shortparallel A$

In infinite arrays, cells may only communicate with cells in a bounded neighbourhood; consequently, sharing of actions between all cells (broadcast) is not allowed.

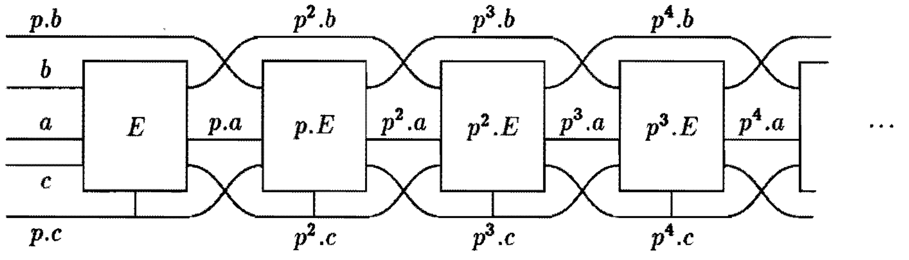


Figure 4.2 Schematic representation of an infinite array F.

In illustrations, such as Figure 4.2, shared actions are denoted by connections that are labeled with (generic) actions. The loose ends at the left-hand side are the external actions.

The general form of a finite array is given by:

(4.3) $F_N \;=\; ((\,\|\; n : 0 \leqslant n < N : p^n.E\,) \;\|\; p^N.F_0) \shortparallel A \qquad \text{for } N \geqslant 0$

where F_0 is the *tail*. In finite arrays we do allow broadcast; for example channel c in Figure 4.4. Arrays with broadcast are also called *semi-systolic*.

The following restrictions on renaming p, and on the naming of actions in the cells, are carefully chosen in order to avoid 'name clashes'. For other types of systolic arrays, such as binary trees and linear arrays with external communication at both sides, other choices have to be made.

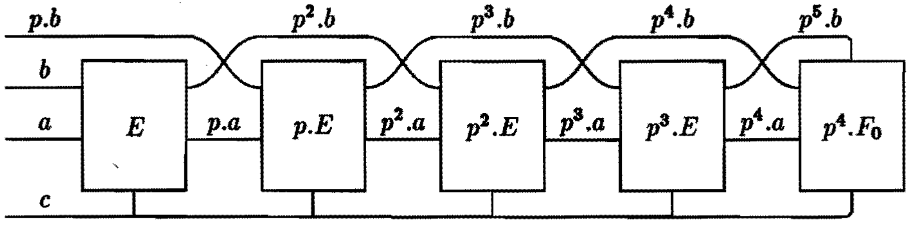


Figure 4.4 Schematic representation of a finite array F_4.

- We distinguish local actions ($p.a \neq a$), and broadcasts ($p.a = a$). Let alphabet B be the 'base' of the local actions:
  - for $a, b \in B$: $p^n.a = p^m.b \;\Leftrightarrow\; n = m \wedge a = b$
  - for $a \notin (\cup n : 0 \leqslant n : p^n.B)$: $p.a = a$.

  When using a renaming p in the description of a systolic array, the default is 'no broadcast', $p.a \neq a$: that is, for any action a the equality $p.a = a$ holds only when it is explicitly mentioned.
- E must satisfy:
  - $\mathbf{i}E \subseteq B$
  - for $a \in B$: $a \in \mathbf{e}E \;\Leftrightarrow\; (\exists n : n > 0 : p^n.a \in \mathbf{e}E)$
  - in case of an infinite array: $(\, \mathbf{lub}\, a, n : p^n.a \in \mathbf{e}E : n \,) \;<\; \infty$

    The left-hand side of this formula is called the *communication distance* of E (this condition implies that broadcast is not allowed in infinite arrays).
- In case of a finite array, F_0 must satisfy
  - $\mathbf{i}F_0 \;\subseteq\; (\cup n : 0 \leqslant n : p^n.B)$
  - $\mathbf{e}F_0 = A$ (same external alphabet as F_N)
- All loose ends at the left-hand side are communicated with the environment:

  $A \;=\; (\,\mathbf{set}\, a, m, n : m < n \wedge p^n.a \in \mathbf{e}E : p^m.a\,)$

In order to save parentheses in expressions like $s.(p.a)_i$ and $e.s.(p.a)_i$, these are abbreviated to $s.pa_i$ and $e.s.pa_i$ respectively (function application without a dot).

Observe that $F_{n+1} = (E \parallel p.F_n) \shortparallel A$, and that $F = (E \parallel p.F) \shortparallel A$.

4.2 Analyzing the behaviour of linear systolic arrays

We define function Φ as the *generating function* for the external behaviour of systolic arrays. Given a mechanism with external behaviour e, the result of placing cell E in front of it, is a mechanism with external behaviour $\Phi.e$. For example, for E as in Figure 4.4, the generating function describes the external behaviour of a mechanism as given in Figure 4.5.

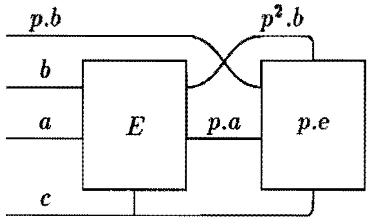


Figure 4.5 The generating function: $\Phi.e \;=\; (E \parallel p.e) \Vert\!\upharpoonright A$.

Definition 4.6

For E, p, and the corresponding alphabet A (according to the conditions in Section 4.1), function $\Phi_{E,p} : \mathcal{EF}.A \rightarrow \mathcal{EF}.A$ is defined by:

$\Phi_{E,p}.e \;=\; (E \parallel p.e) \Vert\!\upharpoonright A$.

□

The subscripts of Φ are only used when they deviate from E and p respectively. Furthermore F and F_N are by default arrays that satisfy Formulae 4.1 and 4.3 respectively.

We mention some elementary divide and conquer properties for finite arrays that follow from the theory of Chapter 2 and Chapter 3.

- For $e_n : n \geqslant 0$ a sequence of enabling functions over A:

  if $e_0 = F_0 \upharpoonright A$ and for all n $e_{n+1} = \Phi.e_n$, then for all n: $e_n = F_n \upharpoonright A$;

  if $e_0 \approx F_0 \upharpoonright A$ and for all n $e_{n+1} \approx \Phi.e_n$, then for all n: $e_n \approx F_n \upharpoonright A$.

- If $E \upharpoonright eE \in [\,Con\,]$, we have also:

  if $F_0 \upharpoonright A \lhd\!\shortmid e_0$ and for all n $\Phi.e_n \lhd\!\shortmid e_{n+1}$, then for all n: $F_n \upharpoonright A \lhd\!\shortmid e_n$;

  if $e_0 \lhd\!\shortmid F_0 \upharpoonright A$ and for all n $e_{n+1} \lhd\!\shortmid \Phi.e_n$, then for all n: $e_n \lhd\!\shortmid F_n \upharpoonright A$.

- $F_{n+1} \approx F_n \;\Rightarrow\; (\forall m : m > n : F_m \approx F_n)$

- If $E \upharpoonright eE \in [\,Con\,]$, we have also:

  $F_{n+1} \lhd\!\shortmid F_n \;\Rightarrow\; (\forall m : m > n : F_m \lhd\!\shortmid F_n)$
  $F_n \lhd\!\shortmid F_{n+1} \;\Rightarrow\; (\forall m : m > n : F_n \lhd\!\shortmid F_m)$

The following theorem supports similar properties for infinite arrays.

Theorem 4.7

If E has a finite communication distance k, then

1 Φ^k is a contraction (see Appendix C)

2 $\mathbf{d}(\Phi^k.\mathrm{e}) \geqslant \mathbf{d}E$

Proof (Let F be as in Formula 4.1.)

F can be rewritten into terms of an array with communication distance 1 by clustering k cells into one:

$F \;=\; (\,\|\, n : 0 \leqslant n : q^n.E_k\,)$

where $q = p^k$ and $E_k \;=\; (\,\|\, j : 0 \leqslant j < k : p^j.E\,) \shortparallel (A \cup q.A)$.

E_k and q satisfy the conditions for E and p in Section 4.1, and E_k has communication distance 1. Furthermore, $\Phi^k_{E,p} \;=\; \Phi_{E_k,q}$ and $\mathbf{d}E_k \geqslant \mathbf{d}E$. So it suffices to prove the theorem for $k = 1$.

1 We prove that Φ is a contraction by proving that

$\mathrm{sim}(\Phi.\mathrm{e}, \Phi.f) \;\geqslant\; \mathbf{d}E + \mathrm{sim}(\mathrm{e}, f)$

for any e and f over A. (see Theorem C.4)

Let e and f as above and let $s \in S.A$.

First observe that:

$\mathrm{sim}(s \uparrow (E \,\|\, p.\mathrm{e}),\; s \uparrow (E \,\|\, p.f)) \;\geqslant\; \mathrm{sim}(\mathrm{e}, f)$

Second observe that for t over $\mathbf{a}(E \,\|\, p.\mathrm{e})$:

$$\begin{aligned} & (E \,\|\, p.\mathrm{e}).t \restriction A \\ = & \quad \{\, A \cap p.A = \varnothing \,\} \\ & E.(t \restriction \mathbf{a}E) \restriction A \end{aligned}$$

Conclude from Proposition 2.62 that $\mathrm{sim}(\Phi.e.s,\; \Phi.f.s) \geqslant \mathrm{sim}(e, f) + \mathbf{d}E$.

2 For s and s' over A holds

$\mathrm{sim}(s \uparrow (E \,\|\, p.\mathrm{e}), s' \uparrow (E \,\|\, p.\mathrm{e})) \;\geqslant\; \mathrm{sim}(s, s')$

Conclude, similar to the previous item, $\mathrm{sim}(\Phi.e.s, \Phi.e.s') \geqslant \mathrm{sim}(\mathrm{e}, \mathrm{e}') + \mathbf{d}E$.

□

The following are some consequences of this theorem and of the contraction theorem, Theorem C.1, and Proposition C.3.3. The last observation also uses the continuity of $\lhd\shortmid$, as given in Theorem C.5.

If E has a finite communication distance:

- $\lim_{n \to \infty} F_n \restriction A \;=\; F \restriction A$

  (independent of the tail F_0 that is used in the finite arrays)

- $F \restriction A$ is the unique solution of e in the equation $e = \Phi.e$, and
  $e \approx \Phi.e \;\Rightarrow\; e \approx F \restriction A$.
- For $E \restriction \mathrm{e}E \in [\,Con\,]$:
  $\Phi.e \lhd\!\mid e \;\Rightarrow\; F \restriction A \lhd\!\mid e \qquad e \lhd\!\mid \Phi.e \;\Rightarrow\; e \lhd\!\mid F \restriction A$.

The second property states that the behaviour of an infinite systolic array is the *unique* fixed point of the generating function. In [13, 27] the behaviour is the *least* fixed point of a generating function. This is due to the fact that the trace formalism does not give information about timing.

Example 4.8 Unique fixed point versus least fixed point.

Consider cell E, with alphabet $\{a, p.a\}$ (both external) that performs $(a; p.a; p.a; a)$:

```
E.s.a_i  =  if  $i = 0 \rightarrow 1$
            []  $i = 1 \rightarrow s.pa_1 + 1$
            []  $i > 1 \rightarrow \infty$
            fi
E.s.pa_i =  if  $i = 0 \rightarrow s.a_0 + 1$
            []  $i = 1 \rightarrow s.pa_0 + 1$
            []  $i > 1 \rightarrow \infty$
            fi
```

Enabling function $\mathbf{n}\,a$ is a fixed point (and thus the unique fixed point) of equation $\Phi.e = e$.

When the timing information is ignored (as in [13, 27]), the process that can perform two a actions is also a fixed point of the generating function. In terms of enabling functions this can be understood as follows.

Let $\mathrm{e} = \mathbf{n}\,a^2$. $\Phi.e$ is given by:

```
Φ.e.s.a_i  =  if  $i = 0 \rightarrow 1$
              []  $i = 1 \rightarrow s.a_{i-1} + 3$
              []  $i > 1 \rightarrow \infty$
              fi
```

Apart from the timing ($+3$ instead of $+1$) this is the same behaviour as e .

□

4.3 The maximum of segments

In this section we exhibit several strategies in designing systolic arrays by means of a rather simple specification. Apart from the behavioural aspect, it also provides an example of 'dividing' a data-specification in order to obtain a parallel computation. More elaborate examples of this technique can for example be found in [14, 23, 27]. In [14], a systolic array is derived that communicates to the environment at both sides. We briefly discuss the behaviour of such arrays in Section 5.6.

For $N : 0 \leqslant N$, we specify program Max_N as follows:

data:

Max_N has input channel a and output channel b, both of type integer. The values to be output along b are given by:

$b(i) \;=\; (\,\mathbf{max}\, j : 0 \leqslant j \wedge i - N \leqslant j \leqslant i : a(j)\,)$

where $b(i)$ is the value that is output during communication b_i, and $a(j)$ is the value that is input during communication a_j.

(real-time) behaviour:

We are interested in the response time of the solution relative to (the enabling function of) $(a\ ;\ b)^*$. In particular, this response time should be bounded by an upper bound independent of N.

The first solution that we give for this problem is not satisfactory because it is slow, proportional to the size of the problem, N. In the second solution the speed is improved at the cost of one additional variable per cell. As the first solution, however, the second solution is only semi-systolic, and a disadvantage of broadcasts is that they tend to be slow, at least proportional to the logarithm of the length of the array. In the third solution the problem is solved without usage of a broadcast. As a variation on the third solution, we discuss the sharing of hardware between adjacent cells.

First solution

First we consider the simple case: $N = 0$.

For $N = 0$ the data part of the specification boils down to $b(i) = a(i)$, so this case can be implemented by the following program that describes a one-place buffer with input channel a and output channel b:

```
program Max_0 ( input a : integer , output b : integer ) :
var va : integer ;
begin
   ( a?va ; b!va )*
end.
```

where $a?va$ denotes receipt of an input value in variable va, and $b!va$ denotes output of the value of va.

We use the following conventions when giving enabling structures to describe (real-time) behaviour:

- A mechanism is described relative to the moment of initiation.

- The scheduling of actions within the model reflects the completion of these actions by the described mechanism.

We assume that a communication takes ρ time, so the behaviour of Max_0 is given by enabling function MA_0:

(4.9) $$\begin{aligned} MA_0.s.a_i &= \textbf{if}\ \ i=0 \ \rightarrow\ \rho \ \ [] \ \ i>0 \ \rightarrow\ s.b_{i-1}+\rho \ \ \textbf{fi} \\ MA_0.s.b_i &= s.a_i+\rho \end{aligned}$$

Next we consider the general case: $N > 0$.

We derive:

for $i = 0$: $b(i) = a(i)$,

for $i > 0$:

$$\begin{aligned} & b(i) \\ = & \quad \{\text{domain split}\} \\ & (\,\mathbf{max}\, j : 0 \leqslant j \wedge i-N \leqslant j < i : a(j)\,)\ \max\ a(i) \\ = & \\ & (\,\mathbf{max}\, j : 0 \leqslant j \wedge (i-1)-(N-1) \leqslant j \leqslant (i-1) : a(j)\,)\ \max\ a(i) \end{aligned}$$

In the quantified expression we recognize the output $b(i-1)$ of Max_{N-1}. So, when using $p.Max_{N-1}$ with $p.a = a$ we get the following:

$$b(i) \ = \ \textbf{if}\ \ i=0 \ \rightarrow\ a(i) \ \ [] \ \ i>0 \ \rightarrow\ p.b(i-1)\ \max\ a(i) \ \ \textbf{fi}$$

We infer the following program for $N > 0$:

```
program $Max_N$ ( input $a$: integer , output $b$: integer ) :
uses $p.Max_{N-1}$ with $p.a = a$ ;
var $va$, $vb$: integer ;
begin
   $a?va$ ; $b!va$ ; $(\,a?va\,, p.b?vb\ ;\ b!(va \max vb)\,)^*$
end.
```

The program text between **begin** and **end** is executed in the head cell. The hierarchical structure of this program (**uses** $p.Max_{N-1}$) is similar to the use of 'sub components' in the 'com moc' programs in [13, 24, 27].

If we assume that the computation of the maximum of two integers takes κ time, the behaviour of the head cell of Max_N ($N > 0$) is given by enabling function M:

(4.10) $$\begin{aligned} M.s.a_i &= \textbf{if}\ \ i=0 \ \rightarrow\ \rho \ \ [] \ \ i>0 \ \rightarrow\ s.b_{i-1}+\rho \ \ \textbf{fi} \\ M.s.b_i &= \textbf{if}\ \ i=0 \ \rightarrow\ s.a_i+\rho \\ &\quad\ [] \ \ i>0 \ \rightarrow\ s.a_i+\kappa+\rho\ \max\ s.pb_{i-1}+\kappa+\rho \\ &\quad\ \textbf{fi} \\ M.s.pb_i &= s.b_i+\rho \end{aligned}$$

The behaviour of Max_N is given by array MA_N:

$$MA_N \ = \ (M \parallel p.MA_{N-1}) \shortparallel \{a,b\} \qquad \text{for } N>0\text{, with } p.a = a\,.$$

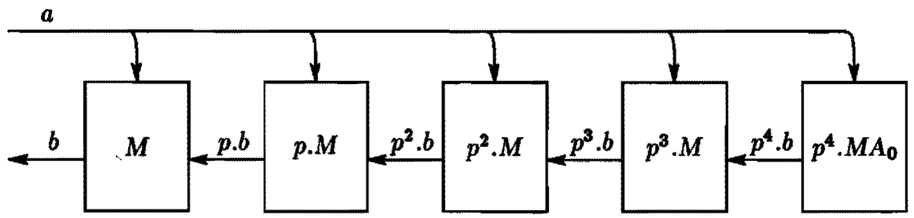


Figure 4.11 Schematic representation of array MA_4.

In order to give an impression of what happens, we illustrate the parallel composition of M and $p.MA_0$ by means of fragments of their dependence relations:

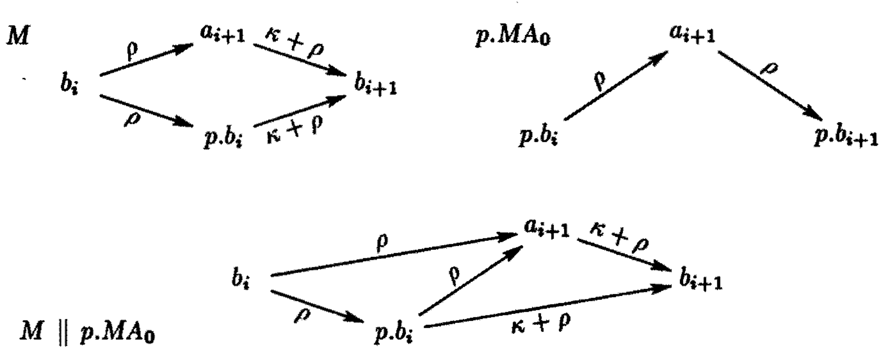


Figure 4.12 Dependencies in the composition of M and $p.MA_0$.

In $M \parallel p.MA_0$, action a_{i+1} cannot be performed until 2ρ after b_i is performed. In general, the problem is that a_{i+1} must wait for $p^n.b_i$ for all $n : 0 \leqslant n \leqslant N$, and that these b actions are ordered. This results in the following external behaviour $f_N \approx MA_N$ of the systolic arrays:

$$
\begin{array}{llllll}
(4.13) & f_N.s.a_i & = & \textbf{if} & i = 0 & \rightarrow \rho \\
 & & & [] & i > 0 & \rightarrow s.b_{i-1} + (N+1)\rho \\
 & & & \textbf{fi} & & \\
 & f_N.s.b_i & = & \textbf{if} & i = 0 \vee N = 0 & \rightarrow s.a_i + \rho \\
 & & & [] & i > 0 \wedge N > 0 & \rightarrow s.a_i + \kappa + \rho \\
 & & & \textbf{fi} & &
\end{array}
$$

A derivation of this external behaviour is given below.

For $N > 0$ we conclude that the response time of MA_N with respect to $(a \ ; \ b)^*$ equals $(N+1)\rho \max \kappa + \rho$. So MA_N is slow proportional to N. When the values ρ and κ are lower bounds for the duration of communications and computations respectively, this negative result is still true (see Section 3.6).

Derivation of the external behaviour as given in Formula 4.13.

Observe that $f_0 = MA_0$; remains to prove $(M \parallel p.f_n) \Vert\!\upharpoonright \{a, b\} \approx f_{n+1}$.

Let $e_0 = M \parallel p.f_n$:

$$\begin{array}{lcl}
e_0.s.a_i & = & \textbf{if } i = 0 \rightarrow \rho \\
 & & [\!]\ \ i > 0 \rightarrow s.b_{i-1} + \rho \text{ max } s.pb_{i-1} + (n+1)\rho \\
 & & \textbf{fi} \\
e_0.s.b_i & = & \textbf{if } i = 0 \rightarrow s.a_i + \rho \\
 & & [\!]\ \ i > 0 \rightarrow s.a_i + \kappa + \rho \text{ max } s.pb_{i-1} + \kappa + \rho \\
 & & \textbf{fi} \\
e_0.s.pb_i & = & \textbf{if } i = 0 \lor n = 0 \rightarrow s.b_i + \rho\ \underline{\text{max } s.a_i + \rho} \\
 & & [\!]\ \ i > 0 \land n > 0 \rightarrow s.b_i + \rho\ \underline{\text{max } s.a_i + \kappa + \rho} \\
 & & \textbf{fi}
\end{array}$$

First we consider the underlined dependencies. Observe that the formula for $e_0.s.b_i$ implies that for $s \in \mathbf{P}e_0$:

$$\begin{array}{lcl}
s.b_i & \geqslant & \textbf{if } i = 0 \rightarrow s.a_i + \rho \\
 & & [\!]\ \ i > 0 \rightarrow s.a_i + \kappa + \rho \\
 & & \textbf{fi}
\end{array}$$

This implies that the terms $\max s.a_i + \rho$ and $\max s.a_i + \kappa + \rho$ in the enabling of $p.b_i$ are redundant. In the sequel we make a habit out of underlining redundant dependencies that are to be pruned. Pruning the underlined dependencies in e_0 yields enabling function e_1 (which is equivalent to e_0 : $e_0 \approx e_1$):

$$e_1.s.pb_i \ = \ s.b_i + \rho \quad \text{and} \quad e_1.s.x \ = \ e_0.s.x \quad \text{for } x \neq pb_i \ .$$

Next $e_2 = e_1 \Vert\!\upharpoonright \{a, b\}$ is computed by assuming that pb_i happens as soon as it is enabled: $s.pb_i = s.b_i + \rho$:

$$\begin{array}{lcl}
e_2.s.a_i & = & \textbf{if } i = 0 \rightarrow \rho \\
 & & [\!]\ \ i > 0 \rightarrow s.b_{i-1} + \rho \text{ max } s.b_{i-1} + (n+2)\rho \\
 & & \textbf{fi} \\
 & = & \textbf{if } i = 0 \rightarrow \rho \\
 & & [\!]\ \ i > 0 \rightarrow s.b_{i-1} + (n+2)\rho \\
 & & \textbf{fi} \\
e_2.s.b_i & = & \textbf{if } i = 0 \rightarrow s.a_i + \rho \\
 & & [\!]\ \ i > 0 \rightarrow s.a_i + \kappa + \rho\ \underline{\text{max } s.b_{i-1} + \kappa + 2\rho} \\
 & & \textbf{fi}
\end{array}$$

The underlined dependency is redundant; pruning yields f_{n+1} .

□

The problem that a_{i+1} must wait for $p^n.b_i$ for all $n : 0 \leqslant n \leqslant N$ and that these b actions are ordered, can be solved in two entirely different ways: one is to remove the ordering between the b actions (second solution), the other is to avoid that a_{i+1} must wait for all those b actions (third solution).

Second Solution

In the first solution, action $p.b_{i+1}$ has to wait for b_i because the value of variable vb may not be changed until the result of the computation va max vb has been output along b. The introduction of additional buffering can solve this dependency; in the program $Max2_N$ this is done by means of an internal channel c.

For $N = 0$ we use the same program as in the first solution, $Max2_0 = Max_0$. For $N > 0$ we have:

```
program Max2_N ( input a : integer , output b : integer ) :
uses p.Max2_{N-1} with p.a = a ;
var va, vb, vc : integer ;
begin
    a?va ; b!va ; ( a?va , c?vc ; b!(va max vc) )*
||
    ( p.b?vb ; c!vb )*
end.
```

The internal communication along c is an alternative for the statement $vc := vb$. Furthermore, the first part of the program is the program in the first solution, modulo a renaming of $p.b$ in c; the second part describes a one-place buffer.

The behaviour of the head cell of $Max2_N$ is given by enabling structure $M2$ with external alphabet $\{a, b, p.b\}$ and internal alphabet c:

$$\begin{array}{llll}
(4.14) & M2.s.a_i & = & \textbf{if } i = 0 \rightarrow \rho \\
 & & & [\!]\ i > 0 \rightarrow s.b_{i-1} + \rho \\
 & & & \textbf{fi} \\
 & M2.s.b_i & = & \textbf{if } i = 0 \rightarrow s.a_i + \rho \\
 & & & [\!]\ i > 0 \rightarrow s.a_i + \kappa + \rho \text{ max } s.c_{i-1} + \kappa + \rho \\
 & & & \textbf{fi} \\
 & M2.s.pb_i & = & \textbf{if } i = 0 \rightarrow \rho \\
 & & & [\!]\ i > 0 \rightarrow s.c_{i-1} + \rho \\
 & & & \textbf{fi} \\
 & M2.s.c_i & = & s.pb_i + \rho \text{ max } s.b_i + \rho
\end{array}$$

The array $MA2_N$ has the same shape as MA_N, but with cells of type $M2$ instead of M.

$$\begin{tikzcd}
a_i \arrow[rd, "\kappa+\rho"] & & a_{i+1} \\
 & b_i \arrow[ru, "\rho"] \arrow[rd, "\rho"] & \\
c_{i-1} \arrow[ru, "\kappa+\rho"] \arrow[rd, "\rho"'] & & c_i \\
 & p.b_i \arrow[ru, "\rho"'] &
\end{tikzcd}$$

Figure 4.15 Dependencies in cell $M2$.

The reader is invited to verify that $MA2_N$ has the following external behaviour:

(4.16)
$$
\begin{array}{rcl}
f_N.s.a_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ \rho \ \ [] \ \ i > 0 \ \rightarrow\ s.b_{i-1} + \rho \ \ \textbf{fi} \\
f_N.s.b_i & = & \textbf{if}\ \ i = 0 \vee N = 0 \ \rightarrow\ s.a_i + \rho \\
 & & []\ \ i > 0 \wedge N > 0 \ \rightarrow\ s.a_i + \kappa + \rho \\
 & & \textbf{fi}
\end{array}
$$

For $N > 0$ we conclude a response time of $\kappa + \rho$ with respect to $(a\ ;\ b)^*$; when κ and ρ are upper bounds, $\kappa + \rho$ is an upper bound for the response time (see Section 3.6). This response time is achieved at the cost of one additional variable per cell (with respect to the first solution).

Second solution revisited

A problem with the first two solutions is the broadcast along a : a broadcast to a lot of receivers tends to be slow, at least proportional to the logarithm of the number of receivers.

Up to now, the solutions are described under the assumption of a universal duration of communications (ρ). Therefore the scheduling of communications was completely described by the moment of their completion. When the duration of communications is not universal, one should be careful with using completion as a criterion.

Let for example e describe a mechanism that needs ρ_1 time to communicate b , and let f describe another mechanism that needs ρ_2 :

$e.s.b_i = s.a_i + \kappa + \rho_1 \qquad f.s.b_i = s.a_i + \rho_2$.

Parallel composition gives: $(e \parallel f).s.b_i = s.a_i + (\kappa + \rho_1 \ \max\ \rho_2)$,

though one would expect that the composition of both mechanisms needs $\rho_1 \ \max\ \rho_2$ time units to communicate, and thus behaves according to:

$g.s.b_i = s.a_i + \kappa + (\rho_1 \ \max\ \rho_2)$.

In order to give a concise description, one has to describe communications with two actions each: an action of initiation and an action of completion. Otherwise, the results of parallel composition of mechanisms with distinct durations of communications may be too 'optimistic'.

With this warning in mind, we give the external behaviour of $MA2_N$ under the assumption that a broadcast to n receivers takes ρ_n time:

(4.17)
$$
\begin{array}{rcl}
f_N.s.a_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ \rho_{N+1} \\
 & & []\ \ i > 0 \ \rightarrow\ s.b_{i-1} + \rho_{N+1} \\
 & & \textbf{fi} \\
f_N.s.b_i & = & \textbf{if}\ \ i = 0 \vee N = 0 \ \rightarrow\ s.a_i + \rho \\
 & & []\ \ i > 0 \wedge N > 0 \ \rightarrow\ s.a_i + \kappa + \rho \\
 & & \textbf{fi}
\end{array}
$$

The response time of f_N relative to $(a\ ;\ b)^*$ is $\rho_{N+1} \ \max\ \kappa + \rho$, which is at least proportional to the logarithm of the number of cells. Therefore we prefer an approach in which *no* broadcast is used.

