SIGNAL V4 – INRIA version: Reference Manual
(working version)

Loïc BESNARD [1] Thierry GAUTIER[2] Paul LE GUERNIC[3]

March 16, 2004

[1]IRISA/CNRS, Campus de Beaulieu, 35042 Rennes Cedex (France) — e-mail: Loic.Besnard@irisa.fr
[2]IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex (France) — e-mail: Thierry.Gautier@irisa.fr
[3]IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex (France) — e-mail: Paul.LeGuernic@irisa.fr

Abstract

SIGNAL is a synchronized data flow language designed for programming real-time systems. A SIGNAL program defines both data and control processing, from a system of equations, the variables of the system are signals. These equations can be organized as sub-systems (or processes). A signal is a sequence of values which has a clock associated with; this clock specifies the instants at which the values are available.

This reference manual defines the syntax and the semantics of the INRIA version of the SIGNAL V4 language. The original official definition of the SIGNAL V4 language was published in french in june 1994. It is available at the following address:
ftp://ftp.irisa.fr/local/signal/publis/research_reports/PI832-94:v4_manual.ps.gz
It was defined together with François DUPONT, from TNI[1]. Some of the evolutions described in this document have been defined too in cooperation with François DUPONT. However, the SIGNAL version implemented by TNI in the SILDEX environment is slightly different in some aspects from the version described here. A description of SILDEX may be found at the following address:
http://www.tni-valiosys.com/

The definition of the SIGNAL version described in this manual is subject to evolutions. It is (partly) implemented in the INRIA POLYCHRONY environment. Consult the following site:
http://www.irisa.fr/espresso/Polychrony

[1]TNI-Valiosys, Technopôle Brest Iroise, Z.I. Pointe du Diable, BP 70801, 29608 Brest Cedex (France) — e-mail: francois.dupont@tni-valiosys.com

Main evolutions of this document

From version dated December 18, 2002 to the present one:

- correction of errors and update of implementation notes;
- addition of the predefined function `indices` (cf. section IX–10, page 156) and other precisions related to spatial processing (cf. chapter IX, page 147).

Table of contents

Part A

INTRODUCTION

Chapter I

Introduction

The SIGNAL language has been defined at INRIA/IRISA with the collaboration and support from the CNET. This reference manual defines the syntax and semantics of the INRIA version of the language, which is an evolution of the V4 version. The V4 version resulted from a synthesis of experiments made by IRISA and by the TNI company. An environment of the SIGNAL language can be built in a style and in a way it is not the objective of this manual to define. However, such an environment will have to provide functions for reading and writing programs in the form specified in this manual; the translation scheme will give the semantics of the texts built in this environment.

I–1 Main features of the language

A program expressed in the SIGNAL language defines some data and control processing from a system of equations, the variables of which are identifiers of *signals.* These equations can be organized in sub-systems (or *processes*). A *model of process* is a sub-system which may have several using contexts; for that purpose, a model is designated by an identifier. It can be provided with parameters specifying data types, initialization values, array sizes, etc. In addition, sets of declarations can be organized in modules.

I–1.1 Signals

A signal is a sequence of values, with which a clock is associated.

1. **All the values of a signal** belong to a same *sub-domain* of a *domain of values,* designated by their common *type.* This type can be:

   - predefined (the Booleans, sub-domains of the Integers, sub-domains of the Reals, sub-domains of the Complex…),
   - defined in the program (Arrays, Tuples),
   - or referenced in the program but known only by the functions that handle it (Externals).

2. **The clock of a signal** allows to define, relatively to a totally ordered set containing at least as much elements as the sequence of values of this signal, the subset of instants at which the signal has a value. A pure signal, the value of which belongs to the singleton $event$, can be associated with each signal. This pure signal is present exactly at the presence instants of the signal; the $event$ type is a sub-domain of the Booleans. By extension, this pure signal will be called *clock.* A pure signal is its own clock. In a process, the clock of a signal is the representative of the equivalence

class of the signals with which this signal is *synchronous* (synchronous signals have their values at the same instants).

3. These values are expressed in equations of definition and in constraints.

I–1.2 Events

A valuation associates, at a logical instant of the program (transition of the automaton), a value with a variable.

An event is a set of simultaneous valuations defining a transition of the automaton. In an event, a variable may have no associated value: it will be said that the corresponding signal is absent and its “value” will be written $\perp$. An event contains at least one valuation.

Determining the presence of a signal (i.e., a valuation) in an event results from the solving of a system of equations in $\mathcal{F}_3$, the field of integers modulo 3.

The value associated with a variable in an event results from the evaluation of its expression of definition (thus it should not be implicit: circular definitions of non Boolean signals are not allowed).

I–1.3 Models

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

A model may be defined outside the program; in that case, it is visible only through its interface. Calling a model defined in a program is equivalent to replacing this call by the associated system of equations (macro-substitution).

Invoking a model defined outside the program can produce side-effects on the context in which the program is executed; these effects can be directly or indirectly perceived by the program and they can affect the set of instants or the set of values of one or more interface signals. Such a model will be said non functional (for example a *random* “fonction” is such a non functional model).

I–1.4 Modules

The notion of module allows to describe an application in a modular way. In particular, it allows the definition and use of libraries written in SIGNAL or external ones, and constitutes an access interface to external objects.

I–2 Model of sequences

A program expressed in the SIGNAL language establishes a relation between the sequences that constitute its external signals. The set of programs of the SIGNAL language is a subset of the space of subsets of sequences (part B, chapter III).

I–3 Static semantics

The relations on sequences presented in the formal model describe a set of programs among them are only considered as legal programs those for which the ordering of each set of instants is in accordance with the ordering induced by the dependencies (causality principle), and which do not contain implicit definitions of values of non Boolean signals.

I–3.1 Causality

A real-time program has to respect the causality principle: according to this principle, the value of an event at some instant t cannot depend on the value of a future event. The respect of this principle is obtained in SIGNAL language by the implicit handling of time: the user has a set of terms that allow him/her to make reference to passed or current values of a signal, not to future ones.

I–3.2 Explicit definitions

The synchronous hypothesis on which is based the definition of the SIGNAL language allows to develop a calculus on the time considered as a pre-order in a discrete set.

I–4 Subject of the reference

This manual defines the syntax, the semantics, and formal resolutions applied by a compiler to a program expressed in the SIGNAL language. The SIGNAL language has four classes of syntactic structures:

1. **The structures of the kernel language** for which a formal definition is given in the model of sequences. The kernel language contains a minimal set of operators on sequences of signals of type $event$ and $boolean$ on which the temporal structure of the program is calculated; it contains also a mechanism allowing to designate signals of external types and non interpreted functions aplying to these signals. Removing anyone of these structures would strictly reduce the expressiveness of the language.

2. **The structures of the minimal language** that can be subdivided in three sub-classes:

   (a) the non Boolean types and the associated operators, which allow to write a program completely in the SIGNAL language; **the open vocation of the SIGNAL language is nevertheless clearly asserted:** it is possible to use external functions/processes, defined in another language, or even realized by some hardware component; this is even advised when specific properties exist, that are not handled by the formal calculi made possible in the SIGNAL language;

   (b) the syntactic structures providing to the language an extensability necessary for its specialization for a particular application domain, and for its opening toward other environments or languages;

   (c) the operators and constructors of general use providing a programming style that favours the development of associated methodologies and tools.

3. **The standard (or intrinsic) process models** which form a library common to all the compilers of the SIGNAL language;

4. **The specific process models** which constitute specific extensions to the standard library.

This manual describes the structures of the kernel language and of the minimal language.

I–5 Form of the presentation

Three classes of terms are distinguished for the description of the syntax of the language:

- the vocabulary of the lexical level: each one of the **terminals** designates an enumerated set of indivisible sequences of characters;
- the lexical structures: the **Terminals** of the syntactic level are defined, at a lexical level, by rules in a grammar the vocabulary of which is the union of the **terminals** sets; no implicit character (separators, for instance) is authorized in the terms constructed following these rules;
- the syntactic structures: the **NON-TERMINALS** are defined, at a syntactic level, by rules in a grammar the vocabulary of which is composed of the **Terminals**; any number of separators can be inserted between two **Terminals**.

Every unit of the language is introduced and then described, individually or by category, with the help of all or part of the following items. Generally, a generic term representing the unit is given:
EXPRESSION(E_1, E_2, ...)
where E_1, E_2, ... are formal arguments of the generic term. This representative is used to define the general properties of the unit in the rubrics that describe them.

The grammar gives the context-free syntax of the considered structure in one of the following forms:

1. **Context-free syntax**

STRUCTURE ::=

DERIVATION1
| DERIVATION2
| ...

Terminal ::=

DERIVATION1
| DERIVATION2
| ...

terminal ::=

SET1
| SET2
| ...

DERIVATION1, DERIVATION2 are rewritings of the variable **STRUCTURE** (respectively, of the variable **Terminal**). SET1, SET2 are rewritings of the variable **terminal**; they are **Derivations** reduced to one single element (cf. below).

Each DERIVATION is a sequence of *element*s, each of them can be:

- a **set** of characters, written in this typography (lexical level only),
- a $\boxed{\textbf{terminal}}$ symbol (of the syntactic grammar) composed of letters, in this typography, for which only the lower case form is explicited in the grammar;
- a terminal $\boxed{\textbf{symbol}}$ (composed of other acceptable characters), in this typography,
- a **Terminal**, in this typography,
- a syntactic **STRUCTURE**, in this typography (syntactic level only),

- a non empty sequence of *elements* in their respective typography, with or without $comment$ $in\ this\ typography$, respectively in the following forms:
  - ***element*** { $\boxed{\textbf{symbol}}$ ***element*** }*
  - { ***element*** }$^+$
- an optional *element*, denoted [***element***],
- a difference of sets, denoted { ***element1*** $\setminus$ ***element2*** }, allowing to derive the texts of *element1* that are not texts of *element2*.

The syntactic structures may appear either in the plural, or in the singular, following the context. They may be completed by a *contextual information,* in this typography. Finally, several derivations may be placed on a same line.

2. **Profile**
   This item describes the sets of input and output signals of the expression. This description is done with the notations $?\,(E)$ that designates the list of input signals (or ports) of E, and $!\,(E)$ that designates the list of output signals (or ports) of E. The notation $?\,\{a_1, \ldots, a_n\}$ (respectively, $!\,\{a_1, \ldots, a_n\}$) designates explicitly the set of input ports (respectively, output ports) $a_1, \ldots, a_n$. Finally, the set operations $A \cap B$, $A \cup B$ and $A - B$ (the latter to designate the set of elements of A that are not in B).

3. **Types**
   This item describes the properties of the types of the arguments using equations on the types of value of the signals. The notation $\tau(E)$ is used to designate the type (domain of value) of the expression E. Given a process model with name P (cf. part E, section XI–1, page 173), the notations $\tau(?P)$ and $\tau(!P)$ are used to designate respectively the type of the tuple formed by the list of the inputs declared in the interface of the model, and the type of the tuple formed by the list of the outputs declared in this interface (cf. part E, section XI–5, page 179).

   (a) EQUATION

4. **Semantics**
   When the term cannot be redefined in the SIGNAL language, its semantics is given in the space of equations on sequences.

5. **Definition in SIGNAL**

   TERM(E_1, E_2, . . .)
   is a generic term of the SIGNAL language, to which is equal, by definition, the representative of the current unit.

6. **Clocks**
   This unit describes the synchronization properties of the arguments (values of Booleans and clocks) with a list of equations in the space of synchronization. The notation $\omega(E)$ is used to designate the clock of the expression E and the notation $\hbar$ to designate the clock of the constant expressions, or more generally, the clock of the context. An equation has generally the following form:

   (a) $\omega(E_1) = \omega(E_2)$

7. **Graph**
   This item defines the conditional dependencies between the arguments with a list of triples:

   (a) $E_1 \xrightarrow{E_3} E_2$

   The signal E_1 precedes the signal E_2 at the clock which is the product of the clock of E_1, the clock of E_2 and the clock representing the instants at which the Boolean signal E_3 has the value $true$: at this clock, E_2 cannot be produced before E_1.

8. **Properties**
   This item gives a list of properties of the construction (for example, associativity, distributivity, etc.).

   (a) PROPERTY

9. **Examples**

   (a) One or more `Examples` in the SIGNAL language illustrate the use of the unit.

Chapter II

Lexical units

The text of a program of the SIGNAL language is composed of words of the vocabulary built on a set of characters.

II–1 Characters

The characters used in the SIGNAL language are described in this section (**Character**). They can be designated by an encoding which is usable only in the comments, the character or string constants, and the directives, as precised in the syntax.

1. **Context-free syntax**

   Character ::= **character** | **CharacterCode**

II–1.1 Sets of characters

The set of characters (denoted **character**) used in the SIGNAL language contains the following subsets:

1. **Context-free syntax**

   character ::= **name-char** | **mark** | **delimitor** | **separator** | **other-character**

(i) The set **name-char** of characters used to build identifiers:

(a) **Context-free syntax**

name-char ::= **letter-char** | **numeral-char** | $\boxed{\_}$

letter-char ::=
upper-case-letter-char | **lower-case-letter-char** | **other-letter-char**

upper-case-letter-char ::=
$\boxed{\mathbf{A}}$ | $\boxed{\mathbf{B}}$ | $\boxed{\mathbf{C}}$ | $\boxed{\mathbf{D}}$ | $\boxed{\mathbf{E}}$ | $\boxed{\mathbf{F}}$ | $\boxed{\mathbf{G}}$ | $\boxed{\mathbf{H}}$ | $\boxed{\mathbf{I}}$
| $\boxed{\mathbf{J}}$ | $\boxed{\mathbf{K}}$ | $\boxed{\mathbf{L}}$ | $\boxed{\mathbf{M}}$ | $\boxed{\mathbf{N}}$ | $\boxed{\mathbf{O}}$ | $\boxed{\mathbf{P}}$ | $\boxed{\mathbf{Q}}$ | $\boxed{\mathbf{R}}$
| $\boxed{\mathbf{S}}$ | $\boxed{\mathbf{T}}$ | $\boxed{\mathbf{U}}$ | $\boxed{\mathbf{V}}$ | $\boxed{\mathbf{W}}$ | $\boxed{\mathbf{X}}$ | $\boxed{\mathbf{Y}}$ | $\boxed{\mathbf{Z}}$

lower-case-letter-char ::=

a | b | c | d | e | f | g | h | i
| j | k | l | m | n | o | p | q | r
| s | t | u | v | w | x | y | z

other-letter-char ::=

À | Á | Â | Ã | Ä | Å | Æ | Ç | È
| É | Ê | Ë | Ì | Í | Î | Ï | Ð | Ñ
| Ò | Ó | Ô | Õ | Ö | Ø | Ù | Ú | Û
| Ü | Ý | Þ | ß | à | á | â | ã | ä
| å | æ | ç | è | é | ê | ë | ì | í
| î | ï | ð | ñ | ò | ó | ô | õ | ö
| ø | ù | ú | û | ü | ý | þ | ÿ

numeral-char ::=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Excepted for the reserved words of the language (keywords), the upper case and lower case forms of a same letter (**letter-char**) are distinguished. The reserved words should appear totally in lower case or totally in upper case.

(ii) The set **mark** composed of the distinctive characters of the lexical units, and the set of characters used in operator symbols:

(a) **Context-free syntax**

mark ::= $\boxed{\textbf{.}}$ *separating character in real constants and distinctive character of matrix products*
| $\boxed{\textbf{'}}$ *start and end of character constants*
| $\boxed{\textbf{"}}$ *start and end of strings*
| $\boxed{\textbf{\%}}$ *start and end of comments*
| $\boxed{\textbf{:}}$ *character used in the definition symbol*
| $\boxed{=}$ *equality sign*
| $\boxed{<}$ *inferior sign*
| $\boxed{>}$ *superior sign and end of the dependency arrow*
| $\boxed{+}$ *positive and additive sign*
| $\boxed{-}$ *negative and subtractive sign, and dash of the dependency arrow*
| $\boxed{*}$ *product sign*
| $\boxed{/}$ *division sign, mark of difference, and sign of confining*
| $\boxed{\textbf{@}}$ *construction of complex*
| $\boxed{\textbf{\$}}$ *delay sign*
| $\boxed{\hat{}}$ *clock sign*
| $\boxed{\textbf{\#}}$ *exclusion sign*
| $\boxed{|}$ *composition symbol*

(iii) The delimitors are terminals of the syntactic level built with other characters than letters and numerals:

(a) **Context-free syntax**

delimitor ::= $\boxed{\textbf{(}}$ | $\boxed{\textbf{)}}$ *parenthesizing, tuple delimitors*
| $\boxed{\textbf{\{}}$ | $\boxed{\textbf{\}}}$ *parameter delimitors, dependencies parenthesizing*
| $\boxed{\textbf{[}}$ | $\boxed{\textbf{]}}$ *array delimitors*
| $\boxed{\textbf{?}}$ *input delimitor*
| $\boxed{\textbf{!}}$ *output delimitor*
| $\boxed{\textbf{,}}$ *separation of units*
| $\boxed{\textbf{;}}$ *end of units*

(iv) The **separator**s given here in their ASCII hexadecimal code (the space character and the **long-separator**s are distinguished) :

(a) **Context-free syntax**

separator ::= $\boxed{\backslash\textbf{x20}}$ ***space***
| **long-separator**

long-separator ::= $\boxed{\backslash\textbf{x9}}$ ***horizontal tabulation***
| $\boxed{\backslash\textbf{xA}}$ ***new line***
| $\boxed{\backslash\textbf{xC}}$ ***new page***
| $\boxed{\backslash\textbf{xD}}$ ***carriage return***

(v) The other *printable* characters, usable in the comments, the directives and the denotations of constants. This subset, **other-character**, is not defined by the manual.

II–1.2 Encodings of characters

All the characters (printable or not) can be designated by an encoded form (**CharacterCode**) in the comments, the character constants, the string constants and the directives. The authorized codes are those of the norm ANSI of the language C (possibly extended with codes for other characters), plus the escape character $\boxed{\backslash\%}$ used in the comments. An encoded character is either a special character (**escape-code**), or a character encoded in octal form (**OctalCode**), or a character encoded in hexadecimal form (**HexadecimalCode**). The numeric codes (**OctalCode** and **HexadecimalCode**) contain at most the number of digits necessary for the encoding of 256 characters; the manual does not define the use of unused codes.

1. **Context-free syntax**

CharacterCode ::= **OctalCode** | **HexadecimalCode**
| **escape-code**

OctalCode ::= $\boxed{\backslash}$ **octal-char [octal-char [octal-char]]**

octal-char ::= $\boxed{0}$ | $\boxed{1}$ | $\boxed{2}$ | $\boxed{3}$ | $\boxed{4}$ | $\boxed{5}$ | $\boxed{6}$ | $\boxed{7}$

HexadecimalCode ::= $\boxed{\backslash x}$ **hexadecimal-char [hexadecimal-char]**

hexadecimal-char ::= **numeral-char**
| $\boxed{A}$ | $\boxed{B}$ | $\boxed{C}$ | $\boxed{D}$ | $\boxed{E}$ | $\boxed{F}$
| $\boxed{a}$ | $\boxed{b}$ | $\boxed{c}$ | $\boxed{d}$ | $\boxed{e}$ | $\boxed{f}$

escape-code ::= $\boxed{\backslash a}$ *audible signal*
| $\boxed{\backslash b}$ *backspace*
| $\boxed{\backslash f}$ *form feed*
| $\boxed{\backslash n}$ *newline*
| $\boxed{\backslash r}$ *carriage return*
| $\boxed{\backslash t}$ *horizontal tab*
| $\boxed{\backslash v}$ *vertical tab*
| $\boxed{\backslash\backslash}$ *backslash*
| $\boxed{\backslash "}$ *double quote*
| $\boxed{\backslash '}$ *single quote*
| $\boxed{\backslash ?}$ *question mark*
| $\boxed{\backslash\%}$ *percent*

II–2 Vocabulary

A text of the SIGNAL language is a sequence of elements of the **Terminal** vocabulary (cf. section I–5, page 16) of the SIGNAL language. Between these elements, **separator**s can appear in any number (possibly zero). A **Terminal** of the SIGNAL language is the longest sequence of contiguous **terminals** and a **terminal** is the longest sequence of contiguous characters that can be formed by a left to right analysis respecting the rules described in this chapter. A terminal can contain a distinctive mark; the next mark is not a **character** (it is used as escape mark):

1. **Context-free syntax**

   prefix-mark ::= $\boxed{\backslash}$ *start of* **CharacterCode**

II–2.1 Names

A name allows to designate a directive, a signal (or a group of signals), a parameter, a constant, a type, a model or a module, in a context composed of a set of declarations. Two occurrences of a same name in distinct contexts can designate distinct objects.

A **Name** is a lexical unit formed by characters among the set composed of **letter-char**s plus the character $\boxed{\_}$ plus **numeral-char**s; a **Name** cannot start with a **numeral-char**. A **Name** cannot be a reserved word. All the characters of a **Name** are significant.

1. **Context-free syntax**

   Name ::= **begin-name-char [{ name-char }$^+$]**

   begin-name-char ::= **{ name-char** $\backslash$ **numeral-char }**

2. **Examples**

   (a) `a` and `A` are distinct **Name**s.

   (b) `X_25`, `The_password_12Xs3` are **Name**s.

In this document we will sometimes designate a **Name** from a particular category *X* by **Name-***X*.

II–2.2 Boolean constants

A Boolean constant is represented by $\boxed{\textbf{true}}$ or $\boxed{\textbf{false}}$ which are reserved words (hence they can also appear under their upper case forms, $\boxed{\textbf{TRUE}}$ and $\boxed{\textbf{FALSE}}$).

1. **Context-free syntax**

   Boolean-cst ::= $\boxed{\textbf{true}}$ | $\boxed{\textbf{false}}$

II–2.3 Integer Constants

An **Integer-cst** is a positive or zero integer in decimal representation composed of a sequence of numerals.

1. **Context-free syntax**

   Integer-cst ::= **{ numeral-char }**$^{+}$

II–2.4 Real constants

A **Real-cst** denotes the approximate value of a real number. There are two sets of reals: the simple precision reals and the double precision ones that contain the former. The **Real-cst**s are words of the lexical level so they cannot contain separators.

1. **Context-free syntax**

   Real-cst ::= **Simple-precision-real-cst**
   | **Double-precision-real-cst**

   Simple-precision-real-cst ::=
   Integer-cst Simple-precision-exponent
   | **Integer-cst** $\boxed{.}$ **Integer-cst [Simple-precision-exponent]**
   (a **Simple-precision-real-cst** may have an exponent)

   Double-precision-real-cst ::=
   Integer-cst Double-precision-exponent
   | **Integer-cst** $\boxed{.}$ **Integer-cst Double-precision-exponent**
   (a **Double-precision-real-cst** must have an exponent)

   Simple-precision-exponent ::= $\boxed{\mathbf{e}}$ **Relative-cst** | $\boxed{\mathbf{E}}$ **Relative-cst**

   Double-precision-exponent ::= $\boxed{\mathbf{d}}$ **Relative-cst** | $\boxed{\mathbf{D}}$ **Relative-cst**

   Relative-cst ::= **Integer-cst**
   | $\boxed{+}$ **Integer-cst**
   | $\boxed{-}$ **Integer-cst**

2. **Examples**

   (a) The notations contained in the following tables are simple precision representations respectively equivalent to the unit value and to the centesimal part of the unit.

	`1e0`	`1e+0`	`10e-1`
`1.0`	`0.1e1`	`0.1e+1`	`10.0e-1`

			`1e-2`
`0.01`	`0.001e1`	`0.001e+1`	`1.0e-2`

II–2.5 Character constants

A **Character-cst** is formed of a character or a code of character surrounded by two occurrences of the character $\boxed{\text{'}}$.

1. **Context-free syntax**

Character-cst ::= $\boxed{\text{'}}$ **Character-cstCharacter** $\boxed{\text{'}}$

Character-cstCharacter ::= **{ Character \ character-spec-char }**

character-spec-char ::= $\boxed{\text{'}}$
| **long-separator**

II–2.6 String constants

A **String-cst** value is composed of a list of sequences of characters surrounded by two occurrences of the character $\boxed{\text{"}}$.

1. **Context-free syntax**

String-cst ::= **{** $\boxed{\text{"}}$ **[{ String-cstCharacter }$^+$]** $\boxed{\text{"}}$ **}$^+$**

String-cstCharacter ::= **{ Character \ string-spec-char }**

string-spec-char ::= $\boxed{\text{"}}$
| **long-separator**

II–2.7 Comments

A comment may appear between any two lexical units and may replace a separator. It is composed of a seuqence of characters surrounded by two occurrences of the character $\boxed{\%}$.

1. **Context-free syntax**

Comment ::= $\boxed{\%}$ **[{ CommentCharacter }$^+$]** $\boxed{\%}$

CommentCharacter ::= **{ Character \ comment-spec-char }**

comment-spec-char ::= $\boxed{\%}$

II–3 Reserved words

A reserved word must be either totally in lower case or totally in upper case. In this manual, only the lower case form (in general) appears explicitly in the grammar rules. It can be replaced, for each reserved word, by the corresponding upper case form.

The reserved words used by the SIGNAL language are the following ones:

1. **Context-free syntax**

signalkw ::= $\boxed{\textbf{action}}$ | $\boxed{\textbf{after}}$ | $\boxed{\textbf{and}}$ | $\boxed{\textbf{array}}$
| $\boxed{\textbf{boolean}}$ | $\boxed{\textbf{bundle}}$
| $\boxed{\textbf{case}}$ | $\boxed{\textbf{cell}}$ | $\boxed{\textbf{char}}$ | $\boxed{\textbf{complex}}$ | $\boxed{\textbf{constant}}$ | $\boxed{\textbf{count}}$
| $\boxed{\textbf{dcomplex}}$ | $\boxed{\textbf{default}}$ | $\boxed{\textbf{defaultvalue}}$ | $\boxed{\textbf{deterministic}}$ | $\boxed{\textbf{dreal}}$
| $\boxed{\textbf{else}}$ | $\boxed{\textbf{end}}$ | $\boxed{\textbf{enum}}$ | $\boxed{\textbf{event}}$ | $\boxed{\textbf{external}}$
| $\boxed{\textbf{false}}$ | $\boxed{\textbf{from}}$ | $\boxed{\textbf{function}}$
| $\boxed{\textbf{if}}$ | $\boxed{\textbf{in}}$ | $\boxed{\textbf{init}}$ | $\boxed{\textbf{integer}}$ | $\boxed{\textbf{iterate}}$
| $\boxed{\textbf{label}}$ | $\boxed{\textbf{long}}$
| $\boxed{\textbf{module}}$ | $\boxed{\textbf{modulo}}$
| $\boxed{\textbf{next}}$ | $\boxed{\textbf{node}}$ | $\boxed{\textbf{not}}$
| $\boxed{\textbf{of}}$ | $\boxed{\textbf{operator}}$ | $\boxed{\textbf{or}}$
| $\boxed{\textbf{pragmas}}$ | $\boxed{\textbf{private}}$ | $\boxed{\textbf{process}}$
| $\boxed{\textbf{real}}$ | $\boxed{\textbf{ref}}$
| $\boxed{\textbf{safe}}$ | $\boxed{\textbf{short}}$ | $\boxed{\textbf{spec}}$ | $\boxed{\textbf{statevar}}$ | $\boxed{\textbf{step}}$ | $\boxed{\textbf{string}}$ | $\boxed{\textbf{struct}}$
| $\boxed{\textbf{then}}$ | $\boxed{\textbf{to}}$ | $\boxed{\textbf{tr}}$ | $\boxed{\textbf{true}}$ | $\boxed{\textbf{type}}$
| $\boxed{\textbf{unsafe}}$ | $\boxed{\textbf{use}}$
| $\boxed{\textbf{var}}$
| $\boxed{\textbf{when}}$ | $\boxed{\textbf{where}}$ | $\boxed{\textbf{window}}$ | $\boxed{\textbf{with}}$
| $\boxed{\textbf{xor}}$

($\boxed{\textbf{operator}}$ is currently hidden in the syntax of the language.)

Part B

THE KERNEL LANGUAGE

Chapter III

Model of sequences

III–1 Syntax

We consider:

- **A = {**a**,** a_1**, . . . ,** a_n**,** b**, . . . }**
  a denumerable set of typed variables (or *ports*);
- **F = {**f**,** f_1**, . . . ,** g**, . . . }**
  a finite set of symbols of typed functions;
- **T = {**`event`**,** `boolean`**, . . . ,** t**, . . . }**
  a finite set of basic types (sets of values);
- **TT =** $\bigcup_{n \in \mathbb{N}}$ **[**$0..n$**]** $\rightarrow \mathcal{TT}$
  the set of array types,
- **SS =** $\bigcup_{\mathbf{B} \in \mathbf{A}}$ **B** $\rightarrow \mathcal{TT}$
  the set of tuple types,
- $\mathcal{TT}$ **= T** $\cup$ **TT** $\cup$ **SS**
  the set of types.
- the symbols `default`, `when`, `$`.

We define the following sets of terms, defining the basic syntax of the SIGNAL language:

- **GD = {**$t\ a$**}**
  the set of *declarations* (association of a type with a variable);
- **GSS = {**a_{n+1} `:=:` f**(**a_1**, . . . ,** a_n**)}**
  the set of *static synchronous* generators (elementary processes), among them the set of generators on arrays and tuples are distinguished;
- **GDS = {**a_2 `:=:` a_1 `$ init` a_0**}**
  where a_0 is a constant with same domain as a_1, the set of *dynamic synchronous* generators (elementary processes);

- **GE = {**a_3 `:=:` a_1 `when` a_2**}** the set of *extraction* generators (elementary processes);
- **GM = {**a_3 `:=:` a_1 `default` a_2**}**
  the set of *merge* generators (elementary processes);
- recursively the set **PROC** of syntactic processes as the least set containing:
  - **G = GD** ∪ **GSS** ∪ **GDS** ∪ **GE** ∪ **GM**
    the set of generators,
  - **PC = {**`P1 | P2` where `P1` and `P2` belong to **PROC}**
    (composition process),
  - **PR = {**`P1 /` a (denoted also `P1 where` a) where `P1` belongs to **PROC** and a belongs to **A}**
    (restriction process).

III–2 Events

Let $\mathbb{D}$ be the set of values that can be taken by the variables, an *event* is an occurrence of the *simultaneous* valuation of distinct variables *(synchronous commmunication).* The values respect the properties resulting from the interpretation of the terms which are used. In $\mathbb{D}$, the set of Boolean values, $\mathbb{B}$=$\{true, false\}$, is distinguished.

For a variable $a_i \in A$, and a subset A_j of variables in A, we consider:

$\mathbb{D}_{a_i}$ the domain of values (Booleans, integers, reals...) that may be taken by a_i.

$$\mathbb{D}_{A_j} = \bigcup_{a_i \in A_j} \mathbb{D}_{a_i}$$

$$\mathbb{D}_A = \mathbb{D}$$

The symbol $\perp$ ($\perp \notin \mathbb{D}$) is introduced to designate the absence of valuation of a variable. Then we denote:

$$\mathbb{D}^{\perp} = \mathbb{D} \cup \{\perp\}$$
$$\mathbb{D}^{\perp}_{A_i} = \mathbb{D}_{A_i} \cup \{\perp\}$$

Considering A_1 a non empty subset of A, we call *event* on A_1 any application

$$e : A_1 \ \rightarrow \ \mathbb{D}^{\perp}_{A_1}$$

- $e(a) = \perp$ indicates that a has no value for the event m.
- $e(a) = v$ indicates, for $v \in \mathbb{D}_a$, that a takes the value v for the event e.
- $e(A_1) = \{x/a \in A_1, e(a) = x\}$

The set of *events* on A_1 ($A_1 \ \rightarrow \ \mathbb{D}^{\perp}_{A_1}$) is denoted $\mathcal{E}^*_{A_1}$.

By convention, 1_e is the single event defined on the empty set of ports $\emptyset$ (it is called *unit event*).

The *absent event* on A_1 ($A_1 \ \rightarrow \ \{\perp\}$) is denoted $\perp_e(A_1)$.

The set

$$\mathcal{E}^*_{\subseteq A_1} = \bigcup_{A_i \subseteq A_1} \mathcal{E}^*_{A_i}$$

is the set of all events on the subsets of A_1.

It is defined a special event on A, denoted $\sharp$, which is called *blocking event* (or impossible event).

The following notations are used:

$$\mathcal{E}_{A_1} = \mathcal{E}^*_{A_1} \cup \{\sharp\}$$
$$\mathcal{E}_{\subseteq A_1} = \mathcal{E}^*_{\subseteq A_1} \cup \{\sharp\}$$

Partial observation of an event

Let $A_1 \subseteq A$ and $A_2 \subseteq A$ two subsets of A and $e \in \mathcal{E}_{A_1}$ some event on A_1.
The restriction of e on A_2, or partial observation of e on A_2, is denoted $e_{|A_2}$:

$$e_{|A_2} \in \mathcal{E}_{A_1 \cap A_2}$$

It is defined as follows:

- $((A_1 \cap A_2 \neq \emptyset) \bigwedge (e \neq \sharp)) \quad \Rightarrow \quad ((\forall a \in A_1 \cap A_2) \quad ((e_{|A_2})(a) = e(a)))$
- $((A_1 \cap A_2 \neq \emptyset) \bigwedge (e = \sharp)) \quad \Rightarrow \quad (e_{|A_2} = \sharp)$
- $(A_1 \cap A_2 = \emptyset) \quad \Rightarrow \quad (e_{|A_2} = e_{|\emptyset} = 1_e)$

Product of events

Let $e_1 \in \mathcal{E}_{A_1}$ and $e_2 \in \mathcal{E}_{A_2}$ two events.
Their product is denoted $e_1 {\cdot} e_2$:

$$e = e_1 {\cdot} e_2 \in \mathcal{E}_{A_1 \cup A_2}$$

It is defined as follows:

- $(e = \sharp) \quad \Leftrightarrow \quad (((e_1 = \sharp) \bigvee (e_2 = \sharp)) \bigvee (e_{1|A_1 \cap A_2} \neq e_{2|A_1 \cap A_2}))$
- $(e \neq \sharp) \quad \Rightarrow \quad ((e_{|A_1} = e_1) \bigwedge (e_{|A_2} = e_2))$

Corollary 1 *($\mathcal{E}_{\subseteq A_1}$,$\cdot$,$1_e$) is a commutative monoid.*
The product operator $\cdot$ is idempotent and $\sharp$ is an absorbent (nilpotent) element.

III–3 Traces

A *trace* is a sequence of events (sequence of observations) without the blocking event.

For any subset A_1 of A, we consider the following definition of the set $\mathcal{T}_{A_1}$ of traces on A_1.

III–3.1 Definition

$\mathcal{T}^*_{A_1}$ is the set of non empty sequences of events on A_1, composed of:

- finite sequences: they are the set of applications $\mathbb{N}_{<k} \rightarrow \mathcal{E}^*_{A_1}$ where $\mathbb{N}_{<k}$ represents the set of finite initial segments of $\mathbb{N}$ (set of natural integers),

- infinite sequences: they are the set of applications $\mathbb{N} \rightarrow \mathcal{E}^*_{A_1}$.

The set

$$\mathcal{T}^*_{\subseteq A_1} = \bigcup_{A_i \subseteq A_1} \mathcal{T}^*_{A_i}$$

is the set of all non empty sequences of events on the subsets of A_1.

The empty sequence of events is denoted 0_T.

A trace on A_1 is either a sequence of $\mathcal{T}^*_{A_1}$ or the empty sequence. The set of traces on A_1 is:

$$\mathcal{T}_{A_1} = \mathcal{T}^*_{A_1} \cup \{0_T\}$$

The set of traces on subsets of A_1 is:

$$\mathcal{T}_{\subseteq A_1} = \mathcal{T}^*_{\subseteq A_1} \cup \{0_T\}$$

The set of traces defined on A, denoted $\mathcal{T}$, is the union of the sets $\mathcal{T}_{A_1}$ for all subsets $A1$ of A.

The single infinite sequence defined on $\mathcal{T}^*_{\emptyset}$ is denoted 1_T and is called *unit trace*. It is equal to the infinite repetition $(1_e)^\omega$ of the unit event 1_e.

The *absent trace* on A_1 ($\mathbb{N} \rightarrow \{\perp_e(A_1)\}$: the infinite sequence formed by the infinite repetition of $\perp_e(A_1)$) is denoted $\perp_{A_1}$.

Notations

The smallest set of variables of A on which a given trace T is defined (definition domain of the events composing T) is referred to as *var*(T). By convention, *var*$(0_T) = A$.

For a trace T and t an integer, we will note frequently T_t the event $T(t)$ of T at the instant t, and we will note sometimes a_t the value of a variable a for this event.

III–3.2 Partial observation of a trace

Let $A_1 \subseteq A$ and $A_2 \subseteq A$ two subsets of A and $T \in \mathcal{T}_{A_1}$ some trace on A_1.

The restriction of T on A_2, or partial observation of T on A_2, is denoted $T_{\|A_2}$.

If $A_1 \cap A_2 \neq \emptyset$, $T_{\|A_2}$ is the trace T_2 such that:

$$\begin{cases} dom(T_2) = dom(T) \\ \forall t \in dom(T) \quad T_2(t) = T(t)_{|A_2} \end{cases}$$

If $A_1 \cap A_2 = \emptyset$, $T_{\|A_2} = T_{\|\emptyset} = 1_T$.

If $A_2 \neq \emptyset$, $0_{T\|A_2} = 0_T$.

III–3.3 Prefix order on traces

The following relation is defined on traces:

$T_1 \angle T_2$ if and only if:

$$\begin{cases} dom(T_1) \subseteq dom(T_2) \\ (\forall t) \quad (\,(t \in dom(T_1)) \quad \Rightarrow \quad (T_1(t) = T_2(t))\,) \end{cases}$$

It is said that T_1 is a prefix of T_2.

Corollary 2

- *$\angle$ is an order relation on $\mathcal{T}$, 0_T is the minimum for this order.*
- *The set of prefixes of a trace is a chain.*
- *Any subset of prefixes of a trace has an upper bound.*

The notation $T_{\leq t}$ represents the prefix of a trace T such that $t \in dom(T_{\leq t})$ and $t+1 \notin dom(T_{\leq t})$.

III–3.4 Product of traces

The product $T = T_1{\cdot}T_2$ of two traces T_1 and T_2 defined respectively on A_1 and A_2 is the greatest trace for the order relation $\angle$ such that:

$$(T_{\|A_1} \angle T_1) \bigwedge (T_{\|A_2} \angle T_2)$$

(it is defined on $A_1 \cup A_2$ and is obtained by termwise products of respective events).

Corollary 3 *($\mathcal{T}_{\subseteq A_1}$,$\cdot$,$1_T$) is a commutative monoid.*
The product operator $\cdot$ is idempotent and 0_T is an absorbent (nilpotent) element.

III–3.5 Reduced trace

A trace T_1 is said to be a *sub-trace* of a non empty trace T_2 if and only if there exists an infinite sequence f_1, strictly increasing (i.e., injective and increasing) on $\mathbb{N}$ (such a sequence is called *expansion function* on T_1), such that:

$$T_2 \circ f_{1|dom(T_1)} = T_1$$

(the notation $f_{|X}$ designates the restriction of a given function f on the domain X).

Remarks

- 0_T is a sub-trace of any trace;
- any prefix T_1 of T_2 is a sub-trace of T_2.

Corollary 4 *The sub-trace relation is a preorder (reflexive and transitive).*

The sub-trace relation is not antisymmetric, as shown by the following sequences: $(\alpha\beta)^\omega$ and $(\beta\alpha)^\omega$ (with $f_1(n) = n+1$).

Definition A trace T_1 (defined on A_1) is said to be a *reduced trace* of a non empty trace T_2 (defined on A_2) if and only if T_1 is a sub-trace of T_2 and:

- ($dom(T_1)$ is finite) $\quad\Rightarrow\quad$ ($dom(T_2)$ is finite)
- for any expansion function f_1 on T_1 such that $T_2 \circ f_{1|dom(T_1)} = T_1$, then:

$$(\forall t \in (dom(T_2)) \setminus f_1(dom(T_1))) \quad (\ T_2(t) = \bot_e(A_2)\ )$$

Proposition The relation *"is a reduced trace of"* is an order relation.
"T_1 is a reduced trace of T_2" is denoted:

$$T_1 \subseteq_{\downarrow} T_2$$

Proof of antisymmetry: $T_1 \subseteq_{\downarrow} T_2$ and $T_2 \subseteq_{\downarrow} T_1$
$dom(T_2) = dom(T_1)$
If $dom(T_1)$ is finite then the single possible expansion function on T_1 is the identity.
For any trace T, T is a prefix of T_1 if and only if it is a prefix of T_2 is proved by recurrence on the length of T.
Then the existence of an upper bound to any subset of prefixes of a trace proves the equality. □

For a given expansion function f and a trace T_1, there exists a least trace (for the prefix order $\angle$), T_2, such that $T_1 \subseteq_{\downarrow} T_2$.
We denote by $\uparrow$ the function that, to an expansion function f and a trace T, associates this least trace $f \uparrow T$ (example on figure B–III.1).
Then we have, by definition:

$$T \subseteq_{\downarrow} f \uparrow T$$

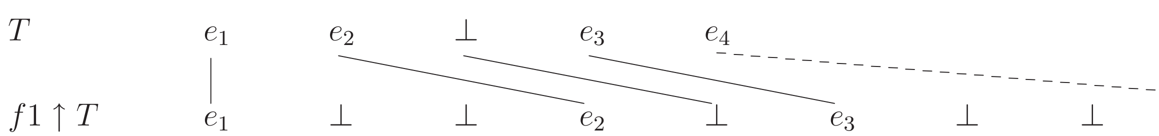


Figure B–III.1: $f1 \uparrow T$ with $f1(0) = 0$, $f1(1) = 3$, $f1(2) = 4$, $f1(3) = 5\ldots$

Property:

$$f_2 \uparrow (f_1 \uparrow T) = (f_2 \circ f_1) \uparrow T$$

For any f, we have also $f \uparrow 0_T = 0_T$.
By convention: $f \uparrow 1_T = 1_T$.

III–4 Flows

Definition A *flow* is a trace which is minimal for the relation $\subseteq_{\downarrow}$.

Comment: A flow F on A_1 is a trace that does not contain the absent event on A_1 between two events which have valued variables.

Corollary 5

- *(F is a flow and $F_1 \angle F$) $\Rightarrow$ (F_1 is a flow);*

- *0_T is a flow;*
- *1_T is a flow;*
- *if F is a finite flow on A_1, then $(F \perp_e (A_1)^\omega)$ is a flow;*
- *$\perp_{A_1}$ is a flow.*

III–4.1 Equivalence of traces

Definition Two traces T_1 and T_2 are said to be equivalent modulo $\perp$ (this is denoted: $T_1 \equiv_\downarrow T_2$) if and only if there exists some trace T such that $T \subseteq_\downarrow T_1$ and $T \subseteq_\downarrow T_2$.

This relation is indeed an equivalence relation.

Property For any trace T, the equivalence class of T modulo $\perp$ is a lattice.

Proof

- By definition, every pair T_1, T_2 in an equivalence class has a lower bound.
- Every pair T_1, T_2 in an equivalence class has an upper bound:
  Let f_1, f_2 such that:

  $$T_1 \circ f_1 = \min(T_1, T_2)$$
  $$T_2 \circ f_2 = \min(T_1, T_2)$$

  The upper bound is the trace

  $$\max(T_1, T_2) = f_1' \uparrow T_1 = f_2' \uparrow T_2$$

  with f_1', f_2' defined as follows:

  $\forall t$, if $\exists s, f_1(s) = t$ then $f_1'(s) = \max(t, f_2(s))$,
  if $s \notin f_1(dom(\min(T_1, T_2)))$ then if $s = 0$ then $f_1'(s) = 0$ else $f_1'(s) = f_1'(s-1) + 1$

  (f_2' is defined symmetrically).
  Then

  $$(f_1' \circ f_1) \uparrow \min(T_1, T_2) = (f_2' \circ f_2) \uparrow \min(T_1, T_2) = \max(T_1, T_2)$$

□

Each equivalence class has a flow as lower bound. For a trace T, this flow is denoted $T_\downarrow$.

Notation The set of flows on A_1 is denoted $\mathcal{S}_{A_1}$.

III–4.2 Partial flow

Let $A_1 \subseteq A$ and $A_2 \subseteq A$ two subsets of A and $F \in \mathcal{S}_{A_1}$ some flow on A_1.

The *projection* of F on A_2, denoted $\Pi_{A_2}(F)$, is defined by:

$$\Pi_{A_2}(F) = (F_{\|A_2})_{\downarrow}$$

The following equalities hold:

- $\forall F,\ \Pi_{\emptyset}(F) = 1_T$
- $\Pi_{A_2}(0_T) = 0_T$
- $\Pi_{A_2}(\perp_{A_1}) = \perp_{A_1 \cap A_2}$

III–4.3 Flow-equivalence

Equivalence modulo $\perp$ is an equivalence relation that preserves the simultaneousness of valuations within an event and the ordering of events within a trace: traces which are equivalent modulo $\perp$ possess the same synchronization relations.

A weaker relation is introduced, which is called flow-equivalence. It allows to compare traces with respect to the sequences of values that variables hold.

Definition A trace T' defined on A_1 is a *relaxation* of a trace T defined on the same set of variables A_1 if and only if for all $a \in A_1$, $T_{\|\{a\}} \subseteq_{\downarrow} {T'}_{\|\{a\}}$. This is denoted: $T \sqsubseteq T'$.

Corollary The relaxation relation $\sqsubseteq$ is an order relation.

Definition Two traces T_1 and T_2 are said to be *flow-equivalent* (this is denoted: $T_1 \approx T_2$) if and only if there exists some trace T such that $T \sqsubseteq T_1$ and $T \sqsubseteq T_2$.

The class of flow-equivalence of a trace T is a semi-lattice. It admits a lower bound which is a flow, written $T_{\approx}$.

III–5 Processes

III–5.1 Definition

A *process* on $A_1 \subseteq A$ is a set of flows on A_1 which are non comparable by the prefix relation.

Example Let us represent a flow by the sequence of its events, where an event is represented by the variables which are valued for it (successive events are separated by the sign ";").

Consider the following flows defined on variables a, b:

$F_1:\ a; ab; b$
$F_2:\ a; ab; ab$
$F_3:\ a; ab; b; b$

The flows F_1 and F_2 (respectively, F_2 and F_3) can belong to a same process. However, F_1 and F_3 cannot belong to a same process since they are comparable.

The set of processes on A_1 is denoted $\mathcal{P}_{A_1}$. It is a subset of $\mathcal{P}(\mathcal{S}_{A_1})$, the set of subsets of $\mathcal{S}_{A_1}$.
The set

$$\mathcal{P}_{\subseteq A_1} = \bigcup_{A_i \subseteq A_1} \mathcal{P}_{A_i}$$

is the set of processes on the subsets of A_1.

The process $\mathbf{1}_{\mathcal{P}} = \{1_T\}$, defined on the empty set of ports $\emptyset$, and with the unit trace as single element, is called *unit process.*

The process on A_1 defined by the empty set of flows is denoted $\mathbf{0}_{\mathcal{P}}(A_1)$.

Notation

The notation *var*(P) is used to designate the smallest set of variables of A on which the process P is defined.

III–5.2 Partial observation of a process

Let $A_1 \subseteq A$ and $A_2 \subseteq A$ two subsets of A and P a process on A_1.

The *projection* of P on A_2, denoted $\Pi_{A_2}(P)$, is defined by:

$$\Pi_{A_2}(P) = \{\Pi_{A_2}(F) \ / \ F \in P \text{ and } \Pi_{A_2}(F) \text{ is maximal for } \angle\}$$

III–5.3 Composition of processes

Let P_1 and P_2 two processes defined respectively on A_1 and A_2.

