Receptive process theory

Receptive Process Theory

by

Mark B. Josephs

90/8

October, 1990

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Receptive Process Theory

Mark B. Josephs*

Department of Mathematics and Computing Science
Eindhoven University of Technology

August 22, 1990

Summary. An algebraic theory of receptive processes is presented. A receptive process models the interaction by input and output between a system and its environment. Input from the environment and output to the environment are never blocked; but if a system is not ready to receive a particular input, its subsequent behaviour is undefined.

In essence, this paper reworks Hoare's theory of Communicating Sequential Processes under the above assumption about communication. The resulting model is more attractive than the failures-divergences model of CSP because the refusal sets of the latter are simplified out of existence. Like CSP, receptive process theory is equipped with a sound and complete set of algebraic laws.

Applications of the theory include the design of asynchronous circuits and the study of data flow networks. As an example, this paper verifies algebraically the design of a Muller C-element from a majority-element.

1 Introduction

A receptive process models the interaction by input and output between a system and its environment. Input from its environment is never blocked by a system. Symmetrically, output from a system is never blocked by its environment. If a system is not ready to receive a particular input, the subsequent behaviour of the system is undefined. It is to be understood that the environment is obliged not to send such an input in these circumstances.

A theory of receptive processes is concerned therefore with a very general communication paradigm, one which is applicable to asynchronous circuits and data flow networks, for example. Even synchronized communication, as modelled in CSP [6, 7], can be implemented by a handshake of inputs and outputs between receptive processes.

*Author's current address: Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, U.K.

In this paper, we develop an algebraic theory based on a mathematical model similar to Dill's model of speed-independent circuits [3]. Indeed, we have borrowed the term "receptive" from him. Dill has been more concerned, however, with automatic verification than with process algebra. Other algebraic theories of concurrency such as CCS [13, 14] and CSP [2, 7] do not make more than syntactic distinctions between inputs and outputs. Instead they are concerned with undirected synchronization events or actions. Inputs and outputs are sometimes distinguished in trace theory [15, 17], but there the emphasis is on deterministic behaviour.

In essence, we rework Hoare's CSP under the assumption that processes are receptive. The resulting model is more attractive than the failures-divergences model of CSP because the refusal sets of the latter are simplified out of existence, and yet nondeterministic behaviour can still be fully expressed. The divergences of CSP remain as an extremely useful way of capturing obligations to be met by the environment, i.e., that certain inputs will not be sent in certain circumstances. Like CSP, receptive process theory is equipped with a sound and complete set of algebraic laws.

Receptive process theory and CSP are alike in another respect: they do not deal with fairness. This is in many ways an advantage because it facilitates the algebraic transformation of networks of processes. They may be implemented under a variety of scheduling strategies, including sequential execution on a single processor and fine-grained concurrent execution in VLSI. Both theories treat the possibility of infinite chatter (which in receptive process theory includes outputting forever without requiring input) as wholly undesirable. This restricts us somewhat when it comes to modelling asynchronous circuits, e.g., a ring oscillator [3] would be outside the scope of our theory.

In the remainder of this paper, we introduce a mathematical model for receptive processes and develop a process algebra by defining a number of CSP-like operators. We show by means of a small example that the algebra can be used in the verification of asynchronous circuits. Finally, we briefly examine the special cases of data flow networks and delay-insensitive circuits.

2 The Model

In this section, we define a receptive process to be a triple (I, O, F) which must satisfy certain conditions.

Consider a system that interacts by input and output with its environment. The set of all possible inputs from the environment to the system is represented by the input alphabet of the process. The set of all possible outputs from the system to the environment is represented by the output alphabet of the process. In the remainder of this section, we shall consider a particular input alphabet I and a particular output alphabet O. We insist that I and O are disjoint and that O is non-empty.

Suppose that the system has engaged in a finite sequence s of inputs and outputs. If the environment were to provide no further input to the system, then the system would continue to output either forever or until it became quiescent, i.e., it required further input.

In the latter case, the system would refuse to output after engaging in some finite sequence t of outputs; we call the sequence st a failure of the process.

Systems that can output forever or become quiescent in infinitely many different ways are modelled as divergent. Let $F \subseteq (I \cup O)^*$ be the set of failures of a particular process. Then the set $F\uparrow$ of divergences of the process is defined by

$$F\uparrow \;=\; \{s | \{t \in O^* | st \in F\} \text{ is infinite}\}.$$

Immediate consequences of this definition are that $F\uparrow$ is closed under curtailment of outputs, and $\uparrow$ distributes through finite unions (and so is monotonic with respect to set containment).

Lemma 1 $st \in F\uparrow \wedge\; t \in O^* \Rightarrow s \in F\uparrow$. $\square$

Lemma 2 $(F_0 \cup F_1)\uparrow = (F_0\uparrow) \cup (F_1\uparrow)$. $\square$

As in CSP, divergence is considered wholly undesirable and so it is convenient to assume that a divergent process can do or fail to do anything whatsoever. This is reflected in the following two closure condition that we impose upon F.

$$s \in F\uparrow \;\Rightarrow\; st \in F\uparrow \tag{1}$$

$$F\uparrow \subseteq F \tag{2}$$

That s is a divergence can be interpreted as meaning that the environment guarantees not to engage in s. The divergences of a process now have a simpler characterization.

