ANALOG PROCESSES

Luca Cardelli

Department of Computer Science
University of Edinburgh
J.C.M.B., The King's Buildings
Edinburgh EH9 3JZ

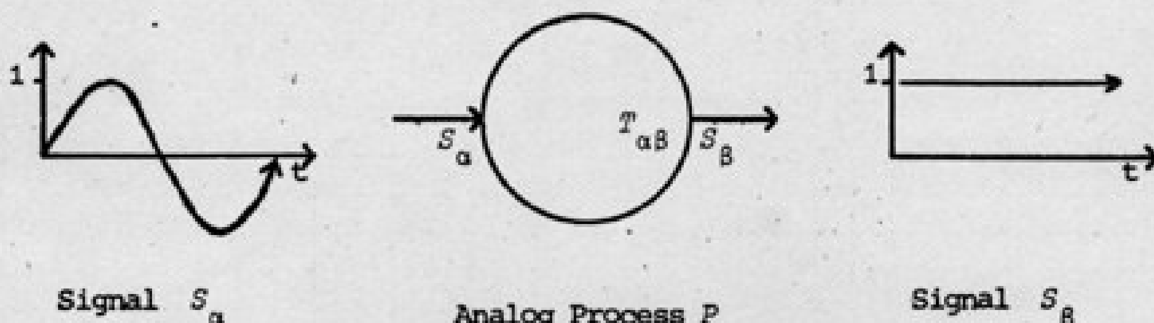
Introduction

We intend to study systems of communicating processes in which processes interact with each other in a continuous asynchronous fashion, as do planets around a star. Such processes cannot be considered "computing agents" as they just behave instant by instant according to laws which are not "computations" in any mechanical sense. Their interactions are not instantaneous synchronous communications, but rather a continuous flow of information which does not fit in the message passing paradigm. These systems develop in continuous time and their interactions are often expressed in terms of continuous values. More importantly, their behaviour cannot be fully understood by forcing them into a discrete environment, as a whole range of interesting phenomena is then lost.

Asynchronous electronic circuits will be used as a source of interesting examples, and we shall be able to model and analyse puzzling behaviours like asynchronous feedbacks, metastable states, arbitration and indeterminacy. All these phenomena will be explained in terms of a single principle, which simply forbids the existence of null-delay feedback loops. This also shows that these complex real-world behaviours can be described by just assuming concurrency in continuous time, and do not necessarely depend on other features of the physical universe, like relativistic or quantum mechanic effects.

Analog Processes

A *signal* is a value varying in continuous time, and an *analog process* is a transformation of signals, for example:



Signal S_α Analog Process P Signal S_β

The signals above can be expressed as functions of time:

$$S_\alpha(t) = \sin t \qquad\qquad S_\beta(t) = 1$$

and the process P transforming S_α into S_β can be described by a single *transition* $T_{\alpha\beta}$ which could be in this case:

$$T_{\alpha\beta}(s)(t) = s(t) - \sin t + 1$$

In fact, applying $T_{\alpha\beta}$ to S_α we get S_β:

$$\begin{aligned} T_{\alpha\beta}(S_\alpha) &= \lambda t.\ S_\alpha(t) - \sin t + 1 \\ &= \lambda t.\ \sin t - \sin t + 1 \\ &= \lambda t.\ 1 \\ &= S_\beta \end{aligned}$$

In general a process will consist of several transitions, and systems will comprise several connected processes.

An algebra of analog processes

A *process* is described by a collection of *transitions* $M \rightarrow \beta$, where the term M is the *signal* produced by the transition, and β is the *output port* of the transition. The signal M is an expression of some of the *input ports* of the process. Here is an example of the syntax we shall use to talk about processes:

$$(\alpha \rightarrow \beta) + ((\alpha \uplus \gamma) \rightarrow \delta)$$

For clarity we shall sometimes prefix processes with their input ports, although this is not strictly necessary as the input ports of a process will always coincide with the free variables of the signal parts of the transitions:

$$\alpha\ \gamma\colon\ \alpha \rightarrow \beta\ +\ \alpha \uplus \gamma \rightarrow \delta \qquad\qquad (1)$$

this is a process with input ports α,γ and output ports β,δ (parenthesis have been omitted).

The intended behaviour of processes will be explained by algebraic laws. We will only be concerned with the most interesting laws and we shall not try to present a complete set of equations. The following three laws express the fact that processes are unordered collections of transitions:

[++] $(T + T') + T'' = T + (T' + T'')$

[+] $T + T' = T' + T$

[*NIL*] $T + NIL = T$

where *NIL* is the empty transition and T, T' and T'' range over transitions.

