The Synchronous Programming Language SIGNAL
A Tutorial

Bernard HOUSSAIS
IRISA. ESPRESSO Project

24th September 2004

Contents

Chapter 1

Introduction

Signal is a programming language designed for *Real Time* applications (or *reactive*, or *embedded* systems)[1]. It is a *synchronous* language, opposite to the *asynchronous* approach, like in Ada. Like *Lustre*[4], its style is *declarative*, to be compared with *imperative* synchronous style in *Esterel*[5].

This tutorial presents only the elementary parts of the language (V4 version). It aims to help programmers beginning to write some pieces of programs, for simple applications. Many advanced concepts, objects, statements are not considered. All definitions - particularly semantic ones - are very informal.

For a further study, see the *Signal V4 Reference Manual*[2]. Look also the web site of *ESPRESSO Project*[3] at IRISA.

1.1 An example : the WATCHDOG process

As an informal survey, we develop and comment a short, but complete example of real time module : a *watchdog*.

1.1.1 The problem

A process emits an *ORDER*, to be executed within some *DELAY*. When finished, a *DONE* signal is made available. The *WATCHDOG* is designed to control this delay. It receives a copy of ORDER and DONE signals. It must emit an *ALARM* whenever a job is not finished in time.

More specifications : if a new ORDER occurs when previous one is not finished, the time counting restarts from zero. A DONE signal out of delay, or not related to an ORDER, will be ignored.

1.1.2 Input and output signals

An ORDER is supposed to be coded by an *integer*. So, the WATCHDOG receives as input a sequence of integers, separated by some undefined amount of time.

The process receives also as other input the DONE signals. A DONE signal is only a pulse, a simple information, as one obtained by pushing a button. Its type is named in Signal an *event*, and is coded by a "boolean" always *true*.

In order to count the time, *synchronous* languages do not use language-defined devices, like *seconds*, whose accuracy is not sufficient. The *source of time* is also an input signal, of type *event*. The amount of time between two such *time events* is defined (and guaranteed !) by the environment. We suppose this time signal is named *TICK*. The parameter *DELAY* is a number of *TICKs*, so the unit of physical time must not be given.

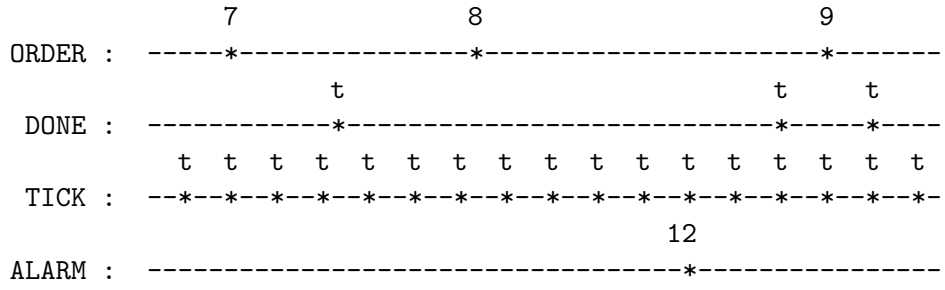
As output, the WATCHDOG produces an *ALARM* when the DELAY between ORDER and DONE is exceeded. This ALARM could be an *event*, or better the HOUR of the alarm, i.e. the number of TICK since the beginning of execution. Its type is *integer*.

So, inputs and outputs are *sequences of values* of some type, each value of the sequence being present at some *instants*. Such a sequence is called a *signal*. The set of instants where a signal takes a value is the *clock* of the signal.

1.1.3 Example of progress

To show a possible scenario for the WATCHDOG, we use a *time diagram*, with the time as abciss. Each signal is represented on an horizontal line, with a mark for each value. Values which occur at the same time are on the same vertical.

We assume here: DELAY = 5 :



ALARM occurs on the 5th TICK following ORDER number 8. It was also the 12th TICK from the beginning.

An other representation (more precise) of a scenario would be to show only these moments where at least one signal is present. Absent signals are then represented by the *bottom* symbol : $\perp$

ORDER :	$\perp$	7	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	8	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	9	$\perp$	$\perp$
DONE :	$\perp$	$\perp$	$\perp$	$\perp$	*t*	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	*t*	$\perp$	*t*	$\perp$
TICK :	*t*	*t*	*t*	*t*	$\perp$	*t*	*t*	*t*	$\perp$	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*
ALARM :	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	12	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

We see more precisely that ORDER no 7 arrived exactly on a TICK; the first DONE occurs between TICK 4 and TICK 5.

1.1.4 Synchronism hypothesis

A Signal program does not deal with the exact duration between two signals. It only knows the *relative* order of signals, with the fact they are simultaneous, or not.

The language assumes the environment is able to say if two input signals are simultaneous, or which of them is before the other. The only instants considered in a program

are those where at least one signal is present. At these moments, some computing can be done on signals, producing new values.

An other hypothesis is that *computations take a null time.* So we consider that new values are produced at the *same logical instant* as data used in their computation.

For example, ALARM signal is produced on some TICK, the 5th after an ORDER, if no DONE has arrived before or at the same time. The decision to produce an ALARM *depends on* TICK and DONE signals, and is necessarily after them. But we assume that the decision can be taken in a null time, and the ALARM is produced at the *same instant* as the 5th TICK.

1.1.5 The process WATCHDOG in Signal language

Numbered lines are those of the program.

```
1 :  process WATCHDOG =
2 :  % emits an ALARM if an ORDER is not DONE within some DELAY %
3 :    { integer DELAY;}
4 :    ( ? integer ORDER;
5 :        event DONE, TICK;
6 :      ! integer ALARM; )
```

These lines are the *interface* of the process. They are the only ones an other process has to know to use it.

Syntactically, DELAY is a *parameter.* It is a constant, to be fixed before compilation. The **?** symbol introduces input signals, with their types. The **!** symbol is for output signals.

```
7 :   (|
```

Beginning of process body, made of a set of *equations.* These equations define values or constraints on signals. They are separated by vertical bars : $(|\ldots|\ldots|\ldots|)$.

```
8 :    HOUR ^= TICK
9 :  | HOUR := (HOUR$ init 0) + 1
```

Signals can be input/output signals (like TICK), or local signals (like HOUR), whose declaration is placed at end of text. HOUR is a counter designed to number TICK signals. It will be output to give the HOUR associated with an ALARM.

Line 8 makes this signal present at the same instants as TICK : HOUR and TICK have the *same clok*, they are *synchronous signals.* $\hat{}=$ is the *clock equality* operator. Equations like 8 are necessary when the context is not sufficient to define by itself the clock of a signal.

The *value* of HOUR signal is defined on line 9. As a declarative language, Signal does not modify variables, like in *HOUR := HOUR+1.* The previous value (at previous instant) is made available at current instant by *HOUR $.* Its value at first instant is set to 0, after *init* keyword.

The present value of HOUR is the previous one + 1. This definition by itself does not define the clock of HOUR, so line 8 is necessary.

```
10 :    | CNT ^= TICK ^+ ORDER ^+ DONE
11 :    | ZCNT := CNT $ init (-1)
12 :    | CNT := DELAY when ^ORDER
13 :             default -1 when DONE
14 :             default  ZCNT - 1 when ZCNT >= 0
15 :             default -1
```

The time between an ORDER and its completion is counted by a decreasing counter, named CNT (for COUNT is a Signal keyword !). Its initial value is set to DELAY when an ORDER arrives. Then, it decreases on each TICK, until arrival of a DONE signal, or until it reaches 0 if no DONE comes.

The equation line 10 defines the instants where CNT is present. Its clock is the union ($\hat{}+$ operator) of clocks of the 3 input signals, i.e. the set of instants where at least one of them is present.

ZCNT has the value of CNT at the previous instant, according to the clock of CNT. Its first value is -1, the "rest" value of CNT when the process is not waiting for a termination.

Line 12 gives to CNT the value DELAY when an ORDER arrives. The $\hat{}$ operator extracts from ORDER its clock, of type *event*. If at the instant, there is no ORDER, but a DONE signal, CNT takes immediately the rest value -1. We have not to write $\hat{}DONE$, because DONE is already an *event*.

If there is no ORDER nor DONE signal, we are on a TICK signal : CNT is decreased as long as it is positive or null; otherwise, it remains to -1.

