# SIGNAL Manual

Patricia Bournai, Bruno Chéron, Thierry Gautier, Bernard Houssais, Paul Le Guernic

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *SIGNAL MANUAL*

Patricia BOURNAI, Bruno CHÉRON, Thierry GAUTIER, Bernard HOUSSAIS and Paul LE GUERNIC

**N° 1969**

Septembre 1993

PROGRAMME 2

Calcul symbolique, programmation et génie logiciel


ISSN 0249-6399

# SIGNAL MANUAL

Patricia BOURNAI, Bruno CHÉRON, Thierry GAUTIER, Bernard HOUSSAIS, and Paul LE GUERNIC

Programme 2

Projet EP-ATR

Rapport de recherche n˚1969

88 pages

**Abstract:** SIGNAL is a language designed for the synchronous programming of real time systems (signal processing, control-command...). A SIGNAL program specifies an automaton via a system of dynamic equations involving “signals”. Such systems of equations can be organized hierarchically in modules. A “signal” is a sequence of data which has a clock associated with; this clock defines the instants when this signal is available. Clocks need not to be related via fixed sampling rates, but can rather have sampling rates which depend on local data or external events (interruptions). This report is a manual for the SIGNAL language; it presents the syntax of the H2 version of the language and introduces its semantics. Examples are given and in the appendix A, a complete sample session using the SIGNAL compiler is provided.

**Key-words:** SIGNAL, real-time language, syntax, reference manual.

*(Résumé : tsvp)*

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : (33) 99 84 71 00 - Télécopie : (33) 99 38 38 32

# MANUEL SIGNAL

**Résumé :** SIGNAL est un langage conçu pour la programmation synchrone de systèmes temps réel (traitement du signal, systèmes de contrôle-commande...). Un pogramme SIGNAL définit un automate à partir d'un système d'équations dont les variables sont les signaux. Ces équations peuvent être organisées en sous-systèmes (ou processus). Un signal est une suite de valeurs à laquelle est associée une horloge qui spécifie les instants auxquels les valeurs sont disponibles. Un programme exprime ainsi les relations fonctionnelles et temporelles qui existent entre tous les signaux mis en oeuvre. Ce rapport est un manuel du langage SIGNAL ; il présente la syntaxe du langage dans sa version H2 et introduit sa sémantique. Un exemple complet d'utilisation du compilateur est donné en annexe A.

**Mots-clé :** SIGNAL, langage temps-réel, syntaxe, manuel de référence.

# Chapter I. Introduction

The language SIGNAL has been defined at INRIA/IRISA in collaboration and with support from CNET. This document is a summary of the characteristics of SIGNAL version H2. This does not constitute a reference manual nor a user's guide.

## I-1 Main features of the language

A program written in SIGNAL defines an automaton originating from a system of equations the variables of which are identifiers of the signals. These equations can be organized in subsystems (or processes). A model of process, named by an identifier, is a subsystem which can have several contexts of use; it is referred by its name.

### I-1.1 Signals

A signal is a sequence of values, to which a clock is associated. All the values of a signal appear in the same predefined domain (pure signals, Booleans, Integers, Reals, Complex numbers) or in a domain defined in the program (Arrays). Its clock defines the set of instants where a signal has a value.

### I-1.2 Events

A communication associates a value with a variable, at a logical instant of the program. An event is a set of simultaneous communications that forms an automaton transition. In an event, a variable can be without an associated value, the signal is then called absent and its value is denoted $\perp$. An event includes at least one communication.

The determination of the presence of a signal in an event results from the solving of an equation system in $\mathcal{F}_3$, the body of integers modulo 3.

The value associated with a variable in an event is the result of the evaluation of its definition expression (that cannot be implicit: circular definitions of signals that are not boolean are forbidden).

### I-1.3 Models

A model associates an identifier with a system of equations which may have local variables, sub-models and external variables (free variables). The parameters of models are constants for the size of arrays or initial values of signals.

A model can be defined outside the program; then it can only be seen through its interface.

The invocation of a model defined in a program is equivalent to the replacement of this invocation by the associated system of equations (macro substitution).

## I-1.4 Causal relations

A real-time program should respect the principle of causal relations, the value of an event can not depend on the value of a future event. SIGNAL respects this principle through the implicit management of time: the user disposes of expressions that allow him to make references to past values or current values of a signal, but not to a future value.

# I-2 Presentation form

This document presents the syntax and introduces the semantics of SIGNAL. There are two classes of terms for the description of the syntax of the language:

- the lexical (or terminal) structures defined, at a lexical level, through rules in a grammar the vocabulary of which is a set of characters; no implicit character (e.g. separator) is authorized in the terms built according to these rules.

- the syntax structure defined at a syntactical level, through rules in a grammar the vocabulary of which is a set of terminals; between two terminals, it is possible to insert separators.

Each language unit is introduced and then described, individually or by category, with help of all or parts of the following paragraphs.

1. **Context free grammar**
It gives the Context free grammar of the considered structure according to one of these following forms:

**# STRUCTURE ::=**
*** Derivation 1***
*** Derivation 2***
*** .....***
**# Terminal::=**
*** Derivation 1 ***
*** Derivation 2 ***
*** .....***

**Derivation 1, Derivation 2** are ways of writing the variable **STRUCTURE** (the same for the variable **Terminal**).
Each **Derivation** is a sequence of *elements* which can be of the following kinds:
***** a **set** of characters, written in this typography (only at lexical level),
***** a **terminal** (formed by letters), in this typography,
***** a terminal **symbol** (formed by other allowed characters), in this typography,
***** a **Terminal**, in this typography,

***** a syntactical **STRUCTURE**, in this typography (only at syntactical level),

***** a sequence of elements with or without separators, respectively in the following forms:
- {*element* **symbol**...}
- {*element* ...}

***** an optional element, written [*element*].

2. **Representation**

>Expression($E_1$,$E_2$,…)<
is a generic term the formal arguments of which are denoted$E_1$,$E_2$,…; this representation is used to define the general properties of a term in the SIGNAL language in the following headings.

3. **Definition in SIGNAL**

>Term($E_1$,$E_2$,…)<
is defined through a term of the SIGNAL language included in this paragraph.

4. **Profile**

This paragraph describes the inputs and outputs of the expression; this is obtained with the notation ?(E) to refer to the list of inputs of E, !(E) to refer to the list of outputs of E and !{$a_1$,…,$a_n$} to refer the explicit set of ports $a_1$,…,$a_n$. The following operations are used: $A \cup B$, $A \cap B$ and A-B (the set of elements in A that does not exist in B).

5. **Types**

This paragraph describes the properties of the argument types with equations in the value domains of signals. The notation $\tau(E)$ is used to refer the type of the expression *E*

- Equation

6. **Clocks**

This paragraph describes the properties of argument synchronization (the values of booleans and clocks), informally or with an equation list in the synchronization

space. The notation $\omega(E)$ is used to name the clock of *E* and the notation $\tilde{h}$ is used to name the clock of constant expressions. An equation has the following form:

- $E_1 = E_2$

7. **Semantics**

When the term cannot be redefined in SIGNAL, its semantics is given with an informal presentation.

8. **Properties**

Here is given a list of properties of the construction (such as associativity, distributivity, etc.).

- Property

9. **Examples**

- Examples in the SIGNAL language complete the presentation.

# I-3 Lexical units

The program text in SIGNAL is composed of words of the vocabulary which are built from a set of characters.

## I-3.1 Characters

The set of characters (noted **character**) used in SIGNAL is the set of ASCII-characters which are distinguished into subgroups shown below:

1. The set **charname** is the characters used to build the identifiers; this set is composed of

- the set **letter**, the letters of the alphabet, divided into two groups:
  - the group **uppercase** consisting of the upper case letters of the alphabet.
  - the group **lowercase** consisting of the lower case letters of the alphabet.

- the set **digit** are the digits: 0-9

Upper and lower cases are not distinguished more than concerning key words.The key words must be written in lower case letters.

2. The suffix separator **_**(underscore character)
3. The **separators**: space character, end of line
4. The special characters used in the language terminals.
5. The other editable characters used in the comments.

# I-3.2 Vocabulary

A text in SIGNAL is a sequence of elements from the **Terminal** vocabulary is the longest sequence of characters that can be built according to the rules below.

## I-3.2 A Names

A name allows to refer to a signal, a parameter or a model, in a context formed of a group of declarations. Two occurrences of the same name in two distinct contexts can refer to different objects.

**1. Context free grammar**

A name is composed of a letter followed by a sequence of letters or numbers separated by "**_**", perhaps followed or preceded by separators. All the characters **charname** and the occurrences of "**_**" are significant.

**# Name ::=**
*** letter [ { { charname... } _ ... } ] ***
*** letter _ { { charname... }_ ... } ***

**2. Examples**

- a and A are identical **Names**.
- x_25, The_pass_word_12Xs3 are **Names**.

## I-3.2 B Constants

A denotation of a constant value is one of the units listed below and described in chapter II:

***Cst-boolean**
***Cst-integer**

### I-3.2 C Comments

A comment is a sequence of characters between two occurrences of the character **%**. A comment can appear anywhere between two syntactic units.

# I-4 Coding of the synchronization

The verification of the properties of a SIGNAL program is done in a space of three values for the coding of the control associated with a boolean signal which may be absent, or if present, have the value *true* or *false*. The body of integers modulo 3 ($\mathcal{F}_3$) uses this coding:
absent -> 0
*false* -> -1
*true* -> 1

# Chapter II. Value domains of signals

A signal is a sequence of values associated with a clock. All these values have the same type, we will take the liberty of considering the type of the sequence as being the type of each value. The object of this chapter is to present the notations for representing these types and the treatment that is attached on them. An element of the set of types in SIGNAL is noted *type*.
Suppose that *E* is a term in SIGNAL; the type associated with the term *E* is denoted $\tau(E)$ and, when *E* is a constant expression, the value of this expression is denoted $\varphi(E)$. The type and the value are calculated from the context where *E* appears.

## II-1 Scalar types

The scalar types contain the synchronization types, the integer types, the real types and the complex types; the integer, real and complex types compose the set of numerical types according to the syntax below:

1. **Context free grammar**

**# Scalar-type ::=**
*** Synchronization-type ***
*** Numerical-type ***

**# Numerical-type ::=**
*** Integer-type ***
*** Real-type ***
*** Complex-type ***

### II-1.1 Synchronization types

The synchronization types are used to build the signal clocks.They are the type *event* (or pure signal) and the type *logical*.

**Denotation of types**

**1. Context free grammar**

**# Type-synchronization::=**
*** event ***
*** logical ***

**2. Types**

- τ(**event**)=*event*
- τ(**logical**)=*logical*

**Denotation of values**

*****A signal of type *event* gets its values from a singleton: there are no associated constants and a parameter cannot have this type.

***** The constants of the type *logical* are logic values denoted according to the syntax of a **Boolean-Cst**

**1. Context free grammar**

**# Boolean-Cst:=**
*** true ***
*** false ***

## II-1.2 Integer types

**Denotation of types**

**1. Context free grammar**

**# Integer-type::=**
*** integer ***

**2. Types**

- τ (**integer**)=*integer*

**Denotation of values**

The positive values of the *integer* type are denoted according to the syntax of **Integer-cst**. A negative value has no direct representation: it is obtained by using the operator **-** applied to a positive value.

**1. Context free grammar**

An **Integer-Cst** is composed of a sequence of digits.

**# Integer-Cst ::=**
*** {digit ... } ***

**2. Types**

- The type of an **Integer-Cst** is the type *integer*.

**3. Semantics**

An **Integer-cst** denotes a positive integer value or the null value represented in the decimal system.

## II-1.3 Real types

The real values can be represented in simple precision (type *real*) or in double precision (type *long real*) . For the Fortran generator, the types *real* and *long real* are identical.

**Denotation of types**

**1. Context free grammar**

**# Real-type::=**
*** real ***
*** dpreal ***

**2. Types**

- $\tau$(**real**)= *real*
- $\tau$(**dpreal**)= *long real*

A value of type real is denoted according to the syntax of a **REAL-CST** identical to the representation of Fortran 77 except for the following:
The exponent mark (**e** or **d**) must always be followed by a separator (space or end of line). A **REAL-CST** denotes the approximate value of a real number.

1. **Context free grammar**

There are two groups of reals: the reals with simple precision and the reals with double precision which include the first ones.

# **REAL-CST ::=**
*** REAL-SIMPLE-PRECISION-CST ***
*** REAL-DOUBLE-PRECISION-CST***

# **REAL-SIMPLE-PRECISION-CST::=**
*** Integer-cst SIMPLE-PRECISION-EXPONENT ***
*** Fraction-part [ SIMPLE-PRECISION-EXPONENT ] ***
A **REAL-SIMPLE-PRECISION-CST** can have an exponent.