Third solution

Both previous solutions are semi-systolic arrays: that is, arrays with a broadcast. In both solutions a_{i+1} has to wait for all actions $p^n.b_i$ because all cells perform a_{i+1} simultaneously. In this solution we consider a program *without* broadcast along channel a. As a result, a_{i+1} has to wait for $p.a_i$ and b_i only.

At the point in the derivation of the first solution where we recognized output $b(i-1)$ of Max_{N-1}, we may also use $p.Max_{N-1}$ with $p.a \neq a$, provided we assure $p.a(i) = a(i)$ for all i. This leads to the same obligation for $b(i)$, and an additional obligation for $p.a(i)$. For $N = 0$ we use the same program as in the first solution: $Max3_0 = Max_0$; for $N > 0$ we use the following program:

```
program Max3_N ( input a: integer , output b: integer ) :
uses p.Max3_{N-1} ;
var va, vb: integer ;
begin
   ( a?va ; p.a!va )*
||
   a?va ; b!va ; ( a?va , p.b?vb ; b!(va max vb) )*
end.
```

Under the same assumptions as in the previous solutions for the duration of a communication (ρ) and the duration of the computation of a maximum (κ), enabling structure *M3* gives the behaviour of the head cell of $Max3_N$ for $N > 0$:

$$
\begin{array}{llll}
(4.18) & M3.s.a_i & = & \textbf{if } i = 0 \rightarrow \rho \\
 & & & [\!]\ i > 0 \rightarrow s.b_{i-1} + \rho \max s.pa_{i-1} + \rho \\
 & & & \textbf{fi} \\
 & M3.s.b_i & = & \textbf{if } i = 0 \rightarrow s.a_i + \rho \\
 & & & [\!]\ i > 0 \rightarrow s.a_i + \kappa + \rho \max s.pb_{i-1} + \kappa + \rho \\
 & & & \textbf{fi} \\
 & M3.s.pa_i & = & s.a_i + \rho \\
 & M3.s.pb_i & = & s.b_i + \rho
\end{array}
$$

$$\begin{tikzcd}
a_i \arrow[r, "\rho"] \arrow[dr, "\kappa+\rho"'] & p.a_i \arrow[r, "\rho"] & a_{i+1} \arrow[r, "\rho"] \arrow[dr, "\kappa+\rho"'] & {} \\
{} \arrow[r, "\kappa+\rho"'] & b_i \arrow[r, "\rho"'] \arrow[ur, "\rho"] & p.b_i \arrow[r, "\kappa+\rho"'] & b_{i+1}
\end{tikzcd}$$

Figure 4.19 Dependencies in cell *M3*.

The behaviour of $Max3_N$ is given by $MA3_0 = MA_0$ and

$$MA3_N \;=\; (M3 \parallel p.MA3_{N-1}) \shortparallel \{a, b\} \quad \text{for } N > 0 .$$

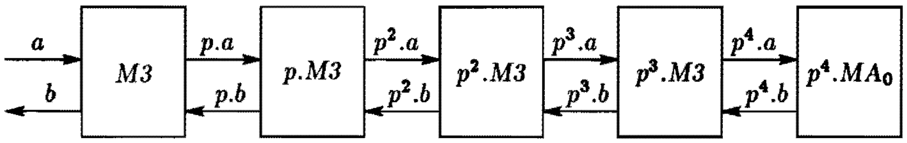


Figure 4.20 Schematic representation of array $MA3_4$.

The external behaviour of $MA3_N$ is the same as for $MA2_N$, without considering the tardiness of broadcasts, as given in Formula 4.16. A derivation of this external behaviour is given below. Consequently, $MA3_N$ has the same response time as $MA2_N$, when the tardiness of broadcasts is ignored. This result, however, is achieved *without* introduction of additional variables (as in $MA2_N$), and *without* the usage of a broadcast.

Derivation of the external behaviour for $MA3_N$, as given in Formula 4.16.

We derive $(M3 \parallel p.f_n) \Vert\!\upharpoonright \{a, b\} \approx f_{n+1}$ for $n > 0$. For $n = 0$ the derivation is similar.

Let $n > 0$ and let $e_0 = M3 \parallel p.f_n$:

$$\begin{array}{lcll}
e_0.s.a_i & = & \mathbf{if} & i = 0 \;\rightarrow\; \rho \\
 & & [\,] & i > 0 \;\rightarrow\; s.b_{i-1} + \rho \;\max\; s.pa_{i-1} + \rho \\
 & & \mathbf{fi} & \\
e_0.s.b_i & = & \mathbf{if} & i = 0 \;\rightarrow\; s.a_i + \rho \\
 & & [\,] & i > 0 \;\rightarrow\; s.a_i + \kappa + \rho \;\max\; s.pb_{i-1} + \kappa + \rho \\
 & & \mathbf{fi} & \\
e_0.s.pa_i & = & \mathbf{if} & i = 0 \;\rightarrow\; s.a_i + \rho \;\max\; \rho \\
 & & [\,] & i > 0 \;\rightarrow\; s.a_i + \rho \;\max\; s.pb_{i-1} + \rho \\
 & & \mathbf{fi} & \\
e_0.s.pb_i & = & \mathbf{if} & i = 0 \;\rightarrow\; s.b_i + \rho \;\max\; s.pa_i + \rho \\
 & & [\,] & i > 0 \;\rightarrow\; s.b_i + \rho \;\max\; s.pa_i + \kappa + \rho \\
 & & \mathbf{fi} &
\end{array}$$

First we hide $p.a$ by assuming that $p.a_i$ happens as soon as it is enabled.

Let $e_1 = e_0 \Vert\!\upharpoonright \{a, b, p.b\}$:

$$\begin{array}{lcll}
e_1.s.a_i & = & \mathbf{if} & i = 0 \;\rightarrow\; \rho \\
 & & [\,] & i = 1 \;\rightarrow\; s.b_{i-1} + \rho \;\underline{\max\; s.a_{i-1} + 2\rho \;\max\; 2\rho} \\
 & & [\,] & i > 1 \;\rightarrow\; s.b_{i-1} + \rho \;\underline{\max\; s.a_{i-1} + 2\rho} \\
 & & & \qquad\qquad \underline{\max\; s.pb_{i-2} + 2\rho} \\
 & & \mathbf{fi} & \\
e_1.s.b_i & = & \mathbf{if} & i = 0 \;\rightarrow\; s.a_i + \rho \\
 & & [\,] & i > 0 \;\rightarrow\; s.a_i + \kappa + \rho \;\max\; s.pb_{i-1} + \kappa + \rho \\
 & & \mathbf{fi} &
\end{array}$$

$$\begin{array}{lcllll}
e_1.s.pb_i & = & \textbf{if} & i=0 & \rightarrow & s.b_i+\rho \text{ max } s.a_i+2\rho \ \underline{\text{max } 2\rho} \\
 & & [\!] & i>0 & \rightarrow & s.b_i+\rho \text{ max } s.a_i+\kappa+2\rho \\
 & & & & & \underline{\text{max } s.pb_{i-1}+\kappa+2\rho} \\
 & & \textbf{fi} & & &
\end{array}$$

Next we prune the underlined (redundant) dependencies, which results in e_2 , $e_2 \approx e_1$:

$$\begin{array}{lcllll}
e_2.s.a_i & = & \textbf{if} & i=0 & \rightarrow & \rho \\
 & & [\!] & i>0 & \rightarrow & s.b_{i-1}+\rho \\
 & & \textbf{fi} & & & \\
e_2.s.b_i & = & e_1.s.b_i & & & \\
e_2.s.pb_i & = & \textbf{if} & i=0 & \rightarrow & s.b_i+\rho \text{ max } s.a_i+2\rho \\
 & & [\!] & i>0 & \rightarrow & s.b_i+\rho \text{ max } s.a_i+\kappa+2\rho \\
 & & \textbf{fi} & & &
\end{array}$$

Now it is time to hide the $p.b$ actions; let $e_3 = e_2 \Vert\!\upharpoonright \{a,b\}$:

$$\begin{array}{lcllll}
e_3.s.a_i & = & \textbf{if} & i=0 & \rightarrow & \rho \\
 & & [\!] & i>0 & \rightarrow & s.b_{i-1}+\rho \\
 & & \textbf{fi} & & & \\
e_3.s.b_i & = & \textbf{if} & i=0 & \rightarrow & s.a_i+\rho \\
 & & [\!] & i=1 & \rightarrow & s.a_i+\kappa+\rho \\
 & & & & & \underline{\text{max } s.b_{i-1}+\kappa+2\rho \text{ max } s.a_{i-1}+\kappa+3\rho} \\
 & & [\!] & i>1 & \rightarrow & s.a_i+\kappa+\rho \\
 & & & & & \underline{\text{max } s.b_{i-1}+\kappa+2\rho \text{ max } s.a_{i-1}+2\kappa+3\rho} \\
 & & \textbf{fi} & & &
\end{array}$$

When this enabling function is pruned properly, one obtains f_{n+1} .

□

Observe that this (the third) solution for computing the maximum of finite segments of the input can be generalized to $N = \infty$. This results in the infinite array $(\ \Vert\ n : n \geqslant 0 : p^n.M3\ ) \shortparallel \{a,b\}$,

that computes: $b(i) \ = \ (\ \mathbf{max}\, j : 0 \leqslant j \leqslant i : a(j)\ )$.

This array has the same external behaviour as $MA3_N$ for $N > 0$, and thus a response time of $\kappa + \rho$.

However, it is far more convenient to use a one cell array that performs the following program:

```
program Max∞ ( input a: integer , output b: integer ) :
var va, vb: integer ;
begin
   a?vb ; b!vb ; ( a?va ; vb := va max vb ; b!vb )*
end.
```

Third solution revisited

When implementing the previous solutions in hardware, one needs a 'function block' that computes the maximum of two integers. In fact, such a function block is used in each cell of the array. It may be interesting to reduce the number of function-blocks in a hardware implementation, possibly at the cost of some speed. We discuss sharing of the max-block between two neighbour cells in the third solution.

When a cell $M3$ and its neighbour $p.M3$ share a max-block, this imposes restrictions on the moments that values $b(i)$ and $p.b(i)$ can be computed. We assume the max-block must be used under mutual exclusion and that:

- a cell initiates a max-computation as soon as the inputs are available and the use of the max-block is assigned to this cell,
- the max-block is released by a cell as soon as the output of the computation has been communicated along b (or $p.b$).

Apart from the first outputs, b_0 and $p.b_0$, for each output b_i and $p.b_i$ the max-block is needed. Since b and $p.b$ actions alternate, the usage of the max-block can be assigned to $M3$ and $p.M3$ in an alternating way; this results in the following additional restrictions on the communications along b and $p.b$:

$$\begin{array}{lll} s.b_i & \geqslant & s.pb_{i-1} + \kappa + \rho \\ s.pb_i & \geqslant & s.b_i + \kappa + \rho \qquad \text{for } i > 0 \end{array}$$

These restrictions can be captured in a modification of $M3$:

$$\begin{array}{llll} (4.21) & M3'.s.x & = & M3.s.x \qquad\qquad \text{for } x \neq p.b_i \\ & M3'.s.pb_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ s.b_i + \rho \\ & & & [\!]\ \ i > 0 \ \rightarrow\ s.b_i + \kappa + \rho \\ & & & \textbf{fi} \end{array}$$

The behaviour of an implementation in which max-blocks are shared by two cells each, can be described by an array, say $MA3''_N$, in which cells of type $M3$ and $M3'$ alternate: cell $p^k.M3'$ shares a max-block with cell $p^{k+1}.M3$.

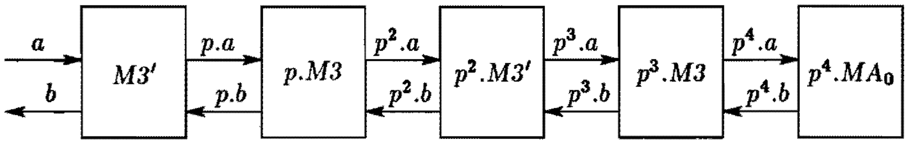


Figure 4.22 Schematic representation of array $MA3''_4$.

In order to estimate the behaviour of such an array, we analyze the behaviour of arrays $MA3'_N$ in which all cells behave as $M3'$. The external behaviour of these arrays are given by:

(4.23) $$\begin{array}{lcl} f'_N.s.a_i & = & \textbf{if}\ \ i=0 \rightarrow \rho \\ & & [\!]\ \ i>0 \rightarrow s.b_{i-1}+\rho \\ & & \textbf{fi} \\ f'_N.s.b_i & = & \textbf{if}\ \ i=0 \vee N=0 \rightarrow s.a_i+\rho \\ & & [\!]\ \ i>0 \wedge N>0 \rightarrow s.a_i+\kappa+\rho \\ & & \qquad\qquad\qquad\qquad \max\ s.b_{i-1}+2(\kappa+\rho) \\ & & \textbf{fi} \end{array}$$

The derivation of this behaviour is left as an exercise to the reader.

For $N>0$ the response time of $MA3'_N$ with respect to $(a\ ;\ b)^*$ is $\kappa+\rho$, which is the same as the response time of $MA3_N$.

But what about the arrays we are interested in: $MA3"_N$? Since MA_0, $M3$, and $M3'$ are conservative, and $M3 \lhd\!\! \mid M3'$, we can estimate the behaviour of these arrays by:

$MA_N \ \lhd\!\!\mid\ MA"_N \ \lhd\!\!\mid\ MA'_N$.

Consequently the response time of MA_N is a lower bound for the response time of $MA"_N$ and the response time of MA'_N is an upper bound. We conclude that for $N>0$ the response time of $MA3"_N$ with respect to $(a\ ;\ b)^*$ is $\kappa+\rho$.

Though $MA3_N$ and $MA3'_N$ have the same response time, they are not equally fast: the response time of $MA3'_N$ with respect to $MA3_N$ is $2(\kappa+\rho)\,/\,(\kappa+2\rho)$ (for $N>0$). The real implementation $MA3"_N$ does not do any better relative to $MA3_N$: for $N \geqslant 2$ there is at least one shared max-block, which results in a 'period' of at least $2(\kappa+\rho)$ while the period of $MA3_N$ is $\kappa+2\rho$.

We did *not* need to compute the external behaviours of the arrays $MA3"_N$ in order to draw these conclusions about their speed. This should not prevent the interested reader from trying to compute these behaviours (observe that for odd values of N, the behaviour dependends on the position of the cell that does not share its max-block).

Chapter 5

Distributed Implementations of FIFO buffers

This thesis is not about data but about real-time behaviour. Therefore it is convenient to perform a case-study on mechanisms that are especially designed for some specific real-time behaviour, under the restriction of a simple relation between data; First In First Out buffers enjoy this property. In this chapter we discuss the performance of distributed implementations of FIFO buffers. The main difference in performance, of the implementations we discuss, is manifested in extreme situations: buffers that are almost full, and buffers that are almost empty.

In this introduction we give a way of reasoning about distributed implementations of FIFO buffers. This reasoning in terms of *packets* and *holes* is used in Section 5.2. Furthermore we exhibit 'the problem of full buffers' as well as 'the problem of empty buffers'.

In Section 4.3 we already encountered the one-place buffer. A program for the one-place buffer with input channel a and output channel b is given by:

```
program BUF_1 ( input a, output b) :
var x ;
begin
   ( a?x ; b!x )*
end.
```

The buffer receives *packets* (in variable x) during communications along channel a , and it gives packets to the environment during communications along channel b . Observed from a dual point of view this can be phrased as: the buffer receives *holes* during b communications, and it gives holes to the environment during a communications. Initially, the one-place buffer contains a hole. The view of communications as exchanges of packets and holes is essential in our reasoning about implementations.

In this chapter we assume delays of one time unit; this results in the following description of the one-place buffer:

$$(5.1)\quad \begin{array}{lcl} B_1.s.a_i &=& \textbf{if}\ \ i=0 \ \rightarrow\ 1 \\ && [\!]\ \ i>0 \ \rightarrow\ s.b_{i-1}+1 \\ && \textbf{fi} \\ B_1.s.b_i &=& s.a_i+1 \end{array}$$

Figure 5.2 shows two states of a cascade of five one-place buffers. The circles symbolize variables, the tokens symbolize packets. Packets can only move from the left to the right; they are obtained from the environment at the left-hand side, and are returned to the environment at the right-hand side. In [2] such implementations are called 'ripple' buffers. In one time unit the first state of Figure 5.2 develops into the second state; during this time unit, two packets have changed places with holes.

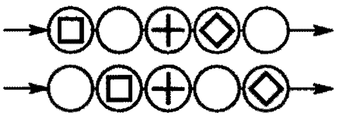
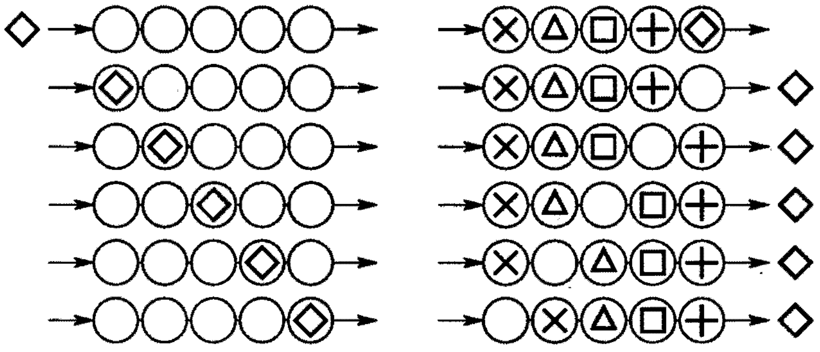


Figure 5.2 A cascade of five one-place buffers.

Now consider, see Figure 5.3 (a), a cascade of one-place buffers that contains no packets. When a packet is obtained from the environment (in exchange for a hole), this packet has to travel through all variables before it can be returned to the environment (again in exchange for a hole). So the response time of cascades that are almost empty is proportional to the number of variables they contain. By exchanging the dual concepts 'hole' and 'packet' in the previous observation, the same conclusion is drawn for cascades that are almost full, see Figure 5.3 (b).



(a) almost empty (b) almost full

Figure 5.3 Successive states of a cascade.

The performance of cascades is optimal when they are approximately half full. In this case a continuous stream of packets moves in one direction, and a continuous stream of holes moves in the opposite direction (Figure 5.4). In this case the number of variables that is used is about twice the number of packets that are in the buffer, which is a considerable overhead. In VLSI implementations such an overhead is un-desired because variables are —relatively— expensive. In software implementations, however, the costs of variables are usually relatively low.

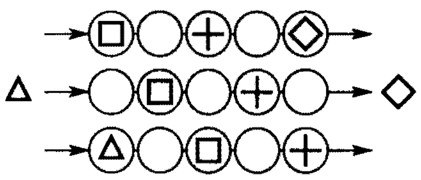


Figure 5.4 Optimal behaviour of a cascade.

Overview

In Section 5.1 we perform a more formal analysis of the behaviour of linear conservative implementations. Furthermore we introduce the type of specification, $B_{q,p}$, we want to implement. $B_{q,p}$ specifies the behaviour of a FIFO buffer that operates with delays of one time unit when the number of packets it contains lies within the capacity range from $q-1$ up to p.

In Section 5.2 we give theoretical bounds for the overhead (in variables) that is needed to implement FIFO buffers of type $B_{p,p}$ with a non-zero quality. These bounds are under the assumption that it takes a hole to move a packet. Furthermore, the bounds depend on the shape of the implementations. We discuss implementations that are linear arrangements of 'cells', implementations in which cells have a limited number of neighbours, and implementations in which variables have 'limited surroundings'; the latter modeling restrictions on the layout of implementations. The overhead that is needed turns out to be (respectively) linear in p, logarithmic in p, and proportional to the square root of p. The implementations in Section 5.1 meet the bound for linear implementations, in that the overhead is linear in p. In Section 5.4 we give hierarchical designs of conservative implementations that meet the bounds for both other types of implementations.

The problem with these conservative implementations is that they are symmetrical in packets and holes. Therefore they do not only have a poor performance when they are almost full, but also when they are almost empty. In order to solve this problem, empty parts of a buffer can be bypassed. Buffers with bypassing are discussed in Section 5.5.

The implementations in Sections 5.1 and 5.4 are special cases of the — hypothetical — implementations we analyze in Section 5.3. In Section 5.5 we use these hypothetical implementations to estimate the behaviour of bypass buffers.

Usually, FIFO buffers are not implemented in a distributed way. The behaviour of distributed implementations of FIFO buffers, however, is typical for pipelines and for systolic arrays that communicate to the environment at both sides. In Section 5.6 we briefly discuss some general examples.

5.1 Linear conservative implementations

Composing two one-place buffers in cascade gives a buffer that can contain two packets.

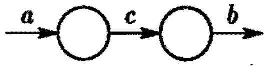


Figure 5.5 A cascade of two one-place buffers.

The external behaviour of this cascade is given by:

$$(5.6)\quad \begin{array}{lcllll}
CAS_2.s.a_i & = & \mathbf{if} & i=0 & \rightarrow & 1 \\
 & & [\!] & i=1 & \rightarrow & s.a_{i-1}+2 \\
 & & [\!] & i \geqslant 2 & \rightarrow & s.a_{i-1}+2 \text{ max } s.b_{i-2}+2 \\
 & & \mathbf{fi} & & & \\
CAS_2.s.b_i & = & \mathbf{if} & i=0 & \rightarrow & s.a_i+2 \\
 & & [\!] & i \geqslant 1 & \rightarrow & s.a_i+2 \text{ max } s.b_{i-1}+2 \\
 & & \mathbf{fi} & & &
\end{array}$$

A derivation of this behaviour is given below.

In the cascade, the delay between an input a_i and output b_i is two, instead of the one in a one-place buffer. This is not surprising, since the packets have to travel through two one-place buffers, instead of one. Similar for the delay between b_i and a_{i+2}: the holes also have to travel through two one-place buffers, instead of one.

Derivation of the behaviour of two one-place buffers in cascade.

The external behaviour of a cascade of two one place buffers (see Figure 5.5) is $e_0 \,\|\!\!\upharpoonright \{a,b\}$, where e_0 is given by $e_0 \;=\; B_{1\,b\rightarrow c} \;\|\; B_{1\,a\rightarrow c}$ (see Formula 5.1 for B_1):

$$\begin{array}{lcllll}
e_0.s.a_i & = & \mathbf{if} & i=0 & \rightarrow & 1 \\
 & & [\!] & i \geqslant 1 & \rightarrow & s.c_{i-1}+1 \\
 & & \mathbf{fi} & & & \\
e_0.s.b_i & = & s.c_i+1 & & & \\
e_0.s.c_i & = & \mathbf{if} & i=0 & \rightarrow & s.a_i+1 \; \underline{\text{max } 1}\,[1] \\
 & & [\!] & i \geqslant 1 & \rightarrow & s.a_i+1 \text{ max } s.b_{i-1}+1 \\
 & & \mathbf{fi} & & &
\end{array}$$

In order to compute $e_1 = e_0 \,\|\!\!\upharpoonright \{a,b\}$, we assume that c actions happen as soon as they are enabled:

[1] As in the previous chapter, redundant dependencies are underlined.

$$
\begin{array}{lcllll}
e_1.s.a_i & = & \textbf{if} & i = 0 & \rightarrow & 1 \\
 & & [] & i = 1 & \rightarrow & s.a_{i-1} + 2 \ \underline{\max\ 2} \\
 & & [] & i > 1 & \rightarrow & s.a_{i-1} + 2 \ \max\ s.b_{i-2} + 2 \\
 & & \textbf{fi} & & & \\
e_1.s.b_i & = & e_1.s.a_{i+1} & & &
\end{array}
$$

Pruning the redundant term $\max 2$ in the enabling of $s.a_1$ (and $s.b_0$) gives CAS_2.

□

In order to obtain bigger buffers, more one-place buffers can be placed in cascade. CAS_v gives the behaviour of a cascade of v one-place buffers:

$$
\begin{array}{llcllll}
(5.7) & CAS_v.s.a_i & = & \textbf{if} & i = 0 & \rightarrow & 1 \\
 & & & [] & 0 < i < v & \rightarrow & s.a_{i-1} + 2 \\
 & & & [] & i \geqslant v & \rightarrow & s.a_{i-1} + 2 \ \max\ s.b_{i-v} + v \\
 & & & \textbf{fi} & & & \\
 & CAS_v.s.b_i & = & \textbf{if} & i = 0 & \rightarrow & s.a_i + v \\
 & & & [] & i \geqslant 1 & \rightarrow & s.a_i + v \ \max\ s.b_{i-1} + 2 \\
 & & & \textbf{fi} & & &
\end{array}
$$

This formula is a generalization of Formula 5.6 for the cascade of two one place buffers. It is a consequence of Formula 5.32 for the cascade of FIFO buffers of type 'IH' with a one place buffer ($CAS_v = IH(v, v, 2, v)$). Observe that though CAS_1 has some redundant dependencies, it is equivalent to the one place buffer B_1.

We are interested in buffers with a good performance over a given capacity range. First we consider specifications of type B_p, for $p > 0$. These specify buffers that have a capacity range from $p - 1$ up to p: that is, apart from the initialization phase, they can contain $p - 1$ or p packets.

$$
\begin{array}{llcllll}
(5.8) & B_p.s.a_i & = & \textbf{if} & i = 0 & \rightarrow & 1 \\
 & & & [] & 0 < i < p & \rightarrow & s.a_{i-1} + 2 \\
 & & & [] & i \geqslant p & \rightarrow & s.b_{i-p} + 1 \\
 & & & \textbf{fi} & & & \\
 & B_p.s.b_i & = & s.a_{i+p-1} + 1 & & &
\end{array}
$$

In terms of choice-free commands, the behaviour is given by $(a\,;\,\tau)^{p-1}\ ;\ (a\,;\,b)^*$. The definition of B_1 is in accordance with Formula 5.1 which gives the behaviour of an implementation of a one-place buffer.

The quality of cascade CAS_v with respect to specification B_p is given by:

$$
\begin{array}{llcllll}
(5.9) & Q(CAS_v, B_p) & = & \textbf{if} & 0 < p \leqslant (v+1)/2 & \rightarrow & (2p-1)/v \\
 & & & [] & (v+1)/2 \leqslant p < v + 1 & \rightarrow & (2v - (2p-1))/v \\
 & & & [] & p \geqslant v + 1 & \rightarrow & 0 \\
 & & & \textbf{fi} & & &
\end{array}
$$

This is a special case of the quality of buffers of type IH, as given in Formula 5.27.

Figure 5.10 gives the quality of CAS_v, dependent on which specification is to be implemented.

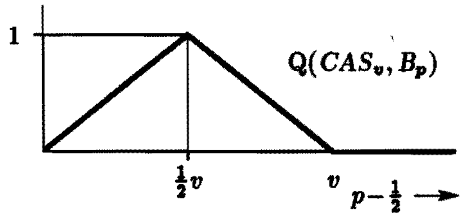


Figure 5.10 Performance of a CAS_v buffer.

The more logical approach is to state a specification B_p and to consider the quality of several implementations; this is done in Figure 5.11.

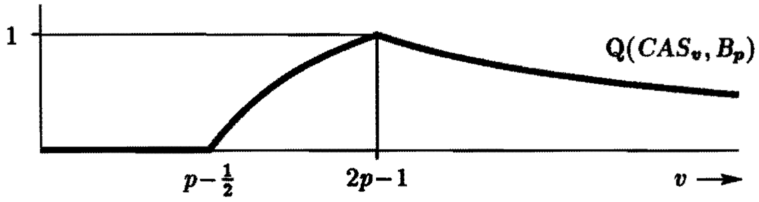


Figure 5.11 Performance of CAS_v buffers.

In both figures, we observe that quality 1 is obtained when the number of variables is approximately twice the number of packets: for implementing a buffer for p packets, a cascade buffer must have about $2p$ variables; this is an overhead of p variables. From Formula 5.9 (second alternative) we infer the amount of overhead that is needed to obtain quality Q, $0 < Q \leqslant 1$:

$$(5.12) \quad v - p \;=\; \frac{p * Q - 1}{2 - Q}$$

This overhead is linear in p.

B_p buffers have a very small capacity range: from $p-1$ up to p. We generalize the specification to buffers with capacity range from $q-1$ up to p, for $0 < q \leqslant p$:

```
(5.13)  $B_{q,p}.s.a_i$  =  if  $i = 0$          $\rightarrow$  $1$
                            ▯  $0 < i < p$     $\rightarrow$  $s.a_{i-1} + 2$
                            ▯  $i \geqslant p$        $\rightarrow$  $s.a_{i-1} + 2 \max s.b_{i-p} + 1$
                            fi
        $B_{q,p}.s.b_i$  =  if  $i = 0$  $\rightarrow$  $s.a_{i+q-1} + 1$
                            ▯  $i > 0$  $\rightarrow$  $s.a_{i+q-1} + 1 \max s.b_{i-1} + 2$
                            fi
```

$B_{p,p}$ has some redundant dependencies; it is, however, equivalent to B_p (see Formula 5.8). The quality of the cascade with respect to specifications of type

$B_{q,p}$ is given by:

(5.14) $\mathrm{Q}(CAS_v, B_{q,p}) \; = \; \mathrm{Q}(CAS_v, B_q) \min \mathrm{Q}(CAS_v, B_p)$

This is a (again) a special case of the quality of buffers of type IH with respect to specifications of type $B_{q,p}$ (see Formulae 5.27 and 5.29).