The *composition* (or *synchronous composition* of P_1 and P_2, denoted $P_1|P_2$, is a process on $A_1 \cup A_2$ defined by:

$$\begin{aligned} P_1|P_2 = \{F \in \mathcal{S}_{A_1 \cup A_2} \ / \quad & (\ (\exists F_1 \in P_1) \quad (\ \Pi_{A_1}(F) \angle F_1 \ ) \ ) \\ & \bigwedge (\ (\exists F_2 \in P_2) \quad (\ \Pi_{A_2}(F) \angle F_2 \ ) \ ) \\ & \bigwedge (F \text{ is maximal for } \angle)\} \end{aligned}$$

Corollary 6 *($\mathcal{P}_{\subseteq A_1}$,$|$,$\mathbf{1}_{\mathcal{P}}$) is a commutative monoid.*
The composition operator $|$ is idempotent and $\mathbf{0}_{\mathcal{P}}(A_1)$ is an absorbent (nilpotent) element.

III–5.4 Order on processes

The following relation is defined on processes:

$P_1 \angle P_2$ if and only if:

$$(\forall F_1 \in P_1) \quad (\ (\exists F_2 \in P_2) \quad (\ F_1 \angle F_2 \ ) \ )$$

This relation is an order relation.

Proof of antisymmetry:
$(P_1 \angle P_2) \quad \Rightarrow \quad (\ (\forall F_1 \in P_1) \quad (\ (\exists F_2 \in P_2) \quad (\ F_1 \angle F_2\ )\ )\ )$
$(P_2 \angle P_1) \quad \Rightarrow \quad (\ (\exists F_3 \in P_1) \quad (\ F_2 \angle F_3\ )\ )$
Then $F_1 = F_3$ since flows in a process are not comparable by $\angle$.
Then $F_1 = F_2$. Thus $P_1 = P_2$. □

Corollary 7

- $\Pi_{A_2}(\mathbf{0}_{\mathcal{P}}(A_1)) = \mathbf{0}_{\mathcal{P}}(A_1 \cap A_2)$
- $\Pi_{var(P)}(P) = P$
- $\Pi_{A_1 \cap A_2}(P) = (\Pi_{A_1} \circ \Pi_{A_2})(P)$
- $\Pi_{A_1 \cup A_2}(P) \angle \Pi_{A_1}(P) | \Pi_{A_2}(P)$
- $\Pi_{var(P_1)}(P_1|P_2) \angle P_1$
- Π *is monotonic:* $(P_1 \angle P_2) \quad \Rightarrow \quad (\Pi_B(P_1) \angle \Pi_B(P_2))$
- $|$ *is monotonic:* $(P_1 \angle P_2) \quad \Rightarrow \quad (Q|P_1 \angle Q|P_2)$
- $\Pi_B(P_1|P_2) \angle \Pi_B(P_1)|\Pi_B(P_2)$

Proposition Let P_1 and P_2 two processes defined respectively on A_1 and A_2.

$$(P_1 = \Pi_{A_1}(P_1|P_2)) \quad \Leftrightarrow \quad (\Pi_{A_1 \cap A_2}(P_1) \angle \Pi_{A_1 \cap A_2}(P_2))$$

Sketch of the proof:
Since $\Pi_{A_1}(P_1|P_2) \angle P_1$ it is sufficient to prove that

$$(P_1 \angle \Pi_{A_1}(P_1|P_2)) \quad \Leftrightarrow \quad (\Pi_{A_1 \cap A_2}(P_1) \angle \Pi_{A_1 \cap A_2}(P_2))$$

$\Rightarrow$) Assume that $P_1 \angle \Pi_{A_1}(P_1|P_2)$.
Let $F \in \Pi_{A_1 \cap A_2}(P_1)$
$(\exists F_1 \in P_1) \quad (\ F = \Pi_{A_1 \cap A_2}(F_1)\ )$
Since $F_1 \in P_1$, by hypothesis, $(\exists F' \in \Pi_{A_1}(P_1|P_2)) \quad (\ F_1 \angle F'\ )$
Thus $(\exists F'' \in P_1|P_2) \quad (\ F_1 \angle \Pi_{A_1}(F'')\ )$
By definition of the composition, $(\exists F_2'' \in P_2) \quad (\ \Pi_{A_2}(F'') \angle F_2''\ )$
Let $F_2''' = \Pi_{A_1 \cap A_2}(F_2'')$
Then $F \angle F_2'''$

$\Leftarrow$) Assume that $\Pi_{A_1 \cap A_2}(P_1) \angle \Pi_{A_1 \cap A_2}(P_2)$.
If $F_1 \in P_1$, then $(\exists F_2 \in \Pi_{A_1 \cap A_2}(P_2)) \quad (\ \Pi_{A_1 \cap A_2}(F_1) \angle F_2\ )$
Thus $(\exists F_2' \in P_2) \quad (\ \Pi_{A_1 \cap A_2}(F_1) \angle \Pi_{A_1 \cap A_2}(F_2')\ )$
Thus $(\exists F \in P_1|P_2) \quad (\ F_1 \angle \Pi_{A_1}(F)\ )$ □

Consequences

- if $A_1 \cap A_2 = \emptyset$: $P_1 = \Pi_{A_1}(P_1|P_2)$ and $P_2 = \Pi_{A_2}(P_1|P_2)$
- if $A_1 \subseteq A_2$: $(P_1 = \Pi_{A_1}(P_1|P_2)) \quad \Leftrightarrow \quad (P_1 \angle \Pi_{A_1}(P_2))$
- if $A_2 \subseteq A_1$: $(P_1 = P_1|P_2) \quad \Leftrightarrow \quad (\Pi_{A_2}(P_1) \angle P_2)$
- if $A_1 = A_2$: $(P_1 = P_1|P_2) \quad \Leftrightarrow \quad (P_1 \angle P_2)$

As an application, if P_2 represents a safety property defined on the same set of variables as P_1, P_1 satisfies the property P_2, which means that any flow of P_1 is a flow of P_2 (P_2 is less constrained than P_1), if and only if $P_1 = P_1|P_2$.

Note that there is the same result when P_2 is defined on a subset of the variables of P_1.

More generally, if $A_2 \subseteq A_1$, $P_1 = P_1|P_2$ means that P_2 is an *abstraction* of P_1.

III–6 Semantics of basic SIGNAL terms

The semantics of each primitive operator is defined by a set of flows: a SIGNAL *process* on $A_1 \subseteq A$ is a non empty set of flows on A_1 (i.e., a subset of $\mathcal{S}_{A_1}$) defined, from primitive operators and composition, by *constraints* (relations) on the flows.

In the following, we denote generically $\mathbf{P} : \mathcal{P}_{A_1}$ a process on A_1, to define the semantics of the corresponding term. In addition, we denote *var*$(x_1, \ldots, x_n)$ the set of the x_i variables ($i = 1, \ldots, n$) which are distinct.

III–6.1 Declarations

Let μ designate a type whose domain of values is $\tau(\mu)$.

The term

$$\mu \ \mathtt{X}$$

defines a process $\mathbf{P} : \mathcal{P}_{\{X\}}$ representing all the possible sequences of values of the signal `X`.

$$\mathbf{P} =_\Delta \{ \quad T \in \mathcal{S}_{\{X\}} \; / \; (\forall t) \quad (\, (T_t(X) \neq \perp) \quad \Rightarrow \quad (T_t(X) \in \tau(\mu)) \,) \quad \}$$

III–6.2 Monochronous processes

A process P defined on A_1 is said *monochronous* if, at each instant t for which one of the signals is present (respectively, absent), all of them are also present (respectively, absent). Flows defining monochronous processes are called also monochronous flows.

$$(\forall T \in P) \quad (\quad (\forall t) \quad (\, (\, (\exists X \in A_1) \quad (\, T_t(X) = \perp \,) \,) \quad \Rightarrow \quad (\, (\forall Y \in A_1) \quad (\, T_t(Y) = \perp \,) \,) \,) \quad)$$

2-a Static monochronous processes

Let F be an operator. Under some interpretation I for which the interpretation of F is denoted $[|F|]_I$, the term

$\mathtt{X_{n+1}}$ `:=: F(` $\mathtt{X_1}$`,...,`$\mathtt{X_n}$ `)`

defines a process $\mathbf{P} : \mathcal{P}_{\mathit{var}(X_1,\ldots,X_n,X_{n+1})}$ by some relation between the sequence of values of the signal $\mathtt{X_{n+1}}$ and the sequence obtained by the pointwise extension of the application of F, under this interpretation, to the sequence of tuples of values of the signals $\mathtt{X_1},\ldots,\mathtt{X_n}$ (note that the sign "`:=:`" makes explicit the fact that this term represents a non oriented equation).

$$\begin{array}{ll} \mathbf{P} =_\Delta \{ & T \in \mathcal{S}_{\mathit{var}(X_1,\ldots,X_n,X_{n+1})} \ / \\ & T \text{ is monochronous and} \\ & (\forall t) \quad (\ (T_t(X_{n+1}) \neq \perp) \quad \Rightarrow \quad (T_t(X_{n+1}) = [|F|]_I(T_t(X_1),\ldots,T_t(X_n)))\ ) \quad \} \end{array}$$

2-b Dynamic monochronous processes: the delay

The term

$\mathtt{X_2}$ `:=:` $\mathtt{X_1}$ `$ init` $\mathtt{V_0}$

defines a process $\mathbf{P} : \mathcal{P}_{\mathit{var}(X_1,X_2)}$ by the relation constraining the equality of the sequence of values of the signal $\mathtt{X_2}$ and the sequence of values of the signal $\mathtt{X_1}$, delayed by 1; $\mathtt{V_0}$ is the initial value of $\mathtt{X_2}$.

$$\begin{array}{ll} \mathbf{P} =_\Delta \{ & T \in \mathcal{S}_{\mathit{var}(X_1,X_2)} \ / \\ & T \text{ is monochronous} \\ & \text{and } (\forall t > 0) \quad (\ (T_t(X_2) \neq \perp) \quad \Rightarrow \quad (T_t(X_2) = T_{t-1}(X_1))\ ) \\ & \text{and } (T_0(X_1) \neq \perp) \quad \Rightarrow \quad (T_0(X_2) = V_0) \quad \} \end{array}$$

III–6.3 Polychronous processes

A process defined on A_1 is said *polychronous* if it contains a flow T for which there exists some instant t in which one of the signals is present while another one is not. By extension, a term is said polychronous if it allows to define polychronous processes.

3-a Sub-signals

The term

$\mathtt{X_3}$ `:=:` $\mathtt{X_1}$ `when` $\mathtt{X_2}$

defines a process $\mathbf{P} : \mathcal{P}_{\mathit{var}(X_1,X_2,X_3)}$ by the relation constraining the equality of the sequence of values of the signal $\mathtt{X_3}$ and the sequence of occurrences of value of the signal $\mathtt{X_1}$ when the Boolean signal $\mathtt{X_2}$ carries the value $true$.

$$\begin{array}{ll} \mathbf{P} =_\Delta \{ & T \in \mathcal{S}_{\mathit{var}(X_1,X_2,X_3)} \ / \ (\forall t) \quad (\\ & \quad ((T_t(X_2) = true) \quad \Rightarrow \quad (T_t(X_3) = T_t(X_1))) \\ & \bigwedge \quad ((T_t(X_2) \neq true) \quad \Rightarrow \quad (T_t(X_3) = \perp))\ ) \quad \} \end{array}$$

3-b Merging of signals

The term

`X`$_3$ `:=: X`$_1$ `default X`$_2$

defines a process $\mathbf{P} : \mathcal{P}_{var(X_1, X_2, X_3)}$ by the relation constraining the equality of the sequence of values of the signal `X`$_3$ and the sequence formed by the occurrences of value of the signal `X`$_1$ or by default the occurrences of value of the signal `X`$_2$.

$$\begin{array}{ll} \mathbf{P} =_{\Delta} \{ & T \in \mathcal{S}_{var(X_1, X_2, X_3)} \ / \ (\forall t) \quad (\\ & \quad ((T_t(X_1) \neq \perp) \quad \Rightarrow \quad (T_t(X_3) = T_t(X_1))) \\ & \bigwedge \quad ((T_t(X_1) = \perp) \quad \Rightarrow \quad (T_t(X_3) = T_t(X_2)))\ ) \quad \} \end{array}$$

III–6.4 Composition of processes

The term

`P`$_1$ `| P`$_2$

where `P`$_1$ and `P`$_2$ define respectively processes $\mathbf{P_1}$ and $\mathbf{P_2}$ on the sets of variables A_1 and A_2, defines a process $\mathbf{P} : \mathcal{P}_{A_1 \cup A_2}$ by the greatest relation constraining their common signals to respect the constraints imposed respectively by `P`$_1$ and `P`$_2$ (see an example on the figure B–III.2).

$$\mathbf{P} =_{\Delta} \mathbf{P_1} | \mathbf{P_2}$$

III–6.5 Restriction

The term

`P`$_1$ `/ a`

(or `P`$_1$ `where a`)
where `P`$_1$ defines a process $\mathbf{P_1}$ on the set of variables A_1, defines a process $\mathbf{P} : \mathcal{P}_{A_1 \setminus \{a\}}$ by the projection of $\mathbf{P_1}$ on the subset of ports of `P`$_1$ which are different from `a`.

$$\mathbf{P} =_{\Delta} \Pi_{A_1 \ \setminus \ \{a\}}(\mathbf{P_1})$$

III–7 Composite signals

The types of the Signal language contain elementary types such as Booleans, integers, etc., but also structured types allowing to declare composite objects. Structured types are tuple types and array types.

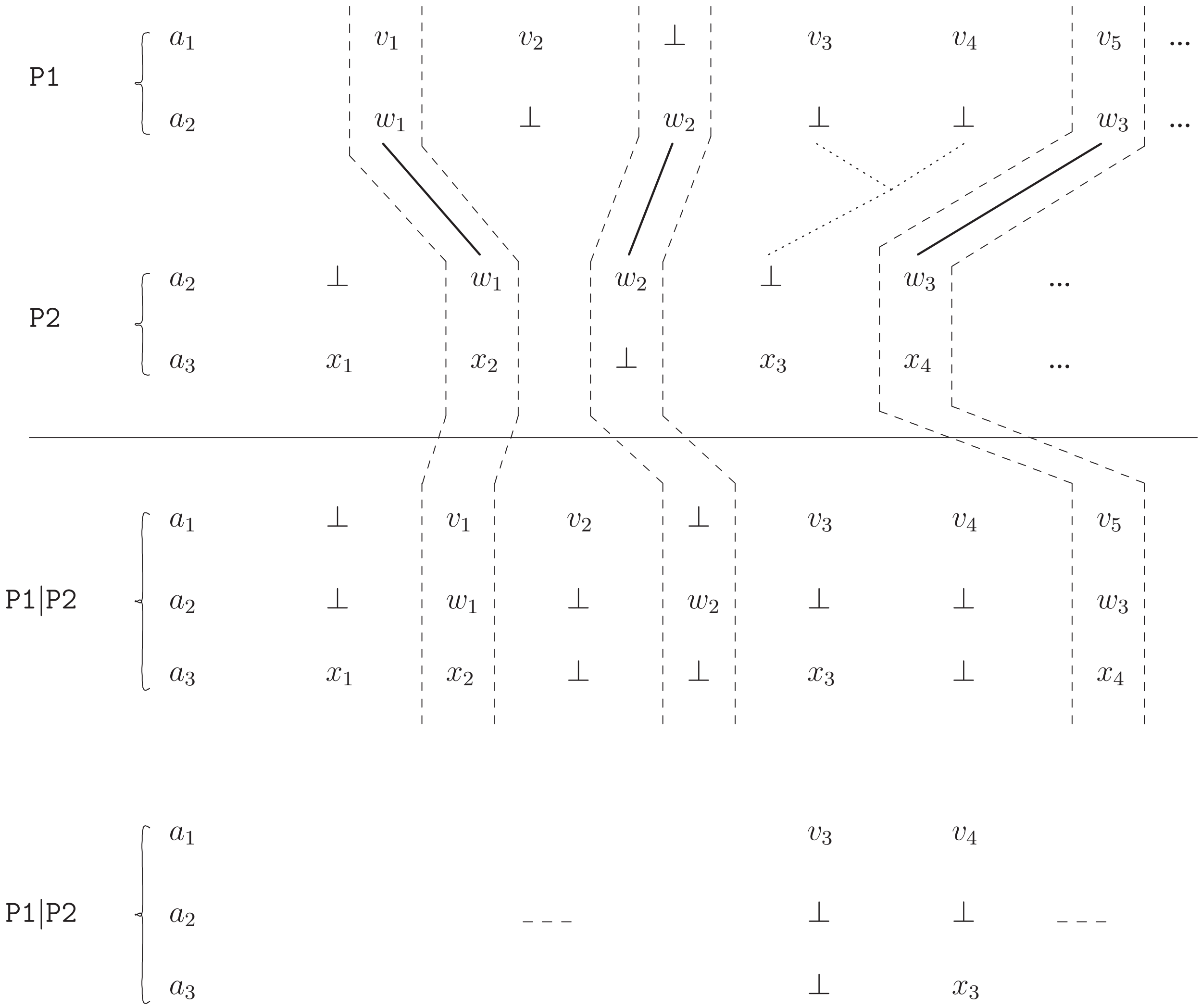


Figure B–III.2: Two flows of the composition of `P1` and `P2`

III–7.1 Tuples

Construction of tuple

If $\mathtt{E}_1, \ldots, \mathtt{E}_\mathtt{m}$ designate m signals of respective types $\mu_1, \ldots, \mu_m$, the term

$(\mathtt{E}_1,\ldots,\mathtt{E}_\mathtt{m})$

defines a tuple of signals, of type $(\mu_1 \times \ldots \times \mu_m)$ (where $\times$ designates the product of domains), such that

$$(\forall t) \quad (\, (\mathtt{E}_1, \ldots, \mathtt{E}_\mathtt{m})_t = (E_{1t}, \ldots, E_{mt}) \,)$$

Tuple types

Let m types $\mu_1, \ldots, \mu_m$, m names of variables $\mathtt{A_1}, \ldots, \mathtt{A_m}$, and a process of synchronization C. The term

$\mathtt{bundle}$ (μ_1 $\mathtt{A_1}$; ...; μ_m $\mathtt{A_m}$;) $\mathtt{spec}$ C

defines a tuple type (with named fields $\mathtt{A_1}, \ldots, \mathtt{A_m}$) as the set of functions:

$\Xi : \{\mathtt{A_1}, \ldots, \mathtt{A_m}\} \to \bigcup_{i=1}^{m} \mathcal{T}(\mu_i)$ such that $\Xi(\mathtt{A_i}) \in \mathcal{T}(\mu_i)$.

It is reminded that the notation $\mathcal{T}(\mu_i)$ designates the domain of values (type) associated with μ_i.

When C is the process of synchronization that defines all the fields of the tuple (recursively) as being synchronous, the corresponding type is then denoted by the term:

$\mathtt{struct}$ (μ_1 $\mathtt{A_1}$; ...; μ_m $\mathtt{A_m}$;)

It can be considered, generically, that a tuple type, represented by a tuple with named or unnamed fields (cf. section V–5, page 76), can be viewed as a product of domains
$(\mu_1 \times \ldots \times \mu_m)$
where μ_k is the type of the k^{th} element of the tuple.

Declaration of a tuple variable (with named fields)

The association of a tuple type with synchronization C, with a variable, denoted by the term

$\mathtt{bundle}$ (μ_1 $\mathtt{A_1}$; ...; μ_m $\mathtt{A_m}$;) $\mathtt{spec}$ C $\mathtt{X}$

defines a polychronous tuple of signals, such that

$$
\begin{array}{ll}
(\forall t) \quad (& \\
& \quad ((\forall i) \quad ((X_t(\mathtt{A_i}) \neq \bot) \quad \Rightarrow \quad (X_t(\mathtt{A_i}) \in \mathcal{T}(\mu_i))\,)\,) \\
& \bigwedge \quad (\text{the relation defined by the process denoted by } C \text{ is verified})\,)
\end{array}
$$

Remark:
Such a declaration is a SIGNAL process with as interface, μ_1 $\mathtt{A_1}$, ..., μ_m $\mathtt{A_m}$ in input, and the empty set in output.

For the particular case of a monochronous tuple, the association denoted by the term

$\mathtt{struct}$ (μ_1 $\mathtt{A_1}$; ...; μ_m $\mathtt{A_m}$;) $\mathtt{X}$

defines a monochronous tuple *signal*, such that

$$(\forall t) \quad ((X_t \neq \bot) \quad \Rightarrow \quad ((\forall i) \quad (X_t(\mathtt{A_i}) \in \mathcal{T}(\mu_i))\,)\,)$$

Access to an element

When $\mathtt{X}$ designates a polychronous tuple the type of which is defined as the set of functions

$\Xi : \{\mathtt{A_1}, \ldots, \mathtt{A_m}\} \to \bigcup_{i=1}^{m} \mu_i$ such that $\Xi(\mathtt{A_i}) \in \mu_i$,

the term

$\mathtt{Y}$ $\mathtt{:=:}$ $\mathtt{X.A_i}$

defines a process allowing to access to a component of the tuple:

$$(\forall t) \quad (\ Y_t = X_t(\mathtt{A_i})\ )$$

Particular case: when X designates a monochronous tuple, the term

```
Y   :=: X.A_i
```

defies a monochronous process allowing to access to a component of the tuple:

$$(\forall t) \quad (\ (X_t \neq \perp) \quad \Rightarrow \quad (Y_t = X_t(\mathtt{A_i}))\ )$$

Pointwise extension

The operators defined on values of elementary types may be extended canonically (pointwise extension) to tuples.

Let us consider some *operator* F defined with the following signature:
$\mu_1 \times \ldots \times \mu_N \ \longrightarrow \ \mu_{N+1}$
(note that operators may be polymorphic on some of their operands, so that a given μ_k may stand here for some set of types).

We will denote
$(\mathtt{X\_a_{1k}},\ldots,\mathtt{X\_a_{mk}})$
the elements of a tuple $\mathtt{X_k}$ with m elements.

If at least one of the $\mathtt{X_k}$ is a tuple the elements of which are correspondingly possible arguments of the operator F, more precisely, if
$(\exists\ m) \quad (\ (\ (\forall\ k) \quad (\ (\tau(\mathtt{X_k}) = (\dot\mu_{k_1} \times \ldots \times \dot\mu_{k_m})) \bigvee (\tau(\mathtt{X_k}) = \ddot\mu_k)\ )\ ) \bigwedge ($
$\qquad (\exists\ k) \quad (\ \tau(\mathtt{X_k}) = (\dot\mu_{k_1} \times \ldots \times \dot\mu_{k_m})\ )\ )\ )$
(where $\dot\mu_{k_1}, \ldots, \dot\mu_{k_m}$ and $\ddot\mu_k$ represent some particular instances of type μ_k),
the term

```
X_N+1 :=: F( X_1,...,X_N )
```

under some interpretation I, specifies a process which defines the tuple with m elements $\mathtt{X_{N+1}}$ by a pointwise application of F:

$$\begin{array}{lll}(\forall t) & (& (\forall i,\ 1 \leq i \leq m) \quad (\ X\_a_{iN+1_t} = [|F|]_I(v_1\_a_{it}, \ldots, v_n\_a_{it})\ ) \\ & & \text{where} \\ & & ((\tau(\mathtt{X_k}) = (\dot\mu_{k_1} \times \ldots \times \dot\mu_{k_m})) \quad \Rightarrow \quad (v_k\_a_{it} = X\_a_{ikt})) \bigwedge (\\ & & ((\tau(\mathtt{X_k}) \neq (\dot\mu_{k_1} \times \ldots \times \dot\mu_{k_m})) \bigwedge (\tau(\mathtt{X_k}) = \ddot\mu_k)) \quad \Rightarrow \quad (v_k\_a_{it} = X_{kt}))\ )\end{array}$$

This defines recursively new signatures of the operators, so that the pointwise extension can be applied recursively.

III–7.2 Arrays

$\mathbb{D}$ being the set of values that can be carried by a variable, we introduce a distinguished value, denoted nil, such that, semantically, $nil \notin \mathbb{D}$ and $nil \neq \perp$. This value is in particular the value of a non defined element of an array. In the language, a value equal to nil may be any (non determined) value of the correct type.

Array types

Let m integers $n_1, \ldots, n_m$ ($n_i \in \mathbb{N}$), and a type ν.
The term

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

defines an array type as the set of functions:
$([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \mathcal{T}(\nu)$,
where $[0..n_i - 1]$ denotes the set of integers included between 0 and $n_i - 1$, and $\mathcal{T}(\nu)$ denotes the domain of values of type ν.

The *curryfied* and *non curryfied* forms of the functions defining an array type are considered as equivalent.
Thus, when the type ν is itself an array type, defined by the set of functions
$([0..n_{m+1} - 1] \times \ldots \times [0..n_{m+p} - 1]) \to \mathcal{T}(\mu)$,
the type denoted by $[n_1, \ldots, n_m]\ \nu$ is defined by the set of functions
$([0..n_1 - 1] \times \ldots \times [0..n_{m+p} - 1]) \to \mathcal{T}(\mu)$.

Declaration of an array variable

The association of an array type with a variable, denoted by the term

$$[n_1, \ldots, n_m]\ \nu\ \mathtt{X}$$

defines an array signal such that

$$\begin{array}{ll} (\forall t) & (\\ & \quad (X_t \neq \perp) \\ & \Rightarrow \quad (\ (\forall k,\ 1 \leq k \leq m) \quad (\ (\forall i_k,\ 0 \leq i_k \leq n_k - 1) \quad (\ X_t(i_1, \ldots, i_m) \in \mathcal{T}(\nu)\ )\ )\ )\ ) \end{array}$$

For X an array of type $([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu$,
the set of tuples of types $[0..n_1 - 1] \times \ldots \times [0..n_p - 1]$ where $1 \leq p \leq m$ is designated by $Dom(X)$.

Complete arrays and partial arrays

An array of type $([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu$ is said *complete* if the function
$([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu$
that defines it is total.

If this function is partial, the array is said *partial*.
In this case, it is defined by the total function
$([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu \cup \{nil\}$
that extends this partial function by associating nil with the non defined elements.

When the array defined by one of the following operators may be partial, the function described by this semantics is necessarily a restriction of the function that defines the array. The corresponding extension is such that any element non defined by the semantics is equal to nil.

Array element

When `X` designates an array the type of which is defined as the set of functions $([0..n_1-1]\times\ldots\times[0..n_m-1])\rightarrow\nu$,
and $\mathtt{I_1}, \ldots, \mathtt{I_m}$ are signals of type integer,
the term

$\mathtt{Y}$ `:=:` $\mathtt{X[I_1,\ldots,I_m]}$

defines a monochronous process allowing to access to an element of the array `X`:

$$\begin{array}{l} (\forall t) \quad (\\ \qquad (X_t \neq \perp) \\ \qquad \Rightarrow \quad ((\,(\forall I_{kt}) \quad (\,0 \leq I_{kt} \leq n_k - 1\,)\,) \bigwedge (Y_t = X_t(I_{1t},\ldots,I_{mt})))\,) \end{array}$$

This operator is generalized below (see "extraction of sub-array").

Static enumeration of array

The term

$\mathtt{X}$ `:=:` $\mathtt{[E_1,\ldots,E_n]}$

defines a monochronous process enumerating the elements of an array:

$$(\forall t) \quad (\,(X_t \neq \perp) \quad \Rightarrow \quad (\,(\forall i = 1,\ldots,n) \quad (\,X_t(i) = E_{it}\,)\,)\,)$$

Iterative enumeration of array

The term

$\mathtt{K}$ `:=: N recur f from` $\mathtt{V_0}$

(where `N`, maximum number of iterations, denotes a positive integer, which has a stricly positive upper bound, $upper\_bound(\mathtt{N})$; $\mathtt{V_0}$ denotes a value (or a tuple of values) of type μ; and `f` is a function from μ into μ),
defines a process enumerating elements of a vector of μ of size $upper\_bound(\mathtt{N})$:

$$\begin{array}{l} (\forall t) \quad (\\ \qquad (K_t \neq \perp) \\ \qquad \Rightarrow \quad (\,(\forall i) \quad (\,((0 \leq i < N_t - 1) \bigwedge ((K_t(0) = V_{0t}) \bigwedge (K_t(i+1) = [|f|]_I(K_t(i))))) \\ \qquad \bigvee \quad ((N_t \leq i < upper\_bound(\mathtt{N})) \bigwedge (K_t(i) = nil))\,)\,)\,) \end{array}$$

The equation $K_t(i) = nil$ expresses the fact that the corresponding value exists (since all the elements of an array have the same clock), but it is not determined. In the language, this can be represented by: $K_t(i) = K_t(i)$.

This form is not provided as such in the concrete syntax of the language.

A particular form is `0..N` $-\ 1$ which represents the term `N recur f from 0` where `f` designates the function on integers such that $\mathtt{f}(x) = x + 1$.

Pointwise extension

The operators defined on values of elementary types may be extended canonically (pointwise extension) to arrays.

Let us consider some operator `F` defined with the following signature:
$\mu_1 \times \ldots \times \mu_N \;\rightarrow\; \mu_{N+1}$
(note that operators may be polymorphic on some of their operands, so that a given μ_k may stand here for some set of types).

If at least one of the $\mathtt{TX_k}$ has one dimension more than the corresponding argument in the definition of the operator `F`, more precisely, if
$(\exists\ \mathrm{m})\quad ((\ (\forall\ \mathrm{k})\quad ((\tau\big(\mathtt{TX_k}\big) = [0..m-1] \rightarrow \dot{\mu}_k) \bigvee (\tau\big(\mathtt{TX_k}\big) = \ddot{\mu}_k)\ )\ ) \bigwedge ($
$\qquad(\exists\ \mathrm{k})\quad (\ \tau\big(\mathtt{TX_k}\big) = [0..m-1] \rightarrow \dot{\mu}_k\ )\ )\ )$
(where $\dot{\mu}_k$ and $\ddot{\mu}_k$ represent some particular instances of type μ_k),
the term

$\mathtt{TX_{N+1}}$ `:=: F( ` $\mathtt{TX_1}$`,...,`$\mathtt{TX_N}$ `)`

under some interpretation I, defines a monochronous process which defines the array $\mathtt{TX_{N+1}}$ by a pointwise application of `F`:

$$
\begin{array}{l}
(\forall t) \quad (\\
\qquad (TX_{N+1t} \neq \bot) \\
\quad \Rightarrow \quad (\ (\forall i,\ 0 \leq i \leq m-1) \quad (\ TX_{N+1t}(i) = [|F|]_I(v_{1t}(i), \ldots, v_{nt}(i))\ ) \\
\qquad \text{where} \\
\qquad ((\tau\big(\mathtt{TX_k}\big) = [0..m-1] \rightarrow \dot{\mu}_k) \quad \Rightarrow \quad (v_{kt}(i) = TX_{kt}(i))) \bigwedge (\\
\qquad ((\tau\big(\mathtt{TX_k}\big) \neq [0..m-1] \rightarrow \dot{\mu}_k) \bigwedge (\tau\big(\mathtt{TX_k}\big) = \ddot{\mu}_k)) \quad \Rightarrow \quad (v_{kt}(i) = TX_{kt})))\ )
\end{array}
$$

This defines recursively new signatures of the operators, so that the pointwise extension can be applied recursively.

Cartesian product

With `I` and `J` arrays of respective types
$\tau\big(\mathtt{I}\big) = [0..m-1] \rightarrow \mu$ and $\tau\big(\mathtt{J}\big) = [0..n-1] \rightarrow \nu$,
the term

`(II,JJ) :=: ≪I,J≫`

defines a monochronous tuple of signals, $(\mathtt{II}, \mathtt{JJ})$, with `II` and `JJ` of respective types
$\tau\big(\mathtt{II}\big) = [0..m*n-1] \rightarrow \mu$ and $\tau\big(\mathtt{JJ}\big) = [0..m*n-1] \rightarrow \nu$,
such that:

$$
\begin{array}{l}
(\forall t) \quad (\\
\qquad (I_t \neq \bot) \\
\quad \Rightarrow \quad (\ (\forall k,\ 0 \leq k \leq m-1) \quad (\ \ (\forall p,\ 0 \leq p \leq n-1) \quad (\\
\qquad (II_t(k*n+p) = I_t(k)) \bigwedge (JJ_t(k*n+p) = J_t(p))\ )\ \ )\ )\ )
\end{array}
$$

More generally, if `I` is a tuple (with unnamed fields) of type
$\tau\big(\mathtt{I}\big) = [0..m-1] \rightarrow \mu_1 \times \ldots \times [0..m-1] \rightarrow \mu_p$

and $\mathtt{J}$ is an array of type
$\mathcal{T}(\mathtt{J}) = [0..n-1] \rightarrow \nu$,
the term

$\mathtt{(II_1,\ldots,II_p,JJ)}$ `:=:` $\ll \mathtt{I,J} \gg$

defines a monochronous tuple of signals, $(\mathtt{II_1}, \ldots, \mathtt{II_p}, \mathtt{JJ})$, with, if II designates the tuple $(\mathtt{II_1}, \ldots, \mathtt{II_p})$, $\mathtt{II}$ and $\mathtt{JJ}$ of respective types
$\mathcal{T}(\mathtt{II}) = [0..m*n-1] \rightarrow \mu_1 \times \ldots \times [0..m*n-1] \rightarrow \mu_p$,
$\mathcal{T}(\mathtt{JJ}) = [0..m*n-1] \rightarrow \nu$,
and:

$$
\begin{array}{ll}
(\forall t) & (\\
& \quad (I_t \neq \perp) \\
& \Rightarrow \quad ((\forall k,\ 0 \leq k \leq m-1) \quad ((\forall p,\ 0 \leq p \leq n-1) \quad (\\
& \quad (II_t(k*n+p) = I_t(k)) \bigwedge (JJ_t(k*n+p) = J_t(p)) \ ) \ ) \ ) \ )
\end{array}
$$

The cartesian product is used in particular to define jointly indexes used for multi-dimensional iterations of processes.

Remark: $\ll \mathtt{I_1,\ldots,I_m} \gg = \ll \mathtt{I_1}, \ll \mathtt{I_2,\ldots,I_m} \gg \gg$

Partial definition of array

The term

$\mathtt{Y}$ `:=:` $(\mathtt{I_1}, \ldots, \mathtt{I_n})$ `:` $\mathtt{X}$

where $\mathtt{I_1}, \ldots, \mathtt{I_n}$ are integers or arrays of integers:
$\mathcal{T}(\mathtt{I_1}) = \ldots = \mathcal{T}(\mathtt{I_n}) = ([0..b_1] \times \ldots \times [0..b_p]) \rightarrow \nu$
with ν an integer type, and the basic integer values of the $\mathtt{I_i}$ are positive or zero,
$\mathcal{T}(\mathtt{X}) = ([0..c_1] \times \ldots \times [0..c_p]) \rightarrow \mu$ with $c_1 \geq b_1, \ldots, c_p \geq b_p$,
and $\mathcal{T}(\mathtt{Y}) = ([0..a_1] \times \ldots \times [0..a_n]) \rightarrow \mu \cup \{nil\}$ with for $1 \leq i \leq n$, $a_i = \max_{K \in Dom(\mathtt{I_i})} \mathtt{I_i}(K)$

defines a monochronous process which specifies, in the general case, a partially defined array:

$$
\begin{array}{ll}
(\forall t) & (\\
& \quad (X_t \neq \perp) \\
& \Rightarrow \quad (\\
& \quad ((p=0) \bigwedge (\\
& \quad (Y_t(I_{1t}, \ldots, I_{nt}) = X_t) \bigwedge (\\
& \quad (\forall J \in Dom(\mathtt{Y})) \quad ((J \neq (I_{1t}, \ldots, I_{nt})) \quad \Rightarrow \quad (Y_t(J) = nil) \ ) \ ))) \\
& \bigvee \quad ((p \geq 1) \bigwedge (\\
& \quad ((\forall (j_1, \ldots, j_n) \in \mathbb{N}^n) \quad (\\
& \quad K = \{ (k_1, \ldots, k_p) \in \mathbb{N}^p \ / \ \forall i, 1 \leq i \leq n, I_{it}(k_1, \ldots, k_p) = j_i \} \ ) \ ) \quad \Rightarrow \quad (\\
& \quad ((K = \emptyset) \quad \Rightarrow \quad (Y_t(j_1, \ldots, j_n) = nil)) \bigwedge (\\
& \quad (K \neq \emptyset) \quad \Rightarrow \quad ((K_{max} = \max_{k \in K} k) \quad \Rightarrow \quad (Y_t(j_1, \ldots, j_n) = X_t(K_{max})))))))) \ )
\end{array}
$$

where the K_{max} are obtained by the maximal elements in the sets K, using the lexicographic order on $\mathbb{N}^p$.

Extraction of sub-array

The definition of the operator of access to an element of array given above is generalized in the following way to define the extraction of sub-array.

The term

```
X   :=: Y[I1,...,In]
```

where $\mathtt{I_1}, \ldots, \mathtt{I_n}$ are integers or arrays of integers:
$\mathcal{T}(\mathtt{I_1}) = \ldots = \mathcal{T}(\mathtt{I_n}) = ([0..b_1] \times \ldots \times [0..b_p]) \to \nu$
with ν an integer type, and the basic integer values of the $\mathtt{I_i}$ are positive or zero,
$\mathcal{T}(\mathtt{Y}) = ([0..a_1] \times \ldots \times [0..a_n]) \to \mu$
and $\mathcal{T}(\mathtt{X}) = ([0..b_1] \times \ldots \times [0..b_p]) \to \mu \cup \{nil\}$

defines a monochronous process which, in the general case, extracts some sub-array from Y:

$$
\begin{array}{l}
(\forall t) \quad (\\
\qquad (Y_t \neq \bot) \\
\quad \Rightarrow \quad (\\
\qquad ((((I_{1t}, \ldots, I_{nt}) \in Dom(\mathtt{Y})) \quad \Rightarrow \quad (X_t = Y_t(I_{1t}, \ldots, I_{nt}))) \bigwedge (\\
\qquad ((I_{1t}, \ldots, I_{nt}) \notin Dom(\mathtt{Y})) \quad \Rightarrow \quad (X_t = nil))) \\
\quad \bigvee \quad ((\forall (j_1, \ldots, j_p) \in \mathbb{N}^p, \forall k, 1 \leq k \leq p,\ 0 \leq j_k \leq b_k) \quad ((\\
\qquad ((I_{1t}(j_1, \ldots, j_p), \ldots, I_{nt}(j_1, \ldots, j_p)) \in Dom(\mathtt{Y})) \quad \Rightarrow \quad (\\
\qquad X_t(j_1, \ldots, j_p) = Y_t(I_{1t}(j_1, \ldots, j_p), \ldots, I_{nt}(j_1, \ldots, j_p)))) \bigwedge (\\
\qquad ((I_{1t}(j_1, \ldots, j_p), \ldots, I_{nt}(j_1, \ldots, j_p)) \notin Dom(\mathtt{Y})) \quad \Rightarrow \quad (\\
\qquad X_t(j_1, \ldots, j_p) = nil))\ )\ ))\ )
\end{array}
$$

Sequential definition

The term

```
T   :=: T1 next T2
```

where:
$\mathcal{T}(\mathtt{T1}) = ([0..c_1] \times \ldots \times [0..c_p]) \to \mu_1 \cup \{nil\}$,
$\mathcal{T}(\mathtt{T2}) = ([0..b_1] \times \ldots \times [0..b_p]) \to \mu_2 \cup \{nil\}$ with $c_1 \geq b_1, \ldots, c_p \geq b_p$,
and $\mathcal{T}(\mathtt{T}) = ([0..c_1] \times \ldots \times [0..c_p]) \to (\mu_1 \sqcup \mu_2) \cup \{nil\}$

defines a monochronous process which specifies, in the general case, a sequential definition of an array:

$$
\begin{array}{l}
(\forall t) \quad (\\
\qquad (T_t \neq \bot) \\
\quad \Rightarrow \quad ((\forall (j_1, \ldots, j_p) \in \mathbb{N}^p, \forall k, 1 \leq k \leq p,\ 0 \leq j_k \leq c_k) \quad ((\\
\qquad (((j_1, \ldots, j_p) \in Dom(\mathtt{T2})) \bigwedge (T2_t(j_1, \ldots, j_p) \neq nil)) \quad \Rightarrow \quad (\\
\qquad X_t(j_1, \ldots, j_p) = T2_t(j_1, \ldots, j_p))) \bigwedge (\\
\qquad (((j_1, \ldots, j_p) \notin Dom(\mathtt{T2})) \bigvee (T2_t(j_1, \ldots, j_p) = nil)) \quad \Rightarrow \quad (\\
\qquad X_t(j_1, \ldots, j_p) = T1_t(j_1, \ldots, j_p)))\ )\ )\ )
\end{array}
$$

III–8 Classes of processes

The following classes of processes are usefully distinguished.

III–8.1 Iterations of functions

Let P a process defined on A_1. P is an *iteration of function* on $A_2 \subseteq A_1$ if and only if:

$$(\forall F_1, F_2 \in P) \quad (\ (\forall t_1, t_2) \quad (\ (F_{1\|A_2}(t_1) = F_{2\|A_2}(t_2)) \quad \Rightarrow \quad (F_1(t_1) = F_2(t_2)) \ ) \ )$$

Remark: An iteration of function does not need memory.

III–8.2 Endochronous processes

Let P a process defined on A_1. P is *endochronous* on $A_2 \subseteq A_1$, where A_2 is considered as a totally ordered set $\{a_1, \ldots, a_n\}$, if and only if the function

$$\begin{array}{l} \Phi : P \rightarrow \Pi_{\{a_1\}}(P) \times \ldots \times \Pi_{\{a_n\}}(P) \\ \text{such that} \\ \Phi(F) = (\Pi_{\{a_1\}}(F), \ldots, \Pi_{\{a_n\}}(F)) \end{array}$$

is injective (and thus bijective, since it is necessarily surjective).

Informally, a process is endochronous on a set of variables if any flow of this process is entirely determined by the sequences of values carried by these variables, independently of their relative presence and absence.

In other words, a process is endochronous on a set of variables if given an external (asynchronous) stimulation of these variables, it is capable of reconstructing a unique synchronous behavior (up to $\perp$-equivalence). Then, it can be implemented as a process which is mostly insensitive to internal and external propagation delays. This implementation and its context have only to agree on activation starts and on the availability of data.

Property A process P defined on A_1 is endochronous on $A_2 \subseteq A_1$ if and only if:

$$(\forall F, F' \in P) \quad (\ ((\Pi_{A_2}(F))_{\approx} = (\Pi_{A_2}(F'))_{\approx}) \quad \Rightarrow \quad (F \equiv_{\downarrow} F') \ )$$

III–8.3 Deterministic processes

A process is deterministic on a set of variables if any flow of this process is entirely determined by its restriction to this set of variables.

Let P a process defined on A_1. P is *deterministic* on $A_2 \subseteq A_1$ if and only if the function

$$\begin{array}{l} \Phi : P \rightarrow \Pi_{A_2}(P) \\ \text{such that} \\ \Phi(F) = \Pi_{A_2}(F) \end{array}$$

is injective (and thus bijective, since it is necessarily surjective).

In other words, a process is deterministic on a set of variables if any two flows of this process that have the same projection on this set of variables upto some instant t, have the same behaviors upto t.

Property A process P defined on A_1 is deterministic on $A_2 \subseteq A_1$ if and only if:

$$(\forall F, F' \in P) \quad (\ (\forall t) \quad (\ ((\Pi_{A_2}(F))_{\leq t} = (\Pi_{A_2}(F'))_{\leq t}) \quad \Rightarrow \quad (F_{\leq t} = F'_{\leq t}) \ ) \ )$$

Remarks and examples:

- For any elementary process P of the SIGNAL language of the form x `:=:` $E(y_1, \ldots, y_n)$, if $x \in \{y_1, \ldots, y_n\}$, then P is deterministic on $\{y_1, \ldots, y_n\}$.
- For any elementary process P of the SIGNAL language of the form x `:=:` $E(y_1, \ldots, y_n)$, if $x \notin \{y_1, \ldots, y_n\}$, then P is deterministic on $\{y_1, \ldots, y_n\}$.
- `X :=: Y default X`

  is not deterministic on $\{$`Y`$\}$.
- The determinism on A_i is not stable by composition and restriction.

Properties:

If a process P is an iteration of function on A_1, then it is deterministic on A_1.
If a process P is endochronous on A_1, then it is deterministic on A_1.

III–8.4 Reactive processes

Reactivity of a process with respect to some set of variables may be defined as the ability of the process to react to each configuration of these variables in all states.

Let P a process defined on A_1. P is *reactive* on $A_2 \subseteq A_1$ if and only if for each flow $F \in P$, for each $t \in dom(F)$, for each event e on A_2, there exists a flow $F' \in P$ such that:

$$(F'_{\leq t-1} = F_{\leq t-1}) \bigwedge (F'(t)_{|A_2} = e).$$

P is *strictly reactive* on $A_2 \subseteq A_1$ if and only if for each flow $F \in P$, for each $t \in dom(F)$, for each event e on A_2 *different from the absent event* $\perp_e(A_2)$, there exists a flow $F' \in P$ such that:

$$(F'_{\leq t-1} = F_{\leq t-1}) \bigwedge (F'(t)_{|A_2} = e).$$

A process which is reactive on a non empty set A_2 is obviously strictly reactive on A_2.

Examples:

- `Z :=: X default Y`

  is strictly reactive on $\{$`X`, `Y`$\}$.
- `Z :=: X and Y`

  is neither strictly reactive, nor reactive on $\{$`X`, `Y`$\}$.

III–9 Composition properties

III–9.1 Asynchronous composition of processes

The partial order of relaxation is used to define the semantics of the *asynchronous* composition of processes: roughly, the asynchronous composition of two processes P_1 and P_2 is defined by the flows the projection of which on common variables of P_1 and P_2 are relaxations of the projections on these common variables of flows of P_1 and of flows of P_2.

Definition Let P_1 and P_2 two processes defined respectively on A_1 and A_2.

The *parallel composition* (or *asynchronous composition* of P_1 and P_2, denoted $P_1 \| P_2$, is a process on $A_1 \cup A_2$ defined by:

$$
\begin{array}{rl}
P_1 \| P_2 = \{F \in \mathcal{S}_{A_1 \cup A_2} \; / & (\; (\exists F_1 \in \mathcal{S}_{A_1}, \exists F_1' \in P_1) \quad (\; (F_1 \angle F_1') \\
& \quad \bigwedge (\Pi_{A_1 \cap A_2}(F_1) \sqsubseteq \Pi_{A_1 \cap A_2}(F)) \\
& \quad \bigwedge (\Pi_{A_1 \setminus A_2}(F_1) \equiv_{\downarrow} \Pi_{A_1 \setminus A_2}(F)) \;) \;) \\
& \bigwedge (\; (\exists F_2 \in \mathcal{S}_{A_2}, \exists F_2' \in P_2) \quad (\; (F_2 \angle F_2') \\
& \quad \bigwedge (\Pi_{A_1 \cap A_2}(F_2) \sqsubseteq \Pi_{A_1 \cap A_2}(F)) \\
& \quad \bigwedge (\Pi_{A_2 \setminus A_1}(F_2) \equiv_{\downarrow} \Pi_{A_2 \setminus A_1}(F)) \;) \;) \\
& \bigwedge (F \text{ is maximal for } \angle)\}
\end{array}
$$

III–9.2 Isochrony

The property of *isochrony* characterizes processes for which synchronous and asynchronous compositions are equivalent. It means that a synchronous design composed of isochronous processes is robust to their distribution.