# **REAL-DOUBLE-PRECISION-CST ::=**
***Integer-cst DOUBLE-PRECISION-EXPONENT***
*** Fraction-part [ DOUBLE-PRECISION-EXPONENT] ***
A **REAL-DOUBLE-PRECISION-CST** should have an exponent**.**

# **Fraction-part::=**
*** . Integer-cst ***
*** Integer-cst . [ Integer-cst ] ***

# **SIMPLE-PRECISION-EXPONENT ::=**
*** e RELATIVE-CST ***

# **DOUBLE-PRECISION-EXPONENT ::=**
*** d RELATIVE-CST ***

# **RELATIVE-CST ::=**
***Integer-cst ***
*** + Integer-cst ***
*** - Integer-cst ***

**2.Representation**

> $E_1.E_2\mathrm{e}\ E_3$ (simple precision ) or $E_1.E_2\mathrm{d}\ E_3$ ( double precision ) <

**3. Types**

- The type of a **REAL-SIMPLE-PRECISION-CST** is *real*

- The type of a **REAL-DOUBLE-PRECISION-CST** is *long real.*

**4. Semantics**

The value $\varphi(E_i)$ is 0 when $E_i$ is absent. If $E_2$ has *n* digits, the value of the constant is the approximate value of $(\varphi(E_1) + \varphi(E_2) \times 10^{-n}) \times 10^{\varphi(E_3)}$

**5. Examples**

- The notations included in the tables below are simple precision representations respectively equivalent to the values 1 and 1/100.

**Table 1:**

| | 1e 0 | 1e +0 | 10e -1 |
|---|---|---|---|
| 1. | 1.e 0 | 1.e +0 | 10.e -1 |
| 1.0 | 0.1e 1 | 0.1e +1 | 10.0e -1 |
| | .1e 1 | .1e +1 | |

**Table 2:**

| | | | 1e -2 |
|---|---|---|---|
| | | | 1.e -2 |
| 0.01 | 0.001e 1 | 0.001e +1 | 1.0e -2 |
| .01 | .001e 1 | .001e +1 | .1e -1 |

## II-1.4 Complex type

**Denotation of types**

**1. Context free grammar**

**# Complex-type-::=**

*** complex ***

**2. Types**

- τ(**complex**)= *complex*

**Denotation of values**

A value of type *complex* is denoted by a pair of synchronous reals where the first element is the real part and the second one the imaginary part.

**1. Context free grammar**

**# COMPLEX-CST::=**
*** ( REAL-SIGNED-CST , REAL-SIGNED-CST ) ***

**# REAL-SIGNED-CST::=**
*** + REAL-SIMPLE-PRECISION-CST***
*** - REAL-SIMPLE-PRECISION-CST ***

**2. Examples**

- (1.0,-1.0)

# II-2 Array types

An array is a structure which includes synchronous elements which have the same type. The description of this structure and the access of its elements is obtained by using constant expressions which have the syntax of expressions on signals (**S-EXPR**).

**Denotation of types**

An array type is defined according to the syntax of the first rule shown below; the element type of an array is described by the variable **Element-type**.

**1. Context free grammar**

**# ARRAY-TYPE::=**
*** [ { S-EXPR , ... } ] Element-type ***

**# Element-type::=**

*** logical ***
*** Numerical-type ***

**2. Representation**

> $[n_1,\ldots,n_m]\,\nu$ <

**3. Types**

- The values of $n_1 \quad (\varphi(n_1)),\ldots,n_m \quad (\varphi(n_m))$ are positive integers.
- The type of the array is:

$\tau([n_1,\ldots,n_m]\,\nu) \;=\; [1\ldots\varphi(n_1)] \times \ldots \times [1\ldots\varphi(n_m)] \rightarrow \tau(\nu)$ .

**4. Clocks**

The integers $n_i$ are defined by constant expressions.

- $\omega(n_i) \;=\; \tilde{h}$

**5. Examples**

- "[10,10] integer T" declares T as a two dimensions integer array; each dimension starts at 1.

**Denotation of values**

An array constant is an enumeration of constant expressions with the same type (cf. Expressions on arrays).

# II-3 Structure of the set of types

A partial order is defined on the types which yields a "natural" immersion of a smaller set into a larger one.

## II-3.1 The set of types

The set of types is composed by types of which the expressions in SIGNAL, described in the recapitulation below, are derived from the variable **SIGNAL-TYPE**:

SIGNAL-TYPE
    **ARRAY-TYPE**
        **[{ S-EXPR , ... }] Element-type**

Generic form of the array types: $[1\ldots n_1] \times \ldots \times [1\ldots n_m] \rightarrow \mu$

**Element-type**

**logical** denotes the *logical* type

**Numerical-type**

**integer** denotes the *integer* type

**Real-type**

**real** denotes the *real* type

**dpreal** denotes the *long real* type

**complex** denotes the *complex* type

**event** denotes the *event* type

**1. Context free grammar**

**# SIGNAL-TYPE ::=**
***ARRAY-TYPE ***
***Element-type ***
* **event** *

## II-3.2 Order on the types

**Order on the synchronization types**
The *event* type is lower than the *logical* type .

**Order on the numerical types**
The *integer* type is lower than the *real* type. The *real* type is lower than the *complex* type .

**Order on the arrays**
The order of the numerical types is extended to arrays:

* $[1\ldots m_1] \times \ldots \times [1\ldots m_k] \rightarrow \mu \subseteq [1\ldots n_1] \times \ldots \times [1\ldots n_l] \rightarrow \nu$ if and only if

* $k=l$
* $\forall i, (0 < i \leq k) \Rightarrow n_i = m_i$
*and $\mu \subseteq \nu$

Note: One uses the notation $\mu \cup \nu$ to refer to the upper bound of two compatible types $\mu$ and $\nu$.

## II-3.3 Conversions

In a binary expression, the argument of type $\mu$ can be implicitly converted into the superior type $\nu$ if this one is the type of the second argument.

# II-4 Declaration of signal identifiers

A sequence of values has a type (the type of its elements); this type is associated with an identifier in a declaration. An identifier can name a formal signal (parameter) or a local signal according to the same syntax of declaration. Signals must be initialized when they are defined by a dynamic expression (delay or sliding window) or by a memorization expression; the initialization is done at the declaration behind the key word *init.*
The declarations of signal identifiers respect the following syntax:

**1. Context free grammar**

**# S-DECLARATION ::=**
***SIGNAL-TYPE { SIGNAL-name ... } ***
***Inherited-type { SIGNAL-name ... } ***
**# SIGNAL ::=**
******Signal***-name [ init S-EXPR ] ***
**# Inherited-type ::=**
*** ***

**2.Representation**

$> \ \mu\ ID_1,\ldots,ID_i \text{init}\, E_i,\ldots,ID_n \ <$

**3. Types**

- The declared names have to be distinct from each other. The same type $\tau(ID_1)$ is attributed to the identifiers $ID_1,\ldots,ID_n$ in the context of the declaration.

  * if $\mu$ is a SIGNAL-TYPE then the type attributed to the identifiers is the type $\tau(\mu)$ ;

  * if $\mu$ is an INHERITED-TYPE (no type in the declaration), the type attributed to the identifiers is the result of the inherited type calculus, if it exists. If an external signal has no type in its declaration, it has the type *real* .

- The type $\tau(E_i)$ of the expression $E_i$ is the type $\tau(ID_i)$ eventually increased of one dimension to the left (cf. Dynamic expressions).

**4. Clocks**

The expressions to initialize $E_i$ are constant expressions.

- $\omega(E_i) = \tilde{h}$

**5. Semantics**

The value $E_i$ is an initial value for the signal $ID_i$; it is necessary if $ID_i$ is a signal obtained when applying a delay, while taking a window, or while memorizing the current value of another signal. This value has no sense for any type of parameters nor for pure signals (*event*).

**6. Examples**

- real X, Y, ZX init 3.14, T
- [n]real T init [{i to n}:i]

**7. Anomaly**

A vector parameter cannot initialize a delay.

# Chapter III. Expressions of signals

The values associated with the signals are determined by signal equations; these equations are built by composition of sub-equations deriving from elementary equations. In this chapter, we present the expressions which allow to define a signal (**S-EXPR**); this presentation is preceded by an introduction to the expressions of composition (**P-EXPR**).

## III-1 Equations to define signals

### III-1.1 Elementary equations

A signal definition defines a signal or a set of signals according to the following syntax:

**1. Context free grammar**

**# DEFINITION-OF-SIGNALS ::=**
******Signal*-**name :=** **S-EXPR ***
*** [ {** *Signal*-**name , ... } ]** **:=** **S-EXPR ***

**2. Representation**

> X := E <

**3. Representation**

> { $X_1,\ldots,X_n$ } := *E* < with < $\upsilon_1,\ldots,\upsilon_n$ > the signals defined by *E*

**4. Clocks**

An identifier and the signal which defines it are synchronous

- $\omega(X) = \omega(E)$
- $\omega(X_i) = \omega(\upsilon_i)$

**5. Types**

- $\tau(X) = \tau(E)$
- $\tau(X_i) = \tau(\upsilon_i)$

**6. Profile**

A signal definition equation has inputs and outputs defined by the following rules:

- The identifiers of the defined signals are the outputs of the equation:
  - ! ($X := E$) = { $X$ }
  - ! ({$X_1$,…,$X_n$ } := $E$) = { $X_1$,…,$X_n$ }
- The inputs of the equation are the inputs of $E$: they are signal identifiers occurring at least once in $E$, directly or as visible inputs of the model calls:
  - ?($X := E$)=?(E)

**7. Semantics**

- the signal $X$ is equal to the signal which is the result of the evaluation of $E$,
- every signal $X_i$ is respectively equal to the signal $\nu_i$

**8. Examples**

- if x,y, z are signals:

x := y + z defines the signal designated by x, equal to the sum of the signals respectively named by y and z; this expression has y and z as inputs, and x as output;

- if x, y, a are signals, and Q is a model owning two outputs and one input:

{x,y} := Q{a-5} defines the signals named by x and y respectively equal to the first and second output of model Q; this expression has a as an input and x and y as outputs;

- if x, y, z, a are signals, P a model which has three outputs and four inputs and Q a model which has two outputs and one input:

{x, y, z} := P { a, Q{a-5}, a+5 } defines the signals named x, y and z respectively equal to the first, the second and the third outputs of model P; this expression has x, y and z as outputs and a as input.

## III-1.2 Composition of signal definitions

The equations of signal definition can be composed by the operator |. An expression $E_1 | E_2$, called an expression on equations, defines the signals (has as outputs the signals) defined in every sub-expression and has as inputs the signal inputs of each of its sub-expressions that are not outputs of the others; since a signal cannot have a double definition, a signal identifier cannot be an output of two sub-expressions. The value of an input of a sub-expression defined in the other is the value associated by this definition.
An equation expression may have brackets to the left (| and to the right |).
A set of outputs may be invisible by using the operator /. An expression $E/a_1,\ldots,a_n$ has as outputs the outputs of *E* that do not exist in the list $a_1,\ldots,a_n$ and as inputs the inputs of *E*.

# III-2 Elementary expressions

**1. Context free grammar**

**# ELEMENTARY-S-EXPR ::=**
*** CONSTANT ***
*** *Signal*-name ***
*** INDEX ***
*** PRODUCTION ***

**# CONSTANT ::=**
*** Boolean-cst ***
*** Integer-cst ***
*** REAL-CST ***
*** CST-COMPLEX ***

## III-2.1 Constant expressions

A constant expression is a CONSTANT, an occurrence of a parameter identifier or one of the following expressions having recursively constant expressions as arguments:

* an **ARRAY-S-EXPR**
* a **BOOLEAN-S-EXPR**
* an **ARITHMETIC-S-EXPR**

The temporal expressions (**TEMPORAL-S-EXPR**) and the function calls (**PRODUCTION**) cannot appear in a constant expression. A constant is a denotation of a value of a scalar type.

**1. Context free grammar**

**# CONSTANT ::=**
*** Boolean-cst ***
*** Integer-cst ***
*** REAL-CST ***
*** COMPLEX-CST ***

**2. Profile**

A constant, and consequently, a constant expression, have neither a named input nor a named output.

A constant expression and its arguments have all of them the same clock $\tilde{h}$.
The type of a constant expression is evaluated in accordance with the type of the **S-EXPR** having the same syntax.

## III-2.2 Occurrence of signal identifiers

An occurrence of a signal identifier has as value the signal that defines this identifier and as type the type of its most internal declaration; its associated profile contains as input this unique identifier and does not contain any named outputs.

## III-2.3 Index

**1. Context free grammar**

**# INDEX::=**
***** *Signal*-**name [ {S-EXPR , ... } ] ***

**2. Profile**

$$? (T[E_1,\ldots,E_m]) = \{T\} \bigcup \bigcup_{i=1}^{m} (? E_i)$$

**3. Clocks**

The signals appearing in an index are synchronous.