Lemma 3 $s \in F\uparrow \;\Leftrightarrow\; \forall t.\; st \in F$.
Proof. ($\Rightarrow$) follows from conditions 1 and 2. ($\Leftarrow$) follows because if $st \in F$ holds for every t, then it certainly holds for all $t \in O^*$; since O is non-empty, the set of such t is infinite and so, by definition, $s \in F\uparrow$. $\square$

A further property of $\uparrow$ is that it distributes through arbitrary intersections of failure sets (and so is $\cap$-continuous).

Lemma 4 $(\bigcap X)\uparrow = \bigcap_{F \in X}(F\uparrow)$.
Proof. ($\subseteq$) $\bigcap X \subseteq F$, for all $F \in X$, and so, by monotonicity of $\uparrow$, $(\bigcap X)\uparrow \subseteq \bigcap_{F \in X}(F\uparrow)$. ($\supseteq$) Suppose $s \in \bigcap_{F \in X}(F\uparrow)$. Then, by Lemma 3, $st \in F$, for all $F \in X$ and all t. Since O is non-empty, it follows that $\{t \in O^* | st \in \bigcap X\}$ is infinite, i.e., $s \in (\bigcap X)\uparrow$. $\square$

Note that if one were to allow empty output alphabets, the set of divergences would have to be modelled explicitly, as in [8].

The set $\widehat{F}$ of traces of the process can also be derived from F.

$$\widehat{F} \;=\; \{s | \exists t \in O^*.\; st \in F\}.$$

Immediate consequences are that every failure is a trace and that $\widehat{\ }$ distributes through arbitrary unions.

Lemma 5 $F \subseteq \widehat{F}$. □

Lemma 6 $\widehat{\bigcup X} = \bigcup_{F \in X} \widehat{F}$. □

The remaining closure conditions on F can be most easily stated as conditions on $\widehat{F}$, namely, $\widehat{F}$ is non-empty, prefix-closed and closed under extension by inputs.

$$\varepsilon \in \widehat{F} \tag{3}$$

$$st \in \widehat{F} \;\Rightarrow\; s \in \widehat{F} \tag{4}$$

$$s \in \widehat{F} \wedge t \in I^* \;\Rightarrow\; st \in \widehat{F} \tag{5}$$

The last condition, called receptiveness by Dill, arises because the environment might send input to the system at any time. The traces of a process now have a simpler characterization.

Lemma 7 $s \in \widehat{F} \;\Leftrightarrow\; \exists t.\; st \in F$.
Proof. ($\Rightarrow$) follows from the definition of $\widehat{F}$. ($\Leftarrow$) follows because if $st \in F$, then $st \in \widehat{F}$ and so, by condition 4, $s \in \widehat{F}$. □

Although $\widehat{\ }$ does not in general distribute through intersections of failure sets, it is nevertheless $\bigcap$-continuous, which follows from our treatment of infinite nondeterminism as divergent behaviour.

Lemma 8 For any chain of failure sets such that $F_i \supseteq F_{i+1}$, $i \geq 0$, $\bigcap_i \widehat{F_i} = \widehat{\bigcap_i F_i}$.
Proof. Since $\widehat{\ }$ is monotonic, we need only show that $\bigcap_i \widehat{F_i} \subseteq \widehat{\bigcap_i F_i}$. Suppose $s \in \bigcap_i \widehat{F_i}$. By the definition of $\widehat{F}$, $\forall i.\; \exists t \in O^*.\; st \in F_i$. Since $F_i \supseteq F_{i+1}$, $i \geq 0$, either $\exists t \in O^*.\; st \in \bigcap_i F_i$ or $\forall i.\; \{t \in O^* | st \in F_i\}$ is infinite. In the latter case, $\forall i.\; s \in F_i\!\uparrow$ and so by condition 2, $s \in \bigcap_i F_i$. Thus, either way, $s \in \widehat{\bigcap_i F_i}$, by the definition of $\widehat{F}$. □

We conclude this section with a theorem concerning the space of receptive processes.

Theorem 1 The failure sets form a c.p.o. under containment.
Proof. Failure sets are clearly partially ordered and have least element $(I \cup O)^*$ which satisfies conditions 1-5. To prove completeness, consider a chain of failure sets such that $F_i \supseteq F_{i+1}$, $i \geq 0$. That $\bigcap_i F_i$ satisfies conditions 1-5 follows easily from the continuity of $\uparrow$ and $\widehat{\ }$. □

3 Process Algebra

In this section, we develop a CSP-like language for expressing the behaviour of receptive processes. The process-expressions are constructed from $\bot$, nondeterministic choice, input-guarded choice, output-guarded choice, *skip*-guarded choice, concealment of output and parallel composition. Algebraic laws are provided that enable us to eliminate the last three operators from process-expressions. Additional laws are provided that enable every process-expression to be transformed into a normal form.

Here, normal form means $\bot$ or a nondeterministic choice between a finite, non-empty set X of guarded choices. X should contain at most one input-guarded choice, each output guard should be distinct and all guarded processes should themselves be in normal form, as well as being as nondeterministic as possible. (In terms of the model, if X contains an input-guarded choice, then ε is a failure of the process; the set of output guards are those outputs that the process can engage in initially.)

The language can be extended to allow (mutual) recursion. In the standard way [1, 5, 16], recursively-defined processes are semantically the limit of their finite syntactic approximations.

We now consider each operator in turn, employing the valuations i, o, t, f and d to define, respectively, the input alphabet, output alphabet, traces, failures and divergences of a process-expression.

3.1 Chaos

The process $\bot_{I,O}$ can do or fail to do anything whatsoever. It is defined by $i\bot_{I,O} = I$, $o\bot_{I,O} = O$ and $f\bot_{I,O} = (I \cup O)^*$. Often we write $\bot$ and leave the alphabets to be deduced from the context.

