Control of Discrete Event Systems

Smedinga, Rein

Publication date:
1989

Citation for published version (APA):
Smedinga, R. (1989). *Control of Discrete Event Systems*. [S.n.].

Rijks*universiteit* Groningen

Control of Discrete Events

Proefschrift
ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus Dr. L.J. Engels
in het openbaar te verdedigen op
vrijdag 20 januari 1989
des namiddags te 2.45 uur precies

door

Reinder Smedinga

geboren op 9 november 1957
te Harlingen

We don't need no education
We don't need no thought control
No dark sarcasm in the classroom
Teachers leave the kids alone
Hey teachers leave us kids alone
All in all it's just another brick in the wall

Another brick in the wall, part two – The Wall

Aan mijn ouders

Acknowledgements

> *All alone, or in twos*
> *The ones who really love you*
> *Walk up and down outside the wall*
> *Some hand in hand*
> *Some gathered together in bands*
> *The bleeding hearts and the artists*
> *Make their stand*
> *And when they've given you their all*
> *Some stagger and fall, after all it's not easy*
> *Banging your heart against some mad buggers*
> *Wall*
>
> Outside the wall – The wall

This thesis could not be written if I did not have the opportunity to meet Jan Willems (who interested me in system theory) and Jan van de Snepscheut (who interested me in trace theory). I owe both Jans a lot of thanks for their help, advise, and encouragement.

Also, I thank the members of the committee to approve this thesis, Roland Back-house, Coen Bron, and Geert-Jan Olsder, for their careful reading of the manuscript.

Moreover I wish to thank everyone who contributed in the accomplishment of this thesis, especially Lenie and Agnes, for their moral support during the last stage of this work and Henderika, for being a "paranimf."

The quotations at the beginning of each chapter are songtexts from the popgroup Pink Floyd.[1] Below each quotation a reference is found to the corresponding song and album. The quotation of chapter 7 is from an individual project of Roger Waters. The front cover is a design of my own. The text is typeset using the work of Donald Knuth made more usable by Leslie Lamport: LaTeX.

---

[1] All quotations ©Pink Floyd Music publ. Ltd. / Chappell Music Ltd.

Inhoudsopgave

Introduction

This thesis is concerned with discrete systems. Instead of continuous systems in which variables are used that take numerical values, in discrete systems we use variables that have some logical or symbolic meaning. Discrete systems play a role in manufacturing, communication protocols, and (more in general) in all situations in which the occurrence of events, and especially the order in which events occur, is of importance.

World views

In discrete event systems we have to deal with discrete, asynchronous, and possibly non-deterministic actions. Discrete event systems appear in many different constitutions:

- sequential processes (doing a number of things in a row),
- concurrent processes (doing things concurrently, i.e., at the same time),
- operating systems (doing different things semi-parallel, i.e., via some form of inter-leaving),
- communication networks.

Different criteria can be considered for discussion. These criteria can be divided in two:

- quantitative criteria (mostly performance measures like throughput, cycle time, and so on),
- qualitative criteria (absence of deadlock, no infinite loops, fairness).

For quantitative criteria networks of queues, (timed) Petri nets and alike can be used. We mention:

- pertubation analysis of queueing networks (developed by Y.C.Ho, see for example [YCH1], [YCH2], and [YCH3]),
- linear systems in the max algebra (see [CDQV1] and [CDQV2]).

Here, we are concerned with the qualitative criteria, i.e., to ensure some orderly flow of events. Modelling this flow of events can be done by:

- finite automata and formal languages (e.g., the supervisory control theory of Wonham, see [CDFV], [InVa], [Rama], [RaWo], [WoRa], see also chapter 5),
- Petri Nets (see [JLP]),
- process algebra (e.g., the calculus of communicating systems (CCS) of Milner, see [Mil], or the theory of communicating sequential processes (CSP), see [Hoa]),
- temporal logic (see [ThWo]),
- trace theory.

Petri Nets can only be used to model a system, not to control it or to compute a controller. Both in CCS and CSP discrete processes are defined in a way very similar to the one in this thesis. However, both lack some formalism to control a discrete process. In chapter 8 we discuss some more aspects of CCS. Temporal logic is (merely) very difficult. No results have yet been found in the direction of control of discrete events. Supervisory control theory comes closest to what we have done here. It is discussed more thoroughly in chapter 5.

In this thesis discrete systems are defined using trace structures. Every symbol in a trace denotes an event, i.e., the occurrence of a discrete action. Although trace theory was developed in the context of concurrent programs and found useful in modelling electronic components (see [JvdS] and [JTU]), it seems to be the natural setting to model discrete events in general. Special attention will be given to the control of these systems. In order to be able to control a discrete process we will split its set of symbols in two classes: one class representing the events that should be controlled (and cannot be influenced directly from outside the system) and one class representing the events that are used to control (and can be influenced from outside the system).

Control of a discrete system is done by adding a second discrete system (the controller) and connecting both systems. This connecting mechanism can be defined using standard operators from trace theory. Special attention will be given to regular trace structures and, equivalently, the finite state machines. The notions deadlock, determinism, and distributed control will be considered in great detail.

Overview

In chapter 1 we give an overview of trace theory as it is needed in the sequel. We introduce state graphs in order to display trace structures graphically and we define regularity of trace structures. In chapter 2 we introduce our formalisation of discrete processes using trace structures. We define connection, joint behaviour, and concatenation of such processes and show a first outline of a (our) control problem. The given control problem is solved in chapter 3 using a number of tools. Chapter 4 deals with regular discrete processes and the relation to finite state machines. The control algorithm is reformulated in terms of finite state machines. Some related problems are discussed in chapter 5. Supervisory control theory of Wonham and Ramadge is compared with our control theory. In chapter 6 we discuss deadlock. We give a definition of deadlock and provide a method to find a deadlock-free controller solving the control problem. The events are given a direction in chapter 7, i.e., we make a distinction between generating an event and receiving one. We introduce inputs and outputs. A new kind of deadlock is introduced this way. In chapter 8 we discuss determinism and find conditions such that controllers are deadlock free. In chapter 9 we investigate distributed control and give some sufficient and some necessary conditions for our distributed control problem to be solvable.

Notation

Throughout this paper the following notation, not common to everyone, will be used:

$(\forall x : B(x) : C(x))$ is true if $C(x)$ holds for every x satisfying $B(x)$, e.g.,
$\quad(\forall x : x \in \mathbb{N} : x \geq 0)$
$(\exists x : B(x) : C(x))$ is true if there exists an x satisfying $B(x)$ for which $C(x)$ holds, e.g.,
$\quad(\exists x : x \in \mathbb{N} : x \geq 10 \wedge x \leq 20)$
$(\exists! x : B(x) : C(x))$ is true if there exists exactly one such an x, e.g.,
$\quad(\exists! x : x \in \mathbb{N} : x = 10)$
$\{x : B(x) : y(x)\}$ is the set constructor and denotes the set of all elements $y(x)$ constructed using elements x satisfying $B(x)$, e.g.,
$\quad\{n : n \in \mathbb{N} : a^n b^n\} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$

For equations (in proofs) we use the following notation:[2]

$$A$$
$$\Leftrightarrow \quad [\text{ hint why } A \Leftrightarrow B]$$
$$B$$
$$\Rightarrow \quad [\text{ hint why } B \Rightarrow C]$$
$$C$$

In the above sequence we have in fact written down the following:

$$((A \Leftrightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$$

If no hint is given, simple calculus is assumed.

A lot of operators will be used. A glossary is added in which all operators are explained and a page number is added referring to the page where the operator is defined. The binding power of the operators can be found in the glossary also.

---

[2] Adapted from [EWD2].

Trace theory

In this chapter we give the basic notions on which the material in the sequel is based, i.e., we discuss some of the aspects of trace theory.

We explain trace structures by using it as a way to model electronical components. In following chapters we make clear that trace structures can as well be used to model more general discrete event systems.

1.1 Trace structures

We consider components as being connected by means of channels. Think of a channel as some kind of wire to send messages along it. Components can communicate with each other by sending messages along those channels. At this stage, we are not concerned with the contents of such a message, just with the sending (and receiving) itself. Of great importance is the order in which messages are sent. We assume that sending a message and receiving a message is in fact one action without any direction, i.e., a channel either contains or does not contain a message.

Each channel is represented by a symbol. The set of all channels connected to that component is called the *alphabet* of that component. The behaviour of the component is modelled by giving all possible finite sequences of symbols. Such a sequence is called a *trace* and a (possibly infinite) set of traces is called the *trace set*. Consider a component with two channels, say a and b, then a possible behaviour could be $abab$, meaning that communication is performed along channel a, channel b, channel a, and channel b in that order.

The pair consisting of the alphabet A and the trace set S is called a *trace structure* and denoted by

$$T = \langle S, A \rangle$$

Example 1.1 With

$$
\begin{aligned}
T &= \langle \{ n : n \in \mathbb{N} : (ab)^n \}, \{a, b\} \rangle \\
&= \langle \{ \epsilon, ab, abab, ababab, \ldots \}, \{a, b\} \rangle
\end{aligned}
$$

we denote a trace structure with channels a and b and possible behaviour given by $(ab)^*$. Here, ab denotes concatenation: first a, next b, and $*$ (the Kleene-star) denotes finite repetition (zero or more times). The notation ϵ stands for the empty string (representing the nothing-has-happened-behaviour).

T can be considered as a one place buffer, with a the event *put an element in the buffer* and b the event *get an element from the buffer*. Notice that the number of occurrences of b is at most the number of occurrences of a, so no element is taken from the buffer if an element is not first put into it.
(end of example)

We introduce two operators in order to obtain the alphabet and the trace set of some given trace structure:

Definition 1.2 *For trace structure $T = \langle S, A \rangle$ the operators* $\mathbf{t}$ *and* $\mathbf{a}$ *are defined by:*

$$
\mathbf{a}T = A \qquad \mathbf{t}T = S
$$

Example 1.3 In the previous example we have:

$$
\begin{aligned}
\mathbf{a}T &= \{a, b\} \\
\mathbf{t}T &= \{ n : n \in \mathbb{N} : (ab)^n \}
\end{aligned}
$$

(end of example)

1.1.1 Relation to general dynamical systems

In [JCW] a general definition is given of a dynamical system $\Sigma = (T, W, B)$ with

T	time axis (normally $T \subseteq \mathbb{R}$ or $T \subseteq \mathbb{Z}$)
W	signal alphabet (some abstract set)
$B \subseteq W^T$	the behaviour

This definition is (to quote Jan Willems) "hopelessly general but nevertheless it captures rather well the crucial features of the notion of a dynamical system." For example, an n-dimensional continuous system can be modeled this way with $W \subseteq \mathbb{R}^n$ and $B \subseteq (\mathbb{R}^n)^T$ a set of n-dimensional functions of t satisfying some (physical) laws and describing the system.

A trace structure S is a special kind of a dynamical system Σ with[1]

$$
\begin{aligned}
T &= \mathbb{N} \\
W &= \mathbf{a}S \cup \{\square\} \\
B &= \{ w : (\exists t_e : t_e \in \mathbb{N} : w[0..t_e) \in \mathbf{t}S \ \wedge \ (\forall t : t \geq t_e : w(t) = \square)) : w \}
\end{aligned}
$$

The blanks ($\square$) are needed for cosmetic reasons: B contains only infinite strings, while $\mathbf{t}S$ contains only finite strings, which have to be completed by adding blanks at the end to fit the definition.

The time index t is interpreted here as logic time (i.e., it parametrizes the order of events) and not as clocktime, which is the usual case in conventional system theory.

---

[1] $w[0..t_e)$ stands for the string $w[0]w[1] \ldots w[t_e - 1]$.

1.2 Alphabet restriction

We are not always interested in the communication along all the channels. In that case, it is possible to restrict a trace t to some subalphabet A, denoted by $t\lceil A$.

Definition 1.4 *Alphabet restriction* $\lceil$ *is defined on a trace by:*

$$
\begin{aligned}
\epsilon\lceil A &= \epsilon \\
(ta)\lceil A &= t\lceil A & \text{if } a \notin A \\
&= (t\lceil A)a & \text{if } a \in A
\end{aligned}
$$

and on trace structures by:

$$T\lceil A = \langle \{t : t \in \mathbf{t}T : t\lceil A\}, \mathbf{a}T \cap A \rangle$$

We write $ta\lceil A$ instead of $(ta)\lceil A$ to save parentheses.

Example 1.5

$$
\begin{aligned}
\langle \{abc\}, \{a,b,c\}\rangle\lceil\{b,c\} &= \langle\{bc\},\{b,c\}\rangle \\
\langle\{ababab\},\{a,b\}\rangle\lceil\{c\} &= \langle\{\epsilon\},\varnothing\rangle \\
\langle\{abc,acb\},\{a,b,c\}\rangle\lceil\{a,b\} &= \langle\{ab\},\{a,b\}\rangle
\end{aligned}
$$

(end of example)

1.3 Connection of trace structures

A finite number of trace structures can be connected. First, we explain connection using an example; next, we give a formal definition.

Example 1.6 Let T and U be as follows:

$$T = \langle\{abcd\},\{a,b,c,d\}\rangle \quad U = \langle\{eace\},\{a,c,e\}\rangle$$

The connection of T and U results in the following trace structure S:

$$S = \langle\{eabcde, eabced\},\{a,b,c,d,e\}\rangle$$

If we look at the traces of the connection S and delete all events, that do not belong to T, all these traces should belong to T. Also deleting all events not belonging to U leads to traces belonging to U, i.e.,

$$eabcde\lceil\mathbf{a}T = abcd \wedge abcd \in \mathbf{t}T \quad eabcde\lceil\mathbf{a}U = eace \wedge eace \in \mathbf{t}U$$

and also:

$$eabcde\lceil\mathbf{a}T = abcd \wedge abcd \in \mathbf{t}T \quad eabcde\lceil\mathbf{a}U = eace \wedge eace \in \mathbf{t}U$$

So the connection of T and U contains all traces with the property that, if we restrict those traces to the alphabet of one of the trace structures, we get traces that belong to that structure.
(end of example)

Connection in this sense is in fact a shuffling, where identical communications occur simultaneously. This kind of operation is called *weaving* and is defined formally in section 1.3.1. If communications that take place in the connection are no longer important, we

can omit them. In that case our previous example ends in *ebde* or *ebed*. This kind of connection is called *blending* and is explained in section 1.3.2.

1.3.1 Weaving

Definition 1.7 *The weaving* **w** *of two trace structures T and S is defined by*

$$T \text{ } \mathbf{w} \text{ } S$$
$$=$$
$$\langle \{x : x \in (\mathbf{a}T \cup \mathbf{a}S)^* \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\}, \mathbf{a}T \cup \mathbf{a}S \rangle$$

Example 1.8 Consider

$$T = \langle \{\epsilon, a, ab\}, \{a, b\} \rangle \quad S = \langle \{\epsilon, c, cb\}, \{b, c\} \rangle$$

Then $T \text{ } \mathbf{w} \text{ } S = \langle \{\epsilon, a, c, ac, ca, acb, cab\}, \{a, b, c\} \rangle$, e.g., $acb\lceil \mathbf{a}T = ab$ and $ab \in \mathbf{t}T$ and also $acb\lceil \mathbf{a}S = ac$ and $ac \in \mathbf{t}S$, hence $acb \in \mathbf{t}(T \text{ } \mathbf{w} \text{ } S)$.
(end of example)

The operator **w** defines a binary operation on the set of trace structures. It has some nice properties, which are listed below:

Property 1.9 *For general trace structures T, U, and S, the following hold:*

(1) *weaving is symmetric:*
   $T \text{ } \mathbf{w} \text{ } S = S \text{ } \mathbf{w} \text{ } T$
(2) *the structure $\langle \{\epsilon\}, \emptyset \rangle$ is the unit element:*
   $T \text{ } \mathbf{w} \text{ } \langle \{\epsilon\}, \emptyset \rangle = T$
(3) *the structure $\langle \emptyset, \emptyset \rangle$ is the zero element:*
   $T \text{ } \mathbf{w} \text{ } \langle \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset \rangle$
(4) *weaving is idempotent:*
   $T \text{ } \mathbf{w} \text{ } T = T$
(5) *weaving is associative:*
   $(T \text{ } \mathbf{w} \text{ } S) \text{ } \mathbf{w} \text{ } U = T \text{ } \mathbf{w} \text{ } (S \text{ } \mathbf{w} \text{ } U)$

1.3.2 Blending

Sometimes, we are only interested in those communications that are not common, i.e., belong to only one of the trace structures. Then we use *blending*.

Definition 1.10 *The blending* **b** *of two trace structures T and S is defined by*

$$T \text{ } \mathbf{b} \text{ } S$$
$$=$$
$$\langle \{x : x \in (\mathbf{a}T \cup \mathbf{a}S)^* \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\lceil (\mathbf{a}T \div \mathbf{a}S)\}, \mathbf{a}T \div \mathbf{a}S \rangle$$

The operator $\div$ stands for symmetric set difference, i.e.,

$$A \div B = (A \cup B) \setminus (A \cap B)$$

Example 1.11 Reconsider T and S from example 1.8, then:

$$T \mathbf{\ b\ } S = \langle \{\epsilon, a, c, ac, ca\}, \{a, c\} \rangle$$

(end of example)

Also the operator **b** is a binary operator on the set of trace structures. Some properties of the blend are listed below:

Property 1.12 *For general trace structures T, U, and S, the following hold:*

(1) *blending is symmetric:*
   $T \mathbf{\ b\ } S = S \mathbf{\ b\ } T$
(2) *the structure $\langle \{\epsilon\}, \emptyset \rangle$ is the unit element:*
   $T \mathbf{\ b\ } \langle \{\epsilon\}, \emptyset \rangle = T$
(3) *the structure $\langle \emptyset, \emptyset \rangle$ is the zero element:*
   $T \mathbf{\ b\ } \langle \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset \rangle$
(4) *blending is not idempotent:*
   $\mathbf{t} T \neq \emptyset \ \Rightarrow\ T \mathbf{\ b\ } T = \langle \{\epsilon\}, \emptyset \rangle$

Blending is not associative. However, we have:

Property 1.13 *Blending is associative if every symbol occurs in at most two of the alphabets.*

Next, we list a few properties that are needed further on:

Property 1.14 *For trace structures T and S with $\mathbf{a}S \subseteq \mathbf{a}T$, the following properties hold:*

(1) $T \mathbf{\ w\ } S \ = \ \langle \{x : x \in \mathbf{t}T \ \wedge\ x \lceil \mathbf{a}S \in \mathbf{t}S : x\}, \mathbf{a}T \rangle$
(2) $T \mathbf{\ b\ } S \ = \ \langle \{x : x \in \mathbf{t}T \ \wedge\ x \lceil \mathbf{a}S \in \mathbf{t}S : x \lceil (\mathbf{a}T \setminus \mathbf{a}S)\}, \mathbf{a}T \setminus \mathbf{a}S \rangle$
(3) $T \mathbf{\ b\ } S \ = \ \langle \{x : x \in \mathbf{t}T \lceil (\mathbf{a}T \setminus \mathbf{a}S) \ \wedge$
   $\qquad\qquad (\exists y : y \in \mathbf{t}T \ \wedge\ y \lceil \mathbf{a}S \in \mathbf{t}S : y \lceil (\mathbf{a}T \setminus \mathbf{a}S) = x)$
   $\qquad\qquad : x\}$
   $\qquad , \mathbf{a}T \setminus \mathbf{a}S$
   $\qquad \rangle$
   $= \ \langle \{x : x \in \mathbf{t}T \lceil (\mathbf{a}T \setminus \mathbf{a}S) \ \wedge$
   $\qquad\qquad (\exists y : y \in \mathbf{t}T \ \wedge\ y \lceil (\mathbf{a}T \setminus \mathbf{a}S) = x : y \lceil \mathbf{a}S \in \mathbf{t}S)$
   $\qquad\qquad : x\}$
   $\qquad , \mathbf{a}T \setminus \mathbf{a}S$
   $\qquad \rangle$

1.4 Ordering of trace structures

In the sequel we use the following partial ordering:

Definition 1.15 *For two trace structures T and S, the ordering $T \subseteq S$ is defined by:*

$$\mathbf{a}T = \mathbf{a}S \ \wedge\ \mathbf{t}T \subseteq \mathbf{t}S$$

We define T to be *at most S* if the trace set of T is at most the trace set of S and if T and S have equal alphabets. We do not define any ordering on trace structures with unequal alphabets.

Property 1.16 *For trace structures T, S_1, and S_2 with $\mathbf{a}S_i \subseteq \mathbf{a}T$ (for $i = 1, 2$), we have:*

(1) $S_1 \subseteq S_2 \Rightarrow (T \mathbf{\ b\ } S_1) \subseteq (T \mathbf{\ b\ } S_2)$
(2) $S_1 \subseteq S_2 \Rightarrow (T \mathbf{\ w\ } S_1) \subseteq (T \mathbf{\ w\ } S_2)$

The following lemma plays a role in finding a suitable controller in chapter 3.

Lemma 1.17 *For trace structures T and S with $\mathbf{a}S \subseteq \mathbf{a}T$ and $S \subseteq T{\upharpoonright}\mathbf{a}S$, the following holds:*

$$T \mathbf{\ b\ } (T \mathbf{\ b\ } S) \supseteq S$$

proof: Use $A = \mathbf{a}T \setminus \mathbf{a}S$, then

$\mathbf{t}(T \mathbf{\ b\ } S)$
$=$ [see proposition 1.14 (2)]
$\{x : x \in \mathbf{t}T \ \wedge \ x{\upharpoonright}\mathbf{a}S \in \mathbf{t}S : x{\lceil}A\}$

Furthermore:

$x{\lceil}\mathbf{a}S \in \mathbf{t}S \ \wedge \ x \in \mathbf{t}T$
$\Rightarrow$ [take $y = x$]
$(\exists y : y \in \mathbf{t}T \ \wedge \ y{\lceil}\mathbf{a}S \in \mathbf{t}S : y{\lceil}A = x{\lceil}A)$

Hence:

$\mathbf{t}(T \mathbf{\ b\ } (T \mathbf{\ b\ } S))$
$=$ [see proposition 1.14 (2)]
$\{x : x \in \mathbf{t}T \ \wedge \ x{\lceil}A \in \mathbf{t}(T \mathbf{\ b\ } S) : x{\lceil}\mathbf{a}S\}$
$=$ [equality above]
$\{x : x \in \mathbf{t}T \ \wedge \ (\exists y : y \in \mathbf{t}T \ \wedge \ y{\lceil}\mathbf{a}S \in \mathbf{t}S : y{\lceil}A = x{\lceil}A) : x{\lceil}\mathbf{a}S\}$
$\supseteq$ [implication above]
$\{x : x \in \mathbf{t}T \ \wedge \ x{\lceil}\mathbf{a}S \in \mathbf{t}S : x{\lceil}\mathbf{a}S\}$
$=$ [$S \subseteq T{\upharpoonright}\mathbf{a}S$]
$\mathbf{t}S$

(end of proof)

In general, it is not true that $T \mathbf{\ b\ } (T \mathbf{\ b\ } S) = S$ as the following example shows.

Example 1.18

$$T = \langle \{ac, ad\}, \{a, c, d\} \rangle$$
$$S = \langle \{c\}, \{c, d\} \rangle$$

Then we have:

$$\mathbf{t}(T \mathbf{\ b\ } (T \mathbf{\ b\ } S))$$
$$=$$
$$\mathbf{t}(T \mathbf{\ b\ } \langle \{a\}, \{a\} \rangle)$$
$$=$$
$$\{c, d\}$$
$$\neq$$
$$\mathbf{t}S$$

The inequality is due to the fact that we can find x and y (both $\in \mathbf{t}T$) with:

$$x \lceil (\mathbf{a}T \setminus \mathbf{a}S) = y \lceil (\mathbf{a}T \setminus \mathbf{a}S) \ \wedge \ y \lceil \mathbf{a}S \in \mathbf{t}S \ \wedge \ x \lceil \mathbf{a}S \notin \mathbf{t}S$$

Here $x = ad$ and $y = ac$.

(end of example)

1.5 Other operators on trace structures

We use set operators on trace structures as well, using the following definition:

Definition 1.19 _Given two trace structures T and S, then we define the_ union _of T and Q, denoted $T \cup S$ (pronounce "T or S"), by_

$$\langle \mathbf{t}T \cup \mathbf{t}S, \mathbf{a}T \cup \mathbf{a}S \rangle$$

and if $\mathbf{a}T = \mathbf{a}S$, then we define the intersection _of T and S, denoted $T \cap S$ (pronounce "T and S"), by_

$$\langle \mathbf{t}T \cap \mathbf{t}S, \mathbf{a}T \rangle$$

and the exclusion _of T and S, denoted $T \setminus S$ (pronounce "T without S"), by_

$$\langle \mathbf{t}T \setminus \mathbf{t}S, \mathbf{a}T \rangle$$

Intersection and exclusion can also be defined in case the alphabets are not equal, but the definitions as given here are all we need further on. We conclude this section with a number of properties.

Property 1.20 *For trace structures T and S, we have:*

(1) $\mathbf{a}T = \mathbf{a}S \;\Rightarrow\; (T\ \mathbf{w}\ S) = T \cap S$
(2) $\mathbf{a}S \subseteq \mathbf{a}T \;\Rightarrow\; (T\ \mathbf{w}\ S)\lceil \mathbf{a}S = (T\lceil \mathbf{a}S) \cap S$

proof: Part (1): see property 1.20 in [JvdS].
Part (2):

$\quad x \in \mathbf{t}(T\lceil \mathbf{a}S \cap S)$
$\Leftrightarrow$
$\quad x \in \mathbf{t}T\lceil \mathbf{a}S \;\wedge\; x \in \mathbf{t}S$
$\Leftrightarrow \qquad [\text{ definition of } \lceil\ ]$
$\quad (\exists y : y \in \mathbf{t}T : y\lceil \mathbf{a}S = x) \;\wedge\; x \in \mathbf{t}S$
$\Leftrightarrow$
$\quad (\exists y : y \in \mathbf{t}T \;\wedge\; y\lceil \mathbf{a}S \in \mathbf{t}S : x = y\lceil \mathbf{a}S)$
$\Leftrightarrow \qquad [\text{ definition of } \mathbf{w}\ ]$
$\quad (\exists y : y \in \mathbf{t}(T\ \mathbf{w}\ S) : x = y\lceil \mathbf{a}S)$
$\Leftrightarrow \qquad [\text{ definition of } \lceil\ ]$
$\quad x \in (T\ \mathbf{w}\ S)\lceil \mathbf{a}S$

(end of proof)

Property 1.21 *For trace structures T and U and alphabet A, we have:*

(1) $(T\ \mathbf{w}\ U)\lceil A \quad \subseteq \quad T\lceil A\ \mathbf{w}\ U\lceil A$
(2) $(T\ \mathbf{b}\ U)\lceil A \quad \subseteq \quad T\lceil A\ \mathbf{b}\ U\lceil A$
(3) $\mathbf{a}T \cap \mathbf{a}U \subseteq A \;\Rightarrow\; (T\ \mathbf{w}\ U)\lceil A = T\lceil A\ \mathbf{w}\ U\lceil A$
(4) $\mathbf{a}T \cap \mathbf{a}U \subseteq A \;\Rightarrow\; (T\ \mathbf{b}\ U)\lceil A = U\lceil A\ \mathbf{b}\ U\lceil A$

proof: See properties 1.15, 1.16, and 1.31 in [JvdS].

Corrollary 1.22

$\quad (T_1\ \mathbf{w}\ T_2)\lceil \mathbf{a}T_1 \subseteq T_1$

proof:

$\quad (T_1\ \mathbf{w}\ T_2)\lceil \mathbf{a}T_1$
$\subseteq \qquad [\text{ property 1.21 (1) }]$
$\quad T_1\lceil \mathbf{a}T_1\ \mathbf{w}\ T_2\lceil \mathbf{a}T_1$
$= \qquad [\text{ property 1.20 (1) }]$
$\quad T_1 \cap T_2\lceil \mathbf{a}T_1$
$\subseteq$
$\quad T_1$

(end of proof)

Property 1.23 *For trace structures T, S, and U, with $\mathbf{a}T = \mathbf{a}S$, we have:*

(1) $U \mathbf{w} (T \cup S)$ $=$ $(U \mathbf{w} T) \cup (U \mathbf{w} T)$
(2) $U \mathbf{w} (T \cap S)$ $=$ $(U \mathbf{w} T) \cap (U \mathbf{w} T)$
(3) $U \mathbf{b} (T \cup S)$ $=$ $(U \mathbf{b} T) \cup (U \mathbf{b} S)$
(4) $U \mathbf{b} (T \cap S)$ $\subseteq$ $(U \mathbf{b} T) \cap (U \mathbf{b} S)$
(5) $U \mathbf{w} (T \setminus S)$ $=$ $(U \mathbf{w} T) \setminus (U \mathbf{w} S)$
(6) $U \mathbf{b} (T \setminus S)$ $=$ $(U \mathbf{b} T) \setminus (U \mathbf{b} S)$

proof: Parts (1) to (4): see properties 1.21 and 1.34 in [JvdS].
Part (5):

$\qquad U \mathbf{w} (T \setminus S)$
$=\qquad$ [definition of $\setminus$ and $\mathbf{a}T = \mathbf{a}S$]
$\qquad U \mathbf{w} \langle \mathbf{t}T \setminus \mathbf{t}S, \mathbf{a}T \rangle$
$=\qquad$ [definition of $\mathbf{w}$]
$\qquad \langle \{ x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \wedge x{\upharpoonright}\mathbf{a}U \in \mathbf{t}U \wedge x{\upharpoonright}\mathbf{a}T \in \mathbf{t}T \setminus \mathbf{t}S : x \}$
$\qquad , \mathbf{a}T \cup \mathbf{a}U$
$\qquad \rangle$
$=\qquad$ [$\mathbf{a}T = \mathbf{a}S$]
$\qquad \langle \{ x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \wedge x{\upharpoonright}\mathbf{a}U \in \mathbf{t}U \wedge x{\upharpoonright}\mathbf{a}T \in \mathbf{t}T : x \} \setminus$
$\qquad \quad \{ x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \wedge x{\upharpoonright}\mathbf{a}U \in \mathbf{t}U \wedge x{\upharpoonright}\mathbf{a}S \in \mathbf{t}S : x \}$
$\qquad , \mathbf{a}T \cup \mathbf{a}U$
$\qquad \rangle$
$=\qquad$ [definition of $\setminus$ and $\mathbf{a}T = \mathbf{a}S$]
$\qquad \langle \{ x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \wedge x{\upharpoonright}\mathbf{a}U \in \mathbf{t}U \wedge x{\upharpoonright}\mathbf{a}T \in \mathbf{t}T : x \}, \mathbf{a}T \cup \mathbf{a}U \rangle \setminus$
$\qquad \langle \{ x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \wedge x{\upharpoonright}\mathbf{a}U \in \mathbf{t}U \wedge x{\upharpoonright}\mathbf{a}S \in \mathbf{t}S : x \}, \mathbf{a}S \cup \mathbf{a}U \rangle$
$=\qquad$ [definition of $\setminus$]
$\qquad (U \mathbf{w} T) \setminus (U \mathbf{w} S)$

Part (6) is similar.
(end of proof)

Property 1.24 *For trace structures T and U and alphabet A, we have:*

(1) $(T{\upharpoonright}A) \setminus (U{\upharpoonright}A)$ $\subseteq$ $(T \setminus U){\upharpoonright}A$
(2) $(T \cup U){\upharpoonright}A$ $\subseteq$ $(T{\upharpoonright}A) \cup (U{\upharpoonright}A)$
(3) $(T \cap U){\upharpoonright}A$ $\subseteq$ $(T{\upharpoonright}A) \cap (U{\upharpoonright}A)$

proof: Part (1):

$\qquad x \in \mathbf{t}((T{\upharpoonright}A) \setminus (U{\upharpoonright}A))$
$\Leftrightarrow\qquad$ [definition of $\setminus$ and ${\upharpoonright}$]
$\qquad (\exists y : y \in \mathbf{t}T : y{\upharpoonright}A = x) \wedge (\forall y : y \in \mathbf{t}U : y{\upharpoonright}A \neq x)$
$\Leftrightarrow$
$\qquad (\exists y : y \in \mathbf{t}T : y{\upharpoonright}A = x) \wedge (\forall y : y{\upharpoonright}A = x : y \notin \mathbf{t}U)$
$\Rightarrow$
$\qquad (\exists y : y \in \mathbf{t}T \wedge y \notin \mathbf{t}U : y{\upharpoonright}A = x)$
$\Leftrightarrow\qquad$ [definition of $\setminus$ and ${\upharpoonright}$]
$\qquad x \in \mathbf{t}(T \setminus U){\upharpoonright}A$

Part (2):

$x \in \mathbf{t}(T \cup U)\lceil A$
$\Leftrightarrow$ [definition of $\lceil$]
$(\exists y : y \in \mathbf{t}(T \cup U) : y\lceil A = x)$
$\Rightarrow$ [definition of $\cup$]
$(\exists y : y \in \mathbf{t}T : y\lceil A = x) \vee (\exists y : y \in \mathbf{t}U : y\lceil A = x)$
$\Leftrightarrow$ [definition of $\cup$ and $\lceil$]
$x \in \mathbf{t}(T\lceil A) \cup (T\lceil A)$

Part (3) is similar.
(end of proof)

We do not have equality:

Example 1.25 Consider $A = \{a\}$ and

$T = \langle\{ab, ac\}, \{a, b, c\}\rangle \quad U = \langle\{ab, bc\}, \{a, b, c\}\rangle$

then

$T\lceil A = \langle\{a\}, \{a\}\rangle$
$U\lceil A = \langle\{a\}, \{a\}\rangle$
$(T \setminus U)\lceil A = \langle\{ac\}, \{a, b, c\}\rangle\lceil A = \langle\{a\}, \{a\}\rangle$
$(T\lceil A) \setminus (U\lceil A) = \langle\emptyset, \{a\}\rangle$

(end of example)

1.6   State graph representation

In conventional system theory a dynamical system can be described in a state space form. A number of internal variables are used to serve as a memory for the system. We can do the same with trace structures, which leads to state graphs.[2]

To do so, we need the notion *prefix closure* and a equivalence relation **E** on trace structures.

Definition 1.26 *The* prefix closure *of a trace set S is defined by:*

$\mathbf{pref}(S) = \{x : (\exists z :: xz \in S) : x\}$

and the prefix closure of a trace structure T by:

$\mathbf{pref}(T) = \langle\mathbf{pref}(\mathbf{t}T), \mathbf{a}T\rangle$

The binary relation $\mathbf{E} : (\mathbf{a}T)^* \times (\mathbf{a}T)^* \to \mathbf{bool}$ *induced by T on two traces is defined by:*

$x \mathbf{E} y \Leftrightarrow (\forall z : z \in (\mathbf{a}T)^* : (xz \in \mathbf{t}T) = (yz \in \mathbf{t}T))$

We have $\mathbf{pref}(\mathbf{t}T) = \mathbf{t}(\mathbf{pref}(T))$. For cosmetic reasons we prefer to write $\mathbf{pref}(\mathbf{t}T)$. If $x \mathbf{E} y$, then x and y have the same continuations.

Property 1.27 *The binary relation* **E** *is an equivalence relation on* $(\mathbf{a}T)^*$.

---

[2]State graphs as defined here correspond to the so-called *evolution laws* (a special kind of a dynamical system in state space form as defined in [JCW]).

Definition 1.28 *The* equivalence classes *corresponding to the relation* **E** *induced by T are denoted by*

$$[x]_T = \{y : x \ \mathbf{E} \ y : y\}$$

When there is no confusion we simply write $[x]$.

The equivalence classes of T are called the *states* of T. The set of all equivalence classes of T will be denoted by $\mathbf{E}(T)$. It should be clear that every trace structure T with non-empty trace set has a state $[\epsilon]$. This state is called the *initial state* of T.

All traces that do not belong to the trace set of T belong to the same equivalence class. This equivalence class is denoted by $[\varnothing]$ and called the *error state* induced by T:

Definition 1.29 *The* error state *induced by the trace structure T is denoted by* $[\varnothing]_T$ *and defined by*

$$[\varnothing]_T = \{x : x \notin \mathbf{t}T : x\}$$

When there is no confusion we write $[\varnothing]$.

If $\mathbf{t}T = \varnothing$, we have that $[\varnothing] = [\epsilon]$. If $\mathbf{t}T = (\mathbf{a}T)^*$, we have $[\varnothing] = \varnothing$ and $[\epsilon] = \mathbf{t}T$. In all other cases a trace structure has at least two states: $[\epsilon]$ and $[\varnothing]$.

Definition 1.30 *The* final states *induced by a trace structure T are denoted by* $\mathbf{F}(T)$ *and defined by*

$$\mathbf{F}(T) = \{p : p \in \mathbf{E}(T) \ \wedge \ (\exists x : x \in p : x \in \mathbf{t}T) : p\}$$

Example 1.31 Reconsider:

$$T = \langle \{n : n \in \mathbb{N} : (ab)^n\}, \{a, b\} \rangle$$

T induces three equivalence classes. We have:

$$
\begin{array}{llll}
[\epsilon] & = & \{n : n \in \mathbb{N} : (ab)^n\} & \mathbf{E}(T) = \{[\epsilon], [a], [\varnothing]\} \\
[a] & = & \{n : n \in \mathbb{N} : (ab)^n a\} & \mathbf{F}(T) = \{[\epsilon]\} \\
[\varnothing] & = & (\mathbf{a}T)^* \setminus ([\epsilon] \cup [a]) &
\end{array}
$$

$[\epsilon]$ is the initial state of T as well as the only final state. The states $[a]$ and $[\varnothing]$ are no final states. The error state is never a final state and $[a]$ is not a final state because $(\forall x : x \in [a] : x \notin \mathbf{t}T)$.
(end of example)

The states of T (i.e., $\mathbf{E}(T)$) can be seen as cells in which some information is held: just knowing the present state of T gives enough information about the past (i.e., about all communication done thus far) to be able to tell what communication is allowed to occur next according to the corresponding trace structure. The states of T therefore serve as a memory to T.

The states of a trace structure, as defined above, have the property that occurrence of a communication in some state brings the trace structure in one (new) unique state:

Property 1.32

$$(\forall a, p : a \in \mathbf{a}T \ \wedge \ p \in \mathbf{E}(T) : (\exists! q : q \in \mathbf{E}(T) : (\forall x : x \in p : xa \in q)))$$

A trace of T can be seen as a path through the states of T. We therefore introduce a *state transition function* δ describing such a path.

Definition 1.33 *For* $p, q \in \mathbf{E}(T)$, *and* $a \in \mathbf{a}T$, *the state transition function* δ *is the function* $\delta : \mathbf{E}(T) \times \mathbf{a}T \to \mathbf{E}(T)$ *defined by:*

$$\delta(p, a) = q$$

$\Leftrightarrow$

$$(\exists x, y : x \in p \ \wedge \ y \in q : xa = y)$$

This is indeed a definition, because from property 1.32 we know that $q = \delta(p, a)$ is a unique state.

As a consequence of the definition of the binary relation $\mathbf{E}$ on $(\mathbf{a}T)^*$ the transition function δ is a totally defined function.

Example 1.34 In example 1.31 δ equals:

$$\delta([\epsilon], a) = [a] \quad \delta([a], a) = [\varnothing] \quad \delta([\varnothing], a) = [\varnothing]$$
$$\delta([\epsilon], b) = [\varnothing] \quad \delta([a], b) = [\epsilon] \quad \delta([\varnothing], b) = [\varnothing]$$

(end of example)

From the state transition function δ we can derive a path x from p to q if $q = \delta^*(p, x)$, with δ^* the *closure* of δ defined by:

Definition 1.35 *For* $x, y \in (\mathbf{a}T)^*$, *and* $a \in \mathbf{a}T$, *the* closure *of the transition function* δ *is the function* $\delta^* : \mathbf{E}(T) \times (\mathbf{a}T)^* \to \mathbf{E}(T)$ *defined by:*

$$\delta^*(p, \epsilon) \ = \ p$$
$$\delta^*(p, x) \ = \ \delta^*(\delta(p, a), y) \quad \textit{if } x = ay$$

Normally the closure of the transition function is simply also denoted by δ.

A string x from $(\mathbf{a}T)^*$ is called *an accepting string* if $\delta([\epsilon], x) \in \mathbf{F}(T)$. The accepting strings are exactly the traces of T.