- $\omega(T[E_1,\ldots,E_m])=\omega(T)$
- $\omega(T[E_1,\ldots,E_m])=\omega(E_i)$

**4. Types**

- if T is an array of type $[1\ldots n_1]\times\ldots\times[1\ldots n_m]\rightarrow\tau(\nu)$, and if the values resulting from the evaluation of the expressions $E_i$ are respectivelyincluded between 1 and $n_i$ then $T[E_1,\ldots,E_m]$ is a scalar of type $\tau(\nu)$
- if T is an array of type $[1\ldots n_1]\times\ldots\times[1\ldots n_m]\times[1\ldots n_{m+1}]\times\ldots\times[1\ldots n_p]\rightarrow\tau(\nu)$, and if the values resulting from the evaluation of the expressions $E_i$ are respectively included between 1 and $n_i$ then T[$E_1$,…,$E_n$] is an array of type $[1\ldots n_{m+1}]\times\ldots\times[1\ldots n_p]\rightarrow\tau(\nu)$
- in the other cases, the index is incorrect.

**5. Semantics**

The value of an index is defined recursively by

- the value of $T[E_1,\ldots,E_m]$ is the value $TT[E_2,\ldots,E_m]$ where *TT* is the value of $T[E_1]$.
- the value of T[$E_1$] is the value of the $k^{th}$ element of *T* when *k* is the integer value produced by the evaluation of $E_1$
- if the number of indexes is correct, but the value of one of these indexes is not included in the corresponding boundaries, then the result is not defined.

# III-2.4 Model invocation

The **PRODUCTION** of a process by a model invocation is realized by the macro-expansion of the model text. The static parameters are parenthezised by **(** and **)** ; these parameters are constant expressions used as initial values of signals or sizes of arrays. The input signals can be associated with a model:

- either “in position” (list of expressions defining them between the characters **{** and **}**),

- or by identity on the signal names (empty list between the characters **{** and **}**).

**1. Context free grammar**

**# PRODUCTION ::=**
*** REFERENCE-MODEL { [ { S-EXPR , ... } ] } ***

**# REFERENCE-MODEL ::=**
*** CALL ***
***** *Model*-**name ***

**# CALL ::=**
***** *Model*-**name ( [ { S-EXPR , ... } ] ) ***

**2. Representation**

> $P\,(V_1,\ldots,V_m)\ \{E_1,\ldots,E_n\}$ <

**3. Types**

- For each formal parameter $P_i$ (name of the parameter identifier in position *i* in the declaration of *P*) and its associated effective value $V_i$, $\tau\,(V_i)\ =\ \tau\,(P_i)$ .
- For each input $SE_i$ (name of the input identifier in position i in the declaration of P) and its corresponding expression $E_i$, $\tau\,(E_i)\ =\ \tau\,(SE_i)$ .
- For each output $SS_i$ and its expected result $F_i$, $\tau\,(SS_i)\ =\ \tau\,(F_i)$ .

**4. Profile**

- $?(\ P\,(V_1,\ldots,V_m)\ \{E_1,\ldots,E_n\}\ ) = \bigcup_{i=1}^{n} ?(E_i)$
- $!(\ P\,(V_1,\ldots,V_m)\ \{E_1,\ldots,E_n\}\ )$ is the set of output names in the *P* declaration.

# III-3 Arithmetical expressions

The arithmetical expressions are synchronous expressions of signals: the input signal and the resulting signal have the same clock. The operators defining this kind of expressions are the standard arithmetic operators extended to sequences of elements .

**1. Context free grammar**

**# ARITHMETIC-S-EXPR::=**
*** S-EXPR + S-EXPR ***
*** S-EXPR - S-EXPR ***
*** S-EXPR * S-EXPR ***
*** S-EXPR / S-EXPR ***
*** S-EXPR ** S-EXPR ***
**** + *S-EXPR* ****
*** - S-EXPR ***

**2. Priorities**

These operators are ordered in increasing priority according to the following:

- binary + and - have the same priority;
- * and / have the same priority;
- **
- unary + and - have the same priority.

**3. Semantics**

The expressions are defined with their usual semantics. When an expression in a division is of type *integer*, the division is integer division.
If the result of an expression is not representable in the type $\mu$ of this expression, its value is a value of type $\mu$ depending on the implementation.

## III-3.1 Addition, substraction, multiplication, division

The binary operators with the same priority are evaluated from left to right.

**1. Representation**

> $E_1$ **Op** $E_2$ <

**2. Types**

- $\tau(E_1)$ and $\tau(E_2)$ are numerical types
- $\tau(E_1 \text{ Op } E_2) = \tau(E_1) \cup \tau(E_2)$

**3. Clocks**

- $\omega(E_1) = \omega(E_2)$
- $\omega(E_1 \text{ Op } E_2) = \omega(E_1)$

## III-3.2 Power

**1. Representation**

> $E_1$ ** $E_2$ <

**2. Types**

- $\tau(E_1)$ is a numerical type
- $\tau(E_2)$ is the *integer* type
- $\tau(E_1 \text{ ** } E_2) = \tau(E_1)$

**3. Clocks**

- $\omega(E_1) = \omega(E_2)$
- $\omega(E_1 \text{ ** } E_2) = \omega(E_1)$

**4. Anomaly**

The generated C code is not good for this operator.

## III-3.3 The unary operators (+ and -)

**1. Representation**

> *op E* <

**2. Types**

- $\tau(E)$ is a numerical type
- $\tau(\text{op } E) = \tau(E)$

**3. Clocks**

- $\omega(\text{op } E) = \omega(E)$

# III-4 Boolean expressions

The boolean expressions are synchronous expressions of signals: the signal arguments, just as the resulting signal, have the same clock. The operators defining this kind of expression are the standard operators of boolean elements extended to the sequences of elements. The boolean expressions (or expressions with boolean result) are either relations or lattice expressions.

## II-4.1 Expressions of booleans

**1. Context free grammar**

**# BOOLEAN-S-EXPR**
*** S-EXPR and S-EXPR ***
*** S-EXPR or S-EXPR ***
*** not S-EXPR ***
*** RELATION ***

**2. Priorities**

These operators are ordered in increasing priority according to the following:

- **or**: logical or
- **and**: logical and
- **not:** logical not
- relation operators

**3. Types**

- $\tau(E_1) \subseteq$ *logical*
- $\tau(E_2) \subseteq$ *logical*
- $\tau(E_1 \text{ or } E_2) \subseteq$ *logical*
- $\tau(E_1 \text{ and } E_2) \subseteq$ *logical*
- $\tau(\text{not } E_1) \subseteq$ *logical*

**4. Clocks**

The clocks of the arguments of the boolean expressions are identical; so, for *op* denoting the operator **or** or **and**:

- $\omega(E_1 \text{ op } E_2) = \omega(E_1)$
- $\omega(E_1 \text{ op } E_2) = \omega(E_2)$
- $\omega(\text{not } E_1) = \omega(E_1)$

**5. Semantics**

The boolean operators have their usual semantics.

**6. Definition in SIGNAL**

The boolean disjunction is not a primitive operator in SIGNAL:
$X := E_1$ or $E_2$ is equal to the process defined below.

```
>
( | X := (when E1) default (when E2) default (not event E1)
| synchro {E1,E2}
| )
<
```

The boolean conjunction is not a primitive operator in SIGNAL:
$X := E_1$ and $E_2$ is equal to the process defined below.

```
>
( | X := (when E1 when E2) default (not event E1)
| synchro {E1,E2}
| )
<
```

## III-4.2 Relations

**1. Context free grammar**

**# RELATION ::=**

*** S-EXPR = S-EXPR ***
*** S-EXPR /= S-EXPR ***
*** S-EXPR > S-EXPR ***
*** S-EXPR >= S-EXPR ***
*** S-EXPR < S-EXPR ***
*** S-EXPR <= S-EXPR ***

**2. Representation**

> $E_1$ **Op** $E_2$ <

**3. Types**

- $E_1$ and $E_2$ have the same scalar type, but different from the *complex* type.

**4. Semantics**

In the order defined on the logical values, *false* is inferior to *true*. With this precision, the relation operators have their usual semantics.

# III-5 Temporal expressions

The temporal expressions are built on possibly different clocks.

**1. Context free grammar**

**# TEMPORAL-S-EXPR ::=**
*** COUNTER ***
*** MERGE ***
*** EXTRACTION ***
*** MEMORIZING ***
*** SIGNAL-CLOCK ***
*** CLOCK-EXTRACTION ***

**2. Priorities**

These operators are ordered in increasing priority according to the following:

- the **COUNTER** operators (#)
- the **MERGE** operator (default)
- the **EXTRACTION** operator (when)
- the **MEMORIZING** operator (cell)

- the **CLOCK-EXTRACTION** operator (unary when) and **SIGNAL-CLOCK** (event) have the same priority.

## III-5.1 Merge

The deterministic merge (with priority) is defined according to the following syntax:

**1. Context free grammar**

**# MERGE ::=**
*** S-EXPR default S-EXPR ***

**2. Representation**

> $E_1$ default $E_2$ <

**3. Types**

- $\tau(E_1 \text{ default } E_2) = \tau(E_1) \cup \tau(E_2)$

**4. Clocks**

- $h = \omega(E_1 \text{ default } E_2)$
- $h = \omega(E_1) \cup h$
- if $\omega(E_2)$ is different from $\tilde{h}$, $h = \omega(E_1) \cup \omega(E_2)$

**5. Properties**

- ( $E_1$ default $E_2$ ) default $E_3$ = $E_1$ default ( $E_2$ default $E_3$ )

**6. Examples**

- the values taken by X default Y are described below as the values corresponding to X and Y as inputs:

**Tableau 3 :**

| | | | | | | |
|---|---|---|---|---|---|---|
| X = | $\perp$ | 1 | 3 | $\perp$ | 7 | ... |
| Y= | 2 | $\perp$ | 6 | 8 | 1 | ... |
| X default Y = | 2 | 1 | 3 | 8 | 7 | ... |

## III-5.2 Extraction

The values of a signal can be produced by extracting the values of an other signal when the values of a boolean signal are *true*.

**1. Context free grammar**

**# EXTRACTION ::=**
*** S-EXPR when S-EXPR ***

**2. Representation**

> *E* when *B* <

**3. Types**

- $\tau(B) \subseteq$ *logical*
- $\tau(E \text{ when } B) = \tau(E)$

**4. Clocks**

- $h = \omega(E \text{ when } B)$
- if $\omega(E) = \tilde{h}$ then $h = \omega(B) \times (-1 - B)$
- if $\omega(E_2)$ is different from $\tilde{h}$, $h = \omega(E) \times \omega(B) \times (-1 - B)$

**5. Properties**

- (*E* when *B*) when *C* = *E* when (*B* when *C*)
- *E* when (*B* when *B*) = *E* when *B*
- ( $E_1$ default $E_2$ ) when *B* = ( $E_1$ when *B*) default ( $E_2$ when *B*)

**6. Examples**

- the values taken by X when C are described below as the values corresponding to X and C as inputs:

**Tableau 4 :**

| X = | 1 | 3 | ⊥ | 5 | ⊥ | 7 | 0 | ... |
|---|---|---|---|---|---|---|---|---|
| C = | T | ⊥ | T | F | F | T | T | ... |
| X when C = | 1 | ⊥ | ⊥ | ⊥ | ⊥ | 7 | 0 | ... |

## III-5.3 Signal clock

The clock of a signal is obtained by applying the *event* operator to this signal which may be of any type.

**1. Context free grammar**

**# SIGNAL-CLOCK::=**
*** <u>event</u> S-EXPR ***

**2. Representation**

> event *B* <

**3. Types**

- $\tau(\text{event } B) = event$

**4. Examples**

- the values taken by event X are described below as the values corresponding to X as input:

**Tableau 5 :**

| X = | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|
| event X = | T | T | T | T | ... |

## III-5.4 Clock extraction

The extraction of true values of a boolean condition is obtained by application of the unary operator *when*:

**1. Context free grammar**

**# CLOCK-EXTRACTION::=**
*** <u>when</u> S-EXPR ***

**2. Representation**

> when *B* <

**3. Types**

- $\tau(B) \subseteq$ *logical*
- $\tau(\text{ when } B) = event$

**4. Definition in SIGNAL**

> (event (*B*)) when *B* <

**5. Clocks**

- $\omega(\text{when } B) = \omega(B) \times (-1 - B)$

**6. Examples**

- the values taken by when C are described below as the values corresponding to C as input:

**Tableau 6 :**

| C = | T | T | F | F | T | ... |
|---|---|---|---|---|---|---|
| when C = | T | T | ⊥ | ⊥ | T | ... |

## III-5.5 Equations of clocks

A **CLOCK-EQUATION** contributes to the construction of the clock equations system of the program. It is the tool for programming by constraints.

**1. Context free grammar**

**# CLOCK-EQUATION ::=**
*** synchro { { S-EXPR , ... } } ***

**2. Representation**

> synchro $\{E_1,E_2\}$ <

**3. Profile**

A clock equation is a process without any output with
?(synchro $\{E_1,E_2\}$)=? $(E_1) \cup ?\ (E_2)$

**4. Types**

- The arguments $E_i$ are of any types.

