Computable Processes

Henk Barendregt[1]
Hanno Wupper[2]
Hans Mulder[3]

[1] *CWI, P.O. Box 94079*
1090 GB Amsterdam
and
Computing Science Institute
Catholic University Nijmegen
e-mail: `henk@cwi.nl`

[2] *Computing Science Institute*
Catholic University Nijmegen
e-mail: `wupper@cs.kun.nl`

[3] *Department of Computer Science*
Eindhoven Technological University
e-mail: `hansm@win.tue.nl`

Abstract

The notions of a (digital synchronous) *computable process* and the corresponding *process control machine* are introduced. These concepts are generalizations of the notions of computable function and, say, register machine. Emphasis is put on the difference between the *specification*, the *design*, the *behaviour*, and the *realization* of computable processes.

AMS Subject Classification (1991): 03D99, 68Q05, 68Q10.
CR Subject Classification (1994): B.1.1, F.1.1, F.3.2.
Note: The third author has been partially supported by the Reduction Machine Project I088 of the European Institute of Technology.
Keywords & Phrases: computable processes, feedback, reactive programming, real-time programming, design, specification.

1. Computation versus control

1.1. The problem

Digital computers were invented to perform computations. A computational job consists of finding for a given input n an output $m = f(n)$. The functions f for which this is always possible are called *computable*. The machines that perform the computations have been studied in the form of Turing or register machines. Both have an idealized element: Turing machines have an infinite tape, while register machines can hold arbitrarily large integers.[1]

The computable functions and the engines that 'run' the computations are well understood. Computers, however, also do other things than just compute functions: they control processes. Think of controlling a network of traffic lights by a system with sensors for cars and push buttons for pedestrians. Another example is controlling a chemical plant by a system with sensors for temperature, pressure and actuators for heating/cooling, for opening valves etcetera. Also the actions performed in text editors and worldwide airline reservation systems can be seen as process control. The actions of computing functions and controlling processes can be mutually simulated. But this can be done only in a way that is not natural. The reason is, that the intention of computation is termination, while that of control is continuation.

The paper discusses a notion of computable process and a corresponding notion of process control machine, that can run these processes.[2]

[1] Such idealizations are realistic. Indeed, if one would put an upper bound on the memory size based on today's technology, then tomorrow the bound may have to be enlarged.

[2] These machines are idealized in the same way as register machines: they can contain arbitrary large natural numbers.

1.2. Two examples

Process controlling computers react to and cause events distributed in time and space. In order to do this, they may have to perform complicated computations, but they also have to do something else. To understand exactly what is this extra aspect, we shall consider two simple example processes. These examples are so simple that controlling them by computer is completely unnecessary. This is deliberate: the examples must be computationally trivial, because we are not trying to make a point about the suitability of these formalisms for expressing computations. Lengthy computationally trivial examples do not illustrate the point any better than short ones. Both are special cases of machines that interact with their environments via a fixed number of sensors and actuators. The sensors measure, at various moments, certain environment variables; the actuators cause changes in the environment. The machines communicate with sensors and actuators by means of values. This paper does not deal with the conversion of (external) environment variables to (internal) values and of (internal) values to (external) effects, nor the actual position of sensors and actuators in physical space.

Two-way switch

The first example is a two-way switch: two push-buttons controlling one lamp. If either button is pressed, the lamp goes on or off as appropriate. What happens if both are pressed simultaneously is left unspecified; in other words, the implementor is free to do what seems most convenient. This is a so-called soft real-time process: it must react fast enough, but it does not really matter if it reacts an instance sooner or later. We would like that this process can be designed in a straightforward way and that from its design it can be derived how fast it will react and what will happen if both buttons are pressed simultaneously.

Three-minute switch

The other example is a three-minute switch. If one presses the switch, the lamp comes on, and three minutes later it will be switched off automatically. It is tempting to add some sort of external clock to the device, but doing so would essentially reduce this example to the previous one. So, instead of deferring the time-keeping to an external device, we want the switch to do its own timing. This is an example of a hard real time process: we have to meet the timing constraints in the specification exactly.

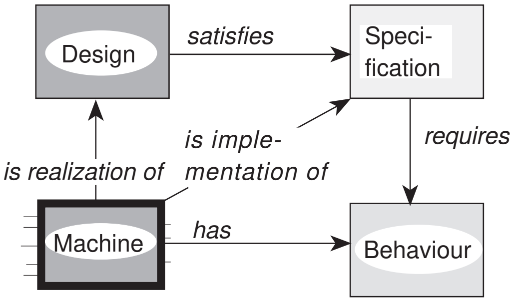
1.3. Adequacy criteria

The main point of this paper will be the definition of a compositional design language for process control machines and its semantics. A computable process then will be defined as a process that has a design in the language. This shall be done in such a way that the resulting notion of computable process meets the following criteria.

1. It must be a generalization of the notion of computable function.
2. It should be complete in the sense that everything digital computers can do can be modelled as computable processes.
3. It should allow straightforward design of self-timing hard real-time processes.
4. The design language should have nice algebraic transformation properties and it should be compositional with respect to functionality and timing.
5. On the other hand, computable processes should be as close to realistic hardware as, say, a register machine.

1.4. Specification, design, and behaviour

Suppose we need a certain machine M with behaviour B. For example, a machine that flies. We say that B is a behaviour that M *has*. Usually the behaviour B is not very specific and moreover the object M is a black box; we do not know how it works. If we want to construct an M having behaviour B, then we need an explanation of the property, the *specification*, and a *design* of the machine. Design and specification are syntactic objects, say expressions in a formal language. The machine is a physical object and its behaviour can be thought of as being an element of some mathematical description space. The design is a blueprint of the structure of the machine and tells us how it can be *realized*. The specification tells us which behaviour is *required*. The relation *satisfies* between design and specification must be defined such that the following diagram 'commutes': if a design satisfies a specification and a machine is a realization of that design, then that machine will be an implementation of the specification. If, furthermore, that specification requires a certain behaviour, then the machine will have that behaviour.



The four notions can be explained by an example in the theory of computable functions. The behaviour of such a function f can be 'squaring the input'; its specification is $f(x) = x^2$; its design is a program for f and a machine for f can be a calculator with just one button for squaring the input. (In this example the specification and the behaviour are hardly different. To see the difference, consider as behaviour 'computes for input n the n-th decimal in the number π'.)

There is another reason why the previous example is particular: the specification uniquely determines the required object. In general the scheme for specifying, designing and realizing (a needed act, a behaviour) is as follows.

1.4.1. Definition. A *setting* is a quintuple $(S, T, L, [\![\ ]\!], \mathrm{int})$ where T is a collection of terms (descriptions of the objects of interest), S is a mathematical space (in which the objects of interest live), and L is a logical language (providing predicates that define subsets of S, not just singletons) together with maps

$$\begin{aligned} [\![\ ]\!] &: T \to S; \\ \mathrm{int} &: L \to \{A \mid A \subseteq S\}. \end{aligned}$$

Here $[\![t]\!]$ is for $t \in T$ the intended interpretation; for $P \in L$ its interpretation $\mathrm{int}(P)$, also written as P^S, is the intended selection of elements of the space S.

Given such a setting, the job of specifying, designing and realizing becomes the following.

1. Specify a wanted property by finding a $P \in L$ coresponding to the intentions.
2. Design a $t \in T$ such that $P^S([\![t]\!])$.

Then one can construct a realization of t.

In the case of computable functions (with input and output elements of $\mathcal{N}$) the setting is as follows. The collection of terms consists of descriptions of recursive functions, the set S is the function space $\mathcal{N} \to \mathcal{N}$ (more precisely $\bigcup_{k \in \mathcal{N}} \mathcal{N}^k \to \mathcal{N}$), for $t \in T$ the function $[\![t]\!] \in S$ is the intended meaning of t, and $\mathrm{int}(P)$ for $P \in L$ is a set of functions with a desired property. The set of computable functions is defined as $\{[\![t]\!]\in S \mid t\in T\}$. For example, $t = \mathrm{REC}(t_1, t_2)$ is a term describing a function obtained by primitive recursion, $[\![t]\!]$ is that function, and

$$P(f) \Leftrightarrow \forall n[f(n) \text{ is divisible by } 1, \ldots, n].$$

is a possible specification. The function factorial ($f(n) = n!$) is one that satisfies the specification. This function can be defined by primitive recursion from known functions $[\![t_1]\!], [\![t_2]\!]$.

The main point of this paper is to provide for the notion of process a setting. This will be done in such a way that T is compositional. Moreover, there is a correspondence between $t\in T$ and realizations real-t preserving this compositionality. We do not pay special attention to specifications, except that some are given in a somewhat clumsy predicate logic fashion.

1.5. Processes

We consider processes with a fixed number of *input* and *output* ports. The number of these is arbitrary, but we do not consider processes that dynamically can add ports to itself. The behaviour of such a process can be modelled by *streams*, viz. sequences of values occurring at a port throughout time. A *process* is a function mapping m (input) streams to n (output) streams.

We shall only consider digital processes that operate in discrete time, much like the theory of computable functions only considers digitial functions that can be computed stepwise by means of algorithms.

In order to be able to reason about real-time problems we shall model time as the set of natural numbers. We denote this set as $\mathcal{T}$, as opposed to $\mathcal{N}$, which we use for the set of all values that can occur, at a given moment, at a given port.

We shall use the same global time reference for all expressions: in all expressions the same $t\in\mathcal{T}$ shall indicate the same moment ('synchronicity'). As a consequence, we shall not have trouble from problems that arise from weaker time models, like the paradox of Brock-Ackermann [1981].

We assume that at any moment, at any input or output port, exactly one value from N can be observed. This allows to model the behaviour at any port as a function of type $\mathcal{T}\to\mathcal{N}$.

1.5.1. Definition. (i) The set of *streams*, notation $\mathcal{S}$, is defined as $\mathcal{T}\to\mathcal{N}$.
(ii) A *scenario* is a tuple ξ of streams, i.e. an element of $\mathcal{S}^k$ for some $k\in\mathcal{N}$.

Notation. (i) The letters $x, y, z, \ldots$ stand for streams and $\xi, \psi, \zeta, \ldots$ for scenarios.
(ii) The value of x at $t\in\mathcal{T}$ is denoted by $x(t)$. If $\xi = (x_1, \ldots, x_k)$, then $\xi(t) = (x_1(t), \ldots, x_k(t))$.

A process now can be modelled as something that for a given input scenario causes an output scenario.

1.5.2. Definition. A *(synchronous digital) process* with k input ports and l output ports is a functional $P : \mathcal{S}^k\to\mathcal{S}^l$. We call $\mathcal{S}^k\to\mathcal{S}^l$ the *type* of P.

Now we turn to the problem of defining which processes are computable.