The *when* and the *default* are *polychronous* operators. The *when* has a higher priority than the *default*.

```
16 :    | ALARM := HOUR when CNT = 0
```

The ALARM signal takes the value of HOUR when the decreasing counter reaches 0. It means that DONE is not arrived in time, to set it directly to -1. Notice that HOUR is always present when CNT=0, because the clock of CNT includes TICK's one, which is the same as HOUR's one.

```
17 :    |)
```

This line terminates the set of equations. These equations could be written in any order : the compiler analyses their dependencies, and determines the moments where they must be calculated.

Declaration of local signals :

```
18 :  where
19 :    integer HOUR, ZCNT, CNT;
20 : end % WATCHDOG %;
```

1.1.6 Using WATCHDOG process

The watchdog process is a piece of a larger application, which will produce its inputs, and knows what to do with the ALARM. Editing of the program can be done textually, or with a graphical syntax-directed editor.

The process can also be edited by itself, in a file *watchdog.sig*. An other file, named *watchdog.par*, contains the value of DELAY. This file can then be compiled separately.

The compiler analyses the syntax, verifies the clock definitions, and defines the order of calculations. An original part is the *clock calculus* : clocks of signals must be well defined, coherent, and without circularities; their analysis may reveal hidden constraints, not explicitly designed by the programmer. Compilation may establish that some clocks are included in others; so the computation of their values is less frequent, and the object code reflecting that is more efficient.

When source program is correct, object code is first generated in C, which is compiled afterwards. Object code is then easier to read. This gives also automatic access to C libraries for input/output, mathematical functions,... and facilitates connection with other parts of the application.

Input signals must be prepared in files, whose name is predefined :

- a *RORDER.dat* file contains the values of ORDER : 7, 8,...

- In this example, clocks of input signals must also be given, in one file for each signal : *RC_ORDER.dat* for the clock of ORDER,... They contain the same number of booleans, one for each instant where at least one signal is present. At some instant, *RC_X.dat* contains a *1* if *X* is present, a *0* if it is absent.

Here follows input files corresponding to the scenario given above :

```
  RORDER.dat :  7  8  9
RC_ORDER.dat :  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1  0  0
 RC_DONE.dat :  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0
 RC_TICK.dat :  1  1  1  1  0  1  1  1  0  1  1  1  1  1  1  1  1  1  1
```

The number of values in RORDER.dat is the number of 1 in RC_ORDER.dat.

The values of output signal are produced in a file *WALARM.dat*, without clock.

This is a time diagram including local signals :

ORDER :	$\perp$	7	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	8	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	9	$\perp$	$\perp$
DONE :	$\perp$	$\perp$	$\perp$	$\perp$	t	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	t	$\perp$	t	$\perp$
TICK :	t	t	t	t	$\perp$	t	t	t	$\perp$	t	t	t	t	t	t	t	t	t	t
HOUR :	1	2	3	4	$\perp$	5	6	7	$\perp$	8	9	10	11	12	13	14	15	16	17
ZCNT :	-1	-1	5	4	3	-1	-1	-1	-1	5	4	3	2	1	0	-1	-1	5	-1
CNT :	-1	5	4	3	-1	-1	-1	-1	5	4	3	2	1	0	-1	-1	5	-1	-1
ALARM :	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	12	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Chapter 2

Signals

2.1 Signals in Signal

In Signal language, a signal is a *sequence of values of the same type, which are present at some instants*. The set of instants where a signal is present is the *clock* of the signal.

These discrete signals differ from continuous analogic signals used in some applications; but they can be obtained by sampling such continuous signal.

The physical amount of time between two values, for instance in seconds, is not relevant in this language. If needed, the physical time must be given as an input signal, among others. It would be the environment's responsability to provide for instance a sequence of pulses separated by exactly one second.

In the matter of time, events are considered only according to their relative order : a value from a signal can occur before a value of another signal, or after, or at the same instant. It is also the environment's responsability to determine if two input values are simultaneous, or not. The language constructs can test if given values are simultaneous; following actions are then defined in a deterministic way.

Consequently, at some instant, a signal may be present, an other one absent, and a third one also present. An "absent value" for a signal is denoted by the *bottom symbol :* $\perp$. It means that some other signal is present at the same instant. Instants where all signals are absent are not considered.

For instance, the set of sequences :

$$\begin{array}{lcccccccccc} a_1: & 1 & 2 & \perp & \perp & 7 & \perp & 5 & \perp & \perp & \perp... \\ a_2: & \perp & 10 & \perp & \perp & 123 & \perp & 5 & -1 & \perp & \perp... \\ a_3: & 0 & 6 & \perp & \perp & 13 & \perp & 2 & \perp & \perp & \perp... \\ a_4: & \perp & 11 & \perp & \perp & \perp & \perp & 1 & \perp & \perp & \perp... \end{array}$$

can be simplified in :

$$\begin{array}{lccccc} a_1: & 1 & 2 & 7 & 5 & \perp \\ a_2: & \perp & 10 & 123 & 5 & -1 \\ a_3: & 0 & 6 & 13 & 2 & \perp \\ a_4: & \perp & 11 & \perp & 1 & \perp \end{array}$$

Two signals are *synchronous* if they are always present (or absent) at the same instants. They have the *same clock* (operator $\hat{}=$). In the example above, a_1 and a_3 are synchronous : $a_1 \hat{}= a_3$.

Clocks have also a (partial) order relation : all instants of a_4 are also instants of a_3 ; a_4 is *less frequent* than a_3, or a_4's clock is *included* in a_3's, or $a_4 \hat{}< a_3$. But, clocks of a_1 and a_2 are not comparable.

2.2 Name of signals

In a Signal program, signals must be declared, with an identifier and an associated type for their values.

A name of signal is not a classical variable : it represents a *sequence* of values, even if, at a given instant, it holds only one value, or is *bottom*. Nevertheless, operators give access to *past* values of a signal : previous one, or the N'th before present, or even the set of N previous values, in a sliding window.

Like in declarative languages, the present value held by a name cannot be modified, as in : $S := S + 1$

2.3 Type of signals

The values of a signal may be : predefined simple values (integer,...), strings, enumerated types, multiples (arrays,..),... In this tutorial, we consider only simple values.

2.3.1 Numerical types

Signals of following types hold numerical values : integer, real, dreal (*double* in C), complex. They have classical operators, with usual priorities. Operands must be of the same type, but we can write : *X + real(N)*. The division between integers yields an integer quotient. Modulo is written *modulo* (% is the comment symbol). Functions are those of the host language (C); their heading must be declared locally (see below).

2.3.2 Boolean type

This type is named *boolean*; it differs from *integer*. A boolean value may be obtained by *true* or *false* constants, by an *event* signal (see below), by comparison of values (*operators :* $= /= < <= > >=$), or by boolean operators *not, and, or*, with usual priorities.

The boolean expression :

```
not OK and A /= B + N modulo 2
```

is implicitly parenthesized as :

```
(not OK) and (A /= (B + (N modulo 2)))
```

2.3.3 Type event

It is a *pure signal*, useful only for its clock. It has a unique value, which is the boolean *true*. An *event* is accepted with this *true* value in a boolean expression.

The ^ *operator* (*clock* operator), applied to any signal, extracts its clock, and yields an event signal :

```
      ORDER : ---- 7 -------- 8 --------------- 9 ------

     ^ORDER : ---- t -------- t --------------- t ------
```

In the WATCHDOG program, integer values of ORDER are not used. This identifier appears only with the clock operator, and it would be sufficient to put this clock as input.

2.4 Declaration of signals

Signals may be declared as *input* signals, *output* signals, or *local* signals. Following example intends to show some points of syntax. Output and local signals may be initialized, especially when they represent past values of other signals.

```
process DECLARE =
  { integer NBL, NBC; } % Parameters = Constants %
  ( ? real a;         % Input signals%
      event EV, HH_2; integer B;
    ! boolean ok init false, FOUND; )    % Output signals %

  (| XX := sqrt(fabs(-A))
   |  ...
   | OK := inter <= NBC when FOUND
   |)
where      %  Local signals %
  real XX, YY init -1.5;   % only YY  initialized %
  integer inter;
     % functions interfaces %
  function sqrt = (?dreal A !dreal B);  % double in C %
  function fabs = (?dreal A !dreal B);
end %DECLARE%;
```

2.5 Constants, parameters

Usual denotations like *18, true,...* are constants. Other constants can be given as parameters, like *DELAY* in WATCHDOG example, or *NBL, NBC* above. They must be known before compilation. For instance, above DECLARE process could be called by :

```
   ... DECLARE {20,M+1} (.. effective input signals ..)
```

where M is also a parameter in its calling context. No clocks are associated with constants (they are not signals). They are present whenever context need them.