3.2 Nondeterministic Choice

The process $P \sqcap Q$ behaves nondeterministically like P or Q. We insist that $iP = iQ$, $oP = oQ$ and define $i(P \sqcap Q) = iP$, $o(P \sqcap Q) = oP$ and $f(P \sqcap Q) = fP \cup fQ$. That $f(P \sqcap Q)$ satisfies conditions 1-5 follows from Lemmas 2 and 6. Continuity (in each operand) follows from the fact that union distributes through intersection. Nondeterministic choice is clearly commutative, associative, idempotent and has $\bot$ as its null element.

3.3 Guarded Choice

Let P and Q_x, for all $x \in I$, be processes with the same input alphabet I and the same output alphabet O. We next define the three kinds of guarded choice, each of which has input alphabet I and output alphabet O. In each case, it is easy to see that conditions 1-5 are met and choice is $\cap$-continuous in each guarded process (because union distributes through intersection).

3.3.1 Input-Guarded Choice

The process $(?x \rightarrow Q_x)$ waits for any input x from its environment and then behaves like Q_x. Formally,

$$f(?x \rightarrow Q_x) = \{\varepsilon\} \cup \{xs | x \in I \wedge s \in fQ_x\}.$$

Lemma 9 $d(?x \rightarrow Q_x) = \{xs | x \in I \wedge s \in dQ_x\}$. □

Lemma 10 $t(?x \rightarrow Q_x) = \{\varepsilon\} \cup \{xs | x \in I \wedge s \in tQ_x\}$. □

The following distributivity law helps us transform a process into normal form.

$$(?x \rightarrow P_x) \sqcap (?y \rightarrow Q_y) = (?z \rightarrow (P_z \sqcap Q_z)).$$

3.3.2 Output-Guarded Choice

The process $(!c \rightarrow P \mid ?x \rightarrow Q_x)$ eventually outputs $c \in O$ to its environment (and behaves like P), unless its environment supplies it earlier with some input x, in which case it subsequently behaves like Q_x.

$$f(!c \rightarrow P \mid ?x \rightarrow Q_x) = \begin{cases} (I \cup O)^* & \text{if } fP = (I \cup O)^* \\ \{cs | s \in fP\} & \\ \cup\{xs | x \in I \wedge s \in fQ_x\} & \text{otherwise.} \end{cases}$$

Lemma 11 $d(!c \rightarrow P \mid ?x \rightarrow Q_x) = \{cs | s \in dP\} \cup \{xs | x \in I \wedge s \in dQ_x\}$ if $fP \neq (I \cup O)^*$. □

Lemma 12 $t(!c \rightarrow P \mid ?x \rightarrow Q_x) = \{\varepsilon\} \cup \{cs | s \in tP\} \cup \{xs | x \in I \wedge s \in tQ_x\}$ if $fP \neq (I \cup O)^*$. □

The first case of the definition is needed to back-propagate divergence through output (Lemma 1). It gives rise to the law

$$(!c \rightarrow \bot \mid ?x \rightarrow P_x) = \bot.$$

This and the following laws are necessary for normalization.

Case $P = (!c \rightarrow P' \mid ?x \rightarrow P_x)$ and $Q = (?y \rightarrow Q_y)$.

$$P \sqcap Q = (!c \rightarrow P' \mid ?z \rightarrow (P_z \sqcap Q_z)) \sqcap (?z \rightarrow (P_z \sqcap Q_z)).$$

Case P as above and $Q = (!c \rightarrow Q' \mid ?y \rightarrow Q_y)$.

$$P \sqcap Q = (!c \rightarrow (P' \sqcap Q') \mid ?z \rightarrow (P_z \sqcap Q_z)).$$

Case P as above and $Q = (!d \rightarrow Q' \mid ?y \rightarrow Q_y)$.

$$P \sqcap Q = (!c \rightarrow P' \mid ?z \rightarrow (P_z \sqcap Q_z)) \sqcap (!d \rightarrow Q' \mid ?z \rightarrow (P_z \sqcap Q_z)).$$

3.3.3 Skip-Guarded Choice

The process $(skip \rightarrow P \mid ?x \rightarrow Q_x)$ eventually chooses to behave like P, unless its environment supplies it earlier with some input x, in which case it subsequently behaves like Q_x.

$$f(skip \rightarrow P \mid ?x \rightarrow Q_x) = fP \cup \{xs | x \in I \wedge s \in fQ_x\}.$$

Lemma 13 $d(skip \rightarrow P \mid ?x \rightarrow Q_x) = dP \cup \{xs | x \in I \wedge s \in dQ_x\}$. □

Lemma 14 $t(skip \rightarrow P \mid ?x \rightarrow Q_x) = tP \cup \{xs | x \in I \wedge s \in tQ_x\}$. □

The following laws, in which we consider the various possibilities for P, enable us to eliminate *skip*-guards.