1.6. Discarded possibilities

Process control in daily life and industry is done by special (or even general) purpose computers. Since we do have these properly working systems, it may seem that the good old universal (Turing or) register machine is an adequate model for the execution of computable processes. Only a small adaptation would be necessary. In, say, a register machine some registers have to be reserved as input channels for the values coming from the sensors; other registers have to be reserved for the values to be given to the actuators. The real adaptation is that proper connections with the world (between these reserved I/O cnannels and the actual sensors and actuators) have to be established. Moreover, there should be no final state in the description of the register machine: if a subtask is fulfilled, then the machine should continue with the next one.

There is, however, a disadvantage in considering register machines as the underlying model for computable processes. The grainsize is not right. One is interested in *combining* processes in well defined ways. If one has to literally use for these combinations actual register machines (together with the adaptations mentioned above), then bulky objects are obtained. The possibility to describe processes as combinations of simple components will be lost.

Another possibility is to simulate compound processes in one single register machine. But then we loose for example the possibility of real parallellism. (It will be difficult, if not impossible, to predict time behaviour of a parallel process.) This is a particular instance of the general disadvantage, that in this way one has to model arbitrary processes by the internal ones of the (universal) machine. This results in general in a loss of internal structure of the process.

Similar difficulties occur when we try to describe processes using other models for the computable functions, like lambda calculus or term/graph rewrite systems. The moral is that the notion of computable process is more fundamental than that of computable function. The latter will have to be described using the former.

1.7. Other work

The presentation of the computable processes in this paper is simple but not as well-known as that of, say, a Turing machine. The definition is possibly not new. Its constituent concepts, like e.g. the constructors for processes, can be found in several places. But little attention has been paid to the choice of primitive processes. In our presentation emphasis is put on the difference between specification and design of a process. The process itself is identified with its behaviour. (This is similar to the way functions are treated in naive settheory; these are seen as the graph of their input/output relation.) Specification is given in terms of the semantics and design in terms of the (syntactically) composed basic units.

We only treat digital processes. For analog processes, see e.g. Boute [1986, 1988, 1990], where unidirectional digital processes are regarded as special cases of bidirectional analogue circuits. In this approach a clock is introduced 'in order to overcome physical problems'. We, on the other hand, discuss a generalization of the notion of algorithm that forms the basis of *reactive programming*.

We treat only synchronous processes, because whenever digital machines are communicating, synchronization takes place at some low level. It is this level of granularity that we are describing.

The computable processes we consider are deterministic. By adding extra primitive processes, the theory could be extended to cover also the notion of non-deterministic processes. Better, however, is to realize that the notion of non-determinism is a feature of specifications, namely those that do not uniquely determine the process.

There are several other approaches to processes. In Process Algebra, see e.g. Bergstra and Klop [1986], a process is seen as a structured collection of events. This is a higher level description. The strength of this theory is that nothing is required about the nature of the atomic processes and that several semantic interpretations are possible. It does not speak about computable processes. Our atomic processes are very concrete and it is immediate how to build hardware devices or programs that perform the given processes. We do specify our processes in first order predicate logic. In many cases, formalisms like process algebra turn out to be a better tool to specify them. We have not yet grasped the exact way to do this or to prove that certain designs satisfy a given specification. In Kahn [1976] and Moschovakis [1989, 1990] another definition of the higher level processes is given.

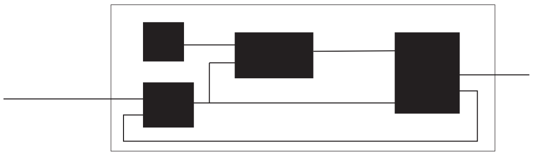
It should be emphasized that our treatment of processes considers communication at a lower, more hardware-oriented level than these approaches and many other ones. Our streams are synchronous, and a defined value is available at every moment in real time. Higher level concepts like 'fairness' or 'buffering' are not relevant within our framework. A buffer or a fair merge, however, can be modelled as a computable process.

Other related work is by Ştefănescu [1987, 1985] and Berry [1989, 1993]. In the first series of articles similar process constructors are used, but a diferent semantics is given. (It is a flow-chart semantics where 'the action' is always localized at one spot. In our diagrams below, action is everywhere.) In the second series of papers the semantics of processes is the same as in this paper, but the description of them is different, namely by systems of equations. Also of interest are Broy [1992, 1992a] and Stark [1992].

2. Simple processes

In this and the next section we will introduce the formal notion of *process control machine*, also called *box*, and *(process) combinator*. Also a semantics of these in the space of processes will be given (remember that a process is a functional on scenarios, i.e. vectors of streams). A box is a two-dimensional description of a machine that 'runs' a process P. Combinators are in a 1-1 correspondence with boxes and are (one dimensional) formulas convenient to manipulate, and reason about the (design of the) corresponding processes. The collection of combinators is the design language $\mathcal{T}$ for processes. The boxes are the *realizations* of the combinators. A combinator M (or its corresponding box real-M) has as semantics a process $P = [\![M]\!](= [\![\text{real-}M]\!])$.

Boxes are suggestive provided there are not too many of them. Therefore they may be used to describe a machine under a certain abstraction: some of the boxes are 'black boxes' and are filled in later.



2.1. Formal description

2.1.1. Definition. (i) A (formal) *type* is of the form $\mathcal{S}[n, m]$, with $n, m \in \mathcal{N}$. The intended meaning of n, m is the number of input and output ports, respectively; the $\mathcal{S}$ indicates that the input and output consist of streams.

(ii) *Combinators* are expressions with a certain type. If a combinator M has as type $\mathcal{S}[n, m]$ we write $M{:}\mathcal{S}[n, m]$.

(iii) There are six *primitive combinators*:

$$\begin{array}{lcll} \mathsf{I} & : & \mathcal{S}[1,1] & \text{(identity)}, \\ \mathsf{L} & : & \mathcal{S}[1,0] & \text{(sink)} \\ \mathsf{O} & : & \mathcal{S}[0,1] & \text{(zero)}, \\ \mathsf{S} & : & \mathcal{S}[1,1] & \text{(successor)}, \\ \mathsf{E} & : & \mathcal{S}[2,1] & \text{(equality test)}, \\ \mathsf{C} & : & \mathcal{S}[3,1] & \text{(conditional)}. \end{array}$$

(iv) *Simple combinators* are defined from the primitive combinators using the constructors $|$, $\cdot$, and $\wedge$. The definition is as follows.

(1) Each primitive combinator is a simple combinator.

(2) If $M{:}\mathcal{S}[n, m], N{:}\mathcal{S}[n', m']$ are simple combinators, then so is

$$M|N \quad : \quad \mathcal{S}[n + n', m + m'] \qquad \text{(parallel composition)}.$$

(3) If $M{:}\mathcal{S}[n, m], N{:}\mathcal{S}[m, k]$ are simple combinators, then so is

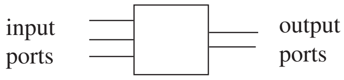
$$M \cdot N \quad : \quad \mathcal{S}[n, k] \qquad \text{(serial composition)}.$$

(4) If $M{:}\mathcal{S}[n, m], N{:}\mathcal{S}[n, k]$ are simple combinators, then so is

$$M \wedge N \quad : \quad \mathcal{S}[n, m + k] \qquad \text{(input sharing)}.$$

Simple combinators can be obtained only through (1)-(4), see toolkit below.

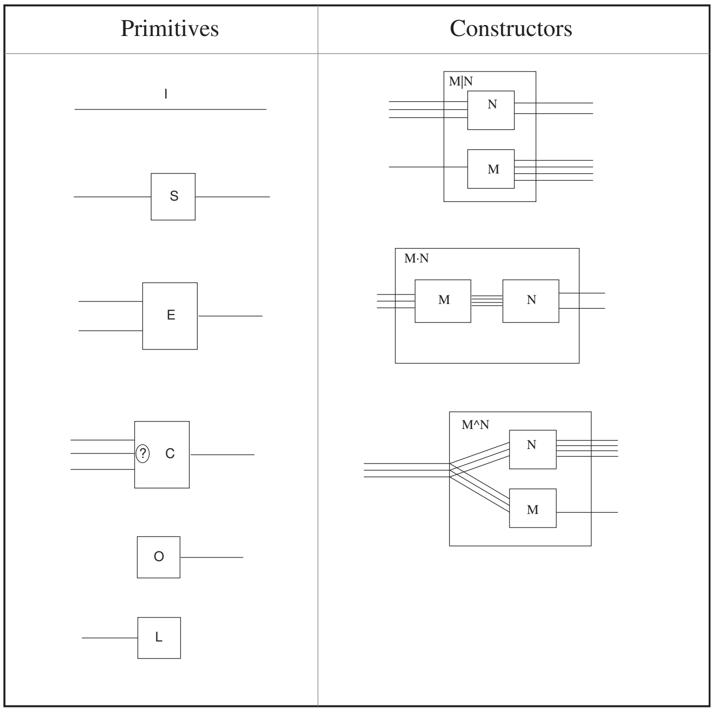
2.1.2. DEFINITION. (i) A *box* (or *process control machine*) of type $\mathcal{S}[n,m]$ is a device with n input ports and m output ports.



In all diagrams in this paper input ports are drawn on the left of a box; moreover, ports are numbered from below, starting with 1.

(ii) The *behaviour* of a box is that it 'runs' a process. There is a global clock. At every tick of the clock a token (an element of $\mathcal{N}$, the set of natural numbers) will be absorbed at each input port. Moreover, at every tick of the clock a token will be produced at each output port.

(iii) Boxes corresponding to the simple combinators will be depicted as two-dimensional pictures. They can be combined following the constructors of simple combinators. For every combinator M, the corresponding box will be denoted by real-M.



Toolkit for simple processes.

2.1.3. DEFINITION. If $M{:}\mathcal{S}[n,m]$ is a simple combinator, then its *semantics* will be defined as a process $\llbracket M \rrbracket : \mathcal{S}^n \to \mathcal{S}^m$. The definition is inductive.