With the above definitions we have introduced a state graph for a trace structure.

Definition 1.36 *A state graph M is defined by:*

$$M = (A, Q, d, q_0, F)$$

with:

A	*the alphabet (a finite set of symbols)*
Q	*the states of the graph*
$d : Q \times A \to Q$	*the state transition function*
$q_0 \in Q$	*the initial state*
$F \subseteq Q$	*the final states*

Formally the following analogue between trace structures and state graphs exists:
With a trace structure T corresponds a state graph given by:

$$\mathbf{sg}(T) = (\mathbf{a}T, \mathbf{E}(T), \delta, [\epsilon], \mathbf{F}(T))$$

with $\mathbf{E}$ and $\mathbf{F}$ as given above and δ such that

$$(\forall a, x : a \in \mathbf{a}T \ \wedge \ x \in (\mathbf{a}T)^* : \delta([x], a) = [xa])$$

With each state graph M corresponds the trace structure given by:

$$\mathbf{ts}(M) = \langle A, \{x : x \in A^* \ \wedge \ d^*(q_0, x) \in F\} \rangle$$
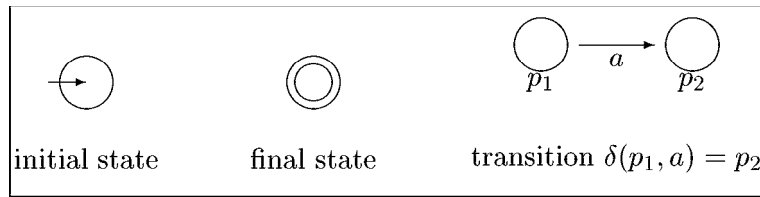
Figuur 1.1: Representation of a state graph
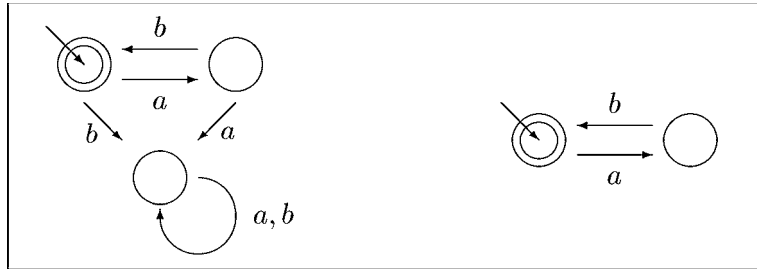
Figuur 1.2: State graph diagram of T (left) and with omitted error state (right)

In general, more state graphs may correspond to the same trace structure. Such state graphs are considered to be equivalent.

Definition 1.37 _A state $q \in Q$ of a state graph is called_ unreachable _if no path starting in the initial state ends in q._

Unreachable states (and all edges leaving this state) can be omitted without changing the meaning of the graph, i.e., it still accepts the same strings.

1.6.1   State graph diagram

State graphs can be displayed graphically. In figure 1.1 the graphical representation of initial state, final state, and transition is displayed.

In general, error states and all edges going to it are **not** drawn. So in a diagram all omitted edges are supposed to go to the error state. In figure 1.2 the corresponding state diagram of the trace structure of example 1.1 is drawn. At the left as it should be drawn, at the right as it is drawn.

1.7   Regular expressions

It is well known that if the number of equivalence classes of some trace structure is finite, the trace structure is _regular_. In that case the trace set can be displayed graphically by means of a finite state machine. Notice that our state graphs are finite state machines if Q is a finite set. According to property 1.32 they are also deterministic (see chapter 4).

It is also possible to describe a regular trace structure by means of regular expressions. First, we explain how a regular expression is defined (recursively).

Definition 1.38 *The empty string (ϵ) and every single symbol is a regular expression and if x and y are regular expressions, then also:*

xy	*concatenation*	*first x, then y*
$x\mid y$	*union*	*x or y*
x^*	*repetition*	*zero or more concatenations of x*
(x)	*binding*	*to overrule operator precedence*
x,y	*weaving*[3]	*shuffling of x and y*

The corresponding trace structures are:

$$
\begin{aligned}
\mathbf{ts}(\epsilon) &= \langle\{\epsilon\}, \varnothing\rangle \\
\mathbf{ts}(a) &= \langle\{a\}, \{a\}\rangle \\
\mathbf{ts}(xy) &= \langle\{t, u : t \in \mathbf{t}(\mathbf{ts}(x)) \ \wedge \ u \in \mathbf{t}(\mathbf{ts}(y)) : tu\}, \mathbf{a}(\mathbf{ts}(x)) \cup \mathbf{a}(\mathbf{ts}(y))\rangle \\
\mathbf{ts}(x\mid y) &= \mathbf{ts}(x) \cup \mathbf{ts}(y) \\
\mathbf{ts}(x^*) &= \langle\{t : t \in \mathbf{t}(\mathbf{ts}(x)) : t^*\}, \mathbf{a}(\mathbf{ts}(x))\rangle \\
\mathbf{ts}(x,y) &= \mathbf{ts}(x) \ \mathbf{w} \ \mathbf{ts}(y)
\end{aligned}
$$

We assume the following order in binding power of the operators: repetition has the greatest binding power, then concatenation, then weaving, and at least union. So $(xy^*, z\mid v)$ should be read as $(((x(y)^*), z)\mid v)$. In addition to the Kleene star we also have the notation x^+, denoting xx^*. A more detailed introduction to this notation and terminology can be found in [JvdS]. In the sequel we use regular expressions to denote trace sets rather than trace structures.[4]

Example 1.39 Consider

$$T = \langle(ab)^*, \{a, b\}\rangle \quad U = \langle(bc)^*, \{b, c\}\rangle$$

We have:

$$
\begin{aligned}
T \ \mathbf{w} \ U &= \langle(ab(acb)^*c)^*, \{a, b, c\}\rangle \\
&= \{((ab)^*, (bc)^*), \{a, b, c\}\} \\
T \ \mathbf{b} \ U &= \{(a(ac)^*c)^*, \{a, c\}\} \\
T \cup U &= \langle(ab)^*\mid(bc)^*, \{a, b, c\}\rangle
\end{aligned}
$$

In $T \ \mathbf{b} \ U$ the number of occurrences of a and c differ at most two and the number of occurrences of a is at least that of c. So $T \ \mathbf{b} \ U$ models a two-place buffer and is constructed using two one-place buffers from example 1.1. Notice the difference between $T \cup U$ and $T \ \mathbf{w} \ U$.

(end of example)

---

[3]In [JvdS] it has been proven that the class of regular trace structures is closed under weaving and blending. The comma operator can thus be considered as an operator in a regular expression.

[4]This choice becomes clear in the next chapter where the symbols get different meanings and it is not clear from the expression which meaning a symbol has. Moreover a trace structure may contain more events than are used in the expression.

2
Discrete processes

All you create, all you destroy
All that you do, all that you say

...

All that is now, all that is gone
All that's to come
And everything under the sun is in tune
But the sun is eclipsed by the moon

Eclipse – Dark side of the moon

In this chapter we show that trace structures can be used to model discrete processes. We do not consider time, i.e., we do not consider that events occur at fixed time instances, but we are only interested in the sequence in which the events occur.

2.1 Discrete processes

In order to be able to introduce control of discrete processes we split the alphabet while considering two kinds of events:

- exogenous[1] events
- communication[2] events

The exogenous events are used to model actions introduced by the process's own dynamics. This means that exogenous events do not appear in other processes.[3] The communication events are used to model actions of a process, that may be common to other processes. This kind of event is of interest in communication with other processes and will be used to control the exogenous events.

Assume that E and C are finite sets of events, such that

$$E \cap C = \emptyset$$

---

[1]**exogenous** (adj.) – growing or originating from outside (here used in the sense of not having to do with communication).

[2]**communication** (adj.) – to do with the exchange of thoughts, messages, etc.

[3]One can argue about the name *exogenous*. *Endogenous* events would perhaps be more convenient (endogenous means growing or originating from within). However it turns out that exogenous events are retained in a connection while communication events disappear. So communication events are internal and exogenous events are external with respect to connection. Moreover, exogenous is a standard term in system theory, although it is used in a slightly different setting here.

Consider a trace structure $T = \langle S, E \cup C \rangle$. Such a trace structure is called a *discrete process* and denoted by

$$P = \langle S, E, C \rangle$$

where S is the trace set of the process (the set of all possible sequences of occurring events), E the set of exogenous events, and C the set of communication events.

Once again we have operators to get the trace set, the exogenous alphabet, and the communication alphabet of a discrete process:

Definition 2.1 *For a discrete process $P = \langle S, E, C \rangle$ we define the operators:*

$$\mathbf{t}P = S \qquad \mathbf{e}P = E \qquad \mathbf{c}P = C \qquad \mathbf{a}P = E \cup C$$

Furthermore, with P we associate a trace structure denoted by $\mathbf{ts}(P)$ and defined by:

$$\langle \mathbf{t}P, \mathbf{a}P \rangle$$

We have defined the operators $\mathbf{t}$ and $\mathbf{a}$ on trace structures as well as discrete processes now.

In the sequel we sometimes use the operators $\mathbf{b}$ and $\mathbf{w}$ (blend and weave) on discrete processes. Formally, we use the following extension of these operators:

Definition 2.2 *The operators $\mathbf{w}$ and $\mathbf{b}$ are defined on discrete processes P and R by:*

$$
\begin{aligned}
P \mathbf{b} R &= \mathbf{ts}(P) \mathbf{b} \mathbf{ts}(R) \\
P \mathbf{w} R &= \mathbf{ts}(P) \mathbf{w} \mathbf{ts}(R)
\end{aligned}
$$

The blend and weave of two discrete processes results in a trace structure.

We are not concerned with how the communication is actually performed, i.e., which process *generates* the event and which process *receives* it. In other words, we do not make any distinction between input and output events here.

If we restrict our attention to the exogenous events, we have what is called the *exogenous behaviour* $\mathbf{t}P\lceil\mathbf{e}P$. If we restrict our attention to the communications, we have the *communication behaviour* $\mathbf{t}P|\mathbf{c}P$.

Example 2.3 Consider the following doctor-process:

$$P = \langle (aed)^*, \{e\}, \{a, d\} \rangle$$

with a meaning *arrival of a patient in the doctor's office*, e meaning *treat a patient*, and d meaning *departure of a patient*. The arrival and departure of patients are communication events (a patient can only arrive if the environment (i.e., some other process) "supplies" one; also, a patient can only depart if the environment can accept one). The treatment of the patient, however, is done by the doctor process P itself. It cannot be influenced by the environment. Notice that e^* is the exogenous behaviour of P and $(ad)^*$ is the communication behaviour of P.
(end of example)

2.1.1 Completed tasks

A trace from the trace set of a discrete process represents a *completed task* of that process. Each prefix of that trace represents an uncompleted task, unless that prefix is a trace itself in which case it is a completed task that can be continued.

With $|x|$ we denote the length of the string x, i.e., the number of symbols in trace x.

Example 2.4 Consider

$$P = \langle (ab|abc|ac), \{a,c\}, \{b\} \rangle$$

Then:

a is an uncompleted task
ab is a completed task that can be continued
abc is a completed task
ac is a completed task

we have: $|ab| = 2$ and $|\epsilon| = 0$
(end of example)

2.1.2 Non-empty processes

If the trace set of some discrete process is *empty*, we say that the process is empty. The process has no behaviour at all (i.e., not even ϵ, the nothing-has-happened behaviour). Because empty processes are of no use, we assume that all discrete processes are non-empty.

Furthermore, if $\epsilon \in tP$, we call the process *legally idle*: the empty trace is also a completed task, i.e., the process may end without doing anything.

Notice that P_1 and P_2 with $tP_1 = \varnothing$ and $tP_2 = \{\epsilon\}$ are completely different processes: P_1 has no behaviour at all: it cannot do anything, while the (only) behaviour of P_2 is doing nothing. The difference becomes clear when such processes are connected with other processes (see for example property 2.6).

2.1.3 Effect on state graphs

A discrete process has a lot of similarity with a trace structure. The only difference (at this moment) is found in the alphabets. Describing the behaviour of a discrete process can also be done using state graphs. To put all the information about a discrete process in a state graph, we denote communication events in a somewhat different way, while exogenous events will be displayed in the normal way: in a state graph an edge labeled with a denotes the occurrence of an *exogenous* event a and an edge labeled with $\bar{a}$ denotes the occurrence of a *communication* event a.[4]

2.1.4 Control of discrete processes

Because a process can only communicate with the environment by means of the communication events, control of discrete processes can thus be described as using the communication events to establish some predefined behaviour. In general, we will use (part of) the communication events to control the exogenous behaviour of the process. Therefore we investigate:

- the (uncontrolled) behaviour tP or the (uncontrolled) exogenous behaviour: $tP\lceil eP$,
- a second discrete process (the controller R) communicating with P by means of the communication events $cR \subseteq cP$,

---

[4]However we are unable to express the situation that a process has an event that does not occur in its behaviour. We have to mention such an event explicitly.

- the resulting controlled behaviour $\mathbf{t}(P \underline{\mathbf{b}} R)$ or the controlled exogenous behaviour: $\mathbf{t}(P \underline{\mathbf{b}} R)|eP$ (where $\underline{\mathbf{b}}$ denotes the connection of the two discrete processes and will be defined below).

In order to be able to discuss control of processes, we have to define what is meant by connection of discrete processes first.

2.2 Connections

In this section we define the notion *connection* of discrete processes.

Definition 2.5 *Given two discrete processes P and R with $eP \cap aR = \emptyset$ and $eR \cap aP = \emptyset$, then the connection of P and R is defined by:*

$$P \underline{\mathbf{b}} R$$
$$=$$
$$\langle \mathbf{t}(P \mathbf{b} R), eP \cup eR, cP \div cR \rangle$$

The behaviour is defined using the blend on the corresponding trace structures $\langle \mathbf{t}P, \mathbf{a}P \rangle$ and $\langle \mathbf{t}R, \mathbf{a}R \rangle$. Notice that all exogenous events of P and R are now exogenous events of the connection. From the communication events those that belong to only one process are left. This guarantees that

$$\mathbf{a}(P \underline{\mathbf{b}} R) = \mathbf{a}P \div \mathbf{a}R$$

so that $\mathbf{ts}(P \underline{\mathbf{b}} R) = \mathbf{ts}(P) \mathbf{b} \mathbf{ts}(R)$.

In the sequel we assume that writing $P \underline{\mathbf{b}} R$ automatically means that the connection is possible, i.e., that the conditions $eP \cap aR = \emptyset$ and $eR \cap aP = \emptyset$ are fulfilled.

Property 2.6 *For the connection $\underline{\mathbf{b}}$ the following properties hold:*

(1) $P \underline{\mathbf{b}} R = R \underline{\mathbf{b}} P$
(2) $P \underline{\mathbf{b}} \langle \{\epsilon\}, \emptyset, \emptyset \rangle = P$
(3) $P \underline{\mathbf{b}} \langle \emptyset, \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset, \emptyset \rangle$

The discrete process $\langle \{\epsilon\}, \emptyset, \emptyset \rangle$ is the *unit element* of the operator $\underline{\mathbf{b}}$. $\langle \emptyset, \emptyset, \emptyset \rangle$ is the *zero element* of $\underline{\mathbf{b}}$.

2.2.1 Total connection

Sometimes, we are interested in the total behaviour of the connected system. Therefore we introduce the *overall* or *total connection* $\underline{\mathbf{w}}$ as well.

Definition 2.7 *Given two discrete processes P and R with $eP \cap aR = \emptyset$ and $eR \cap aP = \emptyset$, then the total connection of P and R is defined by:*

$$P \underline{\mathbf{w}} R$$
$$=$$
$$\langle \mathbf{t}(P \mathbf{w} R), eP \cup eR \cup (cP \cap cR), cP \div cR \rangle$$

We have put all common communication events of P and R in the exogenous event set of the total connection. This guarantees that $\mathbf{a}(P \underline{\mathbf{w}} R) = \mathbf{a}P \cup \mathbf{a}R$ and that these events

can not be used for other communications as well.[5]

We use the total connection only to make the internal communication between the two processes visible in the connection. Again we have $\mathbf{ts}(P \; \underline{\mathbf{w}} \; R) = \mathbf{ts}(P) \; \mathbf{w} \; \mathbf{ts}(R)$.

Example 2.8 Reconsider the doctor example, but with two doctors:

$$P_1 = \langle (aed)^*, \{e\}, \{a, d\} \rangle \quad P_2 = \langle (dgb)^*, \{g\}, \{b, d\} \rangle$$

e and g have the meaning of treating a patient, a arrival of a patient, b departure of a patient, and d has the meaning of sending a patient from the first to the second doctor. Connecting P_1 with P_2 leads to

$$P_1 \; \underline{\mathbf{w}} \; P_2 = \langle (aed((ae, gb)d)^* gb)^*, \{e, g, d\}, \{a, b\} \rangle$$
$$P_1 \; \underline{\mathbf{b}} \; P_2 = \langle (ae(ae, gb)^* gb)^*, \{e, g\}, \{a, b\} \rangle$$

From which we derive that $\mathbf{t}(P_1 \; \underline{\mathbf{b}} \; P_2) \lceil (\mathbf{e}P_1 \cup \mathbf{e}P_2) = (e(e, g)^* g)^*$ so at each time at most two patients are being treated.

(end of example)

2.2.2 Multi-connections

More than two discrete processes can be connected as well. We assume that in such multi-connections all exogenous events are unique, i.e., appear in only one of the discrete processes, and communication events are used in at most one connection, i.e., no communication event occurs in more than two of the alphabets. So, if we connect P, R, and S, we demand that

$$\mathbf{e}P \cap (\mathbf{a}R \cup \mathbf{a}S) = \varnothing \quad \mathbf{c}P \cap \mathbf{c}R \cap \mathbf{c}S = \varnothing$$
$$\mathbf{e}R \cap (\mathbf{a}P \cup \mathbf{a}S) = \varnothing$$
$$\mathbf{e}S \cap (\mathbf{a}P \cup \mathbf{a}R) = \varnothing$$

Under these conditions the connections $(P \; \underline{\mathbf{b}} \; R) \; \underline{\mathbf{b}} \; S$ and $P \; \underline{\mathbf{b}} \; (R \; \underline{\mathbf{b}} \; S)$ are both defined correctly. We even have:

Property 2.9

(1) $(P \; \underline{\mathbf{b}} \; R) \; \underline{\mathbf{b}} \; S \;\; = \;\; P \; \underline{\mathbf{b}} \; (R \; \underline{\mathbf{b}} \; S)$
(2) $(P \; \underline{\mathbf{w}} \; R) \; \underline{\mathbf{w}} \; S \;\; = \;\; P \; \underline{\mathbf{w}} \; (R \; \underline{\mathbf{w}} \; S)$

Notice that for trace structures blending is only associative if no symbol occurs in more than two of the alphabets, which is established here through the above conditions.

Multi-connections are denoted using the continuous weave and continuous blend from trace theory.

_____

[5]Note the assumption here that we use communication events for communication between exactly two discrete processes.

Definition 2.10 *If $(P_1, \ldots, P_n)$ is a finite set of discrete processes, with no communication event occurring in more than two of the alphabets and no exogenous event occurring in more than one of the alphabets, then the multi-connection of the processes P_i is denoted by*

$$(\underline{\mathbf{B}}\, i : 1 \le i \le n : P_i)$$

and defined by

$$(\underline{\mathbf{B}}\, i : 1 \le i \le 0 : P_i) \;=\; \langle \{\epsilon\}, \emptyset, \emptyset \rangle$$
$$(\underline{\mathbf{B}}\, i : 1 \le i \le n : P_i) \;=\; (\underline{\mathbf{B}}\, i : 1 \le i \le n-1 : P_i)\, \underline{\mathbf{b}}\, P_n$$

and the multi-total-connection *is denoted by*

$$(\underline{\mathbf{W}}\, i : 1 \le i \le n : P_i)$$

and defined by

$$(\underline{\mathbf{W}}\, i : 1 \le i \le 0 : P_i) \;=\; \langle \{\epsilon\}, \emptyset, \emptyset \rangle$$
$$(\underline{\mathbf{W}}\, i : 1 \le i \le n : P_i) \;=\; (\underline{\mathbf{W}}\, i : 1 \le i \le n-1 : P_i)\, \underline{\mathbf{w}}\, P_n$$

2.3 Other operators on discrete processes

2.3.1 Joint behaviour

Modelling a system is normally done by modelling parts of the system first which are afterwards joined together. Joining a number of discrete processes to create one new discrete process in this sense does not mean connecting the discrete processes; they do not communicate with each other, they only behave such as to obey each individual behaviour given in the trace sets.

Therefore, the operator $\underline{\mathbf{w}}$ is of no use here and we introduce a similar operator $\underline{\mathbf{s}}$ called the shuffle operator.

Definition 2.11 *The* joint behaviour *of two discrete processes P and R is defined by*

$$P \,\underline{\mathbf{s}}\, R$$
$$=$$
$$\langle \mathbf{t}(P\, \mathbf{w}\, R), \mathbf{e}P \cup \mathbf{e}R, \mathbf{c}P \cup \mathbf{c}R \rangle$$

Notice that common communication events are *not* removed nor placed in the resulting exogenous alphabet.

Example 2.12 With P and R given by

$$P = \langle (ab)^*, \{a\}, \{b\} \rangle \quad R = \langle (bc)^*, \{c\}, \{b\} \rangle$$

we have:

$$
\begin{aligned}
P \,\underline{\mathbf{b}}\, R &= \langle (a(ac)^*c)^*, \{a,c\}, \emptyset \rangle \\
P \,\underline{\mathbf{w}}\, R &= \langle (ab(acb)^*c)^*, \{a,b,c\}, \emptyset \rangle \\
P \,\underline{\mathbf{s}}\, R &= \langle (ab(acb)^*c)^*, \{a,c\}, \{b\} \rangle
\end{aligned}
$$

(end of example)

Property 2.13 *For the shuffle* $\underline{s}$ *, the following properties hold:*

(1) $P \underline{s} R = R \underline{s} P$
(2) $P \underline{s} \langle \{\epsilon\}, \emptyset, \emptyset \rangle = P$
(3) $P \underline{s} \langle \emptyset, \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset, \emptyset \rangle$
(4) $(P \underline{s} R) \underline{s} S = P \underline{s} (R \underline{s} S)$

The multi-shuffle $(P \underline{s} R) \underline{s} S$ is always defined, while the multi-connection $(P \underline{w} R) \underline{w} S$ is not.

2.3.2 Concatenation of discrete processes

As with regular expressions it is also possible to concatenate discrete processes.

Definition 2.14 *The* concatenation *of two discrete processes P and R is defined by*

$$P \; ; R$$
$$=$$
$$\langle \{x, y : x \in \mathbf{t}P \; \wedge \; y \in \mathbf{t}R : xy\}, \mathbf{e}P \cup \mathbf{e}R, \mathbf{c}P \cup \mathbf{c}R \rangle$$

A concatenation of two discrete processes behaves like the first process until that process completes a task (performs a completed trace). Then the behaviour is continued according to the behaviour of the second process. Because the processes do not communicate with each other, again no restrictions are needed on their alphabets.

Example 2.15 With P and R as in example 2.12, we have:

$$P \; ; R = \langle (ab)^*(bc)^*, \{a, c\}, \{b\} \rangle$$

(end of example)

2.3.3 Set operators

As for trace structures, the operators $\cap$, $\cup$, and $\setminus$ are also defined on discrete processes in the usual way:

Definition 2.16 *Given two discrete processes P and R with $\mathbf{e}P = \mathbf{e}R$ and $\mathbf{c}P = \mathbf{c}R$, then the* union, *intersection* and *exclusion of P and R are defined by:*

$$P \cup R \;\; = \;\; \langle \mathbf{t}P \cup \mathbf{t}R, \mathbf{e}P, \mathbf{c}P \rangle$$
$$P \cap R \;\; = \;\; \langle \mathbf{t}P \cap \mathbf{t}R, \mathbf{e}P, \mathbf{c}P \rangle$$
$$P \setminus R \;\; = \;\; \langle \mathbf{t}P \setminus \mathbf{t}R, \mathbf{e}P, \mathbf{c}P \rangle$$

2.3.4 Ordering of discrete processes

Ordering of discrete processes is defined using the ordering of trace structures, i.e.,

Definition 2.17 *For two discrete processes P and R, the ordering $P \subseteq R$ is defined by*

$$\mathbf{c}P = \mathbf{c}R \; \wedge \; \mathbf{e}P = \mathbf{e}R \; \wedge \; \mathbf{t}P \subseteq \mathbf{t}R$$

Again, ordering is only defined on discrete processes with equal alphabets in the sence that P is at most R if the corresponding trace set of P is at most the trace set of R.

2.4   A shop

Before we give a precise description of our control problem we give an illustrative example first.

Suppose a shop sells two kinds of articles and in order to get an article one has to pay for it. Paying for an article is supposed to be a communication action. So we have as events:

a_1 sell article 1
a_2 sell article 2
p_1 pay for article 1
p_2 pay for article 2

The complete process becomes

$$P = \langle((p_1 a_1)|(p_2 a_2))^*, \{a_1, a_2\}, \{p_1, p_2\}\rangle$$

The behaviour of P is a repetitive choice of paying for article 1 and buying it and paying for article 2 and buying it. For example $p_1 a_1 p_1 a_1 p_2 a_2 p_1 a_1$ is a legal trace of P.

A customer can now be described as "pay for every article wanted," for example:

$$R = \langle((p_1 p_1), p_2), \varnothing, \{p_1, p_2\}\rangle$$

if the customer wants to have two articles number 1 and one article number 2. Connecting these two processes results in:

$$P \underline{\mathbf{b}} R = \langle((a_1 a_1), a_2), \{a_1, a_2\}, \varnothing\rangle$$

Notice that the uncontrolled behaviour of the shop P equals $\mathbf{t}P\lceil\mathbf{e}P = (a_1|a_2)^*$, while the controlled behaviour is $\mathbf{t}(P \underline{\mathbf{b}} R) = (a_1 a_1), a_2$. So we have: $P \underline{\mathbf{b}} R \subseteq P\lceil\mathbf{e}P$. The customer has in fact controlled the exogenous behaviour of the shop.

2.5   A control problem

As mentioned before, we wish to use (part of) the communication events to control the exogenous events. We wish to construct, given a discrete process P, a second discrete process, the controller R, with as possible events (part of) the communication events of P, such that the resulting exogenous behaviour of the connection of P and R is within some lower and upper limits given by L_{min} and L_{max}, i.e.,

$$L_{min} \subseteq (P \underline{\mathbf{b}} R) \subseteq L_{max}$$

This problem is called *control of discrete events* (CODE for short). In the following chapter we give a formal outline of CODE and an algorithm to solve CODE (which is called a *deCODEr*).

Control of discrete processes

In this chapter we discuss the following control problem:

Given are a discrete process $P = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{c}P \rangle$ and two discrete processes L_{min} and L_{max} with

$$\mathbf{e}L_{min} = \mathbf{e}P \qquad \mathbf{c}L_{min} = \emptyset$$
$$\mathbf{e}L_{max} = \mathbf{e}P \qquad \mathbf{c}L_{max} = \emptyset$$
$$L_{min} \subseteq L_{max}$$

L_{min} and L_{max} specify the range of resulting exogenous traces that are acceptable. The problem is to find, if possible, a discrete process R with:

$$R = \langle \mathbf{t}R, \emptyset, \mathbf{c}R \rangle \quad \text{with: } \mathbf{c}R \subseteq \mathbf{c}P$$

such that

$$L_{min} \subseteq (P \underline{\mathbf{b}} R) \lceil \mathbf{e}P \subseteq L_{max}$$

This last condition is called the *minmax condition.*[1]

The restriction of the alphabet of L_{max} being equal to $\mathbf{e}P$ is needed because we can only give restrictions on the existing exogenous events. Without loss of generality we assume:

$$L_{min} \subseteq L_{max} \subseteq P \lceil \mathbf{e}P$$

(so we give restrictions to existing exogenous traces only). Of course, L_{max} can be extended with traces not in $P \lceil \mathbf{e}P$, but $(P \underline{\mathbf{b}} R) \lceil \mathbf{e}P \subseteq P \lceil \mathbf{e}P$, so if $(P \underline{\mathbf{b}} R) \lceil \mathbf{e}P \subseteq L_{max}$ with $L_{max} \subseteq P \lceil \mathbf{e}P$, then also $(P \underline{\mathbf{b}} R) \lceil \mathbf{e}P \subseteq L_{max} \cup L_{extra}$ for all $L_{extra} \not\subseteq P \lceil \mathbf{e}P$.

In the sequel this problem is referred to as CODE.

---

[1]This condition is adapted from a similar condition from supervisory control theory (see [RaWo]). See also chapter 5.

Example 3.1 A possible control problem could be:
Given

$$P = \langle (ae|ad|ag|be|cg), \{d, e, g\}, \{a, b, c\} \rangle$$
$$L_{min} = \langle (e), \{d, e, g\}, \varnothing \rangle$$
$$L_{max} = \langle (e|g), \{d, e, g\}, \varnothing \rangle$$

find a controller R with $\mathbf{c}R = \{a, b\}$ such that $L_{min} \subseteq (P \underline{\mathbf{b}} R) \lceil \{d, e, g\} \subseteq L_{max}$.
(end of example)

Example 3.2 Reconsider the shop example of section 2.4. A possible problem could be to buy at most two articles of kind 2 and always one more of kind 1. The desired L_{min} and L_{max} then are

$$L_{min} = \langle (a_1), \{a_1, a_2\}, \varnothing \rangle$$
$$L_{max} = \langle (a_1|(a_1a_1, a_2)|(a_1a_1a_1, a_2a_2)), \{a_1, a_2\}, \varnothing \rangle$$

(end of example)

In conventional system theory a system is controlled using the outputs as observations and computing new inputs for the system in order to get a desired behaviour (for example to make the system stable or decouple disturbances). In fact, the basic idea here is the same: we use the communication events[2] to establish some predefined behaviour. The main difference is of course that our desired behaviour has to do with order of events, not with behaviour in time.

3.1 Solution for CODE

In the sequel we deal (without loss of generality)[3] only with the situation that $\mathbf{c}R = \mathbf{c}P$. So we use *all* communication events to control the process.

3.1.1 A first attempt

Notice that P and L (for some L satisfying $L_{min} \subseteq L \subseteq L_{max}$) have events in common. These events are exogenous events, so connecting P with L is not allowed. However, suppose we interchange the meaning of communication and exogenous events for a moment and indeed perform a connection between P and L. The result of this (pseudo) connection is a discrete process with communication events only. Perhaps this process is the controller process we are looking for.

To do things formally, we use the blend $\mathbf{b}$ instead of the connecting operator $\underline{\mathbf{b}}$ here, because $\underline{\mathbf{b}}$ is not allowed between P and L. So we try $\mathbf{t}R = \mathbf{t}(P \mathbf{b} L)$ for some L satisfying $L_{min} \subseteq L \subseteq L_{max}$.

Notice that $P \mathbf{b} L_{min} \subseteq P \mathbf{b} L_{max}$. Furthermore, reconsider lemma 1.17 which gives $P \mathbf{b} (P \mathbf{b} L_{min}) \supseteq \mathbf{ts}(L_{min})$. Combining these facts, we may conclude that L_{min} is a candidate for L. This leads to:

---

[2]At this point no distinction is made between inputs and outputs. See chapter 7.

[3]In the last section of this chapter we discuss more general settings of CODE.

Lemma 3.3

$$P \text{ b } (P \text{ b } L_{min}) \subseteq \mathbf{ts}(L_{max})$$
$$\Rightarrow$$
$$CODE \text{ is solvable}$$

Example 3.4 Given is:

$$P = \langle (ae|ad|be), \{d, e\}, \{a, b\} \rangle$$

If $\mathbf{t}L_{min} = \mathbf{t}L_{max} = \{e\}$ then:

$$\mathbf{t}(P \text{ b } (P \text{ b } L_{min}))$$
$$=$$
$$\mathbf{t}(P \text{ b } \langle (a|b), \{a, b\}, \varnothing \rangle)$$
$$=$$
$$\{d, e\}$$
$$\nsubseteq$$
$$\mathbf{t}L_{max}$$

The lemma does not lead to a solution for CODE, although there is one. Take $R = \{b\}$, then $\mathbf{t}(P \text{ b } R) = \{e\} = \mathbf{t}L_{max}$. We may conclude that CODE can indeed have a solution even if the condition in the above lemma is not fulfilled.

This example shows that $P \text{ b } (P \text{ b } L)$ may lead to a control trace set that is too large. This is due to the fact that the trace set $\mathbf{t}(P \text{ b } L)$ may become too large, because it contains traces that lead to undesired results (not satisfying the minmax condition) as well: in the example we have $\mathbf{t}(P \text{ b } L_{min}) = (a|b)$ in which the trace a is undesirable. $P \text{ b } \langle \{a\}, \{a, b\}, \varnothing \rangle$ leads to $(d|e)$ that contains the undesired trace d. If we omit the illegal control trace a from $P \text{ b } L$ we find a suitable controller. This idea is used in the algorithm below.
(end of example)

3.1.2 A second (and successful) attempt

In this section we give an algorithm to construct a solution for CODE, which also can be used to investigate if CODE has a solution at all. For the algorithm we need two functions:

Definition 3.5 _With the CODE problem we associate the following discrete processes:_

$$F(P, L) = \langle \mathbf{t}((P \text{ b } L) \setminus (P \text{ b } (P|\mathbf{e}P \setminus L))), \varnothing, \mathbf{c}P \rangle$$

called the friend[4] _of L, and_

$$G(P, L) = P \underline{\text{ b }} F(P, L)$$

called the guardian[5] _of L._

$F(P, L)$ describes a possible candidate for solving the CODE-problem. The corresponding exogenous behaviour using this controller is given by $G(P, L)$.

---

[4]**friend** (n.) – Sympathizer, helper, helpful thing. The term friend is adapted from [Wonh].

[5]**guardian** (n.) – One having custody of person or property.

When it is clear from the context which process P is considered, we write $F(L)$ and $G(L)$ for short.

In most cases, we only need the trace set of $F(L)$ and $G(L)$ so we introduce:

$$f(L) = \mathbf{t}F(L)$$
$$g(L) = \mathbf{t}G(L)$$

Notice that:

$$\mathbf{e}F(L) = \varnothing \quad \mathbf{c}F(L) = \mathbf{c}P$$
$$\mathbf{e}G(L) = \mathbf{e}P \quad \mathbf{c}G(L) = \varnothing$$

The algorithm (called the *deCODEr*) is described as follows:

Algorithm 3.6

> **for all** L **such that** $\quad L_{min} \subseteq L \subseteq L_{max}$:
> > **if** $L_{min} \subseteq G(L) \subseteq L_{max}$
> > **then** $F(L)$ is a solution

Of course, it is not immediately clear that this algorithm works. In the next section we try to make it plausible. In the section thereafter we give a proof of the algorithm and, more importantly, give a necessary and sufficient condition for the CODE-problem to be solvable.

3.1.3 Outline of the algorithm

First let us try

$$f'(L) = \mathbf{t}(P \mathbf{\ b\ } L)$$

So, if $G'(L)$ (with $g'(L) = \mathbf{t}(P \mathbf{\ \underline{b}\ } F'(L)) = \mathbf{t}(P \mathbf{\ b\ } (P \mathbf{\ b\ } L)))$ satisfies the minmax condition, then $R = F'(L)$ should be a solution.

However, starting with an L such that $L_{min} \subseteq L \subseteq L_{max}$ does not guarantee that $L_{min} \subseteq G'(L) \subseteq L_{max}$. See example 3.4 which results (with $L = L_{max}$) in:

$$f'(L) = \mathbf{t}(P \mathbf{\ b\ } L) = (a|b)$$
$$g'(L) = \mathbf{t}(P \mathbf{\ \underline{b}\ } F'(L)) = (d|e) \neq \mathbf{t}L = (e)$$

In $f'(L)$ the trace a does not lead to the desired solution because it also allows exogenous behaviour d to occur while d does not satisfy the minmax condition. Therefore, we repeat our computation, this time not using L itself but its complement in P, i.e., we compute

$$f''(L) = \mathbf{t}(P \mathbf{\ b\ } \neg L)$$

with $\neg L = (P\lceil \mathbf{e}P) \setminus L$ (i.e., all exogenous traces in P that do not belong to L).

In our example $\mathbf{t}(\neg L) = (d)$, so we get $f''(L) = (a)$. $f''(L)$ now gives all communication traces that may lead to undesired results. If we use $f(L) = f'(L) \setminus f''(L)$ now, we find exactly those control traces that lead to the desired results. Here $f(L) = b$. Summarizing:

- First, compute $P \mathbf{\ b\ } L$ to get all possible control traces,
- Next, compute $P \mathbf{\ b\ } \neg L$ to get all control traces that give undesired results,
- Finally, take $(P \mathbf{\ b\ } L) \setminus (P \mathbf{\ b\ } \neg L)$ to find exactly the right control traces.

This is precisely, what $F(L)$ does:

$$f(L) = \mathbf{t}(\underbrace{(P \, \mathbf{b} \, L)}_{\substack{\text{possible} \\ \text{controls}}} \setminus \underbrace{(P \, \mathbf{b} \, \underbrace{(P \lceil \mathbf{e}P \setminus L)}_{\neg L})}_{\text{undesired controls}})$$

$$\underbrace{\phantom{(P \, \mathbf{b} \, L) \setminus (P \, \mathbf{b} \, (P \lceil \mathbf{e}P \setminus L))}}_{\text{desired controls}}$$

3.2 Proof of the algorithm

First, a number of properties of the friend and the guardian are listed.

Lemma 3.7 _The friend and the guardian of L satisfy:_

$$f(L) = \{z : z \in \mathbf{t}P | \mathbf{c}P \wedge (\forall x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}L) : z\}$$
$$g(L) = \{x : x \in \mathbf{t}P \wedge (\forall y : y \in \mathbf{t}P \wedge y | \mathbf{c}P = x | \mathbf{c}P : y \lceil \mathbf{e}P \in \mathbf{t}L) : x \lceil \mathbf{e}P\}$$

proof: We have:

$\mathbf{t}(P \, \mathbf{b} \, (P \lceil \mathbf{e}P \setminus L))$
$=$ 　[property 1.14 (3)]
$\{z : z \in \mathbf{t}P | \mathbf{c}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}(P \lceil \mathbf{e}P \setminus L)) : z\}$
$=$ 　[$x \lceil \mathbf{e}P \in \mathbf{t}P \lceil \mathbf{e}P$]
$\{z : z \in \mathbf{t}P \lceil \mathbf{c}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \notin \mathbf{t}L) : z\}$

Hence:

$f(L)$
$=$ 　[definition of $F(P, L)$]
$\mathbf{t}((P \, \mathbf{b} \, L) \setminus (P \, \mathbf{b} \, (P \lceil \mathbf{e}P \setminus L)))$
$=$ 　[definition of $\setminus$, property 1.14 (2), and previous equation]
$\{z : z \in \mathbf{t}P \lceil \mathbf{c}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}L) : z\} \setminus$
$\{z : z \in \mathbf{t}P \lceil \mathbf{c}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \notin \mathbf{t}L) : z\}$
$=$
$\{z : z \in \mathbf{t}P \lceil \mathbf{c}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}L) \wedge$
$\qquad \neg(\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \notin \mathbf{t}L)$
$\qquad : z\}$
$=$ 　[$x \lceil \mathbf{e}P \in \mathbf{t}P \lceil \mathbf{e}P$]
$\{z : z \in \mathbf{t}P \lceil \mathbf{c}P \wedge (\forall x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}L) : z\}$

Furthermore:

$g(L)$
$=$ 　[definition of $G(P, L)$]
$\mathbf{t}(P \, \underline{\mathbf{b}} \, F(L))$
$=$ 　[definition of $\underline{\mathbf{b}}$]
$\{x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P \in f(L) : x \lceil \mathbf{e}P\}$
$=$ 　[expression of $f(L)$]
$\{x : x \in \mathbf{t}P \wedge (\forall y : y \in \mathbf{t}P \wedge y | \mathbf{c}P = x | \mathbf{c}P : y \lceil \mathbf{e}P \in \mathbf{t}L) : x | \mathbf{c}P\}$

(end of proof)

Lemma 3.8

$$F(L) \subseteq P\lceil \mathbf{c}P$$

proof: Trivial.