Case $P = \perp$.

$$(skip \rightarrow P \mid ?x \rightarrow Q_x) = \perp.$$

Case $P = P' \sqcap P''$.

$$\begin{aligned}(skip \rightarrow P \mid ?x \rightarrow Q_x) = \; & (skip \rightarrow P' \mid ?x \rightarrow Q_x) \\ & \sqcap (skip \rightarrow P'' \mid ?x \rightarrow Q_x).\end{aligned}$$

Case $P = (?y \rightarrow P_y)$.

$$(skip \rightarrow P \mid ?x \rightarrow Q_x) = (?z \rightarrow (P_z \sqcap Q_z)).$$

Case $P = (!c \rightarrow P' \mid ?y \rightarrow P_y)$.

$$(skip \rightarrow P \mid ?x \rightarrow Q_x) = (!c \rightarrow P' \mid ?z \rightarrow (P_z \sqcap Q_z)).$$

3.4 Concealment of Output

The process $P \backslash C$ behaves like P, except that outputs in $C \subset oP$ are concealed from its environment. Thus $i(P \backslash C) = iP$, $o(P \backslash C) = (oP) \backslash C$ and

$$f(P \backslash C) = \{s \backslash C | s \in fP\}.$$

Lemma 15 $d(P \backslash C) = \{s \backslash C | s \in dP\}$. □

Lemma 16 $t(P \backslash C) = \{s \backslash C | s \in tP\}$. □

The last lemma is easily proved; and it follows directly that conditions 1-5 are met. (Proofs of the previous lemma and that $P \backslash C$ is continuous in P can be found in the appendix.) The following laws enable us to eliminate concealment.

Case $P = \perp$.

$$P \backslash C = \perp.$$

Case $P = P' \sqcap P''$.

$$P\backslash C = (P'\backslash C) \sqcap (P''\backslash C).$$

Case $P = (?x \rightarrow P_x)$.

$$P\backslash C = (?x \rightarrow (P_x\backslash C)).$$

Case $P = (!c \rightarrow P' \mid ?x \rightarrow P_x)$.

$$P\backslash C = (\alpha \rightarrow (P'\backslash C) \mid ?x \rightarrow (P_x\backslash C))$$

where α is *skip* if $c \in C$, and $!c$ otherwise.

3.5 Parallel Composition

The process $P \parallel Q$ is the parallel composition of P and Q, which must have disjoint output alphabets. Inputs from the environment that are common to the input alphabets of both components are copied to each. Outputs from one component that are in the input alphabet of the other component are copied to that component and to the environment. Thus the output alphabet O of the parallel composition is the union of oP and oQ, and the input alphabet I is $(iP \cup iQ)\backslash O$. The definition of $f(P \parallel Q)$ is complicated by the possibility of divergence caused by infinite chatter between the two components.

First we define the weave SwT of sets $S \subseteq (iP \cup oP)^*$ and $T \subseteq (iQ \cup oQ)^*$, as in [7, 15, 17], for example. (We write $s\lceil A$ to mean the restriction of s to events in A.)

$$SwT = \{s \in (I \cup O)^* | s\lceil(iP \cup oP) \in S \wedge s\lceil(iQ \cup oQ) \in T\}.$$

Lemma 17 $(\bigcap X)wT = \bigcap_{S \in X}(SwT)$. $\square$

Infinite chatter between P and Q (which includes either process diverging) is possible after any trace in $((tP)w(tQ))\uparrow$. The divergences of $P \parallel Q$ are (the extensions of) such traces. $P \parallel Q$ can refuse to output either because both P and Q can so refuse or because of divergence.

$$f(P \parallel Q) = (fP)w(fQ) \cup \{st | s \in ((tP)w(tQ))\uparrow \wedge t \in (I \cup O)^*\}.$$

Lemma 18 $d(P \parallel Q) = \{st | s \in ((tP)w(tQ))\uparrow \wedge t \in (I \cup O)^*\}$. $\square$

Lemma 19 $t(P \parallel Q) = (tP)w(tQ) \cup d(P \parallel Q)$. $\square$

The above lemmas are easily proved and that conditions 1-5 are met follows directly. (Proof of the continuity of parallel composition can be found in the appendix.) Parallel composition can be eliminated by using the following laws and the fact that it is commutative.

Case $P = \bot$.

$$P \parallel Q = \bot.$$

Case $P = P' \sqcap P''$.

$$P \parallel Q \;=\; (P' \parallel Q) \sqcap (P'' \parallel Q).$$

Case $P = (?x \;\rightarrow\; P_x)$ and $Q = (?y \;\rightarrow\; Q_y)$.

$$P \parallel Q \;=\; (?z \;\rightarrow\; R_z) \text{ where } R_z \;=\; \begin{cases} P_z \parallel Q_z & \text{if } z \in iP \cap iQ \\ P_z \parallel Q & \text{if } z \in iP \backslash (iQ \cup oQ) \\ P \;\parallel Q_z & \text{if } z \in iQ \backslash (iP \cup oP). \end{cases}$$

Case $P = (!c \;\rightarrow\; P' \mid ?x \;\rightarrow\; P_x)$ and Q input-guarded as above.

$$P \parallel Q \;=\; (!c \;\rightarrow\; R' \mid ?z \;\rightarrow\; R_z)$$

where R_z is as above, and R' is $P' \parallel Q_c$ if $c \in iQ$, and is $P' \parallel Q$ otherwise.
Case $P = (!c \;\rightarrow\; P' \mid ?x \;\rightarrow\; P_x)$ and $Q = (d! \;\rightarrow Q' \mid ?y \;\rightarrow\; Q_y)$.

$$P \parallel Q \;=\; (!c \;\rightarrow\; R' \mid ?z \;\rightarrow\; R_z) \sqcap (!d \rightarrow\; S' \mid ?z \;\rightarrow\; R_z)$$

where R' and R_z are as above, and S' is $P_d \parallel Q'$ if $d \in iP$, and is $P \parallel Q'$ otherwise.