**5. Definition in SIGNAL**

synchro $\{E_1,E_2\}$
constrains to equalize the clocks of the signal expressions $E_1$ and $E_2$; this expression is equal to the process without output defined below.
>
( | $X$ := (event $E_1$) = ( event $E_2$ )
| ) / $X$
<

# III-5.6 Counter

The counter expressions allow one to number the occurrences of a clock.

## III-5.6 A Complete counter

The complete counter of a signal is defined according to the following syntax:

**1. Context free grammar**

**# COUNTER ::=**
*** # S-EXPR ***

**2. Representation**

> # $C_1$ <

**3. Types**

- $\tau(C_1) \subseteq$ *logical*
- $\tau(\#\ C_1) = integer$

**4. Clocks**

- $\omega(\#\ C_1) = \omega(\text{when}\ C_1)$

**5. Semantics**

The signal *N*, defined at the clock *H* (obtained by extracting the *true* values of a *logical* signal $C_1$), by the expression *N* := # $C_1$, enumerates the strictly positive integers; *N* counts in this way the occurrences of *true* values of the signal $C_1$.

**6. Definition in SIGNAL**

*N* := # $C_1$ is equal to the process defined below.

```
>
( | N := ZN+1
 | ZN := N $ 1
 | synchro {N, when C1}
 | ) / ZN
 <
```

in which *ZN* has the initial value 0.

**7. Examples**

- The values taken by # $C_1$ are described below as the corresponding values of $C_1$:

**Tableau 7 :**

| $C_1$ = | F | T | T | F | T | ... |
|---|---|---|---|---|---|---|
| #$C_1$ = | ⊥ | 1 | 2 | ⊥ | 3 | ... |

## III-5.6 B Relative counter

The relative counter is defined under two forms, according to the following syntax:

**1. Context free grammar**

**# COUNTER ::=**
*** # S-EXPR after S-EXPR ***
*** # S-EXPR from S-EXPR ***

**2. Representation**

> # $C_1$ after $C_2$
# $C_1$ from $C_2$ <

**3. Types**

- $\tau(C_1) \subseteq$ *logical*
- $\tau(C_2) \subseteq$ *logical*
- $\tau(\#\ C_1 \text{ mode } C_2) = integer$

**4. Clocks**

- $\omega(\#\ C_1 \text{ mode } C_2) = \omega((\text{when } C_1) \text{ default } (\text{when } C_2))$

**5. Semantics**

The signal *N* defined, at the clock *H* obtained by extracting the *true* values of a *logical* signal $C_1$, by the expression $N := \#\ C_1$ *mode* $C_2$, counts in this way the number of occurrences of *true* values of the signal $C_1$ ($o_1$) since the last occurrence of the *true* value of the signal $C_2$ ($o_2$); when the mode is “from”, the occurences $o_1$ simultaneous to the $o_2$ occurrences are counted; when the mode is “after”, the occurrences $o_1$ simultaneous to the $o_2$ occurrences are not counted.

**6. Definition in SIGNAL**

$N := \#\ C_1$ from $C_2$
is equal to the process defined below

>
( | $N_{IN}$ := ( (1 when $C_1$) default 0 ) when $C_2$
| $N$ = $N_{IN}$ default ( ($ZN+1$) when $C_1$)
| $ZN := N \$ 1$
| synchro{$N$,(when $C_1$) default (when $C_2$)}
| ) / $ZN$, $N_{IN}$
<
in which $ZN$ has the initial value 0.

$N := \# C_1$ after $C_2$
is equal to the process defined below

>
( | $N_{IN}$ := 0 when $C_2$
| $N$ = $N_{IN}$ default ( ($ZN+1$) when $C_1$)
| $ZN := N \$ 1$
| synchro{$N$,(when $C_1$) default (when $C_2$)}
| ) / $ZN$, $N_{IN}$
<
in which $ZN$ has the initial value 0.

**7. Examples**

- the values taken by $\# C_1$ *from* $C_2$ and $\# C_1$ *after* $C_2$ are described below as the values corresponding to $C_1$ and $C_2$ as inputs:

**Tableau 8 :**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $C_1$ = | F | F | T | T | T | T | F | T | ⊥ | F |
| $C_2$ = | T | ⊥ | F | ⊥ | T | ⊥ | F | F | ⊥ | T |
| $\#C_1 \text{from} C_2$ | 0 | ⊥ | 1 | 2 | 1 | 2 | ⊥ | 3 | ⊥ | 0 |
| $\#C_1 \text{after} C_2$ | 0 | ⊥ | 1 | 2 | 0 | 1 | ⊥ | 2 | ⊥ | 0 |

The memorization of a signal at the clock defined by the *true* occurrences of a boolean condition respects the following syntax:

**1. Context free grammar**

The two arguments have a priority at the most equal to the priority of the unary extraction operator.

**# MEMORIZATION ::=**
*** S-EXPR cell S-EXPR ***

**2. Representation**

> *E* cell *B* <

**3.Types**

- $\tau(B) \subseteq$ *logical*
- $\tau(E \text{ cell } B) = \tau(E)$

**4. Definition in SIGNAL**

*X* := *E* cell *B*
The value of *X* is either the present value of *E* when *E* is present, or the last received value of *E* when *B* is present and *true*. The *X* signal must be initialized (cf. Declaration of signal identifiers). It is equivalent to the process defined below.
>
( | *X* := *E* default (*X* $1)
| synchro { *X*,*E* default ( when *B*)}
| )
<

**5. Clocks**

- $\omega(\text{ E cell B }) = \omega(E) + (1 - \omega(E)) \times (-B - \omega(B))$

**6. Examples**

- the values taken by Y := X cell C are described below as the values corresponding to X and C, under the hypothesis that Y is initialized with 0:

**Tableau 9 :**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| X = | ⊥ | 1 | 3 | ⊥ | ⊥ | ⊥ | 5 | ⊥ | 7 | ... |
| C = | T | ⊥ | T | T | F | T | F | T | ⊥ | ... |
| Y = | 0 | 1 | 3 | 3 | ⊥ | 3 | 5 | 5 | 7 | ... |

# III-6 Dynamic expressions

The dynamic expressions allow temporal manipulations of signal values.

**1. Context free grammar**

**# DYNAMIC-S-EXPR ::=**
*** DELAY ***
*** WINDOW ***

## III-6.1 Delay

A delay expression is defined according to the following syntax:

**1. Context free grammar**

**# DELAY ::=**
***** *Signal*-**name $ S-EXPR ***

**2. Representation**

> $X \$ N_1$ <

**3. Types**

- $N_1$ is a strictly positive integer
- $\tau(X \$ N_1) = \tau(X)$

**4. Clocks**

- $\omega(N_1) = \tilde{h}$

$\omega(X \$ N_1) = \omega(X)$

**5. Semantics**

The initialization of the signal *ZX*, defined by $ZX := X \$ N_1$ must be:

- if $N_1 = 1$, a constant with the same type as *X*,
- if $N_1 > 1$

- if $X$ is a scalar, an array of dimension $[N_1]$ the elements of which have the type $\tau(X)$
- if $X$ is an array of dimension $[I_1,\ldots,I_2]$, an array of dimension $[N_1,I_1,\ldots,I_2]$ the elements of which have the type of the elements of $X$.

The value of the signal *ZX* is at each instant *t* the value of the delayed signal $X$ at the instant $t - N_1$. If the delay appears as a sub-expression of the definition expression of a signal, the initial value is undefined.

**6. Examples**

- the values taken by X $ 1, with the initial value 0, are described below with the values corresponding to X as input:

**Tableau 10 :**

| X = | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|
| X $ 1 = | 0 | 1 | 2 | 3 | 4 |

## III-6.2 Sliding Window

An expression of a sliding window on a signal is defined by the following syntax:

**1. Context free grammar**

**# WINDOW::=**
***** *Signal*-**name [ $ S-EXPR ] window S-EXPR ***

**2. Representation**

> $X$ \$ $N_1$ window $N_2$ <

**3. Types**

- $N_1$ and $N_2$ are two strictly positive integers; if the delay is absent, $N_1 = 0$
- if $\tau(X) = \mu$ is scalar, then the expression is a vector owning $N_2$ elements
- if $X$ is an array with type $[I_1,\ldots,I_2]\mu$, the type of the window is $[N_2,I_1,\ldots,I_2]\mu$.

**4. Clocks**

- $\omega(N_1) = \tilde{h}$
- $\omega(N_2) = \tilde{h}$
- $\omega(X \$ N_1 \text{ window } N_2) = \omega(X)$

**5. Semantics**

In the declaration of the signal *ZX* defined by $ZX := X \$ N_1$ window $N_2$,

- if $N_1 + N_2 = 1$ the signal *ZX* is not initialized
- if $N_1 + N_2 > 1$ the signal *ZX* must be initialized
  - if *X* is scalar, by a vector of size $N_1 + N_2 - 1$;
  - if *X* is an array of dimension $[I_1,\ldots,I_2]$, by an array of dimension $[N_1 + N_2 - 1, I_1,\ldots,I_2]$
- BUG: the initialization of *ZX* is only obtained by an explicit numbering, even if only one value is necessary for the initialization (for instance [{ to 1 ] : v0] or [{ 1 ] : v0]).

If the window appears as a sub-expression of the definition expression of a signal, its initial value is undefined.
A window is a vector *ZX* each element *ZX*[i] of which is, at the instant *t*, the value of *X* at the instant $[t - N_1 - N_2 + i]$.

**6.Examples**

- the values taken by X $ 1 window 2, with an initialization [{to 2}:0], are described below with the values corresponding to X as input:

**Tableau 11 :**

| X = | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|
| X $ 1 window 2= | [0,0] | [0,1] | [1,2] | [2,3] | ... |

# III-7 Array expressions

The manipulation of arrays is possible by the extension of operations on scalars, the concatenation and the numbering of elements.

**1. Context free grammar**

**# ARRAY-S-EXPR ::=**
*** CONCATENATION ***
*** NUMBERING ***

## III-7.1 Concatenation

**1. Context free grammar**

**# CONCATENATION ::=**
*** {** *Signal*-**name [||] ... } ***

**2. Semantics**

**The CONCATENATION of two arrays**

- $T_1$ of type $[1 \ldots m_1] \times [1 \ldots n_2] \times \ldots \times [1 \ldots n_m] \rightarrow \mu$
- $T_2$ of type $[1 \ldots m_2] \times [1 \ldots n_2] \times \ldots \times [1 \ldots n_m] \rightarrow \mu$

builds the array *T*, of type $[1 \ldots (m_1 + m_2)] \times [1 \ldots n_2] \times \ldots \times [1 \ldots n\ ] \rightarrow \mu$ verifying:

- if $i_1$ is lower or equal than $m_1$, $T[i_1,i_2,\ldots,i_m] = T_1[i_1,i_2,\ldots,i_m]$
- if $i_1$ is strictly higher than $m_1$, $T[i_1,i_2,\ldots,i_m] = T_1[(i_1 - m_1),i_2,\ldots,i_m]$

## III-7.2 Iteration expression

**1. Context free grammar**

**# NUMBERING ::=**
*** [ { ITERATION , ... } ] ***

**# ITERATION :=**
*** SIMPLE-ELEMENT ***
*** MULTIPLE-ELEMENT ***

**# SIMPLE-ELEMENT ::=**

*** [ { S-EXPR , ... } ] : S-EXPR ***

**# MULTIPLE-ELEMENT ::=**
*** { { ITERATOR , ... } } : S-EXPR ***
*** { { ITERATOR , ... } } : SIMPLE-ELEMENT ***

**# ITERATOR ::=**
*** [** *indicator*-**name ] [ BEGIN ] [ END ] [ STEP ] ***

**# BEGIN ::=**
*** in S-EXPR ***

**# END ::=**
*** to S-EXPR ***

**# STEP ::=**
*** step S-EXPR ***

**2. Semantics**

A **NUMBERING** defines an array *T* of type $[1\ldots n_1] \times \ldots \times [1\ldots n_m] \rightarrow \mu$ by a sequence of actions executed sequentially.
Each action is either a definition of an isolated element (**SIMPLE-ELEMENT**), or an **ITERATION** allowing to define a set of values.

- A **SIMPLE-ELEMENT** is defined by the couple of the list $[i_1,\ldots,i_m]$ of its indexes (in the array) and of the value $E_{[I_1,\ldots,I_m]}$ in its indexes.
- An iteration is defined by the couple of a list of iterators and the value expressions for each reached point.
- The iterators constitute a sequence of visible declarations in the rest of the definition of the **MULTIPLE-ELEMENT** where they are contained; they define a set of loops embedded from left to right; the index of each loop take successively the value $\varphi$(**BEGIN**) increased by the value $\varphi$(**STEP**), ... until the last value included between $\varphi$(**END**) and $\varphi$(**BEGIN**)
- If a term is omitted, it is implicitly equal to 1;
- When the definition expression of a value on a point is an **S-EXPR**, it is equivalent to a **SIMPLE-ELEMENT** the list of indexes of which is composed by the sequence of identifiers (created if necessary) of iterators.