(i) The semantics of the primitive combinators is as follows.

$$\begin{array}{rcll}
\llbracket \mathsf{I} \rrbracket(x)(t) & = & x(t); & \\
\llbracket \mathsf{O} \rrbracket(x)(t) & = & 0; & \\
\llbracket \mathsf{L} \rrbracket(x)(t) & = & (); & \\
\llbracket \mathsf{S} \rrbracket(x)(0) & = & 0; & \\
\llbracket \mathsf{S} \rrbracket(x)(t+1) & = & x(t)+1; & \\
\llbracket \mathsf{E} \rrbracket(x,y)(0) & = & 0; & \\
\llbracket \mathsf{E} \rrbracket(x,y)(t+1) & = & 1, & \text{if } x(t)=y(t), \\
\llbracket \mathsf{E} \rrbracket(x,y)(t+1) & = & 0, & \text{else}; \\
\llbracket \mathsf{C} \rrbracket(x,b,y)(0) & = & 0; &
\end{array}$$

$$
\begin{aligned}
[\![\mathsf{C}]\!](x,b,y)(t+1) &= x(t), &\text{if } b(t)=0;\\
[\![\mathsf{C}]\!](x,b,y)(t+1) &= y(t), &\text{if } b(t)\neq 0.
\end{aligned}
$$

(ii) If $M{:}\mathcal{S}[n,m]$ and $N{:}\mathcal{S}[n',m']$, then $[\![M|N]\!] : \mathcal{S}^{n+n'} \to \mathcal{S}^{m+m'}$ is defined by[3]

$$[\![M|N]\!](\xi,\psi)(t) = ([\![M]\!](\xi)(t), [\![N]\!](\psi)(t)).$$

(iii) If $M{:}\mathcal{S}[n,m]$ and $N{:}\mathcal{S}[m,k]$, then $[\![M\cdot N]\!] : \mathcal{S}^{n} \to \mathcal{S}^{k}$ is defined by

$$[\![M\cdot N]\!](\xi) = [\![N]\!]([\![M]\!](\xi)).$$

(iv) If $M{:}\mathcal{S}[n,m]$ and $N{:}\mathcal{S}[n,m']$, then $[\![M \wedge N]\!] : \mathcal{S}^{n} \to \mathcal{S}^{m+m'}$ is defined by

$$[\![M \wedge N]\!](\xi) = ([\![M]\!](\xi), [\![N]\!](\xi)).$$

2.1.4. Definition. A *simple process* is a process P (of some type) such that for some simple combinator M one has $P = [\![M]\!]$.

2.1.5. Definition. Two combinators M, N (of the same type) are *equivalent*, notation $M \cong N$, if $[\![M]\!] = [\![N]\!]$.

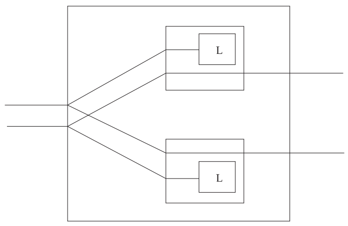
For example $\mathsf{I} \cong \mathsf{I}\cdot\mathsf{I}$.

2.2. Examples

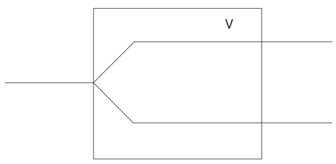
Topological circuits

These are for the proper 'wiring' of more interesting combinators.

2.2.1. Example. Cross over

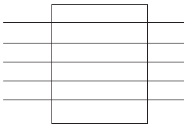
<table>
<tr><td colspan="2">Specification $\mathsf{X}(x,y)=(y,x)$</td></tr>
<tr><td>Design
$\mathsf{X} = (\mathsf{L}|\mathsf{I}) \wedge (\mathsf{I}|\mathsf{L})$</td><td>Box

</td></tr>
</table>

2.2.2. Example. Split

<table>
<tr><td colspan="2">Specification $\mathsf{V}(x)=(x,x)$</td></tr>
<tr><td>Design
$\mathsf{V} = \mathsf{I} \wedge \mathsf{I}$</td><td>Box

</td></tr>
</table>

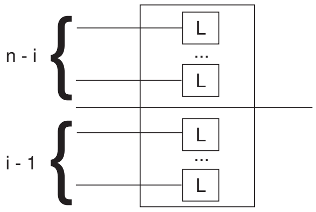
[3] If $\xi = (x_1,\ldots,x_n)$ and $\psi = (y_1,\ldots,y_{n'})$, then $(\xi,\psi) = (x_1,\ldots,x_n,y_1,\ldots,y_{n'})$. This notation holds for vectors of arbitrary but equal types. The space $\mathcal{S}^m = (\mathcal{T}\to\mathcal{N})^m$ is identified with $(\mathcal{T}\to\mathcal{N}^m)$.

2.2.3. Example. Cable

Specification	$\mathsf{B}_5(x_1, \ldots, x_5) = (x_1, \ldots, x_5)$
Design	Box
$\mathsf{B}_5 = \mathsf{I} \mid \mathsf{I} \mid \mathsf{I} \mid \mathsf{I} \mid \mathsf{I}$	

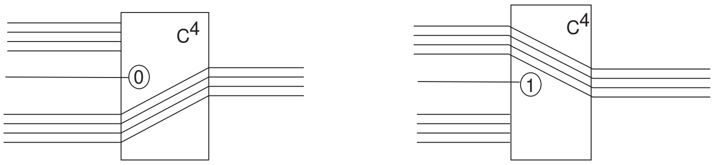
If one needs the i-th wire out of n one can use the following combinator.

2.2.4. Example. Selection

Specification	$\mathsf{P}_i^n(x_1, \ldots, x_n) = x_i$
Design	$\mathsf{P}_i^n = (\mathsf{L} \mid \mid (i-1)) \mid \mathsf{I} \mid (\mathsf{L} \mid \mid (n-i))$
Box	

Switches

One can make multiple switches like C^4 that switches cables of four lines.



4-fold selector switch.

We first design a double switch C^2.

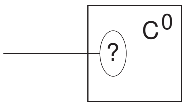
$$\mathsf{C}^2 = ((\mathsf{I} \mid \mathsf{L} \mid \mathsf{I} \mid \mathsf{I} \mid \mathsf{L}) \cdot \mathsf{C}) \wedge ((\mathsf{L} \mid \mathsf{I} \mid \mathsf{I} \mid \mathsf{L} \mid \mathsf{I}) \cdot \mathsf{C}).$$

We can rewrite this using selectors. $\mathsf{C}^2 = ((\mathsf{P}_1^2 \mid \mathsf{I} \mid \mathsf{P}_1^2) \cdot \mathsf{C}) \wedge ((\mathsf{P}_2^2 \mid \mathsf{I} \mid \mathsf{P}_2^2) \cdot \mathsf{C})$. Now the general pattern becomes clear.

2.2.5. Example. Multi-switch

Specification	$\mathsf{C}^n(\xi, b, \xi')(t+1) \;=\; \xi(t)$ if $b(t) = 0$; $\;=\; \xi'(t)$ else.
Design	$\mathsf{C}^n = ((\mathsf{P}_1^n \mid \mathsf{I} \mid \mathsf{P}_1^n) \cdot \mathsf{C}) \wedge \ldots \wedge ((\mathsf{P}_n^n \mid \mathsf{I} \mid \mathsf{P}_n^n) \cdot \mathsf{C})$

Consequently, C^1 must be $((\mathsf{I} \mid \mathsf{I} \mid \mathsf{I}) \cdot \mathsf{C})$, and this indeed is equivalent to C itself. Furthermore, C^0 should be the neutral element of $\wedge$, viz. L. This does conform to our intuition.



A 0*-fold selector switch is the same as* L*.*

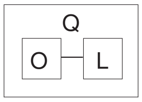
Making use of these topological cirquits, in section 4 we will use wiring, without explicitly writing down the components. But in the combinators we always will be explicit about the topological components.

Generalized composition

The arguments for the two constructors $\cdot$ and $^\wedge$ do not always 'fit'. This by contrast to those of $|$. By extending the definitions of $\cdot$ and $^\wedge$ they will be made fit to all input types. Iterations of the three constructors will be defined and some simple laws are stated.

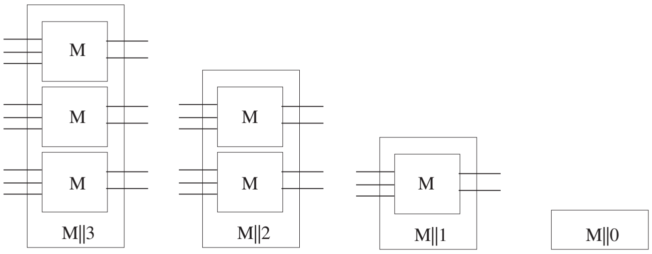
A funny combinator is the following; it has no input or output ports.

2.2.6. Example. The inert process

Specification $\mathsf{Q} : \mathcal{S}[0,0]$	
Design $\mathsf{Q} = \mathsf{O} \cdot \mathsf{L}$	Box 

Using Q one can define iterated parallel composition. We want to have a convenient notation for combinators like $M|M|M$. This one will be denoted as $M||3$.

2.2.7. Definition. (i) $M||0 = \mathsf{Q}$
(ii) $M||(n+1) = M|(M||n)$.



Iterated parallel composition

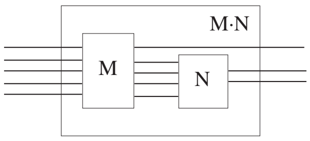
Note that we have the following.

2.2.8. Proposition. (i) $M|(N|L) \cong (M|N)|L$.
(ii) $M|\mathsf{Q} \cong \mathsf{Q}|M \cong M$.

In general one has $M|N \ncong N|M$.

The operation of sequential composition $\cdot$ is not always applicable to arbitrary combinators. It is convenient to extend it so that one can also use it if the number of pins do not match, see Meertens [1989].

2.2.9. Example.

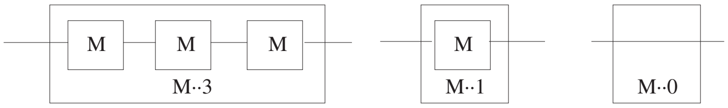
Specification $\cdot : [\mathcal{S}^k \to \mathcal{S}^{l+n}] \times [\mathcal{S}^l \to \mathcal{S}^m] \to [\mathcal{S}^k \to \mathcal{S}^{m+n}]$	
Design $M \cdot N = M \cdot (N|(\mathsf{I}||n))$	Box 

2.2.10. Example.

Specification	$\cdot : [\mathcal{S}^k \rightarrow \mathcal{S}^l] \times [\mathcal{S}^{l+n} \rightarrow \mathcal{S}^m] \rightarrow [\mathcal{S}^{k+n} \rightarrow \mathcal{S}^m]$
Design	$M \cdot N = (M\vert(\mathsf{I}\vert\vert n)) \cdot N$

Also we want to have a notation for iterated serial composition. For example $M \cdot M \cdot M$ shall be denoted by $M {\cdot\cdot} 3$.

2.2.11. Definition. (i) $M {\cdot\cdot} 0 = \mathsf{L}$.
(ii) $M {\cdot\cdot} (n+1) = M \cdot (M {\cdot\cdot} n)$.

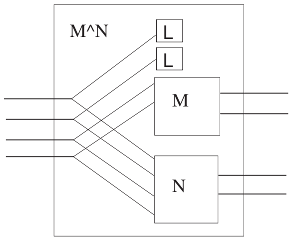


Iterated serial composition

2.2.12. Proposition. (i) $M \cdot (N \cdot L) \cong (M \cdot N) \cdot L$.
(ii) $M \cdot \mathsf{I} \cong \mathsf{I} \cdot M \cong M$.