Lemma 3.9

$$G(L) \subseteq L$$

proof:

$$z \in g(L)$$
$\Leftrightarrow$ [definition of the guardian]
$$z \in \mathbf{t}(P \mathbf{\underline{b}} F(L))$$
$\Leftrightarrow$ [definition of $\mathbf{\underline{b}}$]
$$(\exists x : x \in \mathbf{t}P \wedge x\lceil \mathbf{c}P \in f(L) : x\lceil \mathbf{e}P = z)$$
$\Rightarrow$ [lemma 3.7: $x \in \mathbf{t}P \wedge x\lceil \mathbf{c}P \in f(L) \Rightarrow x\lceil \mathbf{e}P \in \mathbf{t}L$]
$$z \in \mathbf{t}L$$

(end of proof)

Lemma 3.9 implies that by choosing $L = L_{max}$ (the largest possible choice) the solution found by the deCODEr still satisfies the right part of the minmax condition of CODE. Notice that $G(L) = L$ does not hold in general.

Lemma 3.10

$$R \subseteq P\lceil \mathbf{c}P \wedge P \mathbf{\underline{b}} R \subseteq L \Rightarrow R \subseteq F(L)$$

proof:

$$z \in \mathbf{t}R$$
$\Rightarrow$ [$R \subseteq P\lceil \mathbf{c}P \wedge P \mathbf{\underline{b}} R \subseteq L$]
$$z \in \mathbf{t}P\lceil \mathbf{c}P \wedge (\forall x : x \in \mathbf{t}P \wedge x\lceil \mathbf{c}P = z : x\lceil \mathbf{e}P \in \mathbf{t}L)$$
$\Rightarrow$ [lemma 3.7]
$$z \in f(L)$$

(end of proof)

Lemma 3.10 implies that (take $L = L_{max}$) every solution of CODE that is contained in $P\lceil \mathbf{c}P$ is contained in $R_{max} = F(L_{max})$. Using lemma 3.9 we see that R_{max} therefore is the greatest possible solution of CODE and can be constructed using the algorithm.[6]

Example 3.11 Reconsider:

$$P = \langle (ae|ad|be), \{d, e\}, \{a, b\} \rangle \quad L = \langle (e), \{d, e\}, \varnothing \rangle$$

Take $R = \langle (a|b), \varnothing, \{a, b\} \rangle$, then we have $R \subseteq P\lceil \mathbf{c}P$ and $\mathbf{t}(P \mathbf{\underline{b}} R) = (d|e)$ (so: $L \subseteq P \mathbf{\underline{b}} R$), but $f(L) = (b)$ (see section 3.1.3) so $F(L) \not\subseteq R$. We conclude that we cannot prove:

$$R \subseteq P\lceil \mathbf{c}P \wedge L \subseteq P \mathbf{\underline{b}} R \Rightarrow F(L) \subseteq R$$

---

[6]Notice that it is always possible to add to R_{max} traces that have no influence when they are blended with P. So it is only possible to find a greatest solution that is contained in $P\lceil \mathbf{c}P$.

and therefore, we cannot find in general a smallest possible solution. In other words, as we shall see, if a solution exists, the solution constructed via L_{max} is a most *liberal* solution. There may or may not be a most *conservative* one.
(end of example)

Lemma 3.12

$$L_1 \subseteq L_2 \;\Rightarrow\; F(L_1) \subseteq F(L_2)$$

proof:

$$y \in f(L_1)$$
$\Leftrightarrow$ [lemma 3.7]
$$(\exists z : z \in \mathbf{t}P\lceil \mathbf{c}P \;\wedge\; (\forall x : x \in \mathbf{t}P \;\wedge\; x\lceil \mathbf{c}P = z : x\lceil \mathbf{e}P \in \mathbf{t}L_1))$$
$\Rightarrow$ [$x\lceil \mathbf{e}P \in \mathbf{t}L_1 \Rightarrow x\lceil \mathbf{e}P \in \mathbf{t}L_2$]
$$(\exists z : z \in \mathbf{t}P\lceil \mathbf{c}P \;\wedge\; (\forall x : x \in \mathbf{t}P \;\wedge\; x\lceil \mathbf{c}P = z : x\lceil \mathbf{e}P \in \mathbf{t}L_2))$$
$\Leftrightarrow$ [lemma 3.7]
$$y \in f(L_2)$$

(end of proof)

Lemma 3.13

$$L_1 \subseteq L_2 \;\Rightarrow\; G(L_1) \subseteq G(L_2)$$

proof:

$$L_1 \subseteq L_2$$
$\Rightarrow$ [lemma 3.12]
$$F(L_1) \subseteq F(L_2)$$
$\Rightarrow$ [property 1.21 (2)]
$$P \;\underline{\mathbf{b}}\; F(L_1) \subseteq P \;\underline{\mathbf{b}}\; F(L_2)$$
$\Leftrightarrow$ [definition of $G(P, L)$]
$$G(L_1) \subseteq G(L_2)$$

(end of proof)

This lemma states that an increasing set of choices of L leads to an increasing set of resulting exogenous traces $P \;\underline{\mathbf{b}}\; F(L)$ of the CODE problem (monotonicity).

We are able now to prove the following useful result:

Theorem 3.14

$$CODE \text{ is solvable}$$
$\Leftrightarrow$
$$L_{min} \subseteq G(L_{max})$$

proof: Suppose CODE has a solution, say R, then write

$$R = R_{int} \cup R_{ext}$$

with:

$$R_{int} = P\lceil \mathbf{c}P \cap R$$
$$R_{ext} = R \setminus R_{int}$$

(which gives $P \underline{\mathbf{b}} R = P \underline{\mathbf{b}} R_{int}$).
Then we have:

$\quad\quad true$
$\Leftrightarrow \quad [\ R$ is a solution and $P \underline{\mathbf{b}} R_{int} = P \underline{\mathbf{b}} R\ ]$
$\quad\quad P \underline{\mathbf{b}} R_{int} \subseteq L_{max}$
$\Rightarrow \quad [\ \text{lemma 3.10 with } R_{int} \subseteq P | \mathbf{c} P\ ]$
$\quad\quad R_{int} \subseteq F(L_{max})$

Which gives:

$\quad\quad L_{min}$
$\subseteq \quad [\ R$ is a solution: $L_{min} \subseteq \mathbf{t}(P \underline{\mathbf{b}} R) = \mathbf{t}(P \underline{\mathbf{b}} R_{int})\ ]$
$\quad\quad P \underline{\mathbf{b}} R_{int}$
$\subseteq \quad [\ \text{above implication }]$

$\quad\quad P \underline{\mathbf{b}} F(L_{max})$
$= $
$\quad\quad G(L_{max})$

Next, suppose CODE has no solution:

$\quad\quad (\forall R : P \underline{\mathbf{b}} R \subseteq L_{max} : L_{min} \not\subseteq P \underline{\mathbf{b}} R)$
$\Rightarrow \quad [\ \text{choose } R = F(L_{max}) \text{ (such an } R \text{ exists, see lemma 3.9) }]$
$\quad\quad L_{min} \not\subseteq P \underline{\mathbf{b}} F(L_{max})$
$\Leftrightarrow \quad [\ \text{definition of } G(P, L)\ ]$
$\quad\quad L_{min} \not\subseteq G(L_{max})$

(end of proof)

This theorem, together with previous lemmas, implies that if

$\quad\quad L_{min} \subseteq G(L)$

for some L, satisfying $L_{min} \subseteq L \subseteq L_{max}$, the process $F(L)$ is a solution of CODE. If even for $L = L_{max}$ this condition is not fulfilled, no solution exists.

3.3   Some properties of the deCODEr

The following lemmas give some properties of solutions of CODE, as constructed using the friend and the guardian.

Lemma 3.15

$\quad\quad F(L_1) \cup F(L_2) = F(L_1 \cup L_2)$

proof: From lemma 3.7 we easily obtain: $f(L_1) \cup f(L_2) = f(L_1 \cup L_2)$.

Lemma 3.16

$$(L_{min} \subseteq G(L_1) \subseteq L_{max}) \quad \wedge \quad (L_{min} \subseteq G(L_2) \subseteq L_{max})$$
$$\Rightarrow$$
$$L_{min} \subseteq G(L_1 \cup L_2) \subseteq L_{max}$$

proof: trivial, using:

$$
\begin{aligned}
& G(L_1 \cup L_2) \\
=\; & \quad [\text{ definition of } G(P, L)] \\
& P \underline{\mathbf{b}} F(L_1 \cup L_2)) \\
=\; & \quad [\text{ lemma 3.15 }] \\
& P \underline{\mathbf{b}} (F(L_1) \cup F(L_2)) \\
=\; & \quad [\text{ property 1.23 (3) }] \\
& (P \underline{\mathbf{b}} F(L_1)) \cup (P \underline{\mathbf{b}} F(L_2)) \\
=\; & \quad [\text{ definition of } G(P, L)] \\
& G(L_1) \cup G(L_2)
\end{aligned}
$$

(end of proof)

This lemma states that if R_1 and R_2 are both solutions of CODE (and of the form $F(L)$ for some L), then also $R_1 \cup R_2$ is a solution. This lemma implies that a greatest solution (contained in $P\lceil \mathbf{c}P$) exists.

In general, however, $R_1 \cap R_2$ and $R_1 \underline{\mathbf{w}} R_2$ need not be solutions, which prevents the existence of a minimal solution, as is shown in the following example.

Example 3.17 Consider

$$P = \langle (ebc|bea), \{e\}, \{a, b, c\} \rangle \quad R_1 = \langle (bc), \varnothing, \{a, b, c\} \rangle$$
$$R_2 = \langle (ba), \varnothing, \{a, b, c\} $$

then $\mathbf{t}(P \underline{\mathbf{b}} R_1) = (e)$ and $\mathbf{t}(P \underline{\mathbf{b}} R_2) = (e)$ (so both R_1 and R_2 are solutions of CODE), but $\mathbf{t}(P \underline{\mathbf{b}} (R_1 \cap R_2)) = \varnothing \not\supseteq \mathbf{t}L_{min}$ (hence $R_1 \cap R_2$ is no solution of CODE). This is due to the fact that (according to property 1.23):

$$P \underline{\mathbf{b}} (R_1 \cap R_2) \subseteq (P \underline{\mathbf{b}} R_1) \cap (P \underline{\mathbf{b}} R_2)$$

In general, equality does not hold.
(end of example)

Next, we would like to investigate whether every solution of CODE can be written in terms of a friend of some L satisfying the minmax condition. Because every solution of CODE can be extended with traces that have no influence on the result (take $R_{new} = R \cup R_{ext}$ for an R_{ext} with $\mathbf{t}(P \underline{\mathbf{b}} R_{ext}) = \varnothing$, then R_{new} is also a solution), we can only hope that every solution R with $R \subseteq P\lceil \mathbf{c}P$ can be written in terms of a certain friend. Suppose R is a solution of CODE with $R \subseteq P\lceil \mathbf{c}P$, then $P \underline{\mathbf{b}} R$ satisfies the minmax condition. From lemma 3.10 we have:

$$R \subseteq F(P \underline{\mathbf{b}} R)$$

In general, we have no equality here, as is shown in the following example.

Example 3.18 Consider

$$P = \langle (adbe|adae), \{d,e\}, \{a,b\} \rangle$$
$$L_{min} = L_{max} = \langle (de), \{d,e\}, \varnothing \rangle$$

then $x_1 = adbe$ leads to $x_1 \lceil \mathbf{c} P = ab$ and $x_1 \lceil \mathbf{e} P = de$ and $x_2 = adae$ leads to $x_2 \lceil \mathbf{c} P = aa$ and $x_2 \lceil \mathbf{e} P = de$. We know that $R = \langle \{aa\}, \varnothing, \{a,b\} \rangle$ is a solution of CODE, because $\mathbf{t}(P \, \underline{\mathbf{b}} \, R) = \{de\}$, but no L satisfying the minmax condition can be found so that $f(L) = \mathbf{t} R$. The only possible L, namely $L = L_{min}$, leads to $f(L) = \{ab, aa\} \neq \mathbf{t} R$.

In this case there are more communication traces $y \in \mathbf{t} P \lceil \mathbf{c} P$ that lead to the same exogenous trace ($y = aa$ as well as $y = ab$ leads to the exogenous trace de). To find a solution not all these communication traces are needed. Just one will do, but using the deCODEr all possible communication traces are given.

(end of example)

3.4 Observability

It is easily seen that if in P all exogenous traces can be found by applying a unique communication trace only, all solutions of CODE can be found by applying the deCODEr (i.e., are of the form $F(L)$). A process P with this property is called observable.

Definition 3.19 *P is observable, notation* **observable**(P), *if*

$$(\forall x, y : x \in \mathbf{t} P \; \wedge \; y \in \mathbf{t} P : x \lceil \mathbf{e} P = y \lceil \mathbf{e} P \Rightarrow x \lceil \mathbf{c} P = y \lceil \mathbf{c} P)$$

Example 3.20 The processes P in the examples 3.11 and 3.18 are not observable, for example for

$$P = \langle (ae|ad|be), \{d,e\}, \{a,b\} \rangle$$

we have that $x = ae$ and $y = be$ both satisfy $x \lceil \mathbf{e} P = y \lceil \mathbf{e} P$ but $x \lceil \mathbf{c} P \neq y \lceil \mathbf{c} P$. However

$$P' = \langle (ad|be), \{d,e\}, \{a,b\} \rangle \quad P'' = \langle (ae|ad), \{d,e\}, \{a,b\} \rangle$$

are both observable. P'' has no solution for CODE with $\mathbf{t} L_{min} = \mathbf{t} L_{max} = (e)$. P' has precisely one solution, namely R with $\mathbf{t} R = (b)$.

(end of example)

Lemma 3.21

    observable(P)

$\Rightarrow$

    $(\forall R : R \text{ is a solution of } CODE \; \wedge \; R \subseteq P \lceil \mathbf{c} P : F(P \, \underline{\mathbf{b}} \, R) = R)$

proof: We only have to prove that $R \supseteq F(P \, \underline{\mathbf{b}} \, R)$:

    $z \in f(P \, \underline{\mathbf{b}} \, R)$

$\Rightarrow$ [lemma 3.7]

    $z \in \mathbf{t} P \lceil \mathbf{c} P \; \wedge \; (\forall x : x \in \mathbf{t} P \; \wedge \; x \lceil \mathbf{c} P = z : x \lceil \mathbf{e} P \in \mathbf{t}(P \, \underline{\mathbf{b}} \, R))$

$\Rightarrow$ [property 1.14 (3)]

$\Rightarrow$

$\quad z \in \mathbf{t}P\lceil\mathbf{c}P \wedge (\forall x : x \in \mathbf{t}P \wedge x\lfloor\mathbf{c}P = z$

$\qquad\qquad\qquad : (\exists y : y \in \mathbf{t}P \wedge y\lfloor\mathbf{c}P \in \mathbf{t}R : y\lceil\mathbf{e}P = x\lfloor\mathbf{e}P))$

$\Rightarrow\qquad$ [assumption implies $x\lceil\mathbf{c}P = y\lceil\mathbf{c}P$]

$\quad z \in \mathbf{t}P\lceil\mathbf{c}P \wedge (\forall x : x \in \mathbf{t}P \wedge x\lceil\mathbf{c}P = z : x\lceil\mathbf{c}P \in \mathbf{t}R)$

$\Rightarrow$

$\quad z \in \mathbf{t}R$

(end of proof)

3.4.1 Relation to conventional system theory

In [JCW] observability on a dynamical system $\Sigma = (T, W, B)$, with $W = W_1 \times W_2$ and $B \subseteq B_1 \times B_2$, with $B_i = P_{W_i}(B)$ the projection on W_i, is defined by:

$w_2 : T \to W_2$ is observable from $w_1 : T \to W_1$ if there exists a function
$F : B_1 \to B_2$ with $((w_1, w_2) \in B) \Leftrightarrow (w_2 = F(w_1))$.

Translating this to our definition of discrete systems leads to:

$z \in P\lceil\mathbf{e}P$ is observable from $y \in P\lfloor\mathbf{c}P$ if there is some function
$F : P\lceil\mathbf{c}P \to P\lceil\mathbf{e}P$ with $(\exists x : x \in \mathbf{t}P : x\lceil\mathbf{e}P = z \wedge x\lceil\mathbf{c}P = y) \Leftrightarrow (z = F(y))$

If we define F as

$F(y) = \{x : x \in \mathbf{t}P \wedge x\lceil\mathbf{c}P = y : x\lceil\mathbf{e}P\}$

we see that F is a function if and only if P is observable according to our definition.

We conclude that our definition of observability meets the general meaning of the notion in system theory.

3.4.2 CODE for observable processes

From lemma 3.21 it is clear that every solution of CODE for an observable discrete process P has the form $F(L)$. In that case, the deCODEr gives all possible solutions.

From lemma 3.12 we see that

$\quad L_{min} \subseteq L$

$\Rightarrow$

$\quad F(L_{min}) \subseteq F(L)$

holds for every L satisfying the minmax condition. If P is observable, all solutions are of the form $F(L)$. We conclude that for an observable P the solution $F(L_{min})$ is minimal:

Theorem 3.22 _If CODE has a solution, then $F(L_{max})$ is the largest solution contained in $P\lfloor\mathbf{c}P$. If in addition P is observable, then $F(L_{min})$ is the least solution contained in $P\lfloor\mathbf{c}P$._

proof: Combination of previous results.

If P is observable, we also have that $G(P \underline{\mathbf{b}} R) = P \underline{\mathbf{b}} R$. This property, however, holds for every P and every solution R:

Lemma 3.23 *For every solution $R \subseteq P \lceil \mathbf{c}P$ of CODE, we have:*

$$G(P \underline{\mathbf{b}} R) = P \underline{\mathbf{b}} R$$

proof: From lemma 3.9 we have (take $L = P \underline{\mathbf{b}} R$) that $G(P \underline{\mathbf{b}} R) \subseteq P \underline{\mathbf{b}} R$. So it remains to prove $G(P \underline{\mathbf{b}} R) \supseteq P \underline{\mathbf{b}} R$. We have:

$$x \lceil \mathbf{c}P \in \mathbf{t}R$$
$\Rightarrow \quad [\text{ take } u = y]$
$\quad (\forall y : y \in \mathbf{t}P \wedge y \lceil \mathbf{c}P = x \lceil \mathbf{c}P : (\exists u : u \in \mathbf{t}P \wedge u \lceil \mathbf{c}P \in \mathbf{t}R : y \lceil \mathbf{e}P = u \lceil \mathbf{e}P))$
$\Leftrightarrow \quad [\text{ definition of } \underline{\mathbf{b}}]$
$\quad (\forall y : y \in \mathbf{t}P \wedge y \lceil \mathbf{c}P = x \lceil \mathbf{c}P : y \lceil \mathbf{e}P \in \mathbf{t}(P \underline{\mathbf{b}} R))$

Hence:

$$z \in \mathbf{t}(P \underline{\mathbf{b}} R)$$
$\Leftrightarrow \quad [\text{ definition of } \underline{\mathbf{b}}]$
$\quad (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P \in \mathbf{t}R : z = x \lceil \mathbf{e}P)$
$\Rightarrow \quad [\text{ above implication }]$
$\quad (\exists x : x \in \mathbf{t}P \wedge (\forall y : y \in \mathbf{t}P \wedge y \lceil \mathbf{c}P = x \lceil \mathbf{c}P : y \lceil \mathbf{e}P \in \mathbf{t}(P \underline{\mathbf{b}} R)) : z = x \lceil \mathbf{e}P)$
$\Leftrightarrow \quad [\text{ lemma 3.7 }]$
$\quad z \in g(P \underline{\mathbf{b}} R)$

(end of proof)

We conclude this section with a summary of the results found:

Theorem 3.24 *In the CODE-problem we can draw the following conclusions:*

- *If CODE has a solution, we can find one by constructing $R = F(L)$, with L satisfying: $L_{min} \subseteq L \subseteq L_{max}$ and L sufficiently large.*
- *If not even $R = F(L_{max})$ leads to a solution, no solution of CODE exists.*
- *If CODE has a solution and P is observable, all solutions are of the form $R = F(L)$, with L satisfying: $L_{min} \subseteq L \subseteq L_{max}$*

3.5   A more general setting for CODE

So far, we have only considered CODE in the special case in which exactly all communications of P are used to control exactly all exogenous events of P. However, it is possible to define CODE in a more general setting:

1) the controller R may also have exogenous events,[7]
2) R may have a set of communication events that is larger than P,
3) R may have a set of communication events that is smaller than P,
4) L_{min} and L_{max} may be subsets of only part of the exogenous behaviour of P.

The first two cases are of no interest: Extra exogenous or communication events in R do not contribute to the control of the exogenous events of P. The last two cases, however, are of interest. If R has a smaller set of communication events than P, we have to try to control the exogenous behaviour of P using only part of the communication events. If L_{min} and L_{max} are subsets of a partial exogenous behaviour of P, we only have to control a subset of all exogenous events of P.

---

[7]These events do not appear in $\mathbf{e}P$.

We extend the formulation of CODE in such a way that these possibilities are included also. We find the following more general setting of CODE:
Given

$$P = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{c}P \rangle$$
$$E \subseteq \mathbf{e}P$$
$$L_{min} \subseteq L_{max} \subseteq \mathbf{t}P \lceil E$$

Find

$$R = \langle \mathbf{t}R, \emptyset, \mathbf{c}R \rangle \qquad \text{with } \mathbf{c}R \subseteq \mathbf{c}P$$

Such that

$$L_{min} \subseteq \mathbf{t}(P \underline{\mathbf{b}} R) \lceil E \subseteq L_{max}$$

It is possible (as will be shown) to transform this extended CODE problem to the original one, find a solution, and retransform this solution to become a solution of the extended CODE problem.

We first deal with the case that $\mathbf{c}R \subset \mathbf{c}P$. If we may use only fewer communication events than are present in P, we (temporarily) use another discrete process P' in which $\mathbf{c}P' = \mathbf{c}R$ and the remaining communication events are considered as exogenous events:

$$P' = \langle \mathbf{t}P, \mathbf{e}P \cup (\mathbf{c}P \setminus \mathbf{c}R), \mathbf{c}R \rangle$$

We have enlarged the set of exogenous events and have to solve CODE in the case that not all exogenous events have to be controlled.

It turns out that it is satisfactory to consider only the case in which not all exogenous events have to be controlled and $\mathbf{c}P = \mathbf{c}R$ (if not so, first do the transformation as prescribed above).
So consider:

$$P = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{c}P \rangle$$
$$L_{min} \subseteq L_{max} \subseteq P \lceil E \qquad \text{with } E \subset \mathbf{e}P$$

and try to find $R = \langle \mathbf{t}R, \emptyset, \mathbf{c}P \rangle$ such that:

$$L_{min} \subseteq (P \underline{\mathbf{b}} R) \lceil E \subseteq L_{max}$$

To find a solution for this problem, we extend L_{min} and L_{max} to become subsets of $P \lceil \mathbf{e}P$ as follows:

$$L^e_{min} = L_{min} \underline{\mathbf{s}} P \lceil \mathbf{e}P$$
$$L^e_{max} = L_{max} \underline{\mathbf{s}} P \lceil \mathbf{e}P$$

We have arranged that L^e_{min} and L^e_{max} are subsets of $P \lceil \mathbf{e}P$ and that the desired behaviour of events from E is still prescribed by L^e_{min} and L^e_{max}. The CODE problem so arranged is denoted by CODE^e.

We will prove that finding a solution for CODE^e leads directly to the solution for the original CODE-problem, i.e., if $R = \langle \mathbf{t}R, \emptyset, \mathbf{c}P \rangle$ is a solution for CODE^e, then it is a solution for CODE.

Therefore, it remains to prove that (with $L^e_{min} \subseteq L \subseteq L^e_{max}$):

$$L^e_{min} \subseteq G(L) \subseteq L^e_{max}$$
$$\Rightarrow$$
$$L_{min} \subseteq G(L) \lceil E \subseteq L_{max}$$

First notice:

$$L^e_{min} \lceil E$$
$$= \quad [\text{ definition of } L^e_{min}]$$
$$(L_{min} \text{ } \underline{\text{s}} \text{ } P\lceil eP)\lceil E$$
$$= \quad [\text{ property 1.21 (3), note: } \mathbf{a}L_{min} \cap \mathbf{a}(P\lceil eP) = E \subseteq E]$$
$$L_{min} \lfloor E \text{ } \underline{\text{s}} \text{ } P\lceil eP\lfloor E$$
$$= \quad [P\lceil eP\lfloor E = P\lfloor E \text{ and property 1.20 (1) }]$$
$$L_{min} \cap P\lfloor E$$
$$= \quad [L_{min} \subseteq P\lceil E]$$
$$L_{min}$$

and, similarly:

$$L^e_{max} \lfloor E = L_{max}$$

We have:

$$L^e_{min} \subseteq G(L) \subseteq L^e_{max}$$
$$\Rightarrow \quad [\lceil \text{ is monotonic }]$$
$$L^e_{min} \lceil E \subseteq G(L) \lceil E \subseteq L^e_{max} \lceil E$$
$$\Leftrightarrow$$
$$L_{min} \subseteq G(L) \lfloor E \subseteq L_{max}$$

So, in general, each solution of $CODE^e$ is a solution of CODE.

It turns out that control of only part of the exogenous events means that we do not give any constraint for the uncontrolled exogenous events. All possibilities of occurrence of this uncontrolled behaviour (as is possible in P) are simply copied to the constraints L_{min} and L_{max} by means of the shuffle operator.

In the sequel we always consider the CODE-problem as formulated in the beginning of this chapter. In examples we sometimes use the more general CODE-problem and solve the corresponding $CODE^e$-problem.

Example 3.25 Reconsider example 3.1:

$$P = \langle (ae|ad|ag|be|cg), \{d, e, g\}, \{a, b, c\} \rangle$$
$$\mathbf{t}L_{min} = e$$
$$\mathbf{t}L_{max} = (e|g)$$
$$\mathbf{c}R = \{a, b\}$$

Applying the above construction yields:

$$P' = \langle (ae|ad|ag|be|cg), \{d, e, g, c\}, \{a, b\} \rangle$$
$$E = \{d, e, g\}$$
$$L^e_{min} = L_{min} \text{ } \underline{\text{s}} \text{ } P\lceil eP = \langle e, \{d, e, g, c\}, \varnothing \rangle$$
$$L^e_{max} = L_{max} \text{ } \underline{\text{s}} \text{ } P\lceil eP = \langle (e|g|cg), \{d, e, g, c\}, \varnothing \rangle$$

This leads to

$$\begin{aligned}
\mathbf{t}(\neg L^e_{max}) &= d \\
f(P', L^e_{max}) &= \mathbf{t}((P' \text{ } \mathbf{b} \text{ } L^e_{max}) \setminus (P' \text{ } \mathbf{b} \text{ } \neg L^e_{max})) \\
&= (a|b|\epsilon) \setminus (a) \\
&= (b|\epsilon) \\
g(P', L^e_{max}) &= (e|cg)
\end{aligned}$$

The controller $R = \langle (b|\epsilon), \varnothing, \{a, b\} \rangle$ has the property

$$\mathbf{t}(P \ \underline{\mathbf{b}} \ R) \lceil \{d, e, g\} = (e|cg) \lceil \{d, e, g\} = (e|g)$$

so satisfies the minmax condition.

The occurrence of the behaviour cg in $P \ \underline{\mathbf{b}} \ R$ is due to the fact that c does not contribute in the connection of P and R. If we do not want cg, it can be removed by connecting a second controller R_1 to $P \ \underline{\mathbf{b}} \ R$ with $R_1 = \langle \epsilon, \varnothing, \{c\} \rangle$.

(end of example)

Regular discrete processes

What shall we use to fill the empty spaces
Where we used to talk
How shall I fill the final places
How shall I complete the wall

In this chapter we deal with a special kind of discrete process, namely the regular discrete process. Regular processes can be represented using finite state graphs, so part of this chapter treats finite state graphs in more detail. It turns out that for regular processes the deCODEr algorithm can be performed using finite state graphs. For regular discrete processes we therefore can check *effectively* whether the CODE-problem has a solution and if so, we can construct a controller effectively , i.e., by using only a finite number of steps.

4.1 Regular processes

In the first chapter a definition is given for regular trace structures. Because of the similarity between trace structures and discrete processes it is not surprising that the following definition specifies a regular discrete process:

Definition 4.1 *A discrete process P is called* regular *if the corresponding trace structure* $\mathbf{ts}(P)$ *is regular.*

As a consequence, a regular discrete process can be represented as a finite state graph or, equivalently, its behaviour consists of a regular expression (as we have frequently used in the previous chapters). The class of regular discrete processes over a certain alphabet A (being the union of the exogenous and the communication events) is denoted by $\mathbf{R}(A)$.

Lemma 4.2 *The class of regular discrete processes is closed under connections, total connections, joining, and concatenations, i.e., if P and R are regular, then P $\underline{\mathbf{b}}$ R, P $\underline{\mathbf{w}}$ R, P $\underline{\mathbf{s}}$ R, and P ; R are regular.*

proof: Immediately from the fact that the class of regular trace structures is closed under weaving and blending (see chapter 3 of [JvdS]).
(end of proof)

45

Moreover we have:

Lemma 4.3

$$(\forall P : P \in \mathbf{R}(\mathbf{a}P) : L \in \mathbf{R}(\mathbf{e}P) \Rightarrow F(L) \in \mathbf{R}(\mathbf{c}P))$$

proof: Follows directly from the fact that $F(L)$ is defined using connection and set exclusion and these operators preserve the notion of regularity (see [HoUl]).
(end of proof)

4.2   Combining finite state graphs

In case of regular discrete processes it is possible to construct a controller using the state graphs directly. Because these graphs have a finite number of states, it is then possible to construct this controller using an algorithm that contains only a finite number of steps. In this section we give a translation of the deCODEr for finite state graphs.

For using the deCODEr directly on finite state graphs, we have to translate the operations **b** , $\lceil$, and $\backslash$ (blending, alphabet restriction, and exclusion)[1] in algorithms for finite state graphs.

In the sequel we assume that every finite state graph is *complete* (or *completely specified*), i.e., the transition map δ is defined for every pair (p, a) with $p \in Q$ and $a \in A$. Every finite state graph M that is not complete can easily be made complete: we then construct a complete graph M_c out of M that is equivalent with M (in the sense that M and M_c accepts the same strings) by introducing one extra state $[\varnothing]$, called the *error state* (i.e., $Q_c = Q \cup \{[\varnothing]\}$), and constructing a new transition function δ_c as follows:

$$\delta_c(p, a) = \delta(p, a) \quad \text{if } \delta(p, a) \text{ is defined,}$$
$$\delta_c(p, a) = [\varnothing] \quad \text{if } \delta(p, a) \text{ is not defined,}$$
$$\delta_c([\varnothing], a) = [\varnothing] \quad \text{for every } a \in A.$$

For a discrete process P the graph $\mathbf{sg}(P)$ is complete by definition.

In the sequel we write $\delta(p, a) = q$ if a transition labeled a from state p to state q exists and use $\delta(p, \epsilon)$ as another notation for p (to ease notation in definition 4.5).

If we use alphabet restriction (directly or when computing a blend), we derive a non-deterministic graph, i.e., all transitions labeled with an event not in the alphabet are replaced by a transition labeled with ϵ.

In general, a non-deterministic state graph is a state graph with the transition function δ replaced by a mapping into $\mathbf{2}^Q$, i.e., it is possible that from state p a transition labeled a to state q exists and also a transition labeled a to state q' with $q \neq q'$. Also ϵ-transitions are possible in a non-deterministic state graph. In definition 4.5 we only have non-deterministic state graphs containing ϵ-transitions. Formally, the definition of a non-deterministic graph is:

---

[1] In fact we have to translate connection, total connection, and exclusion on discrete processes in algorithms for finite state graphs. However the essential calculations of for example the connection is the blend between two trace sets, so we restrict our attention on trace structures here and use the results on discrete processes.

Definition 4.4 *A non-deterministic state graph M_{nd} is defined by:*

$$M_{nd} = (A, Q, d, q_0, F)_{nd}$$

with:

A	*the alphabet*
Q	*the states of the graph*
$d : Q \times (A \cup \{\epsilon\}) \to 2^Q$	*the state transition map*
$q_0 \in Q$	*the initial state*
$F \subseteq Q$	*the final states*

We always have (by definition) $p \in d(p, \epsilon)$.

It is known (see [HoUl]) that every non-deterministic finite state graph can be made deterministic, such that it still accepts the same language.

If M_{nd} is a non-deterministic finite state graph, we denote by

det(M_{nd})

its deterministic equivalent. See [HoUl] for an algorithm to derive this deterministic graph.

Definition 4.5 *Let $M_1 = (A_1, Q_1, \delta_1, q_{01}, F_1)$ and $M_2 = (A_2, Q_2, \delta_2, q_{02}, F_2)$, then we define:*

(a) $M_1 \mathbf{w} M_2 = (A_1 \cup A_2, Q_1 \times Q_2, \delta, (q_{01}, q_{02}), F_1 \times F_2)$
 with: $(\forall p, q, a : p \in Q_1 \wedge q \in Q_2 \wedge a \in A_1 \cup A_2$
 $: \delta((p, q), a) = (\delta_1(p, a \lceil A_1), \delta_2(q, a \lceil A_2)))$

(b) $M_1 \mathbf{b} M_2 = \mathbf{det}(A_1 \div A_2, Q_1 \times Q_2, \delta, (q_{01}, q_{02}), F_1 \times F_2)_{nd}$
 with: $(\forall p, q, a : p \in Q_1 \wedge q \in Q_2 \wedge a \in A_1 \cup A_2$
 $: (a \in A_1 \div A_2 \Rightarrow \delta((p, q), a) = \{(\delta_1(p, a \lceil A_1), \delta_2(q, a \lceil A_2))\}) \wedge$
 $(a \in A_1 \cap A_2 \Rightarrow (\delta_1(p, a \lceil A_1), \delta_2(q, a \lceil A_2)) \in \delta((p, q), \epsilon))$

and in case $A_1 = A_2 = A$:

(c) $M_1 \cup M_2 = (A, Q_1 \times Q_2, \delta, (q_{01}, q_{02}), F)$
 with: $(\forall p, q, a : p \in Q_1 \wedge q \in Q_2 \wedge a \in A$
 $: \delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a)))$
 and: $F = \{p, q : p \in F_1 \vee q \in F_2 : (p, q)\}$

(d) $M_1 \cap M_2 = (A, Q_1 \times Q_2, \delta, (q_{01}, q_{02}), F_1 \times F_2)$
 with δ as in *(c)*

(e) $M_1 \setminus M_2 = (A, Q_1 \times Q_2, \delta, (q_{01}, q_{02}), F)$
 with δ as in *(c)*
 and: $F = \{p, q : p \in F_1 \wedge q \notin F_2 : (p, q)\}$

(f) $\neg M_1 = (A_1, Q_1, \delta_1, q_{01}, Q_1 \setminus F_1\}$

(g) $\mathbf{pref}(M_1) = (A_a, Q_1, \delta_1, q_{01}, Q_1)$

and for $A \subseteq A_1$:

(h) $M_1 \lceil A = (A, Q_1, \delta, q_{01}, F_1)$
 with: $(\forall p, a : p \in Q_1 \wedge a \in A_1$
 $: (a \in A \Rightarrow \delta(p, a) = \{\delta_1(p, a)\}) \wedge$
 $(a \notin A \Rightarrow \delta_1(p, a) \in \delta(p, \epsilon)))$

Once again we repeat that the above definitions are only valid if used on complete state

graphs. Incomplete state graphs give weird results.

Lemma 4.6 *If M_1 and M_2 are finite state graphs, then we have:*

(a) $\mathbf{ts}(M_1) \mathbf{w} \, \mathbf{ts}(M_2) = \mathbf{ts}(M_1 \mathbf{w} M_2)$

(b) $\mathbf{ts}(M_1) \mathbf{b} \, \mathbf{ts}(M_2) = \mathbf{ts}(M_1 \mathbf{b} M_2)$

If the graph alphabets are the same, we also have:

(c) $\mathbf{ts}(M_1) \cup \mathbf{ts}(M_2) = \mathbf{ts}(M_1 \cup M_2)$

(d) $\mathbf{ts}(M_1) \cap \mathbf{ts}(M_2) = \mathbf{ts}(M_1 \cap M_2)$

(e) $\mathbf{ts}(M_1) \setminus \mathbf{ts}(M_2) = \mathbf{ts}(M_1 \setminus M_2)$

Furthermore, if A is the alphabet of graph M, then

(f) $\langle A^*, A \rangle \setminus \mathbf{ts}(M)) = \mathbf{ts}(\neg M)$

(g) $\mathbf{pref}(\mathbf{ts}(M)) = \mathbf{ts}(\mathbf{pref}(M))$

If in addition $A_1 \subseteq A$, then

(h) $\mathbf{ts}(M) \lceil A_1 = \mathbf{ts}(M \lceil A_1)$

proof: obvious.

We are now able to express the calculation of $F(L)$ in terms of the above operators:

Lemma 4.7

$$\mathbf{sg}(F(L)) = (\mathbf{sg}(P) \mathbf{b} \, \mathbf{sg}(L)) \setminus (\mathbf{sg}(P) \mathbf{b} \neg \mathbf{sg}(L))$$

proof: Trivial. Notice that

$$f(L) = \{z : z \in \mathbf{t}P \lceil \mathbf{e}P \wedge (\forall x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}L) : z\}$$

and

$$\mathbf{t}(P \, \underline{\mathbf{b}} \, L)$$
$$=$$
$$\{z : z \in \mathbf{t}P \lceil \mathbf{e}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \in \mathbf{t}L) : z\}$$

so that

$$f(L)$$
$$=$$
$$\mathbf{t}(P \, \mathbf{b} \, L) \setminus \{z : z \in \mathbf{t}P \lceil \mathbf{e}P \wedge (\exists x : x \in \mathbf{t}P \wedge x \lceil \mathbf{c}P = z : x \lceil \mathbf{e}P \notin \mathbf{t}L) : z\}$$

from which the above algorithm to compute $\mathbf{sg}(F(L))$ follows.
(end of proof)

Furthermore, the condition $L_{min} \subseteq G(P, L_{max})$ can easily be checked using

$$M = (\mathbf{sg}(P) \mathbf{b} \, \mathbf{sg}(F(L_{max}))) \setminus \mathbf{sg}(L_{min})$$

If M results in an empty state graph, i.e., its initial state equals $[\varnothing]$, the condition is fulfilled and $\mathbf{sg}(F(L_{max}))$ represents a controller solving CODE. Otherwise no solution is possible.

If for some L between L_{min} and L_{max} we have that

$$M = (\mathbf{sg}(P) \mathbf{b} \, \mathbf{sg}(F(L))) \setminus \mathbf{sg}(L_{min})$$

is an empty graph, then $\mathbf{sg}(F(L))$ is a representation for a controller solving CODE.
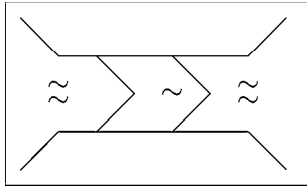
Figuur 4.1: A ship lock

event	meaning
p_1	a ship passes through door 1
p_2	a ship passes through door 2
o_1	open door 1
o_2	open door 2
c_1	close door 1
c_2	close door 2

Tabel 4.1: Meaning of the events of the lock

It is not difficult to make a computer program to compute $\mathbf{sg}(F(L))$ from $\mathbf{sg}(P)$ and $\mathbf{sg}(L)$ according to this lemma. Such a program is used in this thesis to do the computations in the examples.