**3. Examples**

- T := [ {to 10}: 0 ] defines a zero vector
- T := [ { to 10 , to 10}: 0 , { i in 1 to10 , j in i to 10}: (i+j) ] defines an upper triangular matrix

## III-7.3 Extensions of scalar expressions

The expressions on scalars, except the relations, are extended term by term to the arrays.

# III-8 Expressions on signals

## III-8.1 Scalar expressions

The arithmetic (**ARITHMETIC-S-EXPR**), boolean (**BOOLEAN-S-EXPR**), temporal (**TEMPORAL-S-EXPR**) and dynamic (**DYNAMIC-S-EXPR**) expressions compose the scalar expressions according to the grammar below:

**1. Context free grammar**

**# SCALAR-S-EXPR ::=**
*** ARITHMETIC-S-EXPR ***
*** BOOLEAN-S-EXPR ***
*** TEMPORAL-S-EXPR ***
*** DYNAMIC-S-EXPR ***

**2. Profile**

These expressions do not produce named output; they have as inputs respectively:

- ?(*op E)* = ?(*E*) for an unary operator *op*
- $?(E_1\ op\ E_2) = ?(E_1) \cup ?(E_2)$ for a binary operator *op*

A **SCALAR-S-EXPR** is evaluated according to the increasing priorities defined below:

**1.DYNAMIC-S-EXPR**
**2.TEMPORAL-S-EXPR**
**3.BOOLEAN-S-EXPR**
**4.ARITHMETIC-S-EXPR**

The operator priorities of each syntactical structure are described in the corresponding sections. An expression which has a lower priority than the expression to which it is argument, must be parenthesized. The parenthesizing is permitted but not necessary in other cases. For the same priority, unless indicated, the evaluation order of expressions with an arity higher than 1 is from left to right. The only expressions that are allowed without parenthesises are built on the associative operators (or pseudo associative as the when operator).
The extensions of the scalar expression array respect the rules announced below.

## III-8.2 Embedding of expressions on signals

The expressions on signals are organized in elementary expressions, scalar structure expressions and array structure expressions.

**1. Context free grammar**

**# S-EXPR ::=**
*** ( S-EXPR ) ***
*** ELEMENTARY-S-EXPR ***
*** SCALAR-S-EXPR ***
*** ARRAY-S-EXPR ***

Under condition of being correctly typed, an **ELEMENTARY-S-EXPR** can be used as an argument in the whole expression of signals. A **SCALAR-S-EXPR** cannot have an **ARRAY-S-EXPR** as argument and vice versa.

# Chapter IV. Expressions on processes

The expressions on processes allow to compose systems of equations on signals according to the grammar below:

**1. Context free grammar**

**# P-EXPR ::=**
*** ( P-EXPR ) ***
*** ELEMENTARY-PROCESSES ***
*** COMPOSITION ***
*** PROFILE ***
*** PROCESS-ARRAY ***

## IV-1 Elementary process

An elementary process is a definition of signals, a process instance or a clock equation.

**1. Context free grammar**

**# ELEMENTARY-PROCESS ::=**
*** PROCESS-INSTANCE ***
*** SIGNAL-DEFINITION ***
*** CLOCK-EQUATION ***

**# PROCESS-INSTANCE ::=**
*** CALL ***
*** PRODUCTION ***

The elementary processes are defined in the preceding chapter.

## IV-2 Composition

The composition of processes is defined according to the syntax of the rule below:

**1. Context free grammar**

**# COMPOSITION ::=**
*** (| { P-EXPR | ... } |) ***

**2. Representation**

> (| $P_1$ | $P_2$ |) <

**3. Profile**

- !($P_1$) ∩ !($P_2$) is empty
- !((| $P_1$ | $P_2$ |)) = !($P_1$) ∪ !($P_2$)
- ?((| $P_1$ | $P_2$ |)) = (?($P_1$) - !($P_2$)) ∪ ( ?($P_2$) - !($P_1$))

**4. Semantics**

An input signal of $P_1$ (respectively of $P_2$) that has the same name as an output signal of $P_2$ (respectively of $P_1$) has as definition in $P_1$ (respectively in $P_2$) its definition in $P_2$ (respectively in $P_1$ ).

The constraints on the types and the clocks are the ones of $P_1$ and $P_2$ .

# IV-3 Profile

**1. Context free grammar**

**# PROFILE ::=**
*** P-EXPR ? { MODIFICATION , ... } ***
*** P-EXPR ! { MODIFICATION , ... } ***
*** P-EXPR / { *Signal*-name , ... } ***
*** P-EXPR !! { *Signal*-name , ... } ***
*** P-EXPR @ { *Signal*-name , ... } ***

The first argument of a **PROFILE** expression is a **P-EXPR** which cannot directly be a **SIGNAL-DEFINITION**.

## IV-3.1 Renaming

The input or output renaming expressions are obtained by using MODIFICATIONs.

**1. Context free grammar**

**# MODIFICATION ::=**
***** *Signal*-**name : ***Signal*-**name ***

**2. Semantics**

An expression of renaming is a change of signal names.

- in input, the expression $P$ ?$a_1$ : $b_1$,…,$a_n$ : $b_n$ is equal to the process P in which the input names $a_i$ have been respectively changed by the names $b_i$
- in output, the expression $P$ ! $a_1$ : $b_1$,…,$a_n$ : $b_n$ is equal to the process P in which the output names $a_i$ have been respectively changed by the names $b_i$. The names $b_i$ are all distinct; if a name $b_i$ is already a name of an output in *P*, this name has to be renamed.

**3. Examples**

- Considering a process P having three inputs a, b and c and two outputs x and y
- P ? a : b has two inputs b (to which the input a of P is identified) and c
- P ! x : y is forbidden
- P ! x : y , y : x inverts the names x and y

## IV-3.2 Restriction

**1. Semantics**

The restriction operation allows to mask outputs of a process *P*

- by restriction $P$ / $b_1$,…,$b_n$: the resulting process has as outputs, the outputs of *P* that are not in the list $b_1$,…,$b_n$
- by validation $P$ !! $b_1$,…,$b_n$: the resulting process has as outputs, the outputs of P that are in the list $b_1$,…,$b_n$

**2. Examples**

- Considering a process P having a, b and c as inputs and x and y as outputs
- P !! x has one output x

- P / y has one output x
- P / z is equal to P.

## IV-3.3 Closing

**1. Semantics**

The closing allows one to identify an output of a process *P* with an input which has the same *Signal*-**name**; by closing $P\ @b_1,\ldots,b_n$:

- the resulting process has as inputs, the inputs of *P* that are not in the list $b_1,\ldots,b_n$ and as outputs, the outputs of *P*.
- the inputs of *P* that are in the list $b_1,\ldots,b_n$ have respectively as values the values of these outputs.

**2. Examples**

- Considering a process P having a, b and c as inputs and a and y as outputs
- P @ a has two inputs b and c
- P @ y is equal to P

# IV-4 Array of processes

A **PROCESS-ARRAY** builds a process by iteration of the same elementary cell.

**1. Context free grammar**

**#PROCESS-ARRAY ::=**
*** array I-ARRAY of S-EXPR [ with { CONNECTION , ... }] end ***

**# I-ARRAY ::**
***** *Index*-**name to S-EXPR ***

**# CONNECTION ::=**
***** *Signal*-**name [0]:** *Signal*-**name ***
***** *Signal*-**name [.]:** *Signal*-**name ***
***** *Signal*-**name ***

**2. Representation**

> array *I* to *N* of *P* with $c_1$, $c_2$ end <

**2. Types**

- $\tau(N)$ =*integer*

**3. Semantics**

The process *PN* defined by the expression:
array *I* to *N* of *P* with *connections* end
is composed of a succession of processes $P_i$ built on the model of *P*.

- For each output *x* of *P* having the type $\mu$, *PN* owns an output X with the same name and with the type [1..*N*] $\rightarrow \mu$, such as *X*[i] is the output *x* of $P_i$.
- if *a* is an input of *P* that is not present in the *connections*, *a* is an input of *PN* which is diffused in the *N* cells.
- if *a* is an input of *P* present in the *connections* under the form *a*, each input of the cell $P_i$ which has this name is indexed by *i*.
- if *a* is an input of *P* present in the *connections* under the form *a*[.]:b, each input of the cell $P_i$ which has this name is indexed by *i* + 1; *P* possesses an output with the name *a*.
- if *a* is an input of *P* present in the *connections* under the form *a*[0] : b, each input of the cell $P_i$ which has this name is indexed by *i* - 1; *P* possesses an output with the name *a*.
- Each indexed input *a* is connected (is equal) to the output which has the same name and the same index.
- An input *a* with the index 0 (respectively *N* + 1) given by *a* [0] :b (respectively *a*[.]:b) becomes the input *b* of the process *PN*.
- For each input *x* of *P*, indexed and not connected, having the type $\mu$, *PN* possesses an input *X* with the same name of type [1..*N*] $\rightarrow \mu$, such as *X*[*i*] is the input *x* of $P_i$.
- The process *PN* possesses the inputs and the outputs defined according to the rules above.

- **Restriction:** it is not possible to have at the same time, inputs indexed under the form *a*[0]:b and inputs indexed under the form *a*[.]:b.

**5. Examples**

- array i to 10 of s:=a+b with a,b end is equivalent to s:=a+b
- array i to 10 of s:=a[i]+b[i] end is equivalent to s:=a+b
- array i to 10 of (| s:=y+b[i] | y:=a[i] |) with y end / y is equivalent to s:=a+b
- the process:

```
 ( | s0:=0
| array i to 10 of s:=s+a*b with a,b,s[0]:s0 end
 | ps:=s[10]
| )
```

delivers in ps the scalar product of the vectors a and b.

**6. Bug**

- If a signal external to the array of processes is used to define a clock in the array, it is considered as input of the loop, even when it is not necessary.
- A delay in a process array causes an error of the compiler.

# IV-5 Process model

A process model establishes a relation between a name and a set of equations that can be parameterized; each reference to this name is formally replaced by the equations. If the set of equations is empty (invocation of an external function), the replacement is of course partial (limited to external properties of the invoked process); the effect of the call of an external process (which may be already compiled, or not corresponding to its description) cannot be more than theoretically described. Each model invoked in the program must have a visible declaration in the syntactical context of the invocation. A process model is a term derived from **MODEL** according to the grammar below.

**1. Context free grammar**

**# MODEL ::=**
*** EXTERNAL-MODEL ***
*** DESCRIBED-MODEL ***

**# EXTERNAL-MODEL ::=**

*** function** *Model*-**name = INTERFACE ***

**# DESCRIBED-MODEL ::=**
*** process** *Model*-**name = INTERFACE DESCRIPTION end ***

**# DESCRIPTION ::=**
*** P-EXPR [ where DECLARATIONS ] ***
**# DECLARATIONS ::=**
*** [ { S-DECLARATION ; ... } ] [ { MODEL ; ... } ] ***

## IV-5.1 Local declarations of the model process

The set of sub-models $\Psi$ declared in a model *P* are not allowed to contain two models with the same name. A local declaration of a model *Q* is visible (can be the object of a **REFERENCE-MODEL**) in the expression associated with *P* and in the expressions associated with the other sub-models of *P*. For these expressions, it masks a possible model with the same name which, without it, would be visible. The expression associated with a model *P* cannot contain recursively a reference to *P*.

## IV-5.2 The interface of a model

The interface of a model contains an optional description of its formal parameters followed by a description of its visible part; this is composed of possibly empty lists of its input and output signals in this order.

**1. Context free grammar**

**# INTERFACE ::=**
*** [ PARAMETERS ] { INPUTS OUTPUTS } ***

**# PARAMETERS ::=**
*** ( [ { S-DECLARATION ; ... } ] ) ***

**# INPUTS ::=**
*** ? [ { S-DECLARATION ; ... } ] ***

**# OUTPUTS ::=**
*** ! [ { S-DECLARATION ; ... } ] ***

**2. Clocks**

- If a sub-model is an external function, its inputs and outputs are synchronous , except with the compiling option **sep** (see the appendix A).

The names of the parameters, the input signals and the output signals must be distinct from each other. A model must possess at least one input or one output or one communication with an external process with a clock not equal to zero.

## IV-5.3 Local signals

The signal names locally declared in a model must be distinct from each other; they must be distinct from signal names of the interface and parameters.

# Appendix A. Using the compiler

## A-1 Construction of a program

For purposes of explaination, let a program EXAMPLE in SIGNAL contained in a file called SOURCE.
If the program has formal parameters, their values must also be contained in a file, say PARAMETER. The first line of this file must be the line:
[OL_E_SIGN]
It is followed by the values of the effective parameters with the syntax
**{S-EXPR_,_...}**

**Calling the compiler** is done by:

***** if the program EXAMPLE has no parameters
sig { option list } SOURCE

***** if the program EXAMPLE has parameters
sig { option list } SOURCE PARAMETER

Every option of the compiler is prefixed by the character "**-**" for the UNIX version of the compiler and by the character "/" for the VMS version.
By default (without option), the compiler:

- does not generate a listing of the program with the possible errors (see the option **list**).
- does not generate sequential code (see the options **c** and **f77**).
- does not give an external representation of the program obtained after resolving the equations system (see the options **tra** and **z3z**).
- The inputs and outputs of each external model are considered as synchronous (see the option **sep**).