The intended meaning of expression (1) is a process which at any instant of time produces on the output port β the current value of the input port α, and on the output port δ the current value of the *join* ($\uplus$) of α with γ. The join

operation represents the simultaneous presence of two signals on the same "line", and its exact meaning is left unspecified, except that the join operation must exist for every pair of signals (of the same type) and it must satisfy:

[⊎⊎] $(M \uplus N) \uplus P = M \uplus (N \uplus P)$

[⊎] $M \uplus N = N \uplus M$

For example, for boolean-valued signals s_1, s_2 we might define $s_1 \uplus s_2$ to be at any instant of time a boolean *or*, i.e.: $(s_1 \uplus s_2)(t) = s_1(t)$ *or* $s_2(t)$.

The existence of a constant $\dot{-}$ (*nosignal*) is also assumed; it relates to join as follows:

[$\dot{-}$] $M \uplus \dot{-} = M$

In the previous boolean example we can define nosignal as the signal constantly *false*, i.e.: $\dot{-}(t) = false$. The join operation is also used in the following law, which accounts for the presence of repeated output ports:

[⊎+] $M \to \beta + N \to \beta = M \uplus N \to \beta$

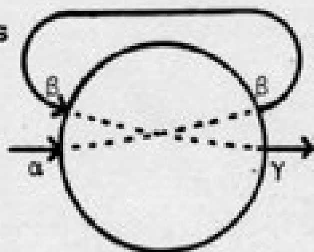
Now we will define some basic operations on processes, together with their algebraic laws.

COMPOSITION. The composition of two processes P and Q is written $P|Q$. The output ports of P are linked to the homonymous input ports of Q, and the output ports of Q are linked to the homonymous input ports of P; the idea being that signals flow through these connections from one process to the other. We have the following rules for composition:

[||] $(P \mid Q) \mid R = P \mid (Q \mid R)$

[|] $P \mid Q = Q \mid P$

[|+] $(\sum_{i \in I} T_i) \mid (\sum_{j \in J} T_j) = \sum_{k \in I \cup J} T_k$
where I and J are disjoint sets of indexes



An example of the law [|+] is:

$(\alpha: \alpha \to \beta) \mid (\beta: \beta \to \gamma) = \alpha\ \beta: \alpha \to \beta + \beta \to \gamma$

Note that composition may introduce loops of signals (β being both an input and an output port) and indeed such loops may be present in the first place. We will come later to the exact semantics of such situations; for the moment it will be intended that a looping signal overwrites itself by a join operation.

RESTRICTION. The restriction $P\backslash\alpha$ of P cancels α from the input and output ports of P, making communication via α impossible. We have:

[\] $P\backslash\alpha = P$ if $\alpha \notin ports(P)$

[\\] $P\backslash\alpha\backslash\beta = P\backslash\beta\backslash\alpha$

$$[\backslash|] \quad (P \mid Q)\backslash\alpha = P\backslash\alpha \mid Q\backslash\alpha$$
$$\text{if not } ((\alpha \in \text{input-ports}(P) \text{ and } \alpha \in \text{output-ports}(Q)) \text{ or } (\alpha \in \text{output-ports}(P) \text{ and } \alpha \in \text{input-ports}(Q)))$$

Now we need a law to distribute \ over +, and at first sight this could be:

$$(\textstyle\sum_{i\in I} T_i)\backslash\alpha = \sum_{i\in I}(T_i\backslash\alpha)$$
$$(M \to \alpha)\backslash\alpha = NIL$$
$$(\ldots\ \alpha\ \ldots \to \beta)\backslash\alpha = \ldots \dot{-} \ldots \to \beta$$

Unfortunately this does not work well in the case:

$$(\alpha\colon M \to \alpha \ + \ \alpha \to \beta)\backslash\alpha = \dot{-} \to \beta$$

In fact we want to interpret \ as a hiding operator, which should not change the inner behaviour of the process. The result we want to get is, at least:

$$(M \to \alpha \ + \ \alpha \to \beta)\backslash\alpha = M \to \beta$$

But even this is not enought in the case that M is an expression $M[\alpha]$ of α itself, e.g. when we have a loop over the restriction variable whose result is exported through another output port (in this case β). To solve this problem we need to introduce recursively defined signals $(\mu\alpha.\ M)$:

$$[\mu] \quad \mu\alpha.\ M = \mu\beta.\ M[\beta/\alpha]$$
$$[\mu\mu] \quad \mu\alpha.\ M = M[\mu\alpha.\ M/\alpha]$$