4 Verification of an Asynchronous Circuit

Asynchronous circuits can be designed to function correctly independent of the speed of the components in the circuit, but assuming instantaneous transmission of signals between the components. In this section, we verify a small speed-independent design using our process algebra. We begin by specifying a wire, majority-element and Muller C-element in our algebra, where input and output events denote voltage-level transitions either up or down. We then verify that the C-element can be constructed from the other two components [12].

A wire W with input alphabet $\{m\}$ and output alphabet $\{c\}$ is specified by the following mutually recursive equations, in which the variable x ranges over $\{m\}$.

$$W = (?x \;\rightarrow\; W') \text{ where } W' = (!c \;\rightarrow\; W \mid ?x \;\rightarrow\; \bot).$$

That is, a signal m is propagated as c, unless a second signal arrives too early causing interference. The divergence indicates that the environment should not send that second signal until it has received the signal c.

A majority-element M with inputs a, b and c and output m is specified by the following mutually recursive equations, in which the variable y ranges over $\{a, b, c\}$.

$$\begin{array}{lcl} M & = & (?y \;\rightarrow\; M_{\{y\}}) \\ M_{\{a\}} & = & (?y \;\rightarrow\; (M \textbf{ if } y = a \textbf{ else } M_{\{a,y\}})) \\ M_{\{a,b\}} & = & (!m \;\rightarrow\; M_{\{c\}} \mid ?y \;\rightarrow\; (M_{\{a,b,c\}} \textbf{ if } y = c \textbf{ else } \bot)) \\ M_{\{a,b,c\}} & = & (!m \;\rightarrow\; M \mid ?y \;\rightarrow\; M_{\{a,b,c\} \backslash \{y\}}). \end{array}$$

The behaviour of M is symmetric in its inputs. In state M_S all inputs in S are at one voltage level, and the other inputs and m are at the other voltage level. Note that once inputs on a and b have been received, it is safe for a second input on a to arrive after an input on c, but not before. This is because of the danger of an output spike should a change back early.

A C-element with inputs a and b and output c is specified by

$$C = (?z \rightarrow C_{\{z\}}) \text{ where } \begin{aligned} C_{\{a\}} &= (?z \rightarrow (C \textbf{ if } z = a \textbf{ else } C_{\{a,b\}})) \\ C_{\{a,b\}} &= (!c \rightarrow C \mid ?z \rightarrow \bot). \end{aligned}$$

The behaviour of C is symmetric in its two inputs, and the variable z ranges over $\{a, b\}$.

We now prove by straightforward algebraic manipulation that

$$C = (W \parallel M)\backslash\{m\}.$$

Our first step is to consider $W \parallel M$, which has input alphabet $\{a, b\}$ and output alphabet $\{c, m\}$. Note that if both the wire and the majority-element are waiting for input, then all that can happen is an input a or b by the latter, which does not affect the state of the wire.

$$\begin{aligned} & W \parallel M \\ = \; & (?x \rightarrow W') \parallel (?y \rightarrow M_{\{y\}}) \\ = \; & (?z \rightarrow (W \parallel M_{\{z\}})) \end{aligned}$$

$$\begin{aligned} & W \parallel M_{\{a\}} \\ = \; & (?x \rightarrow W') \parallel (?y \rightarrow (M \textbf{ if } y = a \textbf{ else } M_{\{a,y\}})) \\ = \; & (?z \rightarrow ((W \parallel M) \textbf{ if } z = a \textbf{ else } (W \parallel M_{\{a,b\}}))) \end{aligned}$$

$$\begin{aligned} & W \parallel M_{\{a,b\}} \\ = \; & (?x \rightarrow W') \parallel (!m \rightarrow M_{\{c\}} \mid ?y \rightarrow (M_{\{a,b,c\}} \textbf{ if } y = c \textbf{ else } \bot)) \\ = \; & (!m \rightarrow (W' \parallel M_{\{c\}}) \mid ?z \rightarrow \bot) \end{aligned}$$

$$\begin{aligned} & W' \parallel M_{\{c\}} \\ = \; & (!c \rightarrow W \mid ?x \rightarrow \bot) \parallel (?y \rightarrow (M \textbf{ if } y = c \textbf{ else } M_{\{c,y\}})) \\ = \; & (!c \rightarrow (W \parallel M) \mid ?z \rightarrow (W' \parallel M_{\{c,z\}})) \end{aligned}$$

($W' \parallel M_{\{a,c\}}$ simplifies to $\bot$, but it turns out that we do not need to know this.)
Our second step is to conceal output m.

$$
\begin{aligned}
& (W \parallel M)\backslash\{m\} \\
= \; & (?z \;\rightarrow\; (W \parallel M_{\{z\}}))\backslash\{m\} \\
= \; & (?z \;\rightarrow\; ((W \parallel M_{\{z\}})\backslash\{m\}))
\end{aligned}
$$

$$
\begin{aligned}
& (W \parallel M_{\{a\}})\backslash\{m\} \\
= \; & (?z \;\rightarrow\; ((W \parallel M) \textbf{ if } z = a \textbf{ else } (W \parallel M_{\{a,b\}})))\backslash\{m\} \\
= \; & (?z \;\rightarrow\; ((W \parallel M)\backslash\{m\} \textbf{ if } z = a \textbf{ else } (W \parallel M_{\{a,b\}})\backslash\{m\}))
\end{aligned}
$$

$$
\begin{aligned}
& (W \parallel M_{\{a,b\}})\backslash\{m\} \\
= \; & (!m \;\rightarrow\; (W' \parallel M_{\{c\}}) \mid ?z \;\rightarrow\; \bot)\backslash\{m\} \\
= \; & (\mathit{skip} \;\rightarrow\; ((W' \parallel M_{\{c\}})\backslash\{m\}) \mid ?z \;\rightarrow\; \bot)
\end{aligned}
$$

$$
\begin{aligned}
& (W' \parallel M_{\{c\}})\backslash\{m\} \\
= \; & (!c \;\rightarrow\; (W \parallel M) \mid ?z \;\rightarrow\; (W' \parallel M_{\{c,z\}}))\backslash\{m\} \\
= \; & (!c \;\rightarrow\; ((W \parallel M)\backslash\{m\}) \mid \ldots).
\end{aligned}
$$