The operation ^ can also be extended to combinators that do not match.

2.2.13. Example.

<table>
<tr><td colspan="2">Specification $\wedge : [\mathcal{S}^{k+n} \rightarrow \mathcal{S}^l] \times [\mathcal{S}^k \rightarrow \mathcal{S}^m] \rightarrow [\mathcal{S}^{k+n} \rightarrow \mathcal{S}^{l+m}]$</td></tr>
<tr><td>Design
$M \wedge N = M \wedge (N\vert(L\vert\vert n))$</td><td>Box

</td></tr>
</table>

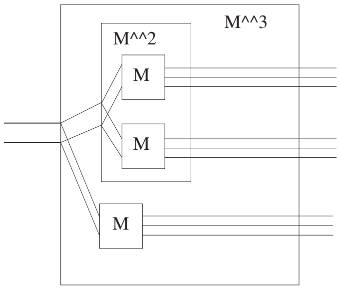
2.2.14. Example.

Specification	$\wedge : [\mathcal{S}^k \rightarrow \mathcal{S}^l] \times [\mathcal{S}^{k+n} \rightarrow \mathcal{S}^m] \rightarrow [\mathcal{S}^{k+n} \rightarrow \mathcal{S}^{l+m}]$
Design	$M \wedge N = (M \wedge (\mathsf{L}\vert\vert n)) \wedge N$

Also we want to have a notation for iterated input sharing. For example $M \wedge M \wedge M$ is denoted by $M{\wedge\wedge}3$.

2.2.15. Definition. (i) $M{\wedge\wedge}0 = \mathsf{I}$.
(ii) $M{\wedge\wedge}(n+1) = M \wedge (M{\wedge\wedge}n)$.

2.2.16. Example. Iterated input sharing

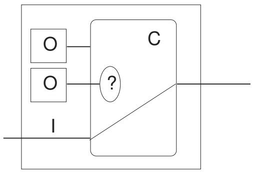
Specification $M{\wedge}{\wedge}3(\xi) = (M(\xi), M(\xi), M(\xi))$	
Design	Box
$M{\wedge}{\wedge}3 = M \wedge M \wedge M$	

2.2.17. Proposition. (i) *$M \wedge (N \wedge L) \cong (M \wedge N) \wedge L$.*
(ii) *$M \wedge L \cong L \wedge M \cong M$.*

Delay

In hard real-time systems, if results of a computation are available earlier than desirable they must be kept in a delay element until they are needed. The most elementary one holds 1 value for 1 time unit.

2.2.18. Example. One step delay

Specification $\mathsf{D}(x)(0) = 0,\ \mathsf{D}(x)(t+1) = x(t)$	
Design	Box
$\mathsf{D} = (\mathsf{I}\vert\mathsf{O}\vert\mathsf{O}) \cdot \mathsf{C}$	

As an example we show how a formal proof can be given using the notation of combinators and their semantics. As is common practice we write M for $[\![M]\!]$.

2.2.19. Proposition. (i) *$\mathsf{D}(x)(0) = 0$.*
(ii) *$\mathsf{D}(x)(t+1) = x(t)$.*

Proof. (i) $\mathsf{D}(x)(0) = ((\mathsf{I}|\mathsf{O}|\mathsf{O}) \cdot \mathsf{C})(x)(0)$

$$
\begin{aligned}
&= (\mathsf{C}((\mathsf{I}|\mathsf{O}|\mathsf{O})(x)))(0)\\
&= (\mathsf{C}(\mathsf{I}(x), \mathsf{O}(), \mathsf{O}()))(0)\\
&= 0.
\end{aligned}
$$

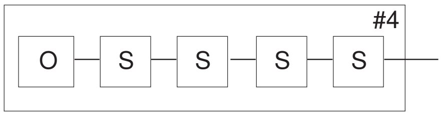
(ii) $$
\begin{aligned}
\mathsf{D}(x)(t+1) &= (\mathsf{C}(x, \mathsf{O}(), \mathsf{O}()))(t+1)\\
&= x(t). \blacksquare
\end{aligned}
$$

Extended delay combinators can be defined easily.

2.2.20. Proposition. (i) $\mathsf{D}||m : \mathcal{S}[m, m]$ *preserves m values during one time unit.*
(ii) $\mathsf{D}{\cdot\cdot}n : \mathcal{S}[1, 1]$ *preserves one value during n time units.*
(iii) $(\mathsf{D}||m){\cdot\cdot}n : \mathcal{S}[m, m]$ *preserves m values during n time units.*

Constants

2.2.21. EXAMPLE. A box $\#n$ with constant output n can be defined as follows. $\#0 = \mathsf{O}$ and $\#(n+1) = \#n \cdot \mathsf{S} = \mathsf{O} \cdot (\mathsf{S}{\cdot\cdot}n)$.



The constant stream 4.

Logic

2.2.22. EXAMPLE. Booleans and logical gates can be defined as combinators.

$$\begin{aligned}
\mathsf{true} &= \#0;\\
\mathsf{false} &= \#1;\\
\mathsf{and} &= ((\mathsf{I} \wedge \mathsf{I})|\mathsf{I}) \cdot \mathsf{C};\\
\mathsf{or} &= (\mathsf{I}|(\mathsf{I} \wedge \mathsf{I})) \cdot \mathsf{C};\\
\mathsf{not} &= (\mathsf{false}|\mathsf{I}|\mathsf{true}) \cdot \mathsf{C}.
\end{aligned}$$

The names are justified, since for example

$$\mathsf{and}(x,y)(t) = \text{ if } x(t) = 1 \ \& \ y(t) = 1 \text{ then } 1 \text{ else } 0.$$

Morover, we have e.g. the following equivalences.

2.2.23. PROPOSITION. (i) $\mathsf{true} \cdot \mathsf{not} \cong \mathsf{false}$.
(ii) $(\mathsf{true}|\mathsf{false}) \cdot \mathsf{or} \cong \mathsf{true}$.

2.3. Properties of simple processes

NOTATION. (i) If $f : \mathcal{N}^n \to \mathcal{N}^m$ is a function, then we write $f : \mathcal{N}[n,m]$. If $n = 0$, then f is identified with a constant.

(ii) In order to avoid confusion between processes and functions we use lower case letters $f, g, h, \ldots$ for functions and upper case letters $M, N, L, \ldots$ for processes.

The usual definition of recursivity is for functions of type $\mathcal{N}[n,1]$. This can be extended in the obvious way.

2.3.1. DEFINITION. (i) Every $f : \mathcal{N}[0,m]$ is called *recursive.*

(ii) A function f of type $\mathcal{N}[n,m]$ is called *recursive* if for all $\vec{a} = (a_1, \ldots, a_n)$ one has

$$f(\vec{a}) = (g_1(\vec{a}), \ldots, g_m(\vec{a}))$$

for some ordinary recursive functions $g_1, \ldots, g_m : \mathcal{N}[n,1]$.

In order to reason about the past of streams the following definition is useful.

2.3.2. DEFINITION. (i) Let $x \in \mathcal{S}$ be a stream and $t, t' \in \mathcal{T} (= \mathcal{N})$. Then

$$\begin{aligned}
x(t{\cdot\cdot}t') &= \langle x(t), x(t+1), \ldots, x(t') \rangle^4 && \text{if } t \leq t';\\
&= \langle\rangle && \text{if } t > t'.
\end{aligned}$$

(ii) Let $\xi = (x_1, \ldots, x_n) \in \mathcal{S}^n$ be a scenario. Then we write

$$\xi(t{\cdot\cdot}t') = (x_1(t{\cdot\cdot}t'), \ldots, x_n(t{\cdot\cdot}t')).$$

2.3.3. DEFINITION. Let P be a process (of type $\mathcal{S}[n,m]$) and let $d{\in}\mathcal{T}$.

(i) *P is determined by the last d (more accurately $d+1$) steps* (of the input), notation $L_d(P)$, if

$$\exists f{:}\mathcal{N}[n,m]\forall\xi{\in}\mathcal{S}^n\forall t{\in}\mathcal{T}\ P(\xi)(t) = f(\xi((t \dot{-} d){\cdot\cdot}t))^5$$

(ii) P is *recursively* determined by on the last d steps (of the input), notation $L_d^{\mathrm{rec}}(P)$, if f in (i) is required to be recursive.

2.3.4. PROPOSITION. *For every simple process P there exists a $d{\in}\mathcal{T}$ such that $L_d^{\mathrm{rec}}(P)$.*

PROOF. Let $P = [\![M]\!]$. By induction on the complexity of M we show that d exists.

For the primitive combinators M one clearly has $L_1^{\mathrm{rec}}(M)$. (If $M = \mathsf{I}$, then even $d = 0$ suffices.) If $L_{d_1}^{\mathrm{rec}}(M_1)$ and $L_{d_2}^{\mathrm{rec}}(M_2)$, then $L_{\max\{d_1,d_2\}}^{\mathrm{rec}}(M_1|M_2)$, $L_{d_1+d_2}^{\mathrm{rec}}(M_1 \cdot M_2)$, and $L_{\max\{d_1,d_2\}}^{\mathrm{rec}}(M_1 \wedge M_2)$. ■

A corollary is that simple processes are invariant under time shift modulo warming up.

2.3.5. DEFINITION (Time shift). (i) Let x be a stream. Then the time shift of x, notation $x{\hookrightarrow}{\in}\mathcal{S}$, is defined by $x{\hookrightarrow}(t) = x(t+1)$.

(ii) If $x{\in}\mathcal{S}$, then we define for $n{\in}\mathcal{T}$ the *n-shift* of x, notation $x{\hookrightarrow}^n$, as follows.

$$\begin{aligned} x{\hookrightarrow}^0 &= x \\ x{\hookrightarrow}^{n+1} &= (x{\hookrightarrow}^n){\hookrightarrow}. \end{aligned}$$

(iii) If $\xi = (x_1, \ldots, x_k)$ is a scenario, then

$$\xi{\hookrightarrow}^n = (x_1{\hookrightarrow}^n, \ldots, x_k{\hookrightarrow}^n).$$

2.3.6. DEFINITION. (i) Let $\xi, \eta{\in}\mathcal{S}^k$ be scenarios. Then ξ and η are *eventually equal*, notation $\xi \simeq \eta$, if

$$\exists d{\in}\mathcal{T}\forall t{\geq}d\ \xi(t) = \eta(t).$$

(ii) Let P, Q be two processes. Then P and Q are *equal up to warming up*, notation $P \simeq Q$, if

$$\forall\xi\ P(\xi) \simeq Q(\xi).$$

Notice that $P \simeq Q \ \Leftrightarrow\ \exists d\ \mathsf{P}{\hookrightarrow}^d = Q{\hookrightarrow}^d \ \Leftrightarrow\ \exists d\ \mathsf{P}{\hookrightarrow}^d \simeq Q{\hookrightarrow}^d$.