Then the law for restriction is:

$$[\backslash+] \quad (\textstyle\sum_{i\in I} T_i)\backslash\alpha = \sum_{j\in J} T'_j$$
where $J = \{i\in I\colon T_i = M_i \to \alpha_i \text{ and } \alpha_i \neq \alpha\}$
and $T'_j = T_j[\mu\alpha.\ N/\alpha]$
with N = the join of all the M_i such that $T_i = M_i \to \alpha_i$ is in $\sum_{i\in I} T_i$ and $\alpha_i = \alpha$ (and $N = \dot{-}$ if no such M_i exists)

Examples:

$$(\alpha\colon \alpha \to \beta)\backslash\alpha = (\mu\alpha.\ \dot{-}) \to \beta = \dot{-} \to \beta$$
$$(\alpha\ \beta\colon \alpha \to \beta \ + \ \beta \to \gamma)\backslash\beta = \alpha\colon \alpha \to \gamma$$
$$(\alpha\ \beta\colon \alpha \to \beta \ + \ \beta \to \alpha)\backslash\beta = \alpha\colon \alpha \to \alpha$$
$$(\alpha\colon \alpha \to \alpha)\backslash\alpha = NIL$$
$$(\alpha\colon \alpha \to \alpha \ + \ \alpha \to \beta)\backslash\alpha = (\mu\alpha.\ \alpha) \to \beta$$

The important point in this definition is that looping situations are somehow hidden or preserved, but never "unfolded" by $\backslash\alpha$.

RELABELLING. The relabelling $P\langle\alpha_1/\beta_1; \ldots; \alpha_n/\beta_n\rangle$ is the process obtained from P simulataneously substituting the (input and/or output) ports $\alpha_1 \ldots \alpha_n$ by $\beta_1 \ldots \beta_n$. A relabelling $\langle R\rangle = \langle\alpha_i/\beta_i\rangle$ is a bijection $R\colon L \to L$ over

the ports L of P, i.e. β_i are all and only the ports of P, and α_i are distinct port names. Dummy substitutions will be omitted, so that $\langle\rangle = \langle\alpha_i/\alpha_i\rangle$.

$$[\langle\rangle] \quad P\langle\rangle = P$$

$$[\langle\rangle\langle\rangle] \quad P\langle R\rangle\langle S\rangle = P\langle S \circ R\rangle$$

$$[\langle\rangle\backslash] \quad (P\backslash\alpha)\langle R\rangle = (P\langle R;\ \beta/\alpha\rangle)\backslash\beta \quad \text{if } \alpha \in \text{ports}(P) \text{ and } \beta \notin \text{range}(R)$$

$$[\langle\rangle|] \quad (P|Q)\langle R\rangle = (P\langle R'\rangle)|(Q\langle R''\rangle) \quad \text{where } R' = R \text{ restricted to ports}(P) \text{ and } R'' = R \text{ restricted to ports}(Q)$$

To distribute $\langle R\rangle$ over $+$ we actually perform a (metasyntactical) substitution:

$$[\langle\rangle+] \quad (\textstyle\sum_{i\in I} T_i)\langle\alpha_j/\beta_j\rangle = \textstyle\sum_{i\in I}(T_i[\alpha_j/\beta_j])$$

Ex:

$$(\alpha\ \beta\colon \alpha \to \beta \ + \ \beta \to \alpha)\langle\alpha/\beta;\ \beta/\alpha\rangle = \beta\ \alpha\colon \beta \to \alpha \ + \ \alpha \to \beta$$

The algebraic laws we have so far presented form what we will call an *analog algebra*. These laws can be grouped into two categories: external laws (relating $|$, $\backslash\alpha$ and $\langle R\rangle$: [||], [|], [\], [\\], [\|], [<>], [<><>], [<>\] and [<>|]) concerning the syntesis of processes from simpler processes, and internal laws (all the others) concerning the inner structure of processes. The external laws hold for Milner's *flow algebras* [Milner 79]. Flow algebras are extended in [Milner 78] by a set of internal laws for communicating processes, and are then called *behaviour algebras*. Our internal laws are quite different from Milner's ones, but they seem to fit very well in the general framework of flow algebras, even if the meaning of $|$, $\backslash\alpha$ and $\langle R\rangle$ is radically different.