Definition Two processes P_1 and P_2 are said *isochronous* if and only if:

$$P_1 | P_2 = P_1 \| P_2$$

III–9.3 Endo-isochrony

A special case of practical interest is the one of endochronous processes.

Definition Let P_1 and P_2 two endochronous processes defined respectively on A_1 and A_2. Their composition $P_1 | P_2$ is said *endo-isochronous* if and only if $\Pi_{A_1 \cap A_2}(P_1) | \Pi_{A_1 \cap A_2}(P_2)$ is endochronous.

Property If $P_1 | P_2$ is endo-isochronous, then P_1 and P_2 are isochronous.

III–10 Clock system and implementation relation

The refinement of a system specification consists in transforming its abstract behaviors into more concrete ones that make intermediate computational steps explicit. Conversely, the abstraction of a behavior consists in discarding some intermediate calculations. Thus it is useful to have an *implementation relation* between processes, that takes into account a notion of time refinement.

Sampler system

Let T a trace on A_1. A *sampler system* for T is a function $s : A_1 \;\rightarrow\; A_1$ such that s is acyclic, and for all $a \in A_1$, $s(a)$ is a Boolean and

$$(\forall t) \quad (\ (T_t(s(a)) = true) \quad \Rightarrow \quad (T_t(a) \neq \perp)\ )$$

A function s is a sampler system for a process P if and only if it is a sampler system for every flow of P.

Clock system

Let T a trace on A_1. A *clock system* for T is a sampler system such that for all $a \in A_1$,

$$(\forall t) \quad (\ (T_t(s(a)) = true) \quad \Leftrightarrow \quad (T_t(a) \neq \perp)\ )$$

A function s is a clock system for a process P if and only if it is a clock system for every flow of P.

Sampling

Let T a trace on A_1 and s a sampler system for T. The *sampling* of T by s is the trace $T' = S_s(T)$ defined on A_1 such that for all $a \in A_1$, $(\forall t) \quad (\ T'_t(a) = S^*(T_t(a))\ )$ where S^* is recursively defined as follows:
if s is not defined on a, then $S^*(T_t(a)) = T_t(a)$,
if s is defined on a, then

$$\begin{array}{l} S^*(T_t(a)) = T_t(a) \text{ if } S^*(T_t(s(a))) = true, \\ S^*(T_t(a)) = \perp \text{ if } S^*(T_t(s(a))) \neq true. \end{array}$$

Let P a process defined on A_1. The sampling of P by a sampler system s for P is the process P', denoted $P' = \Sigma_s(P)$, defined as the set of flows which are equivalent to samplings of flows of P:

$$P' = \{T'_{\downarrow} \in \mathcal{S}_{A_1} \ / \ (T \in P) \bigwedge (T' = S_s(T))\}$$

Well-clocked implementation

Let P a process on A_1 and Q a process on A_2 such that there exists a one-to-one correspondence σ such that $\sigma(A_1) \subseteq A_2$, and let s a clock system on Q.
Q is a *well-clocked implementation* of P with respect to s (denoted $Q \ \preceq_s \ P$) if and only if:

$$\Pi_{\sigma(A_1)}(\Sigma_s(Q)) = P.$$

III–11 Transformation of programs

A general principal of transformation of programs (which is applied for SIGNAL programs all along the design of an application, for example for verification purpose, for implementation purpose, or to calculate abstractions of behaviors) consists in the following generic rewritting scheme: homomorphisms of programs are defined such that a program is contained in the composition of its transformations by these homomorphisms. Typically, one of these transformations is an abstract interpretation of the initial program.

Let A_1 a set of variables. We consider:

- an interpretation homomorphism, f, which associates with each elementary process P defined on A_1 a process $Q_f = f(P)$ on A_2,
- an homomorphism r, which associates with each elementary process P defined on A_1 a process $Q_r = r(P)$ on $A_1' \subseteq A_1$,

such that $\Pi_{A_1 \cap A_2}(P) \angle \Pi_{A_1 \cap A_2}(Q_f | Q_r)$
and thus $P = \Pi_{A_1}(P|(Q_f|Q_r))$.

Then we define a transformation of programs (which is an homomorphism)
$\mathcal{T}_{fr} : \mathcal{P}_{A_1} \rightarrow \mathcal{P}_{A_1' \cup A_2}$
such that
$\mathcal{T}_{fr}(P) = \textit{left}(\mathcal{T}\mathcal{T}_{fr}(P))|\textit{right}(\mathcal{T}\mathcal{T}_{fr}(P))$
with:

- $\textit{left}(< X, Y >) = X$
- $\textit{right}(< X, Y >) = Y$
- $\mathcal{T}\mathcal{T}_{fr}(P) = < f(P), r(P) >$ if P is an elementary process
- $\mathcal{T}\mathcal{T}_{fr}(P_1|P_2) = < \textit{left}(\mathcal{T}\mathcal{T}_{fr}(P_1))|\textit{left}(\mathcal{T}\mathcal{T}_{fr}(P_2)), \textit{right}(\mathcal{T}\mathcal{T}_{fr}(P_1))|\textit{right}(\mathcal{T}\mathcal{T}_{fr}(P_2)) >$

Then, $P = \Pi_{A_1}(P|\mathcal{T}_{fr}(P))$.

Chapter IV

Calculus of synchronizations and dependences

IV–1 Clocks

As said before, the clock of a signal represents the presence instants of this signal, relatively to the other ones. A system of clock relations is associated with any system of SIGNAL equations (SIGNAL process), in order to represent specifically the *synchronizations* of the process.

For that purpose, an homomorphism, $\mathcal{C}lock$, is defined on processes, which has the following property:
$\mathcal{C}lock(\mathtt{P})$ | $\mathtt{P} = \mathtt{P}$
or equivalently: $\mathtt{P} \angle \mathcal{C}lock(\mathtt{P})$
(by abuse of notation, we use the same notation for the syntactic and semantic homomorphisms).

Then, the system of clock relations is encoded as a system of polynomial equations on the field of integers modulo 3.

IV–1.1 Clock homomorphism

Let us consider the following *derived* elementary processes, in order to make easier the expression of clock equations:

- a_2 `:=:` $\hat{\ } a_1$
  is defined by a_2 `:=:` a_1 `==` a_1
  where `==` represents the equality operator defined on values of any type. The signal a_2 is defined at the same instants as the signal a_1 and at each one of these instants, its value is the Boolean value $true$ (the type of a_2 is the subtype called $event$ of the Boolean type, which contains as single value the value $true$). It is said that $\hat{\ } a_1$ represents the event clock of the signal a_1.

- $a_1 \hat{=} a_2$
  is defined by (a_3 `:=:` $\hat{\ } a_1$ `==` $\hat{\ } a_2$) `where` a_3
  and is generalized to n variables ($a_1 \hat{=} \ldots \hat{=} a_n$). It expresses that the signals a_1 and a_2 (more generally, $a_1, \ldots, a_n$) are present at the same instants (their clocks are equal).

The $\mathcal{C}lock$ homomorphism is defined as follows, depending on the types of the signals (the notation $\tau(x)$ designates the type of x): Boolean equations are left unchanged in the homomorphism.

1-a Monochronous definitions

- Definitions by extension:

  if $\tau(b) = \tau(a_1) = \ldots = \tau(a_n) = boolean$:
  b `:=:` $f(a_1, \ldots, a_n) \mapsto b$ `:=:` $f(a_1, \ldots, a_n)$

  else:
  b `:=:` $f(a_1, \ldots, a_n) \mapsto b\hat{=} a_1 \hat{=} \ldots \hat{=} a_n$

- Clock:

  b `:=:` $\hat{}a \mapsto b$ `:=:` $\hat{}a$

- Delay:

  if $\tau(b) = boolean$:
  b `:=:` a `$ init` $v \mapsto b$ `:=:` a `$ init` v

  else:
  b `:=:` a `$ init` $v \mapsto b \hat{=} a$

1-b Polychronous definitions

- Extraction:

  if $\tau(b) = boolean$:
  b `:=:` a_1 `when` $a_2 \mapsto b$ `:=:` a_1 `when` a_2

  else:
  b `:=:` a_1 `when` $a_2 \mapsto b \hat{=} \hat{}a_1$ `when` a_2

- Merging:

  if $\tau(b) = boolean$:
  b `:=:` a_1 `default` $a_2 \mapsto b$ `:=:` a_1 `default` a_2

  else:
  b `:=:` a_1 `default` $a_2 \mapsto b \hat{=} \hat{}a_1$ `default`$\hat{}a_2$

1-c Hiding

$\mathcal{C}lock(P$ `where` $a) = \mathcal{C}lock(P)$ `where` a

1-d Composition

$\mathcal{C}lock(P_1 \mid P_2) = \mathcal{C}lock(P_1) \mid \mathcal{C}lock(P_2)$

IV–1.2 Verification

As a consequence, if R is a safety property satisfied by $\mathcal{C}lock(P)$,
which is expressed by $R \mid \mathcal{C}lock(P) = \mathcal{C}lock(P)$,
R is also satisfied by P since $P = \mathcal{C}lock(P) \mid P$.

IV–1.3 Clock calculus

Since the system of clock relations handles only values of Boolean signals, and presence/absence for the other types of signals, there is a natural encoding of these values in the field $\mathbf{Z}/3\mathbf{Z}$ of integers modulo 3 (or Galois field $\mathcal{F}_3$ with three elements):
$\mathcal{F}_3 = [\{-1, 0, 1\}, +, *]$
with the usual meanings for operations and values ($+$ is the usual addition modulo 3, $*$ is the usual multiplication).

We define the set of polynomials on $\mathcal{F}_3$ and a set of variables isomorphic to the variables of a SIGNAL program. The association of the value 0 with a variable indicates the absence of value for the associated signal in the corresponding instant. With each present Boolean signal, the value -1 (which is equal to 2 in $\mathbf{Z}/3\mathbf{Z}$) is associated if its current value is $false$, and the value $+1$ is associated if its current value is $true$. Thus, the square of the value of the variable associated with a present Boolean signal is equal to 1; for each non Boolean signal, we are interested only in the presence or absence of a value at the current instant. So we associate with such a signal a squared variable.

The synchronization of a SIGNAL program is expressed by a system of equations in the set of polynomials on $\mathcal{F}_3$ defined by the homomorphism described below.

3-a Monochronous definitions

- Definitions by extension:

  b `:=:` $f(a_1, \ldots, a_n) \mapsto b^2 = a_1^2 = \ldots = a_n^2$
  (some relation on the values of $b, a_1, \ldots, a_n$ is obtained when f designates a Boolean operator).

- Clock:

  b `:=:` $\hat{\ } a \mapsto b = a^2$

- Delay:

  b `:=:` a `$ init` $v \mapsto \xi_{n+1} = (1 - a^2) * \xi_n + a,\ \xi_0 = v,\ b = a^2 * \xi_n$

3-b Polychronous definitions

- Extraction:

  b `:=:` a_1 `when` $a_2 \mapsto b = a_1 * (-a_2 - a_2^2)$

- Merging:

  b `:=:` a_1 `default` $a_2 \mapsto b = a_1 + (1 - a_1^2) * a_2$

3-c Hiding

Replaces, in the system, the hidden variable by an internal one.

3-d Composition

The system obtained for `P1 | P2` is the union of the systems obtained for `P1` and for `P2`.

3-e Static and dynamic clock calculus

Then the calculus of synchronizations (clock calculus) of a SIGNAL program is done by studying a dynamic system such as:

$$\begin{cases} X_{n+1} & = & P(X_n, Y_n) \\ Q(X_n, Y_n) & = & 0 \\ Q_0(X_0) & = & 0 \end{cases}$$

where X is a state vector in $(\mathbf{Z}/3\mathbf{Z})^p$ and Y is a vector of events (abstract interpretations of signals) that make the system evolve.

Such a dynamic system is a particular form of finite state transition system. Thus it is a model of discrete event system on which it is possible to verify properties or to make control.

Studying such a system then consists in:

- studying its *static* part, i.e., the set of constraints

$$Q(X_n, Y_n) \quad = \quad 0$$

- studying its *dynamic* part, i.e., the transition system

$$\begin{array}{rcl} X_{n+1} & = & P(X_n, Y_n) \\ Q_0(X_0) & = & 0 \end{array}$$

  and the set of its reachable states, etc.

IV–2 Context clock

The clock relations imposed by SIGNAL operators imply the existence of *context clocks* for the various occurrences of the signal variables.

A particular case of this situation is for the occurrence of constants, since such a context clock is the only way to assign a clock to the occurrence of a constant.

Occurrences of constants are allowed in SIGNAL expressions as a practical way to designate constant signals, i.e., signals with a constant value. The occurrence of such a constant, v, in some expression, stands for the occurrence of some hidden signal x, defined as x `:=:` x `$ init` v.

Each occurrence of a constant has a particular clock (which cannot be fixed explicitly since the corresponding signal is hidden): this clock is defined by the context of utilization of the constant.

It is defined a utilization mode of the constants:

- allowing as much flexible use as possible
  (we want to be able to write `x + 5` but also `x + (y default 5)`);
- allowing intuitive handling of their clocks (a constant is delivered at the clock necessary for the coherence of a synchronous expression);
- *free of interpretation for the synchronous operations* and in particular, preserving possible properties of commutativity, associativity... of these operators;
- preserving the spirit, if not the letter, of the substitution principle;
- *preserving the properties of the temporal operators:*
  - "associativity" of `when`,

– associativity of `default`,
– “right distributivity” of `when` on `default`.

These requirements lead to consider that the occurrence of a constant has a clock which is provided by the context. *This has the consequence that the substitution principle cannot apply in general.*

The rules for the definition of the context clock are introduced informally below.

- For a definition

$$X \texttt{ :=: } E$$

  the context clock of E is the clock of X.

- For a monochronous expression, the context clock of each argument is the context clock of the expression.

- For a delay

$$E_1 \texttt{ \$}$$

  the context clock of E_1 is undefined, which means that the argument of a delay cannot be a constant (note that it has also consequences on derived operators).

- For an extraction

$$E_1 \texttt{ when } C$$

  having H as context clock, the context clock of C is H, that of E_1 is the clock product of H and of the clock at which C has the value $true$
  (this can be used to assign explicitly a clock to a constant).

- For a merging of signals

$$E_1 \texttt{ default } E_2$$

  having H as context clock, the context clock of E_1 and of E_2 is H.
  For example, `5 default x` is equivalent to `5`.

In the sequel, the clock of a constant outside some context will be denoted $\hbar$.

The rules for the calculation of the clock of a constant in a given context apply also for the signals the clock of which is undefined; such a signal is obtained by the operator `var`. The clock of `var` E outside some context is also denoted $\hbar$.

IV–3 Dependences

The equations on signals imply, at the execution, an evaluation order which is described by the dependence graph.

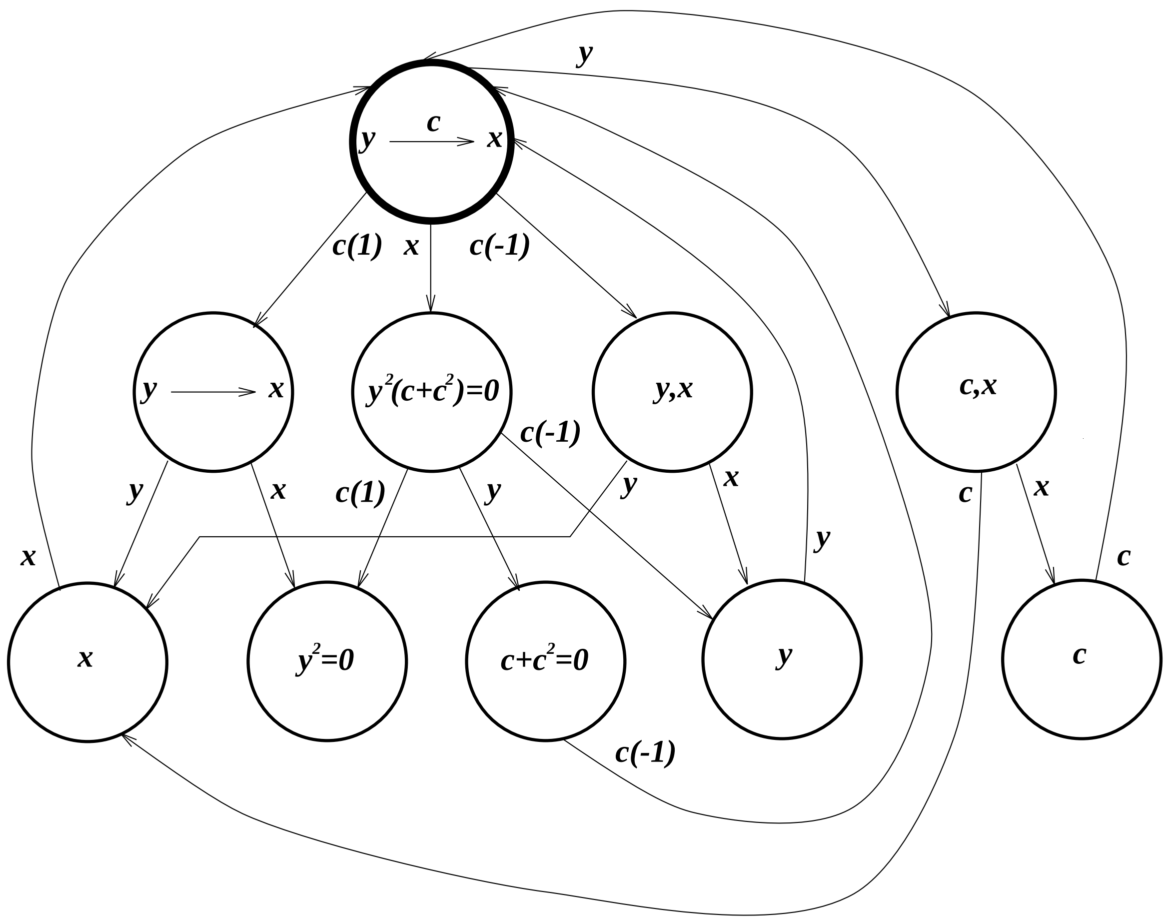


Figure B–IV.1: Formal meaning of the dependence statement.

IV–3.1 Formal definition of dependences

The following informal definition of dependences can be stated:

> A signal x depends on a signal y “at” a Boolean condition c (noted $\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}$) if, at each instant for which c is present and $true$, the event setting a value to x cannot precede the event setting a value to y.

A formal definition in the form of an automaton is presented here. We give the formal meaning of the statement

$$\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x} \qquad \text{(IV.1)}$$

in Figure B–IV.1. In the figure, the clock equations in states can be read as follows: $y^2(c + c^2) = 0$ means “absent(y) $\vee$ (absent(c) $\vee$ $c = false$)” (at the considered instant); $y^2 = 0$ means “absent(y)”; $c + c^2 = 0$ means “(absent(c) $\vee$ $c = false$)”. This figure describes a non deterministic automaton which represents the legal schedulings of calculi in one instant as conform with statement (IV.1).

- States of the automaton are made of dependence graphs and clock equations. Clock equations can be represented as equations in $\mathcal{F}_3$.
- Transitions are labelled by signals (y, c, x), or by the empty word ε. A transition labelled by y reads: “signal y occurs, with any legal value”. A transition labelled by $c(1)$ (respectively, $c(-1)$)

reads: "signal c occurs, with value *true* (respectively, *false*)"; the empty word ε represents the occurrence of any signal but (y, c, x).

- In the automaton of Figure B–IV.1, all the states have an additional transition (not represented in the Figure), labelled by ε, toward the initial state (which is represented with a thick circle in the Figure).

The automaton describing all legal schedulings of calculi for a program in one instant is obtained by a synchronous product of such basic automata, as described in section IV–3.3. Since these automata describe instantaneous behaviors, they are called *micro* automata. The states of the transition system describing the overall behavior of a program are the *forced* states (or *initial* states) of the micro automata.

IV–3.2 Implicit dependences

The equations defining a process may induce implicit dependences, such as described in the following.

Notations: For a Boolean c, we use the notation $[c]$ to represent the clock at which c has the value $true$, and $[\neg c]$ to represent the clock at which c has the value $false$.

In addition to the implicit dependences described below, the following implicit dependences apply equally:

- for any signal x, $\hat{}x \xrightarrow{\hat{}x}$ x
- for any Boolean signal c, c $\xrightarrow{\hat{}c}$ $[c]$ and c $\xrightarrow{\hat{}c}$ $[\neg c]$
- any dependence y $\xrightarrow{\text{c}}$ x implies implicitly a dependence $[c] \xrightarrow{[c]}$ x.

2-a Monochronous definitions

- Definitions by extension:

  b `:=:` $f(a_1, \dots, a_n)$
  The following implicit dependences exist:
  $a_1 \rightarrow b, \dots, a_n \rightarrow b$

- Clock:

  b `:=:` $\hat{}a$
  b is identified with the clock of a, there is no implicit dependence.

- Delay:

  b `:=:` a `$ init` v
  There is no implicit dependence.

2-b Polychronous definitions

- Extraction:

  b `:=:` a_1 `when` a_2
  The following implicit dependences exist:
  $a_1 \xrightarrow{\hat{}b} b$
  $a_2 \xrightarrow{\hat{}b} \hat{}b$

- Merging:

  b `:=:` a_1 `default` a_2

  The following implicit dependences exist:

  $a_1 \xrightarrow{\widehat{~}a_1} b$

  $a_2 \xrightarrow{\widehat{~}a_2 \widehat{~}- \widehat{~}a_1} b$

  where $\widehat{~}a_2 \widehat{~}- \widehat{~}a_1$ designates the clock representing the instants of a_2 that are not instants of a_1.

IV–3.3 Micro automata

3-a Definition of micro automata

The micro automaton associated with a program describes the legal schedulings of calculi in *one instant.*

Let A be a set of variables; $A^s = A^+ \cup A^-$ is the set of variables of A labelled by $+$ or $-$.

A word on A is any subset m of A^s such that

$$a^s \in m \Rightarrow a^{\overline{s}} \notin m \text{ where } \overline{+} = - \text{ and } \overline{-} = +$$

A micro automaton on A is a tuple

$$< S, \mathcal{P}(A^s), S_I, \Gamma \subset S \times \mathcal{P}(A^s) \times S >$$

such that:

- $S_I \subset S$: S is the set of states and S_I is a set of initial states;
- if $s_1 \overset{m_1}{\leadsto} s_2 \in \Gamma$ (Γ is the set of transitions, m_1 is the label of the transition), and $s_2 \overset{m_2}{\leadsto} s_3 \in \Gamma$, and $\ldots$ and $s_n \overset{m_n}{\leadsto} s_{n+1} \in \Gamma$, then:

$$\begin{array}{l}\forall i \neq j, m_i \cap m_j = \emptyset \\ \text{and } m = \displaystyle\bigcup_{i=1}^{n} m_i \text{ is a word on } A.\end{array}$$

- if $s_1 \overset{\emptyset}{\leadsto} s_2 \in \Gamma$ then $s_2 \in S_I$ [1]

The micro automaton is said *saturated* if, in addition,

$$s_1 \overset{m_1}{\leadsto} s_2 \in \Gamma \text{ and } s_2 \overset{m_2}{\leadsto} s_3 \in \Gamma \Rightarrow s_1 \overset{m_1 \cup m_2}{\leadsto} s_3 \in \Gamma$$

Let AUT be a micro automaton, $\mathcal{S}at(AUT)$ is the saturated micro automaton which contains AUT.

Consider two micro automata defined respectively on A_1 and A_2 with $A_1 \cap A_2 = A$. Two labels of transitions, m_1 on A_1, and m_2 on A_2, are said to *coincide* on A if and only if:

$$(m_1 \cap A^s) = (m_2 \cap A^s)$$

Let $AUT_1 =< S_1, \mathcal{P}(A_1^s), S_{1I}, \Gamma_1 >$ and $AUT_2 =< S_2, \mathcal{P}(A_2^s), S_{2I}, \Gamma_2 >$ two micro automata. Their product, denoted $AUT = AUT_1 || AUT_2$, is the micro automaton on $A_1 \cup A_2$, defined by:

$$AUT = \mathcal{S}at(< S_1 \times S_2, \mathcal{P}(A_1^s \cup A_2^s), S_{1I} \times S_{2I}, \Gamma >)$$

[1] $\emptyset$ is denoted ε in IV–3.1.

with Γ defined as follows:

$$
\begin{array}{ll}
(s_1, s_2) \overset{m_1}{\leadsto} (s'_1, s_2) \in \Gamma & \text{iff } m_1 \cap A_2^s = \emptyset \text{ and } s_1 \overset{m_1}{\leadsto} s'_1 \in \Gamma_1 \\
(s_1, s_2) \overset{m_2}{\leadsto} (s_1, s'_2) \in \Gamma & \text{iff } m_2 \cap A_1^s = \emptyset \text{ and } s_2 \overset{m_2}{\leadsto} s'_2 \in \Gamma_2 \\
(s_1, s_2) \overset{m_1 \cup m_2}{\leadsto} (s'_1, s'_2) \in \Gamma & \text{iff } m_1 \text{ and } m_2 \text{ coincide on } A_1 \cap A_2 \\
& \text{and } s_1 \overset{m_1}{\leadsto} s'_1 \in \Gamma_1 \text{ and } s_2 \overset{m_2}{\leadsto} s'_2 \in \Gamma_2
\end{array}
$$

3-b Construction of basic micro automata

(i) Micro automaton associated with a system of equations

Let us consider a system of equations on a set of variables A:

$$\Sigma : R(A) = 0$$

having at least one solution (the system encodes clock equations of a program).

A *partial valuation* of Σ is any system of equations $\Sigma' : R'(A') = 0$ equivalent to $R(A) = 0$ in which a non empty subset $\{a_1, \ldots, a_n\}$ of variables of A have been replaced by values $v_1, \ldots, v_n \in \{-1, 1\}$ such that Σ' has at least one solution.

If σ denotes such a substitution, the following notations are used:

$$
\begin{array}{ll}
\sigma(a_i) = v_i & \text{denotes the value assigned to } a_i \text{ by } \sigma \\
\sigma(R(A)) & \text{denotes the system } R'(A') \text{ obtained by the substitution.}
\end{array}
$$

Then we consider $\mathcal{P}(\Sigma)$ the set of $R'(A')$ such that there exists σ verifying $\sigma(R(A)) = R'(A')$.

The micro automaton associated with Σ is the saturated micro automaton

$$< S, \mathcal{P}(A^s), \{s_0\}, \Gamma >$$

such that:

- there exists a bijection $\phi : \mathcal{P}(\Sigma) \rightarrow S$ with $\phi(R) = s_0$
- for any valuation σ of $R'(A') \in \mathcal{P}(R(A))$,

  $$\phi(R') \overset{T}{\leadsto} \phi(\sigma(R')) \in \Gamma$$

  if and only if:

  $$
  \begin{array}{lll}
  a^+ \in T & \text{iff} & \sigma(a) = 1 \text{ and} \\
  a^- \in T & \text{iff} & \sigma(a) = -1
  \end{array}
  $$

- for all $\Sigma' : R'(A') = 0$ such that

  $$\forall a, a \in A' \Rightarrow a = 0 \text{ is a solution of } \Sigma'$$

  then

  $$\phi(R') \overset{\emptyset}{\leadsto} s_0 \in \Gamma$$

(ii) Micro automaton associated with a dependence

The micro automaton associated with

$$\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}$$

is defined as follows.

We consider the following states of resolution $\mathcal{E}$:
$\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}, \mathrm{y} \rightarrow \mathrm{x}, \{y, x\}, \{c, x\}, (y^2(c + c^2) = 0), \{y\}, \{x\}, \{c\}, (c + c^2 = 0), (y^2 = 0)$

The micro automaton associated with $\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}$ is the saturated micro automaton

$$\mathcal{S}at(< S, \mathcal{P}(\{y, x, c\}^s), \{s_0\}, \Gamma >)$$

such that there exists a bijection $\phi : \mathcal{E} \rightarrow S$ with $\phi(y \xrightarrow{c} x) = s_0$
and with Γ defined as follows:

$$\begin{array}{l}
\phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \overset{c^+}{\rightsquigarrow} \phi(\mathrm{y} \rightarrow \mathrm{x}) \in \Gamma \\
\phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \overset{c^-}{\rightsquigarrow} \phi(\{y, x\}) \in \Gamma \\
\phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \overset{y^\pm}{\rightsquigarrow} \phi(\{c, x\}) \in \Gamma \\
\phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \overset{x^\pm}{\rightsquigarrow} \phi(y^2(c + c^2) = 0) \in \Gamma \\
\phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \overset{\emptyset}{\rightsquigarrow} \phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \in \Gamma \\
\\
\phi(\mathrm{y} \rightarrow \mathrm{x}) \overset{y^\pm}{\rightsquigarrow} \phi(\{x\}) \in \Gamma \\
\phi(\mathrm{y} \rightarrow \mathrm{x}) \overset{x^\pm}{\rightsquigarrow} \phi(y^2 = 0) \in \Gamma \\
\phi(\mathrm{y} \rightarrow \mathrm{x}) \overset{\emptyset}{\rightsquigarrow} \phi(\mathrm{y} \xrightarrow{\mathrm{c}} \mathrm{x}) \in \Gamma
\end{array}$$

In addition, Γ contains all other transitions providing from resolutions such as described in (i).

The corresponding micro automaton is displayed in Figure B–IV.1, where c^+ and c^- are denoted respectively $c(1)$ and $c(-1)$, and $y^\pm$ and $x^\pm$ are denoted y and x; moreover, the $\emptyset$ transitions have been omitted in the figure. The complete micro automaton is the saturated micro automaton which contains this one.

(iii) Micro automaton associated with a memorization

The encoding presented in IV–3.1 considers not only the clocks, but also the *values* of the Boolean flows: delayed Boolean flows are the *state variables* of the program.

The micro automaton associated with x `:=:` y `$ init` v where x and y are Boolean flows is the saturated micro automaton obtained from the micro automaton depicted on Figure B–IV.2. The initial states of this micro automaton are the states represented with a thick circle in the Figure.

(iv) Micro automaton associated with a process

The micro automaton associated with a process is the product of the micro automata associated with each definition involved in the process.

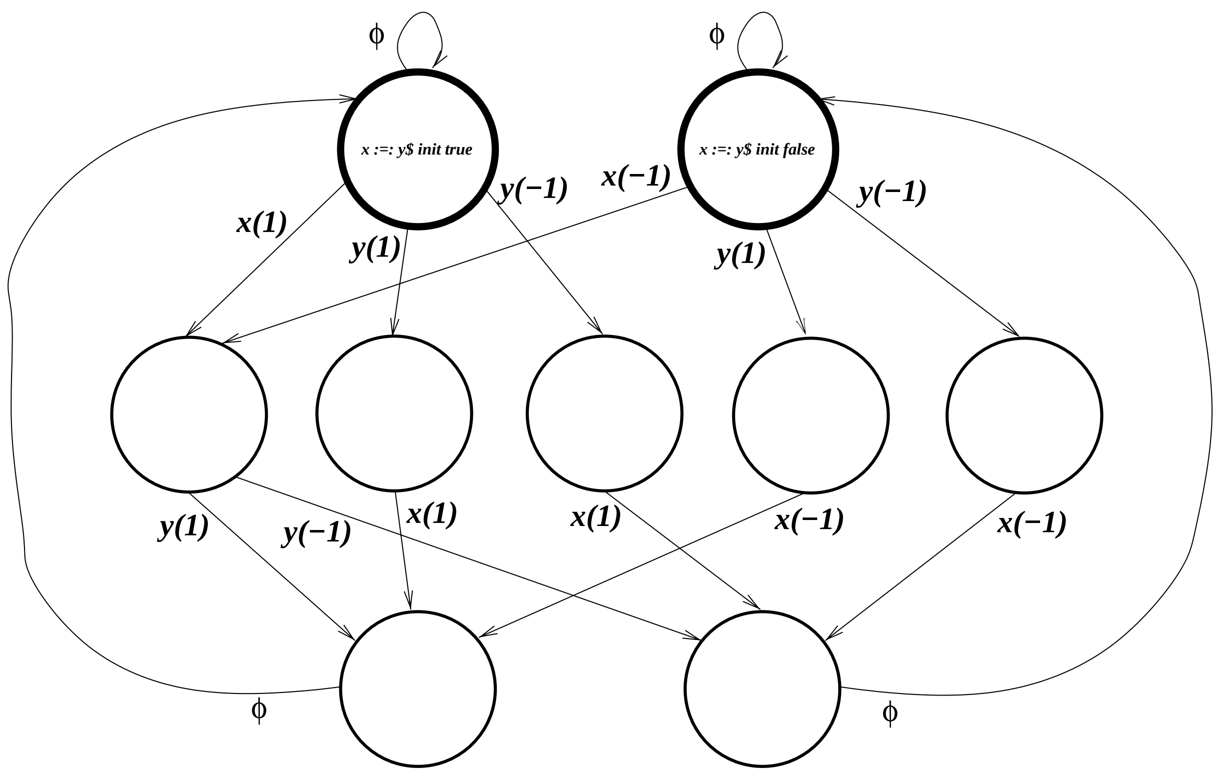


Figure B–IV.2: Micro automaton of x `:=:` y `$ init` v

Part C

THE SIGNALS

Chapter V

Domains of values of the signals

A signal is a sequence of values associated with a clock. These values have all the same type, which is considered as the type of the sequence. The objective of this chapter is to present the notations used to represent these types and the processings which are applied on them. An element of the set of types of the SIGNAL language is denoted $type$.

Let E be a term of the SIGNAL language; we denote by $\tau(\mathsf{E})$ the type associated with the term E and, when E is a constant expression, $\varphi(\mathsf{E})$ the value of this expression, elaborated in the context in which E appears.

The set of types of the SIGNAL language contains the scalar types, the external types, the array types and the tuple types.

1. **Context-free syntax**

SIGNAL-TYPE ::= **Scalar-type**
| **External-type**
| **ENUMERATED-TYPE**
| **ARRAY-TYPE**
| **TUPLE-TYPE**

V–1 Scalar types

Scalar types are the following: synchronization types, integer types, real types, complex types, character type, string type; the integer, real and complex types compose the set of numeric types; character and string types compose the set of alphabetic types.

1. **Context-free syntax**

Scalar-type ::= **Synchronization-type**
| **Numeric-type**
| **Alphabetic-type**

Numeric-type ::= **Integer-type**
| **Real-type**
| **Complex-type**

Alphabetic-type ::= $\boxed{\textbf{char}}$
| $\boxed{\textbf{string}}$

V–1.1 Synchronization types

The synchronization types are used to define the clocks of the signals. They are the type $event$ (or pure signal) and the type $boolean$.

Denotations of types

1. **Context-free syntax**

   Synchronization-type ::= $\boxed{\textbf{event}}$
   | $\boxed{\textbf{boolean}}$

2. **Types**

   (a) τ(`event`) = $event$
   (b) τ(`boolean`) = $boolean$

Denotations of values

- A signal of type $event$ takes its values in a single-element set: there is no associated constant and a parameter cannot be of that type.
- The constants of type $boolean$ are the logical values denoted with the syntax of a **Boolean-cst** (cf. part A, section II–2.2, page 23).
- The default initial value of type $boolean$ is the value $false$.

V–1.2 Integer types

Integer values can be in short representation (type $short$), normal representation (type $integer$), or long representation (type $long$); a given implementation may not distinguish these types. In this document, the notations $max\ long$, $min\ long$, $max\ integer$, $min\ integer$, $max\ short$ and $min\ short$ will be used to designate respectively: the greatest representable integer (of type $long$), the smallest representable integer (of type $long$), the greatest integer of type $integer$, the smallest integer of type $integer$, the greatest integer of type $short$ and the smallest integer of type $short$. These values depend of the implementation and respect the following order:
$min\ long \ \leq\ min\ integer \ \leq\ min\ short \ \leq 0 \ <\ max\ short \ \leq\ max\ integer \ \leq\ max\ long$
$min\ integer \ <\ 0$

Denotations of types

1. **Context-free syntax**

   Integer-type ::= $\boxed{\textbf{short}}$
   | $\boxed{\textbf{integer}}$
   | $\boxed{\textbf{long}}$

2. **Types**

   (a) τ(`short`) = $short$

(b) τ(`integer`) = $integer$
(c) τ(`long`) = $long$

Denotations of values

The positive values of an integer type are denoted following the syntax of an **Integer-cst** (cf. part A, section II–2.3, page 24). A negative value has not a direct representation: it is obtained using the operator $\boxed{-}$ applied to a positive value.

1. **Types**

   (a) The type of an **Integer-cst** E is the smallest integer type that contains it.

2. **Semantics**

   - An **Integer-cst** denotes an integer value represented in decimal base, contained between 0 and $max\ long$.
   - An occurrence of an integer value of type $short$ (respectively, $integer$ and $long$) smaller than $min\ short$ (respectively, $min\ integer$ and $min\ long$) or greater than $max\ short$ (respectively, $max\ integer$ and $max\ long$) results, in the considered type, in a value depending of the implementation.
   - For an **Integer-type**, the default initial value is the value 0.

Bounded integers

Integers have a special role since they can be used to index arrays. In that case, we have to consider bounded values.

In this document, for a given signal E, we will use sometimes the following notations:

- $lower\_bound(E)$ designates the lower bound of the values of E;
- $upper\_bound(E)$ designates the upper bound of the values of E.

These bounds are constant integers.

V–1.3 Real types

The real values can be in simple precision representation (type $real$) or double precision representation (type $dreal$); a given implementation may not distinguish these types.

Denotations of types

1. **Context-free syntax**

   Real-type ::= **[real]**
   | **[dreal]**

2. **Types**

   (a) τ(`real`) = $real$
   (b) τ(`dreal`) = $dreal$

Denotations of values E_1`.`E_2`e`E_3 (simple precision) or E_1`.`E_2`d`E_3 (double precision)

A value of real type is denoted following the syntax of a **Real-cst** (cf. part A, section II–2.4, page 24). A **Real-cst** denotes the approximate value of a real number.

1. **Types**

   (a) A **Simple-precision-real-cst** is of type $real$.

   (b) A **Double-precision-real-cst** is of type $dreal$.

2. **Semantics**

   - The value $\varphi(E_i)$, when E_i is omitted, is 0.
   - If E_2 has n digits, the value of the constant is the approximate value of $(\varphi(E_1) + \varphi(E_2) * 10^{-n}) * 10^{\varphi(E_3)}$.
   - For a **Real-type**, the default initial value is the value `0.0` or `0.0d0` following the type.

V–1.4 Complex types

The complex values have the common representation of their components (simple or double precision, respectively types $complex$ and $dcomplex$); both types are distinguished in a given implementation if and only if the type $dreal$ is distinguished from the type $real$.

Denotations of types

1. **Context-free syntax**

   Complex-type ::= $\boxed{\textbf{complex}}$
   | $\boxed{\textbf{dcomplex}}$

2. **Types**

   (a) $\tau(\texttt{complex}) = complex$

   (b) $\tau(\texttt{dcomplex}) = dcomplex$

Denotations of values

A value of complex type is obtained for example in the following expression, the first element of which is the real part and the second one the imaginary part (cf. part C, section VI–8.1, page 128).

1. **Examples**

   (a) `1.0 @ (-1.0)`

For a **Complex-type**, the default initial value is the pair of default real initial values.

V–1.5 Character type

The type $character$ contains the set of the admitted characters in the language.

Denotation of type

1. **Types**

   (a) $\mathcal{T}($`char`$) = character$

Denotations of values

A value of type $character$ is denoted by a **Character-cst** (cf. part A, section II–2.5, page 24).

The default initial value of type $character$ is the character `'\000'`.

V–1.6 String type

The type $string$ allows to represent any sequence of admitted characters. The value of the maximal authorized size for a string, $maxStringLength$, depends of the implementation.

Denotation of type

1. **Types**

   (a) $\mathcal{T}($`string`$) = string$

Denotations of values

A value of type $string$ is denoted by a **String-cst** (cf. part A, section II–2.6, page 25).
The default initial value of type $string$ is the empty string `""`.

V–2 External types

External types make possible the use of signals the type of which is not a type of the language.

Denotation of type A

An external type is designated by a name.

1. **Context-free syntax**

   External-type ::= **Name-***type*

2. **Types**

   (a) For an external type with name A, $\mathcal{T}(A) = A$
   Two external types with distinct names are not comparable.

3. **Examples**

   (a) `pointer` is an external type with name `pointer`.

Denotations of values

An external constant can be denoted by a name; the value of an external constant can be defined by the environment of the program (cf. part E, chapter XII, page 191).

For example the identifier `nil` can represent a constant of type `pointer`.

For any external type A, it is possible to define a constant that represents the default initial value of type A (cf. section V–7, page 84).

The only operations the semantics of which is defined on external type signals are operations of description of communication graphs (which are polymorphic operations).

V–3 Enumerated types

Enumerated types allow to represent finite domains of values represented by distinct names. These values (the enumerated values) are the constants of the type to which they belong.

Denotation of types `enum (`a_1`, ..., `a_m`)`

An enumerated type is defined by the list (considered as an ordered list) of its values (the enumerated values) and by its name (cf. section V–7, page 84): `type A = enum (`a_1`, ..., `a_m`);`
However, like for the other types, such a name does not necessarily exist. In that case, the name of the type is empty.
The definition of an enumerated type declares its enumerated values.

1. **Context-free syntax**

   ENUMERATED-TYPE ::=
   $\boxed{\textbf{enum}}$ $\boxed{\textbf{(}}$ **Name-***enum-value* **{** $\boxed{\textbf{,}}$ **Name-***enum-value* **}*** $\boxed{\textbf{)}}$

2. **Types**

   (a) The type of the enumerated type is:
   $\mathcal{T}($`A = enum (`a_1`, ..., `a_m`)`$) = A \times \{a_1, \ldots, a_m\}$
   where $\{a_1, \ldots, a_m\}$ represents the finite set of ordered values $a_1, \ldots, a_m$. It means that the name of an enumerated type (the name that is given in the declaration of the type) is part of that type. Depending on the implementation, it can be the case or not that synonyms (cf. section V–7, page 84) are considered in the definition of the type.
   If the enumerated type is not designated by a name, then its type is just the finite set of its ordered values.

   (b) The type of the enumerated values of an enumerated type is this enumerated type: $\mathcal{T}(a_1) = \ldots = \mathcal{T}(a_m) = \mathcal{T}($`enum (`$a_1$`, ..., `$a_m$`)`$)$

   (c) Two enumerated types are considered to be equal if they have both the same name, and the same set of enumerated values, *in the same order.* Two enumerated types that are not designated by a name are considered to be equal if they have the same set of enumerated values, in the same order.

3. **Semantics**
   The enumerated values of an enumerated type are ordered (syntactic order of their declaration). All the values of a given type are distinct; these values are distinguished by their name.

4. **Examples**

   (a) `type color = enum (yellow, orange);` and `type fruit = enum (apple, orange);` are two enumerated types, each one defining an enumerated value named "orange". Both enumerated values named "orange" are distinct values, with different types. The next paragraph describes the way allowing to distinguish them.

Denotation of values

$\#a_i$ or $A\#a_i$
where A is the name of the enumerated type.
Note: the symbol # does not appear in the definition of the type (and its enumerated values), but only for the use of an enumerated value.

1. **Context-free syntax**

   ENUM-CST ::=
   $\boxed{\#}$ **Name-***enum-value*
   | **Name-***type* $\boxed{\#}$ **Name-***enum-value*

2. **Semantics**

   - The notation $\#a_i$ can be used to reference an enumerated value a_i in a context in which there is no possible ambiguity on the referenced value. If it is not the case, the notation $A\#a_i$ has to be used, where A designates the enumerated type.
   - The default initial value of an enumerated type is the first value of its declaration.

3. **Clocks** An enumerated value a_i (designated by $\#a_i$ or $A\#a_i$) is a constant.

   (a) $\omega(a_i) = \hbar$

4. **Examples**

   (a) `color#orange` and `fruit#orange` designate two different enumerated values (of two different types) with the same name.

V–4 Array types

An array is a structure allowing to group together *synchronous* elements of a same type. The description of such a structure and of the access to its elements uses constant expressions that have the general syntax of signal expressions (**S-EXPR**).

Denotation of types $[n_1, \ldots, n_m]\nu$

An array type is defined by its dimensions and by the type of its elements.

1. **Context-free syntax**

ARRAY-TYPE ::=

$\boxed{\textbf{[}}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }*** $\boxed{\textbf{]}}$ **SIGNAL-TYPE**

2. **Types**

   (a) The elaborated values of n_1 ($\varphi(n_1)$), ..., n_m ($\varphi(n_m)$) are strictly positive integers.

   (b) The type of the array is:
   $\tau(\texttt{[}n_1\texttt{, ..., }n_m\texttt{]}\nu) = ([0..\varphi(n_1) - 1] \times \ldots \times [0..\varphi(n_m) - 1]) \rightarrow \tau(\nu)$.

   (c) When the type $\tau(\nu)$ itself is an array type $\texttt{[}n_{m+1}\texttt{, ..., }n_{m+p}\texttt{]}\mu$, then the type of the array is:
   $\tau(\texttt{[}n_1\texttt{, ..., }n_m\texttt{]}\nu) = ([0..\varphi(n_1) - 1] \times \ldots \times [0..\varphi(n_{m+p}) - 1]) \rightarrow \tau(\mu)$.

3. **Clocks** The integers n_i must be constant expressions.

   (a) $\omega(n_i) = \hbar$

4. **Properties**

   (a) The types $\texttt{[}n_1\texttt{, }n_2\texttt{]}\nu$ and $\texttt{[}n_1\texttt{] [}n_2\texttt{]}\nu$ are the same.

5. **Examples**

   (a) `[10,10] integer` is a two dimensions integer array (the bounds of the array begin implicitly at index 0 in each dimension).

   (b) `[n] pointer` is a vector of values of external type `pointer`.

Denotations of values

A constant array is defined by a constant expression of array (cf. part D, section IX–2, page 149); the elements that compose a constant array are from the same domain.

For an **ARRAY-TYPE**, the default initial value is an array of which each element has the default initial value of the type of the elements of the array.

V–5 Tuple types

not yet fully imple-mented

The SIGNAL language allows to define structured types, called in a generic way *tuple* types. Two categories of tuple types, called also tuple types with named fields, can be associated with the objects of the SIGNAL language in declarations:

- polychronous tuples (designated by the keyword `bundle`)[1];
- monochronous tuples (designated by the keyword `struct`)

(remark: the objects declared of tuple type can also be called *tuples*).

An object declared of type polychronous tuple is in fact a gathering of objects (family of objects). In this way, *a polychronous tuple of signals is not a signal* (for example, in the general case, it has no clock); it cannot be used as the type of the elements of an array. At the opposite, an object declared of

[1] not yet implemented in POLYCHRONY: clock properties of bundles are not taken into account.

type monochronous tuple can be a signal: it has a clock (delivered by the operator ^) and it can be used as the type of the elements of an array.

A general rule is that operators on signals do not apply on polychronous tuples, but they are pointwise extended on the fields of these tuples (cf. part D, chapter X, page 169).

The SIGNAL language allows also to manipulate gatherings (or tuples) of objects with no explicit declaration of these gatherings. They define in fact tuples with unnamed fields, the type of which is a product of types (cf. section V–6.2, paragraph "Order on tuples", page 80). The operators defined on signals are pointwise extended to tuples with unnamed fields (cf. part D, chapter X, page 169). By extension, it will be possible to refer to the clock of a tuple of signals if all the signals of the tuple have the same clock.

Denotation of types

`struct (`μ_1 X_1`; ...; `μ_m X_m`;)`
or
`bundle (`μ_1 X_1`; ...; `μ_m X_m`;) spec `C

A tuple type is defined by a list of typed and named fields; in addition, clock properties can be specified on the fields of a tuple.