2.3.7. COROLLARY. *Let P be a simple process. Then we have the following.*

(i) *P is invariant under time shift modulo warming up, i.e. for all ξ*

$$P(\xi{\hookrightarrow}^n) \simeq (P(\xi)){\hookrightarrow}^n.$$

(ii) $\xi \simeq \eta \ \Rightarrow\ P(\xi) \simeq P(\eta)$.

PROOF. (i) Assume by the proposition $L_d(P)$, i.e. for some f

$$\forall\xi\forall t\ P(\xi)(t) = f(\xi((t \dot{-} d){\cdot\cdot}t)).$$

By induction[6] it suffices to show that $P(\xi{\hookrightarrow}) \simeq (P(\xi)){\hookrightarrow}$. Now

$$\begin{aligned} P(\xi{\hookrightarrow}(t)) &= f(\xi{\hookrightarrow}((t \dot{-} d){\cdot\cdot}t)) \\ &= f(\xi(t \dot{-} d + 1, t+1)). \\ P(\xi){\hookrightarrow}(t) &= P(\xi)(t+1) \\ &= f(\xi((t+1 \dot{-} d){\cdot\cdot}(t+1))). \end{aligned}$$

But $t \dot{-} d + 1 = t + 1 \dot{-} d$ if $t \geq d$.

[4] We use the notation of Kleene [1952]: $\langle n_0, \ldots, n_k\rangle{\in}\mathcal{N}$ is the coded sequence $(n_0, \ldots, n_k)$ and $\langle\rangle$ is the code for the empty sequence.

[5] Here $t \dot{-} d$ denotes cut-off subtraction: $t \dot{-} d =$ if $t \geq d$ then $t - d$ else 0.

[6] Use the fact that for scenarios ξ, η one has $\xi \simeq \eta \ \Rightarrow\ \xi{\hookrightarrow} \simeq \eta{\hookrightarrow}$.

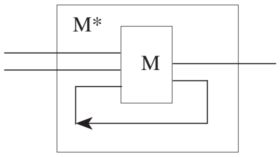
(ii) By (i). ∎

An easy counterexample shows that simple processes are not invariant under time shift absolutely. Take for example $P = [\![\mathsf{S}]\!]$ and as input $x = \lambda t.1$. Now $P(x)(0) = 0, P(x)(t+1) = 2$, and $P(x)\hookrightarrow = \lambda t.2$. But $P(x\hookrightarrow) = P(x) \neq \lambda t.2$.

3. Computable processes

3.1. Feedback

In this subsection processes with *feedback* will be introduced by connecting in a box B of, say, type $\mathcal{S}[3,2]$ the output port 0 to the input port 0. The result is a process of type $\mathcal{S}[2,1]$. The resulting box will be denoted by B^* and will be drawn as



Box with feedback

If $P = [\![B]\!]$ then we want that $P^* = [\![B^*]\!]$ satisfies $P^*(\xi) = x$ with $P(z,\xi) = (z,x)$ for some appropriate stream z. We first have to show that such feedback streams z always exist. For this some theory has to be developed.

Notation. Let $\xi \in \mathcal{S}^n$ of the form $\xi = (x_0, \ldots, x_{n-1})$ be a scenario. Then for $i<n$ we write $\xi_i = x_i$. Let, moreover, $z \in \mathcal{S}$. Then
$(z,\xi) = (z, x_0, \ldots, x_{n-1}) \in \mathcal{S}^{n+1}$.

3.1.1. Definition. Let $P{:}\mathcal{S}[n,m]$ be a process.

1. Let $i<n, j<m$. In P (output port) j is *directly connected to* (input port) i if

$$\forall \xi \forall t \in \mathcal{T}\ P(\xi)_j(t) = \xi(t)_i.$$

2. In P (output port) j *needs recursive processing* if there exists a recursive function $g{:}\mathcal{S}[n,1]$ such that

$$\begin{array}{rcl} P(\xi)_j(0) & = & 0 \\ P(\xi)_j(t+1) & = & g(\xi(0 \cdot\cdot t)). \end{array}$$

3. Process P is *regular* if for every $j<m$ either there exists a (unique) $i<n$ to which it is directly connected, or j needs recursive processing.

3.1.2. Lemma. (i) *The primitive combinators define regular processes.*
(ii) *The regular processes are closed under parallel composition, serial composition and input sharing.*

Proof. (i) Let M be a primitive combinator and $[\![M]\!]$ its corresponding process. Then the output port always needs recursive processing, except if $M = \mathsf{I}$, in which case the output port is directly connected to the input port.

(ii) Suppose that Q_1, Q_2 are regular processes and that P is composed of Q_1, Q_2 in order to show that P is regular.

Case 1. $P = Q_1 | Q_2$. Let j be an output port of P. Then it corresponds to an output port of, say, Q_1. By the induction hypothesis this port either needs recursive processing (in Q_1, but then also in P) or it is directly connected to an input port (of Q_1, but then also of P).

Case 2. $P = Q_1 \cdot Q_2$. Let j be an output port of P. Then it corresponds to an output port of Q_2. By the induction hypothesis this port either needs recursive processing (in Q_2, but then also in P) or it is directly connected to an input port j' of Q_2. In the second case this j' corresponds to an output port of Q_1. Again by the induction hypothesis this output port either needs recursive processing or it is directly connected to an input port i of Q_1. In the first case also j in P needs recursive processing. In the second case j in P is directly connected to i.

Case 3. $P = Q_1 \,^\wedge\, Q_2$. Similar to case 1. ∎

3.1.3. Corollary. *Every simple process is regular.*

Proof. By the lemma. ∎

The following definition is not intended as a characterization of those streams that are computable by boxes from scenarios. It gives an 'approximation' from above and will be a useful invariant.

3.1.4. Definition. Let $\xi \in \mathcal{S}^n$ be a scenario and let $x \in \mathcal{S}$ be a stream. Then x is *box computable from* ξ, notation $x \in \mathrm{BC}(\xi)$, if either for some $i < n$

1. $x = \xi_i$;

or for some recursive function $f{:}\mathcal{N}[1,1]$ and all $t \in \mathcal{T}$

2. $$\begin{aligned} x(0) &= 0; \\ x(t+1) &= f(\xi(0 \cdot\cdot t)). \end{aligned}$$

In this terminology a process is regular iff for all ξ, j one has

$$P(\xi)_j \in \mathrm{BC}(\xi).$$

3.1.5. Lemma. (i) *Let $\xi \in \mathcal{S}^n$ be a scenario and x, y be streams. Then*

$$x \in \mathrm{BC}(\xi) \;\&\; y \in \mathrm{BC}(x, \xi) \;\Rightarrow\; y \in \mathrm{BC}(\xi).$$

(ii) *Let $\xi \in \mathcal{S}^n$ be a scenario and let $z \in \mathcal{S}$ be defined by*

$$\begin{aligned} z(0) &= 0; \\ z(t+1) &= g(z(0 \cdot\cdot t), \xi(0 \cdot\cdot t)), \end{aligned}$$

for some recursive $g{:}\mathcal{N}[n{+}1, 1]$. Then $z \in \mathrm{BC}(\xi)$.

Proof. For notational convenience we assume that $n = 1$. The general proof is analogous.

(i) Assume $x \in \mathrm{BC}(\xi)$ & $y \in \mathrm{BC}(x, \xi)$. If $x = \xi_i$, or $y = x$ or $y = \xi_{i'}$, then one easily has $y \in \mathrm{BC}(\xi)$. Suppose on the other hand

$$\begin{aligned} x(0) &= 0; \\ x(t+1) &= f(\xi(0 \cdot\cdot t)). \\ y(0) &= 0; \\ y(t+1) &= g(x(0 \cdot\cdot t), \xi(0 \cdot\cdot t)). \end{aligned}$$

Without loss of generality we may assume that $f(\langle\rangle) = 0$. By a course of value recursion there exists a recursive function $\tilde{f}{:}\mathcal{N}[1,1]$ such that

$$\begin{aligned} \tilde{f}(\langle\rangle) &= \langle\rangle; \\ \tilde{f}(\alpha * \langle n \rangle) &= \tilde{f}(\alpha) * \langle f(\alpha) \rangle.^7 \end{aligned}$$

Then one easily verifies that for all $t \in \mathcal{T}$ one has $x(0 \cdot\cdot t) = \tilde{f}(\xi(0 \cdot\cdot t))$. Therefore

$$\begin{aligned} y(t+1) &= g(x(0 \cdot\cdot t), \xi(0 \cdot\cdot t)) \\ &= g(\tilde{f}(\xi(0 \cdot\cdot t)), \xi(0 \cdot\cdot t)) \\ &= h(\xi(0 \cdot\cdot t)), \end{aligned}$$

with $h(t) = g(\tilde{f}(t), t)$. It follows that $y \in \mathrm{BC}(\xi)$.

---

[7] If $\alpha = \langle n_0, \ldots, n_k \rangle$, then $\alpha * \langle n \rangle = \langle n_0, \ldots, n_k, n \rangle$.

(ii) By a similar technique one can show that for some recursive $\tilde{g}$ one has $z(0 \cdot\cdot t) = \tilde{g}(\xi(0 \cdot\cdot t))$. Then it follows that $z(t) = g'(\xi(0 \cdot\cdot t))$, for some recursive g', by taking the last component of $z(0 \cdot\cdot t)$. Since $z(0) = 0$, it follows that $z \in \mathrm{BC}(\xi)$. ■

3.1.6. LEMMA. *Let $P{:}\mathcal{S}[n+1, m+1]$ be a regular process.*
(i) *For all $\xi \in \mathcal{S}^n$ there exists a $z \in \mathrm{BC}(\xi)$ such that*
$$P(z, \xi)_0 = z. \tag{3.1}$$
(ii) *The stream z satisfying (3.1) is uniquely determined by ξ, except when in P the output port 0 is directly connected to 0. In that case for all z, ξ one has (3.1).*

PROOF. (i) Since P is regular, either the output port 0 needs recursive processing or 0 is directly connected to some input port $i < n+1$.

Case 1. In P the output port 0 needs recursive processing. Then for some function $g{:}\mathcal{N}[n+1, 1]$ one has for all z, ξ
$$\begin{aligned} P(z,\xi)_0(0) &= 0; \\ P(z,\xi)_0(t+1) &= g(z(0 \cdot\cdot t), \xi(0 \cdot\cdot t)). \end{aligned}$$
Now given ξ define $z_\xi \in \mathcal{S}$ by recursion
$$\begin{aligned} z_\xi(0) &= 0; \\ z_\xi(t+1) &= g(z_\xi(0 \cdot\cdot t), \xi(0 \cdot\cdot t)). \end{aligned}$$
Then $z_\xi \in \mathrm{BC}(\xi)$, by lemma 3.1.5, and satisfies (3.1).