A denotational model

In the rest of this paper we will study a particular analog algebra, built within the denotational semantics framework. This will allow us to study the exact meaning of processes just by computing their semantics and observing their input-output behaviour. The denotational semantics will also prove useful in discussing some tricky situations like feedbacks and recursive signals. Unfortunately we do not have space for full details and we shall only try to sketch the main ideas.

Processes are collections of transitions; in particular a process with n inputs is an association of labels (the output ports) to transitions with n inputs:

$$P_n = L \to T_n$$

where P_n, L and T_n are semantic domains (complete partial orders): L is the flat domain of port labels, T_n is the domain of transitions with n inputs, and P_n is the domain of processes with n inputs. The domain P will also denote (the disjoint union of) the domains P_n for any n.

A transition with n inputs is a function taking n input signals (each labelled with its input ports) and producing an output signal:

$$T_n = S_L^n \to S$$

where S_L^n is some domain of unordered labelled n-tuples of signals.

Signals are functions from time to a domain of values. We can have several types of signals, like boolean signals, real signals etc.

$$S = K \to V$$

where K is the flat domain of positive real numbers, and V is some data domain admitting a constant $\phi \in V$ and an (infix) operation $\upsilon: V \times V \to V$ such that the properties [⊎⊎], [⊎] and [$\dot{-}$] hold by defining:

$$\dot{-}(t) = \phi \qquad\qquad s_1(t) \upsilon s_2(t) = (s_1 \uplus s_2)(t)$$

for all $t \in K$ and $s_1, s_2 \in S$.

We need some notation for elements in these domains; λ-notation will be used for signals $s \in S = K \to V$. Elements of S_L^n will be denoted by expressions like:

$$[\alpha_1 : s_1 ; \ldots ; \alpha_n : s_n] \qquad \text{with } \alpha_1 \ldots \alpha_n \in L,\ s_1 \ldots s_n \in S$$

which are meant to be unordered tuples of labelled signals $\alpha_i : s_i$ with the additional property:

$$[\ \ldots \alpha : s' ; \alpha : s'' \ldots\ ] = [\ \ldots \alpha : s' \uplus s'' \ldots\ ]$$

and operations:

$$[\alpha_i : s_i] \backslash \alpha = [\alpha_j : s_j] \qquad \text{with } i \in I,\ j \in J \text{ and } J = \{i \in I \mid \alpha_i \neq \alpha\}$$
$$[\alpha_i : s_i].\alpha = \uplus\{s_k \mid \alpha_k = \alpha\} \qquad (\text{where } \uplus\{\} = \dot{-})$$
$$[\alpha_i : s_i] \uplus [\alpha'_j : s'_j] = [\alpha_i : s_i ; \alpha'_j : s'_j]$$

Elements of $T_n = S_L^n \to S$ of the form:

$$\lambda x.\ \ldots\ x.\alpha_1\ \ldots\ x.\alpha_n\ \ldots \qquad (\alpha_1 \ldots \alpha_n \in L)$$

will be abbreviated (with a change of font) as:

$$\lambda[a_1 \ldots a_n].\ \ldots\ a_1\ \ldots\ a_n\ \ldots,$$

where $[a_1 \ldots a_n]$ is an unordered tuple of variables. Notice that this notation allows for unordered application by label names, as in:

$$(\lambda[a_1\ a_2].\ a_1 * a_2)[\alpha_2 : 3 ; \alpha_1 : 5] = 5*3$$

Finally, processes $p \in P_n = L \to T_n$ of the form:

$$\lambda x.\ (x{=}\alpha_1) \Rightarrow t_1\ ;\ \ldots\ ;\ (x{=}\alpha_n) \Rightarrow t_n\ ;\ (\lambda[\,].\ \dot{-})$$

will be abbreviated as:

$$\{t_1 \to \alpha_1;\ \ldots\ ;\ t_n \to \alpha_n\}$$

There are three semantic evaluation functions:

T : terms × vars → T	for term expressions
S : signals × ports → S	for signal expressions
P : processes × ports → P	for process expressions

with two kinds of environments: vars = Ide → V; ports = L → S.

We shall first discuss the semantics of process expressions, then the semantics of signal expressions, giving the syntax at the same time. We shall not treat the semantics of terms, as term expressions will always have an evident meaning.