The description of such a type uses lists of declarations of sequence identifiers **S-DECLARATION** (cf. section V–9, page 87) for the designation of the fields, and properties **SPECIFICATION-OF-PROPERTIES** (cf. part E, section XI–6, page 180) to express the clock properties that must be respected by the signals corresponding to the fields defined by the type. These properties should describe exclusively *clock properties* on the fields of the tuple, excluding for instance graph properties. Note that constraints on values can be specified under the form of constraints on clocks.

A tuple type can be multi-clock (polychronous) or mono-clock (monochronous). If it is multi-clock, it is distinguished by the keyword `bundle` and it can contain specifications of clock properties applying on its fields. If it is mono-clock, it is distinguished by the keyword `struct` and all its fields are implicitly synchronous; in this case, it can be used as type of the elements of an array.

1. **Context-free syntax**

   TUPLE-TYPE ::=
   $\boxed{\textbf{struct}}\ \boxed{\textbf{(}}$ **NAMED-FIELDS** $\boxed{\textbf{)}}$
   | $\boxed{\textbf{bundle}}\ \boxed{\textbf{(}}$ **NAMED-FIELDS** $\boxed{\textbf{)}}$
   [SPECIFICATION-OF-PROPERTIES]

   NAMED-FIELDS ::=
   { S-DECLARATION }$^+$

2. **Types**

   (a) From the point of view of the domains of associated values, the polychronous or monochronous tuple types with named fields are designated in the same way in this document. The domain is a non associative product (i.e., preserving the structuring) of typed named fields.

   (b) $\tau($`struct (`μ_1 X_1`; ...; `μ_m X_m`;)`$)$
   $= bundle(\{X_1\} \rightarrow \tau(\mu_1) \times \ldots \times \{X_m\} \rightarrow \tau(\mu_m))$

   (c) $\tau($`bundle (`μ_1 X_1`; ...; `μ_m X_m`;) spec `$C)$
   $= bundle(\{X_1\} \rightarrow \tau(\mu_1) \times \ldots \times \{X_m\} \rightarrow \tau(\mu_m))$

(d) A type
$bundle(\{X_1\} \to \mathcal{T}(\mu_1) \times \ldots \times \{X_m\} \to \mathcal{T}(\mu_m))$
defines a set of functions
$\Xi : \{X_1, \ldots, X_m\} \to \bigcup_{i=1}^{m} \mathcal{T}(\mu_i)$ such that $\Xi(X_i) \in \mathcal{T}(\mu_i)$.

3. **Semantics**
The tuple types with named fields (`struct` and `bundle`) allow to define structured types as non associative grouping of typed named fields: `(`μ_1 X_1`; ...;` μ_m X_m`;)`. The specifications of properties `spec` C apply on the fields of the tuple. They establish constraints that must be respected by the signals defined with such a type (space of synchronization of the values of the domain).

4. **Examples**

(a) `struct (integer X1, X2;)`
is a tuple of two synchronous integers.

(b) `bundle (integer A; boolean B;) spec (| A ^# B |)`
defines a union of types as a tuple the fields of which are mutually exclusive.

Denotations of values

A constant tuple is defined by a constant expression of tuple (cf. part D, section VIII–1, page 143).
For a **TUPLE-TYPE**, the default initial value is recursively the tuple of initial values of its fields.

V–6 Structure of the set of types

A partial order is defined on the types such that there exists a “natural” plunging of a smaller set into a greater one. The types are organized into domains corresponding to theoretical sets (non constrained by the implementation). In this way, the domain of synchronization values (**Synchronization-type**) contains the types $event$ and $boolean$; the domain of integers (**Integer-type**) contains the types $short$, $integer$, and $long$; the domain of reals (**Real-type**) contains the types $real$ and $dreal$; the domain of complex (**Complex-type**) contains the types $complex$ and $dcomplex$.

V–6.1 Set of types

The set of types is composed of the types the expressions of which, in the SIGNAL language, described in the following summary, are derived from the variable **SIGNAL-TYPE**:

SIGNAL-TYPE

- **Scalar-type**
  - **Synchronization-type**
    - $\boxed{\textbf{event}}$ denotes the type $event$
    - $\boxed{\textbf{boolean}}$ denotes the type $boolean$
  - **Numeric-type**
    - **Integer-type**
      - $\boxed{\textbf{short}}$ denotes the type $short$
      - $\boxed{\textbf{integer}}$ denotes the type $integer$
      - $\boxed{\textbf{long}}$ denotes the type $long$
    - **Real-type**
      - $\boxed{\textbf{real}}$ denotes the type $real$
      - $\boxed{\textbf{dreal}}$ denotes the type $dreal$
    - **Complex-type**
      - $\boxed{\textbf{complex}}$ denotes the type $complex$
      - $\boxed{\textbf{dcomplex}}$ denotes the type $dcomplex$
  - **Alphabetic-type**
    - $\boxed{\textbf{char}}$ denotes the type $character$
    - $\boxed{\textbf{string}}$ denotes the type $string$
- **External-type**
  - **Name-***type*
    - Generic form of the external types: $name$
- **ENUMERATED-TYPE**
  - $\boxed{\textbf{enum}}$ $\boxed{\textbf{(}}$ **Name-***enum-value* { $\boxed{\textbf{,}}$ **Name-***enum-value* }* $\boxed{\textbf{)}}$
    - Generic form of the enumerated types: $A \times \{a_1, \ldots, a_m\}$
- **ARRAY-TYPE**
  - $\boxed{\textbf{[}}$ **S-EXPR** { $\boxed{\textbf{,}}$ **S-EXPR** }* $\boxed{\textbf{]}}$ **SIGNAL-TYPE**
    - Generic form of the array types: $([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu$
- **TUPLE-TYPE**
  - $\boxed{\textbf{struct}}$ $\boxed{\textbf{(}}$ **NAMED-FIELDS** $\boxed{\textbf{)}}$
  - $\boxed{\textbf{bundle}}$ $\boxed{\textbf{(}}$ **NAMED-FIELDS** $\boxed{\textbf{)}}$ [**SPECIFICATION-OF-PROPERTIES**]
    - Generic form of the tuple types with named fields:
      $bundle(\{X_1\} \to \mu_1 \times \ldots \times \{X_m\} \to \mu_m)$

V–6.2 Order on types

Order on scalar and external types

The order on scalar and external types of the SIGNAL language is described in the figure C–V.1. A downward solid arrow ($\longrightarrow$) links a type with a type directly superior from the same domain (two types of a same domain are *comparable*); the other arrows represent basic conversions, the semantics of which is described below. The other conversions are obtained by composition of conversions. The partial order is denoted $\sqsubseteq$.

The notion of "comparable types" is extended to arrays and tuples.

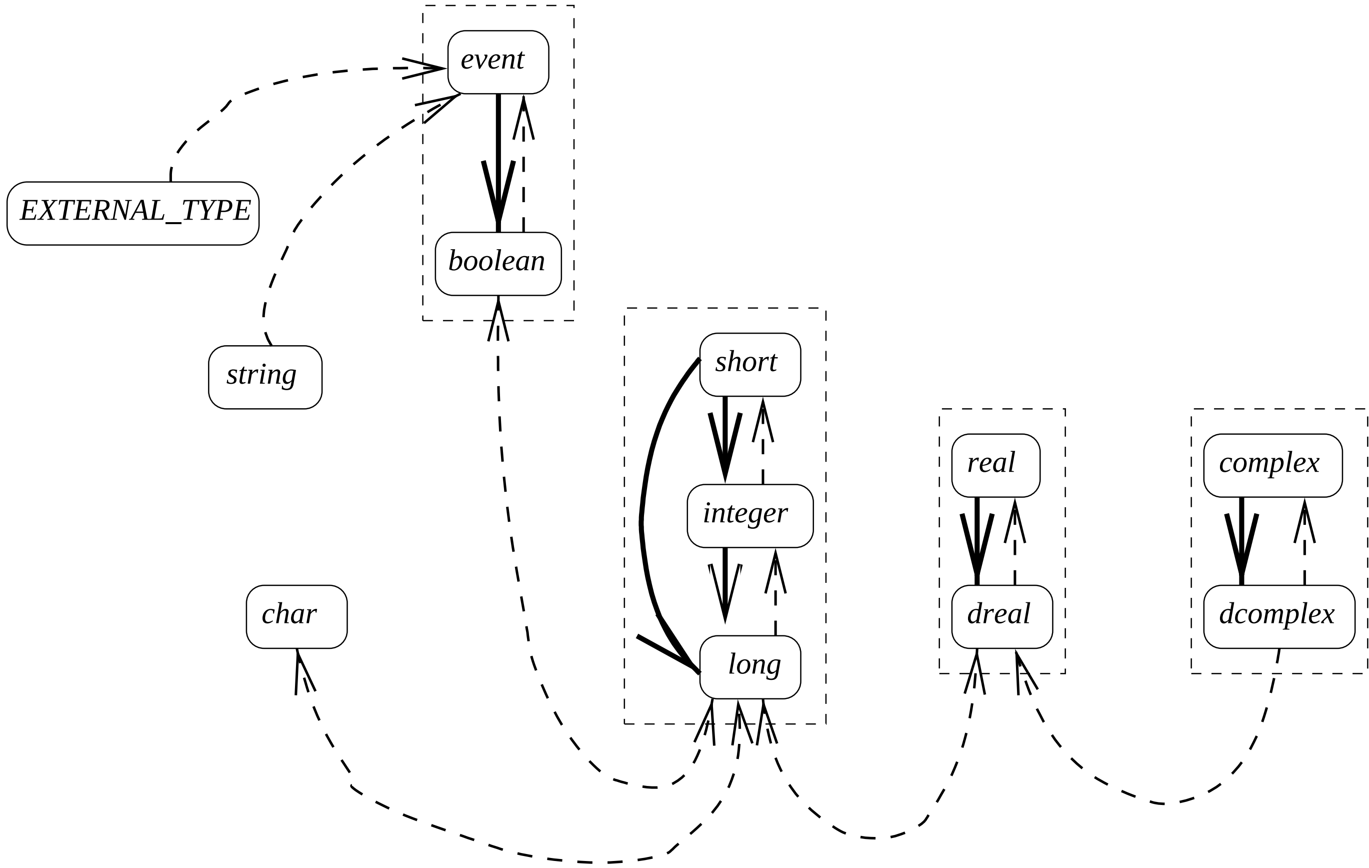


Figure C–V.1: Order and conversions on scalar and external types

Order on arrays

The order on scalar and external types is extended to arrays:

- $([0..m_1 - 1] \times \ldots \times [0..m_k - 1]) \rightarrow \mu \sqsubseteq ([0..n_1 - 1] \times \ldots \times [0..n_l - 1]) \rightarrow \nu$ if and only if
  - $*$ $k = l$
  - $*$ $\forall i \quad 1 \leq i \leq k \Rightarrow m_i \leq n_i$
  - $*$ and $\mu \sqsubseteq \nu$

Order on tuples

A product of types is a type, called tuple type with unnamed fields, which preserves the structuring. There is no syntactic designation of such a type (it is not possible to declare some object of type tuple with unnamed fields); however, it is possible to manipulate objects of type tuple with unnamed fields (product of types). A tuple with unnamed fields with a single element is considered as isomorphic to this element.

The product of types $\mu_1, \ldots, \mu_n$ (in this order) is denoted $(\mu_1 \times \ldots \times \mu_n)$.

The order on the types of signals is extended as follows on tuples:

- $bundle(\{X_1\} \to \mu_1 \times \ldots \times \{X_n\} \to \mu_n) \sqsubseteq bundle(\{Y_1\} \to \nu_1 \times \ldots \times \{Y_p\} \to \nu_p)$ if and only if:
  $p = n$
  and $(\forall\ i)$ $(\ X_i = Y_i$ et $\mu_i \sqsubseteq \nu_i\ )$
- $(\mu_1 \times \ldots \times \mu_n) \sqsubseteq bundle(\{Y_1\} \to \nu_1 \times \ldots \times \{Y_p\} \to \nu_p)$ if and only if:
  $(\mu_1 \times \ldots \times \mu_n) \sqsubseteq (\nu_1 \times \ldots \times \nu_p)$
- $(\mu_1 \times \ldots \times \mu_n) \sqsubseteq (\mu_1 \times (\mu_2 \times \ldots \times \mu_n))$
- $(\mu_1 \times \ldots \times \mu_n) \sqsubseteq (\nu_1 \times \ldots \times \nu_p)$ if and only if:

$$((n = p) \bigwedge (\ (\forall i) \quad (\ \mu_i \sqsubseteq \nu_i\ )\ ))$$

or

$$\begin{array}{l} (\ (\exists k, l) \quad (\ ((i < k) \quad \Rightarrow \quad (\mu_i \sqsubseteq \nu_i)) \\ \bigwedge \quad (((\mu_k \times \ldots \times \mu_{k+l}) \sqsubseteq \nu_k) \\ \bigwedge \quad (((k + l = n) \bigwedge (k = p)) \\ \qquad \text{or} \quad (((k + l < n) \bigwedge (k < p)) \bigwedge ((\mu_{k+l+1} \times \ldots \times \mu_n) \sqsubseteq (\nu_{k+1} \times \ldots \times \nu_p)))))\ )\ ) \end{array}$$

Notation

The notation $\mu \sqcup \nu$ is used to designate the upper bound of two comparable types μ and ν.

V–6.3 Conversions

A conversion is a function for which the image of an object of the type μ of the argument is an object of the type ν required by the context of utilization. The conversion functions for the types defined in the SIGNAL language have the name of the reserved designation of the expected type or in general the name of the expected type. In this document, these functions are denoted as follows, in order to describe their semantics:
$\mathcal{C}_\nu^\mu\ :\ \mu \to \nu$
Direct conversion functions are available in the language, even if their semantics is described in terms of composition of conversions.

3-a Conversions between comparable types

Between two directly comparable types $\mu \sqsubseteq \nu$, the two following conversions are defined:

1. the conversion $\mathcal{C}_\nu^\mu$ from a smaller type μ to a greater type ν lets the values unchanged;
2. the conversion $\mathcal{C}_\mu^\nu\ :\ \nu \to \mu$ which is the inverse of the previous one for the values of type μ.

The conversion functions are extended to any pair of comparable types:

- if $\nu_1 \sqsubseteq \mu \sqsubseteq \nu_2$ then $\mathcal{C}_{\nu_2}^{\nu_1} = \mathcal{C}_{\nu_2}^{\mu} \circ \mathcal{C}_{\mu}^{\nu_1}$;
- $\mathcal{C}_\mu^\mu$ is the identity function.

Implicit conversions

The only implicit conversions are the conversions $\mathcal{C}_\nu^\mu$ for which $\mu \sqsubseteq \nu$. Implicit conversions do not need to be explicited in the language.

3-b Conversions toward the domain "Synchronization-type"

The conversions $\mathcal{C}^{\mu}_{event}$ are defined for each μ (except if μ is a polychronous tuple); Trivially, they deliver the single value of type $event$.

the conversions $\mathcal{C}^{\mu}_{boolean}$ depend of the implementation while respecting the following rules:

- The conversion $\mathcal{C}^{long}_{boolean}$ verifies:
  - $\mathcal{C}^{long}_{boolean}(0) \ = \ false$
  - $\mathcal{C}^{long}_{boolean}(1) \ = \ true$
- For a **Scalar-type** μ distinct from $event$
  $\mathcal{C}^{\mu}_{boolean} \ = \ \mathcal{C}^{long}_{boolean} \circ \mathcal{C}^{\mu}_{long}$

3-c Conversions toward the domain "Integer-type"

The conversions $\mathcal{C}^{\mu}_{short}$ depend of the implementation while respecting the following rules:

- $\mathcal{C}^{integer}_{short}(v) = v$ if v is greater than $min\ short$ and smaller than $max\ short$ (non strictly in both cases),
- $\mathcal{C}^{long}_{short} \ = \ \mathcal{C}^{integer}_{short} \circ \mathcal{C}^{long}_{integer}$
- for a **Scalar-type** or **ENUMERATED-TYPE** μ
  $\mathcal{C}^{\mu}_{short} \ = \ \mathcal{C}^{long}_{short} \circ \mathcal{C}^{\mu}_{long}$

The conversions $\mathcal{C}^{\mu}_{integer}$ depend of the implementation while respecting the following rules:

- $\mathcal{C}^{long}_{integer}(v) = v$ if v is greater than $min\ integer$ and smaller than $max\ integer$ (non strictly in both cases),
- for a **Scalar-type** μ which is not smaller than $integer$ (for the order defined on the types), or for μ an **ENUMERATED-TYPE**
  $\mathcal{C}^{\mu}_{integer} \ = \ \mathcal{C}^{long}_{integer} \circ \mathcal{C}^{\mu}_{long}$

The conversions $\mathcal{C}^{\mu}_{long}$ depend of the implementation while respecting the following rules:

- the conversion $\mathcal{C}^{boolean}_{long}$ is defined by the following rules:
  - $\mathcal{C}^{boolean}_{long}(false) \ = \ 0$
  - $\mathcal{C}^{boolean}_{long}(true) \ = \ 1$
- the value of $\mathcal{C}^{character}_{long}(C)$ is the numerical value of the code of the character C,
- the value of $\mathcal{C}^{dreal}_{long}(v)$ is the integer part n of v if n is greater than $min\ long$ and smaller than $max\ long$ (non strictly in both cases),
- for a **Scalar-type** μ which is not smaller than $long$ (for the order defined on the types)
  $\mathcal{C}^{\mu}_{long} \ = \ \mathcal{C}^{dreal}_{long} \circ \mathcal{C}^{\mu}_{dreal}$
- for an **ENUMERATED-TYPE** μ equal to $A \times \{a_1, \ldots, a_m\}$, the conversion $\mathcal{C}^{\mu}_{long}$ is defined by: $\mathcal{C}^{\mu}_{long}(a_1) \ = \ 0, \ldots, \mathcal{C}^{\mu}_{long}(a_m) \ = \ m-1$.

3-d Conversions toward the domain “Real-type”

For each **Real-type**, a given implementation distinguihes the *safe* numbers (in the same sense as in Ada), which have an exact representation.

The conversions $\mathcal{C}^{\mu}_{real}$ depend of the implementation while respecting the following rules:

- if v, of type $dreal$, is a safe number in the type $real$, $\mathcal{C}^{dreal}_{real}(v) = v$
- the conversion preserves the order on the real numbers included between the smallest and the greatest safe number in the type $real$,
- for a **Scalar-type** μ
  $\mathcal{C}^{\mu}_{real} = \mathcal{C}^{dreal}_{real} \circ \mathcal{C}^{\mu}_{dreal}$

The conversions $\mathcal{C}^{\mu}_{dreal}$ depend on the implementation while respecting the following rules:

- the conversion preserves the order on the real numbers included between the smallest and the greatest safe number in the type $dreal$,
- $\mathcal{C}^{dcomplex}_{dreal}(re@im) = re$
- $\mathcal{C}^{complex}_{dreal} = \mathcal{C}^{dcomplex}_{dreal} \circ \mathcal{C}^{complex}_{dcomplex}$
- if v, of type $long$, is a safe number in the type $dreal$, $\mathcal{C}^{long}_{dreal}(C) = v$
- for a **Scalar-type** distinct of the previous ones,
  $\mathcal{C}^{\mu}_{dreal} = \mathcal{C}^{long}_{dreal} \circ \mathcal{C}^{\mu}_{long}$

3-e Conversions toward the domain “Complex-type”

There are no conversions toward the domain **Complex-type** except those internal to that domain. However, a given implementation can provide such conversion functions. Note that the conversion of a $real$ re into a $complex$ (respectively, of a $dreal$ re into a $dcomplex$) can be obtained by $re@0.0$.

The conversion $\mathcal{C}^{dcomplex}_{complex}$ depends on the implementation while respecting the following rule:

- $\mathcal{C}^{dcomplex}_{complex}(re@im) = \{\mathcal{C}^{dreal}_{real}(re), \mathcal{C}^{dreal}_{real}(im)\}$

3-f Conversions toward the types $character$ and $string$

The conversions $\mathcal{C}^{\mu}_{character}$ depend on the implementation while respecting the following rules:

- the value of $\mathcal{C}^{long}_{character}(v)$ is the character (if it exists) whose decimal value of its code is equal to v,
- for a **Scalar-type** μ $\mathcal{C}^{\mu}_{character} = \mathcal{C}^{long}_{character} \circ \mathcal{C}^{\mu}_{long}$

There is no conversion toward the type $string$.

3-g Conversions of arrays

For any tuple of strictly positive integers $n_1, \ldots, n_m$, and any conversion $\mathcal{C}_{\nu}^{\mu}$,
the conversion $\mathcal{C}_{([0..n_1-1] \times \ldots \times [0..n_m-1]) \rightarrow \nu}^{([0..n_1-1] \times \ldots \times [0..n_m-1]) \rightarrow \mu}$ is defined by:
$\mathcal{C}_{([0..n_1-1] \times \ldots \times [0..n_m-1]) \rightarrow \nu}^{([0..n_1-1] \times \ldots \times [0..n_m-1]) \rightarrow \mu}(T) \;=\; \mathcal{C}_{\nu}^{\mu} \circ T$

3-h Conversions of tuples

Conversions of tuples with unnamed fields

For any conversions $\mathcal{C}_{\nu_1}^{\mu_1}, \ldots, \mathcal{C}_{\nu_n}^{\mu_n}$,
the conversion $\mathcal{C}_{(\nu_1 \times \ldots \times \nu_n)}^{(\mu_1 \times \ldots \times \mu_n)}$ is defined by:
$\mathcal{C}_{(\nu_1 \times \ldots \times \nu_n)}^{(\mu_1 \times \ldots \times \mu_n)}(x_1, \ldots, x_n) = (\mathcal{C}_{\nu_1}^{\mu_1}(x_1), \ldots, \mathcal{C}_{\nu_n}^{\mu_n}(x_n))$

Conversions of tuples with unnamed fields toward tuples with named fields

For any conversions $\mathcal{C}_{\nu_1}^{\mu_1}, \ldots, \mathcal{C}_{\nu_n}^{\mu_n}$ and any tuple with named fields of type $bundle(\{X_1\} \rightarrow \nu_1 \times \ldots \times \{X_m\} \rightarrow \nu_m)$ that defines a function Ξ (cf. section V–5, page 76),
the conversion $\mathcal{C}_{bundle(\{X_1\} \rightarrow \nu_1 \times \ldots \times \{X_m\} \rightarrow \nu_m)}^{(\mu_1 \times \ldots \times \mu_n)}$ is defined by:
$\mathcal{C}_{bundle(\{X_1\} \rightarrow \nu_1 \times \ldots \times \{X_m\} \rightarrow \nu_m)}^{(\mu_1 \times \ldots \times \mu_n)} = \Xi \circ \mathcal{C}_{(\nu_1 \times \ldots \times \nu_n)}^{(\mu_1 \times \ldots \times \mu_n)}$

V–7 Denotation of types

A type can be designated by an identifier, declared in a **DECLARATION-OF-TYPES** (it cannot be an identifier of predefined type). In particular, such a type identifier can designate a generic type, which can represent a type of the language or an external type.

Denotation of type A

1. **Context-free syntax**

   SIGNAL-TYPE ::=

   Name-*type*

2. **Types**

   (a) The type designated by a **Name-***type* A is the type associated with A in the declaration of the type A.

Declarations of types

`type` A `=` μ`;` or
`type` A`;`

1. **Context-free syntax**

   DECLARATION-OF-TYPES ::=

   $\boxed{\textbf{type}}$ **DEFINITION-OF-TYPE { $\boxed{\textbf{,}}$ DEFINITION-OF-TYPE }*** $\boxed{\textbf{;}}$

DEFINITION-OF-TYPE ::=

  Name-*type*
| **Name-***type* $\boxed{=}$ **DESCRIPTION-OF-TYPE**

DESCRIPTION-OF-TYPE ::=

  SIGNAL-TYPE
| **EXTERNAL-NOTATION [TYPE-INITIAL-VALUE]**

TYPE-INITIAL-VALUE ::=

  $\boxed{\textbf{init}}$ **Name-***constant*

2. **Types**

   (a) The declaration `type` A `=` μ`;` defines the type A as being equal to the type μ:
   $\tau(A) = \tau(\mu)$

   (b) When it appears in the formal parameters of a model (cf. part E, section XI–5, page 179), the declaration `type` A`;` defines a formal generic type whose actual value is provided within the call of the model (cf. section VI–1.2, page 97).
   Otherwise, the declaration `type` A`;` is an abbreviated form for `type` A `= external;` that specifies A as an externally defined type. It means that A is either an external type the actual definition of which is provided in the environment of the program, or it is a formal generic type, whose actual value is defined elsewhere in the context or is provided in a module (cf. part E, section XII–1, page 191).

   It is possible to specify, in the declaration of an external type A, a constant name (which must be the name of an external constant of type A—cf. section V–8, page 85), that allows to designate the default initial value of that type.

   A given compiler may consider that such a constant name appearing as default initial value of an external type constitutes an implicit declaration of this external constant.

   (c) If A is defined as an external type, then:
   $\tau(A) = A$

   (d) Two external types with distinct names A and B are considered as different types.

3. **Properties**

   (a) With the declarations `type` A `=` μ`;` and `type` B `=` μ`;`
   then $\tau(A) = \tau(B) = \tau(\mu)$.
   Some implementations may not ensure this property.

4. **Examples**

   (a) `type T = [n] integer;` declares the type `T` as vector of integers, of size `n`.

V–8 Declarations of constant identifiers

`constant` μ $X_1 = E_1, \ldots, Y_j, \ldots, X_n = E_n$`;`

A constant sequence is a sequence each element of which has the same value. Such a sequence can be designated by an identifier.

1. **Context-free syntax**

DECLARATION-OF-CONSTANTS ::=
 $\boxed{\textbf{constant}}$ **SIGNAL-TYPE**
 DEFINITION-OF-CONSTANT { $\boxed{\textbf{,}}$ **DEFINITION-OF-CONSTANT }*** $\boxed{\textbf{;}}$

DEFINITION-OF-CONSTANT ::=
 Name-*constant*
 | **Name-***constant* $\boxed{=}$ **DESCRIPTION-OF-CONSTANT**

DESCRIPTION-OF-CONSTANT ::=
 S-EXPR
 | **EXTERNAL-NOTATION**

2. **Types**

 (a) $(\forall\ i)$ $(\ \tau(\mu) = \tau(X_i)\ )$

 (b) $(\forall\ i)$ $(\ \tau(E_i) \sqsubseteq \tau(X_i)\ )$

 (c) When the constant declaration does not contain an expression, for example for Y_j, it is in fact an abbreviated form for Y_j `= external;` that defines Y_j as an externally defined constant. It means that the value of Y_j is provided either in a module (cf. part E, section XII–1, page 191), or in the environment of the program.

3. **Semantics**

- Any expression defining a constant must be monochronous and functional (without side effect). With this reserve, the set of expressions admitted by a compiler contains the operators and intrinsic functions and can contain a set of functions depending of a particular environment.
- The elaboration of the expression E_i, in the context $\mathcal{C}_D$ of the declaration D, minus the identifier X_i, provides a constant value (determined at compile time) $\varphi(E_i) = v$;
- the declaration D hides any higher declaration of X_i for the context $\mathcal{C}_D$ and the included contexts;
- in a context where D is visible, the elaboration of an occurrence of the identifier X_i provides the value $\varphi(X_i) = v$.

4. **Clocks** An occurrence of use of X_i (or Y_j) is considered as an occurrence of the designated constant.

 (a) $\omega(E_i) = \hbar$

 (b) $\omega(X_i) = \hbar$

 (c) $\omega(Y_j) = \hbar$

5. **Examples**

   (a) The declaration
   `constant real PI = 3.14;`
   defines the identifier `PI` of type $real$ and with value $\varphi($`3.14`$)$.

   (b) The declaration
   `constant [2,2] real UNIT = [[1.0,0.0],[0.0,1.0]];`
   defines the identifier `UNIT` as a unit real matrix.

   (c) The declaration
   `constant RECTANGLE BASE;`
   where `RECTANGLE` is an identifier of external type, defines a constant of that type: `BASE`, the value of which should be provided at code generation.

   (d) The declaration
   `constant integer L = M + N;`
   is incorrect if `M` or `N` does not designate a constant or a parameter; if it is correct, it defines the identifier `L` as being equal to the sum of the constants $\varphi($`M`$)$ and $\varphi($`N`$)$.

V–9 Declarations of sequence identifiers

μ ID_1, ..., ID_j `init` V_j, ..., ID_n;

A sequence of values is provided with a type (the one of its elements); this type is associated with an identifier in a declaration. In such a declaration, an identifier can designate a static parameter (formal "signal"), a signal, or a tuple of signals. Initialization values can be associated with signals and tuples of signals (ID_j `init` V_j) in order to define their initial value(s) when these initial values are not defined elsewhere.

1. **Context-free syntax**

   S-DECLARATION ::=

   SIGNAL-TYPE
   DEFINITION-OF-SEQUENCE { $\boxed{,}$ DEFINITION-OF-SEQUENCE }* $\boxed{;}$

   DEFINITION-OF-SEQUENCE ::=

   Name-*signal*
   | **Name-*signal*** $\boxed{\textbf{init}}$ **S-EXPR**

2. **Types**

   (a) The declared names must be mutually distinct. The same type $\tau(\mu)$ is given to the identifiers $ID_1, \ldots, ID_n$ in the context of the declaration.

   (b) For a signal expression ("assignment", passage of static parameter or positional identification) associating a value v with an identifier ID_i declared with type μ, we must have $\tau(v) \sqsubseteq \mu$.

   (c) The rules applying to initial values are exactly those described in the section "Initialization expression" (cf. section VI–3.1, page 107).

3. **Semantics**

- μ ID_1`, ...,` ID_n`;` declares the sequences (signals or parameters) $ID_1, \ldots, ID_n$. If μ designates a polychronous tuple type then the identifiers $ID_1, \ldots, ID_n$ designate tuples of signals (and not, strictly speaking, signals); the signals represented by these tuples are, recursively, the fields of the tuples (the fields can be themselves tuples). For example, if μ designates a tuple type with named fields `bundle (`μ_1 X_1`; ...;` μ_m X_m`;) ...` then each tuple ID_i gathers the signals (or, recursively, the tuples of signals) designated by ID_i`.`$X_1, \ldots, ID_i$`.`X_m (cf. part D, section VIII–3, page 144), which have respectively the types $\mu_1, \ldots, \mu_m$.
- The semantics of an initialization expression specified in a declaration is exactly the same as that described in the section "Initialization expression" (cf. section VI–3.1, page 107). The association of an initialization with a signal declaration specifies a default initialization for the corresponding signal. It can be overloaded by the definition of that signal (in that case, it is unnecessary or only partly necessary).

4. **Clocks**

   (a) The relations on the clocks of initialization expressions are described in the section "Initialization expression" (cf. section VI–3.1, page 107).

5. **Examples**

   (a) The declaration `real X, Y;` declares the signals `X` and `Y` of type $real$.

   (b) The declaration `[n] integer V;` declares the vector of integers `V`, of size `n`.

V–10 Declarations of state variables

`statevar` μ ID_1 `init` V_1`, ...,` ID_j`, ...,` ID_n `init` V_n`;`

A state variable is a typed sequence the elements of which are present as frequently as necessary (it is available at a clock which is upper than the upper bound of the clocks of all the signals of the compilation unit in which it is declared). A state variable is defined via partial definitions the clock of which are well defined. It keeps its previous value as long as a new one is defined. It should have an initial value associated with its declaration (if it has not, it takes as initial value the default initial value of its type). A state variable can be used only in a context which defines a context clock. A state variable cannot be declared as input or output of a model of process.

1. **Context-free syntax**

   DECLARATION-OF-STATE-VARIABLES ::=

   $\boxed{\textbf{statevar}}$ **SIGNAL-TYPE**

   DEFINITION-OF-SEQUENCE { $\boxed{\textbf{,}}$ **DEFINITION-OF-SEQUENCE }*** $\boxed{\textbf{;}}$

2. **Types**

   (a) The declared names must be mutually distinct. The same type $\tau(\mu)$ is given to the identifiers $ID_1, \ldots, ID_n$ in the context of the declaration.

(b) For a signal expression (partial "assignment" associating a value v with an identifier ID_i declared with type μ, we must have $\tau(v) \sqsubseteq \mu$.

(c) The rules applying to initial values are exactly those described in the section "Initialization expression" (cf. section VI–3.1, page 107).

3. **Semantics**

- `statevar` μ ID_1`, ...,` ID_n`;` declares the state variables $ID_1, \ldots, ID_n$.
- The semantics of an initialization expression specified in a declaration is exactly the same as that described in the section "Initialization expression" (cf. section VI–3.1, page 107).

Note: The INRIA POLYCHRONY environment allows in some cases that the type of a constant, a sequence identifier or a state variable is not provided explicitly in their declaration (the corresponding **SIGNAL-TYPE** is simply omitted). The corresponding type must be deduced from the context of use of the object.

Chapter VI

Expressions on signals

The values associated with signals are determined by equations on signals; these equations are built by composition of sub-systems of equations (named also processes) from elementary equations.

This chapter presents the expressions of definition of signals (**S-EXPR**). This presentation is preceded by an introduction to the expressions of composition of definitions (**P-EXPR**).

VI–1 Systems of equations on signals

Composition of definitions of signals

The equations of definition of signals can be composed by the operator $\boxed{\,|\,}$ (see chapter VII, "Expressions on processes"). An expression on processes

E_1 | E_2

defines the signals (or, equivalently, has as outputs the signals) defined in each one of its sub-expressions, and has as inputs the input signals of each one of these sub-expressions which are not outputs of the other one. The value of an input signal of a sub-expression, which is defined in the other one, is the value associated by this definition. As a signal cannot have a double complete definition, a given signal identifier representing a totally defined signal cannot be output of two sub-expressions. However, it is possible to have several *partial definitions*, in different sub-expressions, for a given signal (partial definitions are syntactically distinguished).

An expression on processes can be parenthesized by $\boxed{(|}$ on the left and by $\boxed{|)}$ on the right (note the presence of the symbol $\boxed{\,|\,}$).

A given output of an expression on processes can be hidden through the operator $\boxed{/}$ (see chapter VII, "Expressions on processes"). An expression on processes

E_1 / a_1

has as outputs the outputs of E_1 distinct from a_1 and for inputs the inputs of E_1.

The signals are defined by explicit elementary equations of **DEFINITION-OF-SIGNALS**, **CONSTRAINT**s (cf. section VI–5.3, page 121), or by referring to systems of equations declared as models of processes (**INSTANCE-OF-PROCESS**).

VI–1.1 Elementary equations

A definition of signals allows to define a signal or a set of signals with the syntax given below. A definition of signals is an expression of processes.

1-a Equation of definition of a signal

X := E

1. **Context-free syntax**

   ELEMENTARY-PROCESS ::=

   DEFINITION-OF-SIGNALS

   DEFINITION-OF-SIGNALS ::=

   Name-*signal* $\boxed{\texttt{:=}}$ **S-EXPR**

2. **Profile**
   An equation of definition of a signal has as output the defined signal and as inputs the inputs of the expression E distinct of the output.

   - $!\,(X \texttt{ := } E) = \{X\}$
   - The inputs of E are the signal identifiers that have at least one occurrence in E.
     $?\,(X \texttt{ := } E) = ?\,(\mathrm{E}) - !\,(X \texttt{ := } E)$

3. **Types**

   (a) $\tau(E) \sqsubseteq \tau(X)$

4. **Semantics**
   The signal X is equal to the signal resulting from the evaluation of E. An occurrence of X in the expression E builds a recursive definition.

5. **Definition in SIGNAL**
   Though it is the most frequently form of equation used in the SIGNAL language, X := E is not the basic form. The sign $\boxed{\texttt{:=}}$ expresses that the equation is oriented, while in the basic form (cf. part B, chapter III, page 29) the sign $\boxed{\texttt{:=:}}$ is used to express the fact that equations are non oriented (cf. section VI–6, page 122).
   It is equal to the following process, where the dependences are made explicit:

   ```
   (|  X  :=:  E
    |  E  -->  X
    |)
   ```

6. **Clocks** A signal represented by an identifier and the signal that defines it are synchronous.

   (a) $\omega(X) = \omega(E)$

7. **Graph**

   (a) $E \rightarrow X$

8. **Examples**

   (a) if x, y, z designate signals:
   x := y + z defines the signal designated by x, equal to the sum of the signals designated respectively by y and z; this expression has as inputs y and z and as output x.

1-b Equation of multiple definition of signals

$(X_1, \ldots, X_n) := E$

1. **Context-free syntax**

   DEFINITION-OF-SIGNALS ::=
   $\boxed{(}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}*** $\boxed{)}$ $\boxed{:=}$ **S-EXPR**

2. **Profile**
   An equation of multiple definition of signals has the inputs and outputs defined by the following rules.

   - The identifiers of defined signals represent the outputs of the equation:
     $!\big((X_1, \ldots, X_n) := E\big) = \{X_1, \ldots, X_n\}$
   - The inputs of the equation are the inputs of E which are not outputs of the equation:
     $?\big((X_1, \ldots, X_n) := E\big) = ?\,(\mathrm{E}) - !\big((X_1, \ldots, X_n) := E\big)$

3. **Types**

   (a) $\tau\big((X_1, \ldots, X_n)\big) = (\tau(X_1) \times \ldots \times \tau(X_n))$
   (b) $\tau(E) \sqsubseteq (\tau(X_1) \times \ldots \times \tau(X_n))$

4. **Semantics**

   - $X_1, \ldots, X_n$ designate signals or tuples of signals.
   - E can be viewed as a tuple of n elements: let $(E_1, \ldots, E_n)$ this tuple.
   - Each signal or tuple X_i is respectively equal to the signal or tuple E_i that corresponds to it positionally as output of E.

5. **Definition in SIGNAL**
   $(X_1, \ldots, X_n) := E$
   is equal to the following process:

   $$\begin{array}{lll} (\,| & X_1 & := \; E_1 \\ & \vdots & \\ \;\;| & X_n & := \; E_n \\ \;\;|\,) & & \end{array}$$

   As a particular case, when the defined signal or tuple is unique, $(X) := E$ is equivalent to:
   $X := E$
   (the syntax without parentheses as described in 1-a can be used when X is a tuple).

6. **Clocks** A *signal* represented by an identifier and the signal E_i that defines it are synchronous. In this case, there is:

   (a) $\omega(X_i) = \omega(E_i)$

7. **Graph**

   (a) $E_i \rightarrow X_i$

8. **Examples**

   (a) if `x`, `y`, `z`, `a` designate signals and `P` a model with one formal parameter, one input and three outputs:
   `(x,y,z) := P{ n }( a+5 )` defines the signals designated by `x`, `y` and `z`, equal respectively to the first, second and third output of the model `P` instantiated with the parameter `n` and taking a_t+5 as input at each occurrence of `a`; this expression has as input `a` and as outputs `x`, `y` and `z`;

   (b) if `w`, `v`, `b` also designate signals:
   `(w,x,y,z,v) := (a,P{ n }( a+5 ),b)` defines the signals `w`, `x`, `y`, `z` and `v`, equal respectively to the signal `a`, to the first, the second and the third output of the process `P`, and to the signal `b`; this expression has as inputs `a` and `b` and as outputs `w`, `x`, `y`, `z` and `v`; it is equivalent to the composition
   `(| (w,v) := (a,b) | (x,y,z) := P{ n }( a+5 ) |);`

   (c) if `x` designates a tuple with named fields whose fields are respectively `x1` and `x2`, and `a`, `b` designate signals:
   `(a,b) := (x.x1,x.x2)` defines the signals `a` and `b` equal respectively to the first and the second component of the tuple `x`;

   (d) if `x` designates a tuple with named fields and `a`, `b` designate signals:
   `x := (a,b)` defines the tuple `x` the components of which are respectively equal to the signals `a` and `b`.

1-c Equation of partial definition of a signal

Equations of partial definition of a signal are a way to avoid the syntactic single assignment rule, even if semantically, this rule applies. Each one of the partial definitions of a given signal contributes to the overall definition of this signal. These partial definitions can appear in different syntactic contexts. All these partial definitions have to be mutually compatible. One default partial definition can appear for a given signal: it allows to complete the definition of the signal by a default value when the partial definitions do not apply. The overall definition of the signal is considered as complete in a compilation unit.

Equations of partial definition are syntactically distinguished by the use of the special symbol $\boxed{\textbf{::=}}$. The use of this symbol is mandatory to allow the presence of different syntactic definitions of a given signal. The syntactic single assignment rule still applies when the assignment symbol $\boxed{\textbf{:=}}$ is used. In particular, it is not possible to have both complete definition and partial ones for a given signal.

X `::=` E
X `::=` `defaultvalue` E

1. **Context-free syntax**

   DEFINITION-OF-SIGNALS ::=

   Name-*signal* $\boxed{\textbf{::=}}$ **S-EXPR**

   | **Name-***signal* $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

2. **Profile**
   An equation of partial definition of a signal has as output the partially defined signal and as inputs

the inputs of the expression E distinct of the output.

- $!\,(X$ `::=` $E) = \{X\}$
- $?\,(X$ `::=` $E) = ?\,(\mathrm{E}) - !\,(X$ `::=` $E)$
- $!\,(X$ `::=` `defaultvalue` $E) = \{X\}$
- $?\,(X$ `::=` `defaultvalue` $E) = ?\,(\mathrm{E}) - !\,(X$ `::=` `defaultvalue` $E)$

3. **Types**

   (a) $\tau(E) \sqsubseteq \tau(X)$

4. **Definition in SIGNAL**
   Let the following composition represent the whole set of partial definitions of a signal X in a given compilation unit:

```
(|  X ::= E1
      ⋮
 |  X ::= En
 |  X ::= defaultvalue En+1
 |)
```

   It is semantically equivalent to:

```
(|  X := E1 default X
      ⋮
 |  X := En default X
 |  X := (En+1 when (X ^- (E1 ^+ ... ^+ En))) default X
 |  X ^= E1 ^+ ... ^+ En ^+ X
 |)
```

5. **Clocks** For the above set of partial definitions of the signal X, any two different expressions E_i must have the same value at their common instants if they have such common instants. The clock of X is greater than the upper bound of the clocks of the expressions E_i, $i = 1, \ldots, n$.

   (a) $\forall i, j = 1, \ldots, n \quad \omega(E_i \hat{*}\, E_j) = \omega($`when` $((E_i$ `when^` $E_j)$ `==` $(E_j$ `when^` $E_i)))$
   (b) $\omega(X) = \omega(E_1 \hat{+} \ldots \hat{+} E_n \hat{+} X)$
   (c) For $i = 1, \ldots, n$, the clock of any expression E_i cannot be a context clock: in particular, E_i cannot be a constant expression or a direct reference to a state variable.
   The clock of E_{n+1} *can be* a context clock.

1-d Equation of partial definition of a state variable

State variables (cf. section V–10, page 88) can be defined exclusively by equations of partial definition. These equations define the *next* values of a state variable. The last defined value, which is the only one that can be accessed at every instant, is referred to via the special notation X`?` (cf. section VI–2.3, page 106).

X `::=` E

1. **Context-free syntax**

   The syntax is the same as that of an equation of partial definition of a signal.

2. **Types**
   (a) $\tau(E) \sqsubseteq \tau(X)$

3. **Definition in SIGNAL**
   Let the following composition represent the whole set of partial definitions of a state variable X in a given compilation unit:

```
(|  X ::= E1
    ⋮
 |  X ::= En
 |)
```

   It is semantically equivalent to:

```
(|  next_X := E1 default next_X
    ⋮
 |  next_X := En default next_X
 |  X := next_X $
 |) /  next_X
```

4. **Clocks** For the above set of partial definitions of the state variable X, any two different expressions E_i must have the same value at their common instants if they have such common instants.
   (a) $\forall i, j \quad \omega(E_i \,\hat{}{*}\, E_j) = \omega(\texttt{when}\ ((E_i\ \texttt{when}\hat{}E_j) == (E_j\ \texttt{when}\hat{}E_i)))$
   (b) The clock of any expression E_i has to be well defined: it cannot be a context clock. In particular, E_i cannot be a constant expression or a non-clocked reference to another state variable.
   (c) The clock of X is upper than the upper bound of the clocks of all the signals of the compilation unit in which X is declared.

1-e Equation of partial multiple definition

$(X_1,\ldots,X_n)$ `::=` E
$(X_1,\ldots,X_n)$ `::=` `defaultvalue` E

1. **Context-free syntax**

   DEFINITION-OF-SIGNALS ::=
   $\boxed{(}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}*** $\boxed{)}$ $\boxed{::=}$ **S-EXPR**
   | $\boxed{(}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}*** $\boxed{)}$ $\boxed{::=}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

2. **Types**
   (a) $\tau((X_1,\ldots,X_n)) = (\tau(X_1) \times \ldots \times \tau(X_n))$
   (b) $\tau(E) \sqsubseteq (\tau(X_1) \times \ldots \times \tau(X_n))$

3. **Semantics**

- $X_1, \ldots, X_n$ designate signals, or tuples of signals, or state variables (only signals or tuples of signals for `(` X_1,...,X_n `) ::= defaultvalue` E)
- This is the same generalization of 1-c and 1-d (only of 1-c for `(` X_1,...,X_n `) ::= defaultvalue` E) as that of 1-b with respect to 1-a.
- Each signal, tuple or state variable X_i is respectively partially defined by the signal or tuple v_i that corresponds to it positionally as output of E.

VI–1.2 Invocation of a model

The invocation of a model of process provides an **INSTANCE-OF-PROCESS** by *macro-expansion* of the text of the model, or by reference to this model if the text of the model is defined externally or is compiled separately.

Depending on the fact that a model:

- has or not parameters,
- has or not inputs,
- has or not outputs,

the invocation of the model can take different syntactic forms. In all cases, the composition with the context is done positionally, on the inputs and on the outputs.

If the model has no outputs, and only in this case, its invocation appears as an expression on processes (**ELEMENTARY-PROCESS**); in any other case, an invocation of model appears as an expression on signals (**S-EXPR**).

The table C–VI.1 gives the generic forms of the invocation of a model (which can be either an expression on processes or an expression on signals).

	Positional definition of the inputs	No inputs
Without parameters	P`(` E_1,...,E_n `)`	P`(  )`
With parameters	P`{` V_1,...,V_m `}(` E_1,...,E_n `)`	P`{` V_1,...,V_m `}(  )`

Table C–VI.1: Syntactic forms of an invocation of model

The different forms are detailed below.

1. **Context-free syntax**

   ELEMENTARY-PROCESS ::=

   INSTANCE-OF-PROCESS

2-a Macro-expansion of a model

One has to take care that this basic form is used here to describe the semantics of any invocation of model. The composition with the context is made by identity of names. *However, this form is not available as an external form in the language, except if the corresponding model of process does not have inputs.*

$P\{\ V_1,\ldots,V_m\ \}$

The static parameters are parenthesized by $\boxed{\{}$ and $\boxed{\}}$; these parameters are types or constant expressions mainly used as initial values of signals or array size. Note that parameters can also be models (cf. part E, section XI–8, page 187).

1. **Context-free syntax**

   INSTANCE-OF-PROCESS ::=

   EXPANSION
   | **Name-***model* $\boxed{(}\,\boxed{)}$

   EXPANSION ::=

   Name-*model*
   $\boxed{\{}$ **S-EXPR-PARAMETER {** $\boxed{,}$ **S-EXPR-PARAMETER }*** $\boxed{\}}$

   S-EXPR-PARAMETER ::=

   S-EXPR
   | **SIGNAL-TYPE**

2. **Profile**

   - $!\ (P\{\ V_1,\ldots,V_m\ \})$ is equal to the set of the names of the outputs of the visible declaration of P, let $\{Y_1,\ldots,Y_q\}$.
   - $?\ (P\{\ V_1,\ldots,V_m\ \})$ is equal to the set of the names of the inputs of the visible declaration of P, let $\{X_1,\ldots,X_p\}$.