Case 2.1. Output port 0 is directly connected to input port $i \neq 0$. Then $z_\xi = \xi_i \in \mathrm{BC}(\xi)$ is a solution of (3.1).

Case 2.2. Output port 0 is directly connected to input port $i = 0$. Then every $z \in \mathcal{S}$ satisfies (3.1). So we can take $z = \lambda t.0$ which trivially is in $\mathrm{BC}(\xi)$.

(ii) By the proof of (i). ■

3.1.7. DEFINITION. Let $P{:}\mathcal{S}[n+1, m+1]$ be a regular process. Then the process $P^*{:}\mathcal{S}[n, m]$ is defined as follows.
$$\begin{aligned} P^*(\xi)_j &= P(z_\xi, \xi)_{j+1}, && \text{if output port 0 is not directly connected to} \\ & && \text{input port 0 and } z_\xi \text{ is the unique element of } \mathcal{S} \\ & && \text{satisfying (3.1) in lemma 3.1.6;} \\ &= P(\lambda t.0, \xi)_{j+1}, && \text{else.} \end{aligned}$$

3.1.8. PROPOSITION. *Let $P{:}\mathcal{S}[n+1, m+1]$ be a regular process. Then P^* is regular.*

PROOF. Suppose that P is regular and let $j < m$ be an output port of P^*. Given an input scenario ξ for P^* we have to show that $P^*(\xi)_j \in \mathrm{BC}(\xi)$. Then $j+1$ is the output port of P corresponding to j. By definition of P^* one has $P^*(\xi)_j = P(z_\xi, \xi)_{j+1}$ or $= P(\lambda t.0, \xi)_{j+1}$. Since P is regular we have $P^*(\xi)_j \in \mathrm{BC}(z_\xi, \xi)$ or $\in \mathrm{BC}(\lambda t.0, \xi)$. By construction one has $z_\xi \in \mathrm{BC}(\xi)$ and trivially $\lambda t.0 \in \mathrm{BC}(\xi)$. Therefore by the regularity of P and lemma 3.1.5(i) one has $P^*(\xi)_j \in \mathrm{BC}(\xi)$ and we are done. ■

The set of *combinators* is defined by modifying definition 2.1.1(iv).

3.1.9. DEFINITION. (i) *Combinators* are defined as follows.

(1) The primitive combinators $\mathsf{I}, \mathsf{S}, \mathsf{E}$ and C are combinators.

(2) If M, N are combinators, then so is $M|N$.

(3) If M, N are combinators (of the right types), then so is $M \cdot N$.

(4) If M, N are combinators (of the right types), then so is $M \wedge N$.

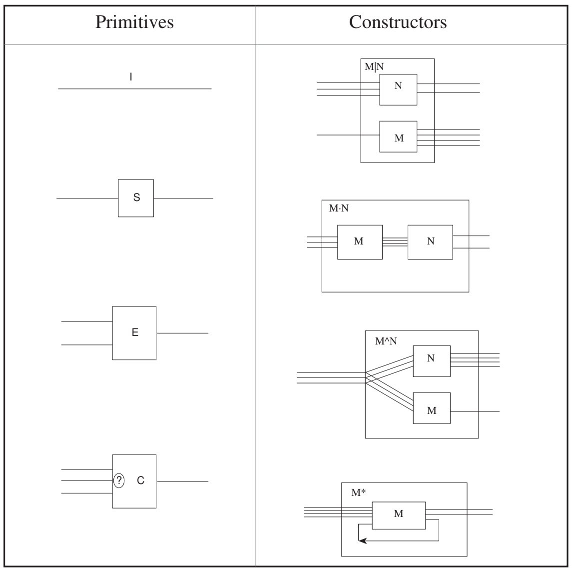
(5) If $M{:}\mathcal{S}[n+1, m+1]$ is a combinator (of given type), then so is $M^*{:}\mathcal{S}[n, m]$.

Combinators can be obtained only through (1)-(5).

3.1.10. Definition. (i) The semantics of combinators (and their corresponding boxes) is defined by extending definition 2.1.3 as follows.

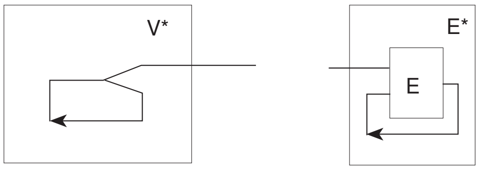
$$[\![M^*]\!] = [\![M]\!]^*.$$

(ii) A process P is called *computable* if $P = [\![M]\!]$ for some combinator M.



Toolkit for computable processes.

The primitive combinators L and O are left out; they can be defined as $\mathsf{O} = \mathsf{V}^*$, $\mathsf{L} = \mathsf{E}^*$.



Thus $\mathsf{Q} = \mathsf{O} \cdot \mathsf{L} = \mathsf{V}^* \cdot \mathsf{E}^*$. A simpler design is $\mathsf{Q} = \mathsf{I}^*$.

3.1.11. Proposition. *Every computable process is regular.*

Proof. Let $P = [\![M]\!]$, with M a combinator. By induction on the structure of M, using lemma 3.1.2 and proposition 3.1.8 it follows that P is regular. ■

3.1.12. Definition. Let $P{:}\mathcal{S}[n,m]$ be a process.

(i) P is *determined by the past* (of the input), notation $L_\infty(P)$, if for some $f{:}\mathcal{N}[n,m]$

$$\forall \xi{\in}\mathcal{S}^n \; P(\xi)(t) = f(\xi(0{\cdot\cdot}t)).$$

(ii) P is *recursively* determined by the past, notation $L_\infty^{\mathrm{rec}}(P)$, if f in (i) is required to be recursive.

3.1.13. Corollary. *Every computable process is recursively determined by the past.*

Proof. Let $P : \mathcal{S}[n,m]$ be a computable process and let $j{<}m$ be an output port of P. By the proposition P is regular. Hence either for some input port $i{<}n$ one has

$$P(\xi)_j(t) = \xi_i(t),$$

or for some recursive $f : \mathcal{N}[n,1]$

$$P(\xi)_j(t) = \text{ if } t{=}0 \text{ then } 0 \text{ else } f(\xi(0{\cdot\cdot}t{-}1)).$$

In both cases one can write for some recursive $g_j : \mathcal{N}[n,1]$

$$P(\xi)_j(t) = g_j(\xi(0{\cdot\cdot}t)).$$

(Note that the two occurrences of t denote the same moment.) Therefore,

$$P(\xi)(t) = g(\xi(0{\cdot\cdot}t)),$$

for the recursive $g = (g_0, \ldots, g_{m-1}) : \mathcal{N}[n,m]$. ■

Remember that for simple processes P we have proved the stronger property, namely $\exists d{\in}\mathcal{N}\ L_d(P)$. This does not hold for general computable processes: an output may depend on an input that happened an arbitrarily long time ago.

3.1.14. Example. There exists a computable process $P{:}\mathcal{S}[1,1]$ such that for all $x{\in}\mathcal{S}$ and all $t{\in}\mathcal{T}$ one has

$$P(x)(t{+}1) = x(0).$$

Therefore for no d one has $L_d(P)$. (But $L_\infty^{\mathrm{rec}}(P)$, by corollary 3.1.13.)

Proof. Take $P = [\![(\mathsf{X}\cdot(\mathsf{I}|\#1|\mathsf{I})\cdot\mathsf{C}\cdot\mathsf{V})^*]\!]$. ■

3.2. Examples

As was the case with the operators $^\wedge$, $\cdot$ and $|$ one can extend the applicability of the operator *.

3.2.1. Definition. (i) We define $^* : [\mathsf{S}^0{\to}\mathsf{S}^{k+1}]{\to}[\mathsf{S}^0{\to}\mathsf{S}^{k+1}]$ by

$$M^* = (M|I)^*.$$

(ii) Similarly $^* : [\mathsf{S}^{k+1}{\to}\mathsf{S}^0]{\to}[\mathsf{S}^{k+1}{\to}\mathsf{S}^0]$ is defined by

$$M^* = (M|I)^*.$$

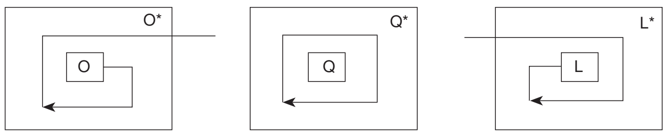
(iii) Finally $^* : [\mathsf{S}^0{\to}\mathsf{S}^0]{\to}[\mathsf{S}^0{\to}\mathsf{S}^0]$ is simply

$$M^* = (M|I)^*.$$

Case (iii) is the least useful: the unique process of type $[\mathsf{S}^0{\to}\mathsf{S}^0]$ is Q and $\mathsf{Q}^* \cong \mathsf{Q}$. The following proposition shows this and something more. It is trivial and just for the amusement of the reader.

3.2.2. Lemma. (i) $\mathsf{O}^* \cong \mathsf{O}$.
(ii) $\mathsf{Q}^* \cong \mathsf{Q}$.
(iii) $\mathsf{L}^* \cong \mathsf{L}$.

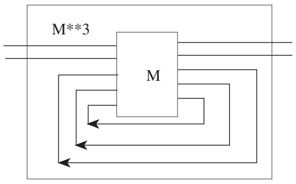
Proof.



■

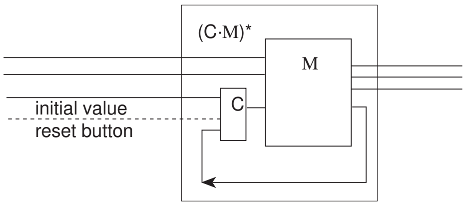
Also the operation * can be iterated.

3.2.3. Definition. (i) $M^{**}0 = M$.
(ii) $M^{**}(n+1) = (M^{**}n)^*$.



Iterated feedback.

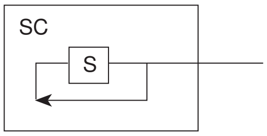
It is often useful to insert a switch in a feedback loop.



Whenever a 1 occurs at the port labelled 'reset button' the loop is initialized with the value then provided at the port labelled 'initial value'. From that time on the behaviour is deterministic (provided that f itself is deterministic), and the whole will behave like f^* as long as 'reset button' remains 0. We shall reserve the term button for input streams which are 0 most of the time and > 0 (usually 1) when a loop has to be reset to a defined initial value. Buttons will be depicted by dotted lines.