In the cascade buffers, the delays between successive inputs and successive outputs are (at least) two: only one of every two time units an input can be performed (similar for outputs). This restriction is imposed by the building block B_1 we used. Under the assumption of unit delays, an implementation can perform at most one input (output) per time unit. This result can be achieved by receiving successive 'input packets' in different variables, and sending successive 'output packets' from different variables. The following 'wagging scheme'[2] can be used to do so:

```
program Wagging Buffer (input a, output b) :
var x, y ;
begin
   a?x ; (a?y , b!x ; a?x , b!y )*
end.
```

Under the assumption of unit delays, the behaviour of this program is given by:

(5.15)
$$
\begin{array}{rcll}
BW_2.s.a_i & = & \textbf{if} & i = 0 \;\rightarrow\; 1 \\
 & & [] & i = 1 \;\rightarrow\; s.a_{i-1} + 1 \\
 & & [] & i \geqslant 2 \;\rightarrow\; s.a_{i-1} + 1 \max s.b_{i-2} + 1 \\
 & & \textbf{fi} & \\
BW_2.s.b_i & = & BW_2.s.a_{i+1} &
\end{array}
$$

Apart from the initial delay, this program is twice as fast as CAS_2 :

(5.16) $BW_2 \; = \; 1/2 \odot (CAS_2 \oplus 1)$

(see Formula 5.6). Consequently, a cascade of k wagging buffers is (apart from the initial delay) twice as fast as a cascade of $2k$ one-place buffers.

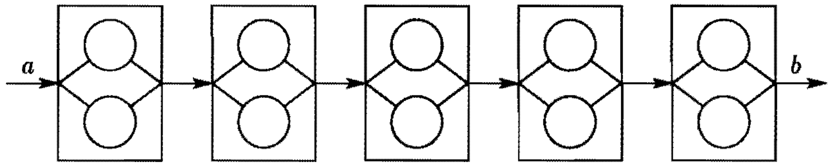


Figure 5.17 Cascade of five wagging buffers.

The comparison of cascades of one-place buffers and cascades of wagging buffers, is under the assumption of identical delays (one time unit). Wagging buffers, however, have a drawback: in hardware, the 'wagging' of an input stream to two variables (plus the merge to one output stream) tends to be *more expensive* and *slower* than sending (and receiving) all values to (from) the same variable.

[2] Terminology from [2].

These drawbacks can be considerably reduced by using one 'split cell' to split the input stream, two parallel cascades of one-place buffers, and one 'merge cell' to merge both packet streams into one output stream.

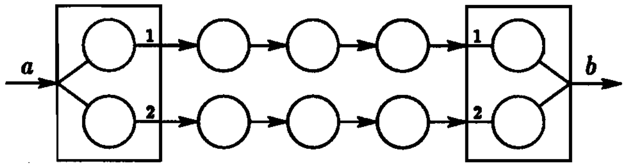


Figure 5.18 Optimization of the cascade in figure 5.17.

The programs of the wagging cells are given by:

```
program Wagging Split(input a, output b1,b2) :
var x, y ;
begin
   a?x ; (a?y ,b1!x ; a?x ,b2!y )*
end.

program Wagging Merge(input a1,a2, output b) :
var x, y ;
begin
   a1?x ; (a2?y ,b!x ; a1?x ,b!y )*
end.
```

Without proof we mention that, under the assumption of unit delays, such a construction has the same behaviour as a cascade of wagging buffers.

The problem with the implementations we presented in this section, is that they have a poor performance when they are relatively full and when they are relatively empty.

In order to implement B_p with quality Q, $0 < Q \leqslant 1$, a linear overhead of variables is needed. The wagging scheme is twice as fast as the cascade of one-place buffers, but the overhead is still linear. In the next section we give lower bounds for the overhead that is needed; in Section 5.4 we give conservative implementations that meet these bounds.

The fastest cascade implementation of $B_{q,p}$ with one-place buffers is a CAS_{p+q-1} buffer; the quality of this implementation is $(2q-1)/(p+q-1)$. When the specification has a relatively long capacity range ($p >> q$) this gives a poor performance. Again, the wagging scheme is twice as fast, but this is only a constant factor. The implementations in Section 5.4 have a better performance for relatively empty buffers; if, however, an implementation must have a good performance when it is even 'more empty', this does not suffice. The implementations with bypassing in Section 5.5 are even good when they are empty.

5.2 Lower bounds for the overhead

In this section we derive lower bounds for the overhead of implementations for B_p buffers. Rather than in the exact overhead for individual implementations, we are interested in the order of magnitude of the overhead for complete 'implementation schemes'. For example, in order to implement buffers of type B_p with cascades of one-place buffers, with a quality that does not tend to zero for large values of p, a linear overhead is required (see Formula 5.12). In the derivation of lower bounds, we assume that the external behaviours of implementations can be described with enabling functions. Furthermore we assume that the implementations communicate with their environment by sending and receiving of packets only. Not all implementations of FIFO buffers enjoy these properties. For example, the programs that are given in [12, 15] interleave their input and output actions (rather than performing them in a concurrent way), and within the enabling model this interleaving cannot be described without introduction of additional channels, along which the environment requests for inputs and outputs. However, the results we achieve can be generalized to all kinds of implementations, as long as the communication of packets in these implementations is based on the principle of exchanging packets and holes. A similar derivation is given in [12].

The lower bounds we give depend critically on the assumption that it takes a hole to move a packet, and that moving a packet takes at least one time unit. The other cornerstone in the derivation of lower bounds is the FIFO strategy; this strategy ensures that the contents of a buffer that initially contains p packets is completely refreshed after p outputs and p inputs.

We distinguish three types of implementations. In the first two, we assume a partition of the implementation into building blocks that have a limited number of variables. These building blocks are called *cells*. The types of implementations are:

- linear implementations (cells have at most two neighbours),
- implementations in which cells have a limited number of neighbours (> 2),
- implementations in which variables have limited surroundings.

First we discuss a general way to determine lower bounds.

A general lower bound

All lower bounds we give are based on the fact that it takes a hole to move a packet, and that moving a packet takes at least one time unit.

Consider the experiment of Figure 5.19, in which a distributed FIFO buffer with v variables, *FIFO* , is connected to a one place buffer that initially produces p packets:

```
program $\tilde{B}_p$ (input $b$, output $a$) :
var $x$ ;
begin
   ( $a$! any value ; $\tau$ )$^{p-1}$ ; $a$! any value ; ( $b$?$x$ ; $a$!$x$ )$^*$
end.
```

where every action, including the internal action τ, takes 1 time unit. The external behaviour of this program is B_p.

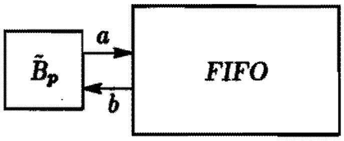


Figure 5.19 The experiment.

Packets move from $\tilde{B}_p$ to *FIFO* along channel a; along channel b they move back again. Consider the situation after communication of the p initial packets from $\tilde{B}_p$ to *FIFO*; the remaining communication pattern is $(b\ ;\ a)^*$.

The number of moves per time unit is at most the number of holes: $v+1-p$. We conclude that, if M moves are required to perform $(b\ ;\ a)^p$, this takes at least $M/(v+1-p)$ time units. Since $(b\ ;\ a)^p$ comprises $2p$ communications, the number of communications per time unit is at most $2p(v+1-p)/M$; this observation is used in the first step of the following derivation. We derive an upper bound for the quality of *FIFO* with respect to B_p:

$$\frac{2p(v+1-p)}{M}$$

$\geqslant$ { see above }

$Q(FIFO \parallel B_p\, ,\, B_p)$

$=$

$Q(FIFO \parallel B_p\, ,\, B_p \parallel B_p)$

$\geqslant$

$Q(FIFO, B_p) \text{ min } Q(B_p, B_p)$

$=$

$Q(FIFO, B_p) \text{ min } 1$

$=$ { see below }

$Q(FIFO, B_p)$

Due to the assumption of unit delays, *FIFO* cannot perform its first action before moment one, which results in a quality of at most one.[3]

Remains to estimate the number of moves, M, that is needed to perform $(b\ ;\ a)^p$. Due to the First In First Out strategy of *FIFO*, each packet vis-

[3] If we allow a longer initial delay, by using specification $B_p \oplus 1$, the term min 1 in the second last formula of the derivation cannot be removed that easily. This does, however, not affect the order of magnitude of the estimated overhead.

its $\tilde{B}_p$ exactly once, during the performance of $(b\ ;\ a)^p$. Consequently, M is the number of moves it takes to get all packets out of their original position, via $\tilde{B}_p$, into their final positions. A lower bound, δ_{out}, for the number of moves towards $\tilde{B}_p$ is obtained by assuming that all packets take the shortest path to $\tilde{B}_p$, and that the initial packing is such that the sum of these shortest paths is minimal. In a similar way, a lower bound, δ_{in}, for the total number of moves from $\tilde{B}_p$ to the final positions is obtained. Since $M \geqslant \delta_{out} + \delta_{in}$, we infer (see derivation above):

$$(5.20)\quad \mathrm{Q}(\mathit{FIFO}, B_p) \ \leqslant\ \frac{2p(v+1-p)}{\delta_{out}+\delta_{in}}$$

This can be rewritten into the following lower bound for the overhead in variables, $v-p$, that is necessary in order to achieve a quality of at least Q :

$$(5.21)\quad v-p \ \geqslant\ \mathrm{Q} * \frac{\delta_{out}+\delta_{in}}{2p} - 1$$

When designing an implementation scheme for buffers of type B_p, the overhead $v-p$ must at least be proportional to $(\delta_{out}+\delta_{in})/p$ in order to obtain a performance that does not tend to zero for large buffers.

Linear implementations

If an implementation can be partitioned into cells, such that each cell has at most two neighbours, we call it *linear*. The neighbours of a cell are those cells with which it can communicate packets (in at least one direction). When a cell can communicate with the environment ($\tilde{B}_p$), the environment is also considered a neighbour of this cell.

In this section we consider linear implementations with *grain size* g. That is, each cell in the partition contains at most g variables. It turns out that under these conditions the overhead is at least linear in the number of packets. The cascades of one-place buffers, and those of wagging buffers (see the previous section) are examples of linear implementations; their grain sizes are one and two respectively, and both have a linear overhead.

All we have to do, in order to obtain a lower bound for the overhead, is to give lower bounds for δ_{out} and δ_{in}. For reasons of symmetry, it suffices to consider δ_{out} only.

In any implementation, there is exactly one cell with distance one to the output: the cell with b as an output channel. In a linear implementation, this cell has at most two neighbours, and the environment is one of them; so there is at most one cell with distance two to the output. In general, for each $i : i > 0$ there is at most one cell with distance i to the output. Since each cell contains at most g packets, an upper bound of the number of packets within distance d to the output is given by:

$$\begin{aligned}
& P.d \\
= \; & (\sum i : 0 < i \leqslant d : g) \\
= \; & d * g
\end{aligned}$$

Let d and g' such that $p = P.d + g'$ for $0 \leqslant g' < g$. A packet at distance i has to move at least i times before it reaches $\tilde{B}_p$. This results in the following estimation for δ_{out}:

$$\begin{aligned}
& \delta_{out} \\
\geqslant \; & (\sum i : 0 < i \leqslant d : g * i) + g'(d+1) \\
= \; & \quad \{\text{calculus}\} \\
& gd(d+1)/2 + g'(d+1) \\
= \; & (d+1)(gd + 2g')/2 \\
\geqslant \; & p^2/2g
\end{aligned}$$

The same lower bound holds for δ_{in}.

From Formula 5.21 we infer the following lower bound for the overhead:

(5.22) $v - p \;\; \geqslant \;\; Q * p/2g - 1$

As expected, the overhead for linear implementation schemes is at least linear in the number of packets.

Implementations with limited neighbourhood

We consider distributed FIFO implementations with grain size g, in which cells have at most $F+1$ neighbours; $F \geqslant 2$. It turns out that under these conditions, the overhead is at least logarithmic in the number of packets.

For δ_{out} we take the sum of the distances of the packets to the output, under the assumption of a close packing to the output. At distance one to the output, at most g packets can be sited; at distance two, at most $g * F$ packets. In general, at distance $i+1$ at most $g * F^i$ packets can be sited. Consequently, an upper bound for the number of packets within distance d to the output is given by:

$$\begin{aligned}
& P.d \\
= \; & g * (\sum i : 0 \leqslant i < d : F^i) \\
= \; & \quad \{\text{calculus}\} \\
& g * \frac{F^d - 1}{F - 1}
\end{aligned}$$

For reasons of simplicity we assume $p = P.d$ (for some d). Without proof we mention that the lower bound we derive for δ_{out} also holds for other values of p.

δ_{out}

$\geqslant$

$g * (\sum i : 0 \leqslant i < d : (i+1) * F^i)$

$=$ $\{\text{calculus}\}$

$g * \dfrac{(d - \frac{1}{F-1}) * F^d + \frac{1}{F-1}}{F-1}$

$\geqslant$

$(d - \dfrac{1}{F-1}) * p$

$\geqslant$ $\{ \; d = \log_F(\frac{F-1}{g} * p + 1) \; \}$

$(\log_F .p - \log_F .g) * p$

Due to symmetry, the same lower bound holds for δ_{in}.

From Formula 5.21 we infer the following lower bound for the overhead:

(5.23) $v - p \;\geqslant\; Q * (\log_F .p - \log_F .g) - 1$

This lower bound is logarithmic in the number of packets.

Implementations with limited surroundings

In order to obtain the previous logarithmic lower bound for the overhead, the number of variables within a bounded 'moving' distance to the output must be exponential ($P.d$ is proportional to F^d). Due to physical limitations, it is unlikely that this is possible for large buffers. In this section we assume that implementations must have a two dimensional layout, in which the distance between neighbour variables is bounded, and in which the number of variables per square unit is bounded. It turns out that under these restrictions, the overhead must be at least proportional to the square root of p in order to obtain qualities that do not tend to zero for large values of p. This lower bound is independent of grain size and number of neighbours.

We choose the unit of (physical) length, such that the distance between neighbour variables is at most 1. With ν we denote the maximal number of variables per square unit of length.

The number of moves it takes for a packet to move from one variable, via a number of variables, to another variable, is at least the physical distance between the initial and the final position. A lower bound for δ_{out} and δ_{in} is thus obtained by assuming a physically close packing of packets to the output in the initial state and in the final state.

For the sake of convenience we assume that the variables are *continuously* distributed over the area, rather than occupying *discrete* positions. The number of variables within a circle of radius d around the output (input) is then at most $\nu\pi\, d^2$. Let d such that $\nu\pi\, d^2 = p$. The total physical distance to the output of the p closest variables is at least

$\int_0^{2\pi}\int_0^d \nu * r^2 \; dr \; d\theta$
$=$ { calculus }
$2\pi\nu * \int_0^d r^2 \; dr$
$=$ { calculus }
$2\pi\nu/3 \, d^3$
$=$
$2/3\,(\pi\nu)^{-1/2}\,p^{3/2}$

This is a lower bound for δ_{out} as well as δ_{in}. From Formula 5.21 we infer:

(5.24) $v - p \;\geqslant\; f * Q\sqrt{p} - 1$

for some factor $f : f > 0$ that is independent of a partition in cells, and of the corresponding values of g and F ($f = 2/3\,(\pi\nu)^{-1/2}$).

This lower bound is proportional to the square root of p. A similar lower bound is obtained when instead of a bounded distance between neighbour variables, the duration of a move is assumed to be proportional to the physical distance to be bridged.

5.3 Performance of hypothetical implementations

The *hypothetical implementations* we analyse in this section are used in the following sections to describe conservative implementations and to estimate implementations with bypassing. They depend on four parameters:

- the *latency* between input and output is α, $\alpha > 0$,
- the *latency* between output and input is β, $\beta > 0$,
- the *throughput delay* is γ, $\gamma > 0$, and
- the capacity is v, $v > 0$.

Independent of the throughput delay, we assume an *initial delay* of one.

The hypothetical implementation is given by:

(5.25)
$$\begin{array}{lcll} IH(\alpha,\beta,\gamma,v).s.a_i & = & \textbf{if} & i = 0 \;\rightarrow\; 1 \\ & & [\!] & 0 < i < v \;\rightarrow\; s.a_{i-1} + \gamma \\ & & [\!] & v \leqslant i \;\rightarrow\; s.a_{i-1} + \gamma \;\max\; s.b_{i-v} + \beta \\ & & \textbf{fi} & \\ IH(\alpha,\beta,\gamma,v).s.b_i & = & \textbf{if} & i = 0 \;\rightarrow\; s.a_i + \alpha \\ & & [\!] & 0 < i \;\rightarrow\; s.a_i + \alpha \;\max\; s.b_{i-1} + \gamma \\ & & \textbf{fi} & \end{array}$$

In all implementations we discuss, the capacity is equal to the number of variables. Furthermore we assume the following relation between the parameters:

(5.26) $\gamma v \;\geqslant\; \alpha + \beta$

This relation assures that a_i cannot be indirectly delayed by a_{i-v} via b_{i-v} (similar for b_i by b_{i-v} via a_i).

We introduce abbreviation $IH(\beta, v)$ for the 'symmetrical' buffer $IH(\beta,\beta,2,v)$. The cascades of one-place buffers are special cases of these implementations: $CAS_v = IH(v,v)$ (see Formula 5.7).

The quality of $IH(\alpha,\beta,\gamma,v)$, as implementation of a buffer with capacity range from $q-1$ up to p, $B_{q,p}$ (see Formula 5.13), is given by:

(5.27) $$Q(IH(\alpha,\beta,\gamma,v),B_{q,p}) \;=\; 0 \max \left(\frac{2q-1}{\alpha} \min \frac{2v-(2p-1)}{\beta} \min \frac{2}{\gamma} \min 1\right)$$

The last term, $\min 1$, is due to the initial delay: for the quality with respect to $B_{q,p} \oplus \mu$ it would be $\min 1+\mu$.

A derivation of this quality is given below.

Figure 5.28 shows the quality with respect to buffers of type B_p in case $\gamma \geqslant 2$; for $\gamma < 2$ the quality is the same as for $\gamma = 2$.

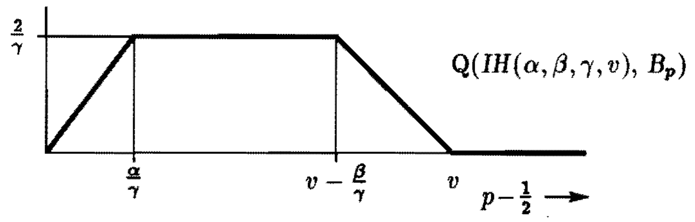


Figure 5.28 Quality of an *IH* buffer with $\gamma \geqslant 2$.

The general case is expressed by:

(5.29) $$Q(IH(\alpha,\beta,\gamma,v),B_{q,p}) \;=\; Q(IH(\alpha,\beta,\gamma,v),B_q) \min Q(IH(\alpha,\beta,\gamma,v),B_p)$$

Derivation of Formula 5.27.

The enabling functions involved are conservative, so it suffices to consider process inclusion. Let

$$\lambda_0 \;=\; 0 \max \left(\frac{2q-1}{\alpha} \min \frac{2v-(2p-1)}{\beta} \min \frac{2}{\gamma} \min 1\right).$$

We subsequently prove:

1 $\mathbf{P}B_{q,p} \subseteq \lambda \odot \mathbf{P}IH(\alpha,\beta,\gamma,v) \;\Rightarrow\; \lambda \leqslant \lambda_0$ (for $0 < \lambda < \infty$).

2 If $0 < \lambda_0$: $\mathbf{P}B_{q,p} \subseteq \lambda_0 \odot \mathbf{P}IH(\alpha,\beta,\gamma,v)$.

$B_{q,p}$ is given in Formula 5.13, *IH* buffers are defined in Formula 5.25.

1 Assume the left-hand side. For each term in the minimum of the formula for λ_0, we have to prove that λ is at most this term.

- $\lambda \leqslant 1$?

  Let $s = \mathbf{h}B_{q,p}$: $s.a_i = 2i+1 \quad s.b_i = 2i+2q$.

  Since $s.a_0 = 1$, and $\lambda \odot IH(\alpha,\beta,\gamma,v).s.a_0 = \lambda$, we conclude $\lambda \leqslant 1$.

- $\lambda \leqslant 2/\gamma$?

  Let $s = \mathbf{h}B_{q,p}$. For $i > 0$ we derive:

  $s \in \mathbf{P}(\lambda \odot IH(\alpha,\beta,\gamma,v))$

  $\Rightarrow$

  $s.a_i \geqslant (\lambda \odot IH(\alpha,\beta,\gamma,v)).s.a_i$

  $\Rightarrow$ $\{ s.a_i = 2i+1 ,\ s.a_{i-1} = 2i-1 \}$

  $2i+1 \geqslant 2i-1+\lambda\gamma$

  $\Leftrightarrow$

  $\lambda \leqslant 2/\gamma$

- $\lambda \leqslant (2q-1)/\alpha$?

  Let again $s = \mathbf{h}B_{q,p}$; we derive:

  $s \in \mathbf{P}(\lambda \odot IH(\alpha,\beta,\gamma,v))$

  $\Rightarrow$

  $s.b_i \geqslant (\lambda \odot IH(\alpha,\beta,\gamma,v)).s.b_i$

  $\Rightarrow$ $\{ s.b_i = 2i+2q ,\ s.a_i = 2i+1 \}$

  $2i+2q \geqslant 2i+1+\lambda\alpha$

  $\Leftrightarrow$

  $\lambda \leqslant (2q-1)/\alpha$

- $\lambda \leqslant (2v-(2p-1))/\beta$?

  Define schedule s by $s.a_i = 2i+1 \quad s.b_i = 2i+2p$.

  Observe that $s \in \mathbf{P}B_{q,p}$. Furthermore we derive for $i \geqslant v$:

  $s \in \mathbf{P}(\lambda \odot IH(\alpha,\beta,\gamma,v))$

  $\Rightarrow$

  $s.a_i \geqslant (\lambda \odot IH(\alpha,\beta,\gamma,v)).s.a_i$

  $\Rightarrow$ $\{ s.a_i = 2i+1 ,\ s.b_{i-v} = 2(i-v)+2p \}$

  $2i+1 \geqslant 2(i-v)+2p+\lambda\beta$

  $\Leftrightarrow$

  $\lambda \leqslant (2(v+1-p)-1)/\beta$

2 Assume $\lambda_0 > 0$.

It suffices to prove $(\lambda_0 \odot IH(\alpha,\beta,\gamma,v)).s \leqslant B_{q,p}.s$ for $s \in \mathbf{P}B_{q,p}$.

Let $s \in \mathbf{P}B_{q,p}$; we derive:

$(\lambda_0 \odot IH(\alpha,\beta,\gamma,v)).s.a_i$

$=$

```
if  $i = 0$        $\rightarrow$  $\lambda_0$
[]  $0 < i < v$   $\rightarrow$  $s.a_{i-1} + \gamma\lambda_0$
[]  $v \leqslant i$        $\rightarrow$  $s.a_{i-1} + \gamma\lambda_0$ max $s.b_{i-v} + \beta\lambda_0$
fi
```

```
≤    { $\lambda_0 > 0$ so $v \geqslant p$ , $\lambda_0 \leqslant 1$ , $\lambda_0 \leqslant 2/\gamma$ }
     if  $i = 0$          → $1$
     []  $0 < i < p$      → $s.a_{i-1} + 2$
     []  $p \leqslant i < v$  → $s.a_{i-1} + 2$
     []  $v \leqslant i$      → $s.a_{i-1} + 2 \text{ max } s.b_{i-v} + \beta\lambda_0$
     fi
≤    { $s.b_{i-v} \leqslant s.b_{i-p} - 2(v-p)$ }
     if  $i = 0$          → $1$
     []  $0 < i < p$      → $s.a_{i-1} + 2$
     []  $p \leqslant i$      → $s.a_{i-1} + 2 \text{ max } s.b_{i-p} + \beta\lambda_0 - (2v - 2p)$
     fi
≤    { $\beta\lambda_0 \leqslant 2v - 2p + 1$ }
     $B_{q,p}.s.a_i$
```

The derivation of $(\lambda_0 \odot IH(\alpha,\beta,\gamma,v)).s.b_i \leqslant B_{q,p}.s.b_i$ is similar.

□

5.4 Conservative implementations that meet lower bounds

In Section 5.1 we did already give linear implementations with a linear overhead. In this section we give a hierarchical design of implementations with limited neighbourhood that have a logarithmic overhead. We use the most severe restrictions: a grain size of one, and at most three neighbours per cell. Under the same restrictions we also give implementations with bounded surroundings that have an overhead proportional to the square root of the number of packets. Before giving these implementations we first introduce the constructions we use.

Hierarchical design of buffers

We discuss three ways to enlarge existing FIFO buffers. All extensions (trivially) exhibit a FIFO behaviour, $b(i) = a(i)$; we focus our attention on the real-time behaviour. We analyze the real time behaviour of the constructions in case they are applied to *IH* buffers. It turns out that the behaviour of these constructions can be obtained by counting variables and adding latencies. We do not prove all statements we make about the behaviour of these constructions. The missing proofs, however, can be produced straightforwardly, in the style of the other proofs. At the end we mention better estimations of the behaviour of constructions, by using descriptions that are 'similar' to *IH* buffers.

The first, and most simple, of the constructions is the cascade of a FIFO buffer with a one-place buffer.

(5.30) $C.FIFO \;=\; (B_{1\,b \to c} \parallel FIFO_{a \to c}) \restriction \{a, b\}$

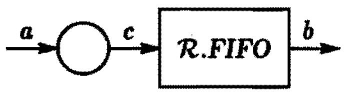


Figure 5.31 Cascade with a one-place buffer: $C.FIFO$.

Counting variables and adding latencies already suggests the following result for application of this construction on IH buffers:

(5.32) $C.IH(\alpha,\beta,2,v) \;\approx\; IH(\alpha+1,\beta+1,2,v+1)$

For IH buffers with delay γ, in general, we mention the following estimation:

(5.33) $C.IH(\alpha,\beta,\gamma,v) \;\lhd\; IH(\alpha+1,\beta+1,\gamma \max 2, v+1)$

Derivation of Formulae 5.32 and 5.33.

The composition $B_{1\,b\to c} \parallel IH(\alpha,\beta,\gamma,v)_{a\to c}$ is given by (see Formulae 5.1 and 5.25):

$$\begin{array}{lcllcl}
e_0.s.a_i & = & \textbf{if} & i=0 & \rightarrow & 1 \\
 & & [\!] & i>0 & \rightarrow & s.c_{i-1}+1 \\
 & & \textbf{fi} & & & \\
e_0.s.c_i & = & \textbf{if} & i=0 & \rightarrow & s.a_i+1 \; \underline{\max\ 1} \\
 & & [\!] & 0<i<v & \rightarrow & s.a_i+1 \max s.c_{i-1}+\gamma \\
 & & [\!] & v \leqslant i & \rightarrow & s.a_i+1 \max s.c_{i-1}+\gamma \max s.b_{i-v}+\beta \\
 & & \textbf{fi} & & & \\
e_0.s.b_i & = & \textbf{if} & i=0 & \rightarrow & s.c_i+\alpha \\
 & & [\!] & 0<i & \rightarrow & s.c_i+\alpha \max s.b_{i-1}+\gamma \\
 & & \textbf{fi} & & &
\end{array}$$

Let $\gamma' = \gamma \max 2$. In case $\gamma = 2$, enabling function e_1, as given below, is equivalent to e_0. This case serves to derive Formula 5.32. In case $\gamma \neq 2$, e_1 is an upper bound of e_0: $e_0 \lhd e_1$. This case serves to derive Formula 5.33.

$$\begin{array}{lcllcl}
e_1.s.a_i & = & \textbf{if} & i=0 & \rightarrow & 1 \\
 & & [\!] & i>0 & \rightarrow & s.c_{i-1}+1 \max s.a_{i-1}+\gamma' \\
 & & \textbf{fi} & & & \\
e_1.s.c_i & = & \textbf{if} & i=0 & \rightarrow & s.a_i+1 \\
 & & [\!] & 0<i<v & \rightarrow & s.a_i+1 \; \underline{\max s.c_{i-1}+\gamma'} \\
 & & [\!] & v \leqslant i & \rightarrow & s.a_i+1 \; \underline{\max s.c_{i-1}+\gamma'} \max s.b_{i-v}+\beta \\
 & & \textbf{fi} & & & \\
e_1.s.b_i & = & \textbf{if} & i=0 & \rightarrow & s.c_i+\alpha \\
 & & [\!] & 0<i & \rightarrow & s.c_i+\alpha \max s.b_{i-1}+\gamma' \\
 & & \textbf{fi} & & &
\end{array}$$

Let e_2 be the result of pruning the redundant (underlined) dependencies in the enabling of c actions. The external behaviour $e_3 = e_2 \Vert\!\upharpoonright \{a,b\}$ is given by:

$$
\begin{array}{lcll}
e_3.s.a_i & = & \textbf{if} \; i=0 & \to 1 \\
 & & [] \; 0 < i \leqslant v & \to s.a_{i-1} + \gamma' \\
 & & [] \; v < i & \to s.a_{i-1} + \gamma' \text{ max } s.b_{i-(v+1)} + \beta + 1 \\
 & & \textbf{fi} & \\
e_3.s.b_i & = & \textbf{if} \; i=0 & \to s.a_i + \alpha + 1 \\
 & & [] \; 0 < i < v & \to s.a_i + \alpha + 1 \text{ max } s.b_{i-1} + \gamma' \\
 & & [] \; v \leqslant i & \to s.a_i + \alpha + 1 \text{ max } s.b_{i-1} + \gamma' \\
 & & & \quad \underline{\text{max } s.b_{i-v} + \alpha + \beta} \\
 & & \textbf{fi} &
\end{array}
$$

The underlined term is redundant because $\gamma v \geqslant \alpha + \beta$ (Formula 5.26) and $\gamma' \geqslant \gamma$. Pruning the redundant term results in $IH(\alpha + 1, \beta + 1, \gamma', v + 1)$.
□

The second construction we discuss consists of placing one-place buffers at both sides of the existing implementation.