4.3 A ship lock

As an example of the use of the deCODEr we look at the following situation. Consider a ship lock with two doors in which ships can pass from west to east (see figure 4.1). The lock is given by:

$$P = <\mathbf{t}P, \mathbf{e}P, \mathbf{c}P>$$

with

$$\mathbf{e}P = \{p_1, p_2\}$$
$$\mathbf{c}P = \{o_1, o_2, c_1, c_2\}$$

The behaviour is given in figure 4.2. The meaning of the events is given in table 4.1. The lock can contain one ship at the time. The desired behaviour therefore is:
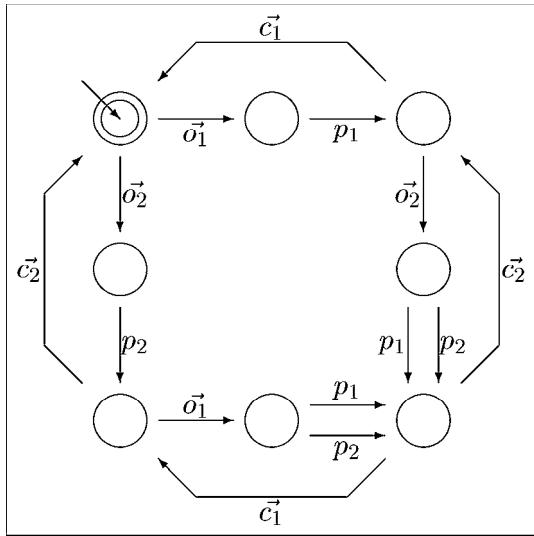
Figuur 4.2: Behaviour of the lock
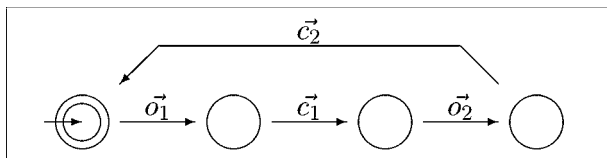
Figuur 4.3: Controller for the lock
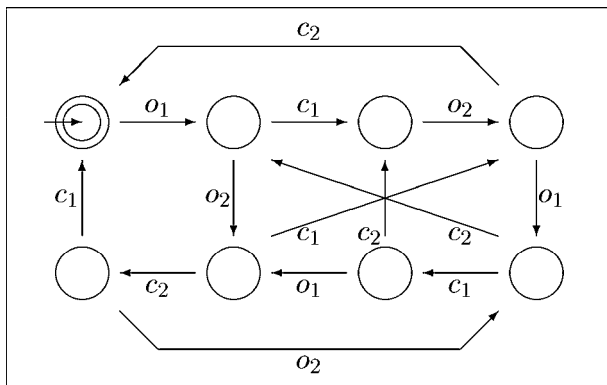
*Figuur 4.4: P **b** L*

$$L_{min} = L_{max} = L = < (p_1 p_2)^*, \mathbf{e}P, \emptyset >$$

Using the deCODEr[2] we find the controller as in figure 4.3. This controller does precisely what we expected it should do: first, let a ship in by opening and closing door 1; next, let the ship go out by opening and closing door 2.

In figure 4.4 we have given P **b** L. Just computing P **b** L in general does not give the right controller: in P **b** L, for example the behaviour $o_1 o_2 c_1 c_2$ is possible. This may

---

[2]We have used a computer program here, so no calculations have been given.

lead to p_1p_2, but also p_1p_1 is possible and this last exogenous behaviour is certainly not desired.

It can easily be verified that the exogenous behaviour of the connection equals:

t$(P$ **b** $R) = (p_1p_2)^*$

and the total behaviour:

t$(P$ **w** $R) = (o_1p_1c_1o_2p_2c_2)^*$

5
Related problems

In this chapter we discuss some problems that are closely related to the CODE problem and can be transformed to CODE or use the deCODEr to find a solution. First, we deal with the problem of *regulation*. Next, we discuss the *extended control problem*, which has some analogy to *supervisory control theory* of Ramadge and Wonham (see [RaWo]). Special attention will be given to the relation between CODE and supervisory control theory.

5.1 Regulation

Regulation of a discrete process P means finding a controller R, such that the connection of P with R results in some desired exogenous behaviour *after some steps*. The notion *after some steps* can be interpreted in at least two ways:

1) after at most a finite number of occurrences of (exogenous) events,
2) after occurrence of some behaviour that is given beforehand.

Example 5.1 Consider the process

$$P = \langle ((ae)^*(bg)^*), \{e,g\}, \{a,b\} \rangle$$

P can be regulated to behave according to

$$L = \langle g^*, \{g\}, \varnothing \rangle$$

after (for example) at most 2 occurrences of exogenous events if we use

$$R = \langle (((\epsilon|a|aa)b^*), \varnothing, \{a,b\} \rangle$$

However, the process

$$S = \langle (ae^*(bg)^*), \{e,g\}, \{a,b\} \rangle$$

cannot be regulated in that sense: it cannot be controlled how many occurrences of e precede the desired behaviour g^*. P can, however, be regulated in the other sense:

$$R = \langle (ab^*), \varnothing, \{a, b\} \rangle$$

regulates P such that $P \underline{\mathbf{b}} R$ behaves according to g^* after behaviour e^*.
(end of example)

First, we define what we mean by "behaviour after some other behaviour."

Definition 5.2 *For L and L_p with $\mathbf{e}L_p = \mathbf{e}L$ and $\mathbf{c}L_p = \mathbf{c}L$, we define the behaviour of L after L_p (denoted by $L \rfloor L_p$ and pronounced "L after L_p") by:*

$$L \rfloor L_p = \langle \{t : (\exists x, u : x \in \mathbf{t}L \ \wedge \ u \in \mathbf{t}L_p : x = ut) : t\}, \mathbf{e}L, \mathbf{c}L \rangle$$

Formally, we can now define the above problem by:
Given

$$P = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{c}P \rangle$$
$$L_{pre} \subseteq \mathbf{pref}(P \lceil \mathbf{e}P)$$
$$L'_{min} \subseteq L'_{max} \subseteq (P \lceil \mathbf{e}P) \rfloor L_{pre}$$

find, if possible, a controller

$$R = \langle \mathbf{t}R, \varnothing, \mathbf{c}P \rangle$$

such that

$$L'_{min} \subseteq (P \underline{\mathbf{b}} R) \rfloor L_{pre} \subseteq L'_{max} \qquad \text{(the min'max'condition)}$$

This problem is called *regulation of discrete events* (RODE for short). If we only want a maximum of N exogenous events to occur before reaching the desired behaviour, we can use

$$L_{pre} = \langle \{x : x \in \mathbf{pref}(\mathbf{t}P \lceil \mathbf{e}P) \ \wedge \ |x| \leq N : x\}, \mathbf{e}P, \varnothing \rangle$$

First, we list a number of properties of the operator $\rfloor$:

Property 5.3 [1]

$$L_p \subseteq \mathbf{pref}(L) \ \Rightarrow \ L \subseteq L_p \ ; L \rfloor L_p$$

Example 5.4 Consider L and L_p with $\mathbf{t}L = (bab|cac)$ and $\mathbf{t}L_p = (b|c)$, then we have:

$$\mathbf{t}(L \rfloor L_p) = (ab|ac)$$
$$\mathbf{t}(L_p \ ; L \rfloor L_p) = (b|c)(ab|ac) = (bab|bac|cab|cac)$$

so, in general, $L \neq L_p \ ; L \rfloor L_p$.
(end of example)

---

[1]Notice that $L_p \ ; L \rfloor L_p$ should be evaluated as $L_p \ ; (L \rfloor L_p)$.

Property 5.5

$$(\forall x_1, x_2 : x_1 \in \mathbf{t}L_p \ \wedge \ x_2 \in \mathbf{t}L_p : [x_1]_L = [x_2]_L) \ \wedge \ L_p \subseteq \mathbf{pref}(L)$$
$$\Rightarrow$$
$$L = L_p \ ; L \rfloor L_p$$

proof: Notice

$$u \in \mathbf{t}L_p \ \wedge \ w \in \mathbf{t}L_p$$
$$\Rightarrow \quad [\text{ assumption }]$$
$$[u]_L = [w]_L$$
$$\Leftrightarrow \quad [\text{ definition of equivalence classes }]$$
$$(\forall z : z \in (\mathbf{a}L)^* : uz \in \mathbf{t}L = wz \in \mathbf{t}L)$$

Hence:

$$x \in \mathbf{t}(L_p \ ; L \rfloor L_p)$$
$$\Leftrightarrow \quad [\text{ definition of } ; \ ]$$
$$(\exists u, t : u \in \mathbf{t}L_p \ \wedge \ t \in \mathbf{t}(L \rfloor L_p) : x = ut)$$
$$\Leftrightarrow \quad [\text{ definition of } \rfloor \ ]$$
$$(\exists u, t : u \in \mathbf{t}L_p \ \wedge \ (\exists y, w : y \in \mathbf{t}L \ \wedge \ w \in \mathbf{t}L_p : y = wt) : x = ut)$$
$$\Leftrightarrow$$
$$(\exists u, t, y, w : u \in \mathbf{t}L_p \ \wedge \ y \in \mathbf{t}L \ \wedge \ w \in \mathbf{t}L_p : y = wt \ \wedge \ x = ut)$$
$$\Rightarrow \quad [\text{ above implication and } y \in \mathbf{t}L \ ]$$
$$x \in \mathbf{t}L$$

(end of proof)

The assumption says that all traces of L_p belong to the same equivalence class of L, i.e., L_p is a prefix of L with all paths ending in the same state in L.

Property 5.6

$$L_1 \subseteq L_2 \ \Rightarrow \ L_1 \rfloor L_p \subseteq L_2 \rfloor L_p$$

We are able now to reformulate the min'max'condition in our original minmax condition of the CODE problem, thus relating RODE to CODE.
Let

$$L_{min} = L_{pre} \ ; L'_{min} \quad L_{max} = L_{pre} \ ; L'_{max}$$

From property 5.3 we have $L_{min} \rfloor L_{pre} \supseteq L'_{min}$. From property 5.5 we have $L_{max} \rfloor L_{pre} = L'_{max}$ if

$$(\forall x_1, x_2 : x_1 \in \mathbf{t}L_{pre} \ \wedge \ x_2 \in \mathbf{t}L_{pre} : [x_1]_{L_{max}} = [x_2]_{L_{max}})$$

hence:

$$L_{min} \subseteq P \ \underline{\mathbf{b}} \ R \subseteq L_{max}$$
$$\Rightarrow \quad [\text{ property 5.6 }]$$
$$L_{min} \rfloor L_{pre} \subseteq (P \ \underline{\mathbf{b}} \ R) \rfloor L_{pre} \subseteq L_{max} \rfloor L_{pre}$$
$$\Leftrightarrow \quad [\text{ see above }]$$

$\Leftrightarrow$

$\qquad L'_{min} \subseteq (P \ \underline{\mathbf{b}} \ R) \rfloor L_{pre} \subseteq L'_{max}$

$\Rightarrow \qquad [\ L_1 \subseteq L_2 \Rightarrow L_1 \ ; L \subseteq L_2 \ ; L \ ]$

$\qquad L_{pre} \ ; L'_{min} \subseteq L_{pre} \ ; (P \ \underline{\mathbf{b}} \ R) \rfloor L_{pre} \subseteq L_{pre} \ ; L'_{max}$

$\Leftrightarrow$

$\qquad L_{min} \subseteq L_{pre} \ ; (P \ \underline{\mathbf{b}} \ R) \rfloor L_{pre} \subseteq L_{max}$

$\Rightarrow \qquad [\ \text{see below}\ ]$

$\qquad L_{min} \subseteq P \ \underline{\mathbf{b}} \ R \subseteq L_{max}$

The last implication is only true if (according to property 5.5):

$$(\forall x_1, x_2 : x_1 \in L_{pre} \ \wedge \ x_2 \in L_{pre} : [x_1]_{P \ \underline{\mathbf{b}} \ R} = [x_2]_{P \ \underline{\mathbf{b}} \ R})$$

We find:

Lemma 5.7 *With $L = L_{pre} \ ; L'_{max}$, we have*

$\qquad L_{pre} \ ; L'_{min} \subseteq G(L_{pre} \ ; L'_{max}) \ \wedge$

$\qquad (\forall x_1, x_2 : x_1 \in \mathbf{t}L_{pre} \ \wedge \ x_2 \in \mathbf{t}L_{pre} : [x_1]_L = [x_2]_L)$

$\Rightarrow$

$\qquad RODE$ *is solvable*

A possible controller then is $F(L_{pre} \ ; L'_{max})$.

Lemma 5.8 *With $R = F(L_{pre} \ ; L'_{max})$ and $S = P \ \underline{\mathbf{b}} \ R$, we have:*

$\qquad L_{pre} \ ; L'_{min} \nsubseteq G(L_{pre} \ ; L'_{max}) \ \wedge$

$\qquad (\forall x_1, x_2 : x_1 \in L_{pre} \ \wedge \ x_2 \in L_{pre} : [x_1]_S = [x_2]_S)$

$\Rightarrow$

$\qquad RODE$ *is not solvable*

Example 5.9 Reconsider

$\qquad P = \langle ((ae)^*(bg)^*), \{e, g\}, \{a, b\} \rangle$

$\qquad L'_{min} = L'_{max} = \langle (g^*), \{e, g\}, \varnothing \rangle$

$\qquad N = 2$

then we have

$\qquad L_{pre} = \langle (gg|eg|ee), \{e, g\}, \varnothing \rangle$

$\qquad L_{pre} \ ; L'_{max} = \langle (\epsilon|e|ee)g^*), \{e, g\}, \varnothing \rangle$

i.e., in $L_{pre} \ ; L'_{max}$ after at most 2 occurrences of events the behaviour is according to L'_{max}. Notice that in $L_{max} = L_{pre} \ ; L'_{max}$ the condition of lemma 5.7 is met. Computing $f(L_{pre} \ ; L'_{max})$ results in

$\qquad (\epsilon|a|aa)b^*$

and indeed $G(L_{pre} \ ; L'_{max}) = L_{pre} \ ; L'_{max}$ so

$\qquad R = \langle f(L_{pre} \ ; L'_{max}), \varnothing, \{a, b\} \rangle$

is our desired controller.
(end of example)
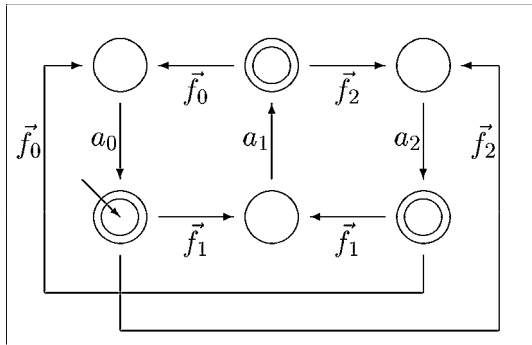
Figuur 5.1: Behaviour of the elevator

Example 5.10 Consider an elevator in some building with 3 floors. The corresponding discrete process P equals $\langle \mathbf{t}P, \{a_0, a_1, a_2\}, \{f_0, f_1, f_2\}\rangle$ with $\mathbf{t}P$ given in figure 5.1 and the meaning of the events as follows:

a_i elevator has arrived at floor i
f_i elevator is sent to floor i

Suppose a person at floor 2 wants to go to floor 1. Initially, the elevator is at floor 0. The desired behaviour is $\mathbf{t}L' = (a_2 a_1)$. However, it takes a number of steps before the elevator reaches floor 2: at most one action is allowed, so we take

$$\mathbf{t}L_{pre} = \{x : x \in \mathbf{pref}(\mathbf{t}P\lceil\mathbf{e}P) \wedge |x| \leq 1 : x\}$$
$$= (\epsilon | a_1)$$

We use $L = L_{pre}$; L' and compute $R = F(L)$. We find $f(L) = (f_1 f_2 f_1 | f_2 f_1)$ and indeed $\mathbf{t}(P \underline{\mathbf{b}} R) = (a_1 a_2 a_1 | a_2 a_1)$. However, $\mathbf{t}((P \underline{\mathbf{b}} R)\rfloor L_{pre}) = (a_1 a_2 a_1 | a_2 a_1)$. The condition in lemma 5.7 is not met: $[\epsilon]_L \neq [a_1]_L$ (especially: $L\rfloor L_{pre} \neq L'$).

This problem can be solved if we use $\mathbf{t}L_{pre} = a_1$ (i.e. exactly one action is allowed before the behaviour should be as desired). Then we find $f(L) = f_1 f_2 f_1$ and indeed $\mathbf{t}(P \underline{\mathbf{b}} R) = a_1 a_2 a_1$ and $\mathbf{t}((P \underline{\mathbf{b}} R)\rfloor L_{pre}) = a_2 a_1$.
(end of example)

5.2 The extended control problem

Instead of looking at $L_{min} \subseteq P \underline{\mathbf{b}} R \subseteq L_{max}$, one might consider looking at

$$L_{min} \subseteq P \underline{\mathbf{s}} R \subseteq L_{max}$$

which has some similarity in supervisory control theory of Wonham and Ramadge (see [RaWo]). First, we give a definition and solution for this new problem. Next, we compare this problem (and CODE itself) with supervisory control.

In this section we discuss the following problem:
Given a discrete process P and two discrete processes L_{min} and L_{max} with

$$L_{min} = \langle \mathbf{t}L_{min}, \mathbf{e}P, \mathbf{c}P \rangle$$
$$L_{max} = \langle \mathbf{t}L_{max}, \mathbf{e}P, \mathbf{c}P \rangle$$
$$L_{min} \subseteq L_{max}$$

Find, if possible, a controller discrete process $R = \langle \mathbf{t}R, \varnothing, \mathbf{c}P \rangle$ such that

$$L_{min} \subseteq P \underline{\mathbf{s}} R \subseteq L_{max}$$

This problem is called the *extended control problem* or ECODE for short. Notice that the minmax condition restricts the total behaviour here. Without loss of generality we assume that

$$L_{min} \subseteq L_{max} \subseteq P$$

(for the same reason as before with CODE: we cannot create traces in $P \underline{\mathbf{s}} R$ that are not in P).

As we shall see, the solution of this problem is very much like the solution of CODE itself: we use the same friend of L, but have to subtract some extra traces from its behaviour to get the right controller.

To solve the problem, we need the following functions on discrete processes.

Definition 5.11 *With the ECODE problem we associate the discrete processes:*

$$E(P, L) = F(P, L\lceil \mathbf{e}P) \setminus (P \setminus L)\lceil \mathbf{c}P$$

called the extended friend *of L, and*

$$H(P, L) = P \underline{\mathbf{s}} E(P, L)$$

called the host[2] *of L.*

$F(P, L\lceil \mathbf{e}P)$ is a possible candidate for solving CODE. $E(P, L)$ is a possible candidate for solving ECODE (created by using $F(P, L\lceil \mathbf{e}P)$ and deleting all communicating traces that are not desired in ECODE); $H(P, L)$ is like the guardian in CODE: it gives the resulting behaviour using the extended friend.

When it is clear from the context we write $E(L)$ and $H(L)$ for short. Notice that for $L\lceil \mathbf{e}P \subseteq P$ we have:

$\mathbf{e}F(L\lceil \mathbf{e}P) = \varnothing$	$\mathbf{c}F(L\lceil \mathbf{e}P) = \mathbf{c}P$
$\mathbf{e}E(L) = \varnothing$	$\mathbf{c}E(L) = \mathbf{c}P$
$\mathbf{e}H(L) = \mathbf{e}P$	$\mathbf{c}H(L) = \mathbf{c}P$

The following theorem will not come as a surprise.

Theorem 5.12

> *ECODE is solvable*

$\Leftrightarrow$

> $L_{min} \subseteq H(L_{max})$

and in case ECODE is solvable a solution is $E(L_{max})$.

To prove the theorem, we need the following lemmas:

---

[2]**host** (n.) – One who receives or entertains another as guest.

Lemma 5.13

$$\mathbf{t}E(L) = \{z : z \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge \ (\forall x : x \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{c}P = z : x \in \mathbf{t}L) : z\}$$

proof:

$\mathbf{t}E(L)$
=   [definition of $E(L)$ and of $\setminus$]
$\mathbf{t}F(L\lceil\mathbf{e}P) \setminus \mathbf{t}(P \setminus L)\lceil\mathbf{c}P$
=   [lemma 3.7 and definition of $|$]
$\{z : z \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge \ (\forall x : x \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{c}P = z : x\lceil\mathbf{e}P \in \mathbf{t}L\lceil\mathbf{e}P) : z\}\setminus$
$\{z : (\exists x : x \in \mathbf{t}P \ \wedge \ x \notin \mathbf{t}L : x\lceil\mathbf{c}P = z) : z\}$
=
$\{z : z \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge \ (\forall x : x \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{c}P = z : x\lceil\mathbf{e}P \in \mathbf{t}L\lceil\mathbf{e}P \ \wedge \ x \in \mathbf{t}L) : z\}$
=   [$x \in \mathbf{t}L \ \Rightarrow \ x\lceil\mathbf{e}P \in \mathbf{t}L\lceil\mathbf{e}P$]
$\{z : z \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge \ (\forall x : x \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{c}P = z : x \in \mathbf{t}L) : z\}$

(end of proof)

Lemma 5.14

$$R \subseteq P\lceil\mathbf{c}P \ \wedge \ P \ \underline{\mathbf{s}} \ R \subseteq L \ \Rightarrow \ R \subseteq E(L)$$

proof:

$z \in \mathbf{t}R$
$\Rightarrow$ [$R \subseteq P\lceil\mathbf{c}P \ \wedge \ P \ \underline{\mathbf{s}} \ R \subseteq L$]
$z \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge \ (\forall x : x \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{c}P = z : x \in \mathbf{t}L)$
$\Rightarrow$ [lemma 5.13]
$z \in \mathbf{t}E(L)$

(end of proof)

Lemma 5.15

$$H(L) \subseteq L$$

proof:

$y \in \mathbf{t}H(L)$
$\Leftrightarrow$ [definition of $H(L)$]
$y \in \mathbf{t}(P \ \underline{\mathbf{s}} \ E(L))$
$\Leftrightarrow$ [definition of $\underline{\mathbf{s}}$]
$y \in \mathbf{t}P \ \wedge \ y\lceil\mathbf{c}P \in \mathbf{t}E(L)$
$\Leftrightarrow$ [lemma 5.13]
$y \in \mathbf{t}P \ \wedge \ (\forall x : x \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{c}P = y\lceil\mathbf{c}P : x \in \mathbf{t}L)$
$\Rightarrow$ [take $x = y$]
$z \in \mathbf{t}L$

(end of proof)
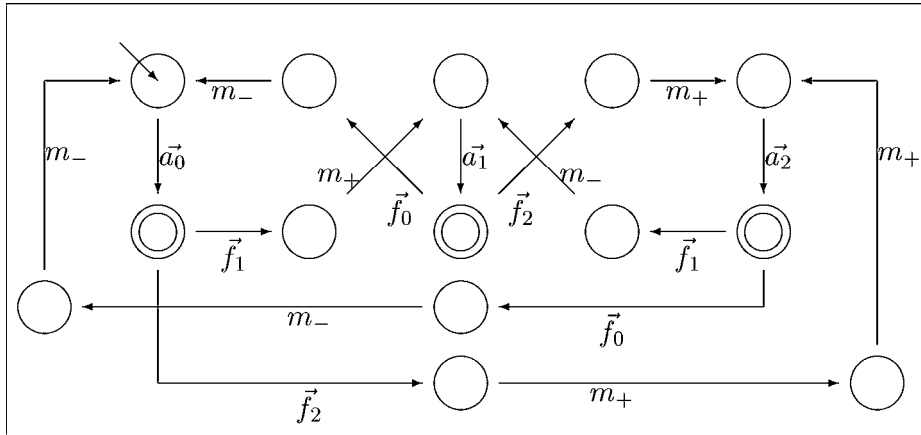
Figuur 5.2: Behaviour of the elevator

proof: (of theorem 5.12)

First, suppose R is a solution for ECODE, write $R = R_{int} \cup R_{ext}$ with:

$$R_{int} = P\lceil cP \cap R$$
$$R_{ext} = R \setminus R_{int}$$

(which gives $P \, \underline{s} \, R = P \, \underline{s} \, R_{int}$), then we have:

$\qquad L_{min}$
$\subseteq \qquad$ [R is a solution: $L_{min} \subseteq P \, \underline{s} \, R \subseteq L_{max}$]
$\qquad P \, \underline{s} \, R_{int}$
$\subseteq \qquad$ [lemma 5.14 (with R replaced by R_{int})]
$\qquad P \, \underline{s} \, E(L_{max})$
$= \qquad$ [definition of $H(L)$]
$\qquad H(L_{max})$

Next, suppose ECODE has no solution:

$\qquad (\forall R : P \, \underline{s} \, R \subseteq L_{max} : L_{min} \not\subseteq P \, \underline{s} \, R)$
$\Rightarrow \qquad$ [take $R = E(L_{max})$, such an R exists, see below]
$\qquad L_{min} \not\subseteq P \, \underline{s} \, E(L_{max})$
$\Leftrightarrow \qquad$ [definition of $H(L)$]
$\qquad L_{min} \not\subseteq H(L_{max})$

The existence of an $R = E(L_{max})$ in the last part of this proof follows from lemma 5.15 with $L = L_{max}$: because $H(L_{max}) \subseteq L_{max}$ we know that $P \, \underline{s} \, E(L_{max}) \subseteq L_{max}$.

(end of proof)

Example 5.16 Consider the following model of an elevator P, as given in figure 5.2, with events:

m_+ elevator going up one floor

m_- elevator going down one floor

f_i elevator needs to go to floor i

a_i elevator arrives at floor i

We suppose $\mathbf{e}P = \{m_+, m_-\}$ and $\mathbf{c}P = \{a_0, a_1, a_2, f_0, f_1, f_2\}$.

A person want to go from floor 1 to floor 2. Exogenous behaviour m_+m_+, however, is not enough to guarantee that the person can use the elevator. It should stop at floor 1 also. Therefore, we use the ECODE formulation here, i.e., we want to have the desired behaviour

$$\mathbf{t}L = a_0 f_1 m_+ a_1 f_2 m_+ a_2$$

Computations give:

$$
\begin{aligned}
\mathbf{t}L \lceil \mathbf{e}P &= m_+ m_+ \\
f(P, L \lceil \mathbf{e}P) &= (a_0 f_1 a_1 f_2 a_2 | a_0 f_2 a_2) \\
\mathbf{t}E(P, L) &= a_0 f_1 a_1 f_2 a_2
\end{aligned}
$$

and indeed $H(P, L) = L$.

(end of example)

5.3   Supervisory control

In this section we discuss the supervisory control theory of Ramadge and Wonham (see [RaWo]).[3] In this theory a discrete process is viewed as a sequential process that, in fact, is equal to a (possibly infinite) state graph. Each label in such a graph represents the occurrence of some event. The labels may be controlled in which case they can be enabled or disabled. A supervisor is a second graph that observes the behaviour of the first one and enables or disables the controlled events in order to get a predefined behaviour. Formally the definitions[4] are as follows:

Definition 5.17 *A sequential process is a tuple $G = (Q, \Sigma, \delta, q_0, Q_m)$ with*

Q *set of states (need not be finite)*

Σ *finite set of events*

$q_0 \in Q$ *initial state*

$Q_m \subseteq Q$ *marker states*

$\delta : Q \times \Sigma \to Q$ *(partial) transition function*

We are also given a subset $\Sigma_c \subseteq \Sigma$, denoting the set of controlled events and $\Sigma_u = \Sigma \backslash \Sigma_c$, denoting the set of uncontrolled events.

A set of control patterns is $\Gamma = \{\gamma : (\gamma : \Sigma \to \{\mathbf{true}, \mathbf{false}\}) \ \wedge \ \gamma(\Sigma_u) : \gamma\}$ and we say $\sigma \in \Sigma$ is enabled if $\gamma(\sigma)$ and is disabled if $\neg\gamma(\sigma)$.

A controlled sequential process is a tuple $\mathcal{G} = (G, \Gamma)$.

---

[3]In fact, we are inspired by the work of Ramadge and Wonham and have adapted their problem formulation in this thesis.

[4]The following definitions do not appear in the glossary, because they are needed in this section only. Moreover, they are slightly modified to become more readable.
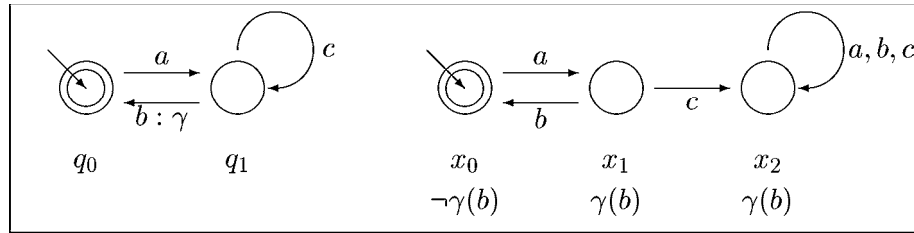
Figuur 5.3: Sequential process G (left) and supervisor S (right).

An extended control grammar *is* $G_c = (Q, \Gamma \times \Sigma, \delta_c, q_0, Q_m)$ *with*[5]

$$\begin{aligned}
\delta_c(q, (\gamma, \sigma)) &= \delta(q, \sigma) \quad \textit{if} \quad \gamma(\sigma) \\
&= \perp \qquad\quad \textit{if} \quad \neg\gamma(\sigma)
\end{aligned}$$

G is in fact a state graph representation of a trace structure with trace set equal to the set of all paths in G starting in q_0 and ending in a marker state. The only difference with state graphs is that δ need not be complete in G. It is straightforward, however, to create a complete graph out of G: the symbol $\perp$ represents the error state if we add:

$$\delta(\perp, \sigma) = \perp \qquad \delta_c(\perp, \gamma, \sigma) = \perp$$

G_c represents the behaviour of G under the control pattern Γ (i.e., all disabled events are removed from the graph G to get the graph G_c). G_c is in fact again a state graph where transitions labeled with events that are disabled are removed (i.e., replaced by a transition to the error state).

Definition 5.18 *A supervisor is a tuple* $\mathcal{S} = (S, \phi)$ *with*

$$\begin{aligned}
S &= (X, \Sigma, \xi, x_0, X_m) \quad \textit{another sequential process} \\
\phi &: X \to \Gamma \qquad\qquad\quad \textit{the state feedback map}
\end{aligned}$$

The closed loop supervised process *can then be defined as*

$$\mathcal{S}|\mathcal{G} = \mathbf{ac}(X \times Q, \Sigma, \psi, (x_0, q_0), X_m \times Q_m)$$

with

$$\begin{aligned}
\psi(x, q, \sigma) &= (\xi(x, \sigma), \delta(q, \sigma)) \qquad \textit{if} \quad \phi(x)(\sigma) \wedge \xi(x, \sigma) \neq \perp \wedge \delta(q, \sigma) \neq \perp \\
&= \perp \qquad\qquad\qquad\qquad\quad \textit{otherwise}
\end{aligned}$$

and **ac** *meaning: the reachable part of the sequential process (graph) only.*

$\mathcal{S}|\mathcal{G}$ is the cartesian product of $\mathcal{S}$ and $\mathcal{G}$ with properly defined transition function and consisting of the accessible (reachable) part only. In fact, $\mathcal{S}|\mathcal{G}$ is the weave of the graphs of S and G, where enabling and disabling of events in G depends on the corresponding state of S.

Example 5.19 Consider the sequential process G and the supervisor S as given in figure 5.3. We have

$$G = (\{q_0, q_1\}, \{a, b, c\}, \delta, q_0, \{q_0\})$$

---

[5] $\perp$ stands for undefined.
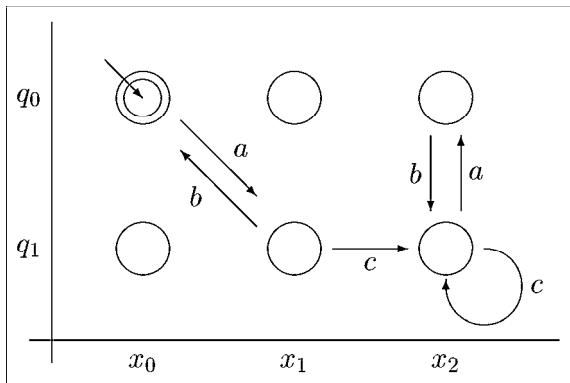
Figuur 5.4: Corresponding closed loop supervised process for figure 5.3

with $\delta(q_0, a) = q_1$, $\delta(q_1, b) = q_0$, $\delta(q_1, c) = q_1$, and all other δ's undefined. Moreover, $\Sigma_c = \{b\}$, denoted by $b : \gamma$ in the graph. The supervisor is given by

$$S = (\{x_0, x_1, x_2\}, \{a, b, c\}, \xi, x_o, \{x_0\})$$

with ξ according to the graph and state feedback map given by $\phi(x_0) = \{\neg\gamma(b)\}$, $\phi(x_1) = \{\gamma(b)\}$, and $\phi(x_2) = \{\gamma(b)\}$, i.e., b is disabled if the supervisor is in state x_0 and enabled if it is in state x_1 or x_2.

According to the above definitions, we find the closed loop supervised process $\mathcal{S}|\mathcal{G}$ as in figure 5.4. For example $\psi(x_1, q_1, b) = (\xi(x_1, b), \delta(q_1, b)) = (x_0, q_0)$, because in x_1 we have $\gamma(b)$. If we had $\neg\gamma(b)$ in x_1, we should have $\psi(x_1, q_1, b) = \bot$.

If we choose $\neg\gamma(b)$ in x_2 of S, we disable b although we could observe it. S is overdone here: we are able to observe an event that is at that moment disabled. If we choose not to have the occurrence of b in state x_2 of S but leave $\gamma(b)$ there, we enable b but are unable to observe it. S is not complete: it cannot observe at every point every event that can occur. If S is not overdone nor incomplete, we call S proper. We only investigate proper supervisors in this section.

(end of example)

These definitions have a lot in common with our definitions of a discrete process. A sequential process is defined here as a (generator) state graph with transitions that may be blocked (i.e., disabled). Each path through the graph gives a trace of the process. If such a path ends in a marker state, the corresponding trace represents a completed task. Marker states can therefore be identified with final states in the state graph corresponding with a discrete process.

A supervisor is an (observer) state graph with all events enabled. The task of the supervisor is to follow the behaviour of the sequential process and compute (after each occurrence of an event) a new control pattern so as to enable or disable future events. However, the behaviour of plant (sequential process) and controller (supervisor) are not the same. The plant is a passive generator (it generates events, but is unable to control) while the supervisor is an active observer (it cannot generate events on its own, it can only enable and disable events). Enabling an event does not mean that this event actually occurs; it is possible that another event that is also enabled actually occurs.

It turns out that this kind of control is in fact a passive kind of control: a process is not forced to do anything, only certain actions may temporarily be disabled. In

CODE an active kind of control is developed (which becomes more clear when inputs and outputs are introduced, as will be seen in later chapters).

The different interpretation of plant and supervisor in supervisory control makes it hard to give a more general definition of connection of processes for example to connect more than two processes or to supervise a supervisor. Indeed, nothing is said about the behaviour of a number of processes if they can influence each other and the observational behaviour of the supervisor cannot be influenced either.

Last, but not least, a sequential process with state graphs seems suitable but is not. As long as the graphs are finite (i.e., as long as the corresponding behaviour is regular), it is always possible to draw such a graph, but how can one draw an infinite graph? In CODE the behaviour is given in language-like terms directly and a solution is given that is independent of the properties of that behaviour. The only restriction is that one should be able to compute the necessary blends and weaves.

In supervisory theory a number of languages is defined also:

Definition 5.20 _In addition to a sequential process G and a supervisor S, we define:_

$$
\begin{aligned}
L_m(\mathcal{G}) &= \{s : s \in \Sigma^* \wedge \delta(q_0, s) \in Q_m : s\} \\
L(\mathcal{S}|\mathcal{G}) &= \{s : s \in \Sigma^* \wedge \psi(x_0, q_0, s) \neq \perp : s\} \\
L_c(\mathcal{S}|\mathcal{G}) &= L(\mathcal{S}|\mathcal{G}) \cap L_m(\mathcal{G}) \\
L_m(\mathcal{S}|\mathcal{G}) &= \{s : s \in \Sigma^* \wedge \psi(x_0, q_0, s) \in X_m \times Q_m\}
\end{aligned}
$$

Example 5.21 In example 5.19 we have:

$$
\begin{aligned}
L_m(\mathcal{G}) &= (ac^*b)^* \\
L(\mathcal{S}|\mathcal{G}) &= (ac^*b)^* \\
L_c(\mathcal{S}|\mathcal{G}) &= (ac^*b)^* \\
L_m(\mathcal{S}|\mathcal{G}) &= (ab)^*
\end{aligned}
$$

(end of example)

The main control problem in supervisory control is formulated as: given a sequential process G, minimal acceptable behaviour L_a, and a legal behaviour L_g, find a supervisor S such that:

$$
L_a \subseteq L_c(\mathcal{S}|\mathcal{G}) \subseteq L_g
$$

called the Supervisory control problem (SCP), or

$$
L_a \subseteq L_m(\mathcal{S}|\mathcal{G}) \subseteq L_g
$$

called the Supervisory marker problem (SMP).

For SCP we need the enabling and disabling of events, for SMP it is enough to find a supervisor that (in every state) enables every event, that can be accepted (i.e., the supervisors should be complete with respect to G_c). This last problem is very similar to ECODE.

For proper supervisors we can in fact identify $L_m(\mathcal{S}|\mathcal{G})$ with $P \underline{\mathbf{s}} R$, where P is the discrete process corresponding to $\mathcal{G}$ and R is the controller corresponding to $\mathcal{S}$, i.e.,

$$
\begin{aligned}
P &= \langle L_m(G), \emptyset, \Sigma \rangle \\
R &= \langle L_m(S), \emptyset, \Sigma \rangle
\end{aligned}
$$

which gives: $\mathbf{t}(P \underline{\mathbf{s}} R) = L_m(\mathcal{S}|\mathcal{G})$ so that we can rewrite SMP as $L_a \subseteq P \underline{\mathbf{s}} R \subseteq L_g$ (ECODE).

The other way round, a ECODE-problem can be written as a SMP only if the exogenous alphabet of the process to be controlled is empty. If $\mathbf{e}P = \emptyset$ we can rewrite ECODE in SMP:

$$L_a = L_{min}$$
$$L_g = L_{max}$$
$$L_m(\mathcal{S}|\mathcal{G}) = \mathbf{t}(P \; \underline{\mathbf{s}} \; R)$$

However, if $\mathbf{e}P \neq \emptyset$, we have that $\mathbf{a}R = \mathbf{c}P$, so $\mathbf{a}R \neq \mathbf{a}P$. We control P with only part of its events observable (and thus controllable) by R. In supervisory control this phenomenon (partial control) is treated by using masks, i.e., every event in P is mapped onto zero or one event of R (see [CDFV]).

It turns out that the theory presented in this thesis has some overlap with supervisory control: SMP is equal to ECODE (with $\mathbf{e}P = \emptyset$). In general, however, supervisory control uses another approach of controlling events (by enabling and disabling). In CODE a number of events control the other events. CODE seems more general because it allows partial control as well as marker control without using any extra mathematical equipment.

6
Deadlock

In this chapter we discuss the possibility of deadlock: the situation that a number of processes in a connection cannot continue while no task is completed. An algorithm is presented that determines whether deadlock may occur.

Deadlock is defined on the joint behaviour of discrete processes. Because it is of no importance here whether an event is an exogenous event or a communication event we restrict our attention to trace structures and use the results on discrete processes.

6.1   Definition of deadlock

To define the notion of deadlock we need the following definition:

Definition 6.1   *The set of ending traces of P is denoted by* **end**(P) *and defined by*

$$\{x : x \in \mathbf{t}P \ \wedge \ (\forall a : a \in \mathbf{a}P : xa \notin \mathbf{pref}(\mathbf{t}P)) : x\}$$

The ending traces in a connection of P and R are denoted by **end**(P, R) *and defined by*

$$\{x : x\lceil \mathbf{a}P \in \mathbf{end}(P) \ \wedge \ x\lceil \mathbf{a}R \in \mathbf{end}(R) : x\}$$

65

Property 6.2

(1) $\mathbf{end}(P, R) = \mathbf{end}(R, P)$
(2) $\mathbf{end}(P) \subseteq \mathbf{t}P$
(3) $\mathbf{end}(P) = \{x : x \in \mathbf{t}P \ \wedge \ (\forall y : y \in (\mathbf{a}P)^* : xy \notin \mathbf{t}P) : x\}$
(4) $\mathbf{end}(P, R) \subseteq \mathbf{end}(P \ \underline{\mathbf{w}} \ R)$

proof: Parts (1), (2), and (3) are trivial. For part (4) notice that (property 1.18 in [JvdS]): $\mathbf{pref}(P \ \mathbf{w} \ R) \subseteq \mathbf{pref}(P) \ \mathbf{w} \ \mathbf{pref}(R)$, so:

$$x \in \mathbf{end}(P, R)$$
$\Leftrightarrow \quad$ [definition of **end**]
$$x\lceil\mathbf{a}P \in \mathbf{t}P \ \wedge \ x\lceil\mathbf{a}R \in \mathbf{t}R \ \wedge \ (\forall a : a \in \mathbf{a}P : xa\lceil\mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P)) \ \wedge$$
$$(\forall a : a \in \mathbf{a}R : xa\lceil\mathbf{a}R \notin \mathbf{pref}(\mathbf{t}R))$$
$\Rightarrow \quad$ [definition of $\underline{\mathbf{w}}$ and above inclusion]
$$x \in \mathbf{t}(P \ \underline{\mathbf{w}} \ R) \ \wedge \ (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R : xa \notin \mathbf{pref}(\mathbf{t}(P \ \underline{\mathbf{w}} \ R)))$$
$\Leftrightarrow$
$$x \in \mathbf{end}(P \ \underline{\mathbf{w}} \ R)$$

(end of proof)

Example 6.3 Consider $P = \langle(a), \{a\}\rangle$ and $R = \langle(a|aa), \{a\}\rangle$, then

$$\mathbf{end}(P) = \{a\} \quad \mathbf{end}(R) = \{aa\} \quad \mathbf{end}(P, R) = \emptyset \quad \mathbf{end}(P \ \underline{\mathbf{w}} \ R) = \{a\}$$

So in part (4) of property 6.2 equality does not hold.
(end of example)

First, we define the possibility of deadlock in one connection, i.e., between two processes. Further on, we define deadlock in multi-connections.