The following is the semantics of a very simple process, consisting of a single transition:

$$\mathsf{P}[\![\alpha_i\colon S \to \beta]\!]\sigma =$$
$$Y\ \lambda P.\ \{\lambda[a_i].\ \mathsf{S}[\![S]\!]\sigma[a_i \uplus P(\beta)[\alpha_i{:}a_i]/\alpha_i] \to \beta\}$$

The fixpoint and the join operation are needed just in case β is equal to one of the α_i, i.e. when there is a feedback. Otherwise the previous expression reduces simply to:

$$\{\lambda[a_i].\ \mathsf{S}[\![S]\!]\sigma[a_i/\alpha_i]\}$$

In case of a feedback, say $\alpha_3=\beta$, the input to α_3 is a_3 (the input to processor P) joined to what comes out of β, which is $P(\beta)[\alpha_i{:}a_i]$. In fact $P(\beta)$ is the transition associated with β, which receives as input the same input of the process: $[\alpha_i{:}a_i]$.

The same idea is used in giving the semantics of composition, in which the component processes may feed each other in complex ways:

$$\begin{aligned}
p|q \;=\;& \text{let } p = \{s_i \to \gamma_i\} \\
& \text{and } q = \{r_j \to \delta_j\} \text{ in} \\
& \text{let } s_i = \lambda[a_h].\ M_i \\
& \text{and } r_j = \lambda[b_k].\ N_j \text{ in} \\
& Y\ \lambda R.\ \{\lambda[a_h b_k].\ p(\gamma_i)[\alpha_h{:}a_h \uplus R(\alpha_h)[\alpha_h{:}a_h;\ \beta_k{:}b_k]] \to \gamma_i\} \\
& \qquad \uplus \{\lambda[a_h b_k].\ q(\delta_j)[\beta_k{:}b_k \uplus R(\beta_k)[\alpha_h{:}a_h;\ \beta_k{:}b_k]] \to \delta_j\}
\end{aligned}$$

This composition is commutative ([|] holds); to prove associativity ([||]) we had to assume absorption of $\uplus$, i.e. $s \uplus s = s$ (which also implies $P|P = P$).

The other laws of analog algebras are easily verified, if we complete the definition of P by the following equations:

$$\mathsf{P}[\![T_1 + \ldots + T_n]\!]\sigma \;=\; \mathsf{P}[\![T_1]\!]\sigma \mid \ldots \mid \mathsf{P}[\![T_n]\!]\sigma$$
$$\mathsf{P}[\![P\backslash\alpha]\!]\sigma \;=\; \lambda\beta.\ \lambda x.\ \beta{=}\alpha \Rightarrow \dot{-}\ ;\ (\mathsf{P}[\![P]\!]\sigma)(\beta)(x \uplus [\alpha{:}\dot{-}])$$
$$\begin{aligned}\mathsf{P}[\![P\langle\beta_i/\alpha_i\rangle]\!]\sigma \;=\; & \text{let } p = \mathsf{P}[\![P]\!]\sigma \\ & \text{in } \lambda\gamma.\ \lambda[b_i].\ \gamma{=}\beta_1 \Rightarrow p(\alpha_1)[\alpha_i{:}b_i]\ ;\ \ldots\ ; \\ & \qquad\qquad\qquad \gamma{=}\beta_n \Rightarrow p(\alpha_n)[\alpha_i{:}b_i]\ ;\ \dot{-}\end{aligned}$$

We pass now to consider signals; a simple way to specify them is to describe their value at any instant of time, using a sort of λ-notation:

$$\mathsf{S}[\![@t.V]\!]\sigma \;=\; \lambda a.\ \mathsf{T}[\![V]\!]\varepsilon[a/t] \qquad (\varepsilon \text{ is the empty environment})$$

for example @t. 3*sin t. We have the equivalences $\dot{-} = @t.\ \phi$ and $a \uplus b = @t.\ a(t) \cup b(t)$. The notation $\hat{}V$ will be used as an abbreviation of $@t.\ V$, when t is not a free variable in V, like in $\hat{}3 = @t.\ 3$.