The different options are the following:

- **list**: produces a file EXAMPLE_LIS.SIG where EXAMPLE is the name of the SIGNAL program. This file contains:

  - the initial SIGNAL program commented when needed with error messages or warnings.

- the possible errors due to a cycle in the data or clock dependencies.

- **nowar**: the compiler does not give any remarks (or warnings )

- **sep**: the inputs and outputs of the external processes are not considered as synchronous. On the other hand, the compiler produces the transitive closure of the graph which represents the program; the transitive closure is generated in a file EXAMPLE.SEP (see the paragraph A-1.7).
- **dbg** (debug): when there is a cycle in the data or clock dependencies, a file EXAMPLE_CYC.SIG is created; this file contains the list of the signals belonging to the cycles (see the paragraph A-1.4) .With the **dbg** option, this file contains only one cycle for one signal and it is more understandable.
- **tra**: the compiler builds a file EXAMPLE_TRA.SIG which contains a SIGNAL program equivalent to the program compiled after the clock calculus (see the paragraph A-1.3)
- **z3z**: the compiler builds a file EXAMPLE.Z3Z which contains an external representation, under polynomial form, of the synchronization expressions after the clock calculus.
- **f77**: generation of FORTRAN code; the compiler builds three files EXAMPLE_M.f, EXAMPLE_S.f, EXAMPLE_E.f (see the paragraph A-1.5 )
- **f77=**x: generation of files specified by the sequence x constituted of a combination of characters “s”, “m” or “e”:
  - “**m**” production of the file EXAMPLE_M.f
  - “**s**” production of the file EXAMPLE_S.f
  - “**e**” production of the file EXAMPLE_E.f
- **g**eneration of **c** code; the compiler builds three files EXAMPLE_M.c, EXAMPLE_S.c, EXAMPLE_E.c (see the paragraph A-1.6 )
- **c=**x: generation of files specified by the sequence x constituted of a combination of characters “s”, “m” or “e”:
  - “**m**” production of the file EXAMPLE_M.c
  - “**s**” production of the file EXAMPLE_S.c
  - “**e**” production of the file EXAMPLE_E.c

**The result of the compiling**: the compiling can fail during the syntactical analysis, during the analysis of contextual properties (type or profile error), or during the generation of code.

## A-1.1 Failure during the syntactical analysis

When there is an error in the syntactical analysis, the compiler outputs the line number where the error is and a recovery code; this code refers the smallest syntactical structure in which an attempt of recovery is done. According to the correctness of this attempt to recover, the errors detected afterwards may not be real.

Example of result:

```
 SIGNAL -H2.4 Compiler
 INRIA 1993 All rights reserved
 -----------------
 You are entitled to use this software only
 if your organization has signed an agreement with INRIA
 -----------------

 ===> Program analysis
** ERROR: Syntax Error 10 line 19
```

## A-1.2 Failure during the analysis of contextual properties

In case of error messages from the compiler and if the user has given the option **list**, the file EXAMPLE_LIS.SIG will be created. This file contains the program SOURCE annotated with messages placed close to the instruction that caused the problem.

### A-1.2 A Warnings

The compiler can find instructions not corresponding to the definitions of SIGNAL. If recovery is possible and judged not dangerous, the problem is only indicated to the user as a message in the following form (extracted from the example below) and without stopping the compiling.

```
%** WARNING: Output ZERO not declared in the process interface
```

### A-1.2 B Errors

In case of error, a message is inserted close to the erroneous use of the signal; a second message is assigned to the declaration of the causing signal.

***** first message:

%** ERROR (ref 1): disagreement between declaration and use of the signal WE%

***** second message:

```
 %** ERROR: cf message No 1
 ** ERROR: cf message No 2
 %[ N ]real WE init[ { to N }: 0.0 e 0 ]
```

## A-1.2 C Complete example

After calling the compiler, the following is shown on the screen:

```
SIGNAL - H2.4 Compiler
 INRIA 1993 All rights reserved
 -----------------
 You are entitled to use this software only
 if your organization has signed an agreement with INRIA
 -----------------

 ===> Program analysis
 ===> Reduction to the kernel language
 ===> Graph generation
 ===> Clock calculus
 ===> Graph processing
```

The file COMPILING_ERROR_SIG contains:

```
process COMPILING_ERROR_LIS=
        { ?
        ! }
       (| COMPILING_ERROR( 4, 3, 3.0 e 0,[ [ 1 ]: 1.0 e 0,[ 2 ]: 2.0 e 0,[ 3 ]:
                                  3.0 e 0 ],[ [ 1 ]: 3.0 e 0,[ 2 ]: 2.
                                  0 e 0,[ 3 ]: 1.0 e 0 ])
        |)
       where
       process COMPILING_ERROR=
              ( integer N, M;
               real B0;
               [ N-1 ]real B;
               [ M ]real A )
              { ? real E
               ! real S }
```

```
            (S:= ( E*B0 )+PRODSCAL( N-1, B ){ E }+PRODSCAL( M, A)
                  { S })@ S
            where
            process PRODSCAL=
                  %** WARNING: Output ZERO not declared in the process
interface
                  %( integer N;
                   [ N ]real COEFF )
                  { ? real E
                   ! real PRSC }
                  %** WARNING: Creation of an instance of ZERO
                  %(| %** ERROR (ref 1): disagreement between dimen
sions of the operands
                        ** ERROR (ref 2): disagreement between dimen
sions of the operands
                        %WE:= E $1 window (N+1)
                        | %** WARNING: Signal or parameter not declared
                         %ZERO:= 0.0 e 0
                        |
                         array I to N
                         of PS:= PS+( WE[ ( N+1 )-I ]*COEFF[ I ])
                         with PS[0]:ZERO
                         end
                        | PRSC:= PS[ N ]
                         |)
                  where
                         %** ERROR: cf message No 1
                         ** ERROR: cf message No 2
                         %[ N ]real WE init[ { to N }: 0.0 e 0 ], PS
            end
      end
end
```

## A-1.3 Result of the compiling

In case of success in the previous steps and if the option **tra** is given, a program equivalent to the program EXAMPLE (where the formal parameters have been substituted by their values) is produced in the file EXAMPLE_TRA.SIG. This program has the same interface as the program EXAMPLE. Its body has the following structure:

- it contains a composition of processes here called **clock-processes**; each **clock-process** is constituted

  - of a clock definition **H_i_H :=** $E_i$
  - of a synchronization instruction **synchro{H_i_H, S_1,..., X_3}** giving the list **S_1,..., X_3** of external or local signals synchronous to **H_i_H**
  - of a call to a process with the same name **H_i_H**;

- it contains the declarations of the local signals communicated between its **clock-processes**;
- it contains the declarations of the models of the local processes **H_i_H** called in its process expression;
- Anomaly**:** *The presence of arrays as parameters makes the program produced by the compiler syntactically incorrect* .

Each model **H_i_H** contains, according to the same structure:

- the set of signals (associated with their processing) which have a clock **H_i_H** or a sub-clock of **H_i_H**;
- the set of clock-processes associated with the sub-clocks of **H_i_H**.

A clock **H** is a sub-clock of **H_i_H** if it is a function of boolean signals which have **H_i_H** as clock, or the clocks of which are (recursively) a sub-clock of **H_i_H**.

The failure of clock calculus results in a set of more than one clock which are not sub-clocks of a common one (there are several “main clocks”). If constraints are nevertheless expressed on these clocks, they appear under the form of synchronization instructions between clocks; in this case, there is not generation of sequential code.

## A-1.4 Failure at the generation of code

Except for the case above, the production of FORTRAN code or C (if it is required) can fail because of a cycle in the dependencies of data or of clocks. In this case, one of these two messages below are displayed on the screen:

** ERROR: Dependency cycle in the graph

** ERROR: Clocks constraints

The file EXAMPLE_CYC.SIG contains the list of signals (or clocks), because of which the generation has become impossible; these are the signals (or the clocks) that belong to the cycle.

Example of a cycle on the signals:

```
process CYCLE_SIG=
        { ? integer a
         ! integer s }
         (s := a + s)@s
end
```

The file CYCLE_SIG_CYC.SIG contains:

```
process CYCLE_SIG_CYC=
        { ?
        ! }
        (| (| S_2--> S_2 |)
         | (H_6_H:= event H_6_H)@ H_6_H
         |)
end
```

Example of a cycle on the clocks:

```
process CYCLE_CLOCK=
        { ? event OK
         ! integer N }
        (| ZN:= N $1
         | N:= ( ZN+1 )when OK
         |)
        where integer ZN init 0
end
```

The file CYCLE_CLOCK_CYC.SIG contains:

```
process CYCLE_CLOCK_CYC=
        { ?     ! }
        (| (| H_7_H--> H_7_H |)
         | (H_7_H:= OK_1 when H_7_H)@ H_7_H
         |)
end
```

## A-1.5 Result of the generation of FORTRAN code

The generation of code produces the following files:

- EXAMPLE_S.f contains a subprogram SEXAM composed by the following elements:

  - the local FORTRAN declarations corresponding to the internal signals

  - an input point IEXAM called at the beginning of the execution

  - an input point CEXAM called at every logical instant with a clock parameter which provokes the stop of the execution when it fails

  - the calls of the external functions realized with help of the instruction FORTRAN CALL

- EXAMPLE_E.f contains

  - a procedure BEGIO to open the input-output files

  - a procedure RSIG to read the data of the signal SIG in the file RSIG, for each input signal; at the end of the input, the clock to stop the execution is set to **false**

  - a procedure WSIG to write the calculated values of the signal SIG in the file WSIG, for each output signal

  -if needed (if there is not only one “main clock”), a procedure HSIG to read the boolean signal associated with the input signal **SIG**, which indicates the instants where the signal **SIG** is present (the value of the boolean is *true*) and the instants where the signal **SIG** is absent (the value of the boolean is *false*).

  - a procedure ENDIO to close input-output files

- EXAMPLE_M.f contains

  - the call to the initialization BEGIO of the input-output files

  - the call to the initialization IEXAM

  - an iteration of the call to CEXAM until the call result is **false**

  - the call to the closing ENDIO of the input-output files

## A-1.6 Result of the generation of C code

The generation of code produces the following files:

- EXAMPLE_S.c contains

  - the declarations corresponding to the signals

- a function **iexam** called at the beginning of the execution

- a function **cexam** called at every logical instant as long as it gives the value **true** and which provokes the stop of the execution when it gives the value **false**

- EXAMPLE_E.c contains

  - a procedure **begio** to open the input-output files

  - a procedure **rsig** to read the data of the signal **sig** in the file RSIG.dat, for each input signal; at the end of the input, the clock to stop the execution is set to **false**

  - a procedure **wsig** to write the calculated values of the signal **sig**, in the file WSIG.dat, for each output signal

  -if needed (if there is not only one “main clock”), a procedure **hsig** to read the boolean signal associated with the input signal **sig** in the file HSIG.dat. This boolean indicates the instants where the signal **sig** is present (the value of the boolean is *true*) and the instants where the signal **sig** is absent (the value of the boolean is *false*).

  - a procedure **endio** to close the input-output files

- EXAMPLE_M.c contains

  - the call to the initialization **begio** of the input-output files

  - the call to the initialization **iexam**

  - an iteration of the call to CEXAM until the call result is **false**

  - the call to the closing ENDIO of the input-output files

## A-1.7 Result with the option sep

The compiler produces the transitive closure of the graph which represents the program; this transitive closure is calculated between the inputs and outputs of the program.
Two types of elementary processes are used:

- synchro{H,A} which specifies that H is the clock of A

- (X --> Y) when H which specifies the dependency of X

```
process ASYNCDEF_TRA=
        { ? integer A_1, B_2
          ! integer X_3 }
        (| (| H_5_H:= event A_1|)
         | (| H_6_H:= event B_2 |)
         | (| H_8_H:= when( ( not H_5_H )default H_6_H )|)
         | (| H_9_H:= H_5_H default H_6_H
            | synchro { H_9_H, X_3 }
```

```
          | X_3:= ( A_1 when H_5_H )default( B_2 when H_8_H )
          |)
        |)
         where
         event H_9_H, H_8_H, H_6_H, H_5_H
end

process ASYNCDEF_SEP=
        { ?
         ! }
        (| synchro { H_5_H, A_1 }
         | synchro { H_6_H, B_2 }
         | ( A_1--> X_3) when H_5_H
         | ( B_2--> X_3) when H_8_H
         | synchro { H_9_H, X_3 }
         |)
end
```

# A-2 A complete example: Recursive filtering

Here we present a program of recursive filtering written in SIGNAL and the different products of the compiling.