Consider two discrete processes P and R. If it is possible to reach a state that is not an endpoint from which no further events are possible, we speak of deadlock.[1] Formally:

Definition 6.4 _Two discrete processes_ P _and_ R _may (in connection) end in_ deadlock, _notation_ **deadlock**(P, R), _if:_

$$(\exists x : x \notin \mathbf{end}(P, R) \ \wedge \ x\lceil\mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ x\lceil\mathbf{a}R \in \mathbf{pref}(\mathbf{t}R)$$
$$: (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R$$
$$: xa\lceil\mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P) \ \vee \ xa\lceil\mathbf{a}R \notin \mathbf{pref}(\mathbf{t}R)))$$

Property 6.5

$$\mathbf{deadlock}(P, R) \Leftrightarrow \mathbf{deadlock}(R, P)$$

The traces $x \in \mathbf{end}(P, R)$ are omitted because completed tasks can never deadlock (but only finish) the connection.

Our definition of a discrete process makes it impossible for one discrete process to be in deadlock by itself (i.e., $\neg\mathbf{deadlock}(P, P)$): this should be considered as termination of the process. We only consider $\mathbf{t}P$ as behaviour, i.e., we only consider completed traces. The following lemma illustrates this:

---
[1]When we have defined the notions input and output we will encounter another possibility of deadlock, called weak deadlock. The kind of deadlock defined here will then be called strong deadlock.

Lemma 6.6

$$x \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ x \notin \mathbf{end}(P)$$
$$\Rightarrow$$
$$(\exists a : a \in \mathbf{a}P : xa \in \mathbf{pref}(\mathbf{t}P))$$

proof:

$$x \notin \mathbf{end}(P) \ \wedge \ x \in \mathbf{pref}(\mathbf{t}P)$$
$$\Leftrightarrow \quad [\text{ definition of } \mathbf{end} \ ]$$
$$(x \notin \mathbf{t}P \ \vee \ (\exists a : a \in \mathbf{a}P : xa \in \mathbf{pref}(\mathbf{t}P)) \ \wedge \ (x \in \mathbf{pref}(\mathbf{t}P))$$
$$\Leftrightarrow$$
$$(x \notin \mathbf{t}P \ \wedge \ x \in \mathbf{pref}(\mathbf{t}P)) \ \vee$$
$$(x \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ (\exists a : a \in \mathbf{a}P : xa \in \mathbf{pref}(\mathbf{t}P))$$
$$\Rightarrow$$
$$(\exists a : a \in \mathbf{a}P : xa \in \mathbf{pref}(\mathbf{t}P))$$

(end of proof)

Some examples to illustrate deadlock are given below.

Example 6.7 Consider

$$P = \langle (dcb)^*, \{b, c, d\} \rangle \qquad R = \langle (ecc)^*, \{b, c, e\} \rangle$$

Take $x = edc$, then $x\lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P)$ and $x\lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R)$, but neither one of xb, xc, xd, or xe belongs (restricted to the corresponding alphabet) to both $\mathbf{pref}(\mathbf{t}P)$ and $\mathbf{pref}(\mathbf{t}R)$. So after x no common event is possible.
(end of example)

Example 6.8 Consider

$$P = \langle (ad|b), \{a, b, d\} \rangle \qquad R = \langle ((ad)^*|b), \{a, b, d\} \rangle$$

then $x = ad$ has the properties:

$$x \in \mathbf{end}(P) \ \wedge \ x \notin \mathbf{end}(R) \ \wedge \ x \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ x \in \mathbf{pref}(\mathbf{t}R)$$

and neither one of ada, adb, and add belongs to both $\mathbf{pref}(\mathbf{t}P)$ and $\mathbf{pref}(\mathbf{t}R)$. So we have deadlock. Here, we have the situation that P completes its task after ad and cannot continue, while R may continue.
(end of example)

6.2 Detecting deadlock

In this section we give (in case P and R are regular) a method to detect deadlock. Therefore we need the corresponding finite state graphs for P and R and construct a product automaton $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ in which deadlock can be detected. First, the construction of $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ is given in the next definition.

Definition 6.9 *Given are two completely defined and deterministic state graphs:*

$$M_P = (A_P, Q_P, \delta_P, q_P, F_P)$$
$$M_R = (A_R, Q_R, \delta_R, q_R, F_R)$$

then we define the state graph $M_d = (A_P \cup A_R, Q_P \times Q_R, \delta_d, q_d, F_P \times F_R)$ *with* $q_d = (q_P, q_R)$ *and* δ_d *defined by:*

$$a \in A_P \wedge a \notin A_R : \quad \delta_d((p_1, p_2), a) = (\delta_P(p_1, a), p_2)$$
$$a \notin A_P \wedge a \in A_R : \quad \delta_d((p_1, p_2), a) = (p_1, \delta_P(p_2, a))$$
$$a \in A_P \cap A_R : \qquad \delta_d((p_1, p_2), a) = ((\delta_P(p_1, a), \delta_R(p_2, a))$$

where we define $(p_1, |\varnothing|_R)$ *and* $(|\varnothing|_P, p_2)$ *to be equal to* $[\varnothing]_d$ *for all* p_1 *and* p_2.

Next, we define the deadlock recognizer $\mathbf{dr}(M_P, M_R)$ *as equal to the state graph* M_d *but with all unreachable states (and corresponding transitions) deleted:*

$$\mathbf{dr}(M_P, M_R)$$
$$=$$
$$(A_P \cup A_R, (Q_P \times Q_R) \setminus Q_u, \delta_d|_{(Q_P \times Q_R) \setminus Q_u}, q_d, (F_P \times F_R) \setminus Q_u)$$

with

$$Q_u = \{p_1, p_2 : \neg(\exists x :: \delta_P(q_P, x \lceil \mathbf{a}P) = p_1 \wedge \delta_R(q_R, x \lceil \mathbf{a}R) = p_2) : (p_1, p_2)\}$$

(the set of unreachable states).

Property 6.10

$$\mathbf{ts}(M_P \ \mathbf{w} \ M_R) = \mathbf{ts}(\mathbf{dr}(M_P, M_R))$$

proof: It is obvious from the definition of the deadlock recognizer that it accepts precisely the same language as the weave of the state graphs as defined in definition 4.5. (end of proof)

To be able to use $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ for detecting deadlock, we need the following definitions as well:

Definition 6.11 *A state* p *of* $\mathbf{sg}(P)$ *is called an* end state *if:*

$$p \in \mathbf{F}(P) \wedge (\forall a : a \in \mathbf{a}P : \delta(p, a) = [\varnothing]_d)$$

The set of all end states of $\mathbf{sg}(P)$ *is denoted by* E_P.
A state (p_1, p_2) *of* $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ *is called a* deadlock state *if:*

$$(p_1, p_2) \neq [\varnothing]_d \wedge (p_1 \notin E_P \vee p_2 \notin E_R) \wedge$$
$$(\forall a : a \in A_P \cup A_R : \delta_d((p_1, p_2), a) = [\varnothing]_d)$$

Property 6.12

(1) $x \in \mathbf{end}(P) \qquad \Leftrightarrow \quad [x]_P \in E_P$
(2) $x \in \mathbf{end}(P, R) \quad \Leftrightarrow \quad [x \lceil \mathbf{a}P]_P \in E_P \wedge [x \lceil \mathbf{a}R]_R \in E_R$

A deadlock ending state in $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ can be recognized as a state with no outgoing edges (except for those leading to the error state) and

- not being a final state or
- being a final state (p_1, p_2) with the property that from p_1 in $sg(P)$ or from p_2 in $\mathbf{sg}(R)$ at least one outgoing edge does not lead to the error state.

The last possibility is difficult to recognize. Further on we give a method in which this situation cannot arise.

We derive the next theorem:

Theorem 6.13

> **deadlock**(P, R)
> $\Leftrightarrow$
> **dr**$(\mathbf{sg}(P), \mathbf{sg}(R))$ *has at least one deadlock state*

proof:

> (p_1, p_2) is a deadlock state of $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$
> $\Leftrightarrow$ [by definition]
> $(p_1, p_2) \neq [\varnothing]_d \ \wedge \ (p_1 \notin E_P \ \vee \ p_2 \notin E_R) \ \wedge$
> $(\forall a : a \in \mathbf{a}P \cup \mathbf{a}R : \delta((p_1, p_2), a) = [\varnothing]_d)$
> $\Leftrightarrow$ [$((p_1, p_2) = [\varnothing]_d) \Leftrightarrow (p = [\varnothing]_P \ \vee \ q = [\varnothing]_R)$]
> $(p_1, p_2) \neq [\varnothing]_d \ \wedge \ (p_1 \notin E_P \ \vee \ p_2 \notin E_R) \ \wedge$
> $(\forall a : a \in \mathbf{a}P \cup \mathbf{a}R$
> $\quad : (a \in \mathbf{a}P \ \wedge \ a \notin \mathbf{a}R \ \Rightarrow \ \delta_P(p_1, a) = [\varnothing]_P) \ \wedge$
> $\quad\ \ (a \notin \mathbf{a}P \ \wedge \ a \in \mathbf{a}R \ \Rightarrow \ \delta_R(p_2, a) = [\varnothing]_R) \ \wedge$
> $\quad\ \ (a \in \mathbf{a}P \cap \mathbf{a}R \ \Rightarrow \ \delta_P(p_1, a) = [\varnothing]_P \ \vee \ \delta_R(p_2, a) = [\varnothing]_R))$
> $\Leftrightarrow$ [all states in $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ are reachable]
> $(\exists x : x \lceil \mathbf{a}P \in p_1 \ \wedge \ x \lceil \mathbf{a}R \in p_2$
> $\quad : (p_1, p_2) \neq [\varnothing]_d \ \wedge \ (p_1 \notin E_P \ \vee \ p_2 \notin E_R) \ \wedge$
> $\quad (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R$
> $\qquad : (a \in \mathbf{a}P \ \wedge \ a \notin \mathbf{a}R \ \Rightarrow \ \delta_P(p_1, a) = [\varnothing]_P) \ \wedge$
> $\qquad\ \ (a \notin \mathbf{a}P \ \wedge \ a \in \mathbf{a}R \ \Rightarrow \ \delta_R(p_2, a) = [\varnothing]_R) \ \wedge$
> $\qquad\ \ (a \in \mathbf{a}P \cap \mathbf{a}R \ \Rightarrow \ \delta_P(p_1, a) = [\varnothing]_P \ \vee \ \delta_R(p_2, a) = [\varnothing]_R))$
> $\Leftrightarrow$ [see property 6.12 and note below]
> $(\exists x : x \lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ x \lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R) \ \wedge \ x \notin \mathbf{end}(P, R)$
> $\quad : (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R : xa \lceil \mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P) \ \wedge \ xa \lceil \mathbf{a}R \notin \mathbf{pref}(\mathbf{t}R)))$
> $\Leftrightarrow$
> **deadlock**(P, R)

Notice that, because $\mathbf{sg}(P)$ and $\mathbf{sg}(R)$ are minimal (by definition), we have:

> $(x \lceil \mathbf{a}P \in p \ \wedge \ p \neq [\varnothing]_P) \Leftrightarrow (x \lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P))$ and
> $(x \lceil \mathbf{a}P \in q \ \wedge \ q \neq [\varnothing]_R) \Leftrightarrow (x \lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R))$

(end of proof)

Example 6.14 Reconsider example 6.7. Then the constructed deadlock recognizer $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ is as in figure 6.1. State 4 is a deadlock state: no edge leaves from state 4 (except for edges to the error state which are not drawn here).
(end of example)

Example 6.15 The deadlock recognizer for P and R from example 6.8 is drawn in figure 6.2. State 3 is a deadlock state. Although it is a final state in the corresponding state in the graph of R an event is possible (namely event a) so state 3 is not an end state. State 4 is no deadlock state: it is an endstate.
(end of example)
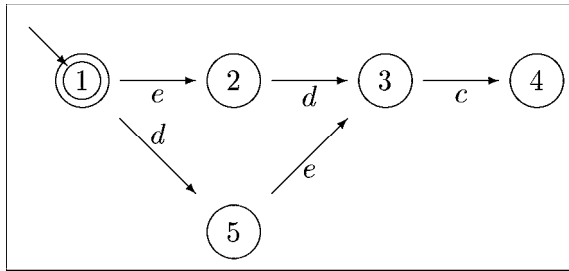
Figuur 6.1: Deadlock recognizer for example 6.7
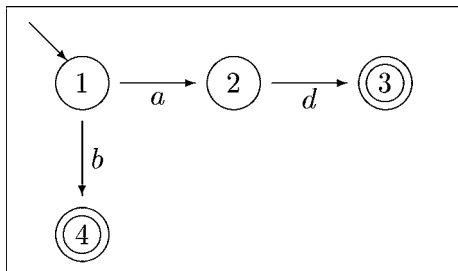
Figuur 6.2: Deadlock recognizer for example 6.8

6.3 Single task and repeating task processes

If we look at discrete processes more carefully, we can divide them into the following classes:

- single task processes: processes that perform exactly one completed task, after which they are unable to continue,
- repeating task processes: processes that are always able to continue after a completion of a task,
- mixed task processes: all other processes.

Notice that the set of all single task and repeating task processes is smaller than the set of all discrete processes. A discrete process can be neither a single task nor a repeating task process, in which case it is a mixed task process: sometimes it performs a completed task and is able to continue, another time it performs a completed task and cannot continue.

In terms of state graphs a single task process only contains final states from which all outgoing edges are leading to the error state. A repeating task process contains only final states from which at least one edge leaves that does not lead to the error state. A mixed task process contains both kinds of final states (or none at all).

Example 6.16 Consider

$$P = \langle (eb)^*c, \{b, c, e\} \rangle$$

This P is a single task process. Its trace set is infinite. Notice that

$$R = \langle (eb)^*, \{b, c, e\} \rangle$$

is a repeating task process and that

$$S = \langle((eb)^*c)|(eb)^*, \{b, c, e\}\rangle$$

is a mixed task process.
(end of example)

We can give the following formal definition of a repeating task process and of a single task process:

Definition 6.17 *A discrete process P is called a* single task process *if*

$$\mathbf{end}(P) = \mathbf{t}P$$

and a repeating task process *if*

$$\mathbf{end}(P) = \emptyset$$

Example 6.18 Reconsider P, R, and S from example 6.16, then:

$$\begin{array}{rcl}
\mathbf{end}(P) & = & ((eb)^*c) \\
\mathbf{end}(R) & = & \emptyset \\
\mathbf{end}(S) & = & ((eb)^*c)
\end{array}$$

(end of example)

In the rest of this chapter we only consider repeating task processes. For such processes the set of ending traces is empty, so the definition of deadlock reduces to a simpler form (i.e., we can omit $x \notin \mathbf{end}(P, R)$).

Property 6.19 *If P and R are repeating task processes, we have:*

$$\mathbf{deadlock}(P, R)$$
$$\Leftrightarrow$$
$$(\exists x : x \lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \wedge x \lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R)$$
$$: (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R$$
$$: xa \lceil \mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P) \vee xa \lceil \mathbf{a}R \notin \mathbf{pref}(\mathbf{t}R)))$$

If the process is not a repeating task process, we can make it a repeating task process by concatenating the single task part of the process with the so called *finishing process*

$$\mathbf{stop} = \langle \square^*, \{\square\} \rangle$$

The event $\square$ (called the *finishing event*) denotes that the process has finished, i.e., has completed a task and will not ever continue.

Definition 6.20 *For a discrete process P the associated repeating task process is denoted by $P_\square$ and defined by*

$$P_\square = (\langle \mathbf{end}(P), \mathbf{a}P \rangle \; ; \mathbf{stop}) \cup \langle \mathbf{t}P \setminus \mathbf{end}(P), \mathbf{a}P \rangle$$
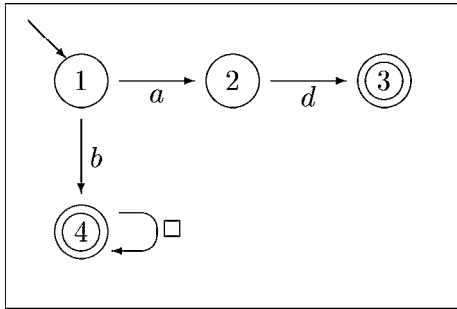
Figuur 6.3: Deadlock recognizer for example 6.8

Example 6.21 Reconsider the process P, R, and S from example 6.16. It is easily seen that $S = P \cup R$, where P is the single task part and R is the repeating task part of S. According to the above construction we find:

$$
\begin{aligned}
S_\square &= (P \; ; \mathbf{stop}) \cup R \\
&= \langle (eb)^* c\square^*, \{b, c, e, \square\} \rangle \cup \langle (eb)^*, \{b, c\}, \{e\} \rangle \\
&= \langle (eb)^* c\square^* | (eb)^*, \{b, c, e, \square\} \rangle
\end{aligned}
$$

(end of example)

Considering only repeating task processes makes it easier to detect deadlock using a deadlock recognizer. The situation of a final state in the recognizer without outgoing edges and still not being a deadlock state is omitted. If we consider only repeating task processes, all states without outgoing edges (being a final state or not) are deadlock states.

Example 6.22 Reconsider P and R from example 6.8. P and R are not repeating task processes, so we use $P_\square$ and $R_\square$ instead. So we have:

$$
P_\square = \langle (ad|b)\square^*, \{a, b, d\square\} \rangle \qquad R_\square = \langle (ad)^* | b\square^*, \{a, b, d, \square\} \rangle
$$

Computing $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ leads to the recognizer as in figure 6.3. We now immediately see that state 3 is a deadlock state and state 4 is not.
(end of example)

6.4 Deadlock-free controllers

Using the deCODEr, we may arrive at a controller such that in the connection deadlock is possible.

Example 6.23 Reconsider the ship lock example of figure 4.2. If we add to this process observer-events x_1 and x_2 with the meaning of *a ship has passed through door 1 (door 2 respectively)* as in figure 6.4 and apply the deCODEr again, we get the (at first unexpected) solution given in figure 6.5. Unfortunately we have deadlock. For example the control trace

$o_1 x_1 o_2 x_2 c_2 c_1$
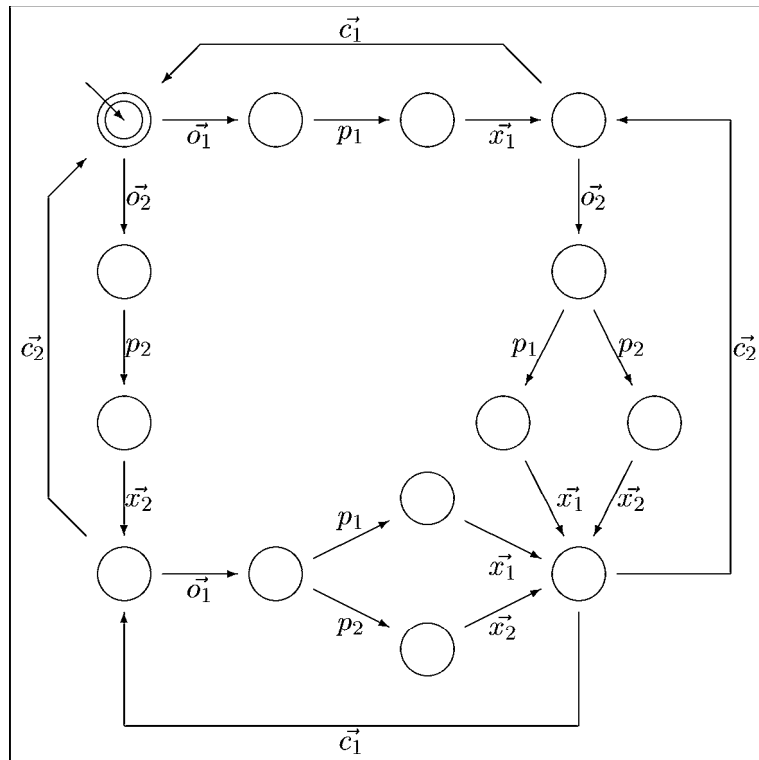
Figuur 6.4: The modified ship lock

may result in deadlock: if after $o_1 x_1 o_2$ P results in the occurrence of p_1 (instead of p_2), event x_1 occurs next, while R expects event x_2 only.

The resulting controller indeed leads to the desired behaviour L, due to the fact that the blend deletes all traces as described above: $P \underline{\mathbf{b}} R = L$ as desired.

(end of example)

The question is how to prevent this deadlock, i.e., how to construct an R such that $P \underline{\mathbf{b}} R = L$ (or in general: satisfying the minmax condition) but without having **deadlock**(P, R).

The problem of solving CODE with the added restriction that the connection of P with the constructed controller R does not deadlock is called *deadlock-free control of discrete events* (DFCODE for short).

We claim that the following solution works:

1) Use the deCODEr to solve the problem first, i.e., compute $F(L)$.

2) Check if **deadlock**$(P, F(L))$. If so, continue.

3) Delete from $F(L)$ all traces from which a prefix may lead to deadlock (this gives R).

4) If this newly constructed controller still satisfies the minmax condition, then R is a solution for DFCODE, otherwise if this method fails even for $F(L_{max})$, then no deadlock-free controller can be found.

Before we give a formal description of this algorithm (especially step 3) and prove its correctness, we consider more general deadlock-free connections.
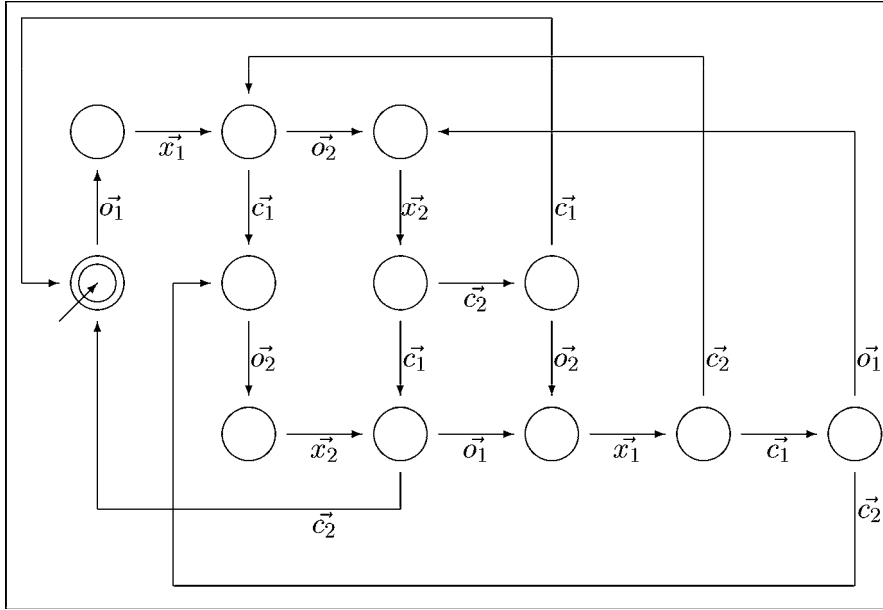
Figuur 6.5: The resulting controller

6.5 Deadlock-free connections

We claim that if the repeating task processes P and R have the possibility of deadlock, we can construct another process, say S, such that $\neg\textbf{deadlock}(P, R \textbf{ w } S)$. Therefore, we need the following notions:

Definition 6.24 *The* deadlock-ending trace set *of the connection of P and R is defined by*

$$d(P, R)$$
$$=$$
$$\{x : x \notin \textbf{end}(P, R) \ \wedge \ x\lceil\textbf{a}P \in \textbf{pref}(\textbf{t}P) \ \wedge \ x\lceil\textbf{a}R \in \textbf{pref}(\textbf{t}R) \ \wedge$$
$$(\forall a : a \in \textbf{a}P \cup \textbf{a}R : xa\lceil\textbf{a}P \notin \textbf{pref}(\textbf{t}P) \ \vee \ xa\lceil\textbf{a}R \notin \textbf{pref}(\textbf{t}R))$$
$$: x\}$$

The deadlock-ending process *of R with respect to P is defined by*

$$\textbf{dl}_P(R) \ = \ \langle d(P, R)\rangle\lceil\textbf{a}R, \textbf{e}R, \textbf{c}R\rangle$$

The extended process *of P is defined by*

$$\textbf{ext}(P) \ = \ \langle\{x : (\exists y : y \in \textbf{pref}(x) : y \in \textbf{t}P) : x\}, \textbf{e}P, \textbf{c}P\rangle$$

$d(P, R)$ denotes all deadlock ending traces of the connection of P and R. Therefore:

Property 6.25

$$\neg\textbf{deadlock}(P, R) \Leftrightarrow (d(P, R) = \varnothing)$$

$\textbf{dl}_P(R)$ denotes the discrete process containing all deadlock ending traces that are possible in the connection of R with P. We have:

Property 6.26

$$\mathbf{dl}_P(R) \subseteq \mathbf{pref}(R)$$

When there is no confusion, we write $\mathbf{dl}(R)$ for short.

The operator $\mathbf{ext}(P)$ defines a discrete process of which all traces have some prefix that is in P. It extends P by adding to all traces of P all finite sequences over the alphabet of P. The following property gives two alternative definitions of $\mathbf{ext}$:

Property 6.27

(1) $\mathbf{ext}(P) = \langle \{x : (\exists y, z : y \in \mathbf{t}P \ \wedge \ z \in (\mathbf{a}P)^* : x = yz) : x\}, \mathbf{e}P, \mathbf{c}P \rangle$
(2) $\mathbf{ext}(P) = P ; \langle (\mathbf{a}P)^*, \mathbf{e}P, \mathbf{c}P \rangle$

Our claim is that $R \setminus \mathbf{ext}(\mathbf{dl}_P(R))$ is the greatest process contained in R for which the connection with P is deadlock-free. We prove this result for repeating task processes only. The result can also be used on non-repeating task processes after performing the construction of adding finishing events as described in example 6.22.

Lemma 6.28 *For repeating task processes P and R, we have:*

$$\neg\mathbf{deadlock}(P, R \setminus \mathbf{ext}(\mathbf{dl}_P(R)))$$

proof: From

$\quad x \in \mathbf{t}(R \setminus \mathbf{ext}(\mathbf{dl}_P(R)))$
$\Leftrightarrow \quad$ [definitions of $\setminus$, $\mathbf{ext}$, $\mathbf{dl}_P(R)$, and $\lceil \ \rceil$]
$\quad x \in \mathbf{t}R \ \wedge \ (\forall y : y \in \mathbf{pref}(x) : (\forall z : z \in d(P, R) : z\lceil \mathbf{a}R \neq y))$
$\Leftrightarrow$
$\quad x \in \mathbf{t}R \ \wedge \ (\forall z : z \in d(P, R) : z\lceil \mathbf{a}R \notin \mathbf{pref}(x))$

we conclude that all traces from which a prefix may lead to deadlock with P have been removed.
(end of proof)

Lemma 6.29 *For repeating task processes P, R, and S, we have:*

$\quad \neg\mathbf{deadlock}(P, S) \ \wedge \ S \subseteq R$
$\Rightarrow$
$\quad S \subseteq R \setminus \mathbf{ext}(\mathbf{dl}_P(R))$

proof:

$\quad y \in d(P, R)$
$\Leftrightarrow \quad$ [P and R are repeating task processes so $\mathbf{end}(P, R) = \emptyset$]
$\quad y\lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ y\lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R) \ \wedge$
$\quad (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R : ya\lceil \mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P) \ \vee \ ya\lceil \mathbf{a}R \notin \mathbf{pref}(\mathbf{t}R))$
$\Rightarrow \quad$ [$S \subseteq R \Rightarrow \mathbf{pref}(S) \subseteq \mathbf{pref}(R)$]
$\quad y\lfloor \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ y\lfloor \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R) \ \wedge$
$\quad (\forall a : a \in \mathbf{a}P \cup \mathbf{a}R : ya\lceil \mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P) \ \vee \ ya\lceil \mathbf{a}R \notin \mathbf{pref}(\mathbf{t}S))$

Also, we have (from $\neg\textbf{deadlock}(P, S)$, see property 6.25) $d(P, S) = \varnothing$, so:

$$y \notin d(P, S)$$
$$\Leftrightarrow \quad [\ \textbf{end}(P, S) = \varnothing\ ]$$
$$y\lceil \textbf{a}P \notin \textbf{pref}(\textbf{t}P) \vee y\lceil \textbf{a}R \notin \textbf{pref}(tS) \vee$$
$$\neg(\forall a : a \in \textbf{a}P \cup \textbf{a}R : ya\lceil \textbf{a}P \notin \textbf{pref}(\textbf{t}P) \vee ya\lceil \textbf{a}R \notin \textbf{pref}(\textbf{t}S))$$

Together this leads to:

$$y \in d(P, R)$$
$$\Leftrightarrow$$
$$y \in d(P, R) \wedge y \notin d(P, S)$$
$$\Rightarrow$$
$$y\lceil \textbf{a}P \in \textbf{pref}(\textbf{t}P) \wedge y\lceil \textbf{a}R \in \textbf{pref}(\textbf{t}R) \wedge y\lceil \textbf{a}R \notin \textbf{pref}(\textbf{t}S)) \wedge$$
$$(\forall a : a \in \textbf{a}P \cup \textbf{a}R : ya\lceil \textbf{a}P \notin \textbf{pref}(\textbf{t}P) \vee ya\lceil \textbf{a}R \notin \textbf{pref}(\textbf{t}R))$$
$$\Rightarrow$$
$$y\lceil \textbf{a}R \notin \textbf{pref}(\textbf{t}S) \wedge y\lceil \textbf{a}R \in \textbf{pref}(\textbf{t}R)$$

Hence:

$$x \in \textbf{t}(\textbf{dl}_P(R))$$
$$\Leftrightarrow$$
$$(\exists y : y \in d(P, R) : y\lceil \textbf{a}R = x)$$
$$\Rightarrow \quad [\ \text{see above}\ ]$$
$$(\exists y : y\lceil \textbf{a}R \notin \textbf{pref}(\textbf{t}S) : y\lceil \textbf{a}R = x)$$
$$\Leftrightarrow$$
$$x \notin \textbf{pref}(\textbf{t}S)$$

This leads to:

$$y \in \textbf{t}(\textbf{ext}(\textbf{dl}_P(R)))$$
$$\Leftrightarrow \quad [\ \text{definition of } \textbf{ext}\ ]$$
$$(\exists x : x \in \textbf{pref}(y) : x \in \textbf{t}(\textbf{dl}_P(R)))$$
$$\Rightarrow \quad [\ x \in \textbf{t}(\textbf{dl}_P(R)) \Rightarrow x \notin \textbf{pref}(\textbf{t}S)\ ]$$
$$(\exists x : x \in \textbf{pref}(y) : x \notin \textbf{pref}(\textbf{t}S))$$
$$\Rightarrow$$
$$y \notin \textbf{t}S$$

So: $y \in \textbf{t}S \Rightarrow y \notin \textbf{t}(\textbf{ext}(\textbf{dl}_P(R)))$
From $S \subseteq R$, we also have $y \in \textbf{t}S \Rightarrow y \in \textbf{t}R$, which leads to:

$$y \in \textbf{t}S$$
$$\Rightarrow$$
$$y \in \textbf{t}R \wedge y \notin \textbf{t}(\textbf{ext}(\textbf{dl}_P(R)))$$
$$\Rightarrow$$
$$y \in \textbf{t}(R \setminus \textbf{ext}(\textbf{dl}_P(R)))$$

(end of proof)

This lemma implies that for repeating task processes P and R:

1) If $\textbf{deadlock}(P, R)$, we can construct a process $R' = R \setminus \textbf{ext}(\textbf{dl}_P(R))$ such that $\neg\textbf{deadlock}(P, R')$.

2) This R' is the largest $R' \subseteq R$ with $\neg\textbf{deadlock}(P, R')$.

Theorem 6.30

$$(\forall P, R : \mathbf{end}(P) = \emptyset \ \wedge \ \mathbf{end}(R) = \emptyset : (\exists S :: \neg\mathbf{deadlock}(P, R \ \mathbf{w} \ S)))$$

proof: According to lemma 6.29, choose

$$S = R \setminus \mathbf{ext}(\mathbf{dl}(R))$$

Then we have:

$$R \ \mathbf{w} \ S$$
$$=$$
$$R \ \mathbf{w} \ (R \setminus \mathbf{ext}(\mathbf{dl}(R)))$$
$$= \quad [\ \text{property 1.20 (1)} \ ]$$
$$R \cap (R \setminus \mathbf{ext}(\mathbf{dl}_P(R)))$$
$$= \quad [\ R \setminus \mathbf{ext}(\mathbf{dl}_P(R)) \subseteq R \ ]$$
$$R \setminus \mathbf{ext}(\mathbf{dl}(R))$$

So $\mathbf{deadlock}(P, R \ \mathbf{w} \ S) \Leftrightarrow \mathbf{deadlock}(P, R \setminus (\mathbf{dl}_P(R)))$ and the theorem follows using lemma 6.28.
(end of proof)

6.6   Solving DFCODE

For solving DFCODE, we need the following additional processes:

Definition 6.31   _Associated with DFCODE, we define:_

$$D(P, L) = \mathbf{ext}(\mathbf{dl}_P(F(P, L)))$$

called the deadlock-ending communication process _and_

$$C(P, L) = F(P, L) \setminus D(P, L)$$

called the clean friend _of L._

When it is clear from the context we write $D(L)$ and $C(L)$ for short. According to lemma 6.29 we have

Property 6.32   _For repeating task processes P, R, and L, we have:_

$$\neg\mathbf{deadlock}(P, R) \ \wedge \ R \subseteq F(L)$$
$$\Rightarrow$$
$$R \subseteq C(L)$$

which leads to:

Theorem 6.33

DFCODE is solvable for repeating task processes
$$\Leftrightarrow$$
$$L_{min} \subseteq P \ \underline{\mathbf{b}} \ C(L_{max})$$

proof: Property 6.32 states that $C(L)$ is the largest possible deadlock-free controller using L. From $C(L) \subseteq F(L)$ we know that $P \ \underline{\mathbf{b}} \ C(L) \subseteq L_{max}$, so $C(L)$ is a good

controller if $P \underline{\mathbf{b}} C(L) \supseteq L_{min}$, which is exactly as stated in the theorem. (end of proof)

6.6.1 Effect on state graphs

For CODE we have developed an algorithm on state graphs. We can do the same for DFCODE. The problem is how to compute $D(L)$. The following algorithm does it:

1) Construct the deadlock recognizer $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(F(L)))$.

2) Detect all deadlock states in this state graph.

3) Change all final states of the recognizer in non-final states and all deadlock states in final states.
   (The graph now represents all deadlock ending traces, i.e., it represents the discrete process $\langle d(P, F(L)), \mathbf{e}P, \mathbf{c}P \rangle$.)

4) Perform the construction on this graph to get the restriction to $\mathbf{c}P$, i.e., find the state graph corresponding to $\mathbf{dl}_P(F(P, L))$.

5) Change all final states in this graph in final states with all outgoing edges returning directly to that state.[2] In this way we have constructed a graph representation for $D(L)$.

6) It remains to compute $F(L) \setminus D(L)$, which can be done in the usual way.

7) Check if the solution found still satisfies the minmax condition. If so, a deadlock-free solution is found, otherwise, if the minmax condition is not even fulfilled with $L = L_{max}$, no deadlock-free solution exists.

The operator **ext** has in fact a very simple effect on state graphs, which is due to the simple form of $\mathbf{sg}(\mathbf{dl}_P(F(P, L)))$. In this state graph all final states are _end states_, i.e., all edges leaving these final states are leading to the error state. The operator has a much less trivial effect on general state graphs. The operator **ext** is needed in the above construction, because a deadlock situation occurs on prefixes, i.e., in order to get a deadlock-free controller, from R those traces have to be removed from which a _prefix_ may result in deadlock.

6.7 Deadlock in multi-connections

So far, we have only considered the possibility of deadlock in exactly one connection. In this section we investigate how deadlock can occur in multi-connections and how it can be detected. We define the possibility of deadlock in a multi-connection as the possibility that no process can continue while no task is completed:

Definition 6.34 _A number of processes_ $P_1, \ldots, P_n$ _have the possibility to end in dead-lock, notation_ **deadlock**$(P_1, \ldots, P_n)$, _if:_

---

[2]instead of leading to the error state. These final states are in fact the former deadlock states, i.e., all leaving edges first led to the error state.

$(\exists x : x \in (\bigcup i : 1 \le i \le n : \mathbf{a}P_i)^*$
$\quad : (\exists i : 1 \le i \le n : x\lceil \mathbf{a}P_i \notin \mathbf{end}(P_i)) \wedge$
$\quad (\forall i : 1 \le i \le n : x\lceil \mathbf{a}P_i \in \mathbf{pref}(\mathbf{t}P_i)) \wedge$
$\quad (\forall a : a \in (\bigcup i : 1 \le i \le n : \mathbf{a}P_i)$
$\quad\quad : (\exists i : 1 \le i \le n : xa\lceil \mathbf{a}P_i \notin \mathbf{pref}(\mathbf{t}P_i))))$

This definition is a precise generalisation from earlier definitions.

We do not define deadlock on more processes as the possibility that two of these processes may have deadlock as in done in [Kal]. If we did define it this way we would not be able to prevent deadlock by connecting an extra process.

Example 6.35 Consider

$P = \langle (a|bc|ca), \{a, b, c\} \rangle$
$R = \langle (aad), \{a, d\} \rangle$
$S = \langle (cd), \{b, c, d\} \rangle$

which gives:

$\neg\mathbf{deadlock}(P, S)$ $\quad\quad \mathbf{deadlock}(P, R)$ $\quad d(P, R) = \{abc\}$
$\neg\mathbf{deadlock}(P, R)$
$\neg\mathbf{deadlock}(P, R, S)$

which disproves $\mathbf{deadlock}(P, R) \Rightarrow \mathbf{deadlock}(P, R, S)$.
(end of example)

Example 6.36 Consider

$P \;=\; \langle (a(ec|db)), \{a, b, c, d, e\} \rangle$
$R \;=\; \langle (ac), \{a, b, c\} \rangle$
$S \;=\; \langle (a(c|d)), \{a, b, c\} \rangle$

which gives $\mathbf{t}(P \mathbf{\,w\,} R) = aec$ and $\mathbf{t}(R \mathbf{\,w\,} S) = ac$. We see:

$\neg\mathbf{deadlock}(P, S)$ $\quad\quad \mathbf{deadlock}(P, R)$ $\quad\quad d(P, R) = \{ad\}$
$\neg\mathbf{deadlock}(R, S)$ $\quad\quad \mathbf{deadlock}(P, R \mathbf{\,w\,} S)$ $\quad d(P, R \mathbf{\,w\,} S) = \{ad\}$
$\neg\mathbf{deadlock}(P \mathbf{\,w\,} R, S)$

which disproves: $\mathbf{deadlock}(P, R \mathbf{\,w\,} S) \Leftrightarrow \mathbf{deadlock}(P \mathbf{\,w\,} R, S)$.
(end of example)

6.7.1 Detecting deadlock in multi-connections

It is straightforward to see that

Property 6.37

$$\mathbf{dr}(M_P, \mathbf{dr}(M_R, M_S)) = \mathbf{dr}(\mathbf{dr}(M_P, M_R), M_S)$$

proof: The essential part of being equal is that both state graphs have the same transition function. If δ_P (δ_R and δ_S) is the transition function of M_P (M_R and M_S respectively) we have the symmetric form of the transition function δ of $\mathbf{dr}(M_P, (\mathbf{dr}(M_R.M_S))$ as given in table 6.1.