(5.34) $\mathcal{B}.FIFO \;=\; (B1_{b \to p.a} \parallel p.FIFO \parallel B1_{a \to p.b}) \Vert\!\upharpoonright \{a, b\}$

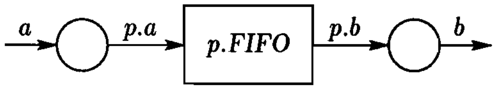


Figure 5.35 *FIFO* between two one-place buffers: $\mathcal{B}.FIFO$.

The result of applying this construction on an *IH* buffer with $\gamma \leqslant 2$ is again an *IH* buffer with the expected latencies, and the expected number of variables:

(5.36) $\gamma \leqslant 2 \;\Rightarrow\; \mathcal{B}.IH(\alpha, \beta, \gamma, v) \approx IH(\alpha + 2, \beta + 2, 2, v + 2)$

The derivation of this behaviour is omitted. Without derivation we also mention the following estimation for the case $\gamma > 2$:

(5.37) $\gamma > 2 \;\Rightarrow\; \mathcal{B}.IH(\alpha, \beta, \gamma, v) \lhd_{\mathrm{l}} IH(\alpha + 2, \beta + 2, \gamma, v + 2)$

So far, the constructions are not very spectacular: the increase of the latencies is proportional to the increase of v. The parallel construction exhibits a better behaviour with respect to these latencies. The parallel construction is similar to the wagging construction in Figure 5.18.

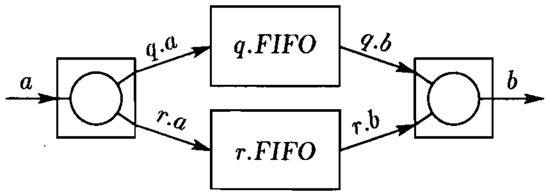


Figure 5.38 The parallel construction: $\mathcal{P}.FIFO$.

The programs for the split and the merge cell are given by:

```
program Split ( input a, output q.a, r.a ) :
var x
begin
   (a?x ; q.a!x ; a?x ; r.a!x )*
end.

program Merge ( input q.b, r.b, output b ) :
var x
begin
   (q.b?x ; b!x ; r.b?x ; b!x )*
end.
```

Let BS and BM be the enabling functions of the split program and the merge program respectively:

(5.39)
$$\begin{array}{lcllcl} BS.s.a_0 &=& 1 & BM.s.qb_0 &=& 1 \\ BS.s.a_{2i+1} &=& s.qa_i+1 & BM.s.rb_i &=& s.b_{2i}+1 \\ BS.s.a_{2i+2} &=& s.ra_i+1 & BM.s.qb_{i+1} &=& s.b_{2i+1}+1 \\ BS.s.qa_i &=& s.a_{2i}+1 & BM.s.b_{2i} &=& s.qb_i+1 \\ BS.s.ra_i &=& s.a_{2i+1}+1 & BM.s.b_{2i+1} &=& s.rb_i+1 \end{array}$$

The parallel construction is given by:

(5.40) $\mathcal{P}.FIFO \;=\; (BS \parallel q.FIFO \parallel r.FIFO \parallel BM) \Vert\!\upharpoonright \{a,b\}$

The behaviour of the parallel construction on $IH(\alpha,\beta,\gamma,v)$ (see Formulae 5.42 and 5.43) can already be guessed by intuition: the number of variables in the composite is $2v+2$; the latencies increase by two; and two buffers in parallel can handle twice as much throughput as one. By renaming of internal actions ($q.a$, $q.b$, $r.a$, and $r.b$), the parallel construction can even be expressed in terms of cascade construction $\mathcal{B}$:

(5.41) $\mathcal{P}.FIFO \;=\; \mathcal{B}.(\mathcal{R}\, x_i \to x_{2i}\,.FIFO \parallel \mathcal{R}\, x_i \to x_{2i+1}\,.FIFO)$

In case $FIFO = IH(\alpha,\beta,\gamma,v)$, the argument of $\mathcal{B}$ can be rewritten into:

$$\begin{array}{lcll} IH'.s.a_i &=& \textbf{if} & i \leqslant 1 \quad\;\;\; \to\; 1 \\ && [\!] & 1<i<2v \;\to\; s.a_{i-2}+\gamma \\ && [\!] & 2v \leqslant i \quad\;\; \to\; s.a_{i-2}+\gamma \text{ max } s.b_{i-2v}+\beta \\ && \textbf{fi} & \\ IH'.s.b_i &=& \textbf{if} & i \leqslant 1 \;\to\; s.a_i+\alpha \\ && [\!] & 1<i \;\to\; s.a_i+\alpha \text{ max } s.b_{i-2}+\gamma \\ && \textbf{fi} & \end{array}$$

With this enabling function we (temporarily) violate the convention for occurrences of generic actions: as far as IH' is concerned, these occurrences may be out of order; for example, a_1 may be scheduled before a_0 . However, the delays in IH' look very much like those in a $IH(\alpha,\beta,\gamma/2,2v)$ buffer, so the following results should not come as a surprise (compare with Formulae 5.36 and 5.37):

(5.42) $\gamma \leqslant 4 \;\Rightarrow\; \mathcal{P}.IH(\alpha,\beta,\gamma,v) \approx IH(\alpha+2,\beta+2,2,2v+2)$

and the estimation for $\gamma > 4$:

(5.43) $\gamma > 4 \;\Rightarrow\; \mathcal{P}.IH(\alpha,\beta,\gamma,v) \lhd_{I} IH(\alpha+2,\beta+2,\gamma/2,2v+2)$

This is a promising result: the latencies increase linearly, the capacity exponentially.

Formulae 5.33, 5.37, and 5.43, give estimations of behaviours only. In fact, these estimations are 'sharp', in that the left-hand sides have the same qualities with respect to specifications of type $B_{q,p}$ as the right-hand sides. Moreover, it is even possible to define a relation $\dot{\approx}$ that satisfies the following

Conditions 5.44

Relation $\dot{\approx}$ is reflexive for IH buffers: $IH(\alpha,\beta,\gamma,v) \;\dot{\approx}\; IH(\alpha,\beta,\gamma,v)$.

Furthermore, for $e \;\dot{\approx}\; IH(\alpha,\beta,\gamma,v)$, the following observations hold:

- $e \lhd_{I} IH(\alpha,\beta,\gamma,v)$
- $Q(e,B_{q,p}) \;=\; Q(IH(\alpha,\beta,\gamma,v),B_{q,p})$ (for all p and q)
- $$\begin{aligned} \mathcal{C}.e &\;\dot{\approx}\; IH(\alpha+1,\beta+1,2\max\gamma,v+1)\\ \mathcal{B}.e &\;\dot{\approx}\; IH(\alpha+2,\beta+2,2\max\gamma,v+2)\\ \mathcal{P}.e &\;\dot{\approx}\; IH(\alpha+2,\beta+2,2\max\gamma/2,2v+2) \end{aligned}$$
- $$\begin{aligned} \mathcal{B}.e &\;\approx\; IH(\alpha+2,\beta+2,2,v+2) && \text{if } \gamma \leqslant 2\\ \mathcal{P}.e &\;\approx\; IH(\alpha+2,\beta+2,2,2v+2) && \text{if } \gamma \leqslant 4 \end{aligned}$$

□

Although intuitively appealing, a formal characterization of relation $\dot{\approx}$ is not trivial. The crux is to consider the throughput over the entire schedule, rather than considering the delays between successive inputs, or outputs. The details are left to the interested reader.

Implementations with limited neighbourhood

The parallel construction is the most important ingredient in achieving implementations that have a logarithmic overhead. By repeating this construction, starting with B_1 , one obtains implementations with grain size one, in which each cell has at most three neighbours. From Formula 5.42 (or by counting variables and adding latencies) we infer that the external behaviours of these implementations are given by:

(5.45) $\mathcal{P}^n.B_1 \;\approx\; IH(2n+1\,,\,3*2^n-2)$

For these implementations, the latencies are logarithmic in the number of variables. From Figure 5.28 we conclude that the number of holes that is needed in order to obtain a quality of one is also logarithmic in the number of variables. To be more precise, $\mathcal{P}^n.B_1$ can be used to implement B_p (with quality one) for $p = v - n$, where v is the number of variables (of $\mathcal{P}^n.B_1$). The overhead n is logarithmic in v as well as in p . In general, when the overhead is logarithmic in v , it is also logarithmic in p .

With sequence $\mathcal{P}^n.B_1$, however, we do not have a complete implementation scheme with logarithmic overhead for all specifications of type B_p. Take for example $p = 3 * 2^n - 1$; the first buffer in the sequence with a non-zero quality with respect to B_p is $\mathcal{P}^{n+1}.B_1$, which has $3 * 2^{n+1} - 2$ variables: this is about twice the number of packets.

In order to show the existence of an implementation scheme with logarithmic overhead, it suffices to give implementations for each number of variables v, such that the latency of these buffers is logarithmic in v. This result can be achieved by using the parallel construction, as well as cascade construction $\mathcal{C}$. We define the following scheme (for $v \geqslant 1$):

(5.46) $$\begin{aligned} Ilog_1 &= B_1 & Ilog_{2v+1} &= \mathcal{C}.Ilog_{2v} \\ Ilog_2 &= \mathcal{C}.Ilog_1 & Ilog_{2v+2} &= \mathcal{P}.Ilog_v \end{aligned}$$

The behaviour of $Ilog_v$ is $IH(\beta, v)$ for a latency β that is logarithmic in v:

(5.47) $$\begin{aligned} \beta \leqslant\ & \textbf{if}\ \ v \text{ is even} \rightarrow 3\log_2 \tfrac{v+4}{6} + 2 \\ & [\!]\ \ v \text{ is odd} \rightarrow 3\log_2 \tfrac{v+3}{6} + 3 \\ & \textbf{fi} \end{aligned}$$

This upper bound is easily verified (by induction).

We have achieved a logarithmic overhead in variables. This result, however, is achieved at the cost of using a number of split cells and a number of merge cells proportional to v. The overhead in costs due to to the wagging of packets in the split and merge cells is thus linear in p.

Implementations with limited surroundings

The idea of these implementations, is to place buffers in parallel only when there is enough room to do so. At any distance from the input (output) the number of variables that is 'used in parallel' is no more than proportional to this distance. We give implementations with a layout that is symmetrical with respect to input and output. In the layout scheme in Figure 5.50, each dot represents a variable and each line represents a channel along which packets can move to the right. When a variable has two outgoing channels, packets alternately take the upper and the lower one. The packet stream is split at distances 2, 2^2, 2^3, etc. from the input. Later on it is merged at the same distances to the output. At that stage, when a variable has two incoming channels, packets are alternately received from the upper and the lower one.

In order to describe these implementations we use cascade construction $\mathcal{B}$ and the parallel construction. As an abbreviation we introduce the combined construction $\mathcal{BP}_m$ that doubles a buffer (in parallel) and adds 2^m variables at the input side and 2^m variables at the output side:

(5.48) $\mathcal{BP}_m.FIFO \ = \ \mathcal{B}^{2^m-1}.\mathcal{P}.FIFO$

In Formulae 5.48 and 5.49 we use the dot for function application in a right associative way.

For any positive integer β there exists a unique pair $n \geqslant 0$ and $0 < k \leqslant 2^{n+1}$ such that $\beta = 2(2^n - 1) + k$. We define implementations $I\!\!\surd_\beta$ in terms of such pairs by:

(5.49) $I\!\!\surd_\beta \;=\; \mathcal{BP}_0 . \mathcal{BP}_1 . \mathcal{BP}_2 \;\ldots\; \mathcal{BP}_{n-2} . \mathcal{BP}_{n-1} . CAS_k$

For $n = 0$, this formula is interpreted as $I\!\!\surd_\beta \;=\; CAS_\beta$.

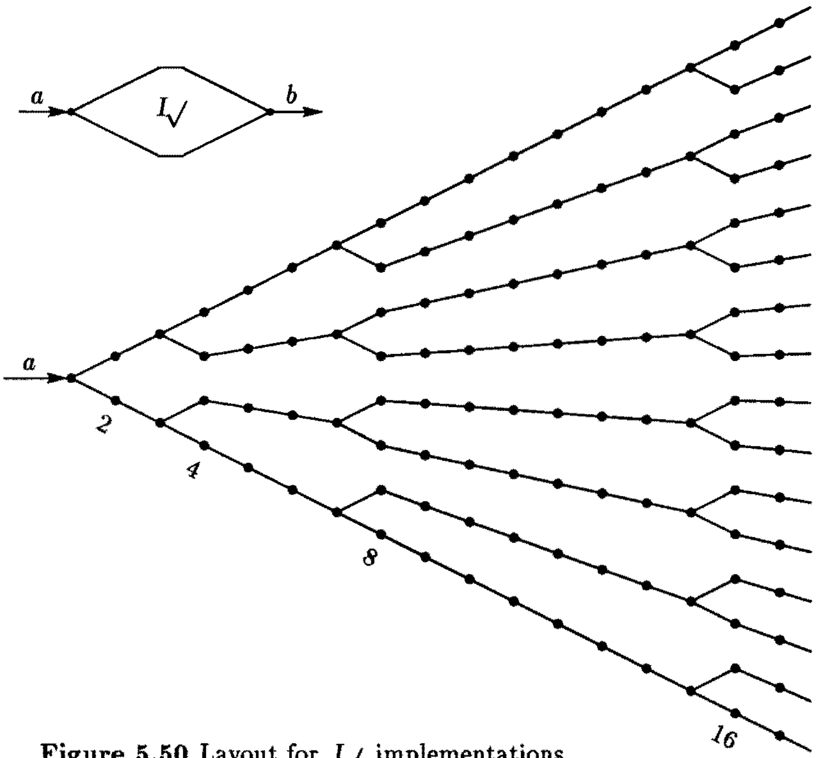


Figure 5.50 Layout for $I\!\!\surd$ implementations.

Counting of variables and adding of latencies gives the following result for the behaviour of $I\!\!\surd_\beta$ buffers:

(5.51) $I\!\!\surd_\beta \;\approx\; IH(\beta,\; 2/3\,(2^{2n} - 1) + 2^n k)$

Let v be the number of variables of $I\!\!\surd_\beta$. From this formula, and from the definition of n and k, we infer that $\beta \leqslant \sqrt{6\,v}$, and that the number of variables of $I\!\!\surd_{\beta+1}$ is no more than $v + \sqrt{3\,v}$. The first observation implies that $I\!\!\surd_\beta$ implementations can be used to implement B_p buffers with an overhead that is proportional to the square root of the number of packets (with a quality that does not tend to zero for large buffers). Due to the second observation, *all* B_p buffers can be implemented in such a way.

An advantage of these implementations, above those that we constructed under limited neighbourhood, is that the number of split and merge cells is proportional to the square root of p. Consequently, the 'wagging overhead' due to splitting and merging of the packet stream is proportional to the square root of p, and not linear in p as in the implementations we constructed under limited neighbourhood.

5.5 Buffers with bypassing

In case one is interested in implementing FIFO buffers in a distributed way, the overhead of variables that is needed is not an insurmountable difficulty: just plug in some more variables. A more serious problem is that all implementations we have given so far, also have a poor performance when they are almost empty: they cannot be used to implement $B_{q,p}$ buffers for which q is less than proportional to the logarithm or (in case of limited surroundings) the square root of p. This is due to the symmetry of these implementations in packets and holes: not only the packets but also the holes are routed under a FIFO strategy.

In this section we introduce implementations that exhibit a better behaviour when they are almost empty. The crux of these implementations is that empty parts of the buffer can be bypassed. As in the previous section, we build buffers in a hierarchical way. Similar implementations are given in [12, 15] and suggested in [7]. In contrast to the implementations in [12, 15], the implementation we give does not synchronize external inputs and outputs. We do not claim this to be the optimal way to do things, but it is a nice example of the use of the enabling model for performance analysis of behaviours with choice. The main differences with the performance analysis of cubic behaviours are:

- When the timing in a device deviates only a little from the analyzed enabling structure, this may result in a dramatic deviation of its behaviour. Therefore, the performance must be analyzed over a whole range of enabling structures, rather than for just one (see Section 3.6, in particular Example 3.56).
- Because a behaviour with bypassing is not conservative, the strategy of divide and conquer must be used very carefully: if e is not conservative, $f \lhd_l g$ does in general *not* imply $e \parallel f \lhd_l e \parallel g$.
- In enabling structures with choice, unexpected dependencies may pop up: dependencies that seem redundant, but —formally— may not be pruned. This phenomenon is already illustrated in Example 2.73.

The bypass extension is similar to cascade construction $\mathcal{B}$ (Figure 5.35), in which one variable is placed before, and one after the existing buffer. In the bypass extension (Figure 5.52), however, there is an additional bypass channel (c). Packets that arrive in an empty buffer take the bypass channel.

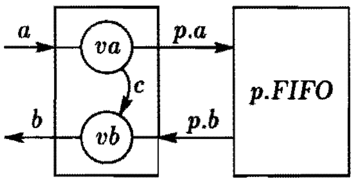


Figure 5.52 *FIFO* with bypass interface: $\Phi_{Byp}.FIFO$.

In contrast to cascade construction $\mathcal{B}$, the added variables are not used independent of each other. Both variables are part of the same program that keeps track of the number of packets in the buffer, in order to decide whether or not to allow a bypass. This behaviour is captured in the (pseudo) program below.

```
program Bypass Buffer ( input a, output b) :
uses p.FIFO
var va, vb ; i,j : index
begin
   a?va , i := 0 ; c!va ;
   ( a?va , i := i + 1 ;
    if     $a_i$ before $b_{i-1}$    →  p.a!va
    ▯   $\neg(a_i$ before $b_{i-1})$  →  c!va
    fi )*
||
   c?vb ; b!vb , j := 0 ;
   ( if     $a_{j+1}$ before $b_j$    →  p.b?vb
    ▯   $\neg(a_{j+1}$ before $b_j)$  →  c?vb
    fi ;
    b!vb , j := j + 1 )*
end.
```

At two places in this program a choice is made. The choice in the first sub-program (between *p.a*! and *c*!) decides whether or not a packet takes the bypass channel. The choice in the second sub-program (between *p.b*? and *c*?) depends on whether the packet has taken the bypass or not. In order to implement the latter choice, it is not necessary to keep track of bypasses and non-bypasses: at any moment at most *one* of the inputs is offered by the environment (of the second sub-program).[4] It suffices to make an environment driven (passive) choice, for example by using a *probe* ([19]). Within the enabling model, however, a description of such a choice is rather baroque. Therefore we use the program 'as is' in the analysis of the behaviour.

We do not discuss efficient ways to implement the program, we just analyze its behaviour. First we introduce enabling structure *BYP* as a reflection of the behaviour of this program under the assumption of delays of one time unit, and under the assumption that the choice between a bypass and no bypass

[4] under the assumption that *FIFO* is a buffer

is performed exactly as prescribed by the program. Later on we adapt this description to devices that do not operate that accurately.

The packet that is received during communication a_i, is bypassed if (and only if) it arrives in an empty buffer. This results in the following definition for a *bypass* at i:

(5.53) $BP(s,i) \;=\; \textbf{if}\;\; i=0 \;\rightarrow\; \textbf{true} \;\;[\!]\;\; i>0 \;\rightarrow\; s.a_i > s.b_{i-1} \;\;\textbf{fi}$

If there is a bypass at i, the index of the performed communication along the internal channel c depends on the number of bypasses that happened before; it is given by:

(5.54) $\mu(s,i) \;=\; |\,(\,\textbf{set}\, j : 0 \leqslant j < i \wedge BP(s,j) : j\,)\,|$

When, on the other hand, there is no bypass at i, the index of the corresponding $p.a$ and $p.b$ communications depends on the number of 'no-bypasses' that happened before; it is given by:

(5.55) $\pi(s,i) \;=\; |\,(\,\textbf{set}\, j : 0 \leqslant j < i \wedge \neg BP(s,j) : j\,)\,|$

In the sequel, when using functions like BP, μ, and π, we tend to leave the schedule implicit.

The enabling of the external actions is defined as follows:

(5.56)
$$\begin{array}{lcl}
BYP.s.a_0 & = & 1 \\
BYP.s.a_{i+1} & = & \textbf{if} \;\; BP.i \;\rightarrow\; s.c_{\mu.i}+1 \\
 & & [\!] \;\; \neg BP.i \;\rightarrow\; s.pa_{\pi.i}+1 \\
 & & \textbf{fi} \;\; \underline{\max\;(\,\textbf{max}\, j : j \leqslant i : s.a_j + 1\,)} \\
BYP.s.b_i & = & \textbf{if} \;\; BP.i \;\rightarrow\; s.c_{\mu.i}+1 \\
 & & [\!] \;\; \neg BP.i \;\rightarrow\; s.pb_{\pi.i}+1 \\
 & & \textbf{fi} \;\; \underline{\max\;(\,\textbf{max}\, j : j < i : s.b_j + 1\,)}
\end{array}$$

The underlined dependencies are of no importance, except that they assure BYP to be an enabling structure.

When defining the enabling of c actions, we need to know the indices of the corresponding a and b actions. Therefore we introduce $\bar{\mu}$ as the 'inverse' of μ. For natural numbers i we define:

(5.57) $\bar{\mu}(s,j)=i \;\Leftrightarrow\; BP(s,i) \wedge \mu(s,i)=j$

However, when there is only a finite number of bypasses, say Nb, such an i does not exist for $j \geqslant Nb$; for these values of j we define $\bar{\mu}(s,j)=\infty$. The enabling of communications over c is given by:

(5.58)
$$\begin{array}{lcl}
BYP.s.c_i & = & \textbf{if} \;\; \bar{\mu}.i = 0 \;\rightarrow\; s.a_{\bar{\mu}.i}+1 \\
 & & [\!] \;\; 0 < \bar{\mu}.i < \infty \;\rightarrow\; s.a_{\bar{\mu}.i}+1 \;\max\; s.b_{\bar{\mu}.i-1}+1 \\
 & & [\!] \;\; \bar{\mu}.i = \infty \;\rightarrow\; \infty \\
 & & \textbf{fi} \;\; \underline{\max\;(\,\textbf{max}\, j : j < \bar{\mu}.i : s.a_j + 1\,)}
\end{array}$$

In order to define the enabling of $p.a$ and $p.b$ actions, we use the inverse $\bar{\pi}$ of π, which satisfies

(5.59) $\bar{\pi}(s,j) = i \;\Leftrightarrow\; \neg BP(s,i) \wedge \pi(s,i) = j$
and $\bar{\pi}(s,j) = \infty$ if no such i exists.

(5.60)
$$\begin{array}{lcll} BYP.s.pa_i & = & \textbf{if} & \bar{\pi}.i < \infty \;\rightarrow\; s.a_{\bar{\pi}.i} + 1 \\ & & [\!] & \bar{\pi}.i = \infty \;\rightarrow\; \infty \\ & & \textbf{fi} & \underline{\max\;(\,\mathbf{max}\,j : j < \bar{\pi}.i : s.a_j + 1\,)} \\ BYP.s.pb_i & = & \textbf{if} & \bar{\pi}.i < \infty \;\rightarrow\; s.b_{\bar{\pi}.i-1} + 1 \\ & & [\!] & \bar{\pi}.i = \infty \;\rightarrow\; \infty \\ & & \textbf{fi} & \underline{\max\;(\,\mathbf{max}\,j : j < \bar{\pi}.i : s.b_j + 1\,)} \end{array}$$

In the sequel we forget about the (semi) redundant underlined dependencies in Formulae 5.56, 5.58, and 5.60.

Enabling structure BYP gives the behaviour of the program under the assumption of delays of exactly one time unit, and under assumption that the choice between taking a bypass or not, is implemented with absolute precision. When the delays vary, and the choice is not so precise, this results in a deviating behaviour. For example, when communication a_i is performed a little after communication b_{i-1}, the arriving packet should be bypassed along channel c to be offered back to the environment after a delay of two time units. However, if the device, that is used to implement the program, decides that a_i and b_{i-1} occurred simultaneously, the packet misses the bypass within an ace, and is routed via $p.FIFO$, which may take considerably longer. In order to describe a device that operates within tolerable deviations δ and ε, both at least zero and less than one, we make the following assumption about bypasses: $BP(s,0) = \mathbf{true}$ and for $i > 0$:

(5.61) $BP(s,i) \;\Rightarrow\; s.a_i \geqslant s.b_{i-1} - \delta \qquad \neg BP(s,i) \;\Rightarrow\; s.a_i \leqslant s.b_{i-1} + \varepsilon$

Let BYP_{BP} be the enabling structure that corresponds to a bypass function BP that satisfies these conditions. The unit delays in this enabling structure are assumed to be upper bounds for the delays in the device; consequently the device can be described by an enabling structure Byp that satisfies $Byp \leqslant BYP_{BP}$ (for some bypass function BP).

The composition of a head cell with behaviour Byp with a FIFO buffer $FIFO$ is described by:

(5.62) $\Phi_{Byp}.FIFO \;=\; (Byp \parallel p.FIFO) \upharpoonright \{a,b\}$

We intend to analyze the behaviour of this bypass construction when it is applied to IH buffers. More precisely, when $FIFO \lhd\!\!\shortmid\, IH(\alpha,\beta,\gamma,v)$, we want to know which IH buffers are implemented by $\Phi_{Byp}.FIFO$, for all possible behaviours Byp. These are the solutions $IH(\alpha',\beta',\gamma',v')$ of the following equation:

(5.63) $(\,\forall\, BP, Byp, FIFO : Byp \leqslant BYP_{BP} \wedge FIFO \lhd\!\!\shortmid\, IH(\alpha,\beta,\gamma,v)$
$$: \Phi_{Byp}.FIFO \lhd\!\!\shortmid\, IH(\alpha',\beta',\gamma',v')\,)$$

We are in particular interested in upper bounds with capacity $v' = v + 2$, that have a small latency α'.

In order to avoid the computation of $\lhd\!\shortmid$ in the quantification over *FIFO*, we rewrite Formula 5.63 into:

$$(5.64)\quad (\forall\, BP, Byp, FIFO : Byp \leqslant BYP_{BP} \wedge FIFO \leqslant \mathcal{N}.IH(\alpha,\beta,\gamma,v)$$
$$: \Phi_{Byp}.FIFO \lhd\!\shortmid IH(\alpha',\beta',\gamma',v'))$$

The following lemma implies that Formulae 5.63 and 5.64 are equivalent.

Lemma

For $f \in [\,Con\,]$: $\quad e \lhd\!\shortmid f \;\Leftrightarrow\; (\exists g : g \approx e : g \leqslant \mathcal{N}.f)$

Proof

Due to Proposition 3.42, it suffices to prove for $f \in [\,Asc\,]$:

$e \lhd f \;\Leftrightarrow\; (\exists g : g \approx e : g \leqslant \mathcal{N}.f)$

The $\Rightarrow$ implication follows from Proposition 3.41.3 (let $g = \mathcal{N}.e$). The $\Leftarrow$ implication follows from Proposition 3.41.2.

□

On the following pages, we derive an upper bound for the behaviour of the extended FIFO buffer, $\Phi_{Byp}.FIFO$. This is a tedious exercise, but the enabling model allows us to give a formal proof of this upper bound. Before doing so, we give an intuitive way to obtain an upper bound $IH(\alpha',\beta',\gamma',v')$.

Forget about tolerances δ and ε and assume $FIFO = IH(\alpha,\beta,\gamma,v)$. As mentioned before, the bypass extension is similar to cascade extension $\mathcal{B}$ (see Figure 5.35). In case the buffer is not (almost) empty, the behaviour is even exactly the same as the behaviour of this cascade extension. Therefore it is reasonable to conclude $\beta' = \beta + 2$, and $v = v + 2$ (see Formulae 5.36 and 5.37). Furthermore, it is obvious that $\gamma' \geqslant \gamma \max 2$. Whether or not we may conclude equality, depends on the behaviour when the buffer is almost empty.

When a packet arrives in an empty buffer, it takes the bypass and it is available as an output after two time units. This results in the restriction $\alpha' \geqslant 2$. When, on the other hand, a packet arrives in a non-empty buffer, it is routed via *p.FIFO*. At first glance, one may conclude that nothing is gained with respect to the $\mathcal{B}$ construction, and that the latency is $\alpha' = \alpha + 2$, which is by no means the result we are striving for. However, assume there is no bypass at i, that is $s.a_i \leqslant s.b_{i-1}$, then also $s.a_i + 2 + \alpha \leqslant s.b_{i-1} + 2 + \alpha$. So, when accepting a throughput delay of (at least) $2 + \alpha$, the dependence of b_i on a_i is redundant. A valid upper bound of the behaviour is thus given by $IH(2, \beta + 2, \gamma \max 2 + \alpha, v + 2)$. This is exactly the result we give in the second part of Formula 5.82.