Finally, we eliminate the *skip*-guard to obtain

$$
\begin{aligned}
& (W \parallel M_{\{a,b\}})\backslash\{m\} \\
= \; & (!c \;\rightarrow\; ((W \parallel M)\backslash\{m\}) \mid ?z \;\rightarrow\; \bot).
\end{aligned}
$$

Thus C and $(W \parallel M)\backslash\{m\}$ satisfy the same set of equations and so, because all recursions are guarded (by inputs or outputs), exhibit the same behaviour. We conclude that a C-element can be implemented by feeding back the output of a majority-element to one of its inputs; signals arriving at that input remain exposed to the environment as outputs.

5 Submodels

In this section, we show that receptive process theory can be specialized so as to model buffered communication between a system and its environment. We first consider communication through buffers of infinite capacity. Thus we are able to reason about data flow networks, which have been widely studied in the literature. (Though much of the literature is concerned with fairness issues which are outside the scope of our model.) We then consider communication through wires, which can carry at most one signal (voltage-level transition) at a time. This has practical application in the design of delay-insensitive circuits.

We do not go into much detail here. The reader is referred to [8, 4] on data flow networks and [9, 10, 11] on delay-insensitive circuits.

5.1 Data Flow Networks

In this case, a system and its environment do not interact directly, but rather through a number of buffers of infinite capacity. A fully abstract model is obtained by considering the overall behaviour of a system equipped with its buffers, i.e., its observable behaviour. This gives rise to a submodel of receptive process theory in which processes meet two conditions concerned with the reordering of inputs and outputs, in addition to conditions 1-5. The failures and divergences of a process are closed under a reordering relation $\sqsubseteq$, i.e.,

$$s \sqsubseteq t \ \wedge \ t \in F \ \Rightarrow \ s \in F \tag{6}$$

$$s \sqsubseteq t \ \wedge \ t \in F\!\uparrow \ \Rightarrow \ s \in F\!\uparrow \tag{7}$$

Informally, reordering a trace involves interchanging inputs to distinct buffers, interchanging outputs from distinct buffers and shifting inputs in front of outputs. If s reorders t ($s \sqsubseteq t$), then the behaviour of the process after engaging in t is more deterministic than its behaviour after engaging in s.

It is possible to re-interpret all process-expressions in this submodel, after some small changes in the definitions of the operators. Additional algebraic laws capture the reordering conditions.

5.2 Delay-Insensitive Circuits

In this case, a process models a system together with the wires that connect it to its environment. The two reordering conditions above are also satisfied by the processes in this submodel. Furthermore, if a second signal is sent along a wire before a previous signal has been received, then the two signals can interfere with undesirable consequences. We model this interference as divergence and so have the condition

$$saa \in \widehat{F} \ \Rightarrow \ saa \in F\!\uparrow \tag{8}$$

Again it is possible to re-interpret process-expressions in this submodel. However, parallel composition and concealment have to be combined into a single operator. Fan-in and fan-out of wires can be achieved by composing with appropriate merge and fork processes, so no generality is lost. Additional algebraic laws capture transmission interference.

6 Conclusion

Receptive process theory, like CSP, is an algebraic theory of processes based on a failures-divergences semantic model. It is equipped with a sound and complete set of algebraic laws. The laws are sound because each equates two expressions that denote the same process. The laws are complete because every (non-recursive) process-expression can be transformed into a normal form.

The theory enables us to specify not only the behaviour of a system, but also limitations that are placed on its environment. (The significance of this can be seen in conventional sequential programming, where we specify not only a postcondition, but also a precondition.)

The parallel composition and concealment operators support a hierarchical approach to design, in which components taken as primitive at one level of design can be implemented independently at another.

In summary, receptive process theory provides an abstract model of asynchronous communication which could form the basis of design methods for both software and hardware systems. In particular, it can be applied directly to the design of asynchronous circuits. Data flow networks can also be studied within the theory.

Acknowledgements. The elegance of this theory owes much to Tony Hoare who has directed and encouraged me in this research. I am also grateful to many other colleagues with whom I have been collaborating in this area, in particular, He Jifeng, Jan Tijmen Udding, Tom Verhoeff and Rudolf Mak. This work was completed while on a Visiting Research Fellowship at Eindhoven University of Technology. The financial support of the ESPRIT Basic Research Action CONCUR is acknowledged.

A Proofs of some stated results

In this appendix, we substantiate some of our claims about the concealment and parallel composition operators. We begin with two useful lemmas expressing the finitary nature of our processes.