$a \in \dots$			$\delta((p_1, p_2, p_3), a) = \dots$
A_P	A_R	A_S	
$\in$	$\notin$	$\notin$	$(\delta_P(p_1, a), p_2, p_3)$
$\notin$	$\in$	$\notin$	$(p_1, \delta_R(p_2, a), p_3)$
$\notin$	$\notin$	$\in$	$(p_1, p_2, \delta_S(p_3, a))$
$\in$	$\in$	$\notin$	$(\delta_P(p_1, a), \delta_R(p_2, a).p_3)$
$\in$	$\notin$	$\in$	$(\delta_P(p_1, a), p_2, \delta_S(p_2, a))$
$\notin$	$\in$	$\in$	$(p_1, \delta_R(p_2, a), \delta_S(p_3, a))$
$\in$	$\in$	$\in$	$(\delta_P(p_1, a), \delta_R(p_2, a), \delta_S(p_3, a))$

With $(p_1, p_2, p_3) = [\varnothing]$

if $p_1 = [\varnothing] \lor p_2 = [\varnothing] \lor p_3 = [\varnothing]$

Tabel 6.1: transition function for $\mathbf{dr}(M_P, (\mathbf{dr}(M_r, M_S))$

Deleting unreachable states is of no essence to the transition function and the possible paths, nor of the graph having deadlock states.
(end of proof)

Because of this lemma we simply write $\mathbf{dr}(M_P, M_R, M_S)$, or more general

$$\mathbf{dr}(M_1, \dots, M_n)$$

Without proof we mention:

Lemma 6.38 *For repeating task processes $P_1, \dots, P_n$, we have:*

$$\mathbf{deadlock}(P_1, \dots, P_n)$$
$$\Leftrightarrow$$
$$\mathbf{dr}(\mathbf{sg}(P_1), \dots, \mathbf{sg}(P_n))) \text{ has at least one deadlock state}$$

where a deadlock state $(p_1, \dots, p_n)$ in $\mathbf{dr}(\mathbf{sg}(P_1), \dots, \mathbf{sg}(P_n))$ is defined by

$$(p_1, \dots, p_n) \neq [\varnothing] \land (\exists i : 1 \le i \le n : p_i \notin E_{P_i}) \land$$
$$(\forall a : a \in (\bigcup i : 1 \le i \le n : \mathbf{a}P_i) : \delta((p_1, \dots, p_n), a) = [\varnothing])$$

6.7.2 The dining philosophers

In this section we illustrate by means of an example[3] how deadlock can be prevented by adding an extra process to the connection.

Consider a number of philosophers (say k) sitting around a round table. Each of them in turn eats and thinks. In order to eat each philosopher needs two forks, one to the left and one to the right of his plate. Between each plate only one fork is present, so each fork has to be shared between two philosophers, but only one philosopher at the time can use it.

The philosophers can be modelled as follows:

$$P_i = < (s_i \, gl_i \, gr_i \, e_i \, ll_i \, lr_i \, t_i)^*, \{gl_i, gr_i, ll_i, lr_i, e_i, t_i\}, \{s_i\} >$$
$$i = 1, \dots, k$$

The interpretation of the events is given in table 6.2. To be able to model the sharing

---

[3]The example of the dining philosophers appears in literature many times. Originally it is from E.W. Dijkstra who first published the problem in [EWD1].

event	meaning
s_i	(get permission to) sit
gl_i	grab fork on the left
gr_i	grab fork on the right
e_i	eat
ll_i	lay down fork on the left
lr_i	lay down fork on the right
t_i	think

Tabel 6.2: Meaning of the events of P_i

of the forks we have:[4]

$$F_i =< ((gl_i \; ll_i)|(gr_{i \oplus 1} \; lr_{i \oplus 1}))^*, \{gl_i, ll_i, gr_{i \oplus 1}, lr_{i \oplus 1}\}, \emptyset >$$
$$i = 1, \ldots, k$$

Process F_i describes the behaviour of the left fork for philosopher P_i that is at the same time the right fork for philosopher $P_{i \oplus 1}$. The behaviour expresses that grabbing a fork should first be followed by laying down that fork by the same philosopher before it can be grabbed again.

The total behaviour we would like to investigate is:

$$T = (\underline{\mathbf{W}} \; i : 1 \leq i \leq k : P_i) \; \underline{\mathbf{s}} \; (\underline{\mathbf{W}} \; i : 1 \leq i \leq k : F_i)$$

Suppose that all philosophers have got permission to eat. Then the following sequence of events is possible (take $k = 3$):

$$s_3 \; s_1 \; s_2 \; gl_3 \; gl_1 \; gl_2$$

Now, all forks are in use, but no philosopher is able to eat. We have deadlock.

Because the events e_i and t_i are not important to us at this moment, we omit these from P_i. So we use:

$$P_i =< (s_i \; gl_i \; gr_i \; ll_i \; lr_i)^*, \{gl_i, gr_i, ll_i, lr_i\}, \{s_i\} >$$
$$i = 1, \ldots, k$$

and consider

$$T = P \; \underline{\mathbf{s}} \; F$$

with

$$P = (\underline{\mathbf{W}} \; i : 1 \leq i \leq k : P_i)$$
$$F = (\underline{\mathbf{W}} \; i : 1 \leq i \leq k : F_i)$$

It turns out that the generalised deadlock recognizer

$$M = \mathbf{dr}(\mathbf{sg}(P_1), \ldots, \mathbf{sg}(P_k), \mathbf{sg}(F_1), \ldots, \mathbf{sg}(F_k))$$

contains a deadlock state.

We can prevent the total connection from ending in deadlock using the deCODEr in the following way: from property 6.10 we know that M accepts the same language as $\mathbf{sg}(T)$, i.e., $\mathbf{ts}(M) = \mathbf{ts}(T)$. Now add to T all deadlock-ending traces by making the deadlock state in M a final state and use F as desired exogenous behaviour of T (notice

---

[4] $\oplus$ denotes addition modulo k here.

Figuur 6.6: First butler

that all events of F are exogenous events in T and only the events s_i are communication events). We have extended the behaviour of the philosophers with the deadlock-ending traces and use the deCODEr to be sure this behaviour cannot be reached (by omitting it in the desired behaviour).

The deCODEr gives us the controller (a *butler* in this case) (for $k = 3$ this butler is displayed in figure 6.6). The butler can only prevent the total process to end in deadlock by forbidding one (randomly chosen) philosopher ever to eat. The controller is not fair. This is not what we want. The resulting controller is unable to notice that a philosopher is done with eating and therefore the controller can not use the fact that this philosopher does not need the forks any more.

To be able to find a nicer controller, we add to P_i an extra event a_i, meaning that philosopher P_i asks the butler if he may sit down. The butler then may give him permission to sit down by letting event s_i occur. So we have:

$$P_i = < (a_i\ s_i\ gl_i\ gr_i\ ll_i\ lr_i)^*, \{gl_i, gr_i, ll_i, lr_i\}, \{a_i, s_i\} >$$
$$i = 1, \ldots, k$$

If we perform the same computation as before and use the deCODEr again, we find (for $k = 3$) the butler as in figure 6.7. This butler behaves precisely as we should think he should. He gives permission to at most two philosophers to start eating. A third demand is retained until one of the philosophers asks again (and thereby letting the butler know that he has finished eating).

This example shows that the deCODEr can also be used to prevent deadlock in those cases, where a number of processes are connected and communication events are left to control the whole. If the deCODEr fails no controller exists to make the connection deadlock-free.

Figuur 6.7: Second butler

<div align="right">

7

</div>

<div align="right">

Input and output

</div>

<div align="right">

Who needs information
When you're living in constant fear
Just give me confirmation
There's some way out of here

Who needs information – Radio Kaos (RW)

</div>

In this chapter input and output are defined. In the previous chapters we considered two kinds of events, the exogenous events (to be controlled) and the communicating events (the controls). We did not make any assumption of which process "generated" an event and which process "accepted" an event. We will do so in this chapter. We will distinguish between events that are generated by a process and events that are accepted by that process. Generated events will be called *outputs* and accepted events *inputs*. When we connect two processes with inputs and outputs we assume that events generated by one process are accepted by the other process. This means that inputs of one process are outputs of another (and vice versa).

7.1 Splitting up the alphabet

Assume that C_i, C_o, E_i, and E_o are pairwise disjoint finite set of events. Consider the trace structure

$$T = \langle S, C_i \cup C_o \cup E_i \cup E_o \rangle$$

Such a trace structure is called a *discrete process with inputs and outputs* (IODP for short) and denoted by $P = \langle S, E_i, E_o, C_i, C_o \rangle$, with S is the behaviour, E_i the exogenous input events, E_o the exogenous output events, C_i the communication input events, and C_o the communication output events.

Accepting an input event is only possible if that event has been generated by another process. Generating an output event is always possible (as far as the behaviour of the process allows it, of course). A process with inputs and outputs (IODP) therefore behaves differently than an ordinary discrete process (DP). This difference will become clear when we reconsider deadlock. It turns out that with IODPs more deadlock-cases are possible, for example when one process can generate an event that cannot be accepted by the other process. In ordinary DPs an event only occurs when it can occur in both processes simultaneously. The only possibility of deadlock then is the situation that the processes

<div align="center">

84

</div>

have, at a certain point, no event in common, but both are involved in a communication event.

In the sequel we will not make any distinction between exogenous input events (E_i) and exogenous output events (E_o) but only consider $E_i \cup E_o$. We will, however, distinguish between communication input events (C_i) and communication output events (C_o). As an abbreviation we call $E_i \cup E_o$ (say E) the exogenous events, C_i the inputs, and C_o the outputs. This leads to the following definition.

Definition 7.1 *A discrete process with inputs, outputs, and exogenous events (IOEDP) is defined by:*

$$P = \langle S, E, C_i, C_o \rangle$$

with S the trace set, E the exogenous alphabet, C_i the input alphabet, and C_o the output alphabet. Associated with such a P, we introduce:

$\mathbf{t}P$	$=$	S	*behaviour of the process*
$\mathbf{i}P$	$=$	C_i	*inputs*
$\mathbf{o}P$	$=$	C_o	*outputs*
$\mathbf{e}P$	$=$	E	*exogenous inputs and exogenous outputs*
$\mathbf{c}P$	$=$	$C_i \cup C_o$	*communication events*
$\mathbf{a}P$	$=$	$C_i \cup C_o \cup E$	*all events*

An IOEDP P is only well defined if:

$$C_i \cap C_o = C_i \cap E = C_o \cap E = \varnothing$$

We have defined the operators $\mathbf{a}$ and $\mathbf{t}$ on trace structures, discrete processes, and IOEDPs[1] now, and $\mathbf{c}$ on discrete processes as well as IOEDPs.

Furthermore, with an IOEDP P we sometimes associate a discrete process $\mathbf{dp}(P)$, defined by

$$\mathbf{dp}(P) = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{c}P \rangle$$

and sometimes even a simple trace structure $\mathbf{ts}(P)$, defined by

$$\mathbf{ts}(P) = \langle \mathbf{t}P, \mathbf{a}P \rangle$$

It should not be surprising that all properties for trace structures and discrete processes also hold for $\mathbf{ts}(P)$ and $\mathbf{dp}(P)$ respectively if P is an IOEDP.

Moreover, we extend all operators and functions defined in earlier chapters and use them on IOEDPs as well. Formally, if $\mathbf{op}$ is such an operator, we write $\mathbf{op}P$ instead of $\mathbf{op}(\mathbf{dp}(P))$ and if $\mathbf{fun}$ is such a function, we write $\mathbf{fun}(P)$ instead of $\mathbf{fun}(\mathbf{dp}(P))$.

7.2 Effect on state graphs

An IOEDP can also be displayed by a state graph:

$$\mathbf{sg}(P) = \mathbf{sg}(\mathbf{dp}(P))$$

---

[1]We use the abbreviation IOEDP instead of "discrete process with inputs, outputs and exogenous events."

In order to be able to distinguish between input and output events we display these events in a graph by postfixing them with ? and ! respectively. So in a state graph a denotes a exogenous event, a? an input event, and a! an output event.[2]

7.3 Connections

Definition 7.2 _Given two IOEDPs P and R with:_

$$\mathbf{e}P \cap \mathbf{a}R = \varnothing \quad \mathbf{e}R \cap \mathbf{a}P = \varnothing$$
$$\mathbf{i}P \cap \mathbf{i}R = \varnothing \quad \mathbf{o}P \cap \mathbf{o}R = \varnothing$$

then the connection of P and R is defined by:

$$P \mathbf{\underline{b}} R$$
$$=$$
$$\langle \mathbf{t}(P \mathbf{b} R)$$
$$, \mathbf{e}P \cup \mathbf{e}R$$
$$, (\mathbf{i}P \setminus \mathbf{o}R) \cap (\mathbf{i}R \setminus \mathbf{o}P)$$
$$, (\mathbf{o}P \setminus \mathbf{i}R) \cap (\mathbf{o}R \setminus \mathbf{i}P)$$
$$\rangle$$

Again, we have that $\mathbf{e}(P \mathbf{\underline{b}} R) = \mathbf{e}P \cup \mathbf{e}R$. Furthermore, we have:

$$\mathbf{c}(P \mathbf{\underline{b}} R)$$
$$=$$
$$((\mathbf{i}P \setminus \mathbf{o}R) \cap (\mathbf{i}R \setminus \mathbf{o}P)) \cup ((\mathbf{o}P \setminus \mathbf{i}R) \cap (\mathbf{o}R \setminus \mathbf{i}P))$$
$$=$$
$$\mathbf{c}P \div \mathbf{c}R$$

which guarantees that:

$$\mathbf{ts}(P) \mathbf{b} \mathbf{ts}(R) = \mathbf{ts}(P \mathbf{\underline{b}} R)$$

The connection is realized by "connecting" inputs of one process to outputs of the other process (just as one should expect). Notice that from the inputs of P and R in the connection only those events are left that are not used in the communication (and similar for the outputs). These remaining inputs and outputs can be used in other connections.

Again, we have the same property as in chapter 2 (see property 2.6):

Property 7.3 _For the connection $\mathbf{\underline{b}}$ of two IOEDPs, the following hold:_

(1) $P \mathbf{\underline{b}} R = R \mathbf{\underline{b}} P$
(2) $P \mathbf{\underline{b}} \langle \{\epsilon\}, \varnothing, \varnothing, \varnothing \rangle = P$
(3) $P \mathbf{\underline{b}} \langle \varnothing, \varnothing, \varnothing, \varnothing \rangle = \langle \varnothing, \varnothing, \varnothing, \varnothing \rangle$

The total connection is redefined as follows:

---

[2]However, events that do not appear in any trace of P cannot be denoted. They have to be mentioned explicitly.

Definition 7.4 *Given two IOEDPs P and R with:*

$\mathbf{e}P \cap \mathbf{a}R = \emptyset \quad \mathbf{e}R \cap \mathbf{a}P = \emptyset$
$\mathbf{i}P \cap \mathbf{i}R = \emptyset \quad \mathbf{o}P \cap \mathbf{o}R = \emptyset$

then the total connection of P and R is defined by:

$$= \frac{P \; \underline{\underline{\mathbf{w}}} \; R}{\begin{array}{l} \langle \mathbf{t}(P \; \mathbf{b} \; R) \\ , \mathbf{e}P \cup \mathbf{e}R \cup (\mathbf{i}P \div \mathbf{o}R) \cup (\mathbf{i}R \div \mathbf{o}P) \\ , (\mathbf{i}P \setminus \mathbf{o}R) \cap (\mathbf{i}R \setminus \mathbf{o}P) \\ , (\mathbf{o}P \setminus \mathbf{i}R) \cap (\mathbf{o}R \setminus \mathbf{i}P) \\ \rangle \end{array}}$$

Property 7.5 *For the total connection $\underline{\underline{\mathbf{w}}}$, the following properties hold:*

(1) $P \; \underline{\underline{\mathbf{w}}} \; R = R \; \underline{\underline{\mathbf{w}}} \; P$
(2) $P \; \underline{\underline{\mathbf{w}}} \; \langle \{\epsilon\}, \emptyset, \overline{\emptyset}, \emptyset \rangle = P$
(3) $P \; \underline{\underline{\mathbf{w}}} \; \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$

7.3.1 Multi-connections

More than two IOEDPs can be connected as well. We assume that no exogenous event appears in more than one of the alphabets, no input or output appears in more than two of the alphabets, and if such an event appears in two alphabets, it should be an input event in one of the processes and an output event in the other.

Under these restrictions we have:

Property 7.6

(1) $(P \; \underline{\mathbf{b}} \; R) \; \underline{\mathbf{b}} \; S \quad = \quad P \; \underline{\mathbf{b}} \; (R \; \underline{\mathbf{b}} \; S)$
(2) $(P \; \underline{\underline{\mathbf{w}}} \; R) \; \underline{\underline{\mathbf{w}}} \; S \quad = \quad P \; \underline{\underline{\mathbf{w}}} \; (R \; \underline{\underline{\mathbf{w}}} \; S)$

7.4   Other operations on IOEDPs

Also, the shuffle and the concatenation of IOEDPs can be defined (we denote them by $\underline{\mathbf{s}}$ and ; respectively). The definitions are similar to those given in chapter 2. We do not repeat them here.

IOEDPs can also be ordered. We use the same ordering as before:

Definition 7.7 *For two IOEDPs P and R, the ordering $P \subseteq R$ is defined by:*

$\mathbf{i}P = \mathbf{i}R \; \wedge \; \mathbf{o}P = \mathbf{o}R \; \wedge \; \mathbf{e}P = \mathbf{e}R \; \wedge \; \mathbf{t}P \subseteq \mathbf{t}R$

P is at most R if all the subalphabets of P and R are the same and the trace set of P is at most the trace set of R.

7.5   CODE for IOEDPs

The CODE-problem for IOEDPs can be redefined as follows:

Given are the IOEDPs $P = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{i}P, \mathbf{o}P \rangle$, L_{min}, and L_{max} with $L_{min} \subseteq L_{max} \subseteq P \lceil \mathbf{e}P$, find, if possible, an IOEDP R with $R = \langle \mathbf{t}R, \varnothing, \mathbf{i}R, \mathbf{o}R \rangle$ with $\mathbf{i}R \subseteq \mathbf{o}P$ and $\mathbf{o}R \subseteq \mathbf{i}P$ such that $L_{min} \subseteq P \underline{\mathbf{b}} R \subseteq L_{max}$.

The solution for this problem is analogous to chapter 3. Take

$$R = \langle f(L), \varnothing, \mathbf{o}P, \mathbf{i}P \rangle$$

with $f(L)$ the friend[3] of some L satisfying $L_{min} \subseteq L \subseteq L_{max}$. With $\mathbf{c}R = \mathbf{i}R \cup \mathbf{o}R$ all properties found in the previous chapters are still valid.

The CODE-problem is similar for discrete processes and for IOEDPs. As mentioned before a difference occurs when we regard deadlock.

7.6 Delayed communication

Thus far, we have considered communication in the sense that "sending" and "receiving" occur at the same time, i.e., if a is an input event in P and an output event in R, then in the connection event a occurs in P and in R at the very same time.

Example 7.8 Consider

$$P = \langle abd, \{d\}, \{a\}, \{b\} \rangle \quad R = \langle cbe, \{e\}, \{b, c\}, \varnothing \rangle$$

The connection results in

$$P \underline{\mathbf{b}} R = \langle (acde|aced|cade|caed), \{d, e\}, \{a, c\}, \varnothing \rangle$$

However, suppose some delay (*slack*) is possible between output by P and input by R. Then it is also possible to get the behaviour *adce* in the connection: first P receives a, then it sends b, and, in the delay between sending b by P and receiving b by R, events d and c may occur (for example in the order dc).

The sending of b and the receiving of b are in fact two different events (denoted by $b!$ and $b?$) with the property that the number of occurrences of $b?$ is at most that of the number of occurrences of $b!$.

(end of example)

In trace theory an operator exists, that performs connection in the sense as mentioned in the previous example. This operator is called *agglutinate* and the connection (in case of trace structures) is denoted by P **g** R. The agglutinate can be defined using a special class of trace structures, named **del**, and a function mapping an alphabet onto an IOEDP, denoted by H. $\mathbf{del}(b, c)$ is a class of trace structures in which, at any time in the behaviour, at least as many b's as c's have occurred. $H(A)$ denotes an IOEDP that describes a possibly delayed transmission of elements of A via a transmission line with infinite capacity, i.e., each event in A may have a delay between sending and receiving. **del** and H are formally defined by:

Definition 7.9 *For two events b and c, the class of trace structures denoted by* **del** *is defined by:*

_____
[3]To be formal: $f(L) = \mathbf{t}F(\mathbf{dp}(P), L)$.

$$\mathbf{del}(b,c) = \langle\{x : x \in \{b,c\}^* \wedge (\forall y,z : x = yz : y\mathbf{N}b \geq y\mathbf{N}c) : x\}$$
$$, \{b,c\}$$
$$\rangle$$

where $y\mathbf{N}b$ means the number of occurrences of event b in trace y.
The function H mapping an alphabet onto an IOEDP is defined by:

$$H(\varnothing) \quad = \quad \langle\{\epsilon\}, \varnothing, \varnothing, \varnothing\rangle$$
$$H(A \cup \{b\}) \quad = \quad H(A) \ \underline{\mathbf{b}} \ \langle\mathbf{t}(\mathbf{del}(b!,b?)), \varnothing, \{b!\}, \{b?\}\rangle$$

Furthermore, with

$$P?!(A_i, A_o)$$

we denote the IOEDP P with all symbols $b \in \mathbf{a}P \cap A_i$ replaced by $b?$ and all symbols $b \in \mathbf{a}P \cap A_o$ replaced by $b!$.

Notice that all symbols in the IOEDPs that are connected in H are different so $\underline{\mathbf{b}}$ may as well be replaced by $\underline{\mathbf{w}}$. Further notice that in H all output events are put in the input alphabet and all input events are put in the output alphabet. This can be explained if we think of H as an intermediate that accepts the outputs of some process (so these events are inputs with respect to H) and (probably with some delay) sends that event as the corresponding input event of some other process (so this event is an output with respect to H).

Definition 7.10 *The agglutinate of P and R is defined if:*

$$\mathbf{e}P \cap \mathbf{a}R = \varnothing \quad \mathbf{e}R \cap \mathbf{a}P = \varnothing$$
$$\mathbf{i}P \cap \mathbf{i}R = \varnothing \quad \mathbf{o}P \cap \mathbf{o}R = \varnothing$$

by:

$$P \ \mathbf{g} \ R$$
$$=$$
$$P?!(\mathbf{i}P \cap \mathbf{o}R, \mathbf{o}P \cap \mathbf{i}R) \ \underline{\mathbf{b}} \ H(\mathbf{a}P \cap \mathbf{a}R) \ \underline{\mathbf{b}} \ R?!(\mathbf{i}R \cap \mathbf{o}P, \mathbf{o}R \cap \mathbf{i}P)$$

The agglutinate of P and R is in fact a connection of P, R, and the intermediate H, where events that play a role in the connection between P and R are directed through H, where they may be delayed.

We illustrate this complicated definition in the following example:

Example 7.11 Computing the agglutinate of P and R from example 7.8 is done as follows:

$$P?!(\mathbf{i}P \cap \mathbf{o}R, \mathbf{o}P \cap \mathbf{i}R) \quad = \quad P?!(\varnothing, \{b\})$$
$$= \quad \langle ab!d, \{d\}, \{a\}, \{b!\}\rangle$$
$$\stackrel{\text{def}}{=} \quad P?!$$
$$R?!(\mathbf{i}R \cap \mathbf{o}P, \mathbf{o}R \cap \mathbf{i}P) \quad = \quad R?!(\{b\}, \varnothing)$$
$$= \quad \langle cb?e, \{e\}, \{c,b?\}, \varnothing\rangle$$
$$\stackrel{\text{def}}{=} \quad R?!$$

$$H(\mathbf{a}P \cap \mathbf{a}R) \quad = \quad H(\{b\})$$
$$= \quad \langle\mathbf{tdel}(b!,b?), \varnothing, \{b!\}, \{b?\}\rangle$$
$$= \quad \langle(\epsilon|b!|b!b?|b!b!|b!b?b!|b!b?b!b?|\ldots), \varnothing, \{b!\}, \{b?\}\rangle$$

which results in:

$$P?! \; \underline{\mathbf{b}} \; H(\{b\}) \qquad = \quad \langle (ad|ab?d|adb?), \{d\}, \{a\}, \varnothing \rangle$$
$$P?! \; \underline{\mathbf{b}} \; H(\{b\}) \; \underline{\mathbf{b}} \; R?! \quad = \quad \langle (aced|acde|caed|cade|adce), \{d, e\}, \{a, c\}, \varnothing \rangle$$

as was given in the example.

(end of example)

Agglutination is similar to blending, but it expresses unbounded finite delay and overtaking.[4] The operator **g** however is much more difficult than the blend. In [JvdS] and [JTU] restrictions have been developed under which composition of structures is delay insensitive and it is sufficient to use the blend instead of the agglutination.

Because the agglutination is expressed in terms of the blend, and no symbol occurs in more than two of the alphabets, all essential properties for the blend also hold for the agglutination. However, because the **del**-process is not regular, it can be shown, that the agglutination of two regular processes need not result in a regular process (see [JvdS]).[5]

It is straightforward to translate the CODE problem in terms of the agglutination: simply replace every $\underline{\mathbf{b}}$ with a **g** . However, because agglutination does not preserve regularity, we are unable to use the methods using state graphs to compute solutions of the corresponding CODE-problem.

Trying to take into account the notion of delay results in an operator, the agglutinate, that is impractical for normal usage. Trace theory and hence CODE are unable to formalise the notion of control of discrete processes with delayed transmission.

7.7 Weak deadlock

Introducing inputs and outputs creates another kind of deadlock, which we will call *weak deadlock*. Deadlock as defined earlier will be referred to as *strong deadlock*.

We consider two IOEDPs, given by:

$$P = \langle \mathbf{t}P, \mathbf{e}P, \mathbf{i}P, \mathbf{o}P \rangle$$
$$R = \langle \mathbf{t}R, \mathbf{e}R, \mathbf{i}R, \mathbf{o}R \rangle$$
$$P \; \underline{\mathbf{w}} \; R \text{ is defined}$$

The behaviour of P and R in this total connection is according to $P \; \underline{\mathbf{w}} \; R$. If at a certain point in communication (i.e., some $x \in \mathbf{pref}(\mathbf{t}(P \; \mathbf{w} \; R)))$, that is not an endpoint ($x \notin \mathbf{end}(P, R)$), it is possible for one process to generate an event that cannot be accepted by the other process, we speak of weak deadlock. Formally:

---

[4]i.e., the order in time between signals is not preserved by their transmission: it is a direct consequence of the independence of delays.

[5]Intuitively, this can be seen as follows: the **del**-process acts as some kind of buffer, an unbounded buffer to be precise, so we need an unbounded number of states to collect all the elements of the buffer and therefore the process need not be regular anymore.
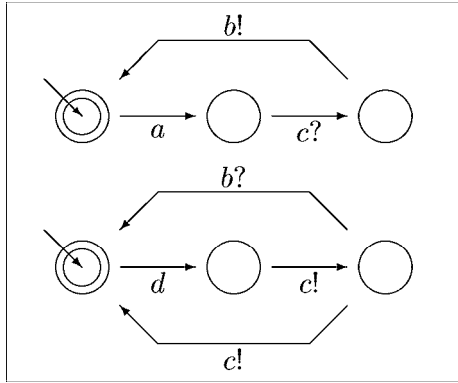
Figuur 7.1: Example of weak deadlock between P (above) and R (below)

Definition 7.12 *Two IOEDPs P and R may (in connection) end in* weak deadlock, *notation* **weakdeadlock**(P, R), *if:*

$$(\exists x : x \notin \mathbf{end}(P, R) \wedge x\lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \wedge x\lceil \mathbf{a}R \in \mathbf{t}(\mathbf{pref}R)$$
$$: (\exists a : a \in \mathbf{o}P \wedge xa\lceil \mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) : xa\lceil \mathbf{a}P \notin \mathbf{t}(\mathbf{pref}R)) \vee$$
$$(\exists a : a \in \mathbf{o}R \wedge xa\lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R) : xa\lceil \mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P)))$$

We explicitly omit the situation that $x \in \mathbf{end}(P, R)$. Such a trace x is a completed task and it is not satisfactory to call a completed task deadlocked.

Example 7.13 We give a very simple example to illustrate the definition with (see figure 7.1):

$$P = \langle (acb)^*, \{a\}, \{c\}, \{b\} \rangle \qquad R = \langle (dcb|dcc)^*, \{d\}, \{b\}, \{c\} \rangle$$

Take $x = adc$, then:

$$x\lceil \mathbf{a}P = ac \in \mathbf{pref}(\mathbf{t}P) \quad xc\lceil \mathbf{a}R \in \mathbf{pref}(\mathbf{t}R) \wedge c \in \mathbf{o}R$$
$$x\lceil \mathbf{a}R = dc \in \mathbf{pref}(\mathbf{t}R) \quad xc\lceil \mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P)$$

We have weak deadlock: R may output event c instead of waiting to receive event b. If the inputs and outputs of P and R are interchanged, weak deadlock will occur at another point.
(end of example)

Example 7.14 Consider $P = \langle a^*, \emptyset, \{a\}, \emptyset \rangle$ and $R = \langle a, \emptyset, \emptyset, \{a\} \rangle$, then we have **deadlock**(P, R), e.g., for $x = a$ we find $x \notin \mathbf{end}(P, R) \wedge xa \notin \mathbf{pref}(\mathbf{t}R)$), but ¬**weakdeadlock**$(P, R)$, e.g., for $x = a$ we have $xa \notin \mathbf{pref}(\mathbf{t}R)$, but also $a \notin \mathbf{o}P$ (I.e., a process waiting for an input is not considered deadlocked). So in general **deadlock**$(P, R) \not\Rightarrow$ **weakdeadlock**(P, R).
(end of example)

7.8 Detecting weak deadlock

In this section we give a method to detect weak deadlock in case P and R are regular. We have already found a method to detect strong deadlock and we will use a similar method to detect weak deadlock.
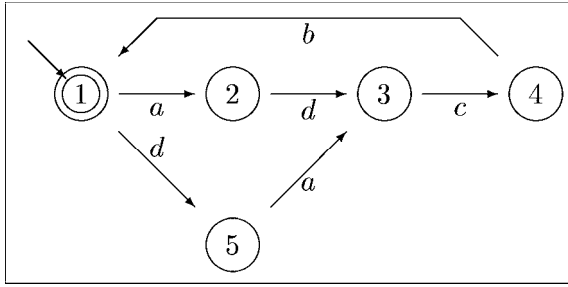
Figuur 7.2: Corresponding deadlock recognizer for example 7.13

Again, we need the corresponding state graphs for P and R and construct the deadlock recognizer $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ in which deadlock can be detected.

Example 7.15 If we reconsider the example from figure 7.1 and construct the corresponding $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ (see figure 7.2), we see that no deadlock state exists.

The connection has no strong deadlock. However, as shown it has weak deadlock. We have to do some extra work to detect weak deadlock.
(end of example)

To use $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ to detect weak deadlock we need the following definition:

Definition 7.16 *A state (p_1, p_2) from $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ is called a trouble state if:*

$$(\exists a : a \in \mathbf{o}P : (\delta_p(p_1, a) = [\varnothing]_P \;\wedge\; \delta_R(p_2, a) \neq [\varnothing]_R) \;\vee$$
$$(\exists a : a \in \mathbf{o}R : (\delta_p(p_1, a) \neq [\varnothing]_P \;\wedge\; \delta_R(p_2, a) = [\varnothing]_R)$$

If we identify the states in $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ in a way, corresponding to the states in $\mathbf{sg}(P)$ and $\mathbf{sg}(R)$, i.e., denote the states with (p_1, p_2) where p_1 is a state in $\mathbf{sg}(P)$ and p_2 is a state in $\mathbf{sg}(R)$, we can use $\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ to detect weak deadlock as well:

Theorem 7.17

weakdeadlock(P, R)

$\Leftrightarrow$

$\mathbf{dr}(\mathbf{sg}(P), \mathbf{sg}(R))$ *has a trouble state*

proof: obvious (similar to the proof of theorem 6.13)
(end of proof)

Example 7.18 Reconsider the last example. If we add the additional information to the diagram in figure 7.2 we derive figure 7.3. If we denote by $A(p)$ the *active set* of state p, i.e.,

$$A(p) = \{a : \delta(p, a) \neq [\varnothing] : a\}$$

we derive table 7.1. Trouble states can be found by comparing $A(p)$, $A(q)$ and $A(p, q)$. If $A(p)$ contains an output event (of P) that is not in $A(p, q)$ or if $A(q)$ contains an output event (of R) that is not in $A(p, q)$, we have a trouble state (p, q).

In the table we detect two trouble states, namely $(1, 2)$ and $(3, 3)$. In $(1, 2)$ we have weak deadlock if R outputs event c before in P event a has occurred. In $(3, 3)$ we have weak deadlock if R decides to output event c (P can only accept b at this point).
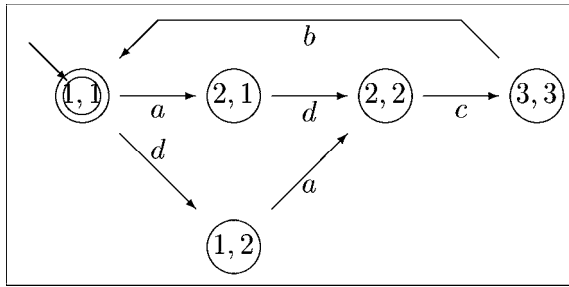
Figuur 7.3: Extended deadlock recognizer for example 7.13

$(p,q) \in \mathbf{dr}(\mathbf{sg}(P),\mathbf{sg}(R))$	$A(p)$	$A(q)$	$A(p,q)$	remarks
1,1	a	d	a,d	
2,1	c	d	d	$c \notin \mathbf{o}P$
1,2	a	c	a	$c \in \mathbf{o}R$
2,2	c	c	c	
3,3	b	b,c	b	$c \in \mathbf{o}R$

Tabel 7.1: Table for detecting weak deadlock

If we interchange inputs and outputs in this example we get one trouble state (namely $(2,1)$).

(end of example)

7.9 Weak deadlock free controllers

Again, we are able to construct controllers for the CODE-problem that are free of weak deadlock. We use the same method as in chapter 6. The only change we have to make before we can use that method is to redefine $d(P,R)$ in

$d(P,R)$

$=$

$\{x : x \notin \mathbf{end}(P,R) \ \wedge \ x\lceil\mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ x\lceil\mathbf{a}R \in \mathbf{pref}(\mathbf{t}R)$
$\quad : (\exists a : a \in \mathbf{o}P \ \wedge \ xa\lceil\mathbf{a}P \in \mathbf{pref}(\mathbf{t}P) : xa\lceil\mathbf{a}R \notin \mathbf{pref}(\mathbf{t}R)) \ \vee$
$\quad\quad (\exists a : a \in \mathbf{o}R \ \wedge \ xa\lceil\mathbf{a}R \in \mathbf{pref}(\mathbf{t}R) : xa\lceil\mathbf{a}P \notin \mathbf{pref}(\mathbf{t}P))$

With this new $d(P,R)$, which denotes all traces that are candidates for weak deadlock, we can follow exactly the same construction as mentioned in chapter 6. Because it is obvious, we do not display it again.

8

Determinism

*There is no dark side of the moon
really
as a matter of fact it's all dark*

Dark side of the moon (epilogue)

In this chapter we deal with the notion determinism. First, we explain a kind of determinism as introduced in [Mil]. Next, we introduce our kind of determinism and the connection to deadlock.

8.1 Non-determinism in CCS

Consider the discrete processes

$$P = \langle (ab|ac)^*, \{a\}, \{b,c\} \rangle \quad P' = \langle (a(b|c))^*, \{a\}, \{b,c\} \rangle$$

Different interpretations are possible concerning the mechanism generating the behaviour of this processes. One of them is that in P exogenous event a occurs after which either b or c is possible, depending on the path chosen. If P "decides" to do path (ab), then after a only b is possible and if P "decides" to do path (ac), then after a only c is possible. Here we have some kind of non-determinism: P itself makes its choice between path (ab) and (ac) and the environment[1] has no ways to influence this choice. It can only observe which path has been chosen. If b and c are input events of an IOEDP P, then choosing one path allows only one of the input events to be accepted: the other one causes deadlock. Problems arise when we use P in connection with other discrete processes. Consider for example

$$R = \langle (bc), \varnothing, \{b,c\} \rangle$$

If we asume the above non-determinism, then connecting P with R may cause problems: if P decides to do (ac) first, we have deadlock.

This kind of non-determinism is common in the calculus of communicating processes (CCS) of Milner (see [Mil]). We shall not adopt it here. In fact, we consider P and P'

---

[1]Think of the environment as a second discrete process that is connected to P and can control the communication events of P.

94
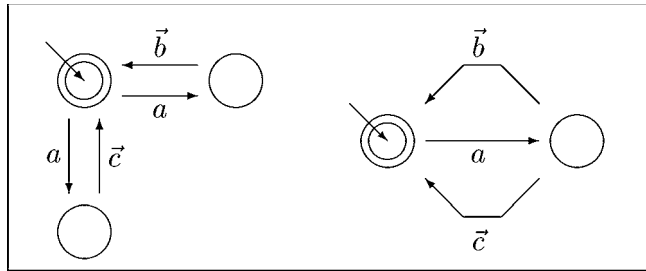
Figuur 8.1: Possible graphs for P (left) and P' (right)

to be equivalent. i.e., we postulate that possible choices in the paths are postponed as much as possible. Notice that the trace sets of P and P' are the same, viz.

$$\{\epsilon, ab, ac, abab, abac, acab, \ldots\}$$

In terms of state graphs we replace every non-deterministic graph (the left graph in figure 8.1) by a deterministic one (the right graph). As a consequence we consider the state graphs in figure 8.1 to be equivalent.

Because, by definition, $\mathbf{sg}(P) = \mathbf{sg}(P')$ (both equal to the right graph of figure 8.1) we cannot tell any difference between P and P'. In our formalisation this kind of non-determinism does not occur. The above problems therefore do not occur. However, another kind of non-determinism is possible and this is discussed in the next section.

8.2 Deterministic discrete processes

Consider

$$P = \langle ((eb)|(dc))^*, \{d, e\}, \{b, c\}\rangle$$

Notice that d and e are exogenous events. The choice between occurrence of d or e therefore is completely made by P itself. The environment cannot influence this. If we connect P with R, where R is given by

$$R = \langle (bc), \emptyset, \{b, c\}\rangle$$

then we have the possibility of ending in deadlock if P decides to do d the first time or e the second time. This time the decision of doing (eb) or (dc) cannot be postponed because d and e are different events. The occurrence of (eb) or (dc) cannot even be controlled by the environment. This phenomenon causes *non-determinism*: the possibility of a process making decisions that cannot be influenced by the environment.

Before giving a formal definition of determinism we recall the definition given by Hoare in [Hoa]:

> A process is deterministic, if, whenever there is more than one event possible, the choice between them is determined externally by the environment of the process.

Exogenous events are considered to be internal and cannot be controlled by the environment directly. The above definition can therefore be understood as:

> A process is deterministic if, at any point in its behaviour, the possible future external behaviour is independent of possible choices in internal behaviour.[2]

---

[2] External should be written as communication here and internal as exogenous.

According to this definition the above process P is not deterministic: initially, P itself "decides" to do d or e, which results in a future communication behaviour starting with event b or c respectively, thus result in different communication behaviour.

Example 8.1 Process P given by

$$P = \langle (ea|ae)^*, \{e\}, \{a\} \rangle$$

is deterministic. Initially, it can choose between event e or event a. The first choice leads to the first communication event a, the second choice also leads to a.
(end of example)

Consider some prefix t of the behaviour of a process P. If in state $[t]$ only communication events are possible, the environment can observe which of these events is "chosen" by P. If exactly one exogenous event is possible in $[t]$, the environment cannot notice its occurrence. However, we reach a unique next state in P, so again we have a deterministic progress in the behaviour. This leaves us with the situation that from $[t]$ more than one event is possible and at least one of them is an exogenous event. If the choice made by P at this point may lead to different future communication behaviour, we say that P is non-deterministic. This results in the following definition of determinism. First, however, we need one more notion.