Chapter 3

Signal definitions, operators

A Signal program body is a set of equations defining output and local signals. For each signal, its *clock* must be defined, and also the *value* it holds when it is present.

Specific operators on signals are used to build new signals. Some of them use only synchronous signals, and their result have the same clock as operands : they are *monochronous* operators. Other operators act on signals with any clock, and their result may have another clock : they are *polychronous* operators.

3.1 Definition of a signal

3.1.1 Defining equation

Each output or local signal must be defined by a unique equation of the form :

```
Name := Signal Expression
```

This equation says :
- Name and Signal Expression have the *same clock*
- the type of Expression must be acceptable by Name; at these instants defined by the clock, the Expression is computed in a *null time* (synchronism hypothesis) ; the Name takes the value given by the Expression.

As long as operators are monochronous, we consider that all signals have the same clock.

3.1.2 Examples

In the program :

```
  process PLUS1 =
    ( ? integer IN;
      ! integer OUT; )
(| OUT := IN + 1 |)
```

the equation says that IN and OUT have the same clock. If this process is inserted in a larger application, this clock equality is added to others equations of the system.

At run time, each time an integer value is available on IN, it is immediately increased, and the result is delivered on OUT, at the same logical instant.

In a testing environment, where this process is run alone, a file containing input values is read at normal speed, and an output file is produced at the same speed. No input clock file is needed.

In this other process :

```
process ONES =
  ( ?     % no input %
    ! integer S; )
(| S := 1 |)
```

the constant expression *1* has no clock by itself. The clock of S has to be set by the environment where this process is placed. Each time S will be "asked to be present", its value will be 1. If run by itself, clock of S will be a "main clock", which is the normal speed of the host machine : an infinite sequence of 1 will be produced !

If a process contains several unrelated signals :

```
process TWO =
  ( ?    % no input %
    ! integer S1, S2;)
(| S1 := 1
 | S2 := 2 |)
```

the environment must set from outside the clocks of S1 and S2. If used alone, clock files for S1 and S2 will be asked at run time (even for output !)

3.1.3 Clock equality equations

In cases like previous one (and many others more useful!), the defining equation of a signal may *not* be sufficient to define its clock. The syntax provides an other form of equation :

```
| S1 ^= S2
```

which tells explicitly that S1 and S2 have the same clock.

If we want a signal S to be produced with the clock of an other signal A, and the values of S are not related to A, we must write such an equality :

```
process S_ON_A = % produces a 1 on each input A %
  ( ? event A;
    ! integer S;)
(| S := 1
 | S ^= A |)  % or A ^= S %
```

If we know two input signals are synchronous, or if we want to impose them to be synchronous, we write :

```
process SYNC_IN =
  ( ? integer A, B;
    ! ... )
(| A ^= B
 | ...
```

Two values from input A and input B will be read at the same time. In a larger system, the set of clock equations (including this process' one) will guarantee that effective signals for A and B are synchronous. If run alone, files for A and B values will be read in parallel. No clock files are required.

The clock equality may have several terms : one of them may be a (polychronous) expression ; see WATCHDOG example.

The equation defining the value of S must be unique, but its clock may be defined in several ways. Of course, the compiler verifies their compatibility.

3.2 Monochronous operators

3.2.1 Operators related to types

Classical operators (addition, comparison, and, or,..) and library functions can act on signal values *only if they are synchronous*. Writing such an operator between signals is even a mean to put equality between their clocks. Result has the same clock as operands. In :

```
process ADD =
  ( ? integer A, B;
    ! integer S; )
(| S := A + B |)
```

the operator + introduces an implicit clock equality between A and B. The expression $A + B$ and the signal S have also this same clock : the time necessary for the addition and the assignment is considered as null.

The result of an unary operator (*-, not*, clock operator ^) is also synchronous with the operand.

3.2.2 Delay operator

Delay operator **$** gives access to past values of a signal. Delayed values of A have the *same clock* as A.

A $ is the previous value of A.

A $ N is the value hold by the signal N instants before present A, if we count only instants where A was present. *N* is a constant positive expression; it may contain parameters, but not signals.

```
PIXEL $ (NBL*NBC-1)
```

Initialization

The N first values of *A $ N* are undefined. They can be initialized :
- on the place where used :

```
| S := A $ init 0
| Y := 5*( X $ 2 init [10,20]) + X$ init 0
```

- or in the declaration of a signal holding delayed values :

```
   | ZA := A $ 1
   | ZB := A $
   | ZC := A $ 3
where
   integer ZA, ZB init 0, ZC init [10,20,30];
```

```
   A  : ---  1 ---  2 -------  3 --- 4 --- 5 ------- 6 ---

  ZA  : ---  ? ---  1 -------  2 --- 3 --- 4 ------- 5 ---

  ZB  : ---  0 ---  1 -------  2 --- 3 --- 4 ------- 5 ---

  ZC  : --- 10 --- 20 ------- 30 --- 1 --- 2 ------- 3 ---
```

Using delayed signals

To program a *counter* 1, 2, 3,.. we write :

```
   | CNT := (CNT$ init 0) + 1
or
   | ZCNT := CNT$
   | CNT := ZCNT + 1 |)
where
   integer ZCNT init 0;
```

ZCNT and CNT have the same clock. But definitions above does not set this common clock. If run alone, an infinite sequence 1,2,3,4,... will be produced for CNT. Generally, a separate clock equality links the counter to some other signal.

```
   HOUR ^= TICK
 | HOUR := (HOUR$ init 0) + 1
```

Exercices :

1. A unique button, activating an *event* signal named BUTTON, is used to start or stop a machine. Write a process emitting a boolean BUSY, with the clock of BUTTON, and whose value reflects the state of the machine after pushing the button.

2. Produce for each value of an input signal A, of type *real*, the *SUM* of all values entered until now.

3. Produce on each A the *MEAN* of these values. To be divided, the counter must be converted to real by *real(CNT)*

4. Produce on each A the mean of the N last values (N : constant identifier). All N previous values have not to be written ! A$N and A are sufficient, with the previous sum.

   A$N may be initialized to 0 by : *init* [{ *to N* } : 0.0]. The N first values of the mean will not be considered.

5. Produce, with the clock of some signal A, a boolean signal FIRST, *true* on the first value of A, then *false* on following ones. Try to avoid using a counter !

We consider now **polychronous** operators : **when** and **default**.

3.3 Sampling, when operators

These operators extract from some boolean expression on signals the instants where its value is *true*. So the *clock* of the result is *less frequent* than the expression's clock (or is *included in* it).

3.3.1 Unary when

```
      when Boolean_expression
Example :
    NULL := when CNT = 0
```

The result is of *event* type : it is present (and always *true*) only when boolean expression is true.

```
    CNT  : --- 1 --- 0 ------ 2 --- 0 --- 0 --- 3 ----

   NULL  : --------- t ------------ t --- t ----------
```

This operator is used to "mark" some instants where signals have particular properties. It can be used in a clock equality to *synchronize* another signal. For instance, to count *false* values of some boolean signal B :

```
   | CNT ^= when not B
   | CNT :=(CNT $ init 0) + 1
```

Exercices

1. Let *A* be an input signal, made of positive integers. Produce an *UP* event signal for each A greater than previous one. For initialization, assume the value "before the first A" is 0.

2. Produce a *TOP* event signal each time a summit for A values has been passed. This is possible only on the first decreasing value !

3. Suppose now that some consecutive values of A may be equal. There is a summit on decreasing A only if A was increasing before it became flat ! Use a boolean signal to remember the up/down status of A.

3.3.2 Binary when

```
      Signal_expression when Boolean_expression
Example :
    ALARM := HOUR when CNT = 0
```

The result is the *Signal_expression* when it is present, and that *Boolean_expression* is present and true.

```
    HOUR  : --- 2 ------------- 4 --- 6 --- 8 -----

     CNT  : -------- 1 -- 0 --- 0 --- 3 --- 0 -----

    ALARM : ------------------- 4 --------- 8 -----
```

The priority of binary *when* operator is lower than priorities of classical operators.

The clock of *A when B* is *less frequent than* clock of A and clock of B. It is the *intersection* of *^A* and of *when B*.