Lemma 20 $\{t \in O^* | st \in F \backslash F\uparrow\}$ is finite.
Proof. We may assume that the set is non-empty. Then $s \notin F\uparrow$ by condition 1, and finiteness follows from the definition of $\uparrow$. □

Lemma 21 $\{t \in O^* | st \in \widehat{F} \backslash F\uparrow\}$ is finite.
Proof.

$$\begin{array}{lll} & \{t \in O^* | st \in \widehat{F} \backslash F\uparrow\} & \\ = & \{t \in O^* | \exists u \in O^*.\ stu \in F \wedge st \notin F\uparrow\} & \text{(definition of } \widehat{\ }) \\ \subseteq & \{t \in O^* | \exists u \in O^*.\ stu \in F \backslash F\uparrow\} & \text{(Lemma 1)} \end{array}$$

which is finite by Lemma 20. □

The next four lemmas pertain to concealment of outputs.

Lemma 22 $\{s \in \widehat{F} \backslash F\uparrow | s' = s \backslash C\}$ is finite.
Proof. Induction on s'.
Case ε. $\{s \in \widehat{F} \backslash F\uparrow | \varepsilon = s \backslash C\} \subseteq \{s \in O^* | s \in \widehat{F} \backslash F\uparrow\}$, which is finite by Lemma 21.
Case $s'a$. $\{s \in \widehat{F} \backslash F\uparrow | s'a = s \backslash C\} = \{tau \in \widehat{F} \backslash F\uparrow | s' = t \backslash C \wedge \varepsilon = u \backslash C\}$, which is finite because there are only a finite choice for t (the ind. hyp. applies since $t \in \widehat{F} \backslash F\uparrow$ by conditions 1 and 4) and a finite choice for u (by Lemma 21). □

We are now ready to prove our previously-stated lemma concerning the divergences that result from concealment.

Lemma 23 $d(P\backslash C) = \{s\backslash C | s \in dP\}$.
Proof.

$$\begin{array}{lll} & s' \in d(P\backslash C) & \\ \Leftrightarrow & \{t' \in (O\backslash C)^* | s't' \in f(P\backslash C)\} \text{ is infinite} & \text{(definition of } \uparrow\text{)} \\ \Leftrightarrow & \{t' \in (O\backslash C)^* | \exists u \in fP.\ s't' = u\backslash C\} \text{ is infinite} & \text{(def. of } f(P\backslash C)\text{)} \\ \Leftrightarrow & \{t\backslash C \in (O\backslash C)^* | \exists s.\ st \in fP \wedge s' = s\backslash C\} \text{ is infinite} & \end{array}$$

($\Leftarrow$) Suppose $s' = s\backslash C$ for some $s \in dP$. Then $st \in dP$ for all $t \in O^*$, and so $\{t\backslash C \in (O\backslash C)^* | st \in fP\}$ is infinite as required, since $O \neq C$ and $dP \subseteq fP$. ($\Rightarrow$) Suppose $s' = s\backslash C$ for no $s \in dP$. This contradicts the above set being infinite because there are only a finite choice for s (by Lemma 22) and a finite choice for t (by definition of $\uparrow$). □

We precede our proof of the continuity of the concealment operator with the following lemma.

Lemma 24 For any chain of failure sets such that $F_i \supseteq F_{i+1}$, $i \geq 0$,

$$(\forall i.\ \exists t.\ st \in F_i\!\uparrow \wedge\, t' = t\backslash C) \;\Rightarrow\; (\exists t.\ st \in (\bigcap_i F_i)\!\uparrow \wedge\, t' = t\backslash C).$$

Proof. Induction on t'. First observe that if $s \in F_i\!\uparrow$ for all i, then we can simply take $t = t'$ because of condition 1 and the continuity of $\uparrow$.
Case ε. If $\exists t.\ st \in F_i\!\uparrow \wedge\, \varepsilon = t\backslash C$, then $s \in F_i\!\uparrow$ by Lemma 1, and the result follows from our observation.
Case at'. If $\exists t.\ st \in F_i\!\uparrow \wedge\, at' = t\backslash C$, then $\exists u \in C^*, v.\ suav \in F_i\!\uparrow \wedge\, t' = v\backslash C$. Because of our observation, we may suppose $s \notin F_j\!\uparrow$ for some j. By Lemma 21, there is only a finite choice for u for that j. Since $F_i \supseteq F_{i+1}$, $i \geq 0$, it follows that $\exists u \in C^*.\ \forall i.\ \exists v.\ suav \in F_i\!\uparrow \wedge\, t' = v\backslash C$. One application of the ind. hyp. completes the proof. □

Lemma 25 For any chain of processes P_i such that $fP_i \supseteq fP_{i+1}$, $i \geq 0$, with l.u.b. P, i.e., $fP = \bigcap_i fP_i$,

$$f(P\backslash C) \;=\; \bigcap_i f(P_i\backslash C).$$

Proof. Since concealment is monotonic, we need only prove containment. Suppose $s' \in \bigcap_i f(P_i\backslash C)$. Then $\forall i. \exists s \in fP_i.\ s' = s\backslash C$. If $\forall i.\ \exists s \in dP_i.\ s' = s\backslash C$, then we are done by Lemma 24. Otherwise, for i sufficiently large, there is only a finite choice for s by Lemma 22 and so, since $fP_i \supseteq fP_{i+1}$, $i \geq 0$, we are also done. □

We now turn to parallel composition. The key step in proving continuity of $P \parallel Q$ is the following lemma.