Signals can also be defined by recursion:

$$\mathsf{S}[\![\mu s.\ S]\!]\sigma \;=\; \mathsf{Y}\ \lambda a.\ \mathsf{S}[\![S]\!]\sigma[a/s]$$

like in $\mu s.\ @t.\ t{<}1 \Rightarrow \phi\ ;\ s(t{-}1) \equiv \dot{-}$. Two other useful abbreviations are conditional signals and delays:

$$S \Rightarrow S'\ ;\ S'' \;=\; @t.\ S(t) \Rightarrow S'(t)\ ;\ S''(t)$$
$$S'\ \Delta\ S'' \;=\; @t.\ t{<}S''(t) \Rightarrow \phi\ ;\ S'(t{-}S''(t))$$

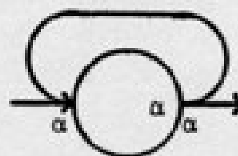
A simple example of delay is $S\ \Delta\ \hat{}3$ which is the signal S constantly delayed by 3 units of time, yielding ϕ during the first three units of time. This notation also allows us to express variable delays.

Notice that the @-notation has too big an expressive power, being able for example to define a signal in terms of the "future" of another signal (or even of itself), but we might impose syntactic restriction to avoid that, leaving Δ as a primitive.

Unfeasibility

Great care has been put into the definition of the algebraic laws and of the denotational semantics, in order to be able to treat circularities; so let us see how it works. The simplest example of a feedback can be found in the following *fast loop* process:

$\alpha{:}\ \alpha \to \alpha$



This process has an input port α, whose input is mixed to the output coming from the output port α. The tricky point is that this process has no internal delay, and the output at any instant

t depends on the input at the same instant t, which depends again on the output at time t. Computing the semantics:

$$\mathbb{P}[\![\alpha\colon \alpha \to \alpha]\!]\sigma$$
$$= \gamma\ \lambda P.\ \{\lambda[a].\ \mathbb{S}[\![\alpha]\!]\sigma[a \uplus P(\alpha)[\alpha{:}a]/\alpha] \to \alpha\}$$
$$= \gamma\ \lambda P.\ \{\lambda[a].\ a \uplus P(\alpha)[\alpha{:}a] \to \alpha\} =_{def} p$$

It is not immediatly clear what p does, but we can try to understand its behaviour by applying some input. We first extract the transition we are interested in (there is only one in this case) applying the output port α:

$$p(\alpha) = \lambda[a].\ a \uplus p(\alpha)[\alpha{:}a]$$

then we apply an input signal to see what is the response of the transition; we choose to apply nosignal:

$$p(\alpha)[\alpha{:}\dot{-}] = \dot{-} \uplus p(\alpha)[\alpha{:}\dot{-}] = p(\alpha)[\alpha{:}\dot{-}] = \bot$$

the result is $\bot$: it happens that the output of the fast loop is $\bot$ for any input, if we assume $\uplus$ to be strict in both its arguments.

Here we have a first example of a clearly "unfeasible" process, which is semantically mapped to undefined. We can also see that a *slow loop* is not mapped to $\bot$ and is perfectly well-defined:

$$\mathbb{P}[\![\alpha\colon \alpha\ \Delta\ \hat{}1 \to \alpha]\!]\sigma$$
$$= \gamma\ \lambda P.\ \{\lambda[a].\ \lambda t.\ t<1 \Rightarrow \phi\ ;\ (a \uplus P(\alpha)[\alpha{:}a])(t-1) \to \alpha\} =_{def} p$$
$$p(\alpha)[\alpha{:}\dot{-}] = \lambda t.\ t<1 \Rightarrow \phi\ ;\ (p(\alpha)[\alpha{:}\dot{-}])(t-1) = \dot{-}.$$

There are also processes whose output signals are only partially undefined; an example is the *zeno loop*:

$$\alpha\colon \alpha\ \Delta\ (@t.\ t<1 \Rightarrow 1-t\ ;\ 0) \to \alpha$$

this is a feedback loop which increases its speed, and at a finite point in time reaches an infinite speed (i.e. a zero delay). The output of the zeno loop for a nosignal input is $\lambda t.\ t<1 \Rightarrow \phi\ ;\ \bot$.

As a general principle, the output of a feedback loop is defined as long as the delay in the loop is greater than zero. This may look trivial, but feedback loops appear in almost any interesting process, and this simple fact has several intriguing consequences. We are going now to look at some of these.

Unexpressibility

We have seen that we can express several physically unfeasible processes. This suggests that our formalism has too big an expressive power, and we might try to impose some constraints in order to exclude unwanted processes. However it would be wrong to think that we can express anything we like.

In particular there are several processes which cannot be exactly expressed, and yet admit approximations up to an arbitrary degree of accuracy. We shall call such unexpressible processes *perfect*, and *imperfect* their expressible approximations.