Examples of use

A when prop(A) extracts those values of A for which the property *prop(A)* is verified.

If B is an *event*, the clock of *A when B* is the intersection of ^A and B. For instance, *A when ^C* holds the values of A only at instants where C is also present.

Beware that *when B* cannot be used to synchronize a signal with B, like in :

```
CNT := (CNT $ init 0) + 1 when ^C    % ERROR !! %
```

Clock of the *when* is seen as *lower than* clock of CNT$+1 ; so the result cannot be given as value of CNT. As seen above, this must be written :

```
| CNT ^= C
| CNT := (CNT $ init 0) + 1
```

This is one of the most common errors in Signal ! But :

```
S := 1 when ^C
```

is correct and equivalent to :

```
S := 1  |  S ^= C
```

Exercices

1. Simplify the expression : *true when ^A*
   Can we detect instants where A is absent by : *when not (^A)* ?
2. Let A be a positive integer signal; produce values of A which are local maximum values. See in an exercice above how to detect these maxima.
3. Extract one value every N from an input signal : 1st, (N+1)st,.. Use *modulo* operator.
4. Let S1 and S2 be independent signals, of any type. Produce an *event* whenever S1 and S2 are present together. Beware of *and*, whose effect is to synchronize its operands !

3.4 Merging, default operator

The expression **A default B** merges signals A and B, with *priority to A* when a value from A comes at the same instant as a value from B.

```
S := A default B

       A  : ---- 1 ----------  2 ---- 3 -------------- 4 ---  5 ---

       B  : ---------- 10 --- 20 --------- 30 -- 40 -------- 50 ---

       S  : ---- 1 --- 10 ---  2 ---- 3 -- 30 -- 40 -- 4 ---  5 ---
```

The types of A and B must be compatibles. The clock of S is the *union* of clocks of A and B. Priority of *default* is lower than priorities of all other operators.

Examples of use

- Signal A can never be a constant. If B is one, the clock of *default* is not set, and must be fixed from outside. In :

  ```
  S := X default false
  ```

  clock of S must be set elsewhere, and must include instants of X. For instance :

  ```
  S ^= X ^+ Y
  ```

  So, S will be false on each instant of Y which not also an instant of X.

- We often use expressions like :

  ```
  S := A when B default C
  ```

  which looks like usual *if then else* statement. But here, clocks have to be considered : S takes the value of C not only when B is false, but also when A or B are absent (see exercice 1).

- In :

  ```
  | SIGN := 1 when X >= 0 default -1
  ```

  SIGN has no reason to have the clock of X. We should write :

  ```
  SIGN ^= X
  ```

  If we had written *when* $X < 0$ after -1, the two operands of *default* would have clocks *lower than* $\hat{}X$: so the union has the same property, and the clock equality would be incorrect. It is not the compiler's job to prove that union of conditions on X covers all possible cases !

- More than 2 signals can be merged by *default*. We often write sequences of tests like :

```
| S := val1 when cond1
    default val2 when cond2
    default val3 when cond3
```

  See CNT definition in WATCHDOG example.

Exercices

1. What is the clock of following expressions, where A and X are independent signals :
   - *f(A) when cond(X) default g(A)*
   - *f(X) when cond(A) default g(A)*

2. Give all possible cases of result for *A when B default C* : A, B, C may be present or absent, and B *true* or *false* when present (12 cases).

3. When the following expression is it correct ?

```
B := true when condition default false
```

   When can it be simplified in *B := condition* ?

4. Let A be an integer positive signal. Produce at the same clock as A a signal MAX equal to the greatest value of A encountered until now.

5. Let S1 and S2 be 2 independent signals, of any type. Produce on each input :
   - 1 if S1 is alone
   - 2 if S2 is alone
   - 3 if S1 and S2 are simultaneous.

6. Keep updated an integer signal PRESENTS counting a number of items in a room, according to 2 event signals IN and OUT. IN or OUT signals can occur at the same instant.

7. Write a *SUBSTRACT* between events A and B, present on instants of A where there is *not* also a B.

8. Produce a *ONLY_ONE* event when there are only one of S1 or S2 (of any type), but not S1 and S2 together.

3.5 Complete problem : a Distributor

We want to program a *fresh goods distributor*. It contains at every moment at most only *one item*, denoted by an integer input signal A. A user wanting an item pushes a button, giving an input event *Demand*. To give the item A, the distributor produces its integer value on output S.

Step 1 : we first suppose a good is for instance an information, and can be delivered several times. On *Demand*, the distributor gives the last A, or 0 if any A has yet been read.

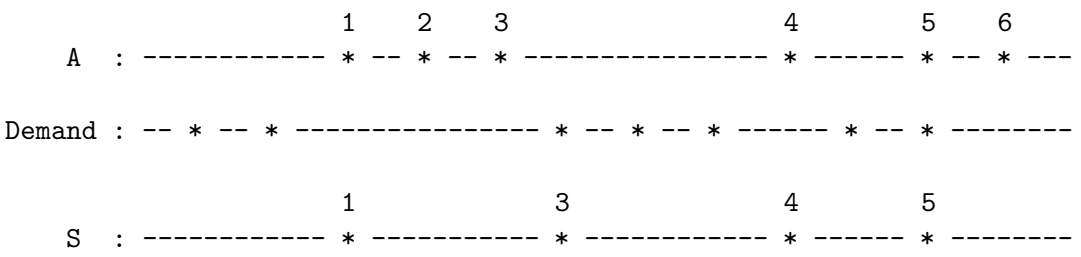
Step 2 : make sure that if a *Demand* arrives exactly on a A, it is the current (more fresh !) A which is delivered.

Step 3 : suppose now that objects cannot be stored. Every demand has to wait until the next A arrives. One (and only one) A is output if (and only if) there was at leat one demand since previous A.

Step 4 : One A is given for all demands before, *including a demand exactly on that* A.

Final step : A good is now some material object, that can be given only once. A new object is given if there was waiting demands. Else, it is stored, until a demand arrives, or a new object replaces it. If a demand arrives exactly with an object, this one is given, and possible pending demands are deleted.

Example of scenario (for final step) :



Chapter 4

More advanced features

4.1 The cell operator

4.1.1 Definition

Some systems have only a unique basic clock, and all signals are constrained to have this clock, to allow classical operations between then. This may give poorly optimized programs, but some confusing clock problems can be solved by this way.

To repeat some signal A on the instants of an other signal, the Signal Language introduces the **cell** operator :

```
C := A cell B
```

where B is a boolean expression. C contains all values of A, and also, on *true* values of B, the value of previous *A*, or some *init* value if no *A* has yet been read.

```
        A  :  --------- 1 ------------------ 2 ------- 3 ------------

        B  :  --- t --------- t -- f -- t ------- t -- f -- f -- t --

C (init 0) :  --- 0 --- 1 --- 1 ------- 1 -- 2 -- 2 -- 3 ------- 3 --
```

Clock of C is the *union* of clock of A and of *when B*.

Priority of *cell* operator is the same as priority of *when*.

As for the *delay*, the *cell* may be initialized where it is used, or when the assigned signal is declared :

```
   C1 := A cell B init 0
 | C2 := A cell B
...
   where integer C1, C2 init 1;
```

The *cell* operator is not part of the language's kernel. It can be rewritten :

```
C ^= A ^+ (when B)
C := A default (C $ init 0)
```

4.1.2 Use of cell operator

To set all signals X, Y,... to some common clock H, we write :

```
H ^= X ^+ Y ^+ ...
XX := X cell H
YY := Y cell H
....
```

To repeat values of A on the instants of another signal X, clock of X is used as a boolean signal always true :

```
AX := A cell ^X
```

If A has only to be *moved* on instants of X, without keeping A itself :

```
A_ON_X := (A cell ^X) when ^X
```

So, *A_ON_X* has the clock of *X*.

For instance, suppose a *COEF* is given at the beginning of a computation; then, COEF is applied to all following *X* :

```
( ? real COEF,  % one value  %
          X;    % sequence of reals %
  ! real Y; )   % Y = COEF * X for all X %

(| Y := ((COEF cell ^X) when ^X ) * X |)
```

4.1.3 Exercices :

Can also be solved without the *cell* !

1. *Pixels* from a sequence of pictures are given as boolean input, line after line, and for each line, by column. A picture has *NL* lines and *NC* columns (constant parameters). The last pixel of a picture is followed by the first one of the next picture. Produce for each pixel its *LINE* and *COLUMN* number.