Definition 8.2

$$\mathbf{rest}(P, t) = \{x : tx \in \mathbf{t}P : x\}$$

Definition 8.3 *A discrete process P is* deterministic, *notation* **deterministic**(P), *if*

$$(\forall t, e, a : te \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ ta \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ a \in \mathbf{a}P \ \wedge \ e \in \mathbf{e}P$$
$$: a(\mathbf{rest}(P, ta))\lceil\mathbf{c}P = e(\mathbf{rest}(P, te))\lceil\mathbf{c}P)$$

where we use the notation:

$$a(\mathbf{rest}(P, t)) = \{y : (\exists z : z \in \mathbf{rest}(P, t) : y = az) : y\}$$

Determinism has to be checked only in those states of P from which more than one event is possible and at least one of them is an exogenous event. Therefore, in order to be able to detect determinism in state graphs, we introduce the choice states, the set of all states in which more than one event is possible, the exo states, the set of all states in which an exogenous event is possible, and the danger sets, the set of all pairs (p, A), with p a choice state as well as an exo state and A the set of all events that are possible in p.

Definition 8.4

> **choicestates**(P)
>
> $=$
>
> $\{t : t \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ (\exists a_1, a_2 : a_1 \in \mathbf{a}P \ \wedge \ a_2 \in \mathbf{a}P \ \wedge \ a_1 \neq a_2$
> $\qquad\qquad\qquad\qquad : ta_1 \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ ta_2 \in \mathbf{pref}(\mathbf{t}P))$
> $\quad : [t]\}$

exostates(P)

$=$

$\{t : t \in \mathbf{pref}(\mathbf{t}P) \ \wedge \ (\exists e : e \in \mathbf{e}P : te \in \mathbf{pref}(\mathbf{t}P))$
   $: [t]\}$

dangersets(P)

$=$

$\{t, a : [t] \in \mathbf{choicestates}(P) \cap \mathbf{exostates}(P) \ \wedge$
   $A \subseteq \mathbf{a}P \ \wedge \ (\forall a : a \in A : ta \in \mathbf{pref}(\mathbf{t}P))$
   $: ([t], A)\}$

Property 8.5

dangersets(P)

$=$

$\{t : [t] \in \mathbf{choicestates}(P) \cap \mathbf{exostates}(P) : ([t], A([t]))\}$

with $A(p)$ the active set[3] of state p.

Lemma 8.6

deterministic(P)

$\Leftrightarrow$

$(\forall t, A : ([t], A) \in \mathbf{dangersets}(P)$
   $: (\forall a_1, a_2 : a_1 \in A \ \wedge \ a_2 \in A$
     $: a_1(\mathbf{rest}(P, ta_1))\lceil \mathbf{c}P = a_2(\mathbf{rest}(P, ta_2))\lceil \mathbf{c}P))$

proof: Consider some arbitrary $t \in \mathbf{pref}(\mathbf{t}P)$. If $[t] \in \mathbf{choicestates}(P) \cap \mathbf{exostates}(P)$, determinism follows from the given condition. Suppose

$[t] \notin \mathbf{choicestates}(P) \cap \mathbf{exostates}(P)$

then we have two possibilities:

1) $(\exists! a : a \in \mathbf{e}P : ta \in \mathbf{pref}(\mathbf{t}P))$
2) $(\forall a : a \in \mathbf{a}P : ta \in \mathbf{pref}(\mathbf{t}P) : a \in \mathbf{c}P)$

In both cases the condition for being deterministic in the definition reduces to a uniform quantification with empty domain.
(end of proof)

According to this lemma we only have to check the danger states of P. All paths leaving such states should lead to the same communication behaviour. We give a number of examples to illustrate the way determinism can be determined.

---

[3]see example 7.18.

Example 8.7 Consider

$$P = \langle (ae|ea)(de|c), \{e\}, \{a,c,d\} \rangle$$

We find **dangersets**$(P) = \{([\epsilon], \{a,e\})\}$ and

$$\textbf{rest}(P,a) = e(de|c) \quad ae(de|c)\lceil \textbf{c}P = (d|c)$$
$$\textbf{rest}(P,e) = a(de|c) \quad ea(de|c)\lceil \textbf{c}P = (d|c)$$

so P is deterministic. It is sufficient to look at all paths in the state graph from $|c|$ to $[ae]$, because all paths lead to the same state $[ae]$.
(end of example)

Example 8.8

$$P = \langle (e(ea|de)), \{e,d\}, \{a\} \rangle$$

Gives **dangersets**$(P) = \{([e], \{e,d\})\}$ and

$$\textbf{rest}(P,ee) = a \quad ea\lceil \textbf{c}P = a$$
$$\textbf{rest}(P,ed) = e \quad de\lceil \textbf{c}P = \epsilon$$

So P is not deterministic.
(end of example)

Example 8.9

$$P = \langle (e(ae|bd)), \{d,e\}, \{a,b\} \rangle$$

Gives **dangersets**$(P) = \emptyset$ (after e only communication events are possible), so P is deterministic.
(end of example)

If P and R are deterministic, then $P \underline{\textbf{b}} R$, $P \underline{\textbf{w}} R$ and $P \underline{\textbf{s}} R$ need not be deterministic. The following examples illustrate this.

Example 8.10

$$P = \langle (aec|bgd), \{e,g\}, \{a,b,c,d\} \rangle \qquad R = \langle (a|b), \emptyset, \{a,b\} \rangle$$

leads to $P \underline{\textbf{b}} R = \langle (ec|gd), \{e,g\}, \{c,d\} \rangle$, which is not deterministic (choice of e and g causes non-determinism), and $P \underline{\textbf{w}} R = \langle (aec|bgd), \{a,b,e,g\}, \{c,d\} \rangle$, which is also not deterministic (the events a and b have become exogenous events now and the choice between a and b causes non-determinism).
(end of example)

Example 8.11

$$P = \langle (db), \{d\}, \{b\} \rangle \qquad R = \langle (ec), \{e\}, \{c\} \rangle$$

leads to $S = P \underline{\textbf{s}} R = \langle (dbec|debc|decb|edbc|edcb|ecdb), \{d,e\}, \{b,c\} \rangle$, which is not deterministic, for example:

$$\textbf{rest}(S,db) = ec \qquad (b(\textbf{rest}(S,db)))\lceil \textbf{c}S = bc$$
$$\textbf{rest}(S,de) = bc|cb \quad (e(\textbf{rest}(S,de)))\lceil \textbf{c}S = bc|cb$$

(end of example)

8.3   Determinism and deadlock

In the previous section we have seen that the connection of two deterministic processes need not be deterministic. In this section we investigate the connection of a process and its controller according to some CODE problem.

We wish to develop a condition under which the connection of two processes P and R with $R \subseteq P\lceil cP$ does not lead to deadlock, i.e., when the controller R is said to be deadlock-free.

Example 8.12   First notice that the condition $R \subseteq P\lceil cP$ itself is not enough to have $\neg\textbf{deadlock}(P, R)$. Consider

$$P = \langle (c|eb), \{e\}, \{b, c\} \rangle \qquad R = \langle c, \varnothing, \{c\} \rangle$$

The prefix $x = e$ has the properties:

$$e \notin \textbf{end}(P) \qquad e|cP \notin \textbf{end}(R)$$
$$e \in \textbf{pref}(tP) \quad e|cP \in \textbf{pref}(tR)$$

and

$$ee \notin \textbf{pref}(tP) \qquad eb\lceil cP \notin \textbf{pref}(tR) \qquad ec \notin \textbf{pref}(tP)$$

so trace $x = e$ leads to deadlock. Notice that P is not deterministic.
(end of example)

According to this example, we might think that $\textbf{deterministic}(P)$ is enough to guarantee a deadlock-free connection. However, deadlock can also occur when we connect for example deterministic repeating task and deterministic single task processes:

Example 8.13   Consider

$$P = \langle (eb|b)b^*, \{a\}, \{b\} \rangle \qquad R = \langle b, \varnothing, \{b\} \rangle$$

then we have $\textbf{deterministic}(P)$, and $R \subseteq P\lceil cP$ but also $\textbf{deadlock}(P, R)$: the prefix $x = b$ leads to deadlock:

$$x \notin \textbf{end}(P) \qquad x \in \textbf{end}(R) \qquad \text{so: } x \notin \textbf{end}(P, R)$$
$$x \in \textbf{pref}(tP) \quad x\lceil cP \in \textbf{pref}(tR)$$

but $(\forall a : a \in aP : xa\lceil cP \notin \textbf{pref}(tR))$.

Notice that $\textbf{end}(P) = \varnothing$ and $R \subseteq P\lceil cP$ do not imply $\textbf{end}(R) = \varnothing$. In this example we have $\textbf{end}(R) = \{b\}$.
(end of example)

However, adding the extra condition $\textbf{end}(R) \subseteq \textbf{end}(P, R)\lceil cP$ is enough to guarantee a deadlock-free connection:

Theorem 8.14

$$R \subseteq P\lceil cP \ \wedge \ \textbf{deterministic}(P) \ \wedge \ \textbf{end}(R) \subseteq \textbf{end}(P, R)\lceil cP$$
$$\Rightarrow$$
$$\neg\textbf{deadlock}(P, R)$$

proof: Choose some x with

$$x \notin \textbf{end}(P, R) \ \wedge \ x \in \textbf{pref}(tP) \ \wedge \ x\lceil cP \in \textbf{pref}(tR)$$

Then we have three possibilities:

1) $(\exists! e : e \in \mathbf{e}P : xe \in \mathbf{pref}(\mathbf{t}P))$
2) $(\forall a : a \in \mathbf{a}P \wedge xa \in \mathbf{pref}(\mathbf{t}P) : a \in \mathbf{c}P)$
3) $(\exists a, e : a \in \mathbf{a}P \wedge e \in \mathbf{e}P \wedge a \neq e : xe \in \mathbf{pref}(\mathbf{t}P) \wedge xa \in \mathbf{pref}(\mathbf{t}P))$

Situation 1) cannot cause deadlock: $xe \lceil \mathbf{c}P = x \lceil \mathbf{c}P \in \mathbf{pref}(\mathbf{t}R)$ and $xe \in \mathbf{pref}(\mathbf{t}P)$.
Situation 2) cannot cause deadlock if one of these a's has the property that $xa \lceil \mathbf{c}P \in$ $\mathbf{pref}(\mathbf{t}R)$, which is fulfilled because $x \notin \mathbf{end}(P, R)$ and thus $x \lceil \mathbf{c}P \notin \mathbf{end}(R)$.
Situation 3) cannot cause deadlock because P is deterministic, so

$$a(\mathbf{rest}(P, xa)) \lceil \mathbf{c}P = e(\mathbf{rest}(P, xe)) \lceil \mathbf{c}P$$

From $x \lceil \mathbf{c}P \notin \mathbf{end}(R)$ we obtain that there is some $c \in \mathbf{c}P$ with $xc \in \mathbf{pref}(\mathbf{t}R) \subseteq$ $\mathbf{pref}(\mathbf{t}P \lceil \mathbf{c}P)$. So a possible event from the set of first communication events after x is event c and because all sets of first communication events (dependent on the chosen path) are equal we cannot have deadlock.
(end of proof)

If R is a repeating task process (i.e. $\mathbf{end}(R) = \emptyset$), we trivially establish the condition $\mathbf{end}(R) \subseteq \mathbf{end}(P, R) \lceil \mathbf{c}P$, so:

Corrollary 8.15 _CODE leads to deadlock-free controllers if P itself is deterministic and the controllers are repeating task processes._

Moreover, if we consider only repeating task processes,[4] we have:

Theorem 8.16 _CODE leads to deadlock-free controllers if P is deterministic._

---

[4]For example by replacing every non-repeating task process by its associated repeating task process according to section 6.3.

<div align="right">

9

Distributed control

</div>

<div align="right">

When you're one of the few
To land on your feet
What do you do
To make ends meet?
Teach

One of the few – The final cut

</div>

In this chapter we investigate the situation of a number of processes working in cooperation with each other. These processes have to be controlled individually, but in such a way that their overall behaviour is as desired. First, we give a formal description of this problem. Next, we give some possible solutions. We use the alternating bit protocol[1] (ABP) as leading example.

9.1   Problem formulation

Suppose we have two systems,[2] working in cooperation with each other. We wish to control each of these systems independently in order to achieve some desired overall behaviour. Formally:
Given is

$$P_i = \langle \mathbf{t}P_i, \mathbf{e}P_i, \mathbf{c}P_i \rangle \qquad i = 1, 2$$

with

$$\mathbf{e}Q = \mathbf{e}P_1 \cap \mathbf{e}P_2 \quad \text{cooperating events}$$
$$\mathbf{a}P_1 \cap \mathbf{a}P_2 = \mathbf{e}Q \quad \text{independency condition}$$

and two discrete processes describing the minimal and maximal desired behaviour L_{min} and L_{max} with

$$L_{min} \subseteq L_{max}$$
$$\mathbf{e}L_{min} = \mathbf{e}L_{max} = \mathbf{e}P_1 \cup \mathbf{e}P_2$$
$$\mathbf{c}L_{min} = \mathbf{c}L_{max} = \varnothing$$

---

[1]For a description of the alternating bit protocol, see [BSW].

[2]In this chapter we only consider the situation of two systems. It is not difficult to extend the problem to an arbitrary number of systems. Furthermore we restrict ourselves to simple discrete processes, i.e., we do not involve input/output systems here.

The problem is to find controllers

$$R_i = \langle \mathbf{t}R_i, \emptyset, \mathbf{c}P_i \rangle \qquad i = 1, 2$$

such that

$$L_{min} \subseteq (P_1 \, \underline{\mathbf{b}} \, R_1) \, \underline{\mathbf{s}} \, (P_2 \, \underline{\mathbf{b}} \, R_2) \subseteq L_{max}$$

The processes P_1 and P_2 can be seen as components of a system situated at different locations. The cooperation between these two processes is done via exogenous events. This assumption needs some explanation: because we want to control each process independently and do not want to control their cooperation, it is reasonable to suppose that all the events needed to perform this cooperation are uncontrollable. These events are internal with respect to the cooperation, so should be exogenous events.

This problem is called distributed control of discrete events (DICODE for short).

Without loss of generality we assume:

$$L_{min} \subseteq L_{max} \subseteq P_1 \lceil \mathbf{e}P_1 \, \underline{\mathbf{s}} \, P_2 \lceil \mathbf{e}P_2$$

(since we cannot reach a resulting exogenous behaviour that is outside the behaviour $P_1 \lceil \mathbf{e}P_1 \, \underline{\mathbf{s}} \, P_2 \lceil \mathbf{e}P_2$ for the same reasons as we cannot find a exogenous behaviour outside $P \lceil \mathbf{e}P$ in CODE) and

$$R_1 \subseteq P_1 \lceil \mathbf{c}P_1 \qquad R_2 \subseteq P_2 \lceil \mathbf{c}P_2$$

(since the only communication traces that have effect are those in the behaviour of P_1 and P_2).

9.2 Some observations

We start with a sufficient condition and a necessary condition for DICODE being solvable, formulated in the next theorem and lemma.

Theorem 9.1 *For $L_1 \subseteq P_1 \lceil \mathbf{e}P_1$ and $L_2 \subseteq P_2 \lceil \mathbf{e}P_2$ with $L_1 \, \underline{\mathbf{s}} \, L_2 \subseteq L_{max}$ we have:*

$$L_{min} \lceil \mathbf{e}P_1 \subseteq G(P_1, L_1) \, \wedge \, L_{min} \lceil \mathbf{e}P_2 \subseteq G(P_2, L_2)$$
$$\Rightarrow$$
DICODE is solvable

proof: We prove that $F(P_1, L_1)$ and $F(P_2, L_2)$ are solutions:

$$L_{min}$$
$$\subseteq$$
$$L_{min} \lceil \mathbf{e}P_1 \, \underline{\mathbf{s}} \, L_{min} \lceil \mathbf{e}P_2$$
$$\subseteq \quad [\text{ assumption }]$$
$$G(P_1, L_1) \, \underline{\mathbf{s}} \, G(P_2, L_2)$$
$$=$$
$$(P_1 \, \underline{\mathbf{b}} \, F(P_1, L_1)) \, \underline{\mathbf{s}} \, (P_2 \, \underline{\mathbf{b}} \, F(P_2, L_2))$$
$$=$$

$=$

$G(P_1, L_1) \; \underline{\mathbf{s}} \; G(P_2, L_2)$

$\subseteq$ [see lemma 3.9: $G(P, L) \subseteq L$]

$L_1 \; \underline{\mathbf{s}} \; L_2$

$\subseteq$ [assumption on L_1 and L_2]

L_{max}

(end of proof)

Lemma 9.2

DICODE is solvable $\wedge$

$(R_1 \; \underline{\mathbf{b}} \; R_1) \; \underline{\mathbf{s}} \; (P_2 \; \underline{\mathbf{b}} \; R_2) = (P_1 \; \underline{\mathbf{s}} \; P_2) \; \underline{\mathbf{b}} \; (R_1 \; \underline{\mathbf{s}} \; R_2)$

$\Rightarrow$

$L_{min} \subseteq G(P_1 \; \underline{\mathbf{s}} \; P_2, L_{max})$

proof:

DICODE solvable

$\Leftrightarrow$

$(\exists R_1, R_2 :: L_{min} \subseteq (P_1 \; \underline{\mathbf{b}} \; R_1) \; \underline{\mathbf{s}} \; (P_2 \; \underline{\mathbf{b}} \; R_2) \subseteq L_{max})$

$\Leftrightarrow$ [assumption]

$(\exists R_1, R_2 :: L_{min} \subseteq (P_1 \; \underline{\mathbf{s}} \; P_2) \; \underline{\mathbf{b}} \; (R_1 \; \underline{\mathbf{s}} \; R_2) \subseteq L_{max})$

$\Rightarrow$ [take $R = R_1 \; \underline{\mathbf{s}} \; R_2$]

$(\exists R :: L_{min} \subseteq (P_1 \; \underline{\mathbf{s}} \; P_2) \; \underline{\mathbf{b}} \; R \subseteq L_{max})$

(end of proof)

The assumption in lemma 9.2:

$(P_1 \; \underline{\mathbf{b}} \; R_1) \; \underline{\mathbf{s}} \; (P_2 \; \underline{\mathbf{b}} \; R_2) = (P_1 \; \underline{\mathbf{s}} \; P_2) \; \underline{\mathbf{b}} \; (R_1 \; \underline{\mathbf{s}} \; R_2)$

is a very strong one. It is not difficult to find an example where it is not satisfied.

Example 9.3   Choose

$P_1 = \langle (ae_1 | ae_2), \{e_1, e_2\}, \{a\} \rangle$ $R_1 = \langle a, \varnothing, \{a\} \rangle$

$P_2 = \langle (ce_1), \{e_1\}, \{c\} \rangle$ $R_2 = \langle c, \varnothing, \{c\} \rangle$

then we have

$\mathbf{t}((P_1 \; \underline{\mathbf{b}} \; R_1) \; \underline{\mathbf{s}} \; (P_2 \; \underline{\mathbf{b}} \; R_2))$ $=$ $e_1 | e_2$

$\mathbf{t}((P_1 \; \underline{\mathbf{s}} \; P_2) \; \underline{\mathbf{b}} \; (R_1 \; \underline{\mathbf{s}} \; R_2))$ $=$ e_1

(end of example)

9.3   Alternating bit protocol

Consider two systems, called *sender* and *receiver*. The sender sends messages to the receiver, one at the time. The sender is allowed to send a next message only after the previous one has been properly received. The receiver acknowledges receiving a message.

Unfortunately, the transmission line between the sender and the receiver is not completely reliable. Sometimes it mutilates messages. We assume, however, that every message sent along the line is received (complete or mutilated) at the other end. We
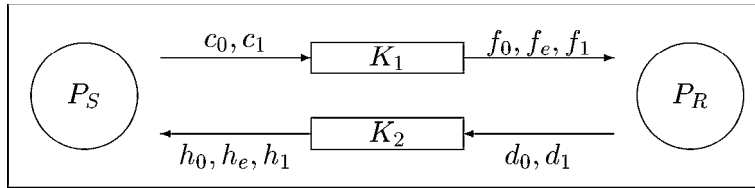
Figuur 9.1: Configuration for ABP

event	meaning	event	meaning
m	mesg to be sent	n	receiving mesg
c_0	send mesg with 0-flag	d_0	send ack with 0-flag
c_1	send mesg with 1-flag	d_1	send ack with 1-flag
a	ready for next mesg	b	ready
e_0	0-flag mesg transmitted	f_0	0-flag mesg received
e_e	mesg transmitted with error	f_e	error in receiving mesg
e_1	1-flag mesg transmitted	f_1	1-flag mesg received
h_0	0-flag ack transmitted	g_0	0-flag ack received
h_e	ack transmitted with error	g_e	error in receiving ack
h_1	1-flag ack transmitted	g_1	1-flag ack received

mesg = message   ack = acknowledgement

Tabel 9.1: Meaning of the events

assume some algorithm is present that can tell if a received message is mutilated. In order to be able to send messages along this line, one bit of information is added to the message before it is sent. It turns out that just one bit of extra information is enough to guarantee successful transmission. The configuration is shown in figure 9.1. The meaning of the events is given in table 9.1. In order to be compatible with the model as introduced in the first section we assume P_S and K_1 to be one system (discrete process P_1) and P_R and K_2 to be one system (discrete process P_2). The behaviour of P_1 and P_2 is given in figure 9.2 and figure 9.3.

From the literature (see [BSW]) we know that this way of data transmission can only be successful if we alternate the value of this extra bit. I.e., we would like to have the exogenous behaviour $L_{min} = L_{max}$ as given in figure 9.4.[3] It is beyond the scope of this chapter to explain why the exogenous behaviour of figure 9.4 leads to successful transmission and simpler behaviour does not. We only remark that $tL_{max}\lceil\{m,n\} = (mn)^*$ (each message is first completely transmitted and handled before a new message is transmitted). P_1 and P_2 are modelled with as much freedom as possible. Our problem consists of finding suitable controllers for P_1 and P_2 to establish L_{max}, the desired exogenous behaviour of the whole system. We emphasize that we do not invent the alternating bit protocol here, nor prove its correctness, but only try to find suitable controllers for both sender and receiver, such that the transmissionline as a whole behaves according to the alternating bit protocol. This problem is referred to as the ABP-problem.

Recalling the previous section we see that lemma 9.2 will not be of much use here. Theorem 9.1 gives possibilities. However, using it on the ABP-problem does not directly

---

[3]We have drawn just one simple edge with multiple labels in case more events causes the same state transition.
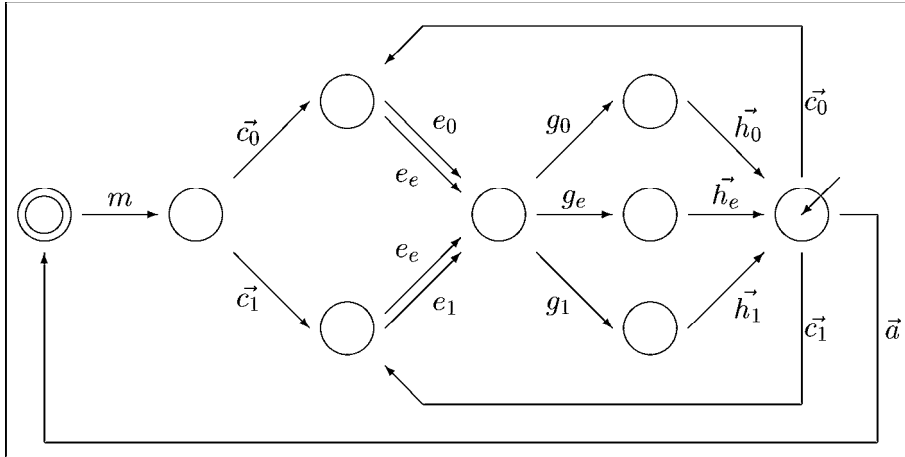
Figuur 9.2: Diagram for process P_1
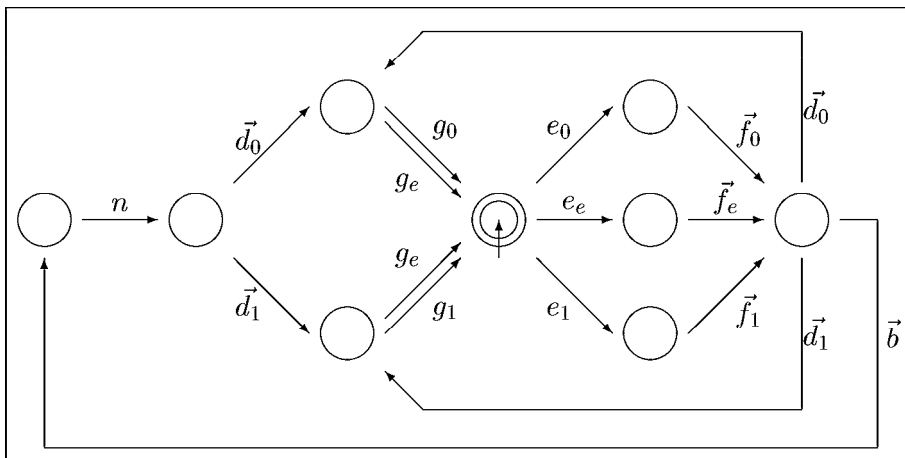
Figuur 9.3: Diagram for process P_2
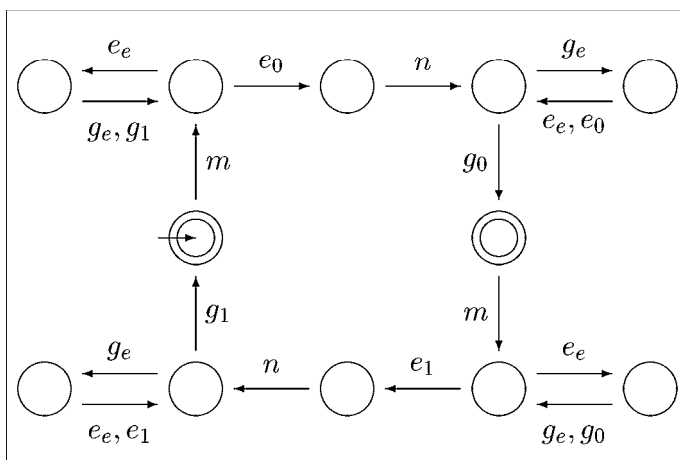
Figuur 9.4: Desired exogenous behaviour L_{max}
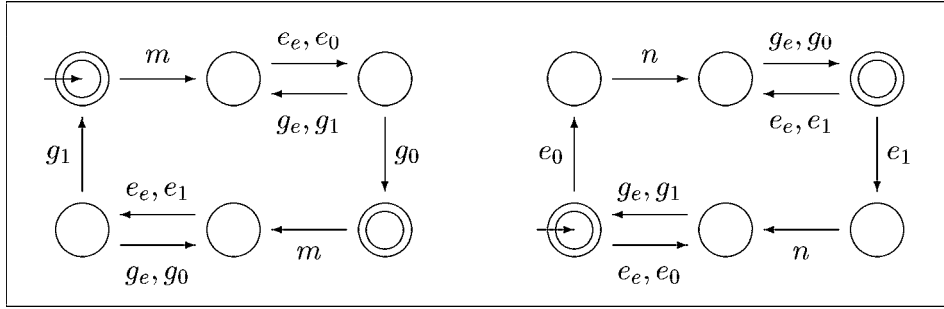
Figuur 9.5: Useful exogenous behaviour for P_1 (left) and P_2 (right) to solve the ABP-problem
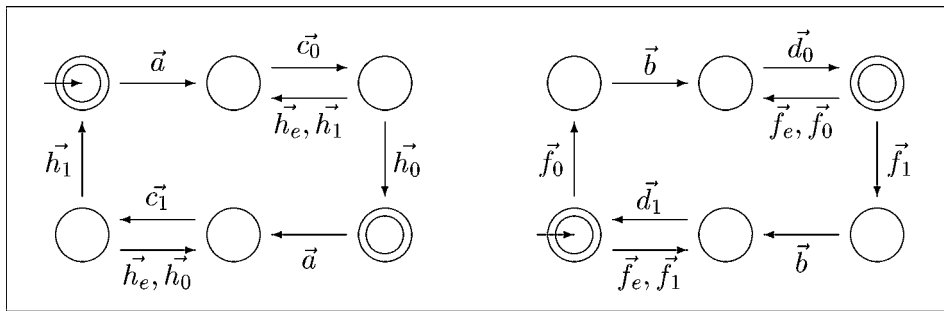
Figuur 9.6: Resulting controllers R_1 (left) and R_2 (right) using the behaviours as given in figure 9.5

lead to success, as is shown in the next section.

9.4 Solving the ABP-problem

It can be shown that for the ABP-problem we have:

$$L_{max}\lceil eP_1 \ \underline{s} \ L_{max}\lceil eP_2 = L_{max}$$

so possible candidates to use for solving the ABP-problem are

$$L_1 = L_{max}\lceil eP_1 \qquad L_2 = L_{max}\lceil eP_2$$

However, computing $F(P_1, L_1)$ and $F(P_2, L_2)$ leads to empty controllers, i.e., the ABP-problem cannot be solved with this L_1 and L_2.

It is, however, possible to find controllers for P_1 and P_2 such that the behaviour is as described by L_{max}. If we use L_1 and L_2 as given in figure 9.5, we find $L_1 \ \underline{s} \ L_2 = L_{max}$ and

$$F(P_1, L_1) \ \underline{b} \ P_1 = L_1 \qquad F(P_2, L_2) \ \underline{b} \ P_2 = L_2$$

The controllers are given in figure 9.6.

From the ABP-problem and the above solution, we conclude that not all L_1 and L_2 with $L_1 \ \underline{s} \ L_2 = L_{max}$ give solutions. Notice that $L_{max}\lceil eP_1 \subseteq L_1$ and $L_{max}\lceil eP_2 \subseteq L_2$, so probably L_1 and L_2 should be as large as possible.

9.5   Non-trivial solution

A necessary and sufficient condition for DICODE to have a solution is given in the next lemma.

Lemma 9.4 *For* $L_1^m \subseteq P_1 | eP_1$ *and* $L_2^m \subseteq P_2 | eP_2$ *with* $L_1^m \underline{s} L_2^m \subseteq L_{max}$ *and*

$$(\forall L_1, L_2 : L_1 \subseteq P_1 \lceil eP_1 \ \wedge \ L_2 \subseteq P_2 \lceil eP_2 \ \wedge \ L_1 \underline{s} L_2 \subseteq L_{max}$$
$$: L_1 \subseteq L_1^m \ \wedge \ L_2 \subseteq L_2^m)$$

we have that:

    DICODE is solvable

$\Leftrightarrow$

    $L_{min} \lceil eP_1 \subseteq G(P_1, L_1^m) \ \wedge \ L_{min} \lceil eP_2 \subseteq G(P_2, L_2)$

proof: That the condition is sufficient can be found in theorem 9.1. That it is necessary is proven below:

    DICODE is solvable

$\Leftrightarrow$

    $(\exists R_1, R_2 :: L_{min} \subseteq (P_1 \ \underline{b} \ R_1) \ \underline{s} \ (P_2 \ \underline{b} \ R_2) \subseteq L_{max})$

$\Rightarrow$

    $(\exists R_1, R_2 :: L_{min} \lceil eP_1 \subseteq ((P_1 \ \underline{b} \ R_1) \ \underline{s} \ (P_2 \ \underline{b} \ R_2)) \lceil eP_1)$

$\Rightarrow$ [see corrollary 1.22]

    $(\exists R_1 :: L_{min} \lceil eP_1 \subseteq P_1 \ \underline{b} \ R_1)$

$\Rightarrow$ [see lemma 3.23]

    $(\exists R_1 :: L_{min} \lceil eP_1 \subseteq G(P_1, P_1 \ \underline{b} \ R_1))$

$\Rightarrow$ [take $L_1 = P_1 \ \underline{b} \ R_1$ and $L_2 = P_2 \ \underline{b} \ R_2$]

    $L_{min} \lceil eP_1 \subseteq G(P_1, L_1)$

$\Rightarrow$ [$L_1 \subseteq L_1^m$]

    $L_{min} \lceil eP_1 \subseteq G(P_1, L_1^m)$

Similar for P_2 and L_2.
(end of proof)

This lemma is useful only if L_1^m and L_2^m can be found. Therefore, we reduce the problem to a problem about trace structures and try to solve this one first.

9.6   The S_1^m, S_2^m-problem

In this section we try to solve the following problem:
Given:

$$T_1 = \langle \mathbf{t}T_1, \mathbf{a}T_1 \rangle \qquad L_{min} = \langle \mathbf{t}L_{min}, \mathbf{a}T_1 \cup \mathbf{a}T_2 \rangle$$
$$T_2 = \langle \mathbf{t}T_2, \mathbf{a}T_2 \rangle \qquad L_{max} = \langle \mathbf{t}L_{max}, \mathbf{a}T_1 \cup \mathbf{a}T_2 \rangle$$
$$L_{min} \subseteq L_{max} \subseteq T_1 \ \mathbf{w} \ T_2$$

find S_1^m and S_2^m with:

$$S_1^m \subseteq T_1 \quad L_{min} \subseteq S_1^m \ \mathbf{w} \ S_2^m \subseteq L_{max}$$
$$S_2^m \subseteq T_2$$

such that

$$(\forall S_1, S_2 : S_1 \subseteq T_1 \ \wedge \ S_2 \subseteq T_2 \ \wedge \ L_{min} \subseteq S_1 \ \mathbf{w} \ S_2 \subseteq L_{max}$$
$$: S_1 \subseteq S_1^m \ \wedge \ S_2 \subseteq S_2^m)$$

We call this problem the S_1^m, S_2^m-problem. If we can solve it, we know a way to compute L_1^m and L_2^m of lemma 9.4 and have a way to check if DICODE is solvable. First, a property that can be derived easily.

Property 9.5

$$(\forall S_1, S_2 : L_{min} \subseteq S_1 \ \mathbf{w} \ S_2 \ \wedge \ S_1 \subseteq T_1 \ \wedge \ S_2 \subseteq T_2$$
$$: L_{min} \lceil \mathbf{a} T_1 \subseteq S_1 \ \wedge \ L_{min} \lceil \mathbf{a} T_2 \subseteq S_2)$$

This property states that we can find S_1^m and S_2^m by starting with $L_{min} \lceil \mathbf{a} T_1$ and $L_{min} \lceil \mathbf{a} T_2$ and enlarge these trace structures. Moreover, it states that:

Property 9.6 S_1^m and S_2^m do not exists if

$$L_{min} \lceil \mathbf{a} T_1 \ \mathbf{w} \ L_{min} \lceil \mathbf{a} T_2 \not\subseteq L_{max}$$

Example 9.7 Consider $\mathbf{a} T_1 = \mathbf{a} T_2$ (in that case $T_1 \ \mathbf{w} \ T_2 = T_1 \cap T_2$). Then we consider

$$S_1 = L_{max} \qquad S_2 = L_{max}$$

Notice that $S_1 = L_{max} \subseteq T_1 \ \mathbf{w} \ T_2 \subseteq T_1$ and (similar) $S_2 \subseteq T_2$, so S_1 and S_2 are possible candidates for S_1^m and S_2^m. However, they are not large enough. They can be enlarged:

$$S_1 = L_{max} \cup (T_2 \setminus T_1) \quad S_2 = L_{max} \cup (T_1 \setminus T_2)$$

We still have $S_1 \ \mathbf{w} \ S_2 = L_{max}$.

Not all traces of T_1 and T_2 have been considered yet. We did not look at $(T_1 \cap T_2) \setminus L_{max}$. Let

$$V = (T_1 \cap T_2) \setminus L_{max}$$

Notice that

$$L_{max} \cup (T_1 \setminus T_2) \cup V = T_1 \quad L_{max} \cup (T_2 \setminus T_1) \cup V = T_2$$

Furthermore, notice that

$$\begin{aligned}
& (S_1 \cup V) \ \mathbf{w} \ S_2 \\
= \ & (L_{max} \cup (T_1 \setminus T_2) \cup V) \ \mathbf{w} \ S_2 \\
= \ & T_1 \ \mathbf{w} \ S_2 \\
= \ & T_1 \cap S_2 \\
= \ & T_1 \cap (L_{max} \cup (T_2 \setminus T_1)) \\
= \ & T_1 \cap L_{max} \\
= \ & L_{max}
\end{aligned}$$

so $S_1 \cup V$ is a possible candidate for S_1^m. Also $S_2 \cup V$ is a possible candidate for S_2^m. However

$$\begin{aligned}
& (S_1 \cup V) \ \mathbf{w} \ (S_2 \cup V) \\
= \\
& T_1 \ \mathbf{w} \ T_2 \\
\not\subseteq \\
& L_{max}
\end{aligned}$$

All traces in V can be put in S_1, or in S_2, or partly in S_1 and partly in S_2, as long as $S_1 \cap V \neq S_2 \cap V$ (i.e., no trace in V may be put in S_1 and S_2).

We see that the S_1^m, S_2^m-problem can only be solved if $\mathbf{t} V = \varnothing$, which means that $L_{max} = T_1 \ \mathbf{w} \ T_2$ and $S_1^m = L_{max}$ and $S_2^m = L_{max}$ (the trivial case).

(end of example)

9.7   The S_1^m, S_2^m-subproblem

From example 9.7 it is clear that in general the S_1^m, S_2^m-problem is unsolvable. Returning to DICODE, we see that lemma 9.4 is of little use. Therefore, we restrict our attention to finding S_1^m and S_2^m as large as possible without having conflicts as in the above example and finding a formula for the remaining traces that may be put in S_1^m or in S_2^m, but not in both. We will refer to this problem as the S_1^m, S_2^m-subproblem.

We restrict the problem to:

(a) $L_{min} \lceil \mathbf{a} T_1 \ \mathbf{w} \ L_{min} \lceil \mathbf{a} T_2 \subseteq L_{max}$
(b) $\mathbf{t} L_{min} \neq \varnothing$
(c) $L_{max} \subset T_1 \ \mathbf{w} \ T_2$

If (a) or (b) are not fulfilled, then no solution is possible and if (c) is not fulfilled, then a trivial solution is possible.