Back to the formal derivation again. First we hide the internal channel c of the bypass cell. This results in the following upper bound for the external behaviour $byp = Byp \Vert\!\upharpoonright \{a, b, p.a, p.b\}$.

$$
\begin{array}{llll}
(5.65) & byp.s.a_0 & \leqslant & 1 \\
& byp.s.a_{i+1} & \leqslant & \textbf{if} \quad BP.i \rightarrow s.a_i + 2 + \delta \\
& & & [\!]\ \neg BP.i \rightarrow s.pa_{\pi.i} + 1 \\
& & & \textbf{fi} \\
& byp.s.b_i & \leqslant & \textbf{if} \quad BP.i \wedge i = 0 \rightarrow s.a_i + 2 \\
& & & [\!]\ BP.i \wedge i > 0 \rightarrow s.a_i + 2 \ \max\ s.b_{i-1} + 2 \\
& & & [\!]\ \neg BP.i \rightarrow s.pb_{\pi.i} + 1 \\
& & & \textbf{fi} \\
& byp.s.pa_i & \leqslant & \textbf{if}\ \bar{\pi}.i < \infty \rightarrow s.a_{\bar{\pi}.i} + 1 \ [\!]\ \bar{\pi}.i = \infty \rightarrow \infty \ \textbf{fi} \\
& byp.s.pb_i & \leqslant & \textbf{if}\ \bar{\pi}.i < \infty \rightarrow s.b_{\bar{\pi}.i-1} + 1 \ [\!]\ \bar{\pi}.i = \infty \rightarrow \infty \ \textbf{fi}
\end{array}
$$

In the sequel we consider the enabling of $p.a_i$ and $p.b_i$ for $\bar{\pi}.i < \infty$ only. Furthermore, we don't bother about the enabling of a_0 any more.

The upper bound for the enabling of a_{i+1} in case of a bypass at i is derived below; for b_i the derivation is even simpler.

Derivation of first alternative for a_{i+1} in Formula 5.65.

$$
\begin{array}{ll}
& byp.s.a_{i+1} \\
\leqslant & \{ BP.i \text{, Formulae 5.56 and 5.58} \} \\
& \textbf{if}\ \bar{\mu}\,\mu i = 0 \rightarrow s.a_{\bar{\mu}\,\mu i} + 1 + 1 \\
& [\!]\ 0 < \bar{\mu}\,\mu i < \infty \rightarrow s.a_{\bar{\mu}\,\mu i} + 1 + 1 \ \max\ s.b_{\bar{\mu}\,\mu i - 1} + 1 + 1 \\
& [\!]\ \bar{\mu}\mu i = \infty \rightarrow \infty + 1 \\
& \textbf{fi} \\
= & \{ \bar{\mu} \text{ is the inverse of } \mu \text{, Formula 5.57} \} \\
& \textbf{if}\ i = 0 \rightarrow s.a_i + 2 \ [\!]\ i > 0 \rightarrow s.a_i + 2 \ \max\ s.b_{i-1} + 2 \ \textbf{fi} \\
\leqslant & \{ BP.i \text{, Formula 5.61} \} \\
& s.a_i + 2 + \delta
\end{array}
$$

□

The enabling of a actions and b actions in case of a bypass imposes the following restrictions upon α' and γ':

(5.66) $\alpha' \geqslant 2 \qquad \gamma' \geqslant 2 + \delta$

From Formula 5.65 we learn that the enabling of a_{i+1} and b_i in case of a bypass at i does not depend on the scheduling of '$p.$' actions. Remains to analyze the enabling of a_{i+1} and b_i in case there is *no* bypass at i.

In Formula 5.64, *FIFO* has as an upper bound the normal form of an *IH* buffer. The normal form of *IH* buffers is given by:

$$
\begin{array}{llll}
(5.67) & \mathcal{N}.IH(\alpha,\beta,\gamma,v).s.a_i & = & \textbf{max}\ \ i\gamma + 1 \\
& & & (\,\textbf{max}\, j : j < i : s.a_j + (i-j)\gamma\,) \\
& & & (\,\textbf{max}\, j : j \leqslant i - v : s.b_j + (i - v - j)\gamma + \beta\,) \\
& \mathcal{N}.IH(\alpha,\beta,\gamma,v).s.b_i & = & \textbf{max}\ \ i\gamma + 1 + \alpha \\
& & & (\,\textbf{max}\, j : j \leqslant i : s.a_j + (i-j)\gamma + \alpha\,) \\
& & & (\,\textbf{max}\, j : j < i : s.b_j + (i-j)\gamma\,)
\end{array}
$$

where **max** is used as a prefix operator that takes the maximum of the expressions on subsequent lines.

The reader is invited to verify this result.

Now that we have the upper bound $\mathcal{N}.IH(\alpha,\beta,\gamma,v)$ for *FIFO*, we can derive an upper bound $f \geqslant \Phi_{Byp}.FIFO$. For a_0, and in case of a bypass at i, we already have an upper bound (Formula 5.65):

$$(5.68)\quad \begin{array}{lcl} f.s.a_0 &=& 1 \qquad\qquad f.s.b_i \;=\; s.a_i + 2 \;\; \mathbf{max} \;\; s.b_{i-1} + 2 \\ f.s.a_{i+1} &=& s.a_i + 2 + \delta \end{array}$$

In case of no bypass at i, an upper bound is given by:

$$(5.69)\quad \begin{array}{lcl} f.s.a_{i+1} &=& \mathbf{max} \quad \pi.i * \gamma + 2 \\ && (\,\mathbf{max}\, j : j \leqslant \pi.i : s.a_{\pi.j} + 2 + (\pi.i - j)\gamma\,) \\ && (\,\mathbf{max}\, j : j \leqslant \pi.i - v : s.b_{\pi.j-1} + (\pi.i - v - j)\gamma + \beta + 2\,) \\ f.s.b_i &=& \mathbf{max} \quad \pi.i * \gamma + 2 + \alpha \\ && (\,\mathbf{max}\, j : j \leqslant \pi.i : s.a_{\pi.j} + 2 + (\pi.i - j)\gamma + \alpha\,) \\ && (\,\mathbf{max}\, j : j \leqslant \pi.i : s.b_{\pi.j-1} + 2 + (\pi.i - j)\gamma\,) \end{array}$$

Derivation of Formula 5.69.

From Formulae 5.65 and 5.67 and from $FIFO \leqslant \mathcal{N}.IH(\alpha,\beta,\gamma,v)$ we infer for $e_0 = byp \parallel p.FIFO$:

$$e_0.s.a_{i+1} \;\leqslant\; s.pa_{\pi.i} + 1 \quad\text{and}\quad e_0.s.b_i \;\leqslant\; s.pb_{\pi.i} + 1$$

in case of no bypass at i, and

$$\begin{array}{lcl} e_0.s.pa_i &\leqslant& \mathbf{max} \quad i\gamma + 1 \;\; \mathbf{max} \;\; s.a_{\pi.i} + 1 \\ && (\,\mathbf{max}\, j : j < i : s.pa_j + (i-j)\gamma\,) \\ && (\,\mathbf{max}\, j : j \leqslant i - v : s.pb_j + (i - v - j)\gamma + \beta\,) \\ e_0.s.pb_i &\leqslant& \mathbf{max} \quad i\gamma + 1 + \alpha \;\; \mathbf{max} \;\; s.b_{\pi.i-1} + 1 \\ && (\,\mathbf{max}\, j : j \leqslant i : s.pa_j + (i-j)\gamma + \alpha\,) \\ && (\,\mathbf{max}\, j : j < i : s.pb_j + (i-j)\gamma\,) \end{array}$$

We are interested in the behaviour of $e_0 \mathbin{\|\!\upharpoonright} \{a,b\}$. First we hide $p.a$. If $p.a$ actions happen as soon as they are enabled, this results in the following upper bound for $s.pa_i$:

$$\begin{array}{ll} & s.pa_i \\ \leqslant & \\ & \mathbf{max} \quad i\gamma + 1 \\ & s.a_{\pi.i} + 1 \\ & (\,\mathbf{max}\, j : j < i : s.pa_j + (i-j)\gamma\,) \\ & (\,\mathbf{max}\, j : j \leqslant i - v : s.pb_j + (i - v - j)\gamma + \beta\,) \\ \leqslant & \quad \{\text{unfold dependencies of } pa_j \text{ on } a_{\pi.j}\,\} \\ & \mathbf{max} \quad i\gamma + 1 \\ & (\,\mathbf{max}\, j : j \leqslant i : s.a_{\pi.j} + 1 + (i-j)\gamma\,) \\ & (\,\mathbf{max}\, j : j < i : s.pa_j + (i-j)\gamma\,) \\ & (\,\mathbf{max}\, j : j \leqslant i - v : s.pb_j + (i - v - j)\gamma + \beta\,) \end{array}$$

$$\begin{array}{ll}
= & \{\,\text{induction}\,\} \\
 & \mathbf{max}\ \ i\gamma + 1 \\
 & (\,\mathbf{max}\, j : j \leqslant i : s.a_{\bar\pi.j} + 1 + (i-j)\gamma\,) \\
 & (\,\mathbf{max}\, j : j \leqslant i - v : s.pb_j + (i-v-j)\gamma + \beta\,)
\end{array}$$

Let $e_1 = e_0 \mathbin{|\!\!\upharpoonleft} \{a, b, pb\}$. Using the derived upper bound, we conclude:

$$\begin{array}{lll}
e_1.s.a_{i+1} & \leqslant & \mathbf{max}\ \ \pi.i * \gamma + 2 \\
 & & (\,\mathbf{max}\, j : j \leqslant \pi.i : s.a_{\bar\pi.j} + 2 + (\pi.i - j)\gamma\,) \\
 & & (\,\mathbf{max}\, j : j \leqslant \pi.i - v : s.pb_j + (\pi.i - v - j)\gamma + \beta + 1\,) \\
e_1.s.b_i & \leqslant & s.pb_{\pi.i} + 1
\end{array}$$

For the enabling of $p.b_i$ we derive:

$$\begin{array}{ll}
 & e_1.s.pb_i \\
\leqslant & \{\,\text{upper bound for } s.pa_j\,\} \\
 & \mathbf{max}\ \ i\gamma + 1 + \alpha\ \ \mathrm{max}\ \ s.b_{\bar\pi.i-1} + 1 \\
 & (\,\mathbf{max}\, j \ \ : j \leqslant i \qquad\qquad\quad : j\gamma + 1 \qquad\qquad\qquad + (i-j)\gamma + \alpha\,) \\
 & (\,\mathbf{max}\, j,k : j \leqslant i \wedge k \leqslant j \qquad : s.a_{\bar\pi.k} + 1 + (j-k)\gamma \qquad + (i-j)\gamma + \alpha\,) \\
 & (\,\mathbf{max}\, j,k : j \leqslant i \wedge k \leqslant j - v : s.pb_k + (j-v-k)\gamma + \beta + (i-j)\gamma + \alpha\,) \\
 & (\,\mathbf{max}\, j : j < i : s.pb_j + (i-j)\gamma\,) \\
= & \\
 & \mathbf{max}\ \ i\gamma + 1 + \alpha\ \ \mathrm{max}\ \ s.b_{\bar\pi.i-1} + 1 \\
 & (\,\mathbf{max}\, j : j \leqslant i : i\gamma + 1 + \alpha\,) \\
 & (\,\mathbf{max}\, k : k \leqslant i : s.a_{\bar\pi.k} + 1 + (i-k)\gamma + \alpha\,) \\
 & (\,\mathbf{max}\, k : k \leqslant i - v : s.pb_k + (i-k)\gamma - v\gamma + \alpha + \beta\,) \\
 & (\,\mathbf{max}\, j : j < i : s.pb_j + (i-j)\gamma\,) \\
= & \{\,\alpha + \beta \leqslant v\gamma\,\} \\
 & \mathbf{max}\ \ i\gamma + 1 + \alpha\ \ \mathrm{max}\ \ s.b_{\bar\pi.i-1} + 1 \\
 & (\,\mathbf{max}\, j : j \leqslant i : s.a_{\bar\pi.j} + 1 + (i-j)\gamma + \alpha\,) \\
 & (\,\mathbf{max}\, j : j < i : s.pb_j + (i-j)\gamma\,)
\end{array}$$

Using this upper bound for e_1, it can be established that f is an upper bound of $e_1 \mathbin{|\!\!\upharpoonleft} \{a, b\} = e_0 \mathbin{|\!\!\upharpoonleft} \{a, b\}$. This derivation similar to the derivation above.

□

Remains to find an *IH* buffer that is an upper bound for f. The idea behind bypassing is that we strive for an upper bound with a small latency, α', between input and output.

In order to compare f with an *IH* buffer we have to rule out the use of π and $\bar\pi$ in Formula 5.69. We use the following implications in order to do so.

(5.70) $\neg BP.i \ \Rightarrow\ \pi.i < i \qquad j \leqslant i \ \Rightarrow\ \pi.i - \pi.j < i - j$

In case of no bypass at i, we infer the following upper bounds for the enabling of a_{i+1} and b_i:

$$(5.71)\quad f.s.a_{i+1} \leqslant \max \quad (i+1)*(\gamma \max \tfrac{1}{2})+1 \\ (\max j : j < i+1 : s.a_j + (i+1-j)(\gamma \max 2)) \\ (\max j : j \leqslant i+1-(v+2) \\ : s.b_j + (i+1-(v+2)-j)\gamma + \beta + 2)$$

$$(5.72)\quad f.s.b_i \leqslant \max \quad i(\gamma \max 1)+1+\alpha \\ (\max j : j \leqslant i : s.a_j + (i-j)\gamma + \alpha + 2) \\ (\max j : j < i : s.b_j + (i-j)*(\gamma \max 2))$$

We give a derivation of the upper bound for $f.s.a_{i+1}$. The derivation for $f.s.b_i$ is similar.

Derivation of Formula 5.71.

$$f.s.a_{i+1}$$

$$= \quad \{\text{Formula 5.69, dummy transformation } j := \bar{\pi}.j\}$$

$$\max \quad \pi.i*\gamma+2 \\ (\max j : j \leqslant i \wedge \neg BP.j : s.a_j + 2 + (\pi.i - \pi.j)\gamma) \\ (\max j : \pi.j \leqslant \pi.i - v \wedge \neg BP.j : s.b_{j-1} + (\pi.i - v - \pi.j)\gamma + \beta + 2)$$

$$\leqslant \quad \{\text{Formula 5.70}\}$$

$$\max \quad (i-1)\gamma+2 \\ (\max j : j \leqslant i \wedge \neg BP.j : s.a_j + 2 + (i-j)\gamma) \\ (\max j : j \leqslant i - v \wedge \neg BP.j : s.b_{j-1} + (i-v-j)\gamma + \beta + 2)$$

$$\leqslant \quad \{\text{extension of range for } j \text{, dummy transformation}\}$$

$$\max \quad (i-1)\gamma+2 \\ (\max j : j \leqslant i : s.a_j + 2 + (i-j)\gamma) \\ (\max j : j \leqslant i-v-1 : s.b_j + (i-v-1-j)\gamma + \beta + 2)$$

$$\leqslant$$

$$\max \quad (i+1)*(\gamma \max \tfrac{1}{2})+1 \\ (\max j : j \leqslant i : s.a_j + (i+1-j)*(\gamma \max 2)) \\ (\max j : j \leqslant (i+1)-(v+2) : s.b_j + (i+1-(v+2)-j)\gamma + \beta + 2)$$

□

From Formula 5.71, and under the assumption $v' = v+2$, we conclude the following restrictions:

$$(5.73)\quad \gamma' \geqslant \gamma \max 2 \qquad \beta' \geqslant \beta + 2$$

From Formula 5.72, we conclude the following additional lower bound:

$$(5.74)\quad \alpha' \geqslant \alpha + 2$$

Together with the results in case of a bypass (Formula 5.66), this gives the following result for the composition Φ_{Byp}:

$$(5.75)\quad FIFO \lhd\!\!\shortmid IH(\alpha,\beta,\gamma,v) \Rightarrow \\ \Phi_{Byp}.FIFO \lhd\!\!\shortmid IH(\alpha+2,\ \beta+2,\ \gamma \max 2 + \delta,\ v+2)$$

This is a disappointing result, the upper bound for the behaviour is no better than the result for cascade construction $\mathcal{B}$: the delay between input and output increases by two! (see Formulae 5.36 and 5.37). It is certainly not the result we want to achieve with bypassing. In order to improve the upper bound for the delay between input and output we reconsider the estimation of $f.s.b_i$. The idea is that (in case of no bypass) the dependence on a actions can be hidden by assuming longer delays between subsequent b actions.

(5.76) $$f.s.b_i \leqslant \ \mathbf{max} \ \ i(\gamma \max 2+\alpha+\varepsilon)+2+1 \\ (\ \mathbf{max}\, j : j < i : s.b_j + (i-j)(\gamma \max 2+\alpha+\varepsilon)\,)$$

Derivation of Formula 5.76.

$$\begin{array}{ll}
 & f.s.b_i \\
= & \quad \{\text{Formula 5.69}\} \\
 & \mathbf{max} \ \ \pi.i*\gamma+2+\alpha \\
 & (\ \mathbf{max}\, j : j \leqslant \pi.i : s.a_{\bar{\pi}.j}+2+(\pi.i-j)\gamma+\alpha\,) \\
 & (\ \mathbf{max}\, j : j \leqslant \pi.i : s.b_{\bar{\pi}.j-1}+2+(\pi.i-j)\gamma\,) \\
\leqslant & \quad \{\ \neg BP(\bar{\pi}.j)\,,\ \text{Formula 5.61}\} \\
 & \mathbf{max} \ \ \pi.i*\gamma+2+\alpha \\
 & (\ \mathbf{max}\, j : j \leqslant \pi.i : s.b_{\bar{\pi}.j-1}+2+(\pi.i-j)\gamma+\alpha+\varepsilon\,) \\
= & \quad \{\text{dummy transformation } j := \bar{\pi}.j-1\ \} \\
 & \mathbf{max} \ \ \pi.i*\gamma+2+\alpha \\
 & (\ \mathbf{max}\, j : j < i \wedge \neg BP(j+1) : s.b_j+2+(\pi.i-\pi.(j+1))\gamma+\alpha+\varepsilon\,) \\
\leqslant & \quad \{\text{Formula 5.70, extension of range for } j\ \} \\
 & \mathbf{max} \ \ (i-1)\gamma+2+\alpha \\
 & (\ \mathbf{max}\, j : j < i : s.b_j+(i-j-1)\gamma+2+\alpha+\varepsilon\,) \\
\leqslant & \\
 & \mathbf{max} \ \ i(\gamma \max 2+\alpha+\varepsilon)+2+1 \\
 & (\ \mathbf{max}\, j : j < i : s.b_j+(i-j)(\gamma \max 2+\alpha+\varepsilon))
\end{array}$$

□

From Formula 5.76 we infer the following restrictions, in addition on Formulae 5.66 and 5.73:

(5.77) $\alpha' \geqslant 2 \qquad \gamma' \geqslant 2+\alpha+\varepsilon$

This results in the following upper bound for Φ_{Byp}:

(5.78) $$FIFO \lhd\!\!\shortmid\ IH(\alpha,\beta,\gamma,v) \Rightarrow \\ \Phi_{Byp}.FIFO \lhd\!\!\shortmid\ IH(2,\beta+2,\gamma \max 2+\alpha+\varepsilon, v+2)$$

At the cost of a —relatively— high throughput delay, we have achieved an estimation with a latency between input and output of two time units.

The bypass construction can be used to define systolic arrays of bypass cells, similar as in Chapter 4. In order to obtain implementations for any number of variables, we use two different tails: B_1 for odd v and CAS_2 for even v. Since bypass cells are not conservative, we have to consider the behaviour of tails that implement B_1 and CAS_2 also. The behaviour of a systolic array with v variables is given by $Ibyp_v$ as follows:

(5.79) $Ibyp_1 \lhd\!\!\shortmid\ B_1$, $Ibyp_2 \lhd\!\!\shortmid\ CAS_2$, and $Ibyp_{v+2} = \Phi_{Byp.v}.Ibyp_v$, for bypass cells $Byp.v$ that operate within tolerances δ and ε. A schematic representation of an $Ibyp_v$ array is given in Figure 5.80.

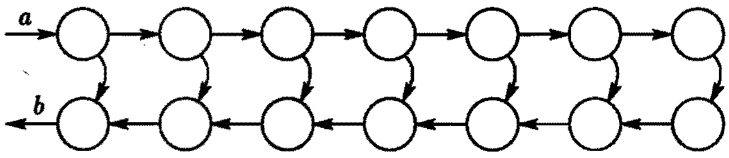


Figure 5.80 A schematic representation of $Ibyp_{2*7}$.

From Formulae 5.75 and 5.78, we infer the following upper bounds:

(5.81) $Ibyp_v \lhd\!\!\shortmid\ IH(v,\ v,\ 2+\delta,\ v) \qquad Ibyp_v \lhd\!\!\shortmid\ IH(2,\ v,\ 4+\varepsilon,\ v)$

An important result, for both upper bounds, is that apparently the decision whether to perform a bypass or not, is not critical: when the bypass cells operate within tolerances δ and ε, the influence on the upper bounds is no more than proportional to δ and ε. For $\delta = \varepsilon = 0$ we obtain:

(5.82) $FIFO \lhd\!\!\shortmid\ IH(\alpha,\beta,\gamma,v) \Rightarrow$

$$\Phi_{Byp}.FIFO \lhd\!\!\shortmid\ IH(\alpha+2,\ \beta+2,\ \gamma \max 2,\ v+2) \ \wedge$$
$$\Phi_{Byp}.FIFO \lhd\!\!\shortmid\ IH(2,\ \beta+2,\ \gamma \max 2+\alpha,\ v+2)$$

and for the systolic arrays:

(5.83) $Ibyp_v \lhd\!\!\shortmid\ CAS_v \qquad Ibyp_v \lhd\!\!\shortmid\ IH(2,v,4,v)$

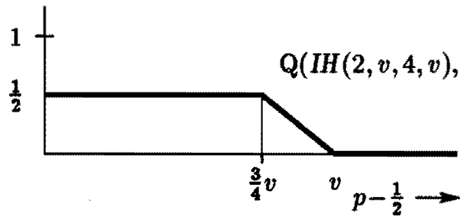


Figure 5.84 A lower bound for $Q(Ibyp_v, B_p)$.

The quality of CAS_v with respect to specifications of type B_p is given in Figure 5.10; the quality of $IH(2,v,4,v)$ is given in Figure 5.84. Both qualities are combined into lower bound $Q_L(Byp_v, B_p)$ for the quality of systolic bypass buffers. This lower bound is given in Figure 5.86. The quality of systolic bypass buffers with respect to specifications of type $B_{q,p}$ has lower bound:

(5.85) $Q(Ibyp_v, B_{q,p}) \geqslant Q_L(Ibyp_v, B_q) \min Q_L(Ibyp_v, B_p)$

So, under the assumption of unit delays, each specification of type $B_{q,p}$ can be implemented with a quality of at least a half. When leaving the assumption of unit delays, there still is a (uniform) positive lower bound for the achievable quality with which specifications of type $B_{q,p}$ can be implemented.

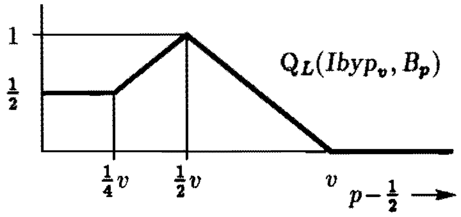


Figure 5.86 A hybrid lower bound for $Q(Ibyp_v, B_p)$.

The main drawback of the bypass scheme is that, in hardware, the implementations of bypass cells are expensive and slow with respect to implementations of B_1, and also with respect to implementations of split and merge cells. When one is satisfied with slower implementations, in particular with a slower throughput, it suffices to use fewer bypass cells. A throughput delay of $2n+2$ can be achieved with linear arrays in which the number of bypasses is reduced by a factor n with respect to *Ibyp* buffers (under the assumption $\delta = \varepsilon = 0$). Consider for example the array in Figure 5.87, in which the number of bypasses is reduced by a factor two. This array implements *IH*(2,14,6,14), whereas array $Ibyp_{14}$ of Figure 5.80 implements *IH*(2,14,4,14).

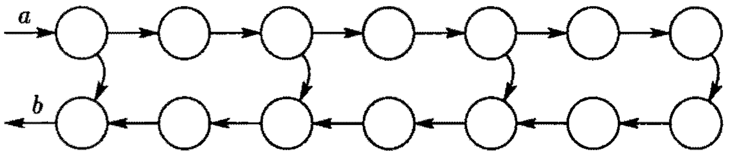


Figure 5.87 An implementation with a reduced number of bypasses.

However, after such a reduction, the number of bypasses is still linear in the number of variables. From Formula 5.43 we learn that the parallel construction can be used to compensate the increase of the throughput delay: it reduces the throughput delay by a factor two. Using this property, it turns out that under limited neighbourhood a logarithmic (in v) number of bypasses suffices; and that under limited surroundings it suffices to use a number of bypasses proportional to the square root of v. The implementations that meet these bounds are similar to the *Ilog*, and $I_{\surd}$ implementations in the previous section (Formulae 5.46 and 5.49). It is left as a challenge to the reader to construct such implementations.

5.6 Buf-like behaviour as a side-effect

The most simple form of a 'buf-like' behaviour, is the behaviour of a one-place buffer. This behaviour is very common for systolic arrays that communicate to the environment via one cell. For example, the solutions of the 'max' problem in Section 4.3. A more general buf-like behaviour is exhibited by systolic arrays with the input at one side, and the output at the opposite side. In such arrays, the latency between input and output is (at least) proportional to the length of the array (the distance between input and output). In general, concurrent programs can be used to decrease the latencies and the throughput delay (with respect to sequential programs). It is clear that arrays with input and output at opposite sides are not particularly suitable to obtain small latencies. In this section we informally discuss three (general) examples implementations that are especially designed to obtain small throughput delays. The first, 'parallel computation' is a blunt approach in which sequential programs are placed in parallel, in a similar way as we did for FIFO buffers with limited neighbourhood (Figure 5.38, Formula 5.45). The second, 'pipeline computation', consists of dividing a computation over a number of cells that are placed in cascade, similar to the cascade buffers (as given in Section 5.1). Finally, we discuss linear systolic arrays that communicate to the environment at both sides.

Parallel computation

Consider the data-specification $b(i) = C.a(i)$, for some function C. A straightforward implementation of this specification is given by

```
program COMP( input a, output b) :
var x ;
begin
   ( a?x ; b!C.x )*
end.
```

For C the identity, this would yield a one-place buffer. We assume, however, that C is a complicated function that takes κ time units (relative to the duration of a communication, which is normalized at 1), for some $\kappa >> 1$. The behaviour of the program is given by:

(5.88) $Comp \;=\; IH(1+\kappa, 1, 2+\kappa, 1)$

The duration of a computation, κ, does not only result in a large latency between input and output ($1+\kappa$), but also in a large throughput delay ($2+\kappa$). Without using specific properties of function C, it is not possible to reduce the latency. By placing programs in parallel, however, it is possible to reduce the throughput delay. We exhibit such a parallel construction under the condition of limited neighbourhood.

Assume, for the sake of simplicity, that $\kappa + 2 = 2^{n+1}$ for some natural n. Applying the parallel construction (Figure 5.38) n times results in the following behaviour (see Conditions 5.44, compare to Formula 5.45):

(5.89) $\mathcal{P}^n.Comp \;\approx\; IH(2n+\kappa+1,\ 2n+1,\ 2,\ 3*2^n-2)$

The right-hand side of this formula is roughly equal to

$IH(\kappa + 2\log\kappa,\ 2\log\kappa,\ 2,\ 3/2\,\kappa)$

Figure 5.90 gives the speed of this construction, relative to B_p buffers. This result is achieved at the cost of approximately $3/2\,\kappa$ variables, and $\kappa/2$ (expensive) function blocks for program *COMP*. In addition to these costs, there is also the overhead (of control) in the implementation of the split and merge cells that are used in the parallel construction $\mathcal{P}$.

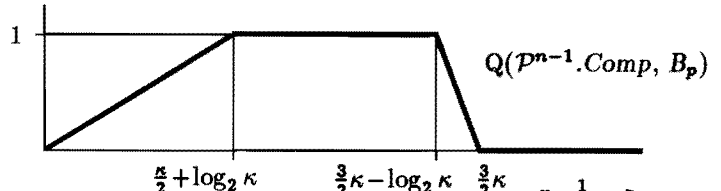


Figure 5.90 Approximate performance of parallel computation.

Pipeline computation

Consider the same specification as in the previous paragraph. Assume that the computation of C can be distributed over N computations of more simple functions C_n as follows:

(5.91) $C\,.\,x \;=\; C_{N-1}\,.\,C_{N-2}\,\ldots\,C_1\,.\,C_0\,.\,x$

where the function application is right associative.