   In a separate compiling, effective values of NL and NC, for instance 4 and 8, are given in a file *picture.par*, containing *4,8*

2. To be added, integers A and B must be synchronous. Suppose they can arrive at "slightly" different instants ; we want nevertheless to add them !

```
  A : -- 1 ------- 3 --- 5 ------ 7 --- 9 ---- 11 ----- 13 --
  B : ----- 2 -- 4 ----- 6 --- 8 -------- 10 ---- 12 -- 14 --
SUM : ----- 3 ---- 7 -- 11 ----- 15 ----- 19 ---- 23 -- 27 --
```

3. A two-tracks *railway crossing* is protected by a barrier. Each track orders closing for itself, through a boolean signal *CLOSEi*, *true* to order closing, *false* to allow opening. Write the barrier's controller making the synthesis between *CLOSE1* and *CLOSE2* signals.

```
CLOSE1 : --T--------F-----T-----F--T--F-----
CLOSE2 : -----T--F-----T--F--T--F--T-----F--
CLOSE  : --T--------F--T--------F--T-----F--
```

   A program with counter can be written, but the *cell* allows a purely boolean solution.

4. *Resynchronizing :* let A be some signal, and H be a clock, much faster than A. To resynchronize A is to *move* it (without repeat) on the next tick of H. Moving is not necessary if A is already on a tick of H.

```
 A  : - 1 -------- 2 -------------- 3 --- 4 ----
 H  : -----*--*--*---*--*--*--*--*--*--*----*---
AH  : ---- 1 ------- 2 ------------ 3 ----- 4 --
```

4.2 Modularity

A process may be used as a sub-process by other parts of a system. The calling process set constant parameter values (if any), gives input signals, and uses output values. For instance, the WATCHDOG process could be called by :

```
PB_TIME := WATCHDOG {5} (DEMAND, when OVER, ^SECONDS)
```

Parameter value may be any constant expression; it may include local parameters, but not signals. Effective input signals are expressions on signals. If called process has only one output, the calling statement can be used as a function call, in any signal expression.

4.2.1 Examples of syntax

Other possible forms are presented in following example :

```
process MODU =
  ( ? integer A;
    ! integer S;)

(| BOOL ^= A
 | S := -1 when BOOL
       default 2 * ONE (A) + 1
       default INT
 | (INT, BOOL) := NO_IN {false} ()
 | NO_OUT (INT, BOOL)
 |)
where
  integer INT; boolean BOOL;
```

```
  process ONE =
    ( ? integer I;
      ! integer RES; )
    (| RES := I/2 when I modulo 2 = 1 |)
  ;
  process NO_IN = {boolean B0;}
    ( ?
      ! integer I; boolean B; )
    (| I := 0
     | ZB := B$ | B := ZB |)
    where
      boolean ZB init B0;
    end % NO_IN %;

  process NO_OUT =
    ( ? integer I; boolean B;
      ! )
    (| I ^= when not B |)
end % MODU %;
```

Exercice : what are the values output by this process ?

4.2.2 Sub-process Counter modulo N

A classical example of modularity is the counting of time, using some instances of a *Counter modulo N*.

An input signal *TICK* is supposed to occur each second. The process produces at these instants the day, hour, minute and second numbers.

The *CNT_MOD* process has two events as input : one to increase counter, an other to obtain (on output *CNT*) the current value of the counter. The process also emits an event when the counter becomes *RESET* to zero.

```
process BIG_BEN =
  ( ? event TICK;
    ! integer DAY, HOUR, MINUTE, SECOND; )

 (| (SECOND, NEW_MINUTE) := CNT_MOD {60} (TICK, TICK)
  | (MINUTE, NEW_HOUR  ) := CNT_MOD {60} (TICK, NEW_MINUTE)
  | (HOUR  , NEW_DAY   ) := CNT_MOD {24} (TICK, NEW_HOUR)
  | DAY ^= TICK
  | DAY := DAY$ + 1 when NEW_DAY default DAY$ init 1
  |)
 where
   event NEW_MINUTE, NEW_HOUR, NEW_DAY;

   process CNT_MOD = % Counter modulo N %
```

```
     { integer N;}
     ( ? event EV_OUT,   % when Counter must be output %
               EV_INC;   % when Counter must be increased %
       ! integer CNT;
         event RESET; )

     (| CNT ^= EV_OUT ^+ EV_INC
      | CNT := (ZCNT+1) modulo N when EV_INC
                default ZCNT
      | ZCNT := CNT$ init 0
      | RESET := when CNT = 0 when EV_INC
      |)
    where
       integer ZCNT;
    end % CNT_MOD %;
end % BIG_BEN %;
```

4.3 Oversampling

We have already seen that clocks of input signals may be constrained :

```
    process ADD =
      ( ? integer A, B;
        ! integer S; )
    (| S := A + B |)
```

Clocks of A and B are requested to be equal.

Clocks of input signals may be related in a more complex way :

```
      ( ? integer X, Y;
      ....
     |  X ^= when Y = 0
```

The clock of Y is the "main clock". A value for X is read only when a null value is read for Y. In a separate run, no clock file has to be given.

The clock of some input can even be related to a local signal :

```
    ( ? integer X;
   ...
    (| FLIP := not (FLIP$ init false)
     |  X ^= when FLIP
   where
     boolean FLIP; ...
```

So, the faster clock of a process is not always an input clock. The FLIP signal is present on an "intermediate" instant between two readings of X. The place of this instant has not to be more defined. If there was several instants between two inputs, they are supposed to have enough time to all take place. This "addition of instants" is named *oversampling.*

This faster signal can be synchronized with other signals, and its clock can then be more precisely set.

Of course, there are limits in constraints that can be imposed on clocks. For instance,

```
when A=0 ^= when B=0
```

is not accepted, because it's impossible at compile time to guarantee that A and B will always be null at the same time.

Even a constraint like :

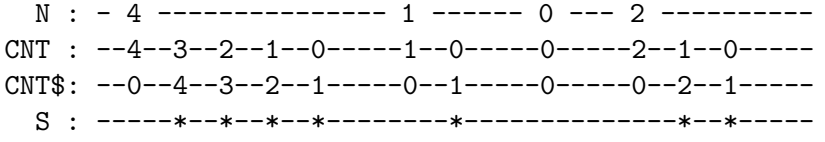
```
A default B ^= when condition
```

may be rejected. We'll see in the *compilation* chapter how the clock equations are managed.

Exercices :

1. Let A be an integer signal, whose value is between 0 and 99. Produce for each A two integers *DIGIT*, one with the *tens*, and the following one with the *units*. For input *35,7,10,..*, we want : *3,5,0,7,1,0,..*

2. Let N be a positive or null integer signal. Produce *after each N*, and *before the next N*, a sequence of N events. Use a decreasing counter, and synchronizes input N on some value of the *delayed* counter (to avoid circularities).



Chapter 5

Applications

We study here larger problems, intended to show some programming methods in Signal.

5.1 Interval between events

Let *START* and *FINISH* be two events separated by some amount of time.

5.1.1 Duration between START and FINISH

It can be measured only as a number of pulses of a clock event H. CNT will be a counter, present on every event, but increased only on H. It is reset on START, and its value on FINISH gives the duration.

```
  | CNT ^= START ^+ FINISH ^+ H
  | CNT := 0 when START
            default CNT$ + 1 when H
            default CNT$
  | DURATION := CNT when FINISH
```

```
  START : -----*-------------*-----------------------
 FINISH : -------------*-------------------------*----
     H  : -*---*---*---*---*---*---*---*---*---*---*--
    CNT : -?---0---1---2---3-0-1---2---3---4---5-5-6--
DURATION: ------------ 2 ---------------------- 5 ---
```

If START and FINISH occur exactly on a H, the time measurement is exact. Else, DURATION is the number of H between START and FINISH, bounds excluded ; the error is less than one interval.

5.1.2 Is S present between START and FINISH ?

The status *in/out* of any instant must be memorized, in a boolean signal *MEM* :

```
  | MEM ^= START ^+ FINISH ^+ S
  | MEM := START default not FINISH default (MEM$ init false)
```

```
          % true when START default false when FINISH default MEM$ %
  | IN := MEM when S
```

What about signals coming exactly on START or FINISH ? With definition above, a S on START is *IN* the interval, a S on FINISH is *OUT*.