Lemma 26 For any chain of processes P_i such that $fP_i \supseteq fP_{i+1}$, $i \geq 0$,

$$\forall i.\ \mathcal{T}_i(s) \text{ is infinite } \Rightarrow \ (\bigcap_i \mathcal{T}_i(s)) \text{ is infinite,}$$

where $\mathcal{T}_i(s) = \{t \in O^* | st \in (tP_i)w(tQ)\}$.
Proof. Observe that $\mathcal{T}_i(s)$ is prefix-closed and enjoys the property

$$t \in O^* \wedge u \in \mathcal{T}_i(st) \ \Rightarrow \ tu \in \mathcal{T}_i(s).$$

It therefore suffices to prove that

$$\forall i.\ \mathcal{T}_i(s) \text{ is infinite } \Rightarrow \ (\bigcap_i \mathcal{T}_i(s)) \text{ is infinite} \\ \vee(\exists c \in O.\ \forall i.\ \mathcal{T}_i(sc) \text{ is infinite}).$$

There are two cases to consider.
Case $\forall i.\ \{c \in O | c \in \mathcal{T}_i(s)\}$ is infinite. Then, by the definitions of $\mathcal{T}_i(s)$ and w and by Lemma 21, $\forall i.\ s\lceil(iP \cup oP) \in dP_i \vee s\lceil(iQ \cup oQ) \in dQ$. Hence, either $(oP)^* \subseteq \bigcap_i \mathcal{T}_i(s)$ or $(oQ)^* \subseteq \bigcap_i \mathcal{T}_i(s)$. Either way, $\bigcap_i \mathcal{T}_i(s)$ is infinite.
Case $\{c \in O | c \in \mathcal{T}_j(s)\}$ is finite for some j. Then, since the $\mathcal{T}_i(s)$ $(i \geq 0)$ are infinite, prefix-closed and ordered by containment, $\{c \in O | \mathcal{T}_k(sc) \text{ is infinite}\}$ is finite and non-empty, for all $k \geq j$. These sets are themselves ordered by containment and so there must exist $c \in O$ such that $\forall i.\ \mathcal{T}_i(sc)$ is infinite. □

It follows that $((tP)w(tQ))\uparrow$ is continuous in P and, since closing up under extension is continuous, $d(P \parallel Q)$ is also continuous in P. Now we only have to observe that $(\bigcap_i S_i) \cup (\bigcap_i T_i) = \bigcap_i S_i \cup T_i$ for sets S_i and T_i such that $S_i \supseteq S_{i+1}$, $T_i \supseteq T_{i+1}$, $i \geq 0$, to see that $f(P \parallel Q)$ is continuous in P.

References

[1] Brookes, S.D.: A Model for Communicating Systems. PhD Thesis, Oxford University (1983).

[2] Brookes, S.D., Roscoe, A.W.: An Improved Failures Model for Communicating Sequential Processes. Lecture Notes in Computer Science, Vol. 197, pp. 281-305, Springer-Verlag (1984).

[3] Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. PhD Thesis, Carnegie Mellon University (1988).

[4] He, J., Josephs, M.B., Hoare, C.A.R.: A Theory of Synchrony and Asynchrony. In Proceedings IFIP Working Conference on Programming Concepts and Methods, Sea of Galilee, to appear (1990).

[5] Hennessy, M.: Algebraic Theory of Processes. Series in Foundations of Computing, MIT Press (1988).

[6] Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM, Vol. 21, pp. 666-677 (1978).

[7] Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985).

[8] Josephs, M.B., Hoare, C.A.R., He, J.: A Theory of Asynchronous Processes. Oxford University Programming Research Group Technical Report (1989).

[9] Josephs, M.B., Udding, J.T.: An Algebra for Delay-Insensitive Circuits. In Proceedings DIMACS/IFIP Workshop on Computer-Aided Verification, Rutgers University, New Jersey, to appear (1990).

[10] Josephs, M.B., Udding, J.T.: Delay-Insensitive Circuits: An Algebraic Approach to their Design. In Proceedings CONCUR '90, Amsterdam, to appear (1990).

[11] Josephs, M.B., Udding, J.T.: The Design of a Delay-Insensitive Stack. In Proceedings Workshop on Designing Correct Circuits, Oxford, to appear (1990).

[12] Mead, C., Conway, L.: Introduction to VLSI Systems. Addison-Wesley (1980).

[13] Milner, A.J.R.G.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, Vol. 92, Springer-Verlag (1980).

[14] Milner, A.J.R.G.: Communication and Concurrency. Prentice Hall International Series in Computer Science (1989).

[15] Snepscheut, van de, J.L.A.: Trace Theory and VLSI Design. PhD Thesis, Eindhoven University of Technology (1983).

[16] Roscoe, A.W., Hoare, C.A.R.: The Laws of Occam Programming. Theoretical Computer Science, Vol. 60, pp. 177-229 (1988).

[17] Udding, J.T.: Classification and Composition of Delay-Insensitive Circuits. PhD Thesis, Eindhoven University of Technology (1984).

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits.
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes.
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films.
85/04	T. Verhoeff H.M.L.J.Schols	Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.
86/01	R. Koymans	Specifying message passing and real-time systems.
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specification of information systems.
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures.
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several systems.
86/05	J.L.G. Dietz K.M. van Hee	A framework for the conceptual modeling of discrete dynamic systems.
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP.
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers.
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987).
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language.
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing.
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86).
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes.
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations of the monitor concept.
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases.
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems.
87/04	T.Verhoeff	Delay-insensitive codes - An overview.
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns - searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language.
87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts.
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas.
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples.

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces.
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen.
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations.
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG.
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence.
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems.
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples.
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to Handle Structured Information.
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems.
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication.
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way.
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification.
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling.
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

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

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

		p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.