Let $Comp_n$ describe the behaviour of the program for C_n . The duration of the computation is κ_n :

(5.92) $Comp_n \;=\; IH(1+\kappa_n, 1, 2+\kappa_n, 1)$

The pipeline that computes C is a cascade of the programs for C_0 , C_1 , up to C_{N-1} . Its behaviour is given by:

(5.93) $Pipe \;=$

$$(\ \|\ n : 0 \leqslant n < N : p^n.IH(1+\kappa_n, 1, 2+\kappa_n, 1)_{b\to p.a}\,) \Vert\!\upharpoonright \{\,a, b\,\}$$

for a renaming p with $p^N.a \,=\, b$.

In contrast to the cascades for FIFO buffers, we have obtained a cascade in which the cells have other than unit delays. Without proof we mention the behaviour of such cascades. Computing such behaviours simply consists of adding latencies, adding the number of variables, and taking the maximum of the throughput delays.

(5.94) $(IH(\alpha,\beta,\gamma,v)_{b\to c}\ \|\ IH(\bar\alpha,\bar\beta,\gamma,\bar v)_{a\to c})\ \Vert\!\upharpoonright \{\,a, b\,\}$

$$\approx\ \ IH(\alpha+\bar\alpha\,,\,\beta+\bar\beta\,,\,\gamma\,,\,v+\bar v)$$

Furthermore, the conditions on $\dot\approx$ (Conditions 5.44) can be extended with:

(5.95) $e \mathrel{\dot\approx} IH(\alpha,\beta,\gamma,v)\ \wedge\ f \mathrel{\dot\approx} IH(\bar\alpha,\bar\beta,\bar\gamma,\bar v)\quad\Rightarrow$

$$(e_{b\to c}\ \|\ f_{a\to c})\ \Vert\!\upharpoonright \{\,a, b\,\}\ \ \dot\approx\ \ IH(\alpha+\bar\alpha\,,\,\beta+\bar\beta\,,\,\gamma\ \max\ \bar\gamma\,,\,v+\bar v)$$

This allows us to rewrite the behaviour of *Pipe* into

(5.96) $Pipe \;\dot\approx\; IH(\,(1+\bar\kappa)*N\,,\,N\,,\,2+\hat\kappa\,,\,N\,)$

where $\bar{\kappa}$ is the average of κ_n and $\hat{\kappa}$ is the maximum of κ_n (for $0 \leqslant n < N$). The performance of this pipeline is given in Figure 5.97. It resembles the experimental results, as well as the results of logic simulation, that are given in [17] (Figures 9 and 10 respectively).

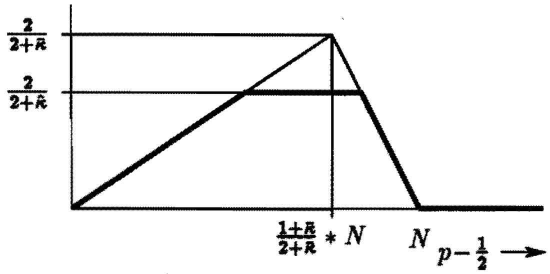


Figure 5.97 Performance of *Pipe*, relative to B_p buffers.

The effect of the duration of computations is illustrated in Figure 5.104. In this figure, III corresponds to the case that all computations are equally fast, $\hat{\kappa} = \bar{\kappa}$. Case I in the picture corresponds to $\hat{\kappa} = \bar{\kappa} + 1$.

Pipeline implementations cannot achieve throughput delays smaller than two times the duration of a communication. Furthermore, an efficient pipeline implementation can only be achieved if a computation can be distributed, as in Formula 5.91, over sufficiently small sub-computations. The costs of pipeline implementations, however, are usually considerably less than for parallel computations. Apart from the overhead, due to distribution of the computation, only *one* realization (implementation) of function C is used, whereas in parallel computations, the number of functions blocks that compute C is proportional to the speeding-up that has to be achieved.

Systolic computation

In general, data specifications are of a less trivial form than $b(i) = C.a(i)$. Take for example the specification of the maximum of segments that is given in Section 4.3. In this section we discuss implementations with systolic arrays that communicate with the environment at both sides.

A basic technique in designing systolic implementations is the introduction of additional channels, see for example [14]. We discuss arrays Pip_N in which only one additional channel (c) is introduced. The arrays consist of N cells and a one-place buffer, as given in Figure 5.98. At the left-hand side, values are received from the environment along channel a. At the right-hand side the results are sent to the environment along channel $b = p^N.a$. Within the array there is an additional 'counter stream' along which old results are sent from the right to the left. The outputs that result from the counter stream are not available to the environment (in contrast to for example the arrays in Section 4.3), in the picture this is symbolized by receiving them in a sink.

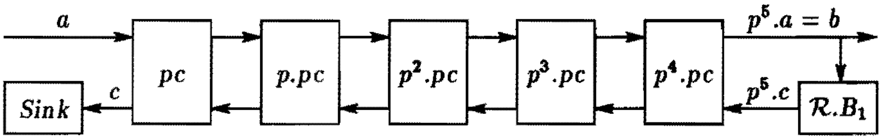


Figure 5.98 Schematic representation of Pip_5.

We assume that the cells meet the following (data) specification:

$$c(i) = p.c(i-1) \qquad p.a(i) = C'(a(i), p.c(i-1))$$

where $p.c(-1)$ is some predefined constant (and *not* a communication along channel $p.c$). In the following program we deliberately separate the computation of va and the communication of va along channel $p.a$. This allows us to investigate the effect of alternative orderings for these actions.

```
program Pip-cell( input $a, p.c$, output $p.a, c$) :
var $va$, $vc$ ;
begin { $vc = p.c(-1)$ }
    ( $c!vc$ ; $p.c?vc$ )*
||
    $a?va$ ; ( $va := C'(va, vc)$ ; ( $p.a!va$ ; $a?va$ ) , $p.c?vc$ )*
end.
```

Let $\bar{\kappa}$ be the duration of a computation (plus an assignment) and let $\kappa = N * \bar{\kappa}$. Enabling structure pc gives the behaviour of the program:

(5.99)
$$\begin{array}{lcl}
pc.s.a_i & = & \textbf{if } i=0 \rightarrow 1 \;\; [] \;\; i>0 \rightarrow s.pa_{i-1}+1 \;\; \textbf{fi} \\
pc.s.pa_i & = & s.\tau_i + 1 \\
pc.s.\tau_i & = & \textbf{if } i=0 \rightarrow s.a_i + \bar{\kappa} \\
 & & [] \;\; i>0 \rightarrow s.a_i + \bar{\kappa} \text{ max } s.pc_{i-1} + \bar{\kappa} \\
 & & \textbf{fi} \\
pc.s.c_i & = & \textbf{if } i=0 \rightarrow 1 \;\; [] \;\; i>0 \rightarrow s.pc_{i-1}+1 \;\; \textbf{fi} \\
pc.s.pc_i & = & s.\tau_i + 1 \text{ max } s.c_i + 1
\end{array}$$

where τ actions symbolize the (completion) of the computations of C'.

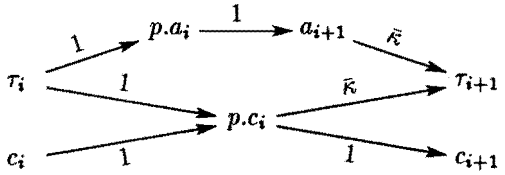


Figure 5.100 Dependencies in pc.

The external behaviour of the array is the same as for a pipeline with a uniform distribution, without overhead: $\hat{\kappa} = \bar{\kappa}$ and $\kappa = \bar{\kappa} * N$. This behaviour is given by

(5.101) $Pip_N \approx IH(N+\kappa,\, N,\, 2+\bar{\kappa},\, N)$

Usually, constructions like $||$ in the program-text of *Pip-cell*, that are freely used in this thesis, are not allowed in programming languages. In particular, the appearance of $pc?vc$ in both parts of the parallel composition may be a problem. A solution is to describe the program with *one* choice-free command. This means, however, that the behaviour has to be restricted: there is no choice-free command that describes the behaviour of pc. We briefly discuss three alternatives:

$$\begin{array}{lrl} (5.102) & \mathrm{I}: & (a,c\ ;\ \tau\ ;\ p.a,p.c)^* \\ & \mathrm{II}: & a;(\tau\ ;\ p.a,c\ ;\ a,p.c)^* \\ & \mathrm{III}: & a;((\tau\,;\,p.a),c\ ;\ a,p.c)^* \end{array}$$

The first behaviour is almost perfect. The only problem is the delay that is imposed by the one-place buffer between $p^N.a$ and $p^N.c$. Apart from a relatively small increase of the latency from b to a, this results in a more significant increase of the throughput delay (see Formula 5.103). In the second behaviour, the combination of restrictions imposed by two adjacent cells results in a throughput delay of $2+2\bar{\kappa}$ (dependencies $p.c_{i-1} \rightarrow \tau_i \rightarrow p.a_i$ and $p.a_i \rightarrow p.\tau_i \rightarrow p.c_i$). Only the third behaviour results in the same external behaviour as the original array with cells of type pc. Verification of these results is left to the interested reader.

$$\begin{array}{lrll} (5.103) & \mathrm{I}: & \dot{\approx} & IH(N+\kappa, N+1, 3+\bar{\kappa}\ , N) \\ & \mathrm{II}: & \dot{\approx} & IH(N+\kappa, N+\kappa, 2+2\bar{\kappa}, N) \\ & \mathrm{III}: & \approx & IH(N+\kappa, N\qquad, 2+\bar{\kappa}\ , N) \end{array}$$

The performance of these three alternatives, as illustrated in Figure 5.104, illustrates that one should be careful when restricting the behaviour of a program. It is advisable to analyze the un-restricted behaviour first in order to obtain an upper bound for the performance. In this case, only the third alternative meets this upper bound.

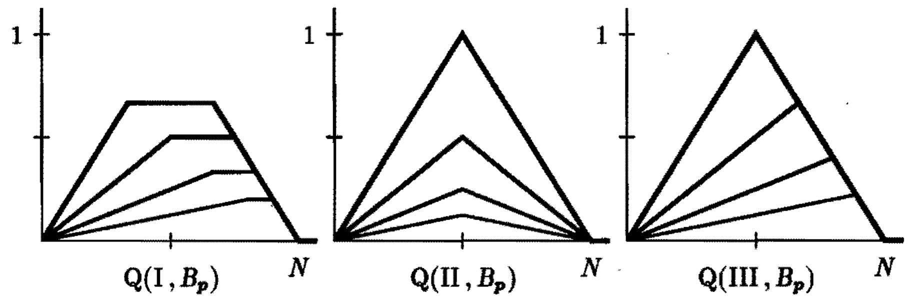


Figure 5.104 Performance of the three alternatives, for $\bar{\kappa} = 0, 1, 3, 7$.[a]

[a]As usual, $p - \frac{1}{2}$ is along the horizontal axis.

Chapter 6

Conclusion

In this thesis we introduced a model that allows us to *express*, *manipulate*, and *compare* (a certain type of) real-time behaviours. The first two topics are discussed in Chapter 2, the third in Chapter 3.

The manipulations concern the parallel composition of mechanisms ($\parallel$), and hiding of actions from the environment, called masking ($\shortparallel$). Another important manipulation is the abstraction from internal actions, called projection ($\upharpoonright$). Masking and projection are completely different operations: masking describes modifications of mechanisms that change external actions into internal actions, projection is used to make the description of mechanisms more abstract, *without* modifying the described mechanisms. When reasoning in terms of external behaviours only, every act of masking should be followed by an act of projection. The combination of both operations is called restriction ($\parallel\!\upharpoonright$).

The comparison concerns the speed of the external behaviour of mechanisms. Both relations, $\lhd$ and $\lhd\!\shortmid$, that have been introduced to compare speed, are suitable for compositional design, in that parallel composition as well as restriction are monotonic (with respect to them). A remarkable feature of both relations is that they are neither reflexive, nor anti-reflexive. The reflexive domains of these relations are considered to be the 'smooth' behaviours. For $\lhd$ these are the ascending behaviours, for $\lhd\!\shortmid$ the conservative behaviours.

In this chapter we discuss some extensions of the *expressive* power, Section 6.1, and the *manipulative* power, Section 6.2, of the model. These include the usage of *zero delays* and the introduction of *serial composition* (catenation). In Section 6.3 we conclude this thesis with some final remarks.

6.1 Expressive Power

Liberal delay conditions

In Example 2.73 we illustrated the phenomenon of dependencies that seem redundant, but formally may not be pruned. Though the necessity of such redundant dependencies does not affect the expressiveness of the model, they are 'unnatural' and may confuse the understanding of a behaviour. In concrete examples, intuition tells us that these dependencies are redundant. In this section we show that one should be careful when trying to formalize 'liberal delay conditions' that would allow such a pruning.

Using Corollary 2.71.1, we can give the following —indirect— definition of 'liberal' enabling structures.

Definition 6.1 Liberal enabling structures.

> A structure F is a liberal enabling structure if (and only if) there exists an enabling structure E over the same alphabets as F that satisfies $E =_{\mathbf{P}E} F$ (that is, the behaviour on $\mathbf{P}E$ is identical).

□

It is, however, far more convenient to have a *direct* characterization of liberal delay conditions. Since, for any enabling structure, only the behaviour on its process really counts (see Corollary 2.71.1), one may be tempted to give the following qualification of liberal enabling structures, using the 'liberal delay' $\mathbf{dr}E$ of enabling structures (compare with Definition 2.21):

A structure E is a liberal enabling structure if (and only if) $\mathbf{b}E > -\infty$ and $\mathbf{dr}E > 0$, where:

$$\mathbf{dr}E \;=\; (\,\mathbf{glb}\, s,t : s,t \in \mathbf{P}E \wedge s \neq t : \mathrm{sim}(\mathbf{f}E.s, \mathbf{f}E.t) - \mathrm{sim}(s,t)\,)$$

for $\mathbf{P}E$ as in Proposition 2.33:

$$\mathbf{P}E \;=\; (\,\mathbf{set}\, s : \mathbf{a}s = \mathbf{a}E \;\wedge\; s \geqslant E.s \;\wedge\; s \restriction \mathbf{i}E = E.s \restriction \mathbf{i}E : s\,)$$

By means of an example, we show that this is not a satisfactory qualification.

Example 6.2 Problems with liberal delays.

> Let e have external alphabet $\{a,b\}$ and an empty internal alphabet. Let furthermore the behaviour of e be defined by:
>
> $$\begin{aligned} e.s.a &= \textbf{if}\ s.a \leqslant 2 \;\rightarrow\; \infty \;\;[]\;\; s.a > 2 \;\rightarrow\; 1 \;\;\textbf{fi} \\ e.s.b &= s.a + 1 \end{aligned}$$
>
> From the formula for $\mathbf{P}E$ we infer $\mathbf{P}\mathrm{e} = (\,\mathbf{set}\, s : s.a > 2 \wedge s.b \geqslant s.a + 1 : s\,)$. From the formula for $\mathbf{dr}E$ we infer $\mathbf{dre} = 1$. However, e does not have a history; the process of $\mathrm{e} \parallel b$ is even empty! In general, each structure E with an empty process has liberal delay $\mathbf{dr}E$ equal ∞ (which is more than zero).

□

Partial enabling structures

In this thesis we assumed that enabling structures must have a 'beginning', in that the behaviour that is described may not begin at $-\infty$. For initiated enabling structures we even assume a beginning after moment zero on the time-axis. Once in a while, however, it may be useful to have partial definitions, or partial estimations, of behaviours, that do not necessarily have such a beginning.

Example 6.3 Usage of partial enabling structures.

Consider the estimation of the behaviour of 'IH' buffers in Section 5.3. Any enabling e function over $\{a, b\}$ that satisfies:

$$e.s.b_i \geqslant s.a_i + \alpha$$

has a quality with respect to $B_{q,p}$ of at most $(2q-1)/\alpha$.

Another way of expressing this, is by using partial enabling functions f_α on $\{a, b\}$ that state latency restrictions as follows:

$$f_\alpha.s.a_i = -\infty \qquad f_\alpha.s.b_i = s.a_i + \alpha$$

From the fact that the quality of f_α with respect to $B_{q,p}$ is $(2q-1)/\alpha$, and $f_\alpha \lhd\!\shortmid\, e$, can be concluded that the quality of e is at most $(2q-1)/\alpha$ (under the assumption that the comparison of enabling functions is extended to partial enabling functions). In this way, the effect of partial behaviour on the total behaviour can be estimated in a compositional way. For example, the 'cascade' of f_α and $f_{\alpha'}$ is $f_{\alpha+\alpha'}$:

$$(f_{\alpha\, b\to c} \parallel f_{\alpha'}\, a \to c) \upharpoonright \{a, b\} \approx f_{\alpha+\alpha'}$$

This knowledge can be used to estimate the latency of a cascade of the buffers e and e', of which the first has latency at least α, $f_\alpha \lhd\!\shortmid\, e$, and the second has latency at least α', $f_{\alpha'} \lhd\!\shortmid\, e'$. Because parallel composition is monotonic with respect to $\lhd\!\shortmid$, we can immediately conclude that the cascade of both buffers has at least latency $\alpha + \alpha'$:

$$f_{\alpha+\alpha'} \lhd\!\shortmid (\mathcal{R}_{b\to c}.e \parallel \mathcal{R}_{a\to c}.e') \upharpoonright \{a, b\} \; .$$

□

Partial enabling structures can be defined as structures[1] E such that for all s over the external alphabet of E, $E \parallel \mathbf{n}s$ is a (non-partial) enabling structure. For partial enabling structures, masking is not always allowed; take for example f_α as in the example above: $f_\alpha \shortparallel b$ is not a partial enabling structure.

Zero delays and almost zero delays

In the enabling model, we assumed a universal, positive lower bound for the delay between cause and effect. This clearly rules out delays of zero time units. Though such zero delays may not be realistic, they can be useful when reasoning 'in abstracto' about the behaviour of some programming constructs.

Example 6.4 Usage of zero delays in modeling interleaving of events.

Consider a device with internal events of type a and b, that share a resource under mutual exclusion. This 'sharing' enforces an interleaving of

[1] In fact as generalizations of structures, for which $E.s.a$ may be $-\infty$.

occurrences of a and b. In order to give an estimation of the (additional) delays that are imposed on the completion of these events, due to interleaving, we first have to give a formal description of this interleaving. We distinguish three stages in the usage of the shared resource:

- the request for the resource: actions $\overline{a}_i$ and $\overline{b}_i$,
- the assignment of the resource, which coincides with the initiation of the event: actions ia_i and ib_i,
- the release of the resource, which coincides with the completion of the event: actions a_i and b_i.

Under the assumption that a actions take α time, and b actions take β time, the enabling of completions is given by:

$$E.s.a_i \;=\; s.ia_i + \alpha \qquad E.s.b_i \;=\; s.ib_i + \beta$$

We assume a 'first requested, first served' regime, in which the causal delays between request and assignment, and between release and next assignment are zero. Furthermore, we assume a slight preference for a actions; that is, when a request for an a action and for a b action occur simultaneously, the resource is assigned to the a action first. This results in the following enabling for actions ia_i and ib_i:

$$E.s.ia_i \;=\; s.\overline{a}_i \max\, (\,\mathbf{lub}\, j : s.\overline{b}_j < s.\overline{a}_i : s.b_j\,)$$

$$E.s.ib_i \;=\; s.\overline{b}_i \max\, (\,\mathbf{lub}\, j : s.\overline{a}_j \leqslant s.\overline{b}_i : s.a_j\,)$$

which is a description with zero delays, under liberal delay conditions.

Finally we assume that the request for a_{i+1} cannot be performed before the completion of a_i; similarly for b events:

$$E.s.\overline{a}_{i+1} \;\geqslant\; s.a_i \qquad E.s.\overline{b}_{i+1} \;\geqslant\; s.b_i$$

Under these conditions, the following bounds for $s.a_i$ and $s.b_i$ can be derived:

$$s.\overline{a}_i + \alpha \;\leqslant\; s.a_i \;\leqslant\; s.\overline{a}_i + \alpha + \beta$$

$$s.\overline{b}_i + \beta \;\leqslant\; s.b_i \;\leqslant\; s.\overline{b}_i + \alpha + \beta$$

Consequently, the 'effective duration' (between request and completion) of a events and b events is at most $\alpha + \beta$.

□

In the above example we used zero delays in a way that can, obviously, be incorporated in the enabling model. In general, however, the usage of zero delays may cause problems. One problem is 'productivity'. A lot of reasoning about enabling structures is based on the fact that, if we know something about schedule s up to moment M, we can conclude something about schedule $E.s$ up to moment $M + \mathrm{d}E$. Unfortunately, an enabling structure E with zero delays has $\mathbf{d}E = 0$. Apart from problems with productivity, there are problems with 'feedback'.

In [10], three important features of 'real-time' semantics are postulated: *responsiveness*, *modularity*, and *causality*. Responsiveness means a reaction delay of zero; modularity states that the behaviour of a composite follows from the behaviours of the participants; causality states that 'for every event that is generated, there is a causal chain of events that leads to this event'. Furthermore, the negative conclusion is drawn that these three features are in conflict: no semantics can exhibit all three of them. The following example, with 'positive feedback' shows a similar result for the enabling model.

Example 6.5 Positive feedback.

Assume that we allow enabling functions with zero delays. Let enabling functions e and f, both over $\{a,b\}$, model mechanisms in which b has to wait for a, and in which a has to wait for b respectively:

$$e.s.a \;=\; 1 \qquad e.s.b \;=\; s.a \qquad f.s.a \;=\; s.b \qquad f.s.b \;=\; 1$$

The parallel composition of e and f is given by:

$$(e \parallel f).s.a \;=\; 1 \max s.b \qquad (e \parallel f).s.b \;=\; 1 \max s.a$$

Let us first consider what happens when actions are performed as soon as they are enabled. That is, let us consider the solutions s of the history equation $(e \parallel f).s = s$ (see Definition 2.27). For each $M : M \geqslant 1$, schedule $\{(a,M),(b,M)\}$ is a solution of this equation. In contrast to enabling functions with positive delays, the history equation for $e \parallel f$ has no unique solution. Under the assumption of 'causality', however, the positive feedback 'a implies b implies a' is no justification of the occurrence of a. This interpretation corresponds to the 'least' solution of the history equation: ε.

The same assumption of causality corresponds to a process $\mathbf{P}(e \parallel f) = \{\varepsilon\}$. However, schedules of type $\{(a,M),(b,M)\}$, for $1 \leqslant M$, are all members of the processes of both e and f, but only ε is a member of $\{\varepsilon\}$. This does not stroke with compositionality: when a schedule is allowed by e as well as f, it should also be allowed by $e \parallel f$.

□

Summarizing, responsiveness, causality, and modularity are conflicting demands. In the enabling model we have chosen for causality and modularity.

An even more exotic example of abusing zero delays is 'negative feedback': action a triggers the enabling of b, but when b happens instantaneously it disables a.

Example 6.6 Negative feedback.

Let enabling function g describe a mechanism that prevents a when b is performed before (or at) a critical moment (2):

```
g.s.a  =  if  $s.b \leqslant 2 \;\rightarrow\; \infty$
          []  $s.b > 2 \;\rightarrow\; 2$
          fi
g.s.b  =  1
```

The history equation of e $\|$ g has no solutions at all (see the previous example for e). When a happens at 2, b is enabled at 2. Assuming that b happens at soon as it is enabled, it also happens at 2 and consequently prevents a from being performed. When on the other hand, a happens after 2, b also happens after 2, and consequently a is enabled at 2. Apparently, a and b cannot both be performed as soon as they are enabled.

□

Both examples illustrate that one should be careful when using enabling functions with zero delays. As soon as delays are non-zero, feedback cannot occur any more. In the model, we even assumed a positive lower bound for all delays. This positive lower bound allows to prove several properties by induction, or by referring to Banach's contraction theorem. A —restricted— way to allow arbitrary small delays is by inserting the following productivity demand, instead of the demand for a positive delay, in the definition of enabling structures (Definition 2.21):

$s \neq t \quad \Rightarrow \quad \mathrm{sim}(E.s, E.t) > \mathrm{sim}(s,t))$

The productivity lies in the ' $>$ ' in this formula.

Example 6.7 Arbitrary small delays.

We present a function e that is allowed by the productivity demand, and a function f that is not. e has a positive 'global delay', whereas for f only the individual delays are positive.

$$\begin{array}{lcl}
e.s.a_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ 1 \\
 & & [\!]\ \ i > 0 \ \rightarrow\ s.a_{i-1} + 2^{-i} \\
 & & \textbf{fi} \\
e.s.b_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ (\,\mathbf{lub}\, i : \ : s.a_i\,) + 1 \\
 & & [\!]\ \ i > 0 \ \rightarrow\ s.b_{i-1} + 1/i \\
 & & \textbf{fi} \\
f.s.a_i & = & \textbf{if}\ \ i = 0 \ \rightarrow\ 1 \\
 & & [\!]\ \ i > 0 \ \rightarrow\ s.a_0 + 1/i \\
 & & \textbf{fi}
\end{array}$$

If $s.a_0 \neq t.a_0$, and for all i $\ s.a_i \geqslant s.a_0$ and $t.a_i \geqslant t.a_0$,
$\mathrm{sim}(f.s, f.t) = \mathrm{sim}(s,t)$.

□

Most propositions remain valid when enabling structures may have arbitrary small delays. The proofs, however, cannot rely on 'simple' induction, or on Banach's theorem any more. For example, it is evident that the history equation has at *most* one solution, but we won't bother the reader with the proof that it has at *least* one solution. Things get even worse when using the following definition of arbitrary small delays, that considers delays for individual actions:

$s \neq t \quad \Rightarrow \quad (\,\forall a : \ : \mathrm{sim}(E.s.a, E.t.a) > \mathrm{sim}(s,t))\,)$

This constraint also allows enabling function f of the previous example.

Since we could not think of 'practical applications' for arbitrary small delays, we took the easy way by forbidding them.

6.2 Manipulative Power

Identification

Identification is a generalization of renaming. Where renaming can be used to connect mechanisms to each other via the proper actions, identification can be used to connect mechanisms to themselves.

Consider, for example, a systolic array such as given in Figure 5.98. The output $p^5.a$ is fed back into the array via a one-place buffer, $\mathcal{R}.B_1$, to input $p^5.c$. Another possibility to establish a feedback, is to connect the output directly to the input, *without* delay. Such a loop can be described by identifying $p^5.c$ and $p^5.a$: $p^5.c = p^5.a = b$ (see Figure 6.8). Such an identification is explicitly prohibited in the definition of renaming.

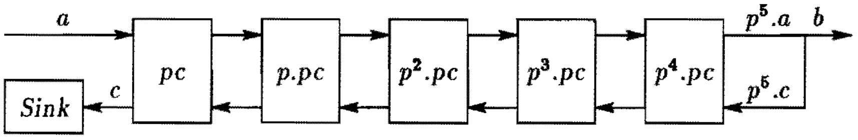


Figure 6.8 Array with instantaneous feedback.

Another way to describe a direct feedback is by using a deviating tail-cell in the first place. The usage of identification on the general cell has the advantage that in this case the array consists of identical cells. It already suggests that the tail-cell can be implemented (in VLSI) in the same way as the other cells: the identification can be established by the surrounding wiring.

Identification of $p.a$ and $p.c$ with b in cell pc (see Formula 5.99) results in the following behaviour:

$$
\begin{array}{llll}
(6.9) & pc'.s.a_i & = & \mathbf{if}\ \ i=0 \rightarrow 1 \\
& & & [\!]\ \ i>0 \rightarrow s.b_{i-1}+1 \\
& & & \mathbf{fi} \\
& pc'.s.\tau_i & = & \mathbf{if}\ \ i=0 \rightarrow s.a_i+\kappa' \\
& & & [\!]\ \ i>0 \rightarrow s.a_i+\kappa'\ \max\ s.b_{i-1}+\kappa' \\
& & & \mathbf{fi} \\
& pc'.s.c_i & = & \mathbf{if}\ \ i=0 \rightarrow 1 \\
& & & [\!]\ \ i>0 \rightarrow s.b_{i-1}+1 \\
& & & \mathbf{fi} \\
& pc'.s.b_i & = & s.\tau_i+1\ \max\ s.c_i+1
\end{array}
$$

Identification may, however, lead to anomalies. Consider for example the alternatives for pc that are given in Formula 5.102. Only the first one gives rise to an acceptable behaviour (in fact to an array that is equally fast as an array with cells of type pc). Identification of $p.a$ and $p.c$ in both other alternatives results in 'lock', because they do not allow $p.a$ and $p.c$ to occur simultaneously. The identifications are given by:

$$
\begin{array}{lrl}
(6.10) & \mathrm{I}_{p.a \to b, p.b \to b}: & (a , c \; ; \; \tau \; ; \; b)^* \\
 & \mathrm{II}_{p.a \to b, p.b \to b}: & a \; ; \; \tau \; ; \; c \; ; \; b^0 \\
 & \mathrm{III}_{p.a \to b, p.b \to b}: & a \; ; \; \tau , c \; ; \; b^0
\end{array}
$$

The reader is invited to give a formal definition of identification.

The introduction of identification as an additional operation on enabling structures causes us to reconsider the way in which we compare enabling structures. We are pleased to inform the reader that identification is not only monotonic with respect to process inclusion, but also with respect to the comparison relations $\lhd$ and $\lhd\!|$. Therefore, both relations can be used for compositional design that includes the possibility of identification.