Exercice : Modify this definition to obtain the 3 other cases : interval opened on left, and/or closed on right.

5.2 Automata

A *finite state automaton* is often used to modelize real-time systems. An automaton has some states S_i ; one of them is the initial state S_0. When an event e_i occurs, the automaton may change its current state, or loop on this same state, and some appropriate action may be executed.

In Signal, a name S holds the arrival state of a transition, and its delayed value ZS is the departure state. The *clock of S* is the union of clocks of all input events. Actions are managed by changing values of other signals. A transition may also take place when such a signal reaches a given value.

In the *asynchronous approach*, automata may be non-deterministic : when 2 events occur at the same time, the transition to take is undefined. The Signal programmer must specify for a given state in which order input events are considered (default operator).

Example : A passage may be lightened by pushing a switch. Light is put off by the same switch, or automatically after 1 minute. The process doing that may be written as a two-state automaton : $S = 1$ when light is OFF, $S = 2$ when ON.

```
process LIGHT =
  ( ? event SWITCH, H;    % H every second %
    ! event PUT_ON, PUT_OFF;
  )
             % State changes %
 (| S ^= SWITCH ^+ H ^= CNT
  | S := 3 - ZS when SWITCH
       default 1 when ZS = 2 when ZCNT = 1
       default ZS  % loop on current state %
  | ZS := S $ init 1
      % Actions on transitions %
  | CNT := (60 when ZS = 1 default ZCNT - 1) when S = 2
         default ZCNT
  | ZCNT := CNT $ init 0
  | PUT_ON := when S = 2 when ZS = 1
  | PUT_OFF := when S = 1 when ZS = 2
  |)
where
   integer S, ZS, CNT, ZCNT;
end % LIGHT %;
```

The counter CNT is useful only in state 2; but its delayed value ZCNT has the same clock, and is also used when S becomes 1. In any way, the defining equation of CNT is recursive; so, its clock has to be set from outside. It's easier to give it the clock of S.

Exercices.

1. Add to the previous process a boolean signal *PRESENCE*, true when somebody is in the passage. It is normally a continuous signal : we suppose it is sampled every second, with the clock of H. This signal can even be used as a source of time, so H becomes no more necessary.

   PRESENCE is used only to delay putting off the light, if it was on, until there is nobody during 10 seconds.

2. A *STOPWATCH* is a classical example of system modelized by an automaton.

   It has two buttons : **SS** to *Start* or *Stop* counting of time, and **LAP** (intermediate time) which stops *Display* without stopping the time counter. A new push on *LAP* makes current counter be displayed. Even when displaying is stopped, time counting may be started/stopped by SS.

   When time counting is stopped, and display is free, LAP resets the stopwatch.

   Add a source of time, which is supposed to have a sufficient accuracy. Displayed values are numbers of pulses of this input signal.

5.3 Picture analysis

An picture is a sequence of pixels, boolean values obtained by reading the picture line after line, and for each line, by column. A *PIXEL* is true when it is a point in an object, false outside.

A picture has *NL* lines and *NC* columns (constant parameters). So, the point just *before* the current point is *PIXEL $* ; the point *above* is *PIXEL $NC*.

If we have a sequence of pictures, and the first pixel of a picture follows immediately the last one of the previous picture, the same point in this previous picture is *PIXEL $(NL*NC)*.

Exercices : We have seen in a previous exercice how to keep updated *line* and *column* number of the current pixel.

1. A pixel is *INTO* an object if it's *true*, and its 4 surrounding pixels (above, under, left, and right) are also *true*. A pixel on the border is never INTO. Produce only coordinates of pixels which are INTO.

2. Supposing pictures are successive views of the same scene, detect only points having *moved*, ie whose *INTO* status has changed since last picture.

3. We assume a picture contains only separate *convex* objects : any line crosses it only once. Detect these points where the sequence of pixels enters an object, or leaves it. Ouput the coordinates of the first and the last point of an object crossing, by a horizontal line. Such a line can cross several objects. An object can be "cut" by the left or right border.

4. Detect for each object its *first crossing* : between the first pixel of an object crossing, and the last one, it is the first crossing if we found no true pixel just above each (true) pixel of this crossing. It's also a first crossing if the current line is *line 1.* True pixels which are neighbours only along a diagonal are not assumed to belong to the same object (see example below).

5. Use this property to detect each object only once. Give the coordinates of one point of the object (anyone), and, at the end of the picture, the *number* of objects.

The following sequence of pictures contains 7 objects ; false values are replaced by a dash for seek of lisibility :

```
- t - - t - t t        t - - ...
t t - t - t t -        - -   ...
- - t - - t - t        ...
t t t t t - - t
```

Here is shown the same pictures with the number of each object, in their order of detection.

```
- 1 - - 2 - 3 3        7 - - ...
1 1 - 4 - 3 3 -        - -   ...
- - 5 - - 3 - 6        ...
5 5 5 5 5 - - 6
```

5.4 A railway level crossing

A two-tracks railway level crossing is an interesting real-time system, where modularity can be exercised.

5.4.1 The problem

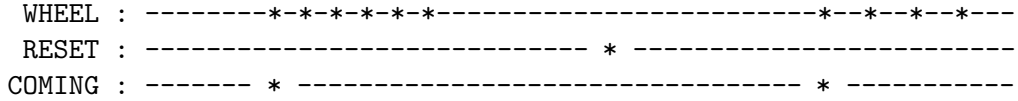
The level crossing area is protected by barriers, that must be closed in time on the arrival of a train, on one or the other track. They remain closed until all trains have leaved the area, and also if some railway vehicle is *present* in this area (maybe stopped).

The barriers must be closed 30 seconds before the expected time of arrival of a train. When the area becomes free, barriers could be opened, but it's not secure to open them for less than 15 s. So the controller must be warned 45 s before a train arrives.

Since the speed of trains may be very different, this speed has to be measured, by detecting the train in two points separated by a known distance. A first detector is placed 2500 m before the crossing, and a second one 100 m after this first. So the time between *Detect1* and *Detect2* has to be multplied by 24 to obtain the remaining time to reach the crossing (at constant speed). A *Detect3* is placed after the crossing area, and records the train's leaving.

5.4.2 The Detect process

The passage of a train is detected by a mechanical device, producing a pulse (type *event*) every time a wheel runs on it. The *Detect* process receives all these pulses, but warns the controller only once, on the first wheel. All following pulses are ignored. So, the detector has to be *reset*, after the full train has gone, to be able to detect a new train.



Write in Signal this process DETECT.

5.4.3 A Track controller

On each track, a controller receives *COMING* signals from the 3 detectors. From DETECT1 and DETECT2, it must compute the train's speed, and warn the barriers 45 seconds, then 30 seconds before expected time of arrival at crossing. So, this process needs a source of time. At the maximum speed of 180 km/h, the 100 m between DETECT1 and DETECT2 are covered in 2 s. So, a clock pulse every 0.1 s would be of good accuracy.

From DETECT3, barriers controller knows the head of the train is leaving the crossing area.

An automaton would be useful to modelize this Track process.

It's also the *Track*'s work to *reset the detectors*. Here occurs an interesting circularity problem. We are tempted to reset 1 and 2 with DETECT3, and 3 with DETECT1 :

```
| DETECT1 := DETECT (WHEEL1, DETECT3)
| DETECT3 := DETECT (WHEEL3, DETECT1)
```

But this program is rejected, because the *RESET* signal is used to define DETECT output, so these definitions contain a circularity. It's the same is we use for RESET some expression including a state of the automaton, or a counter, whose definitions rely on DETECT.

A good solution would be to reset a detector after a fixed amount of time without any wheel. If we insist on a switch between beginning and end of the area, we must use a boolean signal, true on each wheel at the beginning, false at the end. Reset can occur when this signal changes its value.

5.4.4 The Barriers controller

It receives 3 sorts of warning signals from the two tracks. It also receives the *PRESENCE* boolean signal, true if the crossing area is occupied. We suppose this continuous signal is sampled every 0.1 second. So the clock of PRESENCE could be given as source of time for Track processes.

With all these informations, the controller has to fix the opened/closed status required for the barriers. When this status changes, it has to order their effective closing or opening, for instance through a boolean output.

Bibliography

[1] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, Claude Le Maire. *Programming real-time applications with Signal.* Proceedings of the IEEE, v. 79, p. 1321-1336, Sept 1991.