3. **Types**

   (a) Let, in this order, $P_1,\ldots,P_l$ be the names of the formal parameters of the visible declaration of P.

   (b) The actual parameters (**S-EXPR-PARAMETER**) of the invocation of the model must correspond *positionally* to the formal parameters of the declaration of the model (cf. part E, section XI–5, page 179). In particular, to the parameter types can only correspond types (**SIGNAL-TYPE**), and to the "constant sequences" parameters can only correspond expressions on sequences (**S-EXPR**).

   (c) $(\tau(V_1) \times \ldots \times \tau(V_m)) \sqsubseteq (\tau(P_1) \times \ldots \times \tau(P_l))$

   (d) $\tau(P\{\ V_1,\ldots,V_m\ \}) = \tau(!P)$
   (cf. part E, section XI–5, page 179)

4. **Semantics**

   - P being the name of a model of visible process, the expressions $V_1,\ldots,V_m$ are the actual parameters of the expansion, corresponding *positionally* to the formal parameters of this model. The expansion $P\{\ V_1,\ldots,V_m\ \}$ is equivalent to the body of the visible declaration of P in which each formal parameter has been substituted by the corresponding actual parameter.
   - $P(\quad)$ is the expansion of P when P has no parameters.

5. **Clocks** The actual parameters of sequences V_i must be constant expressions.

   (a) $\omega(V_i) = \hbar$

2-b Positional macro-expansion of a model

$P\{\ V_1,\dots,V_m\ \}(\ E_1,\dots,E_n\ )$ or $P(\ E_1,\dots,E_n\ )$ with $n \geq 1$

In the external form of the language, the input signals are associated with an instance of model, respecting their "position": a list of expressions between the symbols $\boxed{(}$ and $\boxed{)}$ redefines the input signals declared in the model respecting the order of these declarations.

1. **Context-free syntax**

   INSTANCE-OF-PROCESS ::=

   PRODUCTION

   PRODUCTION ::=

   MODEL-REFERENCE $\boxed{(}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{)}$

   MODEL-REFERENCE ::=

   EXPANSION
   | **Name-***model*

2. **Profile**

   - $!\ (P\{\ V_1,\dots,V_m\ \}(\ E_1,\dots,E_n\ ))$ is equal to the set of the names of the outputs of the visible declaration of P, let $\{Y_1, \dots, Y_q\}$.
   - $?\ (P\{\ V_1,\dots,V_m\ \}\ (\ E_1,\dots,E_n\ )) = \bigcup_{i=1}^{n} ?\ (E_i) - \{Y_1, \dots, Y_q\}$.

3. **Types**

   (a) Let, in this order, $P_1, \dots, P_l$ be the names of the formal parameters and $X_1, \dots, X_p$ the names of the inputs of the visible declaration of P.

   (b) $(\tau(V_1) \times \dots \times \tau(V_m)) \sqsubseteq (\tau(P_1) \times \dots \times \tau(P_l))$

   (c) $(\tau(E_1) \times \dots \times \tau(E_n)) \sqsubseteq (\tau(X_1) \times \dots \times \tau(X_p))$

   (d) $\tau(P\{\ V_1,\dots,V_m\ \}(\ E_1,\dots,E_n\ )) = \tau(!P)$
   (cf. part E, section XI–5, page 179)

4. **Semantics**
   The form $P(\ E_1,\dots,E_n\ )$ is used when P has no parameters.

5. **Definition in SIGNAL**
   $P\{\ V_1,\dots,V_m\ \}(\ E_1,\dots,E_n\ )$
   is equal to the process defined below in which $\{SX_i\}$ is a set of signal names that do not belong to the inputs of the expressions E_i $(\bigcup_{i=1}^{n} ?\ (E_i))$, or to the sets of input or output names of P.

```
(| (SX_1,...,SX_p) := (E_1,...,E_n)
 |  (| (X_1,...,X_p) := (SX_1,...,SX_p)
     | P{ V_1,...,V_m }
     |) / X_1, ..., X_p

 |) / SX_1, ..., SX_p
```

6. **Clocks** The actual parameters of sequences V_i must be constant expressions.

   (a) $\omega(V_i) = \hbar$

2-c Call of a model

$(SS_1,\ldots,SS_r)$:= $P\{\ V_1,\ldots,V_m\ \}(\ E_1,\ldots,E_n\ )$
(the form $P\{\ V_1,\ldots,V_m\ \}(\ E_1,\ldots,E_n\ )$ is used here generically to represent one of the forms defined in 2-a or in 2-b; moreover, it can also appear as argument of any expression on signals)

1. **Context-free syntax**

   S-EXPR ::=

   INSTANCE-OF-PROCESS

2. **Definition in SIGNAL**
   $(SS_1,\ldots,SS_r)$:= $P\{\ V_1,\ldots,V_m\ \}(\ E_1,\ldots,E_n\ )$, with the model P having the output signals $\{Y_1,\ldots,Y_q\}$, is equal to the process defined below in which $\{SY_i\}$ is a set of signal names that do not belong to the inputs of the expressions E_i ($\bigcup_{i=1}^{n} ?\,(E_i)$), or to the sets of input or output names of P, or to the set $\{SS_1,\ldots,SS_r\}$.

```
(| (SS_1,...,SS_r) := (SY_1,...,SY_q)
 |  (| P{ V_1,...,V_m }( E_1,...,E_n )
     | (SY_1,...,SY_q) := (Y_1,...,Y_q)
     |) / Y_1, ..., Y_q

 |) / SY_1, ..., SY_q
```

The table C–VI.2 gives the different forms of the invocation of a model together with the priority of their arguments (refer to the tables C–VI.3 and C–VI.4).

2-d Expressions of type conversion

$T(\ E\ )$

The conversions of values between distinct effective types can be explicited as call of a model (**INSTANCE-OF-PROCESS**); the name of this model is the name of the destination type of the conversion; the expressions of conversion can only appear as expressions on signals, but not as expressions on processes.

1. **Context-free syntax**

Scheme	Type: Arguments	→ Result
$P\{ V_1^0,\ldots,V_m^0 \}(E_1^0,\ldots,E_n^0)$	$(\mu_1 \times \ldots \times \mu_m) \times (\nu_1 \times \ldots \times \nu_n)$	$\rightarrow (\rho_1 \times \ldots \times \rho_p)$
$P\{ V_1^0,\ldots,V_m^0 \}(\;)$	$(\mu_1 \times \ldots \times \mu_m) \times (\nu_1 \times \ldots \times \nu_n)$	$\rightarrow (\rho_1 \times \ldots \times \rho_p)$
$P\{ V_1^0,\ldots,V_m^0 \}$	$(\mu_1 \times \ldots \times \mu_m) \times (\nu_1 \times \ldots \times \nu_n)$	$\rightarrow (\rho_1 \times \ldots \times \rho_p)$
$P(E_1^0,\ldots,E_n^0)$	$(\nu_1 \times \ldots \times \nu_n)$	$\rightarrow (\rho_1 \times \ldots \times \rho_p)$
$P(\;)$	$(\nu_1 \times \ldots \times \nu_n)$	$\rightarrow (\rho_1 \times \ldots \times \rho_p)$

Table C–VI.2: **INSTANCE-OF-PROCESS** E^{25}

- When the inputs E_i are absent, it is a model without input (the tuple $((\nu_1 \times \ldots \times \nu_n))$ is then empty);
- When the model has at least one input, the types $\nu'_1, \ldots, \nu'_p$ being, in this order, those of the declaration of the inputs of P, there is $(\nu_1 \times \ldots \times \nu_n) \sqsubseteq (\nu'_1 \times \ldots \times \nu'_p)$
- The type ρ_i is that of the signal declaration corresponding positionally in output in P.

S-EXPR ::=

CONVERSION

CONVERSION ::=

Type-conversion $\boxed{(}$ **S-EXPR** $\boxed{)}$

Type-conversion ::=

Scalar-type
| **Name-***type*

2. **Types**

   (a) If the conversion $\mathcal{C}_{\tau(T)}^{\tau(E)}$ exists,
   $\tau(T(\ E\ )) = \tau(T)$

   (b) If the conversion $\mathcal{C}_{\tau(T)}^{\tau(E)}$ does not exist, $T(\ E\ )$ is incorrect.

3. **Semantics**

   - If v is an element of the sequence of values represented by E, the corresponding element is $\mathcal{C}_{\tau(T)}^{\tau(E)}(v)$ in the sequence represented by $T(\ E\ )$ (if the conversion $\mathcal{C}_{\tau(T)}^{\tau(E)}$ exists).
   - If the type T or the type of E is an external type, the applied conversion, when it exists, depends on the environment while respecting the general rules concerning conversions (cf. section V–6.3, page 81).

4. **Clocks** A conversion is a monochronous expression.

   (a) $\omega(T(\ E\ )) = \omega(E)$

5. **Examples**

   (a) `integer(3.14)` has the value 3.

VI–1.3 Nesting of expressions on signals

The expressions on signals can be nested in the respect of the priorities of the operators: any expression with lower priority than the expression of which it is an argument must be parenthesized. Parenthesizing is possible but not necessary in the other cases. Non parenthesized expressions which contain operators with the same priority are evaluated from left to right, unless it is explicitly mentioned.

1. **Context-free syntax**

   S-EXPR ::=

   $\boxed{(}$ **S-EXPR** $\boxed{)}$

2. **Profile**
   The expressions **S-EXPR** do not return a named output; their inputs are the set obtained by the union of the sets of inputs of their operands.

3. **Semantics**
   In the respect of the rules of priority, an equation S `:=:` $T(E_1,\ldots,E_n)$ formed by a function (or an operator) and sub-expressions $E_1,\ldots,E_n$ is equal to the composition

   - of the equations calculating these expressions in auxiliary variables:
     `(`$X_{i,1},\ldots,X_{i,m_i}$`) :=:` E_i
   - of the equation S `:=:` $T(X_{1,1},\ldots,X_{n,n_m})$ equal to the equation S `:=:` $T(E_1,\ldots,E_n)$ in which has been substituted, to each expression E_i, the tuple $(X_{i,1},\ldots,X_{i,m_i})$ of the auxiliary variables in which it is evaluated,
   - and of the clock equations depending on the context of each one of these expressions.

Priorities and types of the operators on signals The tables C–VI.3 and C–VI.4 contain a summary of the properties of expressions on signals. In these tables:

- the priorities are described in the first column (priority of the expression) and the second column (priorities of its arguments) by using E^i to describe an expression of priority i; the expressions are evaluated in the decreasing order of priorities;
- the third column describes the types of the arguments and of the result:
  - any_i represents any type (however, one must refer to the definition of the operators for a more precise description)
  - $bool_i$ is the type $boolean$ or $event$
  - $compar_i$ is any type in which there exists a partial order
  - int_i is an integer type (i.e., among $short$, $integer$, $long$)
  - $real_i$ is a real type (i.e., among $real$, $dreal$)
  - $cmplx_i$ is a complex type (i.e., among $complex$, $dcomplex$)
  - num_i is a numeric type (i.e., among int_i, $real_i$, $cmplx_i$);

Priority	Scheme	Type: Arguments	→ Result	
E^0	$\hat{}0$		$event$	
E^1	E^1 `next` E^2	$([0..n_1] \times \ldots \times [0..n_p]) \to any_1 \times$ $([0..m_1] \times \ldots \times [0..m_p]) \to any_2$	$\longrightarrow ([0..n_1] \times \ldots \times [0..n_p]) \to any_1 \sqcup any_2$	a
E^2	E^3 `:` E^3	$([0..l_1] \times \ldots \times [0..l_p]) \to {int_1}^n \times$ $([0..m_1] \times \ldots \times [0..m_p]) \to any_1$	$\longrightarrow ([0..r_1] \times \ldots \times [0..r_n]) \to any_1$	
E^3	E^3 `default` E^4	$any_1 \times any_2$	$\to any_1 \sqcup any_2$	a
E^4	E^4 `when` E^5	$any_1 \times bool_1$	$\to any_1$	
E^5	E^6 `after` E^6	$event \times event$	$\to integer$	
	E^6 `from` E^6			
	E^6 `count` E^6			
E^6	$E^6 \hat{}+ E^7$, $E^6 \hat{}- E^7$	$any_1 \times any_2$		
E^7	$E^7 \hat{}* E^8$		$\to event$	
E^8	`when` E^8	$bool_1$		
E^9	`if` E^0 `then` E^0 `else` E^9	$bool_1 \times any_1 \times any_2$	$\to any_1 \sqcup any_2$	a
E^{10}	E^{11} `..` E^{11} `step` E^{11}	$int_1 \times int_2 \times int_3$	$\to [0..n] \to int_1 \sqcup int_2$	
	E^{11} `..` E^{11}	$int_1 \times int_2$	$\to [0..n] \to int_1 \sqcup int_2$	
E^{11}	E^{11} `xor` E^{12}	$bool_1 \times bool_2$	$\to bool_1 \sqcup bool_2$	
E^{12}	E^{12} `or` E^{13}			
E^{13}	E^{13} `and` E^{14}			
E^{14}	`not` E^{14}	$bool_1$		
E^{15}	E^{16} `==` E^{16}	$any_1 \times any_2$		a
	E^{16} `«=` E^{16}	$compar_1 \times compar_2$	$\to boolean$	a
E^{16}	E^{17} **Op** E^{17}	$any_1 \times any_2$	$\to boolean$	a, b
		$compar_1 \times compar_2$		a, c
E^{17}	E^{17} `+` E^{18}, $E^{17} - E^{18}$	$num_1 \times num_2$	$\to num_1 \sqcup num_2$	
	$E^{17} \mid\!+ E^{18}$	$[0..m_1] \to any_1 \times [0..m_2] \to any_2$	$\longrightarrow [0..m_1 + m_2 + 1] \to any_1 \sqcup any_2$	a
E^{18}	$E^{18} * E^{19}$, E^{18} `/` E^{19}	$num_1 \times num_2$	$\to num_1 \sqcup num_2$	
	$E^{18} \mid\!* E^{19}$	$any_1 \times int_1$	$\to [0..m] \to any_1$	
	E^{18} `modulo` E^{19}	$int_1 \times int_2$	$\to int_2$	
	$E^{18} *. E^{19}$			d
E^{19}	$E^{20} ** E^{20}$	$num_1 \times int_1$	$\to num_1$	
	E^{20} @ E^{20}	$real_1 \times real_2$	$\to cmplx_1$	e
E^{20}	`+` E^{21}, $- E^{21}$	num_1	$\to num_1$	
E^{21}	`var` E^{22} `init` E^{22}	$any_1 \times any_2$	$\to any_1$	f
	`var` E^{22}	any_1		
	E^{21} `cell` E^{22} `init` E^{22}	$any_1 \times bool_1 \times any_2$	$\to any_1$	f
	E^{21} `cell` E^{22}	$any_1 \times bool_1$		
	S-EXPR-*DYNAMIC*			C–VI.6

Table C–VI.3: Expressions on signals

Priority	Scheme	Type		
		Arguments	$\longrightarrow$ Result	
E^{22}	`tr` E^{22}	$([0..l] \times [0..m]) \rightarrow any_1$	$\longrightarrow ([0..m] \times [0..l]) \rightarrow any_1$	
E^{23}	E^{24}`\\`E^{24}	$any_1 \times any_2$	$\longrightarrow any_1 \sqcup any_2$	a
E^{24}	`^`E^{24}	any_1		
E^{25}	$\ll E^0,\ldots,E^0 \gg$	$[0..m_1-1] \rightarrow any_1 \times \ldots \times [0..m_n-1] \rightarrow any_n$	$\longrightarrow [0..\prod_{k=1}^{n} m_k - 1] \rightarrow any_1 \times \ldots \times [0..\prod_{k=1}^{n} m_k - 1] \rightarrow any_n$	
	`[`$E^0,\ldots,E^0$`]`	$any_1 \times \ldots \times any_n$	$\longrightarrow [0..n-1] \rightarrow \bigsqcup_{i=1}^{n} any_i$	a
	INSTANCE-OF-PROCESS			C–VI.2
	`T(` E^0 `)`	any_1	$\longrightarrow \mathcal{T}(\mathrm{T})$	h
E^{26}	E^{26}`[`$E^0,\ldots,E^0$`]`	$(([0..n_1] \times \ldots \times [0..n_m]) \rightarrow any_1) \times (int_1 \times \ldots \times int_m)$	$\longrightarrow any_1$	
		$([0..l_1] \times \ldots \times [0..l_n]) \rightarrow any_1 \times ([0..m_1] \times \ldots \times [0..m_p]) \rightarrow {int_1}^n$	$\longrightarrow ([0..m_1] \times \ldots \times [0..m_p]) \rightarrow any_1$	
	E^{26}`.`X_i	$bundle(\{X_1\} \rightarrow any_1 \times \ldots \times \{X_m\} \rightarrow any_m)$	$\longrightarrow any_i$	
E^{27}	`(`$E^0,\ldots,E^0$`)`	$any_1 \times \ldots \times any_n$	$\longrightarrow (any_1 \times \ldots \times any_n)$	
	CONSTANT			C–VI.5
	`Id`	$\mathcal{T}(\mathrm{Id})$		i
	`(` E^0 `)`	$\mathcal{T}(\mathrm{E})$		

Table C–VI.4: Expressions on signals

[a] for types belonging to the same domain

[b] for **Op** $=$ or $/=$

[c] for **Op** $<=$ or $>=$ or $<$ or $>$, a partial order being defined in the type $compar$

[d] matrix products

[e] $cmplx_1$ is of type $complex$ if both arguments are of type $real$, it is of type $dcomplex$ otherwise

[f] for $any_2 \sqsubseteq any_1$

[g] Iterative enumeration

[h] Conversion

[i] $\mathcal{T}(\mathrm{Id})$ is the type of the declaration of the signal identifier `Id`

when, on a same line, two notations de type have the same index, the they designate the same type;

- the last column is a reference to the notes that follow the table (lowercase letter) or a reference to another table.

VI–2 Elementary expressions

The expressions of elementary signals are the following:

1. **Context-free syntax**

S-EXPR-*ELEMENTARY* ::=

  CONSTANT
| **Name**-*signal*
| **Label**
| **Name**-*state-variable* $\boxed{\textbf{?}}$

VI–2.1 Constant expressions

A constant expression is a **CONSTANT**, an occurrence of constant identifier, an occurrence of parameter identifier, a constant expression of tuple (cf. part D, section VIII–1, page 143), a constant expression of array (cf. part D, section IX–2, page 149), or one of the following expressions having recursively as arguments constant expressions:

- an **INSTANCE-OF-PROCESS** (only if it is the call of a monochronous function with constant arguments), or a **CONVERSION**,
- an **S-EXPR**-*TEMPORAL*
- an **S-EXPR**-*BOOLEAN*,
- an **S-EXPR**-*ARITHMETIC*.

Clock expressions (**S-EXPR**-*CLOCK*) and dynamic expressions (**S-EXPR**-*DYNAMIC*) cannot be part of a constant expression.

A constant is a denotation of value of a **Scalar-type** or of an **ENUMERATED-TYPE**:

1. **Context-free syntax**

CONSTANT ::=

  Boolean-cst
| **Integer-cst**
| **Real-cst**
| **Character-cst**
| **String-cst**
| **ENUM-CST**

These syntactic categories are described elsewhere (cf. part A, section II–2, page 23).

1. **Profile**
   A constant and consequently a constant expression have neither named input, nor named output.
2. **Types**
   (a) The type of a constant expression is evaluated in accordance with the type of the **S-EXPR** having the same syntax.
3. **Clocks**
   (a) The clock of a constant expression and of its arguments is $\hbar$.

The table C–VI.5 contains a summary of these properties and gives the priority of the constant lexical expressions.

Scheme	Type
`true`	$event$
`false`	$boolean$
Integer-cst	**Integer-type** following its value
Simple-precision-real-cst	$real$
Double-precision-real-cst	$dreal$
Character-cst	$character$
String-cst	$string$

Table C–VI.5: Types of the constants E^{27}

VI–2.2 Occurrence of signal or tuple identifier

An occurrence of signal identifier has as value the signal that defines this identifier, as clock, the clock of this signal and as type the type of its most internal declaration; the profile which is associated with it contains as input this single identifier and does not contain a named output.

An occurrence of tuple identifier has as value the tuple of the signals that define this identifier.

In the rules describing the context-free syntax of the language, **Name-***signal* can designate, following the context, a signal name, a tuple name, or a field name in a tuple.

The occurrence of a label is more specifically described in chapter VII, section VII–5, page 134.

VI–2.3 Occurrence of state variable

The notation X`?` allows to refer to the last defined value of a state variable X (cf. section V–10, page 88). State variables can be defined exclusively by equations of partial definition, that define the next values of the state variable (cf. section VI–1.1, paragraph 1-d, page 91). For a declared state variable X, the direct reference to X is not allowed in expressions on signals; the only way to refer to the last defined value of the state variable is by using the notation X`?`. The notation X`?` designates the value of the state variable X at the beginning of the "current step". Moreover, this notation must be used in a context in which a context clock is well defined.

X`?`

1. **Types**

   (a) $\tau(X\texttt{?}) = \tau(X)$

2. **Definition in SIGNAL**

   Let H be the context clock of X`?`, then, with the definition of X as it is given in section section VI–1.1, paragraph 1-d, page 91, X`?` is equivalent to:
   X `when` H

3. **Clocks**

   (a) The clock of X`?`, which is equal to the clock of X, is upper than the upper bound of the clocks of all the signals of the compilation unit in which X is declared.

VI–3 Dynamic expressions

Dynamic expressions allow the handling of values of signals having distinct dates. They require the definition of the value of the signals at their initial instants.

1. **Context-free syntax**

S-EXPR-*DYNAMIC* ::=

  SIMPLE-DELAY
| **WINDOW**
| **GENERALIZED-DELAY**

The table C–VI.6 gives the different forms of dynamic expressions.

Scheme	Type: Arguments	→ Result
E^{21} `window` E^{22} `init` E^{22}	$A_1 \times E_1 \times W_1$	$\rightarrow W_2$
E^{21} `window` E^{22}	$A_1 \times E_1$	$\rightarrow W_2$
E^{21} `$` E^{22} `init` E^{22}	$A_1 \times E_{11} \times W_{11}$	$\rightarrow A_1$
E^{21} `$` `init` E^{22}	$A_1 \times A_2$	$\rightarrow A_1$
E^{21} `$` E^{22}	$A_1 \times E_{11}$	$\rightarrow A_1$
E^{21} `$`	A_1	$\rightarrow A_1$

Table C–VI.6: **S-EXPR**-*DYNAMIC* E^{21}

A_1 any_1
E_1 constant M of **Integer-type**, strictly positive
W_1 $[0..M-2] \rightarrow A_2$
W_2 $[0..M-1] \rightarrow A_1$
E_{11} signal i of **Integer-type**, positive or zero, bounded by a constant N, of implicit value 1
W_{11} $[0..N-1] \rightarrow A_2$
A_i $A_2 \sqsubseteq A_1$

VI–3.1 Initialization expression

E `init` V

The initialization expression allows to define the initial value(s) of a signal.

1. **Types**

   (a) E is a signal of any type.

   (b) The type of V can be, depending on the context of the initialization:

   - a type ν such that $\nu \sqsubseteq \tau(E)$,
   - a type $[0..n-1] \rightarrow \nu$ such that $\nu \sqsubseteq \tau(E)$.

2. **Semantics**

   - If V has a type ν such that $\nu \sqsubseteq \tau(E)$, the value of V defines an initial value for the expression E `init` V.
   - If V has a type $[0..n-1] \rightarrow \nu$ such that $\nu \sqsubseteq \tau(E)$, then the value of V defines n initial values for the expression E `init` V: the value $\varphi(V\texttt{[0]})$ defines the value of this expression at its first instant, the value $\varphi(V\texttt{[1]})$ defines the value of the expression at its second instant, etc.

   If V defines more values than required by the initialization of the expression E, the extra values are not taken into account.
   If V defines less values than required by the initialization of the expression E, the missing values are defined by the default initial value of type ν.
   An initialization expression can be associated with a signal either in an expression on signals, as it is the case here, or in the declaration of a signal (cf. section V–9, page 87). When both forms of initialization are defined for a same signal, the one which has the priority is that appearing in the expression of definition of the signal. The presence of an initialization expression in the definition of a signal specifies, with the same semantics as above, a *default* initialization for the signal, when no initialization is specified in its expression of definition. For a state variable (cf. section V–10, page 88), it is recommended that its initialization is described in its declaration, and not in its expressions of definition.
   When several initialization expressions are associated with a signal in different partial definitions, they should be compatible.

3. **Clocks**

   (a) $\omega(E \texttt{ init } V) = \omega(E)$
   (b) $\omega(V) = \hbar$

VI–3.2 Simple delay

E `$ init` v_0

1. **Context-free syntax**

   SIMPLE-DELAY ::=
      S-EXPR $\boxed{\textbf{\$}}$ **[** $\boxed{\textbf{init}}$ **S-EXPR]**

2. **Types**

   (a) E is a signal of any type.
   (b) $\tau(E \texttt{ \$ init } v_0) = \tau(E)$
   (c) $\tau(v_0) \sqsubseteq \tau(E)$

3. **Semantics**
   The semantics of the delay is described formally in part B, section III–6.2, page 39.

   The value of the signal E `$ init` v_0 is at each instant t the value of the delayed signal E at the instant $t-1$. Initially, this value is the value defined by the initialization ($\varphi(v_0)$).

4. **Definition in SIGNAL**
When the initial value is omitted, it is equal to the "null" value of type $\tau(E)$ (*which implies that it is defined for any type, including external one*), $0_{\tau(E)}$:
E `$` = E `$ init` $0_{\tau(E)}$,
except if an initial value is associated with the defined signal, in its declaration (cf. section VI–3.1, page 107).

5. **Clocks**

   (a) $\omega(v_0) = \hbar$

   (b) $\omega(E$ `$ init` $v_0) = \omega(E)$

6. **Examples**

   (a) the values taken by `y` for `y` defined by `y := x $ init 0` are described below for the corresponding values of `x` in input:

$$
\begin{array}{ccccccccccc}
\texttt{x} & = & 1 & & 2 & & 3 & & 4 & & \ldots \\
 & & & \searrow & & \searrow & & \searrow & & \searrow & \\
\texttt{y} & = & 0 & & 1 & & 2 & & 3 & & \ldots
\end{array}
$$

VI–3.3 Sliding window

E `window` M `init` TE_0

1. **Context-free syntax**

   WINDOW ::=
   S-EXPR $\boxed{\textbf{window}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

2. **Types**

   (a) E is a signal of any type.

   (b) The size of the window, M, is an integer constant expression the value of which is greater than or equal to 1. If it is equal to 1, the initialization has no effect.

   (c) $\tau(E$ `window` M `init` $TE_0) = [0..\varphi(M) - 1] \rightarrow \tau(E)$

   (d) $\tau(TE_0) = [0..n - 1] \rightarrow \mu$,
   where $\mu \sqsubseteq \tau(E)$, $n \geq \varphi(M) - 1$, and $n > 0$
   (in the particular case where $\varphi(M) = 2$, the single initialization value can be given by an element of type $\tau(TE_0) = \mu$, where $\mu \sqsubseteq \tau(E)$)

3. **Semantics**
For a signal X defined by X `:=` E `window` M `init` TE_0:

   - $(t + i \geq \varphi(M)) \quad \Rightarrow \quad (X_t[i] = E_{t-\varphi(M)+i+1})$
   - $(1 \leq t + i < \varphi(M)) \quad \Rightarrow \quad (X_t[i] = TE_0[\text{t-}\varphi(M) + i + 2])$

4. **Definition in SIGNAL**
   X `:=` E `window` M `init` TE_0
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of sliding window, is equal to the process defined as follows, when $\varphi(M) > 1$:

```
(| X_0 := E
 | X_1 := X_0 $ init TE_0[M - 2]
   ⋮
 | X_{M-1} := X_{M-2} $ init TE_0[0]
 | X := [ X_{M-1}, ..., X_0 ]
 |) / X_0, ..., X_{M-1}
```

5. **Definition in SIGNAL**
   E `window` M is equal, when $\varphi(M) > 1$, to the following expression on signals:

   E `window` M `init` $0_{[0..\varphi(M)-2] \to \tau(E)}$

6. **Definition in SIGNAL**
   X `:=` E `window` 1 is equal to the process defined as follows:

   X `:= [` E `]`

7. **Clocks**
   (a) $\omega(M) = \hbar$
   (b) $\omega(TE_0) = \hbar$
   (c) $\omega($ E `window` M `init` $TE_0) = \omega(E)$

8. **Examples**
   (a) the values taken by `y` for `y` defined by `y := x window 3 init [ -1,0 ]` are described below for the corresponding values of `x` in input:

`x`	=	1	2	3	4	…
`y`	=	`[` $-1, 0, 1$ `]`	`[` $0, 1, 2$ `]`	`[` $1, 2, 3$ `]`	`[` $2, 3, 4$ `]`	…

VI–3.4 Generalized delay

E `$` I `init` TE_0

1. **Context-free syntax**

   GENERALIZED-DELAY ::=
   S-EXPR $\boxed{\textbf{\$}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

2. **Types**
   (a) E is a signal of any type.
   (b) I is a positive or equal to zero integer, with an upper bound.
   Let N be the upper bound (if I is an integer constant, N is equal to I).
   (c) $\tau(E \texttt{ \$ } I \texttt{ init } TE_0) = \tau(E)$
   (d) $\tau(TE_0) = [0..n-1] \rightarrow \mu$,
   where $\mu \sqsubseteq \tau(E)$, $n \geq \varphi(N)$, and $n > 0$
   (in the particular case where $\varphi(N) = 1$, the single initialization value can be given by an element of type $\tau(TE_0) = \mu$, where $\mu \sqsubseteq \tau(E)$)

3. **Definition in SIGNAL**
   X `:=` E `$` I `init` TE_0
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of generalized delay bounded by the maximal value N, is equal to the process defined as follows:

```
(| TX := E window N+1 init TE0
 | X := TX[N – I]
 |) / TX
```

4. **Definition in SIGNAL**
   X `:=` E `$` I
   is equal to the process defined as follws:

```
(| TX := E window N+1
 | X := TX[N – I]
 |) / TX
```

5. **Clocks**
   (a) $\omega(I) = \omega(E)$
   (b) $\omega(TE_0) = \hbar$
   (c) $\omega(E \texttt{ \$ } I) = \omega(E)$

6. **Examples**
   (a) the values taken by `y` for `y` defined by `y := x $ 3 init [ -2,-1,0 ]` are described below for the corresponding values of `x` in input:

`x`	=	1	2	3	4	5	6	…
`y`	=	–2	–1	0	1	2	3	…

   (b) the values taken by `y` for `y` defined by `y := x $ i init [ -2,-1,0 ]` are described below for the corresponding values of `x` and `i` in input:

`i`	=	1	3	3	1	2	1	…
`x`	=	1	2	3	4	5	6	…
`y`	=	0	–1	0	3	3	5	…

VI–4 Polychronous expressions

The polychronous expressions are built on signals which have possibly different clocks.

1. **Context-free syntax**

   S-EXPR-*TEMPORAL* ::=

   MERGING
   | **EXTRACTION**
   | **MEMORIZATION**
   | **VARIABLE**
   | **COUNTER**

VI–4.1 Merging

E_1 `default` E_2

1. **Context-free syntax**

   MERGING ::=

   S-EXPR $\boxed{\textbf{default}}$ **S-EXPR**

2. **Types**

   (a) $\tau(E_1)$ and $\tau(E_2)$ are signals of a same domain.

   (b) $\tau(E_1 \texttt{ default } E_2) = \tau(E_1) \sqcup \tau(E_2)$

3. **Semantics**
   The semantics is described formally in part B, section III–6.3, page 40.

4. **Clocks**

   (a) $\omega(E_1 \texttt{ default } E_2) = \omega(E_1) + ((1 - \omega(E_1)) * \omega(E_2))$ if $\omega(E_2) \neq \hbar$

   (b) $\omega(E_1 \texttt{ default } E_2) = \omega(E_1) + ((1 - \omega(E_1)) * \omega(E_1 \texttt{ default } E_2))$ if $\omega(E_2) = \hbar$

5. **Graph**
   When $\tau(E_1 \texttt{ default } E_2) \neq boolean$ and $\tau(E_1 \texttt{ default } E_2) \neq event$:

   (a) $E_1 \rightarrow E_1 \texttt{ default } E_2$

   (b) $E_2 \xrightarrow{1 - \omega(E_1)} E_1 \texttt{ default } E_2$

6. **Properties**

   (a) $(E_1 \texttt{ default } E_2) \texttt{ default } E_3 = E_1 \texttt{ default } (E_2 \texttt{ default } E_3)$

   (b) $E_1 \texttt{ default } E_2 = E_1 \texttt{ default } (E_2 \texttt{ when not}\hat{\ }E_1 \texttt{ default}\hat{\ }E_2)$

   (c) $(\omega(E_1) * \omega(E_2) = \widehat{0}) \quad \Rightarrow \quad (E_1 \texttt{ default } E_2 = E_2 \texttt{ default } E_1)$

   (d) $((\omega(E_1) \geq \omega(E_2)) \bigvee (\omega(E_1) = \hbar)) \quad \Rightarrow \quad (E_1 \texttt{ default } E_2 = E_1)$

7. **Examples**

   (a) the values taken by `Y` defined by `Y := E1 default E2` are described below for the corresponding values of `E1` and `E2` in input:

`E1`	=	1	3	$\perp$	5	7	...
`E2`	=	2	4	6	$\perp$	8	...
`Y`	=	1	3	6	5	7	...

VI–4.2 Extraction

E `when` B

The values of a signal can be produced by extraction of the values of another signal when the values of a Boolean signal are equal to $true$.

1. **Context-free syntax**

   EXTRACTION ::=

   S-EXPR $\boxed{\textbf{when}}$ **S-EXPR**

2. **Types**

   (a) E is a signal of any type.

   (b) $\tau(B) \sqsubseteq boolean$

   (c) $\tau(E \texttt{ when } B) = \tau(E)$

3. **Semantics**
   The semantics is described formally in part B, section III–6.3, page 40.

4. **Clocks**

   (a) $\omega(E \texttt{ when } B) = \omega(E) * \omega(B) * (-1 - B) \quad$ if $\omega(E) \neq \hbar$

   (b) $\omega(E \texttt{ when } B) = \omega(B) * (-1 - B) \quad$ if $\omega(E) = \hbar$

5. **Graph**
   When $\tau(E \texttt{ when } B) \neq boolean$ and $\tau(E \texttt{ when } B) \neq event$:

   (a) $E \rightarrow E \texttt{ when } B$

   (b) $B \rightarrow \omega(E \texttt{ when } B)$ when B is a free condition

6. **Properties**

   (a) $(\tau(B) = event) \quad \Rightarrow \quad (B \texttt{ when } B = B)$

   (b) $(E \texttt{ when } B_1) \texttt{ when } B_2 = E \texttt{ when } (B_1 \texttt{ when } B_2)$

   (c) $E \texttt{ when } (B \texttt{ when } B) = E \texttt{ when } B$

7. **Examples**

   (a) the values taken by `X when C` are described below for the corresponding values of `X` and `C` in input:

`X`	=	1	3	$\perp$	5	$\perp$	7	...
`C`	=	T	$\perp$	T	F	F	T	...
`X when C`	=	1	$\perp$	$\perp$	$\perp$	$\perp$	7	...

VI–4.3 Memorization

E `cell` B `init` V_0

The memorization allows to memorize a given signal at the clock defined by the upper bound of the clock of the signal and the clock defined by the instants at which a Boolean signal has the value $true$.

1. **Context-free syntax**

   MEMORIZATION ::=

   S-EXPR $\boxed{\textbf{cell}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

2. **Types**

   (a) E is a signal of any type.

   (b) $\tau(B) \sqsubseteq boolean$

   (c) $\tau(E$ `cell` B `init` $V_0) = \tau(E)$

   (d) $\tau(V_0) \sqsubseteq \tau(E)$

3. **Definition in SIGNAL**
   X `:=` E `cell` B `init` V_0
   whose right side of $\boxed{\textbf{:=}}$ represents an expression of memorization of E at the instants at which B is $true$, is equal to the process defined as follows:

   ```
   (| X := E default (X $ init V0)
    | X ^= E ^+ (when B)
    |)
   ```

4. **Definition in SIGNAL**
   When the initial value is omitted, it is equal to the "null" value of type $\tau(E)$, $0_{\tau(E)}$:
   E `cell` B = E `cell` B `init` $0_{\tau(E)}$,
   except if an initial value is associated with the defined signal, in its declaration (cf. section VI–3.1, page 107).

5. **Clocks**

   (a) $\omega(E \texttt{ cell } B \texttt{ init } V_0) = \omega(E) + ((1 - \omega(E)) * \omega(B) * (-1 - B))$

6. **Examples**

   (a) the values taken by `X cell C init 0` are described below for the corresponding values of `X` and `C` in input:

`X`	=	$\perp$	1	3	$\perp$	$\perp$	$\perp$	5	$\perp$	7	...
`C`	=	T	F	T	T	F	T	$\perp$	T	$\perp$	...
`X cell C init 0`	=	0	1	3	3	$\perp$	3	5	5	7	...

VI–4.4 Variable clock signal

`var` E `init` V_0

The `var` operator allows to use a signal at any clock defined by the context.

1. **Context-free syntax**

   VARIABLE ::=
   $\boxed{\textbf{var}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

2. **Types**

   (a) E is a signal of any type.

   (b) $\tau($`var` E `init` $V_0) = \tau(E)$

   (c) $\tau(V_0) \sqsubseteq \tau(E)$

3. **Definition in SIGNAL**
   Let:

   - F an expression on processes containing an occurrence var_i of the expression on signals `var` E `init` V_0,
   - H the context clock of var_i in F,
   - FF the expression on processes equal to F in which XX has been substituted to var_i.

   F is then equivalent to:

   ```
   (| FF
    | X := E default (X $ init V0)
    | XX := X when H
    | X ^= E ^+ H
    |) / X, XX
   ```

4. **Definition in SIGNAL**
   When the initial value is omitted, it is equal to the "null" value of type $\tau(E)$, $0_{\tau(E)}$:
   `var` E = `var` E `init` $0_{\tau(E)}$
   except if an initial value is associated with the defined signal, in its declaration (cf. section VI–3.1, page 107).

5. **Clocks**

   (a) $\omega($`var` E `init` $V_0) = \hbar$

VI–4.5 Counters

H_1 *modality* H_2 or H_1 `count` M

The counter expressions (*modality* `after` or `from`, or counter modulo: `count`) allow the numbering of the occurrences of a clock.

1. **Context-free syntax**

   COUNTER ::=
   S-EXPR $\boxed{\textbf{after}}$ **S-EXPR**
   | **S-EXPR** $\boxed{\textbf{from}}$ **S-EXPR**
   | **S-EXPR** $\boxed{\textbf{count}}$ **S-EXPR**

2. **Types**

   (a) $\tau(H_1) = \tau(H_2) = event$
   (b) M is an integer constant expression.
   (c) $\tau(H_1$ *modality* $H_2) = integer$
   (d) $\tau(H_1$ `count` $M) = integer$

3. **Definition in SIGNAL**
   N `:=` H_1 `after` H_2
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of counter of the events H_1 after the reinitialization H_2, is equal to the process defined as follows:

```
(| CN := (0 when H2) default (((N $ init 0) + 1) when H1)
 | CN ^= H1 ^+ H2
 | N := CN when H1
 |) / CN
```

   The signal N counts the number of occurrences of the signal H_1 (o_1) since the last occurrence of the signal H_2 (o_2); but the occurrences o_1 which are simultaneous to occurrences o_2 are not counted.

4. **Definition in SIGNAL**
   N `:=` H_1 `from` H_2
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of counter of the events H_1 since the reinitialization H_2, is equal to the process defined as follows:

```
(| CN := (((1 when H1) default 0) when H2) default (((N $ init 0) + 1) when H1)
 | CN ^= H1 ^+ H2
 | N := CN when H1
 |) / CN
```

   The signal N counts the number of occurrences of the signal H_1 (o_1) since the last occurrence of the signal H_2 (o_2); the occurrences o_1 which are simultaneous to occurrences o_2 are counted.

5. **Definition in SIGNAL**
   N `:=` H_1 `count` M
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of counter of the events H_1 modulo $\varphi(M)$, is

equal to the process defined as follows:

```
(| N := (0 when ZN >= (M - 1)) default (ZN + 1)
 | ZN := N $ init (M - 1)
 | N ^= H1
 |) / ZN
```

The signal N has 0 as initial value and is incremented by 1, modulo $\varphi(M)$, at each new occurrence of the signal H_1.

6. **Clocks**

   (a) $\omega(H_1 \;\; \mathit{modality} \;\; H_2) = \omega(H_1)$

   (b) $\omega(M) = \hbar$

   (c) $\omega(H_1 \;\; \texttt{count} \;\; M) = \omega(H_1)$

7. **Examples**

   (a) the values taken by `E1 from E2`, `E1 after E2` and `E1 count 3` are described below for the corresponding signals `E1` and `E2` in input:

`E1`	=	$\perp$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\perp$	$\bullet$	...
`E2`	=	$\bullet$	$\perp$	$\perp$	$\bullet$	$\perp$	$\perp$	$\bullet$	$\perp$	...
`E1 from E2`	=	$\perp$	1	2	1	2	3	$\perp$	1	...
`E1 after E2`	=	$\perp$	1	2	0	1	2	$\perp$	1	...
`E1 count 3`	=	$\perp$	0	1	2	0	1	$\perp$	2	...

VI–4.6 Properties of polychronous expressions

- $(E_1$ `default` $E_2)$ `when` B = $(E_1$ `when` $B)$ `default` $(E_2$ `when` $B)$
- $(\tau(B) = \mathit{event}) \quad \Rightarrow \quad (B$ `when` $(E_1$ `default` $E_2)$ = $(B$ `when` $E_1)$ `default` $(B$ `when` $E_1))$

VI–5 Constraints and expressions on clocks

A **CONSTRAINT** is an expression of processes which contributes to the construction of the system of clock equations of the program. It is the tool for constraint programming. Such an expression can take as arguments expressions on clocks or expressions on signals.

1. **Context-free syntax**

   ELEMENTARY-PROCESS ::=

   CONSTRAINT

VI–5.1 Expressions on clock signals

1-a Clock of a signal

$\hat{}E$

The clock of a signal (of any type) is obtained by applying the operator ^ to this signal.

1. **Context-free syntax**

   S-EXPR-*CLOCK* ::=

   SIGNAL-CLOCK

   SIGNAL-CLOCK ::=

   $\boxed{\hat{}}$ **S-EXPR**

2. **Types**

   (a) E is a signal of any type.

   (b) $\tau(\hat{}E) = event$

3. **Definition in SIGNAL**

   E == E
   Remark: this definition uses the operator of relation $==$ defined on any type (cf. section VI–7.2, page 124).

4. **Examples**

   (a) the values taken by ^X are described below for the corresponding values of X in input:

X	=	1	2	3	4	...
^X	=	T	T	T	T	...

Remark: the expression $\hat{}E$ and the conversion `event` (E) have the same result.

1-b Clock extraction

`when` B
The extraction of the $true$ values of a Boolean condition are obtained by applying the operator unary `when`:

1. **Context-free syntax**

   S-EXPR-*CLOCK* ::=

   CLOCK-EXTRACTION

   CLOCK-EXTRACTION ::=

   $\boxed{\textbf{when}}$ **S-EXPR**

2. **Types**

   (a) $\tau(B) \sqsubseteq boolean$

   (b) $\tau(\texttt{when } B) = event$

3. **Definition in SIGNAL**

   $\hat{}B$ `when` B

4. **Clocks**

   (a) $\omega(\texttt{when}\ \ B) = \omega(B) * (-1 - B)$

5. **Examples**

   (a) the values taken by `when` C are described below for the corresponding values of `C` in input:

$$\begin{array}{rcccccccc} \texttt{C} & = & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \ldots \\ \texttt{when C} & = & \mathrm{T} & \mathrm{T} & \perp & \perp & \mathrm{T} & \ldots \end{array}$$

1-c Empty clock

$\hat{}0$
The empty clock is the clock that does not "contain" any instant.

1. **Context-free syntax**

   S-EXPR-*CLOCK* ::=
   $\boxed{\hat{}0}$

2. **Types**

   (a) $\tau(\hat{}0) = event$

3. **Definition in SIGNAL**
   $\hat{}0$ is the lexical expression of the empty clock; it is equal to the solution of the following equation:
   `when not`$\hat{}0 \hat{}= \hat{}0$

4. **Clocks**

   (a) $\omega(\hat{}\texttt{0}) = \hat{0}$

VI–5.2 Operators of clock lattice

$E_1 \ \hat{}\mathbf{Op}\ \ E_2$

1. **Context-free syntax**

   S-EXPR-*CLOCK* ::=
   S-EXPR $\boxed{\hat{}+}$ **S-EXPR**
   | **S-EXPR** $\boxed{\hat{}-}$ **S-EXPR**
   | **S-EXPR** $\boxed{\hat{}*}$ **S-EXPR**

2. **Types**

   (a) E_1 and E_2 are signals of any types.
   (b) $\tau(E_1\ \ \hat{}\mathbf{Op}\ \ E_2) = event$

3. **Definition in SIGNAL**
X := $E_1 \hat{~}{+}\ E_2$
defines a signal equal to the upper bound of the clocks of the signals E_1 and E_2; this expression is equal to the process defined as follows:

```
(| X := ^E1 default ^E2
 |)
```

4. **Definition in SIGNAL**
X := $E_1 \hat{~}{*}\ E_2$
defines a signal equal to the lower bound of the clocks of the signals E_1 and E_2; this expression is equal to the process defined as follows:

```
(| X := ^E1 when ^E2
 |)
```

5. **Definition in SIGNAL**
X := $E_1 \hat{~}{-}\ E_2$
defines a signal equal to the complementary clock of $E_1 \hat{~}{*}\ E_2$ in$\hat{~}E_1$; this expression is equal to the process defined as follows:

```
(| X := when ((not ^E2) default ^E1)
 |)
```

6. **Clocks**

   (a) $\omega\big(E_1\ \hat{~}{+}\ E_2\big) = \omega\big(E_1\big) + ((1 - \omega\big(E_1\big)) * \omega\big(E_2\big))$
   (b) $\omega\big(E_1\ \hat{~}{*}\ E_2\big) = \omega\big(E_1\big) * \omega\big(E_2\big)$
   (c) $\omega\big(E_1\ \hat{~}{-}\ E_2\big) = \omega\big(E_1\big) - (\omega\big(E_1\big) * \omega\big(E_2\big))$

7. **Properties**

   (a) $E_1 \hat{~}{+} (E_2 \hat{~}{+}\ E_3) = (E_1 \hat{~}{+}\ E_2)\hat{~}{+}\ E_3$
   (b) $E_1 \hat{~}{+}\ E_2 = E_2 \hat{~}{+}\ E_1$
   (c) $E \hat{~}{+} \hat{~}0 = \hat{~}E$
   (d) $E \hat{~}{+}\ E = \hat{~}E$
   (e) $E_1 \hat{~}{*} (E_2 \hat{~}{*}\ E_3) = (E_1 \hat{~}{*}\ E_2)\hat{~}{*}\ E_3$
   (f) $E_1 \hat{~}{*}\ E_2 = E_2 \hat{~}{*}\ E_1$
   (g) $E \hat{~}{*} \hat{~}0 = \hat{~}0$
   (h) $E \hat{~}{*}\ E = \hat{~}E$
   (i) $(E_1 \hat{~}{*}\ E_2)\hat{~}{+}\ E_3 = (E_1 \hat{~}{+}\ E_3)\hat{~}{*} (E_2 \hat{~}{+}\ E_3)$
   (j) $(E_1 \hat{~}{+}\ E_2)\hat{~}{*}\ E_3 = (E_1 \hat{~}{*}\ E_3)\hat{~}{+} (E_2 \hat{~}{*}\ E_3)$

VI–5.3 Relations on clocks

E_1 $\hat{}$**Op** E_2

The following expressions are expressions on processes describing constraints between clocks of signals.