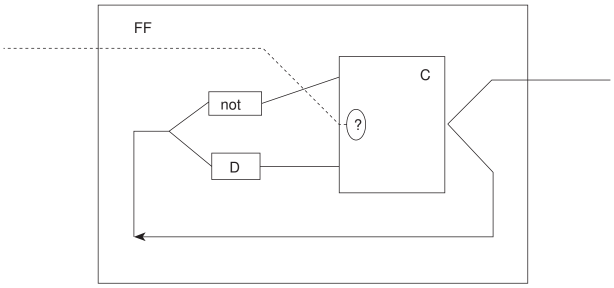
A simple box containing a loop is the 'systemclock'. It has no input and produces at moment t on the unique output channel a t.

3.2.4. Example. Systemclock

Specification	$\mathsf{SC}() = \lambda t.t$
Design $\mathsf{SC} = (\mathsf{S} \cdot \mathsf{V})^*$	Box 

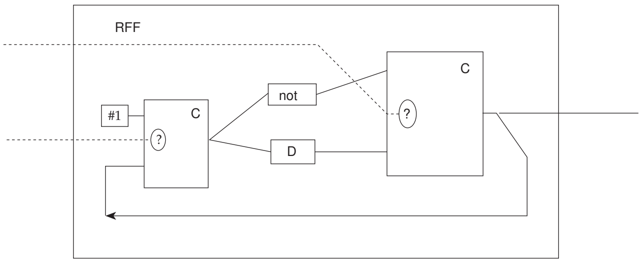
A flipflop is a box that 'flips' its output whenever a 1 occurs at its input button.

3.2.5. Example. Flipflop

Specification	$\mathsf{FF}(x) = y \Leftrightarrow$ $\exists\epsilon\in\mathcal{T}\forall t\in\mathcal{T}[y(t)\in\{0,1\}\&$ $[x(t) \neq 0 \Rightarrow y(\epsilon+t+1) = 1 - y(\epsilon+t)]\&$ $[x(t) = 0 \Rightarrow y(\epsilon+t+1) = y(\epsilon+t)]$
Design	$\mathsf{FF} = ((\mathsf{D} \wedge \mathsf{not}) \cdot (\mathsf{I}\vert\mathsf{X}) \cdot \mathsf{C} \cdot \mathsf{V})^*$
Box	

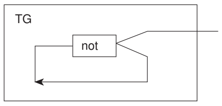
For this flipflop we do not know in advance the state it is in. A more fancy version can be reset: an extra button enforces the state to become 1 without inspecting the flipflop.

3.2.6. Example. Reset flipflop

Specification	$\mathsf{RFF}(b,x) = y \Leftrightarrow$ $\exists\epsilon\in\mathcal{T}\forall t\in\mathcal{T}\ [y(t)\in\{0,1\}\ \&$ $[[x(t)\neq 0\ \&\ b(t)=0] \Rightarrow y(\epsilon+t+1) = 1 - y(\epsilon+t)]\ \&$ $[[x(t)=0\ \&\ b(t)=0] \Rightarrow y(\epsilon+t+1) = y(\epsilon+t)]\ \&$ $[b(t)\neq 0 \Rightarrow y(\epsilon+t)=1]]$
Design	$\mathsf{RFF} = ((\mathsf{I}\vert\mathsf{I}\vert\wedge 1)\cdot\mathsf{C}\cdot(\mathsf{D}\wedge\mathsf{not})\cdot(\mathsf{I}\vert\mathsf{X})\cdot\mathsf{C}\cdot\mathsf{V})^*$
Box	

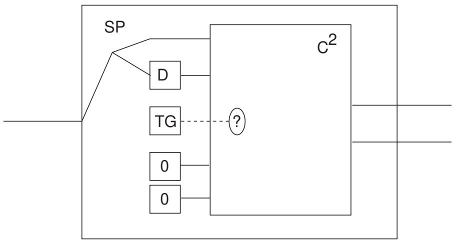
A toggle is a clock modulo 2 that after a warming-up time ϵ alternates between 0 and 1.

3.2.7. Example. Toggle

Specification	$\mathsf{TG}() = y \Leftrightarrow$ $\exists\epsilon\in\mathcal{T}\forall t\in\mathcal{T}\ [y(t)\in\{0,1\}\ \&\ y(\epsilon+t+1) = 1 - y(\epsilon+t)]$
Design $\mathsf{TG} = (\mathsf{not}\cdot\mathsf{V})^*$	Box 

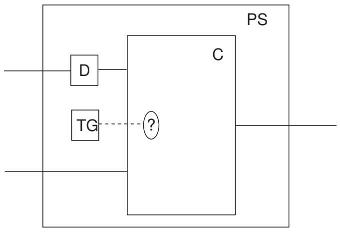
If we want to control the 'phase' of a toggle, then we could construct one with a reset button.
A toggle can be used to construct a serial-to-parallel converter with one data input port, and two output ports. At even moments it outputs in parallel the last and last but one input value, while at odd moments it outputs 0 on both lines.

3.2.8. Example. Serial-to-parallel converter

Specification	$\mathsf{SP}(x) = \langle y,z\rangle \Leftrightarrow \exists\epsilon\in\mathcal{T}\forall t\in\mathcal{T}\ [y(\epsilon+2t) = x(2t)\ \&$ $z(\epsilon+2t) = x(2t+1)\ \&$ $y(\epsilon+2t+1) = z(\epsilon+2t+1) = 0]$
Design	$\mathsf{SP} = (\mathsf{O}\vert\mathsf{O}\vert\mathsf{TG}\vert(\mathsf{D}\wedge\mathsf{I}))\cdot\mathsf{C}^2$
Box	

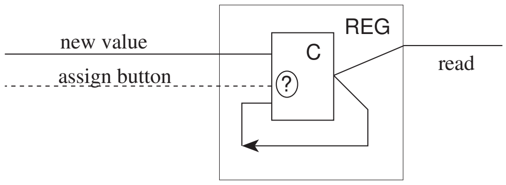
The inverse is a parallel-to-serial converter.

3.2.9. Example. Parallel-to-serial converter

Specification	$\mathsf{PS}(x,y) = z \Leftrightarrow$ $\exists \epsilon \in \mathcal{T} \forall t \in \mathcal{T} [z(\epsilon + 2t) = x(2t) \;\&\; z(\epsilon + 2t + 1) = y(2t)]$
Design $\mathsf{PS} = (\mathsf{I} \mid \mathsf{TG} \mid \mathsf{D}) \cdot \mathsf{C}$	Box 

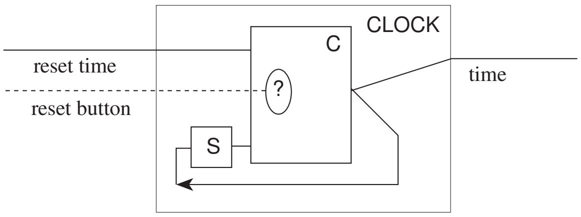
We can now also model registers, or 'variables', as they are called in imperative programming languages. Such registers are assigned new values from time to time and retain the value last assigned. It turns out that a register is nothing but a simple resettable loop.

3.2.10. Example. Register

Specification	$\mathsf{REG}(b,x) = y \Leftrightarrow$ $\exists \epsilon \in \mathcal{T} \forall t \in \mathcal{T}\ [b(t) \neq 0 \Rightarrow y(\epsilon + t) = x(t)] \;\&$ $[b(t) = 0 \Rightarrow y(\epsilon + t + 1) = y(\epsilon + t)]$
Design $\mathsf{REG} = (\mathsf{C} \cdot \mathsf{V})^*$	Box 

A simple modification makes this into a clock that can be set.

3.2.11. Example. Clock

Specification	$\mathsf{CLOCK}(b,x) = y \Leftrightarrow$ $\exists \epsilon \in \mathcal{T} \forall t \in \mathcal{T}\ [[b(t) \neq 0 \Rightarrow y(\epsilon + t + 1) = x(t)] \;\&$ $[b(t) = 0 \Rightarrow y(\epsilon + t + 1) = y(\epsilon + t) + 1]]$
Design	$\mathsf{CLOCK} = (\mathsf{S} \cdot \mathsf{C} \cdot \mathsf{V})^*$
Box	

Having a clock and a register we can construct a 'timer' that converts a value to a duration.

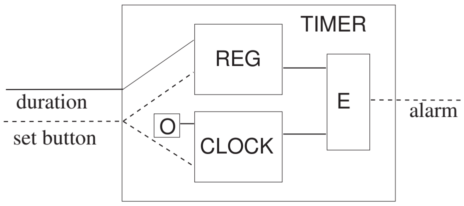
3.2.12. Example. Timer

Specification

$$\begin{aligned}&\mathsf{TIMER}(b,x) = y \;\Leftrightarrow\\ &\exists\epsilon{\in}\mathcal{T}\forall t{\in}\mathcal{T}\;[b(t)\neq 0\;\&\;\forall t' < x(t)\,b(t') = 0]\;\Rightarrow\\ &[[y(\epsilon + t + x(t)) \neq 0\;\&\;\forall t' < x(t)\,y(\epsilon+t+t') = 0]\;\&\\ &[\forall t' \geq x(t)\,b(t') = 0\;\Rightarrow\;\forall t'' > x(t)\,y(\epsilon+t+t'') = 0]]\end{aligned}$$

Design $\mathsf{TIMER} = (((\mathsf{I}|\mathsf{O})\cdot\mathsf{CLOCK} \wedge \mathsf{REG})\cdot\mathsf{E}$

Box



The converse, converting a duration between two moments at which a button b is pressed to a value, is the basic component of a stopwatch. The button B' indicates when the measured duratiuon is ready as output.

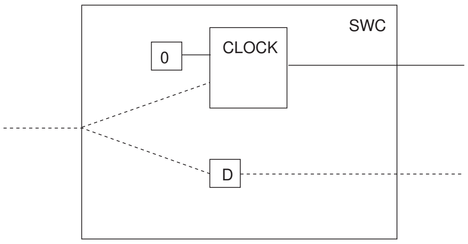
3.2.13. Example. Stopwatch component

Specification

$$\begin{aligned}&\mathsf{SWC}(b) = (b', y) \;\Leftrightarrow\\ &\exists\epsilon{\in}\mathcal{T}\forall t,k{\in}\mathcal{T}\;[b(t)\neq 0\;\&\;b(t+k)\neq 0\\ &\;\&\;\forall k' < k\,b(t+k') = 0]\;\Rightarrow\\ &[y(\epsilon+t+k) = k\;\&\;b'(\epsilon+t+1) = 1\;\&\\ &\forall k' < k\,b'(\epsilon+t+k') = 0]\end{aligned}$$

Design $\mathsf{SWC} = \mathsf{D} \wedge ((\mathsf{I}|\mathsf{O})\cdot\mathsf{CLOCK})$

Box



Combining this box with a register yields a primitive stopwatch.