[2] Loïc Besnard, Thierry Gautier, Paul Le Guernic. *Signal V4 - INRIA version: Reference Manual.* Access through the ESPRESSO web site.

[3] *ESPRESSO web site :* http://www.irisa.fr/espresso/welcome_english.html

[4] Nicolas Halbwachs et al. *The synchronous data flow programming language LUSTRE.* Proceedings IEEE, 79-9, 1991.

[5] Boussinot, De Simone. *The ESTEREL language.* Proceedings IEEE, 79-9, 1991.

Chapter 6

Solutions of exercices

Delay operator

```
process MACHINE =
  ( ? event BUTTON;
    ! boolean BUSY;)
 (| BUTTON ^= BUSY
  | BUSY := not (BUSY $ init false)
  |)

process MEANS =
  {integer N;}
  ( ? real A;
    ! real S, MEAN, MEAN_N; )

 (| S := (S$ init 0.0) + A      % synchronizes S and A %

  | CNT := (CNT$ init 0) + 1
  | MEAN := S / real(CNT)   % synchronizes CNT and S (and also A) %

  | SN := (SN$ init 0.0) - (A $ N init [ {to N}: 0.0 ]) + A
  | MEAN_N := SN / real(N)
  |)
 where
  integer CNT;
  real SN;
end % MEANS %;

% FIRST: true on first A, false on following ones %
  | FIRST := FCA $ init true
  | FCA := not (^A) % always false, with clock of A %
 where
   boolean FCA;
```

Unary when operator

```
(| ZA := A $ init 0
 | UP := A > ZA or A = ZA and ZUP
     % UP takes the clock of A %
     % UP's value is kept when flat %
 | ZUP := UP $ init true
 | SUMMIT := when (ZUP and not UP)
              % we were going up, and now going down %
 |)
where
 integer ZA;
 boolean UP, ZUP init true;
end;
```

Binary when operator

Exercice 1 : *true when ^A* is simplified as *^A*.

Exercice 2 :

```
  | LOCALMAX := A $ when (ZUP and not UP)
ou
  | LOCALMAX := A $ when SUMMIT
```

Exercice 3 :

```
  (| CNT ^= A
   | CNT := (CNT $ init 0) modulo N + 1
   | S := A when CNT = 1
   |)
 where
   integer CNT;
```

Exercice 4 : The expression *^S1 and ^S2* is not correct, for it imposes the same clock for S1 and S2.

Solution: **^S1 when ^S2**

Default operator

Exercice 1 : the two expressions have the clock of A.

Exercice 2 : If A present and B true : **A** (2 cases : C present or not)
else if C present : **C** (5 cases)
else **bottom** (5 cases).

Exercice 3 :

```
B := true when condition default false
```

is correct only when clock of B is fixed from outside, and if the compiler can proof that this clock includes (or is equal to) clock of condition.

Simplification: *true* can always be deleted. $B := condition$ is equivalent only if B and *condition* have exactly the same clock.

Exercice 4 :

```
| ZMAX := MAX $ init 0
| MAX := A when A > ZMAX default ZMAX
    % A > ZMAX makes A and ZMAX (and also MAX) synchronous %
```

Exercice 5 :

```
| S := 3 when ^S1 when ^S2
      default 1 when ^S1 default 2 when ^S2
```

Exercice 6 :

```
| PRESENTS ^= IN default OUT
| ZPRESENTS := PRESENTS $ init 0
| PRESENTS := ZPRESENTS when IN when OUT
         default ZPRESENTS + 1 when IN
         default ZPRESENTS - 1 when OUT
```

The separate definition of clock is necessary : Clock of right hand expression is the intersection of clocks of *ZPRESENTS* and *IN default OUT*. It equals to *PRESENTS* only if *ZPRESENTS* is already equal to *IN default OUT*. The last *when OUT* may be omitted.

Definition of PRESENTS may be shortened :

```
| PRESENTS := (ZPRESENTS when OUT default ZPRESENTS + 1) when IN
         default ZPRESENTS - 1
```

Exercice 7 : events' substract A - B

```
| NOT_B_A := not B default A
| SUB_AB := A when NOT_B_A
```

Exercice 8 :

```
| TOGETHER := ^S1 when ^S2
| ONLY_ONE := when (not TOGETHER default ^S1 default ^S2)
```

Distributor

```
process DISTRIBUTOR =

( ? integer A;
    event Demand;
  ! integer S;)

( | S := A when Demand
        default A when WAIT$ init false
        default MEM_A when AVAIL_A$ init false when Demand
```

```
  | AVAIL_A ^= A ^+ Demand ^= WAIT ^= MEM_A

  | AVAIL_A := not ^S default ^A
              default AVAIL_A$ init false
  | WAIT := not ^S default Demand
            default WAIT$ init false
  | MEM_A := A default MEM_A$
  | )

where
  boolean AVAIL_A, WAIT;
  integer MEM_A;
end % DISTRIBUTOR %;
```

The cell operator

Exercice 1 :

```
process PICTURE =
  {integer NL, NC;} % size of the picture %
 ( ? boolean PIXEL;
   ! integer LINE, COLUMN;)
 (| PIXEL ^= COLUMN
  | COLUMN := (COLUMN$ init 0) modulo NC + 1
  | LINE := (((LINE$ init 0) modulo NL + 1) when COLUMN=1) cell ^PIXEL
  |)
```

For the clock of LINE, left operand of the *cell* has a clock included in PIXEL's, due to the *when*. So, the clock of right hand expression is well defined as PIXEL.

Exercice 2 :

```
process A_NEAR_B = % Sum of A and B arriving at "close" instants %
  ( ? integer A, B;
    ! integer SUM; )

 (| SUM := AA + BB when SECOND
  | AA := A cell ^B
  | BB := B cell ^A
  | SECOND ^= A ^+ B
      % SECOND is true if A and B together, or on the second one %
  | SECOND := ^A when ^B
              default not (SECOND$ init true)
  |)
 where
   integer AA, BB;
   boolean SECOND;
 end % A_NEAR_B %;
```

Exercice 3 :

```
process BARRIER =  % of a two-tracks railway crossing %
  ( ? boolean CLOSE1, CLOSE2;  % true to close, false to allow opening %
    ! boolean CLOSE;)  % effective closing/opening %
(| STATE := (CLOSE1 cell ^CLOSE2 init false)
           or (CLOSE2 cell ^CLOSE1 init false)
 | CLOSE := STATE when STATE /= STATE$ init false
 |)
where
  boolean STATE;
end % BARRIER %;
```

The **or** is allowed between these cell expressions, because their clocks are equal.

Exercice 4 : *Resynchronizing*

```
process RESYNC = % moves A on the nearest H %
  ( ? integer A;
      event H;
    ! integer AH;)
(| AH := (A cell FIRST_H) when H
 | FIRST_H := ON_A $ init false
 | ON_A := not H default ^A
 |)
where
 boolean FIRST_H, ON_A;
end % RESYNC %;
```

FIRST_H is true only on the first H following a A, except when the A was exactly on a H : in this latter case, it remains false. So, *A cell FIRST_H* holds all values of A, plus the previous A on the first H following a A without H. We keep only its values on a H.

```
      A  : - 1 -------- 2 -------------- 3 --- 4 ----
      H  : -----*--*--*---*--*--*--*--*--*--*----*---
    ON_A : --t--f--f--f-t-f--f--f--f--f--f--f--t-f---
 FIRST_H : --f--t--f--f-f-t--f--f--f--f--f--f--f-t---
     AH  : ---- 1 ------- 2 ------------ 3 ----- 4 --
```

Modularity

Values output by process MODU :

From process *NO_IN*, INT is set to 0, and BOOL to the parameter, here *false*. So, the first branch of default is never taken.

ONE(A) is $(A-1)/2$ when A is odd, absent otherwise. So the second branch is A when A is odd.

From *NO_OUT*, INT has the clock of *not BOOL*, i.e. of BOOL, i.e. of A. So, the third branch is taken when A is not odd.

Finally, output has the clock of A, and is A when A is odd, and 0 when even.