According to property 9.5 we start with

$$S_1 = L_{min} \lceil \mathbf{a} T_1 \qquad S_2 = L_{min} \lceil \mathbf{a} T_2$$

We may add to S_1 and S_2 all traces that do not contribute to the weave, i.e.,

$$S_1 = L_{min} \lceil \mathbf{a} T_1 \cup R_1 \quad S_2 = L_{min} \lceil \mathbf{a} T_2 \cup R_2$$

with

$$R_1 = T_1 \setminus (T_1 \ \mathbf{w} \ T_2) \lceil \mathbf{a} T_1 \quad R_2 = T_2 \setminus (T_1 \ \mathbf{w} \ T_2) \lceil \mathbf{a} T_2$$

We have

$$S_1 \mathbf{w} \, S_2$$

$$=$$

$$(L_{min}\lceil\mathbf{a}T_1 \mathbf{w} \, L_{min}\lceil\mathbf{a}T_2) \cup (L_{min}\lceil\mathbf{a}T_1 \mathbf{w} \, R_2) \cup (L_{min}\lceil\mathbf{a}T_2 \mathbf{w} \, R_1) \cup R_1 \mathbf{w} \, R_2$$

$$= \quad [\text{ see below }]$$

$$L_{min}\lceil\mathbf{a}T_1 \mathbf{w} \, L_{min}\lceil\mathbf{a}T_2$$

$$\subseteq \quad [\text{ restriction (a) }]$$

$$L_{max}$$

Notice that

$$\mathbf{t}R_i$$

$$=$$

$$\{x : x \in \mathbf{t}(T_i \setminus (T_1 \mathbf{w} \, T_2)\lceil\mathbf{a}T_i) : x\}$$

$$=$$

$$\{x : x \in \mathbf{t}T_i \wedge x \notin \mathbf{t}(T_1 \mathbf{w} \, T_2)\lceil\mathbf{a}T_i : x\}$$

$$=$$

$$\{x : x \in \mathbf{t}T_i \wedge (\forall y : y \in \mathbf{t}(T_1 \mathbf{w} \, T_2) : y\lceil\mathbf{a}T_i \neq x) : x\}$$

from which we derive

$$L_{min}\lceil\mathbf{a}T_1 \mathbf{w} \, R_1 = \langle\emptyset, \mathbf{a}T_1 \cup \mathbf{a}T_2\rangle$$
$$L_{min}\lceil\mathbf{a}T_2 \mathbf{w} \, R_1 = \langle\emptyset, \mathbf{a}T_1 \cup \mathbf{a}T_2\rangle$$
$$R_1 \mathbf{w} \, R_2 = \langle\emptyset, \mathbf{a}T_1 \cup \mathbf{a}T_2\rangle$$

We will now consider the remaining traces in T_1 and T_2 given by:

$$V_1 = (T_1 \mathbf{w} \, T_2)\lceil\mathbf{a}T_1 \setminus L_{min}\lceil\mathbf{a}T_1 \quad V_2 = (T_1 \mathbf{w} \, T_2)\lceil\mathbf{a}T_2 \setminus L_{min}\lceil\mathbf{a}T_2$$

Notice that $T_i = S_i \cup V_i = L_{min}\lceil\mathbf{a}T_i \cup R_i \cup V_i$ (for $i = 1, 2$). First, we prove that we cannot use

$$U_1 = ((V_1 \mathbf{w} \, S_2) \setminus L_{max})\lceil\mathbf{a}T_1 \quad U_2 = ((V_2 \mathbf{w} \, S_1) \setminus L_{max})\lceil\mathbf{a}T_2$$

to extend S_1 and S_2:

$$x \in \mathbf{t}U_1$$

$$\Leftrightarrow$$

$$x \in \mathbf{t}((V_1 \mathbf{w} \, S_2) \setminus L_{max})\lceil\mathbf{a}T_1$$

$$\Leftrightarrow$$

$$(\exists y : y \in \mathbf{t}(V_1 \mathbf{w} \, S_2) \wedge y \notin \mathbf{t}L_{max} : y\lceil\mathbf{a}T_1 = x)$$

$$\Leftrightarrow$$

$$(\exists y : y \in \mathbf{t}((V_1 \mathbf{w} \, S_2) \wedge y\lceil\mathbf{a}T_1 = x : y \notin \mathbf{t}L_{max})$$

Such an x may cause problems in the weave and leads to traces outside L_{max}.

Next, we prove that we can use

$$W_1 = V_1 \setminus U_1$$

to extend S_1, or

$$W_2 = V_2 \setminus U_2$$

to extend S_2 as long as we do not use both extensions at the same time.

$$x \in \mathbf{t}W_1$$
$$\Leftrightarrow$$
$$x \in \mathbf{t}V_1 \ \wedge \ x \notin \mathbf{t}U_1$$
$$\Leftrightarrow \quad [\text{ see above }]$$
$$x \in \mathbf{t}V_1 \ \wedge \ (\forall y : y \in \mathbf{t}(V_1 \ \mathbf{w} \ S_2) \ \wedge \ y \lceil \mathbf{a}T_1 = x : y \in \mathbf{t}L_{max})$$

Such an x will not cause problems when we weave W_1 with S_2 (we do not get results outside L_{max}).

However, as will be seen in the next example, in general we have

$$W_1 \ \mathbf{w} \ W_2 \nsubseteq L_{max}$$

Example 9.8 Consider

$$T_1 = \langle(a|b|ab|bb), \{a, b\}\rangle$$
$$T_2 = \langle(b|c|bc), \{b, c\}\rangle$$
$$L_{min} = \langle(bc), \{a, b, c\}\rangle$$
$$L_{max} = \langle(abc|bc), \{a, b, c\}\rangle$$

According to the previous theory we have:

$$
\begin{array}{lcl}
L_{min}\lceil \mathbf{a}T_1 & = & \langle(b), \{a, b\}\rangle \\
L_{min}\lceil \mathbf{a}T_2 & = & \langle(bc), \{b, c\}\rangle \\
T_1 \ \mathbf{w} \ T_2 & = & \langle(b|ab|ac|ca|bc|abc), \{a, b, c\}\rangle \\
(T_1 \ \mathbf{w} \ T_2)\lceil \mathbf{a}T_1 & = & \langle(a|b|ab), \{a, b\}\rangle \\
(T_1 \ \mathbf{w} \ T_2)\lceil \mathbf{a}T_2 & = & \langle(b|c|bc), \{b, c\}\rangle
\end{array}
$$

which leads to:

$$
\begin{array}{lcllcl}
S_1 & = & L_{min}\lceil \mathbf{a}T_1 \cup (T_1 \setminus (T_1 \ \mathbf{w} \ T_2)\lceil \mathbf{a}T_1) & = & \langle(b|bb), \{a, b\}\rangle \\
S_2 & = & L_{min}\lceil \mathbf{a}T_2 \cup (T_1 \setminus (T_1 \ \mathbf{w} \ T_2)\lceil \mathbf{a}T_2) & = & \langle(bc), \{b, c\}\rangle \\
V_1 & = & (T_1 \ \mathbf{w} \ T_2)\lceil \mathbf{a}T_1 \setminus L_{min}\lceil \mathbf{a}T_1 & = & \langle(a|ab), \{a, b\}\rangle \\
V_2 & = & (T_1 \ \mathbf{w} \ T_2)\lceil \mathbf{a}T_2 \setminus L_{min}\lceil \mathbf{a}T_2 & = & \langle(b|c), \{b, c\}\rangle \\
U_1 & = & ((V_1 \ \mathbf{w} \ S_2) \setminus L_{max})\lceil \mathbf{a}T_1 & = & \langle\emptyset, \{a, b\}\rangle \\
U_2 & = & ((V_2 \ \mathbf{w} \ S_1) \setminus L_{max})\lceil \mathbf{a}T_2 & = & \langle(b), \{a, b\}\rangle \\
W_1 & = & V_1 \setminus U_1 & = & \langle(a|ab), \{a, b\}\rangle \\
W_2 & = & V_2 \setminus U_2 & = & \langle(c), \{b, c\}\rangle \\
S_1 \ \mathbf{w} \ W_2 & = & \langle\emptyset, \{a, b, c\}\rangle & & \\
S_2 \ \mathbf{w} \ W_1 & = & \langle(abc), \{a, b, c\}\rangle & & \\
W_1 \ \mathbf{w} \ W_2 & = & \langle(ac|ca), \{a, b, c\}\rangle & & \\
S_1 \ \mathbf{w} \ U_2 & = & \langle(b), \{a, b, c\}\rangle & & \\
S_2 \ \mathbf{w} \ U_1 & = & \langle\emptyset, \{a, b, c\}\rangle & &
\end{array}
$$

It is easy to see that $S_1 \ \mathbf{w} \ W_2$ and $S_2 \ \mathbf{w} \ W_1$ results in trace structures with traces inside $\mathbf{t}L_{max}$, while $W_1 \ \mathbf{w} \ W_2$ and $S_1 \ \mathbf{w} \ U_2$ result in trace structures with traces outside L_{max}. (end of example)

From this example we conclude that $T_1^m = S_1 \cup W_1$ is the best candidate for S_1^m and that $T_2^m = S_2 \cup W_2$ is the best candidate for S_2^m. The S_1^m, S_2^m-problem can only be solved if $\mathbf{t}W_1 = \emptyset$ or $\mathbf{t}W_2 = \emptyset$, in which case $T_1^m \ \mathbf{w} \ T_2^m \subseteq L_{max}$.

9.8 A solution for DICODE

We return to the original DICODE-problem. First, we introduce, for a given P_1, P_2, L_{min}, and L_{max}, some additional processes.

Definition 9.9 *Associated with DICODE we introduce (for $i = 1, 2$):*

$$\mathbf{S}_i = L_{min}\lceil \mathbf{e}P_i \cup (P_i\lceil \mathbf{e}P_i \setminus (P_1\lceil \mathbf{e}P_1 \underline{\mathbf{s}} P_2\lceil \mathbf{e}P_2)\lceil \mathbf{e}P_i)$$

called the sureties,[4]

$$\mathbf{V}_i = (P_1\lceil \mathbf{e}P_1 \underline{\mathbf{s}} P_2|\mathbf{e}P_2)|\mathbf{e}P_i \setminus L_{min}\lceil \mathbf{e}P_i$$

called the vagues,[5]

$$\mathbf{U}_i = ((\mathbf{V}_i \underline{\mathbf{s}} \mathbf{S}_{i\oplus 1}) \setminus L_{max})\lceil \mathbf{e}P_i$$

called the unusables,[6] *and*

$$\mathbf{W}_i = \mathbf{V}_i \setminus \mathbf{U}_i$$

called the warpers.[7]
($\oplus$ stands for addition modulo 2).

When there is confusion about which P_1, P_2, L_{min}, and L_{max} are involved, we write $\mathbf{S}_i(P_1, P_2, L_{min}, L_{max})$, etc.

The sureties contain those traces that surely are needed to find L_1^m and L_2^m. The vagues contain traces that may or may not be used to construct L_1^m and L_2^m. Traces that cannot be used are in the unusables, traces that may be used are in the warpers.

Notice the analogy between $\mathbf{S}_i$, $\mathbf{V}_i$, $\mathbf{U}_i$, and $\mathbf{W}_i$ from this definition with S_i, V_i, U_i, and W_i from the previous section. Using the results from the S_1^m, S_2^m-subproblem, we get:

Lemma 9.10 *The sureties and warpers associated with DICODE satisfy:*

(1) $L_{min} \subseteq \mathbf{S}_1 \underline{\mathbf{s}} \mathbf{S}_2 \subseteq L_{max}$
(2) $L_{min} \subseteq (\mathbf{S}_1 \cup \mathbf{W}_1) \underline{\mathbf{s}} \mathbf{S}_2 \subseteq L_{max}$
(3) $L_{min} \subseteq \mathbf{S}_1 \underline{\mathbf{s}} (\mathbf{S}_2 \cup \mathbf{W}_2) \subseteq L_{max}$

From which we conclude:

Theorem 9.11 *Associated with DICODE we have:*

$$L_{min}\lceil \mathbf{e}P_1 \subseteq G(P_1, \mathbf{S}_1) \ \wedge \ L_{min}\lceil \mathbf{e}P_2 \subseteq G(P_2, \mathbf{S}_2)$$
$\Rightarrow$
DICODE is solvable

---

[4]**surety** (n.) – Certainty.
[5]**vague** (adj.) – Indistinct, not clearly expressed or identified.
[6]**unusable** (adj.) – Not capable of being used.
[7]**warper** (n.) – One who has become crooked or perverted, one who is corrupt.

and

$$L_{min}\lceil eP_1 \not\subseteq G(P_1, \mathbf{S}_1 \cup \mathbf{W}_1) \;\wedge\; L_{min}\lceil eP_2 \not\subseteq G(P_2, \mathbf{S}_2 \cup \mathbf{W}_2)$$
$$\Rightarrow$$
  DICODE is not solvable

If neither one of the conditions in this theorem is met, we cannot conclude solvability or unsolvability of DICODE. In that case, we have to do some tedious hand-work and try to find $W_1' \subseteq \mathbf{W}_1$ and $W_2' \subseteq \mathbf{W}_2$ with $\mathbf{t}(W_1' \; \underline{\mathbf{s}} \; W_2') = \emptyset$ and

$$L_{min}\lceil eP_1 \subseteq G(P_1, \mathbf{S}_1 \cup W_1') \;\wedge\; L_{min}\lceil eP_2 \subseteq G(P_2, \mathbf{S}_2 \cup W_2')$$

If we have found such W_1' and W_2', we have solutions for DICODE (namely $F(P_1, \mathbf{S}_1 \cup W_1')$ and $F(P_2, \mathbf{S}_2 \cup W_2')$). No conditions can be given when such W_1' and W_2' can be found.

9.9 ABP reconsidered

If we use the above computations on ABP, we find[8] $\mathbf{S}_1 = L_{min}\lceil eP_1$, $\mathbf{S}_2 = L_{min}\lceil eP_2$, and $\mathbf{W}_1$ and $\mathbf{W}_2$ as given in table 9.2 and 9.3 (see end of this chapter).

  Computations[8] result in:

$$\mathbf{t}(\mathbf{S}_1 \; \underline{\mathbf{s}} \; \mathbf{W}_2) = \emptyset \quad \mathbf{t}(\mathbf{S}_2 \; \underline{\mathbf{s}} \; \mathbf{W}_1) = \emptyset$$

as one should expect. Furthermore, if we use W_1' and W_2' as given in table 9.4 and 9.5 (see end of this chapter), we find:

$$W_1' \subseteq \mathbf{W}_1 \quad \mathbf{t}(\mathbf{S}_1 \; \underline{\mathbf{s}} \; W_2') = \emptyset$$
$$W_2' \subseteq \mathbf{W}_2 \quad \mathbf{t}(\mathbf{S}_2 \; \underline{\mathbf{s}} \; W_1') = \emptyset$$
$$\mathbf{t}(W_1' \; \underline{\mathbf{s}} \; W_2') = \emptyset$$

so we can use

$$L_1 = \mathbf{S}_1 \cup W_1' \quad L_2 = \mathbf{S}_2 \cup W_2'$$

L_1 and L_2 are exactly the same as the processes in figure 9.5 and lead to the controllers as given in figure 9.6.

  This example verifies the results found in the last section. However, it remains unsolved how to find W_1' and W_2' in general.

9.10 Tables of results

Because the processes of the previous section are very large, we use a table in stead of a diagram to display the processes. Along the axis of the table the states can be found (numbered, and the error state omitted). Final states are displayed in boldface letters, the initial state is always 1. There is a transition from state p_1 to state p_2 labeled a if in row p_1 and column p_2 symbol a appears.

_____

 [8]Using a computer program.

Tabel 9.2: Table of $\mathbf{W}_1$

Tabel 9.3: table of $\mathbf{W}_2$

	1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	**15**	16
1	.	m	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2	.	.	e_0	e_e	.	.	.	.	.	.	.	.	.	.	.	.
3	.	.	.	.	g_0	g_e	g_1	.	.	.	.	.	.	.	.	.
4	.	$g_e g_1$	.	.	.	.	.	.	g_0	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	m	.	.	.	.	.	.
6	.	.	$e_0 e_e$	.	.	.	.	.	.	.	.	.	.	.	.	.
7	.	.	.	.	.	.	.	.	.	$e_0 e_e$	.	.	.	.	.	.
8	.	.	.	.	.	.	.	.	.	.	m	.	.	.	.	.
9	.	.	.	.	.	.	.	.	.	.	.	e_e	e_1	.	.	.
10	.	.	.	.	.	.	$g_e g_1$	g_0	.	.	.	.	.	.	.	.
11	.	.	.	.	.	.	.	.	.	.	.	.	.	$e_e e_1$	.	.
12	.	.	.	.	.	.	.	.	$g_0 g_e$	.	.	.	.	.	g_1	.
13	g_1	.	.	.	.	.	.	.	.	.	g_0	.	.	.	.	g_e
14	.	.	.	.	.	.	.	.	.	.	$g_0 g_e$	.	.	.	g_1	.
15	.	.	.	.	.	.	m	.	.	.	.	.	.	.	.	.
16	.	.	.	.	.	.	.	.	.	.	.	.	$e_e e_1$	.	.	.

Tabel 9.4: table of W_1'

	1	2	3	4	5	**6**	**7**	8	**9**	10	11	12	13	14	**15**	**16**	17	**18**
1	.	e_0	e_e	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2	.	.	.	.	n	.	.	.	.	.	.	.	.	.	.	.	.	.
3	.	.	.	.	.	$g_e g_1$	.	.	.	.	.	.	.	.	.	.	.	.
4	.	.	.	.	.	.	$g_e g_1$	.	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	$g_0 g_e$	.	.	.	.	.	.	.	.	.	.
6	.	e_0	e_e	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.
7	.	.	.	$e_e e_1$	.	.	.	.	.	e_0	.	.	.	.	.	.	.	.
8	.	.	.	.	.	.	.	.	.	.	e_0	e_e	e_1	.	.	.	.	.
9	.	.	.	.	$e_0 e_e$	.	.	.	.	.	.	.	.	e_1	.	.	.	.
10	.	.	.	.	.	.	.	.	.	.	n	.	.	.	.	.	.	.
11	.	.	.	.	.	.	.	.	.	.	.	.	.	.	$g_0 g_e$	.	.	.
12	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	$g_0 g_e$	.	.
13	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	n	.
14	.	.	.	n	.	.	.	.	.	.	.	.	.	.	.	.	.	.
15	.	.	.	.	.	.	.	.	.	.	$e_0 e_e$	.	.	e_1	.	.	.	.
16	.	.	.	.	.	.	.	.	.	.	e_0	e_e	e_1	.	.	.	.	.
17	g_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	g_e
18	.	.	.	.	.	.	.	.	.	e_0	.	.	.	.	.	.	$e_e e_1$	.

Tabel 9.5: Table of W_2'

Conclusions

Every year is getting shorter
Never seem to find the time
Plants that either come to naught
Or half a page of scribbled lines
Hanging on in quiet desperation
Is the English way
The time is gone, the song is over
Thought I'd something more to say

Time – Dark side of the moon

In this thesis we have made a succesful attempt to define discrete event systems using trace theory. To be able to control the processes, we have used extended structures, i.e., we have split the events in two kinds: exogenous events that are internal to the process and have to be controlled, and communicating events that may be common to other processes and play the role of controls.

Using trace structures makes it possible to control the order of occurrence of (exogenous) events. Restricting the processes to regular processes makes it possible to give algorithms on finite state machines to compute solutions and check whether solutions exist. From these results a computer program is developed and is used in this thesis to do the computations.

However, we cannot allow communication delay (see chapter 7), because a possible operator we should use then (the agglutinate) is very impractical.[9] In general, timing aspects are not studied and cannot easily be introduced in trace structures.

Moreover, we do not consider so called internal non-determinism as defined in [Mil], i.e., we consider all state graphs with equal accepting language to be equivalent. We consider a process to be deterministic if possible choices in exogenous events cannot be noticed outside the process, i.e., do not lead to different communication behaviour.

Deadlock and (in case we examine inputs and outputs separately) weak deadlock can easily be checked and a deadlock-free controller can easily be found (if it exist). In case we deal with repeating task processes, determinism is a sufficient condition for a process to have deadlock-free controllers.

In the last chapter we studied distributed control. We have found a necessary condition and a sufficient condition in order to find a solution for the distributed control problem. Unfortunately, it is not possible to give a single condition that is both necessary and sufficient. Moreover, it is difficult to find the necessary distributed exogenous behaviours in order to compute the controllers.

---

[9]The agglutinate is the only operator known, that describes communication delay. Perhaps other operators exists that use stronger assumptions about the delay (e.g., only finite delay) and are useful.

In the area of discrete event systems a lot of research remains to be done. We mention distributed control (chapter 9 is just a first attempt), and livelock (how can it be detected, how can it be prevented).

Referenties

[AhUl] A.V. Aho and J.D. Ullman (1972)
The theory of parsing, translation and compiling, vol. 1. parsing, vol. 2 compiling
Prentice Hall series in automatic computation

[BSW] K.A. Bartlett, R.A. Scantlebury, P.T. Wilkinson (1969)
A note on reliable full-duplex transmissions over half-duplex links
Communications of the ACM **12** (5), pp. 260–261

[CDFV] R. Cieslak, C. Desclaux, A. Fawaz and P. Varaiya (1988)
Supervisory control of discrete event processes with partial observations
IEEE trans. automat. contr. **33**, pp. 249–260

[CDQV1] G. Cohen, D. Dubois, J.P. Quadrat and M. Viot (1983)
a linear-system-theoretic view of discrete-event processes
Proc. 22nd IEEE conf. decision contr., IEEE-press, Piscataway

[CDQV2] G. Cohen, D. Dubois, J.P. Quadrat and M. Viot (1985)
a linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing
IEEE trans. automat. contr. **30**, pp. 210–220

[Dict] H.W. Fowler and F.G. Fowler (1982)
The Consise Oxford Dictionary of Current English
Seventh edition, Oxford University Press

[EWD1] E.W. Dijkstra (1971)
Hierarchical ordering of sequentiel processes
Acta Informatica **1**, pp. 115–138

[EWD2] E.W. Dijkstra (1982)
Predicate transformers (EWD835)
(lecture notes) Eindhoven university of technology

[Fish] G.S. Fishman (1978)
Principles of discrete event simulation
Wiley, New York

[Hoa] C.A.R. Hoare (1985)
Communicating sequential processes
Prentice Hall international series in computer science

[HoUl] J.E. Hopcroft and J.D. Ullman (1979)
Introduction to automata theory, languages, and computation
Addison Wesley

[InVa]   K. Inan and P. Varaiya (1988)
          Finitely recursive process models for discrete event systems
          IEEE trans. automat. contr. **33**, pp. 626–639
[JCW]   J.C. Willems (1988)
          Models for dynamicals
          report systems and control group University of Groningen
          to appear in *dynamics reported*
[JLP]   J.L. Peterson (1981)
          Petri Net theory and the modelling of systems
          Prentice-Hall
[JTU]   J.T. Udding (1984)
          Classification and composition of delay-insensitive circuits
          Ph.D.Thesis, Eindhoven University of Technology
[JvdS]   J.L.A. van der Snepscheut (1985)
          Trace theory and VLSI design
          (Lecture notes in computer science, nr. 200), Springer Verlag
[Kal]   A. Kaldewaij (1987)
          A formalism for concurrent processes
          Ph.D.Thesis, Eindhoven university of technology
[Mil]   R. Milner (1980)
          A calculus for communicating systems
          (Lecture notes in computer science, nr. 92), Springer Verlag
[Over]   R. Overwater (1987)
          Processes and interactions
          Ph.D.Thesis, Eindhoven university of technology
[Rama]   P.J. Ramadge (1983)
          Control and supervision of discrete event processes
          Ph.D. Thesis, Dept. of electl. engrg., University of Toronto
[RaWo]   P.J. Ramadge and W.M. Wonham (1987)
          Supervisory control of a class of discrete event processes
          SIAM J. on Contr. and optimalisation **25** (1), pp. 206–230
          See also: systems control group report 8515, Dept. of electl. engrg., University
          of Toronto
[Sme1]   R. Smedinga (1987)
          Control of discrete events
          SION Computer Science in the Netherlands, conference november 1987, pp.
          125–142
[Sme2]   R. Smedinga (1988)
          Using trace theory to model discrete events
          In: P. Varaiya and A.B. Kurzhanski (eds.)
          Discrete event systems: models and applications
          IIASA conference Sopron, Hungary, August 3–7,1987, pp. 81–99
          (Lecture notes in control and information science nr. 103), Springer Verlag
[Sme3]   R. Smedinga (1988)
          Simulatie en Implementatie (in Dutch)
          Addison Wesley
[StWh]   W. Strunk jr. and E.B. White (1979)
          The elements of style
          MacMillan

[ThWo]   J.G. Thistle and W.M. Wonham (1986)
         Control problems in a temporal logic framework
         International Journal on Control **44**, pp. 943–976
[Wonh]   W.M. Wonham (1979)
         Lineair multivariable control: a geometric approach
         2nd ed., Springer Verlag, New York
[WoRa]   W.M. Wonham and P.J. Ramadge (1987)
         On the supremal controllable sublanguage of a given language
         Siam J. on contr. and optimalisation **25** (3), pp. 637–659
[YCH1]   Y.C. Ho, M.A. Eyler, T.T. Chien (1983)
         A new approach to determine parameter sensitivities of transfer lines
         Management science **29**, pp. 700–714
[YCH2]   Y.C. Ho, X.Cao (1983)
         Pertubation analysis and optimization of queueing networks
         J. Opt. Th. Appl. **40**, pp. 559–582
[YCH3]   Y.C. Ho, C.G. Cassandras (1985)
         A new approach to the analysis of discrete event dynamic systems
         Automatica **19**, pp. 149–167

All operators and notation, introduced in this thesis, are explained briefly and a page number is added referring to the page where the operator or notation is defined. To avoid an overload of parentheses, the following order in binding power is used:

a c e t i o	— unary set operators
$\lceil \; \rfloor$	
w b $\underline{\text{w}}$ $\underline{\text{b}}$ $\underline{\underline{\text{w}}}$ $\underline{\underline{\text{b}}}$ s $\underline{\text{s}}$;	
$\cap \; \cup \; \backslash \; \div$	binary set operators
$\in \; \notin$	
$= \; \neq \; \subseteq \; \nsubseteq \; \supseteq \; \nsupseteq$	
$\neg$	— unary boolean operators
$\wedge \; \vee$	binary boolean operators
$\Leftrightarrow \; \nLeftrightarrow \; \Rightarrow \; \nRightarrow \; \Leftarrow \; \nLeftarrow$	

The first row has the highest binding power, the last one the least. Operators on one line have the same binding power and are evaluated from left to right. Other notation will be according to commonly accepted rules.

Trace structures

$\langle S, A \rangle$	trace structure with trace set S over alphabet A	8
tT	trace set of T	9
aT	alphabet of T	9
ϵ	empty trace	9
$x \lceil A$	trace x restricted to alphabet A	10
$T \lceil A$	trace structure T restricted to alphabet A	10
$T \textbf{ w } S$	weaving of T and S	11
$T \textbf{ b } S$	blending of T and S	11
$T \subseteq S$	ordering: T is at most S	13
$T \cup S$	T or S	14
$T \cap S$	T and S	14
$T \backslash S$	T without behaviour of S	14
pref(S)	prefix closure of a trace set S	17
pref(T)	prefix closure of a trace structure T	17
$|x|$	length of a string	24

State graphs

$x \, \mathbf{E} \, y$	equivalence relation on traces	17
$[x]_T$	equivalence class (state) of structure T	17
$\mathbf{E}(T)$	states of T	17
$[\epsilon]_T$	initial state of T	17
$[\emptyset]_T$	error state induced by T	18
$\mathbf{F}(T)$	final states of T	18
$\delta(p,a)$	state transition function: resulting state if in state p communication a occurs	18
(A,Q,d,q_0,F)	state graph with alphabet A, states Q, transition function d, initial state q_0, and final states F	19
$(A,Q,d,q_0,F)_{nd}$	non-deterministic state graph with alphabet A, states Q, transition map d, initial state q_0, and final states F	47
$\mathbf{sg}(T)$	state graph of trace structure T	19
$\mathbf{ts}(M)$	trace structure corresponding with state graph M	19
$\mathbf{det}(M)$	The deterministic equivalent of M	47
$M_1 \, \mathbf{w} \, M_2$	weaving of M_1 and M_2	47
$M_1 \, \mathbf{b} \, M_2$	blending of M_1 and M_2	47
$M_1 \cup M_2$	union of M_1 and M_2	47
$M_1 \cap M_2$	intersection of M_1 and M_2	47
$M_1 \setminus M_2$	difference of M_1 and M_2	47
$\neg M$	complement of M	47
$\mathbf{pref}(M)$	prefix closure of M	47
$M \lceil A$	restriction of M to alphabet A	48
$A(p)$	active set of state p	92

Regular traces

xy	concatenation of x and y	21
$x \vert y$	union of x and y	21
x^*	repetition of x	21
x^+	non-empty repetition	21
x,y	weaving of x and y	21
$\mathbf{ts}(x)$	trace structure corresponding with x	21

Discrete processes

$P = \langle S, E, C \rangle$	discrete process with behaviour S, exogenous events E, and communication events E	23
$\mathbf{t}P$	behaviour of P	23
$\mathbf{a}P$	event set of P	23
$\mathbf{e}P$	exogenous events of P	23
$\mathbf{c}P$	communication events of P	23
$\mathbf{ts}(P)$	trace structure corresponding to P	23
$P \, \mathbf{b} \, R$	blending on discrete processes	23
$P \, \mathbf{w} \, R$	weaving on discrete processes	23
$P \, \underline{\mathbf{b}} \, R$	connection of P and R	25
$P \, \underline{\mathbf{w}} \, R$	total connection of P and R	25

IOEDPs

$\langle S, E, C_i, C_o \rangle$	discrete process with behaviour S, exogenous events E, inputs I, and outputs O	85
$\mathbf{t}P$	behaviour of P	85
$\mathbf{i}P$	inputs of P	85
$\mathbf{o}P$	outputs of P	85
$\mathbf{e}P$	exogenous (input and output) events of P	85
$\mathbf{c}P$	communication events of P	85
$\mathbf{a}P$	all events of P	85
$\mathbf{dp}(P)$	discrete process corresponding with P	85
$\mathbf{ts}(P)$	trace structure corresponding with P	85
$P \; \underline{\mathbf{b}} \; R$	connection of P and R	86
$P \; \underline{\underline{\mathbf{w}}} \; R$	total connection of P and R	87
$b!$	sending event b, i.e., an output event	88
$b?$	receiving event b, i.e., an input event	88
$\mathbf{del}(b, c)$	class of trace structures with, at any time in the behaviour, at least as many c's as b's	88
$H(A)$	an IOEDP denoting delayed transmission of elements of A via a transmission line with infinite capacity	89
$P?!(A_i, A_o)$	P with symbols a in A_i replaced by $a?$ and symbols b in A_o replaced by $b!$	89
$P \; \mathbf{g} \; R$	agglutinate of P and R	89

Determinism

$\mathbf{rest}(P, t)$	remaining part of a prefix of P	96
$\mathbf{deterministic}(P)$	P is deterministic	96
$\mathbf{choicestates}(P)$	states of P with choice in possible events	97
$\mathbf{exostates}(P)$	states of P where an exogenous event is able to occur	97
$\mathbf{dangersets}(P)$	set of pairs (p, A) with p a choice state as well as an exo state and A the set of all events that are possible in p	97

Distributed control

$\mathbf{S}_i$	Surety, associated with DICODE	112
$\mathbf{V}_i$	Vague, associated with DICODE	112
$\mathbf{U}_i$	Unusable, associated with DICODE	112
$\mathbf{W}_i$	warper, associated with DICODE	112

Index

ABP, 101
ABP-problem, 105
accepting, 19
active set, 92
after, 53
agglutinate, 88
alphabet, 8
alphabet restriction, 10
alternating bit protocol, 101, 103
associated repeating task process, 71

behaviour, 8, 9
    legal, 63
    minimal acceptable, 63
blend, 11
    on discrete processes, 23

CCS, 95
clean friend, 77
closed loop supervised process, 61
CODE, 29
$CODE^e$, 42
communication behaviour, 23
communication event, 22
communication input event, 84
communication output event, 84
complete state graph, 46
completed task, 24
concatenation, 28
connection, 25
    of IOEDPs, 86
continuous blend, 26
continuous weave, 26
control of discrete events, 29
control pattern, 60
controlled events, 60

controlled sequential process, 60
cooperating event, 101

deadlock, 65, 94
    strong, 90
    weak, 90
deadlock-ending communication process,
    77
deadlock-ending process, 74
deadlock-ending trace set, 74
deadlock-free controller, 99
deadlock-free control, 73
deadlock recognizer, 68, 92
deadlock state, 68
deCODEr, 29, 33
determinism, 94
    in CCS, 94
DFCODE, 73, 102
dining philosophers, 80
disable event, 60
discrete process, 23
discrete process with inputs and outputs,
    84
discrete process, with inputs, outputs,
    and exogenous events, 85
distributed control of discrete events, 102
dynamical system, 9, 40

ECODE, 57
empty process, 24
enable event, 60
end state, 68
ending trace, 65
endogenous event, 22
environment, 94
error state, 18, 46

126

Samenvatting

Systemen met discrete gebeurtenissen spelen in vele gebieden een rol. In dit proefschrift staat de volgorde van gebeurtenissen centraal en worden tijdsaspecten buiten beschouwing gelaten. In dat geval kunnen systemen met discrete gebeurtenissen goed worden gemodelleerd door gebruik te maken van trace theorie.

Zo'n systeem kan dan worden gemodelleerd als een twee-tuple met als elementen het alfabet (een eindige verzameling symbolen die de gebeurtenissen voorstellen) en een gedrag (een mogelijk oneindige verzameling van eindige rijtjes van symbolen uit het alfabet). Een string (trace) uit het gedrag representeert een mogelijk gedrag: de opvolging van de symbolen geeft de opvolging van de gebeurtenissen aan.

In dit proefschrift staat het regelen van systemen met discrete gebeurtenissen centraal. Daartoe worden de gebeurtenissen opgedeeld in tweeën: gebeurtenissen, die door de eigen dynamica van een systeem plaatsvinden en niet direct van buiten te beïnvloeden zijn: de exogene gebeurtenissen, en gebeurtenissen, die gemeenschappelijk zijn met de omgeving: de communicatie gebeurtenissen.

Het regelen van zo'n systeem is nu het gebruiken van de communicatie gebeurtenissen (door het aankoppelen van een tweede systeem, het regelsysteem) om de exogene gebeurtenissen te regelen, dat wil zeggen het exogene gedrag dient zich binnen een tweetal limieten te bevinden: er mag niet meer mogelijk zijn dat een maximum gedrag en er moet een minimaal gedrag mogelijk zijn. Een dergelijk regelprobleem wordt CODE genoemd: Control Of Discrete Events.

Een groot deel van dit proefschrift bestaat uit het vinden van een oplossing voor CODE en het geven van nodige en voldoende voorwaarden waaronder zo'n oplossing bestaat. Daarnaast worden een aantal gerelateerde problemen besproken, waaronder het regulatie probleem: het gedrag van het systeem dient zich "na enige stappen" tussen de beide limieten te bevinden en het uitgebreide regelprobleem waarbij ook de communicatie gebeurtenissen in het gewenste eindgedrag worden meegenomen.

Het gezamelijke gedrag van twee (of meer) systemen met discrete gebeurtenissen (we spreken dan over de connectie van die systemen) kan eindigen in deadlock: geen der systemen is in staat een gebeurtenis te doen plaatsvinden overeenkomstig de regels zoals die gelden bij connectie. In dit proefschrift worden middelen aangedragen om deze deadlock te detecteren en om regelaars te vinden voor CODE die geen deadlock veroorzaken. Middels een voorbeeld (van de etende filosofen) wordt aangegeven hoe in een connectie van een aantal systemen een extra systeem mogelijke deadlock kan opheffen.

De exogene gebeurtenissen van een systeem kunnen leiden tot een intern gedrag, dat

door de omgeving niet is waar te nemen. Indien ook het uitwendige gedrag (het communicatie gedrag) hierdoor kan verschillen spreken we van niet-determinisme. Mogelijke vormen van niet-determinisme worden besproken.

Tot slot is een hoofdstuk gewijd aan het gedistribueerd regelen van systemen, dat wil zeggen dat een aantal deel-regelaars, elk werkend op een deel-systeem, gezamelijk een gewenst totaal gedrag teweeg brengen. Met het alternating bit protocol als voorbeeld wordt aangegeven hoe in dat geval de deel-regelaars zijn te vinden.

Curriculum vitae

He has laughted and he has cried
He has fought and he has died
He's just the same as all the rest
He's not the worst, he's not the best

Yet another movie – A momentary lapse of reason

Rein Smedinga werd geboren op 9 november 1957 in Harlingen. Na de lagere school volgde hij de VWO-opleiding aan de Rijksscholengemeenschap Simon Vestdijk te Harlingen en haalde in 1976 zijn diploma Atheneum-B.

Daarna volgde de studie wiskunde aan de Rijksuniversiteit te Groningen. In 1979 behaalde hij daar zijn kandidaatsexamen, waarna hij zich specialiseerde in de systeem theory en in januari 1983 cum laude afstudeerde op het afstudeerverslag "Ontwerp van een computer programma voor het oplossen van storingsontkoppelingsproblemen."

In april 1983 werd hij lid van de vakgroep informatica aan de Rijksuniversiteit te Groningen, waar hij o.a. onderzoek deed op het gebied van simulatie, hetgeen resulteerde in het boek *Simulatie en Implementatie* (zie [Sme3]), en discrete event systems, hetgeen resulteerde in dit proefschrift.

	1	2	3	4	5	6	**7**	8	9	**10**	11	12	**13**	14	**15**	16	17	18	19	20	**21**	22	23	**24**	25	**26**	**27**	28	**29**
1	.	m	.	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	e_0	.	.	.	e_e	.	.	.	.	.	.
2	.	.	e_0	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	e_e	.
3	.	.	.	g_0	.	.	g_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	g_e	.	.
4	.	.	.	.	m	e_0	.	.	.	.	.	e_e	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	e_0	.	.	e_e	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	e_1	.	.	.	.
6	.	.	.	.	.	.	$g_0 g_e g_1$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
7	.	.	.	.	.	$e_0 e_e e_1$	.	m	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
8	.	.	.	.	.	$e_0 e_e e_1$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
9	.	.	.	.	.	.	g_1	.	.	$g_0 g_e$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
10	.	.	.	.	.	e_0	.	.	e_e	.	m	.	.	.	.	.	.	.	.	.	.	.	.	.	e_1	.	.	.	.
11	.	.	.	.	.	e_0	.	.	.	.	.	e_e	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
12	.	.	.	.	.	.	g_1	.	.	.	.	.	$g_0 g_e$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
13	.	.	.	.	.	e_0	.	.	.	.	m	e_e	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
14	.	.	.	.	.	.	g_0	.	.	.	.	.	.	.	.	g_e	.	g_1	.	.	.	.	.	.	.	.	.	.	.
15	.	.	.	.	.	e_0	.	.	.	.	.	.	.	.	$e_e e_1$	.	m	.	.	.	.	.	.	.	.	.	.	.	.
16	.	.	.	.	.	e_0	.	.	.	.	.	.	.	.	$e_e e_1$	.	.	.	.	.	.	.	.	.	.	.	.	.	.
17	.	.	.	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	m	e_0	.	.	.	e_e	.	.	.	.	.	.
18	.	.	.	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	e_0	.	.	.	e_e	.	.	.	.	.	.
19	.	.	.	.	.	.	g_1	.	.	.	.	.	.	.	.	.	.	.	.	$g_0 g_e$	.	.	.	.	.	.	.	.	.
20	.	.	.	.	.	e_0	.	.	.	.	m	e_e	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
21	.	.	.	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	$e_0 e_e$	.	.	m	.	.	.	.	.
22	.	.	.	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	$e_0 e_e$	.	.	.	.	.	.	.	.
23	.	.	.	.	.	.	g_0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	$g_e g_1$	.	.	.	.	.
24	.	.	.	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	m	e_0	.	.	.	e_e	.	.	.	.	.	.
25	g_1	.	.	.	.	.	g_0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	g_e	.	.	.
26	.	.	.	.	.	e_0	.	.	.	.	.	.	.	.	.	.	m	.	.	.	.	.	.	$e_e e_1$	.	.	.	.	.
27	.	.	$e_0 e_e$	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	m	.	.	.	.	.	.	.
28	.	.	.	.	.	.	g_0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	$g_e g_1$	
29	.	.	e_0	.	.	e_1	.	.	.	.	.	.	.	.	.	.	.	m	.	.	.	.	.	.	.	.	.	e_e	.

	1	2	3	4	5	6	**7**	8	9	**10**	11	12	13	**14**	15	16	17	18	**19**	20	21	**22**	**23**	24	25	**26**	27	**28**	29	30	**31**
1		e_0			e_1																									e_e	
2			n				g_1										g_0	g_e													
3				g_0			g_1																						g_e		
4					e_0			e_e															e_1								
5						n	$g_0g_eg_1$																								
6							$g_0g_eg_1$																								
7					$e_0e_ee_1$																										
8							g_1		n													g_0g_e									
9							g_1			g_0g_e																					
10					e_0						e_c	e_1																			
11							g_1		n	g_0g_e																					
12							g_0						n	g_e	g_1																
13							g_0							g_e	g_1																
14					e_0							e_e	e_1																		
15					e_1											e_0				e_e											
16							g_1										n	g_0	g_e												
17							g_1											g_0	g_e												
18					e_0						e_e	e_1																			
19					e_1											e_0	e_e														
20							g_0														n	g_e	g_1								
21							g_0															g_e	g_1								
22					e_1											e_0				e_e											
23					e_0			e_e															e_1								
24							g_0							g_e	g_1									n							
25	g_1						g_0																			g_e					
26					e_0																					e_e	e_1				
27	g_1						g_0						n													g_e					
28					e_1																							e_0e_e			
29				g_0			g_1										n											g_e			
30							g_0														n									$g_e g_1$	
31		e_0			e_1																									e_e	