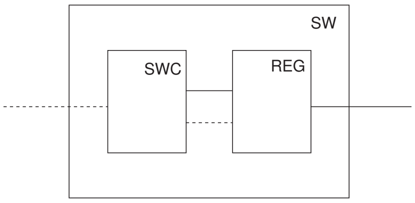
3.2.14. Example. Stopwatch

Specification

$$\begin{aligned}&\mathsf{SW}(b) = y \;\Leftrightarrow\\ &\exists\epsilon{\in}\mathcal{T}\forall t,k{\in}\mathcal{T}\\ &[b(t)\neq 0\;\&\;b(t+k)\neq 0\;\&\;\forall k' <\; b(t+k') = 0]\;\Rightarrow\\ &[y(\epsilon+t+k) = k\;\&\\ &[\forall k'' > k\,b(t+k'') = 0\;\Rightarrow\\ &\forall k' > k\,y(\epsilon+t+k') = k]]\end{aligned}$$

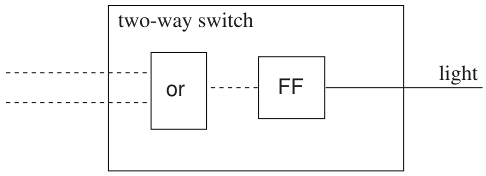
Design $\mathsf{SW} = \mathsf{SWC}\cdot\mathsf{REG}$

Box



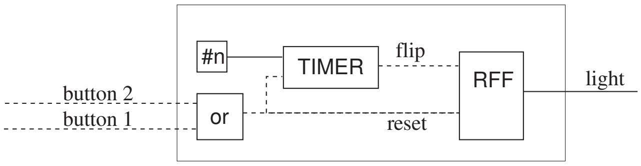
We end this section with the two testcases: two-way switch and three minute switch.

3.2.15. Example. A two-way switch can be designed as follows.



Design: $\mathsf{or} \cdot \mathsf{FF}$.

3.2.16. Example. A three minute switch may be designed as follows.



Design: $(\mathsf{or} \mid \#n) \cdot (\mathsf{I} \wedge \mathsf{TIMER}) \cdot \mathsf{RFF}$.

We must take $n = 177$ if the clockspeed is one cycle per second.

We have seen that the specification of computable processes is less easy than one would expect. In a future paper we hope to compare existing formalisms for this. Also the need for methods to proof that a design satisfies a given specification are important.

4. Discussion

The main point of this paper is the definition of the notions (synchronous, digital) process, computable process, and design of a computable process. Although the constituents of our definitions are more or less well-known, we have not seen them together in this form. What have we achieved and what not?

4.1. Adequacy

Does the definition meet the adequacy criteria stated in 2.4?

1. The concept of computable processes is indeed a generalization of the concept of computable function. We have seen, that computable processes can do things that computable functions cannot do. They can control reactive systems, among which our test cases from section 2.2. In particular, they can convert numbers to durations in real time and vice versa (cf. examples 4.2.12 and 4.2.13). But we have not shown yet that they can *at least* do what computable functions can do. To this end we must prove that for every recursive function there exists a computable process that computes it. The proof involves strenuous technical details and will not be given here. The idea is to show how every expression defining a recursive function can be translated to a design for a corresponding computable process. Whenever recursion or minimalization is involved, computations may require an unbounded amount of time, which implies that the moment at which a result is produced cannot be known beforehand. The values of streams will not be relevant most of the time. Therefore, a protocol will be needed by means of which boxes can tell each other when a valid value is delivered at a port, like in the examples of the previous section, where an auxiliary stream ('button') is involved, that is zero most of the time and one at the moments only when the values at the other ports are meaningful.

2. Computable processes cover anything that digital computers can do, because every digital computer can be modelled by a large combinator. The idea is simple. A computer besically consists of a cpu with a number of input and output channels and memory. The cpu is a device constructed from logical gates, synchronized by a clock. It realizes a function that is repeatedly applied to the 'state'

of the machine to determine a 'successor state'. We have shown in example 2.2.22 that synchronized logical gates can be modelled as simple processes. A cpu for n input channels, k output channels and m memory cells, registers etcera can essentially be modelled as a simple (although quite large) process CPU of type $S[m+n, m+k]$. Memory is a device that feeds back a successor state as initial state for the next processor cycle. Therefore, $\mathsf{CPU}^{**}m$ makes a complete computer.

3. We leave it to the reader to judge wether the examples given earlier show that our notion of computable process allows straightforward design of self-timing and real-time properties.

4. The design language is based on a small number of primitive combinators $(\mathsf{I}, \mathsf{S}, \mathsf{E}, \mathsf{C})$ together with the constructors $^\wedge, \cdot, |$, and *. The compositionality and the algebraic properties of the latter have been recognized in the literature.

5. Like register machines, computable processes are able to handle unbounded numbers; in that sense they are not very realistic. Under the assumption of a fixed upper bound for all occurring values, however, they can be realized in hardware or as programs in a straigtforward way. A design expression for a computable process can be used as a blueprint of an electronic circuit.

4.2. Related theories

In the work of Berry [1989], [1993] on the language Esterel for reactive programming, process descriptions are given that have the same semantics as ours. The description uses systems of equations in order to deal with feedback. In the related languages Lustre and Signal a distinction is made between processes of streams of natural numbers and of streams of bits. Of course the latter is more realistic. (In this theory the relation $\cong$ becomes decidable.) But like the infinite tape of a Turing machine (or the capcity of register machines to store arbitrary integers) our theory is technology independent. It holds for 16-bit but also for 32-bit machines. Of course 64-bit processing can be done on a 32-bit machine by using two clock cycles. But we did not want to go into these details and described essentially ∞-bit machines.

Berry has another ideal element in his theory, one that we do not have: the throughput time of simple processes is taken to be 0. This means for example that $\mathsf{S}(x)(t) = x(t) + 1$ and not $\mathsf{S}(x)(t+1) = x(t) + 1$. In order to make a consistent theory, the process combinator D is taken as primitive and is required to be inserted in a feedbackloop. Berry motivates this choice by stating that the clockspeed should be such that all corners of the process control machine are reachable within one cycle. Then the throughput speed is indeed 0. In our theory we have the disadvantage that, for example, the combinator #4 has warming up time 4.

4.3. Future work

Axiomatizing equalities

One can look for an axiomatization of the valid equalities $(\cong, \simeq)$ between process combinators. For example the following fact should be a consequence of this axiomatization.

4.3.1. FACT. For every computable process P there exists a simple process Q such that $P \cong Q^{**}n$ for some $n \in \mathcal{N}$.

Similar laws are interesting as well.

Also it should follow from this axiomatization that $\cong$ and $\simeq$ on process combinators are undecidable.

Higher level descriptions

As stated before, specifications in predicate logic are not very satisfactory. There are specifications of several versions of a timer in real-time process algebra that are very elegant, see Baeten and Bergstra [1990]. It therefore is worthwhile to investigate whether the process algebra specifications can be given also for our theory. In any case, a higher level specification is necessary, when dealing with processes that wait for other ones.

We have only treated synchronous processes. The asynchronous ones will have to be covered by a similar higher level described in the previous paragraph.

Characterizing computable processes

It is not clear to us what are the recursive functions needed in order to show that a computable process is L_∞^{rec} or L_d^{rec} for some $d \in \mathcal{N}$. In other words, how does the output depend recursively on the past? For simple processes the characterization of these functions must be easy. For general processes it probably is not too hard.

Acknowledgements

The authors had interesting feedback from E. Barendsen, G Berry, R. Boute, D. van Leijenhorst, M. Massink, H. Meijer, H. van Thienen, and W. Vree.

References

Baeten, J. and J.A. Bergstra
[1990] *Real-time process algebra.* Report P8916b, Programming research group, University of Amsterdam, PO Box 41882, 1009 DB Amsterdam.

Bergstra, J.A. and J.W. Klop
[1986] Process algebra: specification and verification in bisimulation semantics, in: *Mathematics and Computer Science II*, CWI Monographs 4, North-Holland, Amsterdam, 61-94.

Berry, G.
[1989] Real-time programming: General purpose or special-purpose languages, in: G. Ritter, Ed., *Information processing 89*, Elsevier, Amsterdam, 11-17.
[1993] The semantics of Pure Esterel, obtainable from berry@cma.cma.fr.

Boute, R.
[1986] System semantics and formal circuit description, *IEEE Transactions on Circuits and Systems*, CAS-33, 12, 1219-1231.
[1988] System semantics: principles, applications and implementation, *ACM Trans. Prog. Lang. and Syst.*, 10 (1), 118-155.
[1990] ESPRIT Project 881 - FORFUN: Formal description of digital and analog systems by means of functional languages, in: *ESPRIT '90 Proc. Annual Conf. (Brussels)*, Kluwer, Dordrecht, 212-226.

Brock, J.D. and W.B. Ackermann
[1981] Scenarios: a model of non-determinate computation, in: J. Diaz and I. Ramos (eds.) *Formalisation of programming concepts*, Springer LNCS 107, 252-259.

Broy, M.
[1992] *(Inter-)Action Refinement: The Easy Way*, Institut für Informatik, TU München, Postbox 202420, 8 München 2, Germany.
[1992a] *Compositional Refinement of Interactive Systems*, Institut für Informatik, TU München, Postbox 202420, 8 München 2, Germany.

Kahn, G.
[1974] The Semantics of a Simple Language for Parallell Programming, in: *Information Processing 74: Proceedings of the IFIP Congress 1974*, ed.: J.L. Rosenfeld, 471-475.

Kleene, S.C.
[1952] *Introduction to metamathematics*, van Norstrand.

Meertens, L.
[1989] *Constructing a calculus of programs*, CWI reports R 8914, Kruislaan 413, 1098 SJ Amsterdam.

Misra, J.
[1990] Equational reasoning about non-deterministic processes, *Formal aspects of computing*, 167-195.

Moschovakis, Y.N.
[1989] A game-theoretic model of concurrency, in: *Proceedings of the 4-th annual symposium on Logic in Computer Science*, IEEE Computer Society Press, 154-163.
[1990] Computable processes, in: *Proceedings of the conference on Principles of Programming Languages*, 72-80.

Stark, E.W.
[1992] A calculus of dataflow networks, in: *Proceedings of the 7-th annual IEEE symposium on Logic and Computer Science (Santa Cruz)*, 125-136.

Ştefănescu, G.

[1987] On flowchart theories, Part I. The deterministic case. *J. Computer and System Sciences*, 35 (2), 163-191.

[1985] On flowchart theories, Part II. The non-deterministic case. *Theoretical Computer Science, 52, 307-340.*