Oversampling

Exercice 1 :

```
process DIGITS = % output tens, then units, from A %
  ( ? integer A;
    ! integer DIGIT;)
(| TENS := not TENS$ init false
 | A ^= when TENS
 | DIGIT := A/10 default (A cell (not TENS)) modulo 10
 |)
where
  boolean TENS;
end % DIGITS %;
```

Exercice 2 :

```
process N_INTER = % N intermediate events after input N %
  ( ? integer N;
    ! event I;)
(| CNT := N default ZCNT - 1
 | ZCNT := CNT$ init 0
 | N ^= when ZCNT = 0
 | I := when ZCNT > 0
 |)
where
  integer CNT, ZCNT;
end % N_INTER %;
```

S in an interval

```
  | MEM ^= START ^+ FINISH ^+ S
  | MEM := START default not FINISH default ZMEM
  | ZMEM := MEM $ init false
  | IN := MEM when S
```

gived START in the interval, and FINISH out.

```
  | IN := ZMEM when S       % makes a shift: START out, FINISH in %
...
  | IN := (FINISH default MEM) when S      % START in,  FINISH in %
...
  | IN := ((not START) default MEM) when S % START out, FINISH out %
```

Lightening a passage

H is replaced by *PRESENCE* in input list, and in clock definitions. *CNT* definition becomes :

```
  | CNT := (10 when ZS=1 default 10 when PRESENCE default ZCNT-1) when S = 2
          default ZCNT
```

The first term is in cases where PRESENCE is not immediately true when SWITCH is pushed on.

Stopwatch

The 4 states are best defined as the product of two booleans : RUNNING and DISPLAY_ON.

```
process STOPWATCH =
 ( ? event SS, LAP, H;
   ! integer DISPLAY;)

(| CNT ^= RUNNING ^= DISPLAY_ON ^= DISPLAY ^= SS ^+ LAP ^+ H

 | RUNNING := not ZRUNNING when SS default ZRUNNING
 | DISPLAY_ON := not ZDISPLAY_ON when LAP when RUNNING
             default LAP when not RUNNING  % always set true if not running %
             default ZDISPLAY_ON
 | CNT := 0 when LAP when ZDISPLAY_ON and not RUNNING
         default ZCNT + 1 when H when RUNNING default ZCNT
 | DISPLAY := CNT when DISPLAY_ON default DISPLAY$ init 0
 | ZCNT := CNT$ init 0
 | ZRUNNING := RUNNING$ init false
 | ZDISPLAY_ON := DISPLAY_ON$ init true
 |)
where
  boolean RUNNING, ZRUNNING, DISPLAY_ON, ZDISPLAY_ON;
  integer CNT, ZCNT;
end % STOPWATCH %;
```

Picture analysis

See in *The cell operator* how to compute LINE and COLUMN.

```
 | INTO := LINE>2 and COLUMN>1 and COLUMN<NC and
     PIXEL and PIXEL$(NC-1) and PIXEL$NC and PIXEL$(NC+1) and PIXEL$(2*NC)
 | L_INTO := LINE$NC when INTO
 | C_INTO := COLUMN$NC when INTO
```

The INTO boolean for a given pixel can only be set NC pixels after. In INTO definition, *and operators* are sufficient, because all operands have the same clock. We assume the evaluation stops as soon as an operand is false, so the initial values in delays have not to be set.

Movements are detected by :

```
  | MOVED := when (INTO /= INTO $(NL*NC))
```

In a picture containing convex objects, FIRST and LAST are pixels where a line enters and leaves an object :

```
  | FIRST := PIXEL and (COLUMN=1 or not PIXEL$)
  | LAST := (PIXEL$ init false) and (COLUMN=1 or not PIXEL)
  | L_LAST := LINE$ when LAST
  | C_LAST := COLUMN$ when LAST
```

In fact, LAST is the first pixel *following* the object crossing.

We are crossing an object for the first time if we are in an object (*PIXEL true*), and if there is no *true* pixel just above.

```
  | NEW := PIXEL and (LINE=1 or (not PIXEL$NC) and (FIRST or NEW$))
```

Inside each object, NEW remains true on first line. Otherwise, it becomes (and remains) false in case of true pixel above; else, it is true on first pixel, and may keep this value until last pixel.

```
  | NEW_OBJECT := when LAST and NEW$
  | L_OBJECT := LINE$ when NEW_OBJECT
  | C_OBJECT := COLUMN$ when NEW_OBJECT
  | NUM_OBJECT := 1 + NUM_OBJECT$ init 0
  | NUM_OBJECT ^= NEW_OBJECT
```

Railway level crossing

```
process DETECT =
  ( ? event WHEEL, RESET;
    ! event COMING;)

 (| READY := RESET default not WHEEL
  | COMING := WHEEL when READY$ init true
  |)
where
  boolean READY;
end % DETECT %;
```

RESET is the first operand of READY, because it **must** be catched, even if a wheel passed at the same time.

```
process TRACK =
  ( ? event WHEEL1, WHEEL2, WHEEL3, TENTHS;
    ! integer CLOSE; % 1=crossing in 45s, 2=in 30s, 3=gone %
  )
 (| S ^= CNT ^= DETECT1^+ DETECT2^+ DETECT3^+ TENTHS
                % States of the automaton %
  | S :=    -1 when DETECT1
     default 0 when DETECT2
```

```
     default 1 when ZS = 0 and CNT <= 450
     default 2 when ZS = 1 and CNT <= 300
     default 3 when DETECT3
     default ZS
  | ZS := S$ init 3
                % counting of time, unit assumed: 0.1 sec %
  | CNT := 0 when DETECT1
       default 240*ZCNT when DETECT2
       default ZCNT+1 when ZS = -1
       default ZCNT-1 when ZCNT > 0
       default 0
  | ZCNT := CNT$ init 0
  | CLOSE := S when S /= ZS and S > 0

  | BUSY := WHEEL1 default not WHEEL3
  | ZBUSY := BUSY$ init false
  | DETECT1 := DETECT (WHEEL1, when ZBUSY and not BUSY)
  | DETECT2 := DETECT (WHEEL2, when ZBUSY and not BUSY)
  | DETECT3 := DETECT (WHEEL3, when BUSY and not ZBUSY)
  |)
where
  integer S, ZS, CNT, ZCNT;
  event DETECT1, DETECT2, DETECT3;
  boolean BUSY, ZBUSY;

process DETECT = ...
end % TRACK %;
```

Numbering of states S is choosen to be equal to output values for CLOSE. Initial and rest state is 3.

CNT counts tenths of seconds, increasingly between DETECT 1 and 2, decreasingly after. Changing of states at 45s, then 30s before crossing, are tested by <= operators : if a train had an excessive speed, the delays could be over, but actions have nevertheless to be taken.

For detector reset, boolean BUSY reflects the fact that a train (more exactly, its first wheel) is present in the area.

```
process BARRIER =
  ( ? integer CLOSE_A, CLOSE_B;  % closing orders from tracks A or B %
      boolean PRESENCE;        % sampled every 0.1 s %
    ! boolean CLOSE;      % effective action on barriers %
  )
 (| COMMON := ^CLOSE_A default ^CLOSE_B default ^PRESENCE
  | C_A := CLOSE_A cell COMMON
  | C_B := CLOSE_B cell COMMON
  | IS_CLOSED := PRESENCE cell COMMON
```

```
            or C_A = 2 or C_B = 2     % imperative closing %
            or WAS_CLOSED and (C_A = 1 or C_B = 1)
  | WAS_CLOSED := IS_CLOSED$ init false
  | CLOSE := IS_CLOSED when IS_CLOSED /= WAS_CLOSED
  |)
where
  event COMMON;
  integer C_A, C_B;
  boolean IS_CLOSED, WAS_CLOSED;
end % BARRIER %;
```

At each moment, the closed/opened status of barriers has to be known. The *or* operators are necessary to ensure that every closing order will be catched, even when it is simultaneous with another order allowing open. So, all events have to be put to the same COMMON clock. Main events are PRESENCE, or CLOSING emitted 30 s before crossing (value 2). When CLOSE is 1 (crossing in 45 s), the barrier keeps only its previous status.

Main process is the following :

```
process RAILWAY =   % Two-tracks railway level crossing %
  ( ? event A1, A2, A3, B1, B2, B3;  % wheels on tracks A and B %
      boolean PRESENCE;
    ! boolean CLOSE;)

(| CLOSE_A := TRACK (A1, A2, A3, ^PRESENCE)
 | CLOSE_B := TRACK (B1, B2, B3, ^PRESENCE)
 | CLOSE := BARRIER (CLOSE_A, CLOSE_B, PRESENCE)
 |)
where
  integer CLOSE_A, CLOSE_B;

process TRACK =...
process BARRIER = ...
end % RAILWAY %;
```