Consider for example the following (*naive*) *memory cell*.

$$\alpha\ \beta\colon\ \alpha \uplus \beta\ \Delta\ {}^\wedge 1 \rightarrow \beta$$

To work properly as a (write once) memory cell, this process must receive a *set* impulse of length 1 on α. Then this impulse gets into the loop and is "remembered". This memory cell presents two main defects: it will not work properly (i) if the set impulse is longer than 1, or (ii) if the set impulse is shorter than 1. We can solve the first problem by the following (*improved*) *memory cell*:

$$\alpha\ \beta\colon\ (\alpha = \dot{-}\ \Rightarrow\ \alpha\ ;\ \beta)\ \Delta\ {}^\wedge 1 \rightarrow \beta$$

This process will cut off its α line after having received a signal different from $\dot{-}$ for 1 unit of time. But the second problem still remains; if the α signal differs from $\dot{-}$ for less then one unit of time, the output β is not constant. The same problem occurs when the set impulse changes its value during the setting time. Then a varying signal is recorded into the feedback loop, and the output of the memory cell oscillates: we get a (quench free) *metastable state*.

In effect what we really want is a *perfect memory cell* which stores constantly the value of an instantaneous setting spike, so that there can be no indeterminacy due to fluctuations of the input signal. Notice that starting from our improved memory cell we can get better and better approximations to a perfect cell, simply by reducing the delay in the feedback loop. Unfortunatly if we reduce the delay to zero, we do not get a perfect storage device, but only an undefined output. Hence (conjecture) there is no expression denoting a perfect memory cell (which yet exists inside our semantic domains) because there seems to be no way to define a storing device without the use of feedbacks.

Therefore, expressible memory cells are imperfect. It is important to notice that many useful processes have memory cells (or their equivalent) as basic building blocks, and such processes must take into account this imperfection and are likely to be themselves imperfect. In general an imperfect process works "correctly" under some classes of input signals, but in certain critical circumstances there is no way to guarantee its intended operation.

Indeterminacy

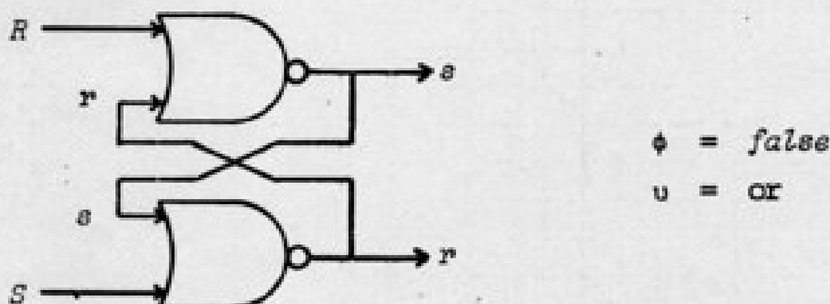
Consider the problem of designing a process which determines the time of occurrence of an event, or which measures the value of a signal when some event (e.g. "measure it now") occurs. First we must agree on a definition of *determining* or *measuring*, and a sensible one seems to be *storing constantly for an unlimited amount of time*. We will not go into the details of such design because it is very similar to the problem of producing a perfect memory cell. In fact it is not difficult to see that perfect determination is impossible, just because perfect storage devices are unfeasible.

A well known case of indeterminacy is *arbitration*, where a device attempts to *determine* which of two events arrives first. A simple way of implementing an arbiter is to use a *decider* and a memory cell. The decider tells at any instant whether the first, the second or both signals are arriving, and the memory cell tries to remember the first decision of the decider. But memory cells are imperfect and so are arbiters based on memory cells. If the two signals arrive too close, the decider changes its decision while the cell is storing it, and the output of the cell is unstable. An alternative way of building an arbiter is by using two *detectors* to determine the time of occurrence of two events, and then compare these times. But it can be shown that detectors are imperfect, and so are arbiters built in this way.

In general the order or coincidence in time of two events cannot be determined. The order cannot be determined when the signals are too close, and the coincidence cannot be determined when the simultaneous signals are too short.