## A-2.1 The SIGNAL program

```
process FILTER_REC_2 =
        ( integer N, M ;
         real B0 ;
         [N-1] real B ;
         [N] real A )
        { ? real E
         ! real S
        }
        (| S:= (E * B0) +
                PRODSCAL (N-1,B) {E}
                +
                PRODSCAL (M, A) {S}
        |) @ S
        where
```

```
        process PRODSCAL =
                (integer N ;
                [N] real COEFF )
                { ? real E
                        ! real PRSC
                }
                (| WE := E $ 1 window N
                 | ZERO := 0.0e 0
                 | array I to N
                        of PS := PS + (WE [ (N + 1) -I] * COEFF [I] )
                        with PS [0] : ZERO
                 end
                 | PRSC := PS [N]
                 |)
                where
                        [N] real WE init [ { to N } : 0 .0e 0] , PS ;
                        real ZERO
        end
end
```

## A-2.2 The parameters

```
[OL_E_SIGN]
4,3,3.0,[[1]:1.0,[2]:2.0,[3]:3.0],[[1]:3.0,[2]:2.0,[3]:1.0]
```

## A-2.3 The SIGNAL program produced by the compiler

```
process FILTER_REC_2_TRA =
        {? real E_6
         ! real S_7
        }
        (| (| H_9_H := event E_6
           | synchro {H_9_H, S_7 }
           | H_9_H ()
           |)
        |)
        where event H_9_H
        process H_9_H =
                { ? event H_9_H ;
                   real E_6
```

```
            ! real S_7
            }
            (| synchro {H_9_H, PRSC_8, WE_14, PS_15, ZERO_16,
                        PRSC_18, WE_23, PS_24, ZERO_25}
            |       (| WE_23 := S_7 $ 1 window 3
                     | WE_14 := E_6 $ 1 window (4 - 1 )
                     | ZERO_25 := 0. 0e 0 when H_9_H
                     | array I_26 to 3
                           of PS_24 := (PS_24 + (WE_23 [ ( 3 + 1 ) - I_26 *
                                 [ [ 1 ]: 3. 0e 0, [ 2 ]: 2. 0e 0, [ 3 ]: 1. 0e 0 ] [ I_26
                                 ] ) ) when H_9_H
                           with PS_24 [ 0 ]: ZERO_25
                     end
                     | PRSC_18 := PS-24 [ 3 ] when H_9_H
                     | ZERO_16 := 0. 0e 0 when H_9_H
                     | array I_17 to 4 - 1
                           of PS_15 := (PS_15 + (WE_14 [ ( ( 4-1 ) + 1 - I_17 ] *
                                 [ [ 1 ]: 1. 0e 0, [ 2 ]: 2. 0e 0, [ 3 ]: 3. 0e 0 ] [ I_17
                                 ] ) ) when H_9_H
                           with PS_15 [ 0 ]: ZERO_16
                     end
                     | PRSC_8 := PS_15 [ 4 - 1 ] when H_9_H
                     | S_7 := ( ( E_6 * 3. 0e 0) + PRSC_8 + PRSC_18 ) when
                           H_9_H
                     |)
            |)
            where
                     real PRSC_8, ZERO_16, PRSC_18, ZERO_25 ;
                     [3] real WE_14 init [ {XZX_11 to 4 - 1} : 0. 0e 0 ] ;
                     [3] real PS_15 ;
                     [3] real WE_23 init [ {XZX_20 to 3} : 0. 0e 0 ] ;
                     [3] real PS_24 ;
      end
end
```

## A-2.4 The FORTRAN programs produced by the compiler

### A-2.4 A The main program

C

```
      LOGICAL H
     H = .TRUE.
      CALL BEGIO
     CALL IFILT
1    CALL CFILT(H)
     IF(H) GOTO 1
     CALL ENDIO
      END
```

## A-2.4 B The input-output subprogram

```
C
      SUBROUTINE EFILT
      LOGICAL H4H,H9H
      INTEGER IE6
      REAL TE6(0:3)
      INTEGER IS7
      REAL TS7(0:3),PRSC8
      INTEGER XZX11,IWE14
      REAL PS15(0:4),ZER16
      INTEGER I17
      REAL PRS18
      INTEGER XZX20,IWE23
      REAL PS24(0:4),ZER25
      INTEGER I26
C Input-output initializations
      INPUT BEGIO
      OPEN (10,FILE = 'RE',STATUS = 'OLD ')
      REWIND (10)
      OPEN (11,FILE = 'WS',STATUS = 'NEW ')
      REWIND (11)
      RETURN
C Close input-output files
      INPUT ENDIO
      CLOSE (10)
      CLOSE (11)
      RETURN
C Read input : E
      INPUT RE(TE6,IE6,H4H)
      READ (10,*,END = 1) TE6(IE6)
       RETURN
C Write output : S
      INPUT WS(TS7,IS7)
```

```
          WRITE (11,*) TS7(IS7)
          RETURN
1         H4H= .FALSE.
        END
```

## A-2.4 C The treatment subprogram

```
C
          SUBROUTINE SFILT
C Declaration of signals
          LOGICAL H4H
          REAL A5(1:3),B4(1:3)
          INTEGER IE6
          REAL TE6(0:3)
          INTEGER IS7
          REAL TS7(0:3),PRSC8
          INTEGER XZX11,IWE14
          REAL PS15(0:4),ZER16
          INTEGER I17
          REAL PRS18
          INTEGER XZX20,IWE23
          REAL PS24(0:4),ZER25
          INTEGER I26
C Declaration of clocks
          LOGICAL H9H
C Body of the initialization procedure
          INPUT IFILT
          A5(1)=3.0E0
          A5(2)=2.0E0
           A5(3)=1.0E0
           B4(1)=1.0E0
          B4(2)=2.0E0
          B4(3)=3.0E0
          IE6=2
          IS7=2
          IWE14=0
          DO 1 XZX11=1,3
1         TE6(MOD(IWE14+ XZX11- 1,4))=0.0E0
          IWE14=3
          IWE23=0
          DO 2 XZX20=1,3
2         TS7(MOD(IWE23+ XZX20- 1,4))=0.0E0
          IWE23=3
          ZER16=0.0E0
```

```
      ZER25=0.0E0
      RETURN
C Body of the program
      INPUT CFILT(H4H)
      H9H= .TRUE.
      IF (IE6 .EQ. 3)THEN
      IE6=0
      ELSE
       IE6=IE6+ 1
      ENDIF
       CALL RE(TE6,IE6,H4H)
       IF ( .NOT. (H4H))RETURN
      IF (IWE14 .EQ. 3)THEN
      IWE14=0
       ELSE
      IWE14=IWE14+ 1
      ENDIF
      PS15(0)=0.0E0
      DO 3 I17=1,3
      PS15(I17)=(PS15(I17- 1))+ ((TE6(MOD(IWE14+ ((3)+ 1)- (I17)- 1,4)))
      &* (B4(I17)))
3     CONTINUE
      PRSC8=PS15(3)
      IF (IWE23 .EQ. 3)THEN
      IWE23=0
       ELSE
      IWE23=IWE23+ 1
      ENDIF
      PS24(0)=0.0E0
      DO 4 I26=1,3
      PS24(I26)=(PS24(I26- 1))+ ((TS7(MOD(IWE23+ ((3)+ 1)- (I26)- 1,4)))
      & * (A5(I26)))
4     CONTINUE
      PRS18=PS24(3)
      IF (IS7 .EQ. 3)THEN
       IS7=0
      ELSE
      IS7=IS7+ 1
      ENDIF
      TS7(IS7)=((TE6(IE6))* (3.0E0))+ (PRSC8)+ (PRS18)
      CALL WS(TS7,IS7)
      RETURN
      END
```

## A-2.5 The C programs produced by the compiler

### A-2.5 A The main program

```
typedef int event;
typedef int logical;

#define TRUE 1
#define FALSE 0

extern logical cfilter_rec_2();
extern void ifilter_rec_2();
extern void begio();
extern void endio();

extern int main()
{
        begio();
        ifilter_rec_2();
       while(cfilter_rec_2());
       endio();
}
```

### A-2.5 B The input-output subprogram

```
#include <stdio.h>

typedef int event;
typedef int logical;

#define TRUE 1
#define FALSE 0

FILE
        *fre_6,
        *fws_7;

int i01, i02, i03, i04, i05, i06, i07, i08, i09, i010;

extern void begio()
{
```

```
        fre_6 = fopen("RE.dat","r");
        fws_7 = fopen("WS.dat","w");
}

extern void endio()
{
        close(fre_6);
        fclose(fws_7);
}

extern void re_6(te_6,ie_6,h_4_h)
float te_6[4];
int ie_6;
event *h_4_h;
{
       *h_4_h = (fscanf(fre_6,"%f",&te_6[ie_6])!=EOF);
}

extern void ws_7(ts_7,is_7)
float ts_7[4];
int is_7;
{
        fprintf(fws_7,"%f\n",ts_7[is_7]);
}
```

## A-2.5 C The treatment subprogram

```
typedef int event;
typedef int logical;

#define TRUE 1
#define FALSE 0


extern void re_6();

extern void ws_7();

/*C Declaration of signals */

float a_5[4], b_4[4];

int ie_6;
```

```
float te_6[4];

int is_7;
float ts_7[4];

float prsc_8;
int xzx_11, iwe_14;
float ps_15[5], zero_16;
int i_17;
float prsc_18;
int xzx_20, iwe_23;
float ps_24[5], zero_25;
int i_26;

int i01, i02, i03, i04, i05, i06, i07, i08, i09, i010;

/*C Declaration of clocks */

event h_9_h;

event h_4_h;


/*C Body of the program */

extern void ifilter_rec_2()
{
        a_5[1] = 3.0e0;
        a_5[2] = 2.0e0;
        a_5[3] = 1.0e0;
        b_4[1] = 1.0e0;
        b_4[2] = 2.0e0;
        b_4[3] = 3.0e0;
        ie_6 = 2;
        is_7 = 2;
        iwe_14 = 0;
        for (xzx_11 = 1;xzx_11<=3;xzx_11++)
                te_6[(iwe_14+xzx_11-1)%4] = 0.0e0;
       iwe_14 = -1;
        iwe_23 = 0;
        for (xzx_20 = 1;xzx_20<=3;xzx_20++)
                ts_7[(iwe_23+xzx_20-1)%4] = 0.0e0;
```

```
        iwe_23 = -1;
        zero_16 = 0.0e0;
        zero_25 = 0.0e0;
}

/* Body of the program */

extern logical cfiltre_rec_2()
{
        h_9_h = TRUE;
        ie_6 = (ie_6+1)%4;
        re_6(te_6,ie_6,&h_4_h);
        if (!h_4_h) return FALSE;

        iwe_14 = (iwe_14+1)%4;
       ps_15[0] = 0.0e0;
       for (i_17 = 1;i_17<=3;i_17++)
                {
               ps_15[i_17] = ps_15[i_17-1] + te_6[(iwe_14+((3 + 1) - i_17)-
                                1)%4] * b_4[ i_17];
                }
        prsc_8 = ps_15[3];
        iwe_23 = (iwe_23+1)%4;
       ps_24[0] = 0.0e0;
        for (i_26 = 1;i_26<=3;i_26++)
                {
                ps_24[i_26] = ps_24[i_26-1] + ts_7[(iwe_23+((3 + 1) - i_26)-
                                1)%4] * a_5[ i_26];
               }
        prsc_18 = ps_24[3];
       is_7 = (is_7+1)%4;
        ts_7[is_7] = (te_6[ie_6] * 3.0e0 + prsc_8) + prsc_18;
        ws_7(ts_7,is_7);


        return TRUE;
}
```

# A-3 An example of hierarchy

Here we present a program of a mouse handler written in SIGNAL and the SIGNAL program resulting from the compiling.