Sequential composition

In this thesis we introduced parallel composition as a way to compose mechanisms. An operation with which most programmers are more familiar is *sequential composition*, or *catenation*. The catenation of two mechanisms, $E;F$, can be defined as a mechanism that first performs E, and initiates F at the moment that all external activities of E are completed. Under the assumption that F describes a mechanism relative to its moment of initiation, the initiation of this mechanism at moment M results in behaviour $F \oplus M$ (the translation of F over M, see Definition 2.41).

Example 6.11 Catenation.

Let enabling structure E over external alphabet $\{a, b\}$ and internal alphabet c be given by:

$$
\begin{array}{lcl}
E.s.a & = & 1 \\
E.s.b & = & s.a + 1 \min 3 \\
E.s.c & = & s.b + 5
\end{array}
$$

Let furthermore enabling function f over d be given by:

$$f.s.d \;=\; 2$$

The catenation $E;f$ has external alphabet $\{a, b, d\}$ and internal alphabet c, and its behaviour is given by:

$$
\begin{array}{lcl}
(E;f).s.a & = & 1 \\
(E;f).s.b & = & s.a + 1 \min 3 \\
(E;f).s.c & = & s.b + 5 \\
(E;f).s.d & = & (s.a \max s.b) + 2
\end{array}
$$

□

A formal definition of catenation can be given as follows:

Definition 6.12 Catenation of enabling structures.

For E and F such that $\mathbf{e}E \neq \varnothing$, $\mathbf{b}F > 0$, and $\mathbf{a}E \cap \mathbf{a}F = \varnothing$, the catenation $E\,;F$ is an enabling structure with external alphabet $\mathbf{e}E \cup \mathbf{e}F$, internal alphabet $\mathbf{i}E \cup \mathbf{i}F$, with the following behaviour:

```
(E;F).s.a  =  if  $a \in \mathbf{a}E$  →  $E.(s \restriction \mathbf{a}E).a$
              []  $a \in \mathbf{a}F$  →  if  $C.s = \infty$  →  $\infty$
                                  []  $C.s < \infty$  →  $(F \oplus C.s).(s \restriction \mathbf{a}F).a$
                                  fi
              fi
```

where $C.s$ gives the moment that the external activity of E is completed:
$C.s \;=\; (\,\mathbf{lub}\, a : a \in \mathbf{e}E : s.a\,)$.

□

In contrast to identification, catenation affects the way we compare enabling structures: catenation is not monotonic with respect to $\lhd$.

Example 6.13 Catenation and $\lhd$.

Let e and f be defined by:

$e.s.a \;=\; 1$

```
$f.s.b$  =  1
$f.s.c$  =  if  $s.b < 1.5$  →  2
            []  $s.b \geqslant 1.5$  →  $s.b + 2$
            fi
```

Both e and f are ascending, consequently $e \lhd e$ and $f \lhd f$. The catenation of e and f has the following behaviour:

```
$(e;f).s.a$  =  1
$(e;f).s.b$  =  $s.a + 1$
$(e;f).s.c$  =  if  $s.b - s.a < 1.5$  →  $s.a + 2$
                []  $s.b - s.a \geqslant 1.5$  →  $s.b + 2$
                fi
```

This catenation, however, is not ascending, it is not even a member of $[\,Asc\,]$. Let for example $s = \{(a,1),(b,3),(c,5)\}$ and $t = \{(a,2),(b,3),(c,5)\}$.
Though $s \leqslant t$, and both s and t are members of the process of the catenation, $(e\,;f).s.c = 5$ and $(e\,;f).t.c = 3$.
We have to conclude that $\neg((e\,;f) \;\lhd\; (e\,;f))$.

□

Fortunately, catenation *is* monotonic with respect to the robust implementation relation $\lhd\!\mathrel{\mid}$. In fact, $\lhd\!\mathrel{\mid}$ is a maximal implementation relation under the condition of monotonicity of catenation. Therefore, $\lhd\!\mathrel{\mid}$ can be used for compositional design that includes catenation of mechanisms, while $\lhd$ cannot.

Another way to incorporate catenation in the model, is by enhancing enabling structures with explicit external actions for *initiation*, $\perp$, and *completion*, $\top$. A convenient way to do this, is to define $E.\bot = -\infty$ (partial enabling structures), or $E.\bot = 0$, and to allow zero delays in the enabling of $\top$. Catenation of enabling structures with disjunct alphabets (except for $\bot$ and $\top$) can now be defined as a parallel composition in which the completion of the first and the initiation of the second are identified, and subsequently hidden.

Example 6.14 '$\bot - \top$' enabling structures and their catenation.

We rewrite the enabling functions of Example 6.13 into '$\bot - \top$' enabling functions.

$$\begin{array}{lcl} e'.s.\bot & = & 0 \\ e'.s.a & = & s.\bot + 1 \\ e'.s.\top & = & s.a \end{array}$$

$$\begin{array}{lcl} f'.s.\bot & = & 0 \\ f'.s.b & = & s.\bot + 1 \\ f'.s.c & = & \textbf{if} \;\; s.b - s.\bot < 1.5 \;\rightarrow\; s.\bot + 2 \\ & & [\!] \;\; s.b - s.\bot \geqslant 1.5 \;\rightarrow\; s.b + 2 \\ & & \textbf{fi} \\ f'.s.\top & = & s.c \end{array}$$

Observe that f' is *not* ascending, in contrast to f. In fact, a '$\bot - \top$' enabling structure E' is ascending, if (and only if) the corresponding 'normal' enabling structure, E, is conservative. The catenation of e' and f' is given by:

$$\begin{array}{lcl} (e' ;' f').\bot & = & 0 \\ (e' ;' f').s.a & = & s.\bot + 1 \\ (e' ;' f').s.b & = & s.a + 1 \\ (e' ;' f').s.c & = & \textbf{if} \;\; s.b - s.a < 1.5 \;\rightarrow\; s.a + 2 \\ & & [\!] \;\; s.b - s.a \geqslant 1.5 \;\rightarrow\; s.b + 2 \\ & & \textbf{fi} \\ (e' ;' f').\top & = & s.c \end{array}$$

□

We leave formal definitions of this type of enabling structure, and this type of catenation, to the imagination of the reader.

The introduction of catenation, with either definition, is not sufficient to deal properly with enabling structures over generic actions; this is already clear from the convention that alphabets must contain all occurrences of generic actions, or none: the behaviour of an enabling structure over at least one generic action is not completed until ∞. So, a first requirement is to allow enabling structures over the first occurrences of generic actions. Catenation involves then a renumbering of occurrences. Consider for example the enabling function that can perform one occurrence of a: $\bar{e}.s.a_0 = 1$. An enabling function that can perform two occurrences of a is given by $\bar{f} = \bar{e} \; ; \; \bar{e}_{a_0 \rightarrow a_1}$. It is clear that for enabling structures over generic actions the renumbering should be incorporated in the definition of catenation, as is done in Table 6.15. From this table we also learn that, where the old description fails to model catenation properly, the new description fails to model parallel composition properly. Apparently, both operations impose conflicting demands on the way in which enabling structures are used to describe mechanisms over generic actions.

command	old description	new description
a	$e.s.a_i = \textbf{if } i=0 \rightarrow 1 \;[]\; i>0 \rightarrow \infty \textbf{ fi}$	$\bar{e}.s.a_0 = 1$ $\mathbf{a}\bar{e} = \{a_0\}$
a^2	$f.s.a_i = \textbf{if } i=0 \rightarrow 1 \;[]\; i=1 \rightarrow s.a_0+1 \;[]\; i>1 \rightarrow \infty \textbf{ fi}$	$\bar{f}.s.a_0 = 1$ $\bar{f}.s.a_1 = s.a_0+1$ $\mathbf{a}\bar{f} = \{a_0, a_1\}$
compositions	$e\,;e = e$ $e \parallel f = e$	$\bar{e}\,;\bar{e} = \bar{f}$ $\bar{e} \parallel \bar{f} = \bar{f}$

Table 6.15 Comparison between 'old descriptions' and 'new descriptions'.

We mention two solutions for the incompatibility of catenation and parallel composition for mechanisms over generic actions. The first imposes restrictions upon hierarchical design, the second yields a redefinition of parallel composition.

- Hierarchical descriptions of mechanisms must start with a phase of compositions that are similar to those in choice-free commands (Section 2.4): catenation and parallel composition without shared actions. Both are described properly with the 'new description'. This phase must be followed by 'blowing up' the use of generic actions to *all* occurrences of these actions (the added occurrences must be enabled on ∞). This blowing up results in building blocks according to the 'old description' that can be composed in parallel.

- Another solution is to use the 'new description' and to add missing occurrences before computing parallel composition. For example, $\bar{e}$ (see Table 6.15) has one occurrence of a less than $\bar{f}$. Before performing the parallel composition of both, $\bar{e}$ should be blown up to enabling function g over $\{a_0, a_1\}$ that is defined by:

  $g.s.a_0 = 1$ (as for $\bar{e}$)
  $g.s.a_1 = \infty$ (to prohibit a_1)

  The parallel composition of g and f trivially is g. A disadvantage of this blowing up method, is that one obtains whole ranges of enabling structures over distinct alphabets that are equivalent. For example, g is equivalent to e.

An essential remark about blowing up, for both solutions, is that it is monotonic with respect to $\lhd\!\!\mid$.

The problems with catenation become even more interesting, when the number of occurrences can depend on the schedule. For example as in the bypass cell (Section 5.5) in which c actions happen in case of a bypass only (Formula 5.58).

6.3 Miscellaneous

The enabling model provides a means to analyze, and compare, the real-time behaviour of a variety of mechanisms. It supports parallel composition of mechanisms, abstraction from internal actions (projection), and hiding of external actions (masking). In this chapter we showed that the model also supports identification of actions and sequential composition of mechanisms. The speed of behaviours is compared by means of a relation ' $\lhd\!\!\mid$ ' that can be used in compositional design. The analysis of conservative behaviours, is straightforward and relatively easy. For example, process inclusion suffices to conclude that a conservative implementation is at least as fast as a specification. Furthermore, variation of delays in conservative implementations has at most a proportional effect on their speed.

The computation of behaviours, in this thesis, is done by hand. The computation of parallel composition is rather easy; the computation of restriction is usually far more laborious. An interesting topic of research is the automated computation of these operations. For enabling structures in general, this can be considered Utopian. A lot of mechanisms, however, exhibit a kind of 'repetitive' behaviour; for example, of the form $S\,;T^*$. For repetitive mechanisms that have fixed delays automated computation, or estimation, appears to be a realistic option. In [4], the 'timing simulation' (history) of 'pseudorepetitive systems', which have AND-causal dependencies with fixed delays only, is approximated by means of linear programming. We are, however, more interested in efficient computation, or estimation, of the complete external behaviour of mechanisms.

For non-conservative behaviours the analysis becomes less attractive. Take for example the bypass cell, as given in Section 5.5. In contrast to conservative behaviours, there is no —predefined— relation between the index of an action, and the indices of the actions it depends on. This gives rise to delicate manipulations with 'index transformations' (and their inverses), dependent on choices that are made in the past. Furthermore, the behaviour critically depends on the accuracy with which these choices are implemented in 'real world' devices. A suitable way to specify (or describe) such a device is by means of a set of enabling functions that gives all allowed (or possible) functional behaviours of the device. This is a generalization of specifications by means of relation $\lhd\!\!\mid$, in which specification f in fact specifies the set of enabling functions e that satisfy $e \lhd\!\!\mid f$. A typical way to analyze mechanisms that have to meet a conservative specification, but consist of such non-conservative building blocks, is to use comparison relation $\lhd\!\shortmid$ on a global level only, and to use other criteria on a local level.

The model is developed to describe true concurrency: it does not enforce *artificial* interleaving of actions in the description of concurrency. A drawback of the model is the effort that is required to describe *intended* interleaving (e.g. Example 6.4). Another problem is that the way in which we compare the speed of mechanisms, is not really fit for mechanisms that interleave (external) actions.

In this thesis we concentrated on real-time behaviour, neglecting the data. This separation of concerns is only possible when the real-time behaviour is data independent. In some cases of data dependency it suffices to decouple the real-time behaviour by introducing non-determinism. This is done by describing a device with a set of the enabling structures that correspond to the real-time behaviours of the device for each possible (input) data. This leads, however, to loss of information that may be crucial to prove correctness of the real-time behaviour. For a concise description of devices that exhibit data-dependence, the enabling of actions must be given dependent on values that are received (or sent).

In the enabling model, we have chosen to describe the real-time behaviour of a wide class of deterministic, non-disabling mechanisms, rather than to restrict the model to a subclass of these behaviours. If only the conservative behaviours are considered, one obtains a relatively simple model in which process inclusion suffices to compare speed. It may be worthwhile to study properties of even more restricted subclasses of behaviours, such as the possibility for automated computation of operations, which is mentioned before. In the opposite direction, it is interesting to study generalizations, such as behaviours that may be non-deterministic, possibly probabilistic. Another generalization is the above-mentioned inclusion of the description of data. It needs no argument that such modifications require other ways to compare and specify behaviours — instead of the relation $\lhd\!\shortmid$ and the induced notion of quality, which are used in this thesis. Even for the model as is, it is interesting to consider other types of comparison and specification.

Appendix A

Notational Conventions

Functions

A function in $A \to B$ is a function with domain A and range B; we use this notation rather than the notation B^A. For function application we use the notation with a dot, as well as a notation without a dot: $f.x = f.(x) = f(x) = f\,x$.

For functions with a totally ordered range, lub (glb) is defined as the pointwise maximum (minimum), where $\leqslant$ is defined as the related partial order. That is, for f and g functions of the same type:

$(f \text{ lub } g).a \;=\; f.a \max g.a$ (for all a in the domain of these functions)

and $f \leqslant g \Leftrightarrow (f \text{ lub } g) = g$.

For functions we also use a set-notation. For example, $\{(a,3),(b,1)\}$ is used to denote the function with domain $\{a,b\}$ that maps a on 3 and b on 1 .

Miscellaneous

The number of elements of a set X is denoted by $|X|$.

For the sake of convenience, we frequently use a *hidden and*. For example, we write $a = b = c$ instead of $(a \doteq b) \wedge (b = c)$ and $a,b \in A$ instead of $(a \in A) \wedge (b \in A)$. The way we formulate derivations is also based on this hidden and. For example the following derivation of $(f \text{ lub } g).a \geqslant f.a$:

$\quad (f \text{ lub } g).a$

$=\quad$ { definition of lub }

$\quad f.a \max g.a$

$\geqslant\quad$ { definition of max }

$\quad f.a$

The (optional) remarks between braces are called hints; they are inserted to clarify the individual steps of the derivation.

Quantified expressions

The general pattern we use for quantification is:

(**quantifier** *dummies* : *range of the dummies* : *quantified expression*)

Restrictions in the range that are evident from the context are usually omitted. Quantification over an empty range gives the unit element of the quantifier.

Below an example of quantified expressions, under the convention that x and y are reals.

$$\begin{array}{lcl}
(\exists x,y : \ : x^2 = -(1+y^2)) & = & \mathbf{false} \\
(\mathbf{lub}\, x,y : x^2 = -(1+y^2) : x) & = & -\infty \\
(\mathbf{set}\, x : x \in \{-1,0,1\} : x^2) & = & \{0,1\} \\
(\mathbf{glb}\, x : x \in \{-1,0,1\} : x^2) & = & 0
\end{array}$$

The quantifiers we use are:

quantifier	meaning	unit element
$\forall$	universal quantification	**true**
$\exists$	existential quantification	**false**
glb	greatest lower bound	dependent on type of expression
lub	least upper bound	idem
max	maximum	idem
set	set constructor	$\varnothing$
$\sum$	summation	0
$\cup$	union of sets	$\varnothing$
$\cap$	intersection of sets	dependent on type of expression
$\parallel$	parallel composition	idem

Binding power

In order to save parentheses we attach different binding powers to several operators. The lowest binding power is given to binary relations, then follow the other binary operations and then the unary operations. A refinement of this tri-partition is given in the table below.

$\Rightarrow\,,\ \Leftarrow\,,\ \Leftrightarrow$	Increasing Binding Power $\downarrow$
$\wedge\,,\vee$	
other binary relations	
$\Vert\,,\ \max\,,\ \min\,,\ \mathrm{lub}\,,\ \mathrm{glb}$	
$\uparrow,\ \upharpoonright,\ \Vert\!\upharpoonright,\ \backslash,\ ",\ \downharpoonright$	
$+\,,\ \oplus\,,\ -$	
$*\,,\ \odot$	
function application with a dot	
function application without a dot	
unary operations	

In order to save even more parentheses, binary operations with the same binding power are (mutually) left associative; except for function application without a dot, which is right associative. For example: $5-2+3 = (5-2)+3$ and $e.s.a = (e.s).a$, but $\mathbf{b}\,\mathbf{P}E = \mathbf{b}(\mathbf{P}E)$.

Appendix B

{AND,OR} Causality[1]

In Section 2.2 we used dependence functions to introduce the basic concept of enabling. Dependence functions state the dependence of one action on an arbitrary number of (other) actions. In this appendix we introduce two types of composition on the level of dependence functions: *AND-causal* composition and *OR-causal* composition. The first is similar to parallel composition ($\|$): an action is enabled as soon as *all* preconditions are satisfied. In the latter, an action is enabled by a composition as soon as *at least one* of the preconditions is satisfied. Furthermore we introduce *1-dependence functions* as dependence functions that describe dependence on at most one action.

This appendix is devoted to the composition of dependence functions out of 1-dependence functions by means of AND- and OR-composition. Theorem B.7 states that by {AND,OR} composition of 1-dependence functions, one can obtain all dependence functions, and thus the behaviour of all enabling structures. Using ascending 1-dependence functions one can even obtain all ascending dependence functions. Conservative dependence functions can be described in terms of {AND,OR} composition of 1-dependence functions with fixed delays (between cause and effect). The latter result is used in Sections 2.5 and 3.6.

We only compose dependence functions over the *same* alphabet by means of AND- and OR-composition; we leave this alphabet implicit. Furthermore we adopt the conventions $\infty - \infty = 0$ and $0 * \infty = 0$. The symbols ϕ and ψ denote dependence functions, Φ and Ψ denote sets of dependence functions.

The AND-composition of dependence functions ϕ and ψ is given by ϕ lub ψ; their OR-composition is given by ϕ glb ψ. Both, AND- and OR-composition, are generalized to composition over sets of dependence functions. AND- and OR-composition of dependence functions may result in functions that are no dependence functions. Apart from a step that may become 0 (as for enabling structures), the function value may become $-\infty$, which is not a member of the time domain. For example:

[1] **The terminology of '{AND,OR} causality' is borrowed from [8].**

$$\begin{array}{rcl} (\,\mathbf{lub}\,\phi : \phi \in \varnothing : \phi\,).s & = & -\infty \\ (\,\mathbf{glb}\, i : i > 0 \wedge \phi_i.s = -i : \phi_i\,).s & = & -\infty \end{array}$$

In the sequel when referring to AND (OR) -composition, we implicitly assume that the result is a dependence function.

A *composition class* is a set of dependence functions that is closed under AND-composition as well as OR-composition. Φ is a *base* of composition class Ψ if $\Phi \subseteq \Psi$ and if all elements of Ψ can be composed out of elements of Φ (by means of AND-composition and OR-composition).

A *1-dependence function* is a dependence function that depends on the scheduling of at most one action. For example, $s.a + 1$ and the function identical 1 are 1-dependence functions, but $s.a + 1$ max $s.b + 3$ is *no* 1-dependence function (it depends on both a and b).

Definition B.1 1-dependence function.

Dependence function ϕ is a *1-dependence function* if there exists an action a such that:

$(\,\forall s,t : s.a = t.a : \phi.s = \phi.t\,)$

□

The *variation* of a dependence function gives a kind of 'smoothness criterion'; dependence functions with low variation are not sensitive for small deviations of the schedule.

Definition B.2 Distance, variation: dis , v .

The *distance* ' dis ' between two schedules is given by:

$$\mathrm{dis}(s,t) \;=\; (\,\mathbf{lub}\, a : \;: |\, s.a - t.a\,|\,)$$

The *variation* of dependence function ϕ, $\mathrm{v}\phi$, is defined by:

$$\mathrm{v}\phi \;=\; (\,\mathbf{lub}\, s,t : s \neq t : \frac{|\,\phi.s - \phi.t\,|}{\mathrm{dis}(s,t)}\,)$$

□

Proposition B.3 (without proof)

A dependence function is conservative if (and only if) it is ascending and has variation at most 1 .

□

The *monus* is used in Definition B.5.

Definition B.4 (monus) $\dot{-}$.

$$M \dot{-} N \;=\; M - N \max 0$$

□

The following dependence functions are used as 1-dependent building blocks in the {AND,OR} construction of dependence functions.

Definition B.5 $\bar\pi$, $\acute\pi$, $\hat\pi$, and $\check\pi$.

We define the 1-dependence functions $\bar\pi(N)$, $\acute\pi(\delta,a)$, $\hat\pi(x,M,\delta,a)$, and $\check\pi(x,M,\delta,a)$ by:

$$\begin{array}{lcll}
\bar\pi(N).s & = & N & \text{for } N \in \mathbf{T} \\
\acute\pi(\delta,a).s & = & s.a+\delta & 0<\delta<\infty \\
\hat\pi(x,M,N,a).s & = & x*(s.a \mathbin{\dot{-}} M)+N & 1\leqslant x\leqslant\infty ,\quad M<N<\infty \\
\check\pi(x,M,N,a).s & = & x*(M \mathbin{\dot{-}} s.a)+N & 0<x\leqslant\infty ,\quad M<N<\infty
\end{array}$$

□

These functions are 1-dependence functions, some of their characteristics are given in Table B.6. Dependence functions of type $\bar\pi$ and $\acute\pi$ are also called the 1-dependence functions *with fixed delays.* The delay of $\acute\pi(\delta,a)$ is δ . In case of the description of a mechanism relative to the moment of initiation, $\bar\pi(N)$ is only used with $N>0$; in this case N is the (initial) delay of $\bar\pi(N)$.

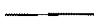
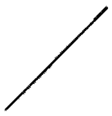
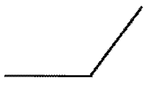
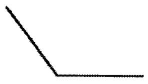
	$\bar\pi(N)$	$\acute\pi(\delta,a)$	$\hat\pi(x,M,N,a)$	$\check\pi(x,M,N,a)$
delay	∞	δ	$N-M$	$N-M$
variation	0	1	x	x
type	constant	conservative	ascending	
shape [a]				

Table B.6 Some characteristics of the building blocks.

[a] $s.a$ along the horizontal axis, $\pi.s$ along the vertical axis.

Theorem B.7 (see below) gives a nice theoretical result about the expressiveness of 1-dependence functions and {AND,OR} composition. Since the real-time behaviour of a lot of programs can be described in terms of fixed delays and {AND,OR} composition, and because the conservative enabling functions are the reflexive domain of comparison relation $\lhd|$, we consider the last item of particular interest.

Theorem B.7

1 Let $\nabla : 1\leqslant\nabla\leqslant\infty$ and $\Delta : 0<\Delta<\infty$.
The functions of type $\hat\pi(\nabla,M,N,a)$ and $\check\pi(\nabla,M,N,a)$ with $N-M\geqslant\Delta$ are a base of the composition class of dependence functions with variation at most ∇ and delay at least Δ .

2 The functions of type $\hat\pi(\infty,M,N,a)$ and $\check\pi(\infty,M,N,a)$ are a base of the composition class of all dependence functions.

3 Let $\nabla : 1 \leqslant \nabla \leqslant \infty$ and $\Delta : 0 < \Delta < \infty$.
The functions of type $\hat{\pi}(\nabla, M, N, a)$ with $N - M \geqslant \Delta$ and $\bar{\pi}(N)$ are a base of the composition class of ascending dependence functions with variation at most ∇ and delay at least Δ.

4 The functions of type $\hat{\pi}(\infty, M, N, a)$ and $\bar{\pi}(N)$ are a base of the composition class of all ascending enabling functions.

5 The functions of type $\bar{\pi}(N)$ and $\acute{\pi}(\delta, a)$ are a base of the composition class of all conservative dependence functions.

Proof

It is left to the reader to verify that all suggested composition classes are indeed composition classes, and that the suggested bases are subsets of these classes. Remains to prove that all elements of the classes can be composed out of elements of the suggested bases.

1 Let $\nabla : 1 \leqslant \nabla \leqslant \infty$ and $\Delta : 0 < \Delta < \infty$, and let ϕ be a dependence function with variation $\mathbf{v}\phi \leqslant \nabla$ and delay $\mathbf{d}\phi \geqslant \Delta$.

We exhibit dependence functions $\phi.(t, M)$ that are AND-causal compositions of elements of the suggested base, and that can be OR-causally composed into ϕ.

Define for t and $M : M < \infty$ such that $\phi.t < M + \Delta$ the function $\phi.(t, M)$ by: [2]

$$\begin{aligned}\phi(t,M) \;=\; & (\,\mathbf{lub}\, b : t.b < M : \check{\pi}(\nabla, t.b, M+\Delta, b)\,) \quad \max \\ & (\,\mathbf{lub}\, b : t.b < M : \hat{\pi}(\nabla, t.b, M+\Delta, b)\,) \quad \max \\ & (\,\mathbf{lub}\, b : t.b \geqslant M : \check{\pi}(\nabla, M, M+\Delta, b)\,)\end{aligned}$$

Remains to prove: $\phi \;=\; (\,\mathbf{glb}\, t, M : \phi.t < M + \Delta \wedge M < \infty : \phi(t, M)\,)$

The proof of this decomposition is divided in two parts:

- For any s: $\phi.s \;=\; (\,\mathbf{glb}\, M : \phi.s < M + \Delta \wedge M < \infty : \phi(s, M).s\,)$.

  In case $\phi.s = \infty$ this is evident; for the case $\phi.s < \infty$ we derive:

$$\begin{aligned} & \phi(s,M).s \\ = \quad & \{\text{definitions of } \phi(t,M) \text{ and of } \hat{\pi} \text{ and } \check{\pi}\,\} \\ & (\,\mathbf{lub}\, b : s.b < M : \nabla * (s.b \dot{-} s.b) + M + \Delta\,) \quad \max \\ & (\,\mathbf{lub}\, b : s.b < M : \nabla * (s.b \dot{-} s.b) + M + \Delta\,) \quad \max \\ & (\,\mathbf{lub}\, b : s.b \geqslant M : \nabla * (M \dot{-} s.b) + M + \Delta\,) \\ = \quad & \{\text{definition of } \dot{-}\text{, alphabet non-empty}\,\} \\ & M + \Delta \end{aligned}$$

[2] If $\mathbf{d}\phi > \Delta$, it suffices to observe the cases $\phi.t = M + \Delta$; but we also want to capture the case $\mathbf{d}\phi = \Delta$.

Because $\Delta < \infty$, this establishes the equality.

- For any s , t , and $M < \infty$ such that $\phi.t < M + \Delta$: $\phi.s \leqslant \phi(t, M).s$.

  Let s , t , and M as in the precondition. Define u by:

```
u.b  =  if  $t.b < M$                  →  $t.b$
        []  $t.b \geqslant M \wedge s.b < M$  →  $M$
        []  $t.b \geqslant M \wedge s.b \geqslant M$  →  $s.b$
        fi
```

  Observe that $u \downharpoonright M = t \downharpoonright M$; since $\phi.t < M + \Delta \leqslant M + \mathbf{d}\phi$, we conclude: $\phi.u = \phi.t$.

  We derive:

$$
\begin{array}{ll}
 & \phi(t,M).s \\
= & \quad \{\text{definitions of } \phi(t,M) \text{ and of } \hat{\pi} \text{ and } \check{\pi}\ \} \\
 & (\,\mathbf{lub}\, b : t.b < M : \nabla * (t.b \dot{-} s.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b < M : \nabla * (s.b \dot{-} t.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b \geqslant M : \nabla * (M \dot{-} s.b) + M + \Delta\,) \\
= & \quad \{\text{definition of } \dot{-}\ \} \\
 & (\,\mathbf{lub}\, b : t.b < M : \nabla * (t.b \dot{-} s.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b < M : \nabla * (s.b \dot{-} t.b) + M + \Delta\,) \quad \max \\
 & \\
 & (\,\mathbf{lub}\, b : t.b \geqslant M \wedge s.b < M : \nabla * (M \dot{-} s.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b \geqslant M \wedge s.b < M : \nabla * (s.b \dot{-} M) + M + \Delta\,) \quad \max \\
 & \\
 & (\,\mathbf{lub}\, b : t.b \geqslant M \wedge s.b \geqslant M : \nabla * (s.b \dot{-} s.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b \geqslant M \wedge s.b \geqslant M : \nabla * (s.b \dot{-} s.b) + M + \Delta\,) \\
= & \quad \{\text{definition of } u\ \} \\
 & (\,\mathbf{lub}\, b : \ : \nabla * (u.b \dot{-} s.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : \ : \nabla * (s.b \dot{-} u.b) + M + \Delta\,) \\
= & \quad \{\text{definition of } \dot{-}\ \} \\
 & (\,\mathbf{lub}\, b : \ : \nabla * \mid s.b - u.b \mid + M + \Delta\,) \\
= & \quad \{\text{definition of dis}\ \} \\
 & \nabla * \text{dis}(s,u) + M + \Delta \\
\geqslant & \quad \{\ \nabla \geqslant \mathbf{v}\phi\ \} \\
 & \mid \phi.s - \phi.u \mid + M + \Delta \\
\geqslant & \quad \{\ \phi.u = \phi.t \text{ and } \phi.t < M + \Delta\ \} \\
 & \mid \phi.s - \phi.u \mid + \phi.u \\
\geqslant & \\
 & \phi.s
\end{array}
$$

2 Follows from 1.

3 This proof is similar to the proof of 1.

Let $\nabla : 1 \leqslant \nabla \leqslant \infty$ and $\Delta : 0 < \Delta < \infty$, and let ϕ be an ascending dependence function with variation $\mathbf{v}\phi \leqslant \nabla$ and delay $\mathbf{d}\phi \geqslant \Delta$.