Flip-flops

In this last section we analyse a particular analog process, showing how its detailed behaviour can be derived from its semantics:



ϕ = *false*

$\cup$ = or

This is an *SR flip-flop*. In its *steady state condition* we have the following values on the ports:

$$R = S = s = false; \qquad r = true$$

Starting from this condition and applying a *set* pulse to the port S we get

s = *true* and *r* = *false*. Another set pulse has no effect. Then applying a *reset* pulse to the port *R* we change the output back to *s* = *false* and *r* = *true*. Another reset pulse has no effect. Applying both a set and a reset signal, the output signals oscillate between *true* and *false*, and this is called a *metastable state*. The actual behaviour of a real flip-flop in a metastable state can be rather different from the one described above. We believe it can be modelled by introducing some "quench", but here we shall not undertake this analysis.

The *SR* can be synthetized from smaller components:

```
OR   =  in1 in2: (in1 or in2) Δ ^d' → out
NOT  =  in: (not in) Δ ^d" → out

OR1  =  OR<R/in1;r/in2;w1/out>
OR2  =  OR<S/in1;s/in2;w2/out>
NOT1 =  NOT<w1/in;s/out>
NOT2 =  NOT<w2/in;r/out>

SR   =  (OR1 | NOT1 | OR2 | NOT2)\w1\w2
```

It is an easy exercise to show that this is equivalent to:

```
SR  =  S R s r: not(R or r) Δ ^d → s  +  not(S or s) Δ ^d → r
```

where d=d'+d". Unfortunatly if we try to switch on the flip-flop without supplying any signal (i.e. supplying *false* on all inputs) we immediatly get a metastable state. This happens because starting with *false* on all the inputs, we are not in the steady state condition. To enforce a well defined start, we supply *true* to *r* for the first d seconds. At that time the signal from *S* reaches *r* and the system is ready to work. Hence we redefine:

```
SR  =  S R s r:
            not(R or r) Δ ^d → s  +
            (not(S or s) Δ ^d) ⊎ (@t. t<d) → r
```

Computing the semantics:

```
SR  =  P[[SR]]σ
    =  γ λSR. {λ[S R s r]. λt. t<d => false ;
                      not(R(t-d) or r(t-d) or SR(r)[S:S;R:R;s:s;r:r](t-d))→s;
                λ[S R s r]. λt. t<d => true  ;
                      not(S(t-d) or s(t-d) or SR(s)[S:S;R:R;s:s;r:r](t-d))→r}
```

and extracting the output transitions:

SR(s) = γ λT. λ[S R s r]. λt. t<d => *false* ;
not(R(t-d) or r(t-d) or
(t<2d => *true* ;
not(S(t-2d) or s(t-2d) or
T[*S*:S;*R*:R;*s*:s;*r*:r](t-2d)))

SR(r) = ...

We look at the output signals in absence of input:

SR(s)[*S*:$\dot{-}$;*R*:$\dot{-}$;*s*:$\dot{-}$;*r*:$\dot{-}$] = γ λS. λt. t<2d => *false* ; S(t-2d)
= λt. *false*

SR(r)[*S*:$\dot{-}$;*R*:$\dot{-}$;*s*:$\dot{-}$;*r*:$\dot{-}$] = λt. *true*

This means that for *S*=^*false*, *R*=^*false* we obtain *s*=^*false*, *r*=^*true*; we are in the steady state condition. Now we supply a pulse (λt. t<π) of an unspecified length π:

SR(s)[*S*:(λt. t<π);*R*:$\dot{-}$;*s*:$\dot{-}$;*r*:$\dot{-}$] =
γ λS. λt. t<2d => *false* ; t<2d+π => *true* ; S(t-2d)

There are two cases: (i) the length of the set pulse is $\pi \geq 2d$; then the flip-flop is properly set (the expression above reduces to λt. t<2d => *false* ; *true*) or (ii) the length of the set pulse is $\pi < 2d$; then the flip-flop is in a metastable state and the output signal oscillates between *true* and *false*.

Acknowledgements

Milner's papers on concurrent behaviours have been of inspiration and guide to this work. Robin Milner and Gordon Plotkin also contributed with discussion to the clarification and refinement of several points. I had encouraging talks with Matthew Hennessy at the very beginning of this research, which has been carried out under a scholarship from the Italian National Research Council.

References

[MacQueen 79] D. B. MacQueen, "Models for Distributed Computing", Report 351, IRIA-Laboria, April 1979.

[Milner 78] R. Milner, "Synthesis of Communicating Behaviour", 7th Symposium on Mathematical Foundations of Computer Science, Zakopane, Poland, 1978.

[Milner 79] R. Milner, "Flowgraphs and Flow Algebras", J.ACM, vol 26, n 4, Oct 1979, pp. 794-818.