1. **Context-free syntax**

   CONSTRAINT ::=

   S-EXPR { $\boxed{\hat{}=}$ **S-EXPR }***
   | **S-EXPR {** $\boxed{\hat{}<}$ **S-EXPR }***
   | **S-EXPR {** $\boxed{\hat{}>}$ **S-EXPR }***
   | **S-EXPR {** $\boxed{\hat{}\#}$ **S-EXPR }***

2. **Profile**
   A relation on clocks of signals is a process with no output and with:
   $?\,(E_1\ \hat{}\mathbf{Op}\ \ldots\ \hat{}\mathbf{Op}\ E_n) = \bigcup_{i=1}^{n} ?\,(E_i)$.

3. **Types**

   (a) The arguments E_i are signals of any types, possibly distinct.

4. **Definition in SIGNAL**
   E_1 $\hat{}$**Op** E_2 $\hat{}$**Op** EE
   (where $\hat{}$**Op** is one of the operators $\hat{}=$, $\hat{}<$, $\hat{}>$ and $\hat{}\#$, and where EE is an expression on clocks or recursively a relation on clocks), builds the composition of the expressions E_i $\hat{}$**Op** E_j, for any pair of distinct indexes i and j, and thus expresses the conjunction of the associated relations. It is recursively defined by the composition of the following expressions of processes:

   (| E_1 $\hat{}$**Op** E_2
   | E_1 $\hat{}$**Op** EE
   | E_2 $\hat{}$**Op** EE
   |)

5. **Definition in SIGNAL**
   $E_1\ \hat{}=\ E_2$
   constrains the clock of the expression on signals E_1 to be equal to that of E_2; this expression, when $H_1 \notin ?\,(E_1\ \hat{}=\ E_2)$, is equal to the process with no output defined as follows:

   (| H_1 := ($\hat{}E_1$) == ($\hat{}E_2$)
   |) / H_1

6. **Definition in SIGNAL**
   $E_1\ \hat{}<\ E_2$
   constrains the clock of the expression on signals E_1 to be smaller than (or equal to) that of E_2; this

expression is equal to the process with no output defined as follows:
$E_1\hat{} = E_1\hat{}* E_2$

7. **Definition in SIGNAL**
$E_1 \;\hat{}> \; E_2$
constrains the clock of the expression on signals E_1 to be greater than (or equal to) that of E_2; this expression is equal to the process with no output defined as follows:
$E_1\hat{} = E_1\hat{}+ E_2$

8. **Definition in SIGNAL**
$E_1 \;\hat{}\# \; E_2$ specifies the mutual exclusion of the clocks of the expressions on signals E_1 and E_2; hence $\omega(E_1) * \omega(E_2) = \widehat{0}$. This expression is equal to the process with no output defined as follows:
$\hat{}0\hat{} = E_1\hat{}* E_2$

VI–6 Identity equations

not yet imple-mented

E_1 `:=:` E_2

Identity equations are expressions on processes describing equality constraints between the sequences of values (and clocks) of two expressions.

1. **Context-free syntax**

   CONSTRAINT ::=
   S-EXPR $\boxed{\textbf{:=:}}$ **S-EXPR**

2. **Profile**
An identity equation is a process with no output and with:
$?\,(E_1 \texttt{ :=: } E_2) = ?\,(E_1) \cup ?\,(E_2)$.

3. **Types**

   (a) E_1 and E_2 are of comparable types.

4. **Semantics**
If E_1 and E_2 can be viewed respectively as tuples $(E_{11},\ldots,E_{1n})$ and $(E_{21},\ldots,E_{2n})$, the identity equation E_1 `:=:` E_2 constrains the sequences of values of the expressions E_{1i} and E_{2i} to be respectively equal.
An equation E_1 `:=:` E_2 is the basic identity equation between signals in the language (cf. part B, chapter III, page 29). It is a non oriented equation, that does not induce dependences between E_1 and E_2.

5. **Clocks**
If E_{1i} and E_{2i} designate *signals*, they are synchronous. In this case:

   (a) $\omega(E_{1i}) = \omega(E_{2i})$

6. **Properties**

   (a) E_1 `:=:` E_2
   is equal to the following process:
   `(|` `(when (`E_{11} `==` E_{21}`))` `^=` E_{11}
   $\vdots$
   `|` `(when (`E_{1n} `==` E_{2n}`))` `^=` E_{1n}
   `|)`

VI–7 Boolean synchronous expressions

The Boolean expressions are synchronous expressions on signals. The operators defining such expressions are the standard operators on Boolean elements extended to sequences of elements. The Boolean expressions (or expressions with Boolean result) are either expressions of the Boolean lattice, or relations.

VI–7.1 Expressions on Booleans

1-a Negation

`not` E_1

1. **Context-free syntax**

   S-EXPR-*BOOLEAN* ::=
   $\boxed{\textbf{not}}$ **S-EXPR**

2. **Types**

   (a) $\tau(E_1) \sqsubseteq boolean$
   (b) $\tau($`not` $E_1) = boolean$

3. **Semantics**
   The operator of negation has, on the occurrences of signals, its usual semantics.

4. **Clocks**

   (a) $\omega($`not` $E_1) = \omega(E_1)$

1-b Operators of Boolean lattice

E_1 **Op** E_2

1. **Context-free syntax**

   S-EXPR-*BOOLEAN* ::=
   S-EXPR $\boxed{\textbf{or}}$ **S-EXPR**
   | **S-EXPR** $\boxed{\textbf{and}}$ **S-EXPR**
   | **S-EXPR** $\boxed{\textbf{xor}}$ **S-EXPR**

2. **Types**

   (a) $\tau(E_1) \sqsubseteq boolean$

   (b) $\tau(E_2) \sqsubseteq boolean$

   (c) $\tau(E_1 \ \textbf{Op} \ E_2) = boolean$

3. **Semantics**
   The expressions on Boolean signals have, on the synchronous occurrences of these signals, their usual semantics; however, they are not primitive operators of the SIGNAL language.

4. **Definition in SIGNAL**
   X `:=` E_1 `and` E_2
   is equal to the process defined as follows:

   ```
   (|  X := (E1 when E2) default  (not ^E1)
    |  E1 ^= E2
    |)
   ```

5. **Definition in SIGNAL**
   X `:=` E_1 `or` E_2
   is equal to the process defined as follows:

   ```
   (|  X := (E1 when not E2) default  ^E1
    |  E1 ^= E2
    |)
   ```

6. **Definition in SIGNAL**
   X `:=` E_1 `xor` E_2
   is equal to the process defined as follows:

   ```
   (|  X := not (E1 == E2)
    |)
   ```

7. **Clocks**

   (a) $\omega(E_1) = \omega(E_2)$

   (b) $\omega(E_1 \ \textbf{Op} \ E_2) = \omega(E_1)$

VI–7.2 Boolean relations

The Boolean relations are equality, difference, and strict and non strict greater and lower relations.

Two classes of relation operators are distinguished according to their denotation:

- the operators which have a pointwise extension on elements of arrays (cf. part D, chapter X, page 169), denoted respectively $\boxed{=}$, $\boxed{/=}$, $\boxed{>}$, $\boxed{>=}$, $\boxed{<}$ et $\boxed{<=}$; for example, the operator $\boxed{=}$ applied on two vectors has as result a vector of Booleans;
- the operators which have a Boolean result, whatever is the type of the *signals* on which they are applied; in this class are only defined the operator of equality, denoted $\boxed{==}$ and the operator of

inferior or equal relation order, denoted $\boxed{<<=}$ (these operators are pointwise extended to *families of signals:* polychronous tuples with named fields and tuples with unnamed fields).

E_1 **Op** E_2

1. **Context-free syntax**

   S-EXPR-*BOOLEAN* ::=

   RELATION

   RELATION ::=

   S-EXPR $\boxed{=}$ **S-EXPR**
   | **S-EXPR** $\boxed{/=}$ **S-EXPR**
   | **S-EXPR** $\boxed{>}$ **S-EXPR**
   | **S-EXPR** $\boxed{>=}$ **S-EXPR**
   | **S-EXPR** $\boxed{<}$ **S-EXPR**
   | **S-EXPR** $\boxed{<=}$ **S-EXPR**
   | **S-EXPR** $\boxed{==}$ **S-EXPR**
   | **S-EXPR** $\boxed{<<=}$ **S-EXPR**

2. **Types**

   (a) $\tau(E_1$ **Op** $E_2)$ = $boolean$

   (b) For E_1 $==$ E_2:
   E_1 and E_2 are *signals* of a same domain, which is any domain.

   (c) For E_1 $=$ E_2 and E_1 $/=$ E_2:
   E_1 and E_2 are signals of a same domain **Scalar-type** or **ENUMERATED-TYPE**.

   (d) For E_1 $<<=$ E_2:
   E_1 and E_2 are signals of a same domain **Scalar-type** (other than a **Complex-type**), or of **ENUMERATED-TYPE**, or of a same type for which the environment defines this operator while respecting the properties enounced in this section.

   (e) For E_1 $>$ E_2, E_1 $>=$ E_2, E_1 $<$ E_2, and E_1 $<=$ E_2:
   E_1 and E_2 are signals of a same domain **Scalar-type** (other than a **Complex-type**), or of **ENUMERATED-TYPE**.

3. **Semantics**

   - Two objects of array types are equal if and only if both arrays have the same dimension, are of comparable types and the elements of same index are respectively equal.
   - Two objects of monochronous tuple types are equal if and only if both objects are of comparable types and the elements of corresponding fields are respectively equal.
   - In the order defined on the values of type $boolean$, $false$ is lower than $true$.
   - The order defined on the values of type $character$ is the order on the decimal values of their encoding.
   - The order defined on the values of type $string$ is the corresponding lexicographic order.

- The order defined on the values of an **ENUMERATED-TYPE** is the syntactic order of their declaration in the definition of the type (cf. section V–3, page 74).

With these precisions, the operators of relation have their usual semantics. The operators $\boxed{==}$ and $\boxed{=}$ denote the relation of equality; the operators $\boxed{<<=}$ and $\boxed{<=}$ denote the relation inferior or equal.
The comparisons are made in the greatest type (of a same domain). Then if v_1 is an element of the sequence of values represented by E_1 and if v_2 is the corresponding element in the sequence of values represented by E_2,
the corresponding element is v_1 **Op** E_2 in the sequence represented by E_1 **Op** E_2.

4. **Definition in SIGNAL**
   The expression E_1 `/=` E_2 is equal to the following expression:
   `not` (E_1 `=` E_2)

5. **Definition in SIGNAL**
   The expression E_1 `<` E_2 is equal to the following expression:
   (`not` (E_1 `=` E_2)) `and` (E_1 `<=` E_2)

6. **Definition in SIGNAL**
   The expression E_1 `>=` E_2 is equal to the following expression:
   E_2 `<=` E_1

7. **Definition in SIGNAL**
   The expression E_1 `>` E_2 is equal to the following expression:
   E_2 `<` E_1

8. **Clocks**
   (a) $\omega(E_1) = \omega(E_2)$
   (b) $\omega(E_1 \;\mathbf{Op}\; E_2) = \omega(E_1)$

9. **Graph**
   When the E_i are not of a domain **Synchronization-type**:
   (a) $E_1 \rightarrow E_1 \;\mathbf{Op}\; E_2$
   (b) $E_2 \rightarrow E_1 \;\mathbf{Op}\; E_2$

10. **Properties**
    The relation $\boxed{<<=}$ is an order relation on all the types of signals for which it is defined; it has all the properties of an order relation:
    (a) reflexivity
    (b) transitivity
    (c) anti-symmetry: ((E_1 `<<=` E_2) $\bigwedge$ (E_2 `<<=` E_1)) $\Rightarrow$ (E_1 `==` E_2)

11. **Properties**
    The relation $\boxed{<=}$ is an order relation on the domains of values on which it is defined; it is:
    (a) reflexive,
    (b) transitive,
    (c) anti-symmetric: ((E_1 `<=` E_2) $\bigwedge$ (E_2 `<=` E_1)) $\Rightarrow$ (E_1 `=` E_2)

VI–8 Synchronous expressions on numeric signals

The synchronous expressions on numeric signals are defined by pointwise extension of the standard arithmetic operators on sequences of elements.

VI–8.1 Binary expressions on numeric signals

E_1 **Op** E_2

1. **Context-free syntax**

S-EXPR-*ARITHMETIC* ::=
  S-EXPR $\boxed{+}$ **S-EXPR**
| **S-EXPR** $\boxed{-}$ **S-EXPR**
| **S-EXPR** $\boxed{*}$ **S-EXPR**
| **S-EXPR** $\boxed{/}$ **S-EXPR**
| **S-EXPR** $\boxed{\textbf{modulo}}$ **S-EXPR**
| **S-EXPR** $\boxed{**}$ **S-EXPR**
| **DENOTATION-OF-COMPLEX**

2. **Semantics**
If the result of an expression cannot be represented in the type μ of this expression, its value is a value of type μ depending on the implementation.

If v_1 is an element of the sequence of values represented by E_1 and if v_2 is the corresponding element of the sequence of values represented by E_2, the corresponding element in the sequence represented by $E_1 \ \ Op \ \ E_2$ is:
v_1 **Op** v_2

3. **Clocks**
   (a) $\omega\big(E_1\big) = \omega\big(E_2\big)$
   (b) $\omega\big(E_1 \ \ \textbf{Op} \ \ E_2\big) = \omega\big(E_1\big)$

4. **Graph**
   (a) $E_1 \rightarrow E_1 \ \ \textbf{Op} \ \ E_2$
   (b) $E_2 \rightarrow E_1 \ \ \textbf{Op} \ \ E_2$

Operators $+ \, -$, $*$, $/$ $\quad E_1$ **Op** E_2

1. **Types**
   (a) $\tau\big(E_1\big)$ and $\tau\big(E_2\big)$ are of any **Numeric-type** in a same domain,
   (b) $\tau\big(E_1 \ \ \textbf{Op} \ \ E_2\big) = \tau\big(E_1\big) \sqcup \tau\big(E_2\big)$

2. **Semantics**
When an expression of division is of domain **Integer-type**, the division is the integer division.

Operator `modulo` E_1 `modulo` E_2

1. **Types**

   (a) $\tau(E_1)$ and $\tau(E_2)$ are of domain **Integer-type**.
   In addition, E_2 must be a constrained integer (strictly positive and with an upper bound).

   (b) $\tau(E_1 \texttt{ modulo } E_2) = \tau(E_2)$

2. **Semantics**
   If r is defined by r `:=` a `modulo` b,
   then at each instant, the following property is true:
   ($\exists$ an integer q) $\quad$ ((a=b*q+r) $\bigwedge$ ($0 \leq r < b$))

Operator $**$ $\quad E_1 \; ** \; E_2$

1. **Types**

   (a) $\tau(E_1)$ is a **Numeric-type**.

   (b) $\tau(E_2)$ is an **Integer-type**.

   (c) $\tau(E_1 \; ** \; E_2) = \tau(E_1)$

Operator `@` $\quad E_1$@E_2

A pair of synchronous elements of **Real-type** defines a signal of domain **Complex-type**.

1. **Context-free syntax**

   DENOTATION-OF-COMPLEX ::=
   S-EXPR $\boxed{\textbf{@}}$ **S-EXPR**

2. **Types**

   (a) $\tau(E_1)$ is a **Real-type**,

   (b) $\tau(E_2)$ is a **Real-type**,

   (c) if $\tau(E_1) \sqcup \tau(E_2) = real$, then $\tau(E_1@E_2) = complex$
   if $\tau(E_1) \sqcup \tau(E_2) = dreal$, then $\tau(E_1@E_2) = dcomplex$

3. **Examples**

   (a) `1.0 @ (−1.0)` defines a complex constant.

VI–8.2 Unary operators

Op E_1

1. **Context-free syntax**

   S-EXPR-*ARITHMETIC* ::=
   $\boxed{+}$ **S-EXPR**
   | $\boxed{-}$ **S-EXPR**

2. **Types**
   (a) $\tau(E_1)$ is a **Numeric-type**.
   (b) $\tau(\mathbf{Op}\ \ E_1) = \tau(E_1)$
3. **Semantics**
   If the result of an expression cannot be represented in the type μ of this expression, its value is a value of type μ depending on the implementation.

   If v_1 is an element of the sequence of values represented by E_1,
   the corresponding element in the sequence represented by **Op** E_1 is:
   Op v_1
4. **Clocks**
   (a) $\omega(\mathbf{Op}\ \ E_1) = \omega(E_1)$
5. **Graph**
   (a) $E_1 \rightarrow \mathbf{Op}\ \ E_1$

VI–9 Synchronous condition

`if` B `then` E_1 `else` E_2

The synchronous condition is an expression on signals of same clock.

1. **Context-free syntax**

   S-EXPR-*CONDITION* ::=
   $\boxed{\textbf{if}}$ **S-EXPR** $\boxed{\textbf{then}}$ **S-EXPR** $\boxed{\textbf{else}}$ **S-EXPR**

2. **Types**
   (a) $\tau(B) \sqsubseteq boolean$
   (b) E_1 and E_2 are signals of a same domain **Scalar-type**, **External-type** or **ENUMERATED-TYPE**.
   (c) $\tau($`if` B `then` E_1 `else` $E_2) = \tau(E_1) \sqcup \tau(E_2)$
3. **Definition in SIGNAL**
   X `:=` `if` B `then` E_1 `else` E_2
   whose right side of $\boxed{\textbf{:=}}$ represents an expression of synchronous condition, is equal to the process defined as follows:

   ```
   (|  X := (E1 when B) default E2
    |  B ^= E1 ^= E2
    |)
   ```
4. **Clocks**
   (a) $\omega(E_1) = \omega(E_2)$

(b) $\omega(B) = \omega(E_1)$

(c) $\omega(\texttt{if } B \texttt{ then } E_1 \texttt{ else } E_2) = \omega(E_1)$

Chapter VII

Expressions on processes

The expressions on processes allow to compose systems of equations on signals with the following syntax:

1. **Context-free syntax**

   P-EXPR ::=

     ELEMENTARY-PROCESS
   | **HIDING**
   | **LABELLED-PROCESS**
   | **GENERAL-PROCESS**

   GENERAL-PROCESS ::=

     COMPOSITION
   | **CONFINED-PROCESS**
   | **CHOICE-PROCESS**

VII–1 Elementary processes

An elementary process is an instance of process (cf. section VI–1.2, page 97), a definition of signals (cf. section VI–1.1, page 91), a constraint on clocks (cf. section VI–5, page 117) or on values (cf. section VI–6, page 122), or an expression of dependence (cf. part E, section XI–6.2, page 182).

VII–2 Composition

The composition of two processes P_1 and P_2 produces a process for which each execution observed on the variables of P_1 (respectively, P_2) is an execution of P_1 (respectively, P_2). This composition is similar to the aggregation of two systems of equations in a single one.

$P_1 \mid P_2$

1. **Context-free syntax**

   COMPOSITION ::=

   $\boxed{(|}$ **[P-EXPR {** $\boxed{|}$ **P-EXPR }*** **]** $\boxed{|)}$

2. **Profile**

   - $!\,(P_1 \mid P_2) = !\,(P_1) \cup !\,(P_2)$
   - $?\,(P_1 \mid P_2) = (?\,(P_1) - !\,(P_2)) \cup (?\,(P_2) - !\,(P_1))$

3. **Types**

   (a) If their names are identical, an output x of P_1 (respectively, P_2) and an input x of P_2 (respectively, P_1) have also the same type.

   (b) If their names are identical, an input x of P_1 and an input x of P_2 have also the same type.

4. **Semantics**
   A signal, input of P_1 (respectively, P_2), having as name the name of a signal, output of P_2 (respectively, P_1) and totally defined in it, has as definition in P_1 (respectively, in P_2) its definition in P_2 (respectively, in P_1).
   If the definitions of such a signal are partial definitions, in P_1 and in $P2$, its resulting definition is the combination of both partial definitions, as it is specified in section VI–1.1, paragraph 1-c, page 91.

5. **Clocks**

   (a) If their names are identical, an output x of P_1 (respectively, P_2) and an input x of P_2 (respectively, P_1) have also the same clock.

   (b) If their names are identical, an input x of P_1 and an input x of P_2 have also the same clock.

VII–3 Hiding

The hiding is an expression that modifies the profile of an expression of processes by hiding some of its outputs.

P `/` A_1`, ...,` A_n

1. **Context-free syntax**

   HIDING ::=
   GENERAL-PROCESS $\boxed{/}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}***
   | **HIDING** $\boxed{/}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}***

2. **Profile**

   - $?\,(P \;/\; A_1, \;\ldots,\; A_n) = ?\,(P)$
   - $!\,(P \;/\; A_1, \;\ldots,\; A_n) = !\,(P) - \{A_1, \ldots, A_n\}$

3. **Semantics**
   The hiding operation allows to hide outputs of the process P: the outputs of the resulting process are the outputs of P *which do not appear* in the list A_1`, ...,` A_n.
   The A_i can be names of tuples: in that case, the hiding applies globally on the tuples.

4. **Examples**

   Let `P` be a process with `A`, `B` and `C` as inputs and `X` and `Y` as outputs.

   (a) `P / Y` has only `X` as output;

   (b) `P / Z` is equal to `P`.

VII–4 Confining with local declarations

Local declarations can be associated with any expression of processes.

1. **Context-free syntax**

   CONFINED-PROCESS ::=

   GENERAL-PROCESS DECLARATION-BLOCK

   DECLARATION-BLOCK ::=

   $\boxed{\textbf{where}}$ **{ DECLARATION }**$^+$ $\boxed{\textbf{end}}$

The **DECLARATION**s are local to the **CONFINED-PROCESS**; they are described in part E, section XI–2, page 176 (chapter “Models of processes”).

Local declarations of sequences

The signals (or tuples) that appear in a list of **S-DECLARATION**s associated with an expression of processes are hidden in output of this **CONFINED-PROCESS**.

$$P \texttt{ where } \mu_1\ A_1\texttt{, ..., } A_{n_1}\texttt{; ...; } \mu_m\ A_1\texttt{, ..., } A_{n_m}\texttt{ ... end}$$

The names $A_1, \dots, A_{n_1}, \dots, A_1, \dots, A_{n_m}$ must be mutually distinct.

1. **Profile**

   - $?\big(P \texttt{ where } \mu_1\ A_1\texttt{, ..., } A_{n_1}\texttt{; ...; } \mu_m\ A_1\texttt{, ..., } A_{n_m}\texttt{ ... end}\big) = ?\,(P)$
   - $!\big(P \texttt{ where } \mu_1\ A_1\texttt{, ..., } A_{n_1}\texttt{; ...; } \mu_m\ A_1\texttt{, ..., } A_{n_m}\texttt{ ... end}\big) =$
     $!\,(P) - \{A_1, \dots, A_{n_1},\ \dots,\ A_1, \dots, A_{n_m}\}$

2. **Types** The expression
   $P \texttt{ where } \mu_1\ A_1\texttt{, ..., } A_{n_1}\texttt{; ...; } \mu_m\ A_1\texttt{, ..., } A_{n_m}\texttt{ end}$
   establishes a new syntactic context of P.
   The declarations
   $\texttt{where } \mu_1\ A_1\texttt{, ..., } A_{n_1}\texttt{; ...; } \mu_m\ A_1\texttt{, ..., } A_{n_m}\texttt{ end}$
   are called “local declarations” for P.

   (a) In this context, the type $\tau\big(\mu_i\big)$ is that associated with the signals $A_1, \dots, A_{n_i}$, in accordance with the rules defined in part C, chapter V, “Domains of values of the signals”.

3. **Definition in SIGNAL**

   P `/` A_1`, ...,` A_{n_1}`, ...,` A_1`, ...,` A_{n_m}
   with, in the context of P, the associations of types defined above.

The following rules help to specify the context of visibility established by the local declarations of a confined process (see also in part E, section XI–2, page 176).

- An identifier of sequence X (or an identifier of constant, or an identifier of type) used in an expression on processes that does not contain a declaration of X is said external to this expression of processes.
- An identifier of sequence (or of constant, or of type) X local to an expression of processes P, or external to P and declared in a list of **DECLARATION**s D, is local to the **CONFINED-PROCESS** P `where` D `end`.
- An identifier of sequence (or of constant, or of type) X external to an expression of processes P, and not declared in a list of **DECLARATION**s D, is external to the **CONFINED-PROCESS** P `where` D `end`.
- Let A be an identifier of input signal of an expression of processes P (used but not defined in P), then A must be external to P.
- Let B be an identifier of output signal of a model M, then B must be an output signal defined (at least partially) in the expression of processes associated with M, external to this expression of processes.
- Any sequence used in a **MODEL** but not declared in the interface of this **MODEL** must be either local to the associated expression of processes, or external to the **MODEL** (visible in a syntactic context that includes it). In the same way, any constant or type identifier used in a **MODEL** must be either local to the associated expression of processes, or external to that **MODEL**.

VII–5 Labelled processes

It is possible to label an expression of processes:

XX `::` P

1. **Context-free syntax**

   LABELLED-PROCESS ::=

   Label $\boxed{\textbf{::}}$ **P-EXPR**

   Label ::=

   Name

The labelled process XX `::` P has the same semantics as the process P, but the label XX defines a context clock for the process P, and implicit signals are added to the graph.

The label XX associated with P can be used to designate the process P in some expressions (dependences, for example).

In particular, the label XX can be used to define or to reference a characteristic clock of P: the *tick* of P. For that purpose, the label is considered as a signal of special type label, for which it is always possible to reference its clock (in the usual ways: $\hat{~}XX$ for example).
This clock of the label XX (the *tick* of P) is defined as being greater than the upper bound of the clocks of all the signals designated in P (including the clocks of the labels contained in P, including also the clocks of the signals designated in the macro-expansion of the models referenced in P, limited to the models which are not externally defined or separately compiled).

For the actions called in P, which are directly under the "scope" of the label XX (i.e., for which there is no embedded labelled process containing these invocations of actions), the clock of the label XX defines the activation clock of these actions. This clock can be fixed outside the process P (in conformity with the constraint stated above: it is always greater than the upper bound of the clocks of the signals of P).

The clock of the label XX represents the context clock of P.

The other effect of labelling a process is to add the two following signals to the graph: let us denote them respectively `?` XX and `!` XX, although these notations are not available in the syntax of the language.

Both `?` XX and `!` XX have the clock $\hat{~}XX$ as their common clock. The implicit signal `?` XX is a signal that precedes all the nodes of the graph of the process P: there is a dependence from `?` XX to each one of the signals designated in P. Symmetrically, the implicit signal `!` XX is a signal which is preceded by all the nodes of the graph of P: there is a dependence from each one of the signals designated in P to the signal `!` XX.

This feature is used to specify explicit dependences between processes (cf. part E, section XI–6.2, page 182).

The labels declared in a model of process (cf. part E, chapter XI, page 173) are visible (i.e., can be referenced) everywhere in this model, but not in its included models of processes: a label is in some way local to a model.

In one model, a label cannot have the same name as another visible object (signal, parameter, constant, type, model).

VII–6 Choice processes

not yet fully implemented

A choice process is an expression of processes that allows to compose definitions according to the different values of a signal[1].

`case` X `in`
  `{`$E_{1,1}$`, ..., `E_{1,n_1}`} :` P_1
  ⋮
  `{`$E_{m,1}$`, ..., `E_{m,n_m}`} :` P_m
  `else` P_{m+1}
`end`

[1]not yet implemented in POLYCHRONY: intervals of values.

The “else” part is optional.
Other forms of enumeration of values can also be used in the different branches of the choice process. They are described below.

1. **Context-free syntax**

CHOICE-PROCESS ::=
$\boxed{\textbf{case}}$ **Name-*signal*** $\boxed{\textbf{in}}$ **{ CASE }**$^+$ **[ELSE-CASE]** $\boxed{\textbf{end}}$

CASE ::=
ENUMERATION-OF-VALUES $\boxed{\textbf{:}}$ **GENERAL-PROCESS**

ELSE-CASE ::=
$\boxed{\textbf{else}}$ **GENERAL-PROCESS**

ENUMERATION-OF-VALUES ::=
$\boxed{\textbf{\{}}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }*** $\boxed{\textbf{\}}}$
| $\boxed{\textbf{[.}}$ **[S-EXPR]** $\boxed{\textbf{,}}$ **[S-EXPR]** $\boxed{\textbf{.]}}$
| $\boxed{\textbf{[.}}$ **[S-EXPR]** $\boxed{\textbf{,}}$ **[S-EXPR]** $\boxed{\textbf{[.}}$
| $\boxed{\textbf{.]}}$ **[S-EXPR]** $\boxed{\textbf{,}}$ **[S-EXPR]** $\boxed{\textbf{.]}}$
| $\boxed{\textbf{.]}}$ **[S-EXPR]** $\boxed{\textbf{,}}$ **[S-EXPR]** $\boxed{\textbf{[.}}$

2. **Profile**

- $?\,\big(P_i\big) = \{e_{i,1}, \ldots, e_{i,p_i}\}$
- $?\,\big(\texttt{case } X \texttt{ in ... end}\big) = \{X\} \cup \bigcup_i ?\,(P_i) - \bigcup_i !\,(P_i)$
- $!\,\big(\texttt{case } X \texttt{ in ... end}\big) = \bigcup_i !\,(P_i)$

3. **Types**

   (a) X has a **Scalar-type** or **ENUMERATED-TYPE** and
   $\forall i, j\ \tau\big(E_{i,j}\big) \sqsubseteq \tau\big(X\big)$

4. **Semantics**
   Each **ENUMERATION-OF-VALUES** enumerates some subset of constant values which are in the same domain as the signal X, signal on which the choice is based, and which are possible values of X.
   All the enumerations of values (the “guards” of the choice) must be mutually exclusive. When there is an “else” part, the different sub-types corresponding to the different guards form a partition of the type of X.
   The enumerations of values can take the form of explicit enumerations (used for the description below), or of intervals. The four possible forms of intervals are usable only if the values of the type of X are totally ordered: they define intervals of values that can be, for both sides of the interval, opened or closed. The bounds of an interval are optional (one of the two must be present): if the lower bound is absent, the interval represents all the values smaller than the upper bound (included

or not); if the upper bound is absent, the interval represents all the values greater than the lower bound (included or not).

5. **Definition in SIGNAL**
When the processes P_i have inputs (as specified above), these inputs are filtered by the instants at which the signal X on which the choice is based takes as value one of the values enumerated in the corresponding branch. Then the above choice process is equivalent to:

$$
\begin{array}{ll}
(| & e'_{1,1}\ \texttt{:=}\ e_{1,1}\ \texttt{when ((}X\ \texttt{=}\ E_{1,1}\texttt{) or ... or (}X\ \texttt{=}\ E_{1,n_1}\texttt{))} \\
 & \vdots \\
| & e'_{1,p_1}\ \texttt{:=}\ e_{1,p_1}\ \texttt{when ((}X\ \texttt{=}\ E_{1,1}\texttt{) or ... or (}X\ \texttt{=}\ E_{1,n_1}\texttt{))} \\
| & P_1\ \texttt{[}e'_{1,1}\texttt{/}e_{1,1}\texttt{, ..., }e'_{1,p_1}\texttt{/}e_{1,p_1}\texttt{]} \\
 & \vdots \\
| & e'_{m,1}\ \texttt{:=}\ e_{m,1}\ \texttt{when ((}X\ \texttt{=}\ E_{m,1}\texttt{) or ... or (}X\ \texttt{=}\ E_{m,n_m}\texttt{))} \\
 & \vdots \\
| & e'_{m,p_m}\ \texttt{:=}\ e_{m,p_m}\ \texttt{when ((}X\ \texttt{=}\ E_{m,1}\texttt{) or ... or (}X\ \texttt{=}\ E_{m,n_m}\texttt{))} \\
| & P_m\ \texttt{[}e'_{m,1}\texttt{/}e_{m,1}\texttt{, ..., }e'_{m,p_m}\texttt{/}e_{m,p_m}\texttt{]} \\
| & e'_{m+1,1}\ \texttt{:=}\ e_{m+1,1}\ \texttt{when ((}X\ \texttt{/=}\ E_{1,1}\texttt{) and ... and (}X\ \texttt{/=}\ E_{m,n_m}\texttt{))} \\
 & \vdots \\
| & e'_{m+1,p_{m+1}}\ \texttt{:=}\ e_{m+1,p_{m+1}}\ \texttt{when ((}X\ \texttt{/=}\ E_{1,1}\texttt{) and ... and (}X\ \texttt{/=}\ E_{m,n_m}\texttt{))} \\
| & P_{m+1}\ \texttt{[}e'_{m+1,1}\texttt{/}e_{m+1,1}\texttt{, ..., }e'_{m+1,p_{m+1}}\texttt{/}e_{m+1,p_{m+1}}\texttt{]} \\
|) & \texttt{/ }e'_{1,1}\texttt{, ..., }e'_{1,p_1}\texttt{, ..., }e'_{m+1,1}\texttt{, ..., }e'_{m+1,p_{m+1}}
\end{array}
$$

where $P_i\ [e'_{i,1}/e_{i,1}, \ldots, e'_{i,p_i}/e_{i,p_i}]$ represents the process P_i in which new identifiers $e'_{i,j}$ are substituted to the identifiers $e_{i,j}$ which are inputs of P_i.
For all the processes P_i, the new identifiers $e'_{i,j}$ are mutually distinct and do not appear elsewhere.
If some process P_i does not have inputs, there is of course no sampling of inputs for it but the "call" of this process is made at the clock at which the signal X on which the choice is based takes as value one of the values enumerated in the corresponding branch of the choice. In the definition, this can be expressed by:

$$
\begin{array}{ll}
(| & \vdots \\
| & label_i\ \hat{}=\ \texttt{when ((}X\ \texttt{=}\ E_{i,1}\texttt{) or ... or (}X\ \texttt{=}\ E_{i,n_i}\texttt{))} \\
| & label_i\ \texttt{::}\ P_i \\
 & \vdots \\
|) &
\end{array}
$$

for $i = 1..m$
and by:

$$
\begin{array}{ll}
(| & \vdots \\
| & label_{m+1}\ \hat{}=\ \texttt{when ((}X\ \texttt{/=}\ E_{1,1}\texttt{) and ... and (}X\ \texttt{/=}\ E_{m,n_m}\texttt{))} \\
| & label_{m+1}\ \texttt{::}\ P_{m+1} \\
|) &
\end{array}
$$

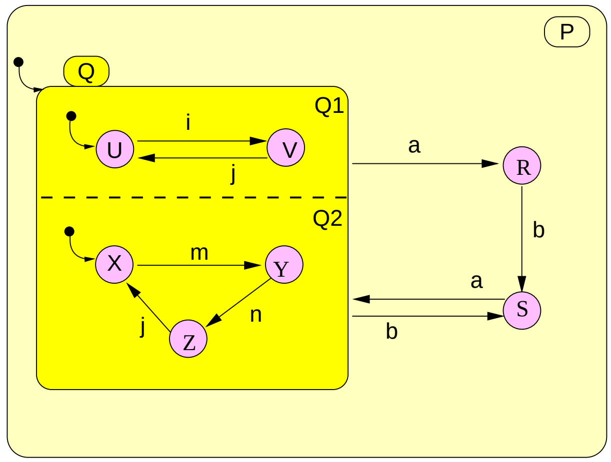
for the "else" part
(in place of the corresponding subsamplings and references to P_i in the previous definition).

6. **Clocks** The values $E_{i,j}$ are constant expressions:

(a) $\omega(E_{i,j}) = \hbar$

Example

The statechart:



may be described by the following program (process models and modules are described respectively in chapter XI, page 173 and chapter XII, page 191):

```
module P_statechart =

type P_states = enum (Q, R, S);
type Q1_states = enum (U, V);
type Q2_states = enum (X, Y, Z);

process P_chart =
   ( ? event Tick;
       event a, b, i, j, m, n;
     ! P_states P_currentState;
       Q1_states Q1_currentState;
       Q2_states Q2_currentState;
   )
   (| (| case P_currentState in
           {#Q}: (| P_nextState ::= (#R when a) default (#S when b) |)
           {#R}: (| P_nextState ::= #S when b |)
           {#S}: (| P_nextState ::= #Q when a |)
         end
       | P_nextState ::= defaultvalue P_currentState
       | P_currentState := P_nextState $ init #Q
       | P_currentState ^= Tick
       |)
    | clk_Q_chart := when (P_currentState = #Q)
    | start_Q_chart := when (P_nextState = #Q) when (P_currentState /= #Q)
    | Q1_State ^= Q2_State ^= clk_Q_chart ^+ start_Q_chart
    | (| case Q1_State in
           {#U}: (| Q1_newState ::= #V when i |)
           {#V}: (| Q1_newState ::= #U when j |)
         end
       | Q1_newState ::= defaultvalue Q1_State
       | Q1_newState ^= Q1_State
       | Q1_nextState := (#U when start_Q_chart) default Q1_newState
```

```
       | Q1_State := Q1_nextState $ init #U
       | Q1_currentState := Q1_State when clk_Q_chart
       |)
    | (| case Q2_State in
           {#X}: (| Q2_newState ::= #Y when m |)
           {#Y}: (| Q2_newState ::= #Z when n |)
           {#Z}: (| Q2_newState ::= #X when j |)
         end
       | Q2_newState ::= defaultvalue Q2_State
       | Q2_newState ^= Q2_State
       | Q2_nextState := (#X when start_Q_chart) default Q2_newState
       | Q2_State := Q2_nextState $ init #X
       | Q2_currentState := Q2_State when clk_Q_chart
       |)
    |)
   where
     P_states P_nextState;
     event clk_Q_chart, start_Q_chart;
     Q1_states Q1_State, Q1_newState, Q1_nextState;
     Q2_states Q2_State, Q2_newState, Q2_nextState;
   end;
end;
```

(note that the program could be better structured using several process models).

Part D

THE COMPOSITE SIGNALS

Chapter VIII

Tuples of signals

An expression of tuple is an enumeration of elements of tuple, or a designation of field.

1. **Context-free syntax**

   S-EXPR-*TUPLE* ::=

   TUPLE-ENUMERATION
   | **TUPLE-FIELD**

VIII–1 Constant expressions

A constant expression of tuple is an **S-EXPR**-*TUPLE* which has recursively as arguments constant expressions, or any expression defining a tuple the elements of which are constants.

VIII–2 Enumeration of tuple elements

A tuple represents a list (finite sequence) of signals or tuples.
$\textbf{(}\, E_1, \ldots, E_n \,\textbf{)}$

1. **Context-free syntax**

   TUPLE-ENUMERATION ::=

   $\boxed{\textbf{(}}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }*** $\boxed{\textbf{)}}$

2. **Types**

   (a) $\tau\big(\textbf{(}\, E_1, \ldots, E_n \,\textbf{)}\big) = (\tau\big(E_1\big) \times \ldots \times \tau\big(E_n\big))$

3. **Semantics**
   The tuple $\textbf{(}\, E_1,\ \ldots,\ E_n \,\textbf{)}$ is equal to $< v_1, \ldots, v_n >$ where $< v_1, \ldots, v_n >$ is the sequence of signals or tuples resulting from the evaluation of the expressions $E_1, \ldots, E_n$.
   The semantics is described formally in part B, section III–7.1, page 42.

VIII–3 Denotation of field

$X \texttt{.} X_i$

1. **Context-free syntax**

   TUPLE-FIELD ::=

   S-EXPR $\boxed{\texttt{.}}$ **Name-***field*

2. **Types**

   (a) $\tau(X) = bundle(\{X_1\} \rightarrow \mu_1 \times \ldots \times \{X_m\} \rightarrow \mu_m)$

   (b) $\tau(X \texttt{.} X_i) = \mu_i$

3. **Semantics**

   If X is a tuple with named fields $X_1, \ldots, X_m$, $X \texttt{.} X_i$ designates the signal or the tuple corresponding to the field with name X_i.

   In particular, the denotation of field may apply on an **INSTANCE-OFPROCESS** when the output of the corresponding model is a tuple with named fields. It may also apply on an array element if the elements of the array are monochronous tuples with named fields.
   The semantics is described formally in part B, section III–7.1, page 42.

VIII–4 Destructuration of tuple

The syntax of an **INSTANCE-OF-PROCESS** is used to denote the call of predefined functions of destructuration of tuples:

- `tuple(` X `)`
  - If X is a tuple with named fields of type $bundle(\{X_1\} \rightarrow \mu_1 \times \ldots \times \{X_m\} \rightarrow \mu_m)$, `tuple(` X `)` is the corresponding tuple with unnamed fields, `(`$X_1, \ldots, X_m$`)`, of type $(\mu_1 \times \ldots \times \mu_m)$
  - If X is a tuple with unnamed fields, the components of which are, in this order, $X_1, \ldots, X_m$, `tuple(` X `)` is the tuple with unnamed fields `(tuple(` X_1 `)`$, \ldots,$ `tuple(` X_m `))`
  - If X is not of tuple type, then `tuple(` X `)` is equal to X.

- `rtuple(` X `)`
  - If X is a tuple with named fields of type $bundle(\{X_1\} \rightarrow \mu_1 \times \ldots \times \{X_m\} \rightarrow \mu_m)$, `rtuple(` X `)` is the tuple with unnamed fields `(rtuple(` X_1 `)`$, \ldots,$ `rtuple(` X_m `))`
  - If X is a tuple with unnamed fields, the components of which are, in this order, $X_1, \ldots, X_m$, `rtuple(` X `)` is the tuple with unnamed fields `(rtuple(` X_1 `)`$, \ldots,$ `rtuple(` X_m `))`
  - If X is not of tuple type, then `rtuple(` X `)` is equal to X.

VIII–5 Equation of definition of tuple component

A tuple can be defined component by component. An equation of definition of component of tuple is an expression of processes the syntax of which extends the **DEFINITION-OF-SIGNALS** given in part C, section VI–1.1, page 91. The general form can contain both definitions of components of tuples and global definitions of tuples and signals.

$$(\,X_1\,.\,A_1, \ldots, X_n\,.\,A_n\,) := E$$

1. **Context-free syntax**

   DEFINITION-OF-SIGNALS ::=

   COMPONENT $\boxed{\textbf{:=}}$ **S-EXPR**
   | **COMPONENT** $\boxed{\textbf{::=}}$ **S-EXPR**
   | **COMPONENT** $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**
   | $\boxed{\textbf{(}}$ **COMPONENT {** $\boxed{\textbf{,}}$ **COMPONENT }*** $\boxed{\textbf{)}}$ $\boxed{\textbf{:=}}$ **S-EXPR**
   | $\boxed{\textbf{(}}$ **COMPONENT {** $\boxed{\textbf{,}}$ **COMPONENT }*** $\boxed{\textbf{)}}$ $\boxed{\textbf{::=}}$ **S-EXPR**
   | $\boxed{\textbf{(}}$ **COMPONENT {** $\boxed{\textbf{,}}$ **COMPONENT }*** $\boxed{\textbf{)}}$ $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

   COMPONENT ::=

   Name-*signal*
   | **Name-***signal* $\boxed{\textbf{.}}$ **COMPONENT**

2. **Types**

   (a) $\tau\big((\,X_1\,.\,A_1, \ldots, X_n\,.\,A_n\,)\big) = (\tau\big(X_1\,.\,A_1\big) \times \ldots \times \tau\big(X_n\,.\,A_n\big))$
   (b) $\tau\big(E\big) \sqsubseteq (\tau\big(X_1\,.\,A_1\big) \times \ldots \times \tau\big(X_n\,.\,A_n\big))$

3. **Semantics**

   - $X_1\,.\,A_1, \ldots, X_n\,.\,A_n$ designate signals or tuples of signals, respectively components of the tuples $X_1, \ldots, X_n$.
   - Each signal or tuple $X_i\,.\,A_i$ is respectively equal to the signal or tuple v_i that corresponds positionally to it in output of E.

4. **Clocks** A *signal* and the signal v_i that defines it are synchronous. In that case:

   (a) $\omega\big(X_i\,.\,A_i\big) = \omega\big(v_i\big)$

Chapter IX

Spatial processing

Spatial processing is obtained by manipulations of arrays.

The following operators are provided:

- operators of definition by enumeration (**ARRAY-ENUMERATION**, **CONCATENATION**, **ITERATIVE-ENUMERATION**);
- an operator of definition of indices (**INDEX**);
- operators of access to elements of arrays (**ARRAY-ELEMENT**, **SUB-ARRAY**);
- an operator of array restructuration (**ARRAY-RESTRUCTURATION**);
- operators of sequential definition (**SEQUENTIAL-DEFINITION**, **ITERATIVE-ENUMERATION**);
- global operators on matrices such as transposition (**TRANSPOSITION**) and products (**ARRAY-PRODUCT**).

Moreover, structures of iteration are also defined on processes (**ITERATION-OF-PROCESSES**), with an associated operator of definition of multiple indices (**MULTI-INDEX**).

1. **Context-free syntax**

S-EXPR-*ARRAY* ::=

ARRAY-ENUMERATION
| **CONCATENATION**
| **ITERATIVE-ENUMERATION**
| **INDEX**
| **ARRAY-ELEMENT**
| **SUB-ARRAY**
| **ARRAY-RESTRUCTURATION**
| **MULTI-INDEX**
| **SEQUENTIAL-DEFINITION**
| **TRANSPOSITION**
| **ARRAY-PRODUCT**
| **REFERENCE-SEQUENCE**

IX–1 Dimensions of arrays and bounded values

Dimensions of arrays

The syntax of an **INSTANCE-OF-PROCESS** is used to denote the call of predefined functions with constant result giving the dimension of an array and the size of a dimension:

- `dim(` T `)`
  If T has a type $([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu$ where ν is a **Scalar-type** or **External-type** or **ENUMERATED-TYPE**,
  then $\varphi($`dim(` T `)`$) = m$.
  If T has a type ν where ν is a **Scalar-type** or **External-type** or **ENUMERATED-TYPE**,
  then $\varphi($`dim(` T `)`$) = 0$.

- `size(` T,I `)`
  If T has a type $([0..n_1 - 1] \times \ldots \times [0..n_m - 1]) \to \nu$ where ν is a **Scalar-type** or **External-type** or **ENUMERATED-TYPE**,
  and if $1 \leq \varphi(I) \leq m$,
  then $\varphi($`size(` T,I `)`$) = n_I$,
  else $\varphi($`size(` T,I `)`$)$ is not defined: it is an error in the program.

- `size(` T `)` is, by definition, equivalent to
  `size(` T,1 `)`

Bounded values

The syntax of an **INSTANCE-OF-PROCESS** is used to denote the call of a predefined function used to deliver bounded values.

`bounds(` E_1, E_2, E_3 `)`

The values of E_1 are compelled to evolve between that of E_2 and E_3.

1. **Types**
   (a) E_1, E_2 and E_3 are signals of a same domain **Scalar-type** (other than a **Complex-type**), or **ENUMERATED-TYPE**.
   (b) $\tau($`bounds(` E_1, E_2, E_3 `)`$) = \tau(E_1) \sqcup \tau(E_2) \sqcup \tau(E_3)$
   (c) The pointwise extension is described in part D, chapter X, page 169.

2. **Definition in SIGNAL**
   X `:=` `bounds(` E_1, E_2, E_3 `)`
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of bounded values, is equal to the process defined as follows:

   `(|` X `:= if` E_1 `<` E_2 `then` E_2 `else if` E_1 `>` E_3 `then` E_3 `else` E_1
   `|)`

3. **Clocks**
   (a) $\omega(E_1) = \omega(E_2)$

(b) $\omega\big(E_1\big) = \omega\big(E_3\big)$

(c) $\omega\big(\texttt{bounds(}\ E_1\texttt{,}\ \ E_2\texttt{,}\ \ E_3\ \texttt{)}\big) = \omega\big(E_1\big)$

IX–2 Constant expressions

A constant expression of array is an **S-EXPR**-*ARRAY* which has recursively as arguments constant expressions, or any expression defining an array the elements of which are constants.