Define for t and $M : M < \infty$ such that $\phi.t < M + \Delta$ the function $\phi.(t, M)$ by:

$$\phi(t,M) \;=\; (\,\mathbf{lub}\, b : t.b < M : \hat{\pi}(\nabla, t.b, M+\Delta, b)\,)$$

where $\mathrm{lub}\,\varnothing = \bar{\pi}(M+\Delta)$.

Remains to prove: $\phi \;=\; (\,\mathbf{glb}\, t, M : \phi.t < M + \Delta \wedge M < \infty : \phi(t,M)\,)$

The proof of this decomposition is divided in two parts:

- For any s: $\phi.s \;=\; (\,\mathbf{glb}\, M : \phi.s < M + \Delta \wedge M < \infty : \phi(s,M).s\,)$.

  This is proven similar to the proof of the same property in 1.

- For any s, t, and $M < \infty$ such that $\phi.t < M + \Delta$: $\phi.s \leqslant \phi(t,M).s$.

  Let s, t, and M as in the precondition. Define u by:

$$\begin{array}{lll} u.b & = & \mathbf{if}\;\; t.b < M \wedge s.b \geqslant t.b \;\rightarrow\; t.b \\ & & [\!]\;\; t.b \geqslant M \vee s.b < t.b \;\rightarrow\; s.b \\ & & \mathbf{fi} \end{array}$$

  Observe that $u \downharpoonleft M \leqslant t \downharpoonleft M$; since $\phi.t < M + \Delta \leqslant M + \mathbf{d}\phi$ and ϕ is ascending, we conclude: $\phi.u \leqslant \phi.t$.

  Observe also that $u \leqslant s$ and consequently $\phi.u \leqslant \phi.s$.

  We derive:

$$\begin{array}{ll}
 & \phi(t,M).s \\
= & \quad \{\text{definitions of } \phi(t,M),\ \bar{\pi},\text{ and } \hat{\pi}\ \} \\
 & M + \Delta \;\max\; (\,\mathbf{lub}\, b : t.b < M : \nabla * (s.b \dot{-} t.b) + M + \Delta\,) \\
= & \quad \{\text{definition of } \dot{-}\ \} \\
 & (\,\mathbf{lub}\, b : t.b < M \wedge s.b < t.b : \nabla * (s.b \dot{-} s.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b < M \wedge s.b \geqslant t.b : \nabla * (s.b \dot{-} t.b) + M + \Delta\,) \quad \max \\
 & (\,\mathbf{lub}\, b : t.b \geqslant M : \nabla * (s.b \dot{-} s.b) + M + \Delta\,) \\
= & \quad \{\text{definition of } u\ \} \\
 & (\,\mathbf{lub}\, b : \; : \nabla * (s.b \dot{-} u.b) + M + \Delta\,) \\
= & \quad \{\text{definition } \dot{-},\ s \geqslant u\ \} \\
 & (\,\mathbf{lub}\, b : \; : \nabla * |\, s.b - u.b\,| + M + \Delta\,) \\
= & \quad \{\text{definition of dis}\ \} \\
 & \nabla * \mathrm{dis}(s,u) + M + \Delta \\
\geqslant & \quad \{\ \nabla \geqslant \mathbf{v}\phi\ \} \\
 & |\, \phi.s - \phi.u\,| + M + \Delta \\
= & \quad \{\ \phi.s \geqslant \phi.u\ \}
\end{array}$$

$$
\begin{array}{ll}
& (\phi.s - \phi.u) + M + \Delta \\
\geqslant & \qquad \{ \ \phi.u \leqslant \phi.t \ \text{ and } \ \phi.t < M + \Delta \ \} \\
& \phi.s
\end{array}
$$

4 Follows from 3.

5 A dependence function ϕ is conservative if it is ascending and has variation at most one (Proposition B.3). Consequently, the claim in the theorem follows from 3, with $\nabla = 1$, and from the following 'decomposition':

$$\hat{\pi}(1, M, N, a) \ = \ \bar{\pi}(N) \ \text{max} \ \hat{\pi}(N - M, a)$$

□

Appendix C

Metric Spaces

First we give a brief introduction to (ultra-) metric spaces, including a generalization of Banach's contraction theorem. Furthermore we introduce metric spaces of schedules, processes, and enabling structures. In order to support the theory about infinite systolic arrays in Chapter 4, we prove the continuity of $\lhd\!\shortmid$ in the metric spaces of enabling functions.

For set X , and function $d : X * X \rightarrow$ the real numbers, the pair (X,d) is a *metric space* if for all x , y , and z in X :

$$\begin{array}{lll} d(x,y) & \geqslant & 0 \\ d(x,y) & = & 0 \quad \Leftrightarrow \quad x = y \\ d(x,y) & = & d(y,x) \\ d(x,z) & \leqslant & d(x,y) + d(y,z) \end{array}$$

It is called an *ultra*-metric space if the following —stronger— triangle inequality holds:

$$d(x,z) \quad \leqslant \quad d(x,y) \max d(y,z)$$

A metric space is *complete* if each Cauchy sequence has a limit; where a *Cauchy sequence* is a sequence $x_i : i \geqslant 0$ such that

$$(\forall \varepsilon : \varepsilon > 0 : (\exists k : \ : (\forall l, m : l, m \geqslant k : d(x_k, x_l) \leqslant \varepsilon)))$$

A function Φ in $X \rightarrow X$ is a contraction if there exists a real number $\phi : 0 \leqslant \phi < 1$ such that for all x and y :

$$d(\Phi.x, \Phi.y) \quad \leqslant \quad \phi * d(x,y)$$

Theorem C.1 Generalized contraction theorem.

For Φ a function in a complete metric space, such that for for some $k : k > 0$, Φ^k is a contraction:

- Φ has a unique fixed-point, say x , and
- any sequence $x_n : n \geqslant 0$ such that $x_{n+1} = \Phi.x_n$ has limit x .

□

The original theorem of Banach is for $k = 1$.

To apply the theory of metric spaces we have to introduce a concept of distance; we use the concept of similarity in order to do so.

Definition C.2 Metric d.

Let X and Y such that $\mathrm{sim}(X,Y)$ is defined (see Definitions 2.15 and 2.23). The distance $\mathrm{d}(X,Y)$ of X and Y is defined by:

$$\mathrm{d}(X,Y) \;=\; 2^{-\mathrm{sim}(X,Y)}$$

□

A similar distance is used in [26] between sets of traces. The metric for processes is also called the *Hausdorff distance*.

Without proof we mention:

Proposition C.3

1 For any alphabet A, $(\mathcal{S}.A, \mathrm{d})$ is a complete ultra-metric space.

2 For any alphabet A, the set of processes over A is, with metric d, a complete ultra-metric space.

3 For $A, B : A \cap B = \varnothing$, $(\mathcal{ES}(A,B), \mathrm{d})$ is an ultra-metric space; for any $\Delta : \Delta > 0$, the subset of enabling structures with delay at least Δ is a complete ultra-metric space.

□

Theorem C.4 Contractions in terms of similarity.

Function Φ, in one of the metric spaces as given in the previous proposition, is a contraction if and only if for some Δ, $\Delta > 0$:

$$(\forall\, x, y : \; : \mathrm{sim}(\Phi.x, \Phi.y) \geqslant \mathrm{sim}(x,y) + \Delta\,)$$

Proof

$$\begin{array}{ll}
 & \mathrm{sim}(\Phi.x, \Phi.y) \;\geqslant\; \mathrm{sim}(x,y) + \Delta \\
\Leftrightarrow & 2^{-\mathrm{sim}(\Phi.x, \Phi.y)} \;\leqslant\; 2^{-(\mathrm{sim}(x,y) + \Delta)} \\
\Leftrightarrow & 2^{-\mathrm{sim}(\Phi.x, \Phi.y)} \;\leqslant\; 2^{-\mathrm{sim}(x,y)} * 2^{-\Delta} \\
\Leftrightarrow & \quad \{\text{definition of d}\} \\
 & \mathrm{d}(\Phi.x, \Phi.y) \;\leqslant\; 2^{-\Delta} * \mathrm{d}(x,y)
\end{array}$$

□

From this theorem we immediately infer that (the behaviour) of an enabling structure is a contraction. Consequently the history (see Definition 2.27) of an enabling structure exists, and is uniquely defined.

We conclude this appendix with a theorem that is used in Section 4.2, in order to estimate the external behaviour of infinite linear systolic arrays.

Theorem C.5 Continuity of $\lhd$ and $\lhd\!\!\shortmid$.

For sequences $e_n : n \geqslant 0$ and $f_n : n \geqslant 0$ with limits e and f respectively:

1 $(\forall n : : e_n \lhd f_n) \;\Rightarrow\; e \lhd f$

2 $(\forall n : : e_n \lhd\!\!\shortmid f_n) \;\Rightarrow\; e \lhd\!\!\shortmid f$

Proof

Define $M_n \;=\; \mathrm{sim}(e_n, e) \;\mathrm{min}\; \mathrm{sim}(f_n, f)$.

1 Let $s \in \mathbf{P}e$ and $t \in \mathbf{P}f$ such that $s \leqslant t$. We derive for all n:

$$\begin{array}{ll}
& e_n \lhd f_n \\
\Rightarrow & \{\text{definition of } \lhd \text{ (3.27)}, \; s \downharpoonright M_n \in \mathbf{P}e_n \text{, similar for } t\} \\
& e_n.(s \downharpoonright M_n) \leqslant f_n.(t \downharpoonright M_n) \\
\Rightarrow & \\
& e_n.s \downharpoonright M_n \leqslant f_n.t \downharpoonright M_n \\
\Leftrightarrow & \\
& e.s \downharpoonright M_n \leqslant f.t \downharpoonright M_n
\end{array}$$

Since $\lim_{n\to\infty} M_n = \infty$ we conclude $e.s \leqslant f.t$. From Definition 3.27 we conclude $e \lhd f$.

2 We derive:

$$\begin{array}{ll}
& (\forall n : : e_n \lhd\!\!\shortmid f_n) \\
\Leftrightarrow & \{\text{Proposition 3.38}\} \\
& (\forall n, \mu : \mu \geqslant 0 : e_n \lhd f_n \oplus \mu) \\
\Rightarrow & \{\text{previous item}\} \\
& (\forall \mu : \mu \geqslant 0 : e \lhd f \oplus \mu) \\
\Leftrightarrow & \{\text{Proposition 3.38}\} \\
& e \lhd\!\!\shortmid f
\end{array}$$

□

Bibliography

[1] J.C.M. Baeten and J.A. Bergstra,
Real Time Process Algebra,
Formal Aspects of Computing Vol. 3, No. 2, April-June 1991. Springer International: pp 142-188.

[2] C.H. (Kees) van Berkel, Martin Rem, and Ronald W.J.J. Saeijs,
VLSI Programming,
Proceedings, 1988 IEEE International Conference on Computer Design: pp 152-156.

[3] C.H. (Kees) van Berkel and Ronald W.J.J. Saeijs,
Compilation of Communicating Processes into Delay-Insensitive Circuits,
Proceedings, 1988 IEEE International Conference on Computer Design: pp 157-162.

[4] Steven M. Burns and Alain J. Martin,
Performance Analysis and Optimization of Asynchronous Circuits,
Advanced Research in VLSI 1991, UC Santa Cruz: pp 71-86.

[5] Jo C. Ebergen,
Translating Programs into Delay-Insensitive Circuits,
Ph.D. thesis, October 1987, Eindhoven University of Technology

[6] P.C. Fischer, A.R. Meyer, and A.L. Rosenberg,
Real-time Simulation of Multihead Tape Units,
Journal of the ACM, Vol. 19, No. 4, October 1972: pp 590-607.

[7] Leo J. Guibas and Frank M. Liang,
Systolic Stacks, Queues, and Counters,
Proceedings, Conference on Advanced Research in VLSI, Paul Penfield Jr. editor, January 1982, Massachusetts Institute of Technology: pp 55-164.

[8] J. Gunawardena,
Causal Automata I: Confluence $\equiv$ *{AND,OR} Causality*,
Semantics for Concurrency, Proceedings of the international BCS-FACS Workshop, July 1990, University of Leicester UK. Springer 1990: pp 137-156.

[9] M. Hennessy,
Algebraic Theory of Processes,
The MIT Press, 1988.

[10] C. Huizing and R. Gerth,
On the semantics of reactive systems
Technical report, 1987, Eindhoven University of Technology.
Also appeared in: C. Huizing,
Semantics of reactive systems: comparison and full abstraction,
Ph.D. thesis, March 1991, Eindhoven University of Technology: pp 103-120.

[11] C.A.R. Hoare,
Communicating Sequential Processes,
Prentice Hall, New York 1985.

[12] Wout J. Janse,
A Queue with Bounded Response Time and Maximum Storage Utilization?,
Master's thesis, August 1988, Eindhoven University of Technology.

[13] A. Kaldewaij,
A Formalism for Concurrent Processes,
Ph.D. thesis, May 1986, Eindhoven University of Technology.

[14] Anne Kaldewaij and Martin Rem,
A derivation of a systolic rank order filter with constant response time,
Mathematics of Program Construction, J.L.A. van de Snepscheut editor,
Lecture Notes in Computer Science 375, Springer 1989: pp 281-296.
Or, from the same authors,
The Derivation of Systolic Computations,
Science of Computer Programming 14 (1990): pp 229-242.

[15] Joep L.W. Kessels and Martin Rem,
Designing systolic, distributed buffers with bounded response time,
Distributed computing (1990) 4, Springer 1990: pp 37-43.

[16] W.E.H. Kloosterhuis,
Livelock in Concurrent Processes,
Master's thesis, September 1987, Eindhoven University of Technology.

[17] Shinji Komori, et al.,
An Elastic Pipeline Mechanism by Self-Timed Circuits,
IEEE Journal of Solid-State Circuits, Vol. 23, No. 1, Februari 1988: pp 111-117.

[18] Leslie Lamport,
What Good is Temporal Logic,
Information Processing 83, R.E.A. Mason editor, IFIP Congress series 1983: pp 657-668.

[19] A.J. Martin,
The Probe, An addition to communication primitives,
Information Processing Letters 20 (1985): pp 125-130.

[20] Mazurkiewicz,
Trace Theory,
Advances in Petri Nets 1986, part II, W. Brauer, W. Reisig, and G. Rozenberg editors, Lecture Notes in Computer Science 255, Springer: pp 279-324.

[21] J. van de Mortel-Fronczak,
Models of Trace Theory Systems,
Ph.D. thesis, scheduled for 1991/92, Eindhoven University of Technology.

[22] A. Pnueli,
Specification and Development of Reactive Systems,
Information Processing 86, H.J. Kugler editor, IFIP Congress series 1986: pp 845-858.

[23] M. Rem,
Trace theory and systolic computations,
Proceedings, Parallel Architectures and Languages Europe, June 1987, volume I, J.W. de Bakker, A.J. Nijman, and P.C. Treleaven editors, Lecture Notes in Computer Science 258, Springer 1987: pp 14-33.

[24] Jan L. A. van de Snepscheut,
Trace theory and VLSI design,
Ph.D. thesis, October 1983, Eindhoven University of Technology.
Also appeared as Lecture notes in computer science 200, Springer 1985.

[25] R.E. Tarjan,
Amortized Computational Complexity,
SIAM Journal Alg. Discrete Mathematics, Vol. 6, No. 2, April 1985: pp 306-318.

[26] J.T. Udding and T. Verhoeff,
Using a Partial Order and a Metric to Analyze a Recursive Trace Set Equation,
Technical Report, Department of Computer Science, Washington University in St. Louis WUCS-88-17.

[27] G. Zwaan,
Parallel Computations,
Ph.D. thesis, January 1989, Eindhoven University of Technology.

Glossary of Symbols of Chapter 2

Fat unary operators		Page(s)
a	alphabet of anything	13
b	base of schedules, processes, and enabling structures	12, 13, 20
d	delay of dependence functions and enabling structures	18, 20
e	external alphabet of (enabling) structures	18
f	functional behaviour of (enabling) structures	18
h	history of enabling structures	21
i	internal alphabet of (enabling) structures	18
n	enabling function of schedules and choice-free commands	21, 30
P	process of enabling structures	23

'Types'		Page
Asc	ascending enabling functions	34
Con	conservative enabling functions	35
Const	constant enabling functions	21
$\mathcal{ES}$	enabling structures	20
$\mathcal{EF}$	enabling functions	20
$\mathcal{N}$	normal enabling structures	51
$\mathcal{S}.A$	schedules over alphabet A	12
SC, SU, SD	scales, speeding ups, slowing downs	28
T	the time-domain: $\langle -\infty, \infty]$	12

Types of variables

		Page
a, b, c, d	action	12
A, B, C	alphabet	12
e, f, g, h	enabling function	20
E, F, G	enabling structure	20
$\mathcal{E}$	set of enabling structures or – functions	20
λ, μ	magnification and translation of scales, $0 < \lambda < \infty$, $-\infty < \mu < \infty$	28
M, N, O	moment in time	12
$\mathcal{P}, \mathcal{Q}$	process (or schedule-set)	13
$\mathcal{R}$	renaming	27
ρ, σ	scale	28
s, t, u, v	schedule	12
x, y, z	any	
X, Y, Z	any	

Miscellaneous

		Page(s)
$\varepsilon_A, \varepsilon$	empty schedule	13
sim	similarity	17, 20
$\odot, \oplus$	magnification and translation in scaling	28
$\lfloor$	until, restriction in time domain	14
$\|$	parallel composition	15, 23
$\shortparallel$	masking of enabling structures	23
$\uparrow$	extrapolation of schedules	36
$\upharpoonright, \backslash$	projection, hiding	14, 37
$\|\!\upharpoonright$	restriction of enabling structures	38
$\leqslant_{\mathcal{P}}, =_{\mathcal{P}}$	comparison over $\mathcal{P}$ of enabling structures	45
$[\ ]_R$	equivalence class modulo internal renaming	49
$[\ ]$	equivalence class or closure over $\approx$	53
$\mathcal{N}$	normal form of enabling structures	51
$\sim_M$	equality of behaviour up to moment M	43
$\approx_M$	equality of external behaviour up to M	43

Index

Samenvatting

Het enabling model kan worden gebruikt voor het analyseren en vergelijken van het real-time gedrag van een ruime klasse mechanismen. Het ondersteunt parallelle compositie, abstractie van interne acties en het afschermen van externe acties voor de omgeving. Ook 'identificatie' van acties en catenatie kunnen binnen het model beschreven worden.

Het model beschrijft mechanismen die parallel samenwerken met synchrone communicatie, maar *zonder* synchronisatie door middel van een globale klok. Gedacht kan worden aan VLSI programma's of realisaties hiervan op het niveau van 'handshake protocols'. De beschrijving van mechanismen is gebaseerd op het 'enabling concept': afhankelijk van het verleden verklaart een mechanisme zich bereid om acties uit te voeren; voor zover dit *interne* acties betreft worden ze uitgevoerd zodra het mechanisme bereid is, *externe* acties (communicaties) worden uitgevoerd zodra *zowel* het mechanisme als zijn omgeving *beide* bereid zijn tot uitvoering. We beperken ons tot deterministische mechanismen die niet 'disabelen', d.w.z. de bereidheid van een mechanisme om acties uit te voeren volgt eenduidig uit het 'verleden' en zodra een mechanisme bereid is om een (externe) actie uit te voeren, zal het mechanisme bereid *blijven* om deze actie uit te voeren totdat de actie daadwerkelijk is gebeurd. Binnen het model definiëren we parallelle compositie, abstractie van interne acties (projecteren) en het afschermen van externe acties voor de omgeving (maskeren).

Het model kan worden gebruikt voor het vergelijken van performance. Hierbij moet worden gedacht aan vragen als '*Is het ene mechanisme (een implementatie) tenminste zo snel als het andere (de specificatie)?*' en, meer algemeen, '*Wat is de snelheid van dit mechanisme ten opzichte van dat mechanisme?*'. In hoofdstuk 3 introduceren we een relatie, voor het vergelijken van performance, die geschikt is voor het compositioneel ontwerpen van mechanismen. Dat wil ondermeer zeggen dat deze relatie transitief is, en dat parallelle samenstelling monotoon is ten opzichte van deze relatie. Een opmerkelijk detail is dat de relatie slechts op de klasse der conservatieve gedragingen reflexief is. Een gevolg hiervan is dat de relatie slechts voor conservatieve specificaties zinvol is.

We illustreren het gebruik van het model aan de hand van twee case-studies. De eerste, in hoofdstuk 4, betreft het uitrekenen van een 'segment som' met behulp van een systolisch array. We bespreken verschillende aspecten van zo'n implementatie. De tweede, in hoofdstuk 5, betreft het gedistribueerd implementeren van FIFO (First In First Out) buffers. We leiden, onder verschillende condities, ondergrenzen af voor het aantal (extra) variabelen dat nodig is om FIFO buffers met een gegeven capaciteit te implementeren. Bovendien geven we implementaties die deze grenzen benaderen. Bij het implementeren van buffers met 'bypassing' maken we gebruik van niet-conservatieve 'cellen'. Dit leidt ertoe dat we —op lokaal niveau— andere correctheidscriteria, dan die van hoofdstuk 3, gebruiken.

Curriculum Vitae

Wim Kloosterhuis werd op zondag 20 mei 1962 geboren te Onstwedde. In Venray doorliep hij aan de scholengemeenschap 'Jerusalem' het Atheneum B.

Vanaf september 1980 studeerde hij Wiskunde (oude stijl) aan de Technische Hogeschool (later Universiteit) Eindhoven, waar hij koos voor de afstudeerrichting Informatica. Bij de Faculteit der Electrotechniek deed hij een zgn. grote stage op het gebied van adaptieve data-compressie algorithmen. Als afstudeerder bij professor Rem kreeg hij de opdracht om 'systolische arrays' te bestuderen. Dit evolueerde tot een studie van het begrip 'livelock', wat resulteerde in het afstudeerverslag 'Livelock in concurrent mechanisms'. Hierop studeerde hij in september 1987 met lof af.

In oktober 1987 verruilde hij zijn studiebeurs voor een AIO salaris. Als AIO bij de sectie Parallellisme en Architectuur kreeg hij wederom alle vrijheid om zelf zijn onderzoeksgebied te bepalen. Dit resulteerde uiteindelijk in een studie van de 'performance' van mechanismen, die zijn neerslag vond in dit proefschrift.

STELLINGEN

behorend bij het proefschrift

The Enabling Model

A Tool for Performance Analysis of Concurrent Mechanisms

van

Wim Kloosterhuis

Technische Universiteit Eindhoven,

september 1991

1. In [1] worden sequence functies gebruikt voor de performance analyse van *cubische* processen. Met de theorie in dit proefschrift kan worden aangetoond dat sequence functies betrouwbaar zijn bij het geven van mogelijke *externe* gedragingen van mechanismen die worden beschreven met *conservatieve* (trace-theorie) processen. (De cubische processen vormen hiervan een echte subset.) Sequence functies blijven echter een primitief hulpmiddel bij het bepalen van de performance.

   [1] G. Zwaan, *Parallel Computations*, proefschrift, januari 1989, TUE.

2. Voor een conservatief trace-theorie proces P (zie [2], confluent in de terminologie van [1]) geldt, voor alle s en t in P:

   $s \approx t \;\Rightarrow\; (\forall u : us \in P \wedge ut \in P : us \approx ut)$

   waar $s \approx t \;\Leftrightarrow\; (\forall v : : sv \in P \Leftrightarrow tv \in P)$.

   Deze implicatie bevestigt nog eens dat conservatieve processen 'behoudend' zijn.

   [1] J. Gunawardena,
   Causal Automata I: Confluence $\equiv$ *{AND,OR} Causality*,
   Semantics for Concurrency, Proceedings of the international BCS-FACS Workshop, July 1990, University of Leicester UK, Springer 1990: pp 137-156.

   [2] G. Zwaan, *Parallel Computations*, proefschrift, januari 1989, TUE.

3. Bij de beschrijving van mechanismen dient, naast kostenaspecten, het gedrag ten opzichte van de omgeving centraal te staan. In dit opzicht is het ongewenst om een onderscheid te maken tussen het uitsluiten van communicaties met de omgeving door 'activiteit' (livelock) enerzijds en door 'het ontbreken van activiteit' (refusals) anderzijds.

4. Systolische arrays waarin de cellen worden gesynchroniseerd door een globale klok, zijn per definitie slechts 'semi-systolisch'.

5. De 'elastic pipelines' in [1] zijn slechts in beperkte mate 'elastisch' te noemen. De timing is namelijk onafhankelijk van de complexiteit van de berekeningen die worden uitgevoerd. Hierdoor duren eenvoudige berekeningen even lang als meer ingewikkelde berekeningen.

   [1] Shinji Komori, et al.,
   An Elastic Pipeline Mechanism by Self-Timed Circuits,
   IEEE Journal of Solid-State Circuits, vol. 23. no. 1, February 1988: pp 111-117.

6. Het enabling concept kan worden uitgebreid door aan elke actie een deelverzameling van de tijd-as toe te kennen waarin de actie kan gebeuren (i.p.v. deelverzamelingen van de vorm $[M, \infty]$ als in dit proefschrift). Bij parallelle compositie moet dan de doorsnede van deze deelverzamelingen genomen worden. Deze benadering vindt zijn tegenhanger in de beschrijving van datacommunicatie in [1] en [2] (Sectie 2.3, communicatie in TNP).

   [1] M. Rem, *Trace theory and systolic computations*,
   Proceedings, Parallel Architectures and Languages Europe, June 1987, volume I, J.W. de Bakker, A.J. Nijman, and P.C. Treleaven editors, Lecture Notes in Computer Science 258, Springer 1987: pp 14-33.

   [2] J. Zwiers, *Compositionality, Concurrency and Partial Correctness*,
   proefschrift, februari 1988, TUE, *of*
   Lecture Notes in Computer Science 321, Springer 1989.

7. Voor een relatie die de snelheid van programma's vergelijkt is *transitiviteit* een handige eigenschap. Hoewel transitiviteit 'intuïtief' voor de hand ligt, mag ze niet als vanzelfsprekend worden aangenomen.

   Definieer bijvoorbeeld dat een programma P *sneller* is dan een programma Q, notatie $P < Q$, als P met een kans groter dan $1/2$ eerder termineert dan Q (onder bepaalde aannamen over de input). Mits de verzameling mogelijke gedragingen van programma's ruim genoeg gekozen wordt, is deze 'sneller dan' relatie *niet* transitief. Het is zelfs mogelijk een drietal gedragingen voor programma's P, Q en R te geven zodanig dat $P < Q$, $Q < R$ èn $R < P$.

8. Het alfabet van een proces is de verzameling van acties waarin het proces kan participeren. Een fundamentele eis die aan acties moet worden gesteld, is dat we er vrijelijk over kunnen beschikken. Dit wil ondermeer zeggen dat alfabetten vrijelijk kunnen worden hernoemd en dat er altijd 'verse' (ongebruikte) acties zijn. Indien alfabetten oneindig groot mogen zijn, kan de vereiste uitgebreidheid van het *universum* van acties, Ω, worden gerealiseerd door de maximale cardinaliteit van alfabetten $\aleph_i$ te kiezen, en de cardinaliteit van Ω $\aleph_{i+1}$. Doorgaans zal $i = 0$ volstaan.

   [1] G. Cantor, *Beiträge zur Begründung der tranfiniten Mengenlehre*, Georg Cantor, Gesammelte Abhandlungen Mathematischen und Philosophischen Inhalts, 1962, Georg Olms Verlagsbuchhandlung, Hildesheim: pp 282-356.

9. Er zijn precies tien gedragingen van 'general implementation relations' (definitie 3.1 in dit proefschrift) op de klasse der constante enabling functies.

10. De kwalificaties *vermomd plagiaat* (Rhapsodie in blue) en *melodisch leentjebuur* (pianoconcert in F) zeggen meer over de kwaliteiten van de criticus, [1], dan over de kwaliteiten van de bekritiseerde, George Gershwin.

    [1] Casper Höweler, *XYZ der Muziek*, 27^e druk, 1987, Unieboek Houten.

11. 'Mensenrechten' zijn voor ons erg belangrijk. Zo weegt *ons* recht op een goedkoop bakje koffie en op goedkope textiel nog steeds zwaarder dan het recht van een koffieplukker of textielarbeider op een menswaardig bestaan.

    [1] *Max Havelaar: Koffie met toekomst*, december 1989, Stichting Max Havelaar, Utrecht.

    [2] *STOF TOT NADENKEN: van wever tot keuken, eerlijke handel?*, 1990, S.O.S. Wereldhandel.