## A-3.1 The SIGNAL program

```
process MOUSE=
      { ? event CLICK, TOP
       ! integer M }
      (| (| N:= ( 0 when RESET )default( ZN+1 )
         | synchro { N, TOP }
         | ZN:= N $1
         | RESET:= when( ZN=4 )
         |)
       | (|X:= ( 0 when RESET )default NEW_X
         | synchro { X, CLICK default RESET }
         | NEW_X:= MIN{ 2, ZX+1 }
         | ZX:= X $1
         |)
      | (| M:= ( NEW_X when CLICK when RESET )default( X when RESET )|)
      |)
 where
         integer X, NEW_X, ZX init 0, N, ZN init 0;
         event RESET
 function MIN=
         { ? integer U, V
          ! integer V }
 end
end
```

## A-3.2 The SIGNAL program produced by the compiler

```
process MOUSE_TRA=
      { ? event CLICK_1, TOP_2
       ! integer M_3 }
      (| (| synchro { TOP_2, TOP_2 }
         | TOP_2( )
         |)
       | (| H_14_H:= when( ( not RESET_15 )default CLICK_1 ) |)
```

```
    | (| synchro { CLICK_1, CLICK_1 } |)
     | (| H_22_H:= RESET_15 when CLICK_1 |)
     | (| H_27_H:= when( ( not H_22_H )default RESET_15 ) |)
     | (| H_29_H:= RESET_15 default CLICK_1
       | synchro { H_29_H, X_10, NEW_X_11 }
       | H_29_H( )
       |)
    |)
where
    event H_29_H, H_27_H, H_22_H, H_14_H, RESET_15;
    integer X_10, NEW_X_11
process TOP_2=
    { ? event H_27_H, H_22_H;
       event TOP_2;
       integer X_10, NEW_X_11
     ! event RESET_15;
       integer M_3 }
    (| synchro { TOP_2, N_13, ZN_14 }
     | (| H_8_H:= when( ( not RESET_15 )default TOP_2 ) |)
     | (| RESET_15:= when( ZN_14=4 )
       | synchro { RESET_15, M_3 }
       | M_3:= ( NEW_X_11 when H_22_H )default( X_10 when H_27_H )
       |)
     | (| ZN_14:= N_13 $1
       | N_13:= ( 0 when RESET_15 )default( ( ZN_14+1 )when H_8_H )
       |)
     |)
    where
        event H_8_H;
        integer N_13, ZN_14 init 0
end;
process H_29_H=
    { ? event H_29_H, H_14_H, RESET_15
     ! integer X_10, NEW_X_11 }
    (| synchro { H_29_H, ZX_12, V_18, U_19 }
     | (| ZX_12:= X_10 $1
       | X_10:= ( 0 when RESET_15 )default( NEW_X_11 when H_14_H )
       | V_18:= ( ZX_12+1 )when H_29_H
       | U_19:= 2 when H_29_H
       | NEW_X_11:= MIN{ U_19, V_18 }
       |)
     |)
    where integer ZX_12 init 0, V_18, U_19
```

```
 end;
 function MIN=
       { ? integer U, V
       ! integer V }
end
```

# Appendix B. The messages from the compiler

## B-1 Principles

The messages sent by the compiler are either errors or warnings, prefixed respectively by “** ERROR:” and “** WARNING:”. They are given under one of these following forms:

***** simple message, as:
“** ERROR: constant not allowed”

***** message specified by the name of the signal, parameter, process,...causing the error, such as:
“** ERROR Double declaration of TOTO”

*****set of three messages concerning the errors in a process call as in the following example:

```
process P=
        { ? %** ERROR: cf message No 1
            %logical B
          ! integer C }
        %** ERROR (ref 1): disagreement between type and use of
the signal
        %C := PP {B}
        where
        process PP=
            { ? integer I
              ! integer Z }
            %** ERROR: disagreement between types of the
operands
            %Z := I**2
        end
end
```

- the first message is *attached* to the declaration of the concerned signal.
- the second one is *attached* to the call of the process.

- the third one is *attached* to the instruction in the called process where the signal is incorrectly used.

This appendix contains the list of messages, with comments if necessary.

# B-2 Errors

## B-2.1 Syntactical analysis

The errors found during the syntactical analysis appear under the following form:
**Syntax Error** *code* **line** *n*
in which *n* is the number of the erroneous line and *code* refers the erroneous syntactical structure according to the coding below:

***** 10: syntax error in a **MODEL**
***** 13: syntax error in a model **INTERFACE**
***** 14: syntax error in a **P-EXPR** (process expression)
***** 16: syntax error in an **S-DECLARATION** (signals declaration)
***** 54: syntax error in an **S-EXPR (**signals expression)

## B-2.2 Compiler call

# **Conflicting number of parameters**: compiler call with an incorrect file of parameters.

## B-2.3 Declaration of a MODEL

# **Process XXX undeclared**: process or external function undeclared.

# **Incorrect process declaration**: needs input or output

# **Parameters not allowed in an external process declaration**: an external process is a signal function which cannot have any parameter.

# **Parameter not declared**: an identifier has been determined as a parameter and is not declared.

# **Conflicting number of input signals**

# **Input XXX not declared in the process interface**

# **Conficting number of ouput signals**

# **Output XXX not defined in the process body interface**

## B-2.4 S-DECLARARATION

# **Signal-index identifier conflict: XXX**

# **Double declaration of XXX**

## B-2.5 P-EXPR

# **Conflicting output names : XXX**

# **Conflict while relabelling a process array**: opposite connections of signals in a process array (X and Y in the following example).
example:
array I to N of
.....
with X[.]:B, Y[0]:A end

# **Not an output**: Closing operation on an input without output with the same name, as in the following example:

(C := B*2) @ B

## B-2.6 S-EXPR

# **Identifier conflict**: use of a signal (resp. a parameter or an index), in a context where a signal (resp. a parameter or an index) is forbidden.

# **Expected a process call**: in the context of a multiple definition, the right term must be a call.

## B-2.7 Dimensions

# **Dimensions of XXX incompatible with its use**

# **Dimensions of operands incompatible**: for instance, the sum of two arrays which do not have the same dimension.

# **Disagreement between types of the concatenation operands**: only a Vector-Name is allowed as argument of a concatenation.

# **Scalar operands expected for the relation expression**: the arrays are not allowed in a relation expression.

## B-2.8 Types

# **Disagreement between type and use of the signal**

# **Disagreement between dimensions of operands**

# **XXX, used in a process array, is not defined**

## B-2.9 Constant integers

# **Process array and data dimension conflict for XXX**

# **Strictly positive integer expression expected**

# **Disagreement between the iteration bound on XXX and its step size**

# **Step size of iteration on XXX equal to zero**

# **Integer type expected**

## B-2.10 Clocks

# **Constant not allowed**: the current context does not allow the use of a constant.

# **Cannot define the clock of the expression**: the use of a constant in a context in which no clock can be associated, for instance X:= 3 cell C.

## B-2.11 Diverse

# **Error**: error caused by a division by zero
or
by a cycle in the process instantiations

# B-3 Warnings

## B-3.1 MODEL declaration

$ **External function lacks input**

$ **Input XXX unused in the process body**

$ **Output XXX not declared in the process interface**

## B-3.2 S-DECLARATION

$ **Identical name for output and local signal (local ignored)**

$ **Signal or parameter not declared**

## B-3.3 P-EXPR

$ **Unknown output**: masking of an output which does not exist.
$ **Double relabelling of an input**: for example, relabelling X[.]:X in a process array.

## B-3.4 S-EXPR

$ **Useless operation**: for instance, P!A:B where A is not an output of P.

## B-3.5 Dimensions

$ **Range of an index of XXX can't be verified**: an array index is a signal or an arithmetic expression the bounds of which cannot be determined.

$ **The bounds on XXX cannot be controlled**: iteration expression the bounds of which cannot be determined.

$ **A vector element cannot be controlled**: an index of a vectorial element is either a signal or an arithmetic expression where the bounds cannot be determined.

## B-3.6 Diverse

$ **The two operands have no type (no verification possible)**

$ **The initialization of XXX is not necessary**: initialization of XXX not used.

$ **XXX should be initialized**: XXX is the result of a delay expression ($ or window) or of a cell.

$ **Creation of an instance of XXX**: a signal is used but not declared.

# Appendix C. SIGNAL grammar

## C-1 The equations

### C-1.1 The expressions on equations

As first profile operators argument (?, !, /, !!, @), a signal definition (:=) must be surrounded by parentheses.

* **Context free grammar** P-EXPR

**# P-EXPR ::=**
***** *Signal*-**name :=** **S-EXPR ***
*** { {** *Signal*-**name , ... } } := S-EXPR ***
***** *Model*-**name ( [ { S-EXPR , ... } ] ) ***
***** *Model*-**name ( [ { S-EXPR , ... } ] ) { [ { S-EXPR , ... } ] } ****
***** *Model*-**name ( [ { S-EXPR , ... } ] ) ***
*** synchro { { S-EXPR , ... } } ***
*** ( P-EXPR ) ***
*** (| { P-EXPR | ... } |) ***
*** P-EXPR ? {** *Signal*-**name :** *Signal*-**name , ... } ***
*** P-EXPR ! {** *Signal*-**name :** *Signal*-**name , ... } ***
*** P-EXPR / {** *Signal*-**name , ... } ***
*** P-EXPR !! {** *Signal*-**name , ... } ***
*** P-EXPR @ {** *Signal*-**name , ... } ***
*** array I-ARRAY of S-EXPR [ with { CONNECTION , ... } ] end ***

**# I-ARRAY ::=**
***** *Index*-**name to S-EXPR ***

**# CONNECTION ::=**
***** *Signal*-**name ***
***** *Signal*-**name [0]:** *Signal*-**name ***
***** *Signal*-**name [.]:** *Signal*-**name ***

### C-1.2 Declaration of equation models

* **Context free grammar** MODEL

**# MODEL ::=**
*** function** *Model*-**name = INTERFACE ***
*** process** *Model*-**name = INTERFACE P-EXPR**
**[ where [ { S-DECLARATION ; ...} ] [ { MODEL ; ... } ] ] end ***

### C-1.3 Interface of a model

* **Context free grammar** INTERFACE

**# INTERFACE ::=**
*** [ ( [ { S-DECLARATION ; ...} ] ) ]**
**{ ? [ { S-DECLARATION ; ...} ]**
**! [ { S-DECLARATION ; ...} ] } ***

## C-2 The signals

### C-2.1 The signal and parameter declarations

* **Context free grammar** S-DECLARATION

***# S-DECLARATION ::=***
**** {*** *Signal*-**name** ***[ init S-EXPR ] , ...} ****
**** SIGNAL TYPE {*** *Signal*-**name** ***[ init S-EXPR ] , ...} ****

**# SIGNAL-TYPE ::=**
*** event ***
*** Element-type ***
**** [ { S-EXPR , ... } ] Element-type ****

**# Element-type ::=**
*** logical ***
*** Numerical-type ***

**# Numerical-type ::=**
*** integer ***
*** real ***
*** dpreal ***
*** complex ***

**# Scalar-type ::=**
*** Synchronization-type ***
*** Numerical-type ***

**# Synchronization-type ::=**
*** event ***
*** logical ***

## C-2.2 The expressions on signals

An elementary expression (**ELEMENTARY-S-EXPR**) can be argument of a scalar expression (**SCALAR-S-EXPR**) or of an array expression (**ARRAY-S-EXPR**) without being within parentheses.

*** Context free grammar S-EXPR**

**# S-EXPR ::=**
***ELEMENTARY- S-EXPR ***
*** SCALAR-S-EXPR ***
*** ARRAY-S-EXPR ***

**# ELEMENTARY-S-EXPR ::=**
***** *Signal*-**name [ [ { S-EXPR , ... } ] ] ***
***** *Model*-**name [ ( { S-EXPR , ... } ) ] { [ { S-EXPR , ... } ] } ***
*** ( S-EXPR ) ***

### C-2.2 A The scalar expressions

They appear in decreasing priority order except the dynamical expressions which cannot be direct argument of unary arithmetic operators and which allow as second integer argument these same operators.

* **Context free grammar** SCALAR-S-EXPR

**# SCALAR-S-EXPR ::=**
*** DYNAMICAL-S-EXPR ***
*** ARITHMETIC-S-EXPR ***
*** BOOLEAN-S-EXPR ***
*** TEMPORAL-S-EXPR ***

**#TEMPORAL-S-EXPR ::=**
*** *Signal*-name $ S-EXPR ***
*** *Signal*-name [ $ S-EXPR ] window S-EXPR ***

**# ARITHMETIC-S-EXPR ::=**
*** Integer-cst ***
***REAL- CST ***
*** COMPLEX-CST ***
*** + S-EXPR ***
*** - S-EXPR ***
*** S-EXPR * S-EXPR ***
*** S-EXPR / S-EXPR ***
*** S-EXPR ** S-EXPR ***
*** S-EXPR + S-EXPR ***
*** S-EXPR - S-EXPR ***

**# BOOLEAN-S-EXPR ::=**
*** true ***
*** false ***
*** S-EXPR = S-EXPR ***
*** S-EXPR /= S-EXPR ***
*** S-EXPR > S-EXPR ***
* S-EXPR >= S-EXPR *
*** S-EXPR < S-EXPR ***
*** S-EXPR <= S-EXPR ***
*** not S-EXPR ***
*** S-EXPR or S-EXPR ***
*** S-EXPR and S-EXPR ***

**# TEMPORAL-S-EXPR ::=**
*** # S-EXPR [ from S-EXPR ] ***
*** # S-EXPR after S-EXPR ***
*** S-EXPR default S-EXPR ***
*** S-EXPR when S-EXPR ***
*** S-EXPR cell S-EXPR ***
*** event S-EXPR ***
*** when S-EXPR ***

## C-2.2 B The array expressions

* **Context free grammar** ARRAY-S-EXPR

**# ARRAY-S-EXPR ::=**
*** {** *Signal*-**name || ... } ***
*** [ { ITERATION , ... } ] ***

**# ITERATION ::=**
*** [ { S-EXPR , ... } ] : S-EXPR ***
*** { { [** *Index*-**name ] [ in S-EXPR ] [ to S-EXPR ] [ step S-EXPR ] , ... }**
**} : [ [ { S-EXPR , ... } ] ] : S-EXPR ***

# Table of contents

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399