IX–3 Enumeration

The enumeration of the elements of an array defines a vector by the ordered list of its elements.

$\texttt{[}E_1, \ldots, E_n\texttt{]}$

1. **Context-free syntax**

   ARRAY-ENUMERATION ::=

   $\boxed{\textbf{[}}$ **S-EXPR** { $\boxed{\textbf{,}}$ **S-EXPR** }* $\boxed{\textbf{]}}$

2. **Profile**

   $$?\ \big(\texttt{[}E_1, \ldots, E_n\texttt{]}\big) = \bigcup_{i=1}^{n} ?\ \big(E_i\big)$$

3. **Types**

   (a) $\tau\big(\texttt{[}E_1, \ldots, E_n\texttt{]}\big) = [0..n-1] \rightarrow \bigsqcup_{i=1}^{n} \tau(E_i)$

4. **Semantics**

   $\texttt{[}E_1, \ldots, E_n\texttt{]}$ designates the vector the n components of which are, in this order, $E_1, \ldots, E_n$ (cf. part B, section III–7.2, page 44).

5. **Clocks**

   (a) $\omega\big(\texttt{[}E_1\texttt{,}\ \ \texttt{...,}\ \ E_n\texttt{]}\big) = \omega\big(E_i\big) \quad \forall i = 1, \ldots, n$

6. **Examples**

   (a) With M1 := [[M11,M12,M13],[M21,M22,M23]],
   M1[1] is equal to [M11,M12,M13].

IX–4 Concatenation

The concatenation allows to concatenate arrays along to their first dimension.

$E_1 \mid\!+ E_2$

1. **Context-free syntax**

   CONCATENATION ::=

   S-EXPR $\boxed{\mid\!+}$ **S-EXPR**

2. **Types**

   (a) $\tau(E_1) = [0..m_1 - 1] \rightarrow \mu_1$

   (b) $\tau(E_2) = [0..m_2 - 1] \rightarrow \mu_2$

   (c) $\tau(E_1 \mid+ E_2) = [0..m_1 + m_2 - 1] \rightarrow \mu_1 \sqcup \mu_2$

3. **Definition in SIGNAL**
   X := $E_1 \mid+ E_2$ is equal to the process defined as follows:

   X := [E_1[0], ..., E_1[$m_1 - 1$], E_2[0], ..., E_2[$m_2 - 1$]]

4. **Clocks**

   (a) $\omega(E_1) = \omega(E_2)$

   (b) $\omega(E_1 \ \mid+ \ E_2) = \omega(E_1)$

IX–5 Repetition

The repetition is a simple form of iterative enumeration which allows the finite repetition of a value.
$E \mid* N$

1. **Context-free syntax**

   ITERATIVE-ENUMERATION ::=
   S-EXPR $\boxed{\mid*}$ **S-EXPR**

2. **Types**

   (a) $\tau(E) = \mu$

   (b) N is a positive integer expression, with a strictly positive upper bound, N_{max}.

   (c) $\tau(E \mid* N) = [0..N_{max} - 1] \rightarrow \mu$

3. **Semantics**
   At a given instant, all the elements of the vector defined by $E \mid* N$ have the same value, which is the value of E.
   The semantics is described formally in part B, section III–7.2, page 44, using the "iterative enumeration of array". The maximum number of iterations is given by N, and the iteration function which is used here is the identity function with first value the value E itself.

4. **Clocks**

   (a) $\omega(E) = \omega(N)$

   (b) $\omega(E \ \mid* \ N) = \omega(E)$

IX–6 Definition of index

E_1 `..` E_2 `step` E_3

1. **Context-free syntax**

   INDEX ::=

   S-EXPR $\boxed{\textbf{..}}$ **S-EXPR [** $\boxed{\textbf{step}}$ **S-EXPR]**

2. **Types**

   (a) E_1 and E_2 are bounded integers such that the difference $E_1 - E_2$ has always the same sign (at every instant): $\forall t$, $E_{1t} \leq E_{2t}$ or $\forall t$, $E_{1t} \geq E_{2t}$.
   $lower\_bound(E_1)$, $upper\_bound(E_1)$, $lower\_bound(E_2)$ and $upper\_bound(E_2)$ will denote respectively the lower bounds and upper bounds of E_1 and E_2.

   (b) E_3 is an integer constant different from 0, such that
   if $\forall t$, $E_{1t} \leq E_{2t}$ then $\varphi\big(E_3\big) > 0$
   and if $\forall t$, $E_{1t} \geq E_{2t}$ then $\varphi\big(E_3\big) < 0$.
   When the step expression, E_3, is omitted, its value is implicitly equal to 1.

   (c) If $\varphi\big(E_3\big) > 0$,
   $\tau\big(E_1 \texttt{..} E_2 \texttt{ step } E_3\big) =$
   $[0..((upper\_bound(E_2) - lower\_bound(E_1))/\varphi\big(E_3\big) + 1) - 1] \rightarrow \tau\big(E_1\big) \sqcup \tau\big(E_2\big)$
   If $\varphi\big(E_3\big) < 0$,
   $\tau\big(E_1 \texttt{..} E_2 \texttt{ step } E_3\big) =$
   $[0..((upper\_bound(E_1) - lower\_bound(E_2))/(-\varphi\big(E_3\big)) + 1) - 1] \rightarrow \tau\big(E_1\big) \sqcup \tau\big(E_2\big)$
   In any case, the size of the vector must be strictly positive.

3. **Semantics**
   The vector of integers defined by E_1 `..` E_2 `step` E_3 has as successive elements the values E_{1t}, $E_{1t} + \varphi\big(E_3\big)$, $E_{1t} + (2 * \varphi\big(E_3\big))$, etc., up to the last value between E_{1t} and E_{2t} (included).
   The semantics is described formally in part B, section III–7.2, page 44, using the "iterative enumeration of array".
   The iteration function is the function f such that $f(x) = x + \varphi\big(E_3\big)$. The first value is E_1.
   If $\varphi\big(E_3\big) > 0$, the maximum number of iterations is given by
   N = $(E_2 - E_1)$ / $\varphi\big(E_3\big)$ + 1.
   If $\varphi\big(E_3\big) < 0$, the maximum number of iterations is given by
   N = $(E_1 - E_2)$ / $(-\varphi\big(E_3\big))$ + 1.

4. **Clocks**

   (a) $\omega\big(E_1\big) = \omega\big(E_2\big) = \omega\big(E_1 \texttt{..} E_2 \texttt{ step } E_3\big)$

   (b) $\omega\big(E_3\big) = \hbar$

IX–7 Array element

An array element is obtained by indexing following the syntax of the first rule below. Every index of array must be a positive bounded integer, whose upper bound is strictly inferior to the size n of the considered dimension; the second rule provides a syntax of "local recovery" which defines the value of the expression for the values of index outside the segment $[0..n-1]$.

1. **Context-free syntax**

   ARRAY-ELEMENT ::=

   S-EXPR $\boxed{[}$ **S-EXPR** { $\boxed{,}$ **S-EXPR** }* $\boxed{]}$

   | **S-EXPR** $\boxed{[}$ **S-EXPR** { $\boxed{,}$ **S-EXPR** }* $\boxed{]}$ **ARRAY-RECOVERY**

   ARRAY-RECOVERY ::=

   $\boxed{\backslash\backslash}$ **S-EXPR**

IX–7.1 Access without recovery

$T\,[\,E_1, \ldots, E_m\,]$

1. **Profile**

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

2. **Types**

   (a) For all i, E_i is a positive (or zero) integer, with an upper bound. Let n_i the value of its upper bound.

   (b) $\tau\big(T\big) = ([0..n_1-1] \times \ldots \times [0..n_m-1]) \rightarrow \mu$
   (remark: μ can be an array type.)

   (c) $\tau\big(T\,[\,E_1, \ldots, E_m\,]\big) = \mu$

3. **Semantics**
   If $v_1, \ldots, v_m$ represent respectively the self-corresponding elements in the sequences of values represented by $E_1, \ldots, E_m$, the corresponding element in the sequence represented by $T\,[\,E_1$, $\ldots$, $E_m\,]$ is $T(< v_1, \ldots, v_m >)$.
   The semantics is described formally in part B, section III–7.2, page 44.

4. **Clocks**

   (a) $\omega\big(E_1\big) = \omega\big(T\big), \ldots, \omega\big(E_m\big) = \omega\big(T\big)$

   (b) $\omega\big(T\,[\,E_1, \ldots, E_m\,]\big) = \omega\big(T\big)$

5. **Properties**

   (a) $(E_1, \ldots, E_m \text{ of type integer}) \quad \Rightarrow \quad (T\,[\,E_1, \ldots, E_m\,] = T\,[\,E_1\,] \ldots [\,E_m\,])$

IX–7.2 Access with recovery

$T\,[\,E_1, \ldots, E_m\,]\backslash\backslash V$

1. **Types**

   (a) $\tau\big(T\big) = ([0..n_1-1] \times \ldots \times [0..n_m-1]) \rightarrow \mu_1$

(b) For all $i = 1, \ldots, m$, $\tau(E_i)$ is an **Integer-type**.

(c) $\tau(V) = \mu_2$

(d) $\tau(T\,[E_1, \ldots, E_m]\backslash\backslash V) = \mu_1 \sqcup \mu_2$

2. **Definition in SIGNAL**

X := $T\,[E_1, \ldots, E_m]\backslash\backslash V$

whose right side of $\boxed{:=}$ represents an expression of access to an array element with recovery, is equal to the process defined as follows:

```
(| X1 := T[E1 modulo n1, ..., Em modulo nm]
 | B1 := (0 <= E1) and (E1 <= (n1 - 1))
   ⋮
 | Bm := (0 <= Em) and (Em <= (nm - 1))
 | B := (B1 and ... and Bm) when ^T
 | X2 := V when ^T
 | X := (X1 when B) default X2
 |) / X1, X2, B, B1, ..., Bm
```

3. **Clocks**

(a) $\omega(E_1) = \omega(T), \ldots, \omega(E_m) = \omega(T)$

(b) $\omega(V) = \omega(T)$

(c) $\omega(T\,[E_1,\ \ldots,\ E_m]\backslash\backslash V) = \omega(T)$

IX–8 Extraction of sub-array

The expression of extraction of sub-array is a generalization, with the same syntax, of the expression of access to an array element (cf. section IX–7, page 151). Only the form where the accesses are obtained via "generalized indices" (represented as arrays of integers) is given here; when they are integers, the description of the corresponding expression is given in IX–7.

$T\,[I_1, \ldots, I_n]$

1. **Context-free syntax**

SUB-ARRAY ::=

S-EXPR $\boxed{[}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{]}$

2. **Types**

(a) $\tau(I_1) = \ldots = \tau(I_n) = ([0..b_1] \times \ldots \times [0..b_p]) \longrightarrow \nu$
with ν an integer type, and the basic integer values of the `Ii` are positive or zero.

(b) More generally, the list of indices $I_1, \ldots, I_n$ can be specified by any expression denoting a function $([0..b_1] \times \ldots \times [0..b_p]) \longrightarrow \nu^n$ (with ν an integer type).

(c) $\tau(T) = ([0..a_1] \times \ldots \times [0..a_n]) \longrightarrow \mu$
(μ can be an array type).

(d) $\tau(T\texttt{[}I_1, \ldots, I_n\texttt{]}) = ([0..b_1] \times \ldots \times [0..b_p]) \to \mu$

3. **Semantics**
$T\texttt{[}I_1, \ldots, I_n\texttt{]}$ extracts some sub-array from T.
The semantics is described formally in part B, section III–7.2, page 44 (non defined values, represented by nil in the semantics, are any values of correct type).

If T has at least n dimensions (and has the basic type μ for the elements corresponding to these n first dimensions), it can be traversed using jointly n indices I_1, ..., I_n (one per dimension), that allow to extract elements of type μ.

Each one of the indices is an array with the same number of dimensions, let p.

The result, let X, has the same number of dimensions as the indices, which is p. Its basic elements have the type μ (type of the extracted elements).

With each "position" $(j_1, \ldots, j_p)$ in X, it is associated the element of T the position of which is given by the value of the n indices in $(j_1, \ldots, j_p)$, i.e., in the position
$(I_1\texttt{[}j_1, \ldots, j_p\texttt{]}, \ldots, I_n\texttt{[}j_1, \ldots, j_p\texttt{]})$ in T.

4. **Clocks**

(a) $\omega(I_1) = \omega(T), \ldots, \omega(I_n) = \omega(T)$
(b) $\omega(T\texttt{[}I_1\texttt{, ..., }I_n\texttt{]}) = \omega(T)$

5. **Properties**

(a) If V is a vector of type $[0..n-1] \to \mu$ and if I is an index defined by I `:= 0..`$n-1$, then the expressions V and $V\texttt{[}I\texttt{]}$ are equivalent.

6. **Examples**

(a) `([[10,20],[30,40]])[1,0]` value is `30`.
(b) `(0..10)[2..4]` value is `[2,3,4]`.
(c) if `M` is a $n \times n$ matrix, then `M[0..n−1,0..n−1]` is the vector containing its diagonal.

IX–9 Array restructuration

The array restructuration allows to define partially (in the general case) an array, by defining some indices-defined coordinate points of this array. Non defined values are any values of correct type. This operator is the "reverse" of the operator of extraction of sub-array (cf. section IX–8, page 153) in the following informal way: let T be the result of $\texttt{(}I_1, \ldots, I_n\texttt{) : }S$; if the indices are such that each element of S is used only once by the definition, then $T\texttt{[}I_1, \ldots, I_n\texttt{]}$ value is S.

$\texttt{(}I_1, \ldots, I_n\texttt{) : }S$

1. **Context-free syntax**

ARRAY-RESTRUCTURATION ::=

S-EXPR $\boxed{\textbf{:}}$ **S-EXPR**

2. **Types**
   Depending on $I_1, \ldots, I_n$ being integers or arrays of integers, one of the following sets of relations on types applies:

   (a)
   - For any k, $\tau(I_k)$ is a positive or null integer, with an upper bound. Let a_k this upper bound.
   - $\tau(S) = \mu$
   - $\tau(\texttt{(}I_1\texttt{,}\ldots\texttt{,}I_n\texttt{)}\ \texttt{:}\ S) = ([0..a_1] \times \ldots \times [0..a_n]) \rightarrow \mu$

   (b)
   - $\tau(I_1) = \ldots = \tau(I_n) = ([0..b_1] \times \ldots \times [0..b_p]) \rightarrow \nu$
     with ν an integer type, and for $1 \leq i \leq n$, $\min\limits_{K \in Dom(I_i)} I_i(K) \geq 0$
   - More generally, the tuple of indices $\texttt{(}I_1\texttt{,}\ldots\texttt{,}I_n\texttt{)}$ can be specified by any expression denoting a function $([0..b_1] \times \ldots \times [0..b_p]) \rightarrow \nu^n$ (with ν an integer type).
   - $\tau(S) = ([0..c_1] \times \ldots \times [0..c_p]) \rightarrow \mu$
     with $c_1 \geq b_1, \ldots, c_p \geq b_p$
   - $\tau(\texttt{(}I_1\texttt{,}\ldots\texttt{,}I_m\texttt{)}\ \texttt{:}\ S) = ([0..a_1] \times \ldots \times [0..a_n]) \rightarrow \mu$
     with for $1 \leq i \leq n$, $a_i = \max\limits_{K \in Dom(I_i)} I_i(K)$

3. **Semantics**
   $\texttt{(}I_1\texttt{,}\ldots\texttt{,}I_n\texttt{)}\ \texttt{:}\ S$ specifies a partial definition of array, using the coordinate points defined by the tuple of "generalized indices" $\texttt{(}I_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{)}$ and the values of S obtained by skimming through these coordinates.
   The semantics is described formally in part B, section III–7.2, page 44 (non defined values, represented by nil in the semantics, are any values of correct type).

   Let T be the array defined by the expression $\texttt{(}I_1\texttt{,}\ldots\texttt{,}I_n\texttt{)}\ \texttt{:}\ S$. If the indices $I_1, \ldots, I_n$ are such that they allow to scan exactly the array T (each position is visited only once using these indices), then the restructuration $T\ \texttt{:=}\ \texttt{(}I_1\texttt{,}\ldots\texttt{,}I_n\texttt{)}\ \texttt{:}\ S$ defines the array T such that the extraction of sub-array $T\texttt{[}I_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{]}$ (cf. section IX–8, page 153) is equal to S.

   In other words, $T\texttt{[}I_1\texttt{[}k_1\texttt{,}\ldots\texttt{,}k_p\texttt{]}\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{[}k_1\texttt{,}\ldots\texttt{,}k_p\texttt{]}\texttt{]} = S\texttt{[}k_1\texttt{,}\ldots\texttt{,}k_p\texttt{]}$.

   If $(I_1\texttt{[}k_1, \ldots, k_p\texttt{]}, \ldots, I_n\texttt{[}k_1, \ldots, k_p\texttt{]})$ defines the same position for several distinct values of $(k_1, \ldots, k_p)$, it is the element corresponding to the *max* of the $(k_1, \ldots, k_p)$ (in lexicographic order) which is used.

4. **Clocks**

   (a) $\omega(I_1) = \omega(S), \ldots, \omega(I_n) = \omega(S)$

   (b) $\omega(\texttt{(}I_1\texttt{, ..., }I_n\texttt{)}\ \texttt{:}\ S) = \omega(S)$

5. **Examples**

   (a) `2 : 1`   is a vector $\texttt{[}any\texttt{,}any\texttt{,1]}$.
   where any represents any well-typed value (nil in the semantics).
   Its type is $[0..2] \rightarrow integer$ since the maximal value of 2 is 2.

   (b) `(1,2) : 3`   is a matrix $\texttt{[[}any\texttt{,}any\texttt{,}any\texttt{],[}any\texttt{,}any\texttt{,3]]}$.
   Its type is $([0..1] \times [0..2]) \rightarrow integer$.

(c) `1 : [[1,2],[3,4]]` is a 3-dimensions array `[[[`any`,`any`],[`any`,`any`]],[[1,2],[3,4]]]`.
Its type is $([0..1] \times [0..1] \times [0..1]) \to integer$.

(d) `([0,1],[2,1]) : [4,5]` is a matrix `[[`any`,`any`,4],[`any`,5,`any`]]`.

IX–10 Generalized indices

The syntax of an **INSTANCE-OF-PROCESS** is used to denote the call of a predefined function that delivers generalized "unit" indices. Such indices can be used for standard array traversal in extraction of sub-array (cf. section IX–8, page 153) or array restructuration (cf. section IX–9, page 154).

`indices(` a_1,...,a_n `)`

Let the expression `indices(` a_1,...,a_n `)` define jointly n indices $I_1, \ldots, I_n$:
`(`I_1,...,I_n`) := indices(` a_1,...,a_n `)`

1. **Types**

   (a) The elaborated values of a_1 ($\varphi(a_1)$), ..., a_n ($\varphi(a_n)$) are strictly positive integers.

   (b) For all $j = 1, \ldots, n$,
   $\tau(I_j) = ([0..\varphi(a_1) - 1] \times \ldots \times [0..\varphi(a_n) - 1]) \to \nu$
   where ν is an **Integer-type**.

2. **Semantics**
   For all $j = 1, \ldots, n$,
   for all k_l such that $0 \leq k_l \leq \varphi(a_l) - 1$,

$$(\forall t) \quad (\ I_{j_t}(k_1, \ldots, k_n) = k_j\ )$$

3. **Definition in SIGNAL**
   `(`I_1,...,I_n`) := indices(` a_1,...,a_n `)`
   may be obtained by the process defined as follows:

```
(| (II_1,...,II_n) := <<0..a_1 - 1,...,0..a_n - 1>>
 | iterate (II_1,...,II_n) of
      (I_1[II_1,...,II_n], ..., I_n[II_1,...,II_n]) := (II_1, ..., II_n)
   end
 |) / II_1, ..., II_n
```

   (cf. section IX–12, page 157 and section IX–13, page 158).

4. **Clocks**

   (a) $\omega(a_1) = \hbar, \ldots, \omega(a_n) = \hbar$

   (b) $\omega($`indices(` $a_1, \ldots, a_n$ `)`$) = \hbar$

IX–11 Extended syntax of equations of definition

not yet fully imple- mented

The following syntax[1] extends the syntax of **DEFINITION-OF-SIGNALS** given in VIII–5, page 145:

1. **Context-free syntax**

DEFINITION-OF-SIGNALS ::=

  DEFINED-ELEMENT $\boxed{\textbf{:=}}$ **S-EXPR**
| **DEFINED-ELEMENT** $\boxed{\textbf{::=}}$ **S-EXPR**
| **DEFINED-ELEMENT** $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **DEFINED-ELEMENT** { $\boxed{\textbf{,}}$ **DEFINED-ELEMENT** }* $\boxed{\textbf{)}}$
  $\boxed{\textbf{:=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **DEFINED-ELEMENT** { $\boxed{\textbf{,}}$ **DEFINED-ELEMENT** }* $\boxed{\textbf{)}}$
  $\boxed{\textbf{::=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **DEFINED-ELEMENT** { $\boxed{\textbf{,}}$ **DEFINED-ELEMENT** }* $\boxed{\textbf{)}}$
  $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

DEFINED-ELEMENT ::=

  COMPONENT
| **COMPONENT** $\boxed{\textbf{[}}$ **S-EXPR** { $\boxed{\textbf{,}}$ **S-EXPR** }* $\boxed{\textbf{]}}$

An equation
$X\,[\,I_1,\ldots,I_m\,]\ :=\ E$
is another way to write:
$X\ :=\ (\,I_1,\ldots,I_m\,)\ :\ E$
The definition is similar when the symbol $\boxed{\textbf{::=}}$ is used.

If one equation defines only partially an array, this array can be defined using several equations, defining different parts or elements of this array.

Independently of non defined elements (represented by nil in the semantics), like any signal, a given element cannot be defined by distinct values at a same instant.

All the elements of an array have the same clock, which is the clock of the array. In particular, if some element is undefined at a given instant at which other elements are defined, this element is considered to have any well-typed value.

IX–12 Cartesian product

The cartesian product is used mainly to define jointly indices, to be used in the provided structure of iteration of processes (cf. section IX–13, page 158). Intuitively, the sequence of iteration is represented by the first dimension of the indices (which are vectors). Thus, it is different from the generalized indices used in extraction of sub-array (cf. section IX–8, page 153) or array restructuration (cf. section IX–9, page 154), which are, in the more general case, multi-dimensional indices.

[1] not yet implemented in POLYCHRONY: multiple partial definitions for different elements of an array.

$\ll I_1, \ldots, I_n \gg$

1. **Context-free syntax**

   MULTI-INDEX ::=

   $\boxed{<<}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{>>}$

2. **Types**

   (a) $\forall k, \tau\big(I_k\big) = [0..m_k - 1] \rightarrow \mu_k$

   (b) $\tau\big(\ll I_1, \ldots, I_n \gg\big) = [0..\prod_{k=1}^{n} m_k - 1] \rightarrow \mu_1 \times \ldots \times [0..\prod_{k=1}^{n} m_k - 1] \rightarrow \mu_n$

3. **Semantics**
   The cartesian product $\ll I_1, \ldots, I_n \gg$ defines a tuple of n vectors $II_1, \ldots, II_n$, the size of which is equal to the product of the sizes of the vectors $I_1, \ldots, I_n$. These vectors $II_1, \ldots, II_n$ are such that the tuples obtained by their elements of same index describe successively the respective values of the elements of $I_1, \ldots, I_n$ in embedded loops such that the most external one enumerates the elements of I_1 and the most internal one enumerates the elements of I_n.
   The semantics is described formally in part B, section III–7.2, page 44.

4. **Clocks**

   (a) $\omega\big(I_1\big) = \ldots = \omega\big(I_n\big)$

   (b) Each one of the defined II_k has the same clock as I_k.

IX–13 Iterations of processes

not yet fully imple-mented

Structures of iteration are provided as process expressions[2].

1. **Context-free syntax**

   GENERAL-PROCESS ::=

   ITERATION-OF-PROCESSES

   ITERATION-OF-PROCESSES ::=

   $\boxed{\textbf{array}}$ **ARRAY-INDEX** $\boxed{\textbf{of}}$ **P-EXPR [ITERATION-INIT]** $\boxed{\textbf{end}}$
   $\boxed{\textbf{iterate}}$ **ITERATION-INDEX** $\boxed{\textbf{of}}$ **P-EXPR [ITERATION-INIT]** $\boxed{\textbf{end}}$

   ARRAY-INDEX ::=

   Name $\boxed{\textbf{to}}$ **S-EXPR**

   ITERATION-INDEX ::=

   DEFINED-ELEMENT
   | $\boxed{(}$ **DEFINED-ELEMENT {** $\boxed{,}$ **DEFINED-ELEMENT }*** $\boxed{)}$
   | **S-EXPR**

[2] not yet implemented in POLYCHRONY: creation of the implicit added dimension when necessary; multiple associated indices.

ITERATION-INIT ::=

$\boxed{\textbf{with}}$ **P-EXPR**

REFERENCE-SEQUENCE ::=

S-EXPR $\boxed{\textbf{[}}$ $\boxed{\textbf{?}}$ $\boxed{\textbf{]}}$

The structure of array is used in the SIGNAL language to represent a notion of iteration.

The signals which are defined iteratively have a virtual additional first dimension (with respect to their declaration), the size of which is the number of iterations. Moreover, a virtual index -1 in this first dimension is used to represent the initial value of the considered signal, at the beginning of the iterations. The current value of the signal at a given iteration step may be a function of its value at the previous iteration step.

Note that this representation of bounded iterations using an additional spatial dimension is only a means to represent simply such iterations within the existing semantic context. *In practice, this added dimension has not necessarily to be created.*

Let us first consider the following form:

`iterate (`I_1,...,I_p`) of` P `with` P_{init} `end`

where P is a process expression with equations that may contain the following occurrences of signal expressions:

- in the left hand side:
  X `[` $f(I_1, \ldots, I_p)$ `]` (or just X)

- in the right hand side:
  X `[` $g(I_1, \ldots, I_p)$ `]` (or just X)
  and:
  X `[?][` $h(I_1, \ldots, I_p)$ `]` (or just X `[?]`)

P_{init} is also a process expression with equations that may contain signal expressions of the form X `[` $u(I_1, \ldots, I_p)$ `]` (or just X) in the left hand side.

The equations which are under the scope of a structure of iteration ("iteration of processes") in a given unit of compilation are rewritten as a new system of equations according to the context of rewritting established by the embedding of iteration structures. An indexing function (which can be represented as some list of indexes) corresponds to such a context. The indexing function is a function:

$[0..(n_1 * \ldots * n_p) - 1] \rightarrow [0..n_1 - 1] \times \ldots \times [0..n_p - 1]$

For simplicity, let this function be represented here by the tuple of indexes $I_1, \ldots, I_p$ (in this order): each index has a size equal to $n_1 * \ldots * n_p$. We note $m = n_1 * \ldots * n_p$.

Let us consider also the following "generic" forms of equations in P_{init}:

X `[` $u(I_1, \ldots, I_p)$ `] ::=` E

and in P:

X `[` $f(I_1, \ldots, I_p)$ `] ::=` $k(X$ `[?][` h(I_1,...,I_p) `]` $, Y$ `[` g(I_1,...,I_p) `]` $, \ldots)$

(X, Y represent any variable—Y may be X— defined in the iteration, the functions f, g, h, $u\ldots$ on indexes can represent tuples...; although it is not mandatory, the symbol $\boxed{\texttt{::=}}$ is used here—improperly, in some way—to make visible the fact that, besides the representation of the iteration in an added dimension for the signals, each defined element has several definitions along the iteration.)

Considering this iteration context, the equations affected by this context are rewritten in the following way ("expanded", in some way), as a composition of equations (XX, $YY\ldots$ are new variables, corresponding to the variables defined in the iteration, with the same type as the corresponding variable, but with an additional first dimension of size $m+1$):

- initialization equations:
  $X\,\texttt{[}u(I_1,\ldots,I_p)\texttt{]}$ `::=` E
  is rewritten as the composition of equations:
  $\forall i_1,\ldots,i_p, \forall\varphi\big(I_1\,\texttt{[}i_1\texttt{]}\big),\ldots,\varphi\big(I_p\,\texttt{[}i_p\texttt{]}\big),$
  $XX\,\texttt{[}-1\texttt{]}\,\texttt{[}u(\varphi\big(I_1\,\texttt{[}i_1\texttt{]}\big),\ldots,\varphi\big(I_p\,\texttt{[}i_p\texttt{]}\big))\texttt{]}$ `:=` E
  where -1 refers to the virtual first index of the added dimension.

- equations of the body:
  $X\,\texttt{[}f(I_1,\ldots,I_p)\texttt{]}$ `::=` $k(X\,\texttt{[?]}\,\texttt{[}\mathrm{h}(I_1,\ldots,I_p)\texttt{]}, Y\,\texttt{[}\mathrm{g}(I_1,\ldots,I_p)\texttt{]},\ldots)$
  is rewritten as the composition of equations:
  $\forall l=0,\ldots,m-1,$

  - $XX\,\texttt{[}l\texttt{]}\,\texttt{[}f(I_1\,\texttt{[}l\texttt{]},\ldots,I_p\,\texttt{[}l\texttt{]})\texttt{]}$ `:=`
    $k(XX\,\texttt{[}l-1\texttt{]}\,\texttt{[}h(I_1\,\texttt{[}l\texttt{]},\ldots,I_p\,\texttt{[}l\texttt{]})\texttt{]}, YY\,\texttt{[}l\texttt{]}\,\texttt{[}g(I_1\,\texttt{[}l\texttt{]},\ldots,I_p\,\texttt{[}l\texttt{]})\texttt{]},\ldots)$
  - $\forall j\neq f(I_1\,\texttt{[}l\texttt{]},\ldots,I_p\,\texttt{[}l\texttt{]}),$
    $XX\,\texttt{[}l\texttt{]}\,\texttt{[}j\texttt{]}$ `:=` $XX\,\texttt{[}l-1\texttt{]}\,\texttt{[}j\texttt{]}$

- final results:
  X `:=` $XX\,\texttt{[}m-1\texttt{]}$

This rewritting is some sort of preprocessing. In particular, the typing of a program has to be considered on the rewritten program.

As mentioned above, the iteration indexes can be represented as some list of indexes. A particular case is to have such a list defined as a tuple resulting from the cartesian product of indexes. More generally, the iteration indexes can be specified by any expression denoting a function
$[0..(n_1*\ldots*n_p)-1]\rightarrow[0..n_1-1]\times\ldots\times[0..n_p-1]$ (where the n_i are integer constants).

For a given set of equations, the context of iteration is established, in some unit of compilation, by the whole embedding structure of the iterations containing these equations. As it will be easier to understand it in a regular context, let us consider as typical example the embedding of two structures of iteration, the indexing functions of them, taken separately, are given by cartesian products of indexes: let $\ll I_1,\ \ldots,\ I_p\gg$ for the most external one, and $\ll I_{p+1},\ \ldots,\ I_{p+q}\gg$ for the inner one. Then, for the equations which are under the scope of both structures of iteration, the indexing function (which determines the rewritting) is given by the following cartesian product: $\ll I_1,\ \ldots,\ I_{p+q}\gg$. This rule is generalized following the same principle for any indexing function and for any embedding of structures of iteration.

The "array" notation is a special case of the "iterate" one, inherited from the previous version of the SIGNAL language.

`array` I `to` N `of` P `with` P_{init} `end`
where N is an expression defining a constant integer
is equal to the process defined as follows:

```
(| I := 0..N
 | iterate I of P with P_init end
 |) / I
```

Examples

- ```
  array I to N-1 of
      array J to N-1 of
        U[I,J] := if I=J then 1 else 0
      end
  end
  ```
  defines U as a unit matrix.

- ```
  array I to N-1 of
      array J to N-1 of
        T[I,J] := if J>=I then I+J else 0
      end
  end
  ```
  defines T as a triangular matrix.

- ```
  array I to N-1 of
      D[I] := M[I,I]
  end
  ```
  defines D as a vector equal to the diagonal of matrix M.

- ```
  array I to N-1 of
      T[I] := if I=K then A else (T$)[I]
  end
  ```
  defines the vector T which at each instant keeps the values it had at the previous instant, except in K where it takes the values of A (K and A can be signals).

- ```
  array I to N-1 of
      V[I] := T[I] + V[?][I-1]\\0
  end
  ```
  defines the vector V in which each element, of index i, contains the sum of the first i elements of a vector T.

- ```
  array I to N-1 of
      R := op(T[I],R[?])
  with R := v0
  end
  ```
  defines in R the scalar obtained by the *reduction* of the vector T by the operator op (v0 is the initial value).

- ```
  array I to N-1 of
      Y[I] := FILTER(Y[?][I-1]\\X)
  end
  ```
  defines a cascade of N processes FILTER connected in series. The process model FILTER is declared with one input and one output of some basic type. Each input of an instance of the process FILTER is supplied by the output of the previous process FILTER (the signal X provides the input of the first process FILTER). The vector Y is delivered as output.

- ```
  array I to N of
      F := if I=0 then 1 else I*F[?]
  end
  ```
  defines in F the factorial of N.

- ```
  array I to N−1 of
     FOUND := if FOUND[?]  /= −1
                then FOUND[?]
                else if ELEM = TABLE[I]
                       then I
                       else FOUND[?]
  with FOUND := −1
  end
  ```
  specifies the research of the element `ELEM` in an unsorted `TABLE`.

- With `fulladd` a model of function defined as follows (cf. chapter XI, page 173):
  ```
  function fulladd =
     ( ?  boolean cin, x, y; !  boolean cout, s; )
     (| s := x xor y xor cin
      | cout := (x and y) or (y and cin) or (cin and x)
      |)
  ;
  ```
  then the following model of function defines an unsigned byte adder:
  ```
  function byte_adder =
     ( ?  [8] boolean X, Y; !  [8] boolean S; boolean overflow; )
     (| array i to 7 of
           (overflow, S[i]) := fulladd (overflow[?], X[i], Y[i])
        with overflow := false
        end
      |)
  ;
  ```

- Using the model of function `exchg`:
  ```
  function exchg =
     ( ?  integer a, v; !  integer aa, w; )
     (| aa := v | w := a
      |)
  ;
  ```
  then the following model of function (cf. chapter XI, page 173) defines in `W` a circular permutation of `V`:
  ```
  function Rotate =
     { integer n; } ( ?  [n] integer V; !  [n] integer W; )
     (| array i to n-1 of
           (aa, W[i]) := exchg (aa[?], V[i])
        with aa := V[n-1]
        end
      |)
     where integer aa; ... end
  ;
  ```

- The following model of function sorts the vector `A` in increasing order in `T`:
  ```
  function Sort =
     { integer n; } ( ?  [n] integer A; !  [n] integer T; )
     (| array i to n-1 of
  ```
  ```

```
            array i to (n-2)-j of
               (| T := T[?]
                     next (i :  if T[?][i] > T[?][i+1]
                                   then T[?][i+1] else T[?][i])
                     next (i+1 :  if T[?][i] > T[?][i+1]
                                    then T[?][i] else T[?][i+1])
                 |)
            end
         with T := A
         end
      |)
;
```

(the sequential expression is defined in section IX–14, page 163).

Some other examples are given in the definition of operators on matrices (cf. section IX–16, page 165).

IX–14 Sequential definition

The sequential definition is used mainly for the redefinition of elements of arrays.

T_1 `next` T_2

1. **Context-free syntax**

   SEQUENTIAL-DEFINITION ::=

   S-EXPR $\boxed{\textbf{next}}$ **S-EXPR**

2. **Types**

   (a) $\tau(T_1) = ([0..c_1] \times \ldots \times [0..c_p]) \rightarrow \mu_1$

   (b) $\tau(T_2) = ([0..b_1] \times \ldots \times [0..b_p]) \rightarrow \mu_2$
   with $c_1 \geq b_1, \ldots, c_p \geq b_p$ and μ_1 and μ_2 are comparable types
   (T_1 and T_2 are, in the general case, arrays with the same number of dimensions, but on each of them, T_2 may be smaller than T_1)

   (c) $\tau(T_1$ `next` $T_2) = ([0..c_1] \times \ldots \times [0..c_p]) \rightarrow \mu_1 \sqcup \mu_2$

3. **Semantics**
   T_1 `next` T_2 defines, in the general case, the array which takes the value of T_2 at each point at which T_2 is defined (i.e., is semantically different from nil), and the value of T_1 elsewhere.
   The semantics is described formally in part B, section III–7.2, page 44.

4. **Clocks**

   (a) $\omega(T_1) = \omega(T_2)$

   (b) $\omega(T_1$ `next` $T_2) = \omega(T_1)$

5. **Examples**

   (a) `T := T $  next K : A`
   defines the vector `T` which at each instant keeps the values it had at the previous instant, except in `K` where it takes the values of `A` (`K` and `A` can be signals).

IX–15 Sequential enumeration

The sequential enumeration is a form of iterative enumeration that allows to define arrays using sequential multi-dimensional iterations.

1. **Context-free syntax**

ITERATIVE-ENUMERATION ::=

$\boxed{[}$ **ITERATION {** $\boxed{,}$ **PARTIAL-DEFINITION }*** $\boxed{]}$

PARTIAL-DEFINITION ::=

DEFINITION-OF-ELEMENT
| **ITERATION**

DEFINITION-OF-ELEMENT ::=

$\boxed{[}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{]}$ $\boxed{:}$ **S-EXPR**

ITERATION ::=

$\boxed{\{}$ **PARTIAL-ITERATION {** $\boxed{,}$ **PARTIAL-ITERATION }***
$\boxed{:}$ **DEFINITION-OF-ELEMENT**
| $\boxed{\{}$ **PARTIAL-ITERATION {** $\boxed{,}$ **PARTIAL-ITERATION }***
$\boxed{:}$ **S-EXPR**

PARTIAL-ITERATION ::=

[Name] [$\boxed{\textbf{in}}$ **S-EXPR] [** $\boxed{\textbf{to}}$ **S-EXPR] [** $\boxed{\textbf{step}}$ **S-EXPR]**

Let us consider the following definition of an array T by sequential enumeration:
T `:=` `[`D_1`,`$\ldots$`,`D_m`]`
This definition is equivalent to:
T `:=` D_1 `next` $\ldots$ `next` D_m
where D_1 should be a complete definition of the array.
Let us now consider the following general form of a given D_k:
`{`i_1 `in` b_1 `to` c_1 `step` d_1`,`$\ldots$`,`i_p `in` b_p `to` c_p `step` d_p`}` `:` `[`$f(i_1, \ldots, i_p)$`]` `:` E
It can be considered that the definition of D_k is obtained by the following composition:
(| (i_1,...,i_p) := $\ll b_1$..c_1 step d_1,...,b_p..c_p step $d_p \gg$
| iterate (i_1,...,i_p) of D_k[$f(i_1, \ldots, i_p)$] := E end
|) / i_1, ..., i_p

If the denotation of the indices, `[`$f(i_1, \ldots, i_p)$`]`, is omitted, it is equivalent to `[(`i_1,...,i_p`)]`.
If the lower bound of an index is omitted, it is by default equal to 0. An upper bound can be omitted if it corresponds without ambiguity to the upper bound of the corresponding dimension of the array. If a step is omitted, it is by default equal to 1. The name of an index can be omitted if it has not to be used explicitly.
A D_k with the simple form:
`[`I`]` `:` E
can be considered as being defined by the equation:
D_k`[`I`]` `:=` E

IX–16 Operators on matrices

IX–16.1 Transposition

1. **Context-free syntax**

   TRANSPOSITION ::=
   $\boxed{\textbf{tr}}$ **S-EXPR**

Transposition on matrix `tr` E

1. **Types**
   (a) $\tau(E) = ([0..l-1] \times [0..m-1]) \rightarrow \mu$
   (b) $\tau(\texttt{tr } E) = ([0..m-1] \times [0..l-1]) \rightarrow \mu$

2. **Definition in SIGNAL**
   X `:=` `tr` E
   whose right side of $\boxed{\textbf{:=}}$ represents an expression of transposition of matrix, is equal to the process defined as follows:

```
array i to m − 1 of
   array j to l − 1 of
      X[i,j] := E[j,i]
   end
end
```

3. **Clocks**
   (a) $\omega(\texttt{tr } E) = \omega(E)$

Transposition on vector To create a matrix-column, it is possible to create a matrix-line and then to transpose it as follows:
`tr [`V`]`

IX–16.2 Matrix products

1. **Context-free syntax**

   ARRAY-PRODUCT ::=
   S-EXPR $\boxed{*.}$ **S-EXPR**

2. **Types**
   (a) The elements of the operands of an expression of matrix product have a basic type which is a **Numeric-type**.

3. **Clocks**
   (a) The operators of matrix product are synchronous.

2-a Product of matrices

$E_1 *. E_2$

1. **Types**

   (a) $\tau(E_1)$ = $([0..l-1] \times [0..m-1]) \to \mu_1$

   (b) $\tau(E_2)$ = $([0..m-1] \times [0..n-1]) \to \mu_2$

   (c) $\tau(E_1 *. E_2)$ = $([0..l-1] \times [0..n-1]) \to \mu_1 \sqcup \mu_2$

2. **Definition in SIGNAL**
   X := $E_1 *. E_2$
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of product of matrices, is equal to the process defined as follows:

```
array i to l − 1 of
   array j to n − 1 of
      array k to m − 1 of
         X[i,j] := X[?][i,j] + E1[i,k] ∗ E2[k,j]
      with X[i,j] := 0
      end
   end
end
```

2-b Matrix–vector product

$E_1 *. E_2$

1. **Types**

   (a) $\tau(E_1)$ = $([0..l-1] \times [0..m-1]) \to \mu_1$

   (b) $\tau(E_2)$ = $[0..m-1] \to \mu_2$

   (c) $\tau(E_1 *. E_2)$ = $[0..l-1] \to \mu_1 \sqcup \mu_2$

2. **Definition in SIGNAL**
   X := $E_1 *. E_2$
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of matrix–vector product, is equal to the process defined as follows:

```
array i to l − 1 of
   array k to m − 1 of
      X[i] := X[?][i] + E1[i,k] ∗ E2[k]
   with X[i] := 0
   end
end
```

2-c Vector–matrix product

$E_1 *. E_2$

1. **Types**

   (a) $\tau(E_1) = [0..l-1] \to \mu_1$

   (b) $\tau(E_2) = ([0..l-1] \times [0..m-1]) \to \mu_2$

   (c) $\tau(E_1 *. E_2) = [0..m-1] \to \mu_1 \sqcup \mu_2$

2. **Definition in SIGNAL**
   X `:=` $E_1 *. E_2$
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of vector–matrix product, is equal to the process defined as follows:

```
array j to m-1 of
   array k to l-1 of
      X[j] := X[?][j] + E1[k] * E2[k,j]
   with X[j] := 0
   end
end
```

2-d Scalar product

$E_1 *. E_2$

1. **Types**

   (a) $\tau(E_1) = [0..l-1] \to \mu_1$

   (b) $\tau(E_2) = [0..l-1] \to \mu_2$

   (c) $\tau(E_1 *. E_2) = \mu_1 \sqcup \mu_2$

2. **Definition in SIGNAL**
   X `:=` $E_1 *. E_2$
   whose right side of $\boxed{\texttt{:=}}$ represents an expression of scalar product, is equal to the process defined as follows:

```
array i to l-1 of
   X := X[?] + E1[i] * E2[i]
with X := 0
end
```

Chapter X

Extensions of the operators

X–1 Rules of extension

not yet fully imple-mented

The operators defined in the SIGNAL language are termwise extended to arrays and tuples, provided that there is no possible ambiguity between the new operator resulting from the extension and some other operation[1].

The extension of a given operator defines a new operator, so that termwise extension may be applied recursively.

The semantics of the extension on tuples is described formally in part B, section III–7.1, page 42. The semantics of the extension on arrays is described formally in part B, section III–7.2, page 44.

Instances of processes and conversions follow the same rules of extension than operators.

A given extension is either an extension on tuples, or an extension on arrays. Mixed extensions are not defined. If the types of the arguments of an operator are such that both extension on tuples and extension on arrays can be applied, the extension on tuples applies first.

When an extension is applied, the rules associated with the operator (type relations, clock relations…) apply element by element. Moreover, for the arrays, the constraint that all the elements have the same clock has to be respected.

For tuples, there are different categories of tuples: monochronous tuples, which are signals, and polychronous tuples, which are gatherings of signals (they have not, in general, one proper clock). Monochronous tuples are tuples with named fields and polychronous tuples may be tuples with named or unnamed fields. Whatever is the type of the arguments, the results of an extension on tuples are always tuples with *unnamed* fields (remind that a tuple with unnamed fields can always be assigned to a tuple with named fields with a compatible type). Moreover, if the extension applies on tuples with named fields, the operator applies on the elements of these tuples, independently of their names in the considered tuples. In other words, if X is such a tuple with named fields on which the extension applies, this extension applies effectively on `tuple(` X `)`.

The possibly existing extensions for the operators of the SIGNAL language are easily deduced from the examination of authorized types for the arguments of there operators.

For example, the operator $\boxed{==}$ is defined on *signals* of any types (in particular, on arrays and on monochronous tuples with named fields). Thus the extension of $\boxed{==}$ on arrays or on monochronous tuples with named fields has no purpose. On the other hand, this extension is defined on polychronous tuples (in that case, the result is a polychronous tuple with unnamed fields of Booleans).

The same rules apply, for instance, on the polychronous operator `default`.

[1] not yet implemented in POLYCHRONY: extensions to tuples; some extensions to arrays.

Concerning the other equality operator, $\boxed{=}$, it is defined only on signals of scalar types. Thus the extension on arrays (for example) can apply and in this case, the result is an array of Booleans. The extension on tuples (monochronous or polychronous) applies too.

The extension of the operator `when` on polychronous tuples applies, on the first argument as well as on the second one. But the extension on arrays is not defined in the general case on the second argument since the resulting array would have elements with different clocks.

X–2 Examples

- If `V1` and `V2` are two vectors, the expression `V1 * V2` defines the termwise product of the vectors `V1` and `V2`.
- If `K` is a scalar and `V` a vector, the expression `K * V` defines the vector each element of which is equal to the product of `K` with the corresponding element of `V`.
- If `M1` and `M2` are two matrices, the expression `M1 * M2` defines the termwise product of the matrices `M1` and `M2`.
- If `P` designates a process model which defines two outputs `X` and `Y`,
  the expression `P() when C` defines the signals `X when C` and `Y when C`.
- If `P` designates a process model with two inputs,
  the expression `P ((A,B) when C)` specifies a subsampling by the condition `C` on each one of the inputs of `P`.

Part E

THE MODULARITY

Chapter XI

Models of processes

The language allows to describe signals (synchronized sequences of typed values) and relations between signals by equations; these equations can be grouped together in parameterized models of systems of equations: the *models of processes*. The call of a model in a system is, in principle (when the corresponding model is not compiled separately), equivalent to the direct writing of the equations of this model.

XI–1 Classes of process models

A process model establishes a designation between a name and a set of parameterized equations; any reference to this name is formally replaced by the designated equations.

The set of equations may be empty, or equivalently, simply defined by the keyword `external` (cf. section XII–1, page 191). In that case, it is an *external* process model (or model of external process). Its definition is provided either in a module, or in the environment of the program.

If the process model is external, or if the considered model is compiled separately, the replacement of a reference to this model by its equations remains partial. Such a partial replacement is limited to the **EXTERNAL-GRAPH** of the called process (cf. section XI–6, page 180). The result of the invocation of a model of external process or of a separately compiled process model (which could be not in accordance with its description) can be only theoretically described.

For a model of external process, its graph properties are established by the **EXTERNAL-GRAPH**. For a described process model, the graph properties are established by the composition of the **EXTERNAL-GRAPH** and the body of the model. A good situation is that the **EXTERNAL-GRAPH** verifies the properties deduced from the body of the model.

The following classes of processes are distinguished:

- A process is said *safe* if it does not make any side effect:

  $$\texttt{(|}\ Y_1\ \texttt{:=}\ f\texttt{(}X\texttt{)}\ \texttt{|}\ Y_2\ \texttt{:=}\ f\texttt{(}X\texttt{)}\ \texttt{|)} \quad \equiv \quad \texttt{(|}\ Y_1\ \texttt{:=}\ f\texttt{(}X\texttt{)}\ \texttt{|}\ Y_2\ \texttt{:=}\ Y_1\ \texttt{|)}$$

  Two different instantiations of a *safe* process with the same input values will provide the same results.

  This corresponds to the notion of *iteration of function* (on the inputs), highlighted in part B, section III–8.1, page 50.

- A process is said *deterministic automaton*—or more shortly, *deterministic*—(or *memory safe*), if its only possible "side effects" are changes to its private memory. A *deterministic automaton* is a function of sequences, from initial states, trajectories of the inputs and trajectories of the clocks of the outputs (considered, in some sense, as inputs), into trajectories of the outputs.

  Two different instantiations of a *deterministic automaton* process with the same sequences of input values (and output clocks), and in the same initial conditions, will provide the same sequences of outputs.

  This corresponds to the notion of *deterministic process* (on the inputs), highlighted in part B, section III–8.3, page 50.

  Any *safe* process is *deterministic automaton.*

- A process is *unsafe* in all other cases.

  Two different calls of an *unsafe* process are never supposed to return the same results.

  The following SIGNAL processes are examples of *unsafe* processes:

  – `x := a or x`

  – `(| x := a default ((x$1 init 0)+1) | b:= x when` $\hat{}b$ `|)/x`

The class of the process described by a process model may be precised by a specific keyword in the **EXTERNAL-GRAPH** of the model.

In addition, it is possible to specify non normalized complementary informations (cf. section XI–7, page 184) in the **DIRECTIVES**.

Besides the above characterization of processes, different classes of process models are syntactically distinguished. These are models of:

- functions,
- nodes,
- actions,
- processes.

Any process model called in the program must have a declaration visible in the syntactic context of the call.

A process **MODEL** is defined according to the following syntax:

1. **Context-free syntax**

   MODEL ::=

   PROCESS
   | **ACTION**
   | **NODE**
   | **FUNCTION**

   PROCESS ::=

   $\boxed{\textbf{process}}$ **Name-***model* $\boxed{=}$
   DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

ACTION ::=

$\boxed{\textbf{action}}$ **Name-***model* $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

NODE ::=

$\boxed{\textbf{node}}$ **Name-***model* $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

FUNCTION ::=

$\boxed{\textbf{function}}$ **Name-***model* $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

BODY ::=

DESCRIPTION-OF-MODEL

DESCRIPTION-OF-MODEL ::=

GENERAL-PROCESS
| **EXTERNAL-NOTATION**

XI–1.1 Processes

A *process* (described by a model of process) belongs to the most general class of processes.

There are no required particular relations regarding clocks as well as dependences. It is the job of the compilation (clock calculus, dependence calculus) to synthesize these relations.

A process may be *safe*, *deterministic automaton*, or *unsafe*. This may be specified in the **EXTERNAL-GRAPH**. By default, unless it can be proved different, it is considered as *unsafe*.

XI–1.2 Actions

Actions are processes that are called (activated) at a specific clock, which is designated via a label (cf. part C, section VII–5, page 134). Syntactically, the activation of an action has to be under the scope of such a label, in a labelled process.

An *action* (described by a model of action) has to respect some relations regarding its clocks and dependences:

- Its *tick* (which is a clock greater than the upper bound of the clocks of all the signals designated in the action) is necessarily accessible from the outside of the action. This *tick* is the clock designated by the label under the scope of which the action call is.

- For the dependence relation, each input of an action precedes each output of that action at the product (intersection) of their clocks.

An action may be *safe*, *deterministic automaton*, or *unsafe*. This may be specified in the **EXTERNAL-GRAPH**. By default, unless it can be proved different, it is considered as *unsafe*.

XI–1.3 Nodes

not yet imple-mented

Nodes are essentially *endochronous* processes (cf. part B, section III–8.2, page 50).

Roughly speaking, an endochronous process knows when it has to read its inputs, thus it is autonomous when run in a given environment.

It may be shown that if the clock relations associated with a process can be organized as a tree of clocks, the root of the tree representing the *most frequent* clock (which is the single greatest clock) of the system, then this process is endochronous.

Besides the property that it is endochronous, a *node* (described by a model of node) has to respect some relations regarding its clocks and dependences:

- Its *tick* (which is in that case a clock equal to the upper bound of the clocks of all the signals designated in the node) is necessarily the clock of an input or output of the node.
  In addition, the clock of any input or output signal of the node should be considered as explicitly defined (without clock calculus synthesis). For those input or output signals the clock of which is not explicitly defined, their clock is fixed to the *tick*.
  Thus, there is no synthesis from the clock calculus for the clocks of inputs and outputs, but only verification. However, there is possibly such a synthesis for the clocks of internal signals.

- For the dependence relation, each input of a node precedes each output of that node at the product (intersection) of their clocks.

A node may be *safe* or *deterministic automaton*. This may be specified in the **EXTERNAL-GRAPH**. By default, unless it can be proved *safe*, it is considered as *deterministic automaton*.

XI–1.4 Functions

A function is a process that specifies an *iteration of function* such as defined in part B, section III–8.1, page 50.

A *function* (described by a model of function) is a particular case of node and has to respect all the relations respected by a node regarding its clocks and dependences (cf. section XI–1.3, page 175). In addition, all the inputs and outputs of a function must have the same clock.

A function is constant on time and does not produce any side effect. In particular, it cannot contain delay operators (or other operators derived from delay), that define some memory.

Note that it is nevertheless possible to specify some assertions on the input signals (for instance) of a function. For example, the equation $x \hat{=}$ `when` $(x > 0)$ specifies that when it is present, x must be positive.

A function is necessarily *safe* (this has not to be specified in the **EXTERNAL-GRAPH**).

XI–2 Local declarations of a process model

The local declarations of a process model may be declarations of signals (or tuples), declarations of state variables, declarations of constants, declarations of types, declarations of labels, declarations of references to signals with extended visibility, or declarations of local models.

1. **Context-free syntax**

DECLARATION ::=

S-DECLARATION
| **DECLARATION-OF-STATE-VARIABLES**
| **DECLARATION-OF-CONSTANTS**
| **DECLARATION-OF-TYPES**
| **DECLARATION-OF-LABELS**
| **REFERENCES**
| **MODEL**

A given zone of local declarations constitutes a given *level* of declarations; this level is that of the process expression that defines this zone. When this expression is the expression that defines the process model, this zone is said the zone of the local declarations of the model. When this expression is the expression that defines the external graph of the model, this zone is said the zone of the local declarations of the external graph.

The zones of declaration of the formal parameters and of the inputs and outputs of a process model constitute a same *level* of declarations, the one of the model.

The levels of declarations are ordered in the following way:

- the level of a model is greater than the level of any sub-expression of this model;
- the level of an expression is greater than the level of any sub-expression of this expression;
- the level of a model is greater than the level of any local model declared in this model.

A local declaration of a model in a given level is visible (and thus, this model can be called as **INSTANCE-OF-PROCESS**) in this whole level and in all lower levels, everywhere it is not hidden by a declaration with the same name in a lower level. In particular, a model Q declared in the zone of the local declarations of a model P can be called in the expression associated with P and in the expressions associated with the other sub-models of P. For these expressions, it possibly hides a model with the same name that, without it, would be visible.

The set of sub-models declared in a model P cannot contain two models with the same name. More generally, any two objects (models, types, signals, etc.) declared in a same level of declaration cannot have the same name (see below).

The parameters declared in a process model are visible (and thus, may be referenced) in this whole process model (in particular, the other parameters, the inputs and outputs, etc.) and in all the embedded process models, everywhere they are not hidden by a declaration with the same name in a lower level.

The constants declared in a given level are visible in this whole level and in all lower levels, everywhere they are not hidden by a declaration with the same name in a lower level.

The types declared in a given level are visible in this whole level and in all lower levels, everywhere they are not hidden by a declaration with the same name in a lower level.

The declaration of labels and their visibility obey to specific rules, which are more detailed in section XI–3, page 178.

As a general rule, the local declarations of signals (or tuples) and state variables correspond to the confining of these objects (cf. part C, section VII–4, page 133) to the corresponding level and the lower ones. However, the visibility of signals, tuples and state variables obey to specific rules, which are more detailed in section XI–4, page 178.

The names of declared objects (models, signals or tuples, state variables, parameters, constants, types, labels) can mutually mask themselves. In a given level, there cannot have two such identical names.

A given compiler may adapt the visibility rules for some classes of objects in the following way: in the level where it is declared, a given object can be used only in a syntactic position that *follows* its declaration (in this case, the order of declarations is significant). The rules for names redefinitions may be adapted accordingly.

XI–3 Declarations of labels

1. **Context-free syntax**

DECLARATION-OF-LABELS ::=
  $\boxed{\textbf{label}}$ **Name-***label* **{** $\boxed{\textbf{,}}$ **Name-***label* **}*** $\boxed{\textbf{;}}$

The labels declared in a process model, at any declaration level of this model, are visible (and can be referenced) anywhere in this model, except in its interface (parameters, inputs and outputs, external graph). The labels declared in the external graph of a process model are visible (and can be referenced) anywhere in this model.

However, the labels declared in a process model are not visible in the sub-models of that model.

A label declared in a model cannot have the same name as any other object declared in that model (it cannot be masked).

XI–4 References to signals with extended visibility

not yet imple-mented

1. **Context-free syntax**

REFERENCES ::=
  $\boxed{\textbf{ref}}$ **Name-***signal* **{** $\boxed{\textbf{,}}$ **Name-***signal* **}*** $\boxed{\textbf{;}}$

The rules for the visibility of signals in the previous versions of SIGNAL were that this visibility was always limited to the process model in which the signal was declared, excluding the sub-models of that model.

This version offers the possibility to extend the visibility of signals (or tuples), and state variables, with the same rules as for most of the other objects of the language. In that case, a signal (or tuple, or state variable) declared in a given level is visible in this whole level and in all lower levels, everywhere it is not hidden by a declaration with the same name in a lower level.

However, some freedom is left to the compilers to accept or not (possibly according to specific options) signals with extended visibility. The three following cases may be distinguished:

1. Signals with extended visibility are not allowed.

2. Signals with extended visibility are allowed, but the use of such a signal must be explicitly referenced as such when it crosses a frontier of process model with respect to its declaration.

   Such a use is pointed by a “ref” declaration, under the scope of which is the considered use (with the general scoping rules, restricted here to the considered process model).

   A signal with extended visibility cannot be used if it has been hidden by the declaration of another object with the same name.

   A “ref” declaration cannot mask some object with the same name.

3. Signals with extended visibility are allowed, and their use may be explicitly referenced (previous case), though it is not mandatory.

XI–5 Interface of a model

The interface of a model contains an optional description of its formal static parameters, followed by a description of its visible part. This one is composed of the lists (possibly empty) of its input and output signals, and an optional description of the external behavior of the model.

1. **Context-free syntax**

   DEFINITION-OF-INTERFACE ::=

   INTERFACE

   INTERFACE ::=

   [PARAMETERS] $\boxed{\textbf{(}}$ **INPUTS OUTPUTS** $\boxed{\textbf{)}}$ **EXTERNAL-GRAPH**

   PARAMETERS ::=

   $\boxed{\textbf{\{}}$ **[{ FORMAL-PARAMETER }$^+$]** $\boxed{\textbf{\}}}$

   FORMAL-PARAMETER ::=

   S-DECLARATION
   | **DECLARATION-OF-TYPES**

   INPUTS ::=

   $\boxed{\textbf{?}}$ **[{ S-DECLARATION }$^+$]**

   OUTPUTS ::=

   $\boxed{\textbf{!}}$ **[{ S-DECLARATION }$^+$]**

   The formal parameters of the interface of a model can contain type parameters. These type parameters necessarily appear under the form of names of types, without a **DESCRIPTION-OF-TYPE** definition (cf. part C, section V–7, page 84).

2. **Types**
   The list of inputs (respectively, outputs) declared in the interface of a process model named P constitutes a tuple the type of which is denoted $\tau(?P)$ (respectively, $\tau(!P)$).

   The type of the tuple of inputs and the type of the tuple of outputs are tuples with unnamed fields.

   Thus:

   (a) if the inputs and outputs of a process model P appear as
   `(?` μ_1 E_1`;` `...` μ_m E_m`;` `!` ν_1 S_1`;` `...` ν_n S_n`;)`
   (to simplify the presentation, we consider that each designation of type qualifies one single name of signal or tuple; the generalization to the case with lists of names is trivial)
   then
   $\tau(?P) = (\tau(\mu_1) \times \ldots \times \tau(\mu_m))$
   $\tau(!P) = (\tau(\nu_1) \times \ldots \times \tau(\nu_n))$

3. **Semantics**

   A model must have at least one input, or one output, or one communication with non null clock with some external process.

   The names of parameters, input signals and output signals must be mutually distinct.

   The declarations of the input signals (**INPUTS**) and the output signals (**OUTPUTS**) of a model are declarations of sequences. The declarations of formal parameters (**PARAMETERS**) can contain declarations of parameter types (**DECLARATION-OF-TYPES**) and declarations of constant sequences (**S-DECLARATION**). In particular, the declarations of sequences can contain tuples of parameters or signals. The declaration of a model sets up a context in which:

   - the parameter types define formal types, in a way similar to the declarations of types described in part C, chapter V, "Domains of values of the signals";
   - a type is associated with the declared parameters, input signals, and output signals, in a similar way to the association of a type to local signals of a process (cf. part C, chapter VII, "Expressions on processes"), according to the rules defined in the chapter "Domains of values of the signals".

   The invocation of a model sets up an expansion context in which:

   - an effective type is associated with the parameter types, in a similar way to the definition of type obtained by a **DESCRIPTION-OF-TYPE** (cf. part C, section V–7, page 84): if μ is the effective parameter corresponding, positionally, to the formal parameter type `type` A`;` then the type A is defined as being equal to the type μ in the context of this invocation of model;
   - a value (or a tuple of values) is associated with each identifier of formal parameter, and a signal (or a tuple of signals) is associated with each name of input or output signal (or tuple).

   The declaration of a process model induces the existence of a given order on the parameters (whatever they are parameter types or not), an order on the input signals of the model, and an order on its output signals. Each one of these orders is the order of specification of the objects of the considered class (parameter, input or output) in the interface. Any positional invocation of the model is made respectively to these orders.

   Example: a process model P the interface of which is specified as
   `{`Y_1`; ...` Y_l`;} ( ?` A_1`; ...` A_n`; !` B_1`; ...` B_m`;)`
   can be called such as
   `(`BB_1`, ...,` BB_m`) :=` P `{`YY_1`, ...,` YY_l`} (`AA_1`, ...,` AA_n`)`
   where each signal or parameter XX_i corresponds to the signal or parameter X_i.

XI–6 Graph of a model

The **EXTERNAL-GRAPH** of a model allows to specify clock and graph properties of the model, such as the properties necessary and sufficient to be able to use this model after a separate compilation. These properties may be provided by the designer or calculated by the compiler. They refer to input and output signals of the model.

1. **Context-free syntax**

EXTERNAL-GRAPH ::=

[PROCESS-ATTRIBUTE] [SPECIFICATION-OF-PROPERTIES]

PROCESS-ATTRIBUTE ::=

$\boxed{\textbf{safe}}$
| $\boxed{\textbf{deterministic}}$
| $\boxed{\textbf{unsafe}}$

SPECIFICATION-OF-PROPERTIES ::=

$\boxed{\textbf{spec}}$ **GENERAL-PROCESS**

The **PROCESS-ATTRIBUTE** allows to qualify the corresponding model as *safe* (keyword `safe`), *deterministic automaton* (keyword `deterministic`), or *unsafe* (keyword `unsafe`)—cf. section XI–1, page 173. It must be in accordance with the syntactic class of the model.

The **SPECIFICATION-OF-PROPERTIES** of an **EXTERNAL-GRAPH** uses a process expression that can make reference to the formal parameters and input and output signals of the **MODEL**. Any other identifier used in this expression is that of a local object (signal, process model, etc.), that must have a declaration in this expression.

When the **EXTERNAL-GRAPH** is that of a described process model, the process defined by the model is obtained by the composition of this **EXTERNAL-GRAPH** and of the body of this model. A recommended situation is that the properties established by the **EXTERNAL-GRAPH** may be deduced from the properties verified by the body of the model.

When the **EXTERNAL-GRAPH** is that of an external process model, the properties it describes establish the properties of the model for any invocation of this model.
In that case, the invocation X `{`V_1`, ...,` V_l`}` of an external process model

```
process X = {F1; ... Fl;}
     ( ?  E1; ... Em;
       !  S1; ... Sn; )
     spec C;
```

is equal to the process defined as follows:

```
(| X {V1, ..., Vl}
 | C
 |)
```

If $\mathcal{C}_1$ is the syntactic context of expansion established by the invocation of the model of external process by the association of a value with each identifier of formal parameter, and by the association of a signal with each input or output signal name, then, the invocation of this model results in the context of expansion $\mathcal{C}_2$ equal to $\mathcal{C}_1$ enriched by the equations (in particular, clock equations and dependences) resulting from the construction of the **EXTERNAL-GRAPH**.

XI–6.1 Specification of properties

The **SPECIFICATION-OF-PROPERTIES** is described by a usual process expression, the elementary expressions of which are typically an instance of process (whixh may be, in that case, an instance of a model of synchronization), a definition of signals, a clock equation, or an expression of dependence.

XI–6.2 Dependences

An expression of explicit **DEPENDENCES** may appear in the **EXTERNAL-GRAPH** of a **MODEL**, but also in its body. The purpose of a specification of dependences in the external graph is to make explicit dependences between input and output signals of the model, or to establish these dependences in the case of a model of external process. The explicit dependences between signals are defined with the following syntax:

1. **Context-free syntax**

ELEMENTARY-PROCESS ::=

DEPENDENCES

DEPENDENCES ::=

SIGNALS { $\boxed{-->}$ **SIGNALS }***
| $\boxed{\{}$ **SIGNALS** $\boxed{-->}$ **SIGNALS** $\boxed{\}}$ $\boxed{\textbf{when}}$ **S-EXPR**

SIGNALS ::=

ELEMENTARY-SIGNAL
| $\boxed{\{}$ **ELEMENTARY-SIGNAL {** $\boxed{,}$ **ELEMENTARY-SIGNAL }*** $\boxed{\}}$

ELEMENTARY-SIGNAL ::=

DEFINED-ELEMENT
| **Label**

We distinguish first the case where some of the "signals" for which dependences are specified are labels (cf. part C, section VII–5, page 134). In that case, for a label XX, the designated signal is either ! XX (that is preceded by all the signals of the process labelled by XX), or ? XX (that precedes all the signals of the process labelled by XX), depending that XX appears at the left side or at the right side of the dependence arrow.

If XX is a label:

- $XX \;\; --> \;\; E$

  1. **Definition in SIGNAL**

     ! $XX \;\; --> \;\; E$

- $E \;\; --> \;\; XX$

  1. **Definition in SIGNAL**

     $E \;\; -->$? XX

Then, with the designated signals:

- $E_1 \;\; --> \;\; E_2 \;\; --> \;\; E_3$

  1. **Definition in SIGNAL**

(| E_1 --> E_2
| E_2 --> E_3
|)

Note that for the particular case where a label XX appears as
E_1 --> XX --> E_3
this expression is equivalent to:

(| E_1 --> ? XX
| ! XX --> E_3
|)

- {X_1, ..., X_n} --> E

  1. **Definition in SIGNAL**

     (| X_1 --> E
     ⋮
     | X_n --> E
     |)

- E --> {Y_1, ..., Y_m}

  1. **Definition in SIGNAL**

     (| E --> Y_1
     ⋮
     | E --> Y_m
     |)

- {E --> {Y_1, ..., Y_m}} when B

  1. **Definition in SIGNAL**

     (| {E --> Y_1} when B
     ⋮
     | {E --> Y_m} when B
     |)

- {X --> Y} when B

  1. **Types**

     (a) $\tau(B) \sqsubseteq boolean$

  2. **Semantics**

     The result of the expression {X --> Y} when B
     is to add to the dependence graph a dependence from X to Y labelled by the condition B, representing the clock at which B has the value $true$.
     The semantics of such a dependence is described formally in part B, section IV–3.1, page 60.

3. **Graph**

   (a) $X \xrightarrow{B} Y$

4. **Examples**

   (a)
```
(| S1 ::  ERASE (X)
 | S2 ::  DISPLAY (X)
 | S1 --> S2 |)
```
   allows to sequentialize the actions `ERASE` and `DISPLAY`.

XI–7 Directives

not yet fully imple-mented

The **DIRECTIVES** allow to associate specific informations, or *pragmas*, with the objects of a program[1]. These informations may be used by a compiler or another tool.

A **PRAGMA** contains a **Name**, the list of the designations of objects with which it is associated, and a **Pragma-statement**.

PR `{`X_1`, ..., `X_n`}` `"`YYY`"`

1. **Context-free syntax**

   DIRECTIVES ::=

   $\boxed{\textbf{pragmas}}$ **{ PRAGMA }**$^+$ $\boxed{\textbf{end}}$ $\boxed{\textbf{pragmas}}$

   PRAGMA ::=

   Name-*pragma*
   [$\boxed{\textbf{\{}}$ **PRAGMA-OBJECT {** $\boxed{\textbf{,}}$ **PRAGMA-OBJECT }*** $\boxed{\textbf{\}}}$ **]**
   [Pragma-statement]

   PRAGMA-OBJECT ::=

   Label
   | **Name**

   Pragma-statement ::=

   String-cst

2. **Semantics**

   The pragma with name PR and with (optional) statement `"`YYY`"` is associated with each one of the objects designated by $X_1, \ldots, X_n$.
   The designations (that should reference objects which are visible at the level of the model) can be:

   - labels (in that case, the designated object is a process expression),
   - names of signals, parameters, constants, types, etc. (the designated object is the corresponding signal, parameter, constant, type, etc.).

   By default (when there is no designated object), the pragma is associated with the current process model.

[1] not yet implemented in POLYCHRONY: association of a pragma with named objects.

A pragma has no semantic effect. It can be ignored by a compiler, or it can trigger a specific processing.

3. **Examples**

   The following pragmas are recognized in the INRIA POLYCHRONY environment:

   (a) General information

   - `Comment`:
     - Associated with the current model.
     - Comment on this model.
   - `Note`:
     - Associated with the current model.
     - Comment on this model.

   (b) Compilation directives

   - `main`:
     - Associated with the current model.
     - In a module, means that the corresponding model is an "entry point" of this module: it constitutes a *compilation unit* (cf. section XII–1, page 191).
   - `unexpanded`:
     - Associated with the current model.
     - Means that the model is not expanded when it is called. Each one of its instances is compiled *in its calling context* and has an associated object (graph) in the internal representation.
   - `separate`:
     - Associated with the current model.
     - Means that the model is not expanded when it is called. It is compiled separately and has an associated object (graph) in the internal representation.

   (c) Partitioning information

   - `RunOn`:
     - Associated with the current model, P.
     - The statement of this pragma is a string representing a constant integer value i.
     - Each "node" (or vertex) of the internal representation of P (this internal representation is a graph) is attributed by the value i.
       When a partitioning based on the use of the pragma `RunOn` is applied on an application, the global graph of the application is partitioned according to the n different values of the pragma so as to obtain n sub-graphs, corresponding to n sub-models. The tree of clocks and the interface of these sub-models may be completed in such a way that they represent endochronous processes.

   (d) Separate compilation

   - `Black_Box`:
     - Associated with the current model.
     - Represents the "black box" abstraction of a model (may be the result of a compilation). Only the interface of the model, including its external graph, is represented: its body is empty.

- `Grey_Box`:
  - Associated with the current model.
  - Represents a "grey box" abstraction of a model. It contains an external graph that represents clock and dependence relations of the interface, but also a restructuring of the model into *clusters* together with a representation of the *scheduling* of these clusters (clock and dependence relations between these clusters). Each cluster is represented as a "black box" abstraction which is such that any input of the cluster precedes any of its outputs.
- `DelayCluster`:
  - Associated with the current model.
  - May qualify one of the clusters of a "grey box" abstraction when code generation is expected from this abstraction: in that case, one of the clusters, the "delay cluster" (represented, like the other ones, by its "black box" abstraction), groups together the delay operations of the model and is preceded by each one of the other clusters (in the generated code, memories will be updated at the end of one instant).

(e) Code generation directives

The pragmas `C_CODE`, `CPP_CODE`, `Java_CODE` are specific to code generation.

They are associated with the current model.

Their statement is a "parameterized" string representing a piece of code in the considered implementation language. Each call of the model is translated by this string in the generated code, after substitution of the encoded parameters by the corresponding signals in the considered call.

The following encoded parameters may be used in the string:

- `&p`j (where j is a constant integer value) represents the j^{th} parameter of the call;
- `&i`j (where j is a constant integer value) represents the j^{th} input signal of the call;
- `&o`j (where j is a constant integer value) represents the j^{th} output signal of the call;
- `&n` represents the name of the model.

- `C_CODE`: is used for C code generation.
- `CPP_CODE`: is used for C++ code generation.
- `Java_CODE`: is used for Java code generation.

(f) Profiling directives

- `PROCESSOR_TYPE`:
  - Associated with the current model.
  - The statement of this pragma is a string representing a name, for example, `"DSP"`, that should be the name of a file `DSP.LIB` containing a module that defines the cost of each operator by particular models.
  - When profiling (performance evaluation) is required on a given program implemented on some processor represented as a model with the `PROCESSOR_TYPE` pragma, a morphism of this program is applied, that defines a new program representing cost evaluation of the original program. The image of the original program by this morphism uses the library designated by the pragma to interpret the cost evaluation operators.

(g) Link with the SIGALI prover

- `SIGALI`:
  - Associated with the current model.

– Used for models contained in a specific library dedicated to the SIGALI prover. The calls of these models are external calls that are interpreted when translated into the SIGALI representation.

XI–8 Models as types and parameters

not yet implemented

The notion of type presented so far is enriched with the notion of *model type*, that represents the interface of a process model. Then model types can be used to specify formal process models as formal parameters of process models: a process model with the corresponding model type as interface must then be provided as effective parameter.

Model types

A model type is an interface of process model.

The following rules for a **DEFINITION-OF-TYPE** extend those given in part C, section V–7, page 84.

The rule for a **DEFINITION-OF-INTERFACE** extends those given in section XI–5, page 179.

`process` T `=` I
(the corresponding **DECLARATION-OF-TYPE** is: `type process` T `=` I`;`),
or `action` T `=` I, etc.

1. **Context-free syntax**

   DEFINITION-OF-TYPE ::=

   $\boxed{\textbf{process}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**
   | $\boxed{\textbf{action}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**
   | $\boxed{\textbf{node}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**
   | $\boxed{\textbf{function}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**

   DEFINITION-OF-INTERFACE ::=

   Name-*model-type*

2. **Types**

   (a) The declaration `type process` T `=` I`;` defines the model type with name T as being equal to the interface of *process* model I.
   Let us denote this equality:
   $\tau(T) = \mathit{interface}_{\texttt{process}}(I)$

3. **Semantics**

   - The same scoping rules as for other types apply to model types.

4. **Properties**

   (a) With the declarations
   `type process` A `=` I`;`

and `type process` B `=` I`;`
then $\tau(A) = \tau(B) = interface_{\texttt{process}}(I)$.
Some implementations may not ensure this property.
On the opposite, the declarations
`type process` A `=` I`;`
and `type function` B `=` I`;` (for instance)
define distinct model types.

5. **Examples**

   (a) `type process T = ( ? integer a; ! integer b; );` declares the model type `T`.

   (b) `type process TT = T;` declares the model type `TT` which is equal to `T`.

   (c)
   ```
   process PP =
     T
     (| ...|);
   ```
   declares the process model `PP` with its interface specified by `T`.

Models as parameters

The following rules for a **FORMAL-PARAMETER** extend those given in section XI–5, page 179. The rule for **S-EXPR-PARAMETER** extends those given in part C, section 2-a, page 97.

1. **Context-free syntax**

   FORMAL-PARAMETER ::=

   FORMAL-MODEL

   FORMAL-MODEL ::=

   $\boxed{\textbf{process}}$ **Name-***model-type* **Name-***model*
   | $\boxed{\textbf{action}}$ **Name-***model-type* **Name-***model*
   | $\boxed{\textbf{node}}$ **Name-***model-type* **Name-***model*
   | $\boxed{\textbf{function}}$ **Name-***model-type* **Name-***model*

   S-EXPR-PARAMETER ::=

   Name-*model*

   The formal parameters of the interface of a model P can contain model parameters, that appear as a formal name of model, say Q, typed with a model type, say T, which is visible in the current syntactic context: typically, `process` T Q.

2. **Semantics**
   To complete the description that was given in section XI–5, page 179, the declaration of a model sets up a context in which the model parameters define formal models, that is to say, models for which only the interface (described by a model type) is known (analogous to model of external processes).
   The same scoping rules as for other parameters apply to model parameters.

In the body of the process model P, the formal model Q is invoked using the usual syntax for the invocation of models.

The invocation of a model sets up an expansion context in which an effective model, designated by its name (which must be the name of a process model visible in the context of this invocation), is associated with each model (positional association, just like other parameters).

3. **Examples**

   (a)
   ```
   process P =
      { process T Q; }
      ( ?  ... !  ... )
      (| ... x := Q(y) ... |);
   ```
   declares the process model `P` wich has a model parameter `Q`, the interface of which is described by the model type `T` (in that case, it has, for instance one input and one output).
   The model `P` must be called with a visible process model as effective parameter; the interface of this process model must be equal to `T`.
   For example: `... P{PP}(...)...`

Chapter XII

Modules

XII–1 Declaration and use of modules

A module is a named set of declarations of constants, types and models.

The syntax of **DECLARATION-OF-CONSTANTS**, **DECLARATION-OF-TYPES**, **PROCESS**, **ACTION**, **NODE** and **FUNCTION** given below extends the syntax of these declarations such as defined in part C, section V–8, page 85, part C, section V–7, page 84 and part E, section XI–1, page 173. The presence of the `private` attribute is reserved to declarations which are in a module. The syntax of **EXTERNAL-NOTATION** may be used as well for a **DESCRIPTION-OF-CONSTANT**, a **DESCRIPTION-OF-TYPE** or a **DESCRIPTION-OF-MODEL**, either they appear in a model or in a module. It is provided in this section.

The importation of objects of a module in another module or in a model is done via a `use` importation command that may be found in a list of **DECLARATION**s. Then, the syntax of **DECLARATION** given below extends that defined in part E, section XI–2, page 176.

1. **Context-free syntax**

MODULE ::=

$\boxed{\textbf{module}}$ **Name-***module* $\boxed{\textbf{=}}$
[DIRECTIVES] { DECLARATION }$^+$ $\boxed{\textbf{end}}$ $\boxed{\textbf{;}}$

DECLARATION-OF-CONSTANTS ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{constant}}$ **SIGNAL-TYPE**
DEFINITION-OF-CONSTANT { $\boxed{\textbf{,}}$ **DEFINITION-OF-CONSTANT }*** $\boxed{\textbf{;}}$

DECLARATION-OF-TYPES ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{type}}$
DEFINITION-OF-TYPE { $\boxed{\textbf{,}}$ **DEFINITION-OF-TYPE }*** $\boxed{\textbf{;}}$

PROCESS ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{process}}$ **Name-*model*** $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

ACTION ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{action}}$ **Name-*model*** $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

NODE ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{node}}$ **Name-*model*** $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

FUNCTION ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{function}}$ **Name-*model*** $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

EXTERNAL-NOTATION ::=

$\boxed{\textbf{external}}$ **[String-cst]**

DECLARATION ::=

IMPORT-OF-MODULES

IMPORT-OF-MODULES ::=

$\boxed{\textbf{use}}$ **IMPORTED-OBJECTS {** $\boxed{,}$ **IMPORTED-OBJECTS }*** $\boxed{;}$

IMPORTED-OBJECTS ::=

Name-*module*

Pragmas may be associated with the objects of a module in the same way they can be associated with the objects of a model (cf. section XI–7, page 184). When there is no designated object for a pragma specified in a module, it is by default associated with the current module.

The set of declarations of a module constitutes a same level of declarations: the level of a module. The level of a module is greater than the level of any model declared in this module. With the usual rule, there cannot have two objects with the same name declared in a module.

The visibility of the objects declared in a module may be restricted to this module using the attribute `private`: when a declaration of constants, types or model is preceded by the keyword `private` (`private constant ...`, `private type ...`, `private process ...`, etc.), then the visibility of the corresponding objects is confined to the module that contains that *private* declaration, even if this module is referenced by a `use` command.

In a module M, but also in a model, the description of a constant, a type or a model can be given by an expression of the SIGNAL language, or it can be described as external (relatively to the current module) by using the `external` attribute (or equivalently, by the absence of description).

The objects declared in a module can be totally or partially imported from a model or another module thanks to the `use` command. Such a module provides a context of definition for some of the objects described as external in the model or the module containing the `use` command (and visible at this level). These external objects are *redefined* in this way if they are imported (as corresponding objects with the same name) from a *used* module, or transitively, from a module imported in an imported module. The redefined constants must have the same type as that appearing in their declaration as external (or a redefinition of this type if it is an external type). In the same way, the redefined models must have compatible interfaces.

More generally, any object described as external in some zone of declarations L may *inherit* a (re-)definition from any context, visible in L, that provides such a definition.

Though it is not mandatory, it may be a good policy to systematically declare as external in a module M the objects referenced in M, but imported by a `use` command from another module. However, in this case, they should be used only as external objects: for example, if some signal is declared with an external type, only polymorphic operators could be applied to it.

A model or a module are a *compilation unit* when all the objects they use (except predefined or intrinsic ones) have a declaration (which may be that of an external object) in this entity, taking into account the `use` commands contained in it. In any case, a module necessarily constitutes a compilation unit.

The objects whose definitions or redefinitions are imported in a model or module P by a `use` command situated in a zone of local declarations of P are made visible at the level of the expression containing these local declarations and at all lower levels (with the usual scoping rules, everywhere another object with the same name is not declared at such a level). More precisely, a `use` command inside the local declarations of an expression establishes a new level of declaration which is just greater than that of the expression. For example, an expression

E `where` L`; use` M`; end`

may be considered, from the point of view of the scoping rules, as equivalent to the following one:

`(`E `where` L`; end) where` *`Decl`*(M) `end`

where *Decl*(M) represents the declarations of M. This equivalence holds wherever the `use` command is located in the local declarations.

A similar rule also applies for a `use` command located in the declarations of a module.

The importable objects of a module are the objects of this module that are not declared as *private*. The objects imported by a `use` command are all the importable objects of the module.

When several `use` commands appear at a same level of declaration, their syntactic order determines a corresponding nesting of the importations, thus avoiding multiple definitions of a same object at a given level. For example, to:

E `where` L`; use` M_1`; ...; use` M_n`; end`

corresponds the following nesting:

`(((`E `where` L`; end) where` *`Decl`*(M_n) `end) ...)  where` *`Decl`*(M_1) `end`

(the declarations of M_1 are visible in M_n, but the converse is not true).

In this way, if several objects with the same name are imported in a given context from different modules, the single one which is effectively visible is the one from the last module containing it in the ordered list of the `use` commands. Note that the rule applies differently for external objects since external definitions are overloaded by corresponding non external ones.

The nesting of declarations also allows to overload, in some way, declarations of imported modules (libraries) by local declarations, since the local ones have priority.

When several modules are specified in a same `use` command, the corresponding declarations are imported at the same level. For example,
E `where` L`; use` M_1`, ...,` M_p`;end`
would correspond to:
`(`E `where` L`; end) where` *Decl*(M_1) `...` *Decl*(M_p) `end`
In this case, there is a potential risk of conflicts of the declarations imported from different modules.

In a given compilation unit, when an object is described as external (for example, using the `external` notation), then:

- either it is defined in an imported module,
- either is is defined in the context in which this compilation unit is used,
- or it is externally defined, in another language for instance, in the implementation environment of the compilation unit.

The description of an object as external may be followed by a string, such as `external "`X`"`, which is an attribute allowing to describe specific characteristics of the implementation of this object: implementation language, for instance (this is indeed a short notation for a specific pragma).

The name M used in a command "`use` M`;`" is the name of a module visible in the design environment. The way this module is made available is not normalized.

As an example, in the INRIA POLYCHRONY environment, there is an environment variable,
SIGNAL_LIBRARY_PATH,
which defines the paths at which library files may be found in the design environment. Such a file has a name with the suffixe ".LIB" or ".SIG", for example, "M.LIB" or "M.SIG" (in principle, the first part of the name could be different from "M"), and contains the definition of a module named M, in SIGNAL.

Examples

- ```
  module Stack =
     use my_elem;
     type elem;
     type stack;
     process initst = ( ! stack p;);
     process push = ( ? stack p; elem x; ! event except; )
        spec (| x ^> except | x --> except |);
     process pop = ( ? stack p; ! elem x; ! event except; )
        spec (| x ^# except |);
     ...
  end;
  ```
  ```

Chapter XIII

Intrinsic processes

Intrinsic process models constitute libraries of processes that may be used in SIGNAL programs. These models have not to be declared. The names of the intrinsic process models are not reserved words of the SIGNAL language.

XIII–1 Assertions

not yet imple-mented

An assertion is a process with no output which specifies that a Boolean signal must have the value $true$ each time it is present. It has the syntax of a process call with no output.

Assertions are used in particular to specify hypotheses on some inputs of a model.

`assert(` B `)`

1. **Types**

   (a) $\tau(B) = boolean$

2. **Semantics**

   An assertion is obtained by a call to the intrinsic process model:

   ```
   process assert = ( ?  boolean x; !  );
   ```

   A property specified by an assertion can be assumed by the clock calculus.

3. **Definition in SIGNAL**

   `assert(` B `)`

   is equal to the process defined as follows:

   (| B ^= `when` B
   |)

4. **Examples**

   (a) The process

   `assert( A < 5 )`

   expresses that the values of `A` must be always lower than 5 (when `A` is present).

XIII–2 "Left true" process

The following `left_tt` process is defined as intrinsic process model:

```
process left_tt = ( ?  boolean y, z; !  boolean x; )
   (| x := y default false when ^z |)
;
```

XIII–3 Mathematical functions

The following mathematical functions are defined as intrinsic process models. They correspond to functions of the "math.h" library of the language C. A full description of them may be found in the documentation of this library.

- arc cosine function:
  `function acos = ( ?  dreal x; !  dreal y; );`
- arc sine function:
  `function asin = ( ?  dreal x; !  dreal y; );`
- arc tangent function:
  `function atan = ( ?  dreal x; !  dreal y; );`
- arc tangent function of two variables:
  `function atan2 = ( ?  dreal x1; dreal x2 !  dreal y; );`
- cosine function:
  `function cos = ( ?  dreal x; !  dreal y; );`
- sine function:
  `function sin = ( ?  dreal x; !  dreal y; );`
- tangent function:
  `function tan = ( ?  dreal x; !  dreal y; );`
- hyperbolic cosine function:
  `function cosh = ( ?  dreal x; !  dreal y; );`
- hyperbolic sine function:
  `function sinh = ( ?  dreal x; !  dreal y; );`
- hyperbolic tangent function:
  `function tanh = ( ?  dreal x; !  dreal y; );`
- exponential function:
  `function exp = ( ?  dreal x; !  dreal y; );`
- multiply floating-point number by integral power of 2:
  `function ldexp = ( ?  dreal x; integer i !  dreal y; );`
- logarithmic function:
  `function log = ( ?  dreal x; !  dreal y; );`

- base-10 logarithmic function:
  `function log10 = ( ?  dreal x; !  dreal y; );`
- power function:
  `function pow = ( ?  dreal x1; dreal x2; !  dreal y; );`
- square root function:
  `function sqrt = ( ?  dreal x; !  dreal y; );`
- smallest integral value not less than x:
  `function ceil = ( ?  dreal x; !  dreal y; );`
- absolute value of an integer:
  `function abs = ( ?  integer x; !  integer y; );`
- absolute value of floating-point number:
  `function fabs = ( ?  dreal x; !  dreal y; );`
- largest integral value not greater than x:
  `function floor = ( ?  dreal x; !  dreal y; );`
- floating-point remainder function:
  `function fmod = ( ?  dreal x1; dreal x2; !  dreal y; );`
- convert floating-point number to fractional and integral components:
  `function frexp = ( ?  dreal x; !  dreal y1; integer y2; );`
- extract signed integral and fractional values from floating-point number:
  `function modf = ( ?  dreal x; !  dreal y1; dreal y2; );`

XIII–4 Complex functions

The following complex functions are defined as intrinsic process models.

- conjugate of a complex:
  `function conj = ( ?  complex x; !  complex y; );`
  and
  `function conjd = ( ?  dcomplex x; !  dcomplex y; );`
- module of a complex:
  `function modu = ( ?  complex x; !  real y; );`
  and
  `function modud = ( ?  dcomplex x; !  dreal y; );`
- argument of a complex:
  `function arg = ( ?  complex x; !  real y; );`
  and
  `function argd = ( ?  dcomplex x; !  dreal y; );`
- real part of a complex:
  `function rpart = ( ?  complex x; !  real y; );`
  and
  `function rpartd = ( ?  dcomplex x; !  dreal y; );`

- imaginary part of a complex:
  `function ipart = ( ?  complex x; !  real y; );`
  and
  `function ipartd = ( ?  dcomplex x; !  dreal y; );`

XIII–5 Input-output functions

The following input-output functions are defined as intrinsic process models of the INRIA POLYCHRONY environment. They allow to read and write signals of basic types on standard input and output.

The `read` and `write` functions below are described with no explicit type for the input or output signal `x`: it means that they are polymorphic functions for which the effective type of the considered argument is provided by the type of the corresponding signal in the call of the function.

- `function read = ( ?  string message; !  x );`
  A `message` is displayed and a value is read for `x`.
  A standard read function is used in the generated code for the following possible types of `x`: $boolean$, $short$, $integer$, $long$, $real$, $dreal$, $complex$, $dcomplex$, $character$, $string$.

- `function write = (?  string message; x; !  );`
  A `message` is displayed and the value of `x` is written.
  A standard write function is used in the generated code for the following possible types of `x`: $boolean$, $short$, $integer$, $long$, $real$, $dreal$, $complex$, $dcomplex$, $character$, $string$.

- `function writeString = ( ?  string message; !  );`
  A `message` is displayed on the standard output.

Part F

ANNEX

Chapter XIV

Grammar of the SIGNAL language

XIV–1 Lexical units

XIV–1.1 Characters

Character ::= **character** | **CharacterCode**

Sets of characters

character ::= **name-char** | **mark** | **delimitor** | **separator** | **other-character**

name-char ::= **letter-char** | **numeral-char** | $\boxed{\_}$

letter-char ::=

upper-case-letter-char | **lower-case-letter-char** | **other-letter-char**

upper-case-letter-char ::=

$\boxed{A}$ | $\boxed{B}$ | $\boxed{C}$ | $\boxed{D}$ | $\boxed{E}$ | $\boxed{F}$ | $\boxed{G}$ | $\boxed{H}$ | $\boxed{I}$
| $\boxed{J}$ | $\boxed{K}$ | $\boxed{L}$ | $\boxed{M}$ | $\boxed{N}$ | $\boxed{O}$ | $\boxed{P}$ | $\boxed{Q}$ | $\boxed{R}$
| $\boxed{S}$ | $\boxed{T}$ | $\boxed{U}$ | $\boxed{V}$ | $\boxed{W}$ | $\boxed{X}$ | $\boxed{Y}$ | $\boxed{Z}$

lower-case-letter-char ::=

$\boxed{a}$ | $\boxed{b}$ | $\boxed{c}$ | $\boxed{d}$ | $\boxed{e}$ | $\boxed{f}$ | $\boxed{g}$ | $\boxed{h}$ | $\boxed{i}$
| $\boxed{j}$ | $\boxed{k}$ | $\boxed{l}$ | $\boxed{m}$ | $\boxed{n}$ | $\boxed{o}$ | $\boxed{p}$ | $\boxed{q}$ | $\boxed{r}$
| $\boxed{s}$ | $\boxed{t}$ | $\boxed{u}$ | $\boxed{v}$ | $\boxed{w}$ | $\boxed{x}$ | $\boxed{y}$ | $\boxed{z}$

other-letter-char ::=
À | Á | Â | Ã | Ä | Å | Æ | Ç | È
| É | Ê | Ë | Ì | Í | Î | Ï | Ð | Ñ
| Ò | Ó | Ô | Õ | Ö | Ø | Ù | Ú | Û
| Ü | Ý | Þ | ß | à | á | â | ã | ä
| å | æ | ç | è | é | ê | ë | ì | í
| î | ï | ð | ñ | ò | ó | ô | õ | ö
| ø | ù | ú | û | ü | ý | þ | ÿ

numeral-char ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

mark ::= . | ’ | " | % | : | = | < | > | +
| − | * | / | @ | $ | ^ | # | \| | \

delimitor ::= (|) | { | } | [|]
| ? | ! | , | ;

separator ::= \x20
| **long-separator**

long-separator ::= \x9
| \xA
| \xC
| \xD

Encodings of characters

CharacterCode ::= **OctalCode** | **HexadecimalCode**
| **escape-code**

OctalCode ::= \ **octal-char [octal-char [octal-char]]**

octal-char ::= $\boxed{\mathbf{0}}$ | $\boxed{\mathbf{1}}$ | $\boxed{\mathbf{2}}$ | $\boxed{\mathbf{3}}$ | $\boxed{\mathbf{4}}$ | $\boxed{\mathbf{5}}$ | $\boxed{\mathbf{6}}$ | $\boxed{\mathbf{7}}$

HexadecimalCode ::= $\boxed{\backslash\mathbf{x}}$ **hexadecimal-char [hexadecimal-char]**

hexadecimal-char ::= **numeral-char**
| $\boxed{\mathbf{A}}$ | $\boxed{\mathbf{B}}$ | $\boxed{\mathbf{C}}$ | $\boxed{\mathbf{D}}$ | $\boxed{\mathbf{E}}$ | $\boxed{\mathbf{F}}$
| $\boxed{\mathbf{a}}$ | $\boxed{\mathbf{b}}$ | $\boxed{\mathbf{c}}$ | $\boxed{\mathbf{d}}$ | $\boxed{\mathbf{e}}$ | $\boxed{\mathbf{f}}$

escape-code ::= $\boxed{\backslash\mathbf{a}}$ | $\boxed{\backslash\mathbf{b}}$ | $\boxed{\backslash\mathbf{f}}$ | $\boxed{\backslash\mathbf{n}}$ | $\boxed{\backslash\mathbf{r}}$ | $\boxed{\backslash\mathbf{t}}$
| $\boxed{\backslash\mathbf{v}}$ | $\boxed{\backslash\backslash}$ | $\boxed{\backslash\mathbf{"}}$ | $\boxed{\backslash\mathbf{'}}$ | $\boxed{\backslash\mathbf{?}}$ | $\boxed{\backslash\mathbf{\%}}$

XIV–1.2 Vocabulary

prefix-mark ::= $\boxed{\backslash}$

Names

Name ::= **begin-name-char [{ name-char }$^+$]**

begin-name-char ::= **{ name-char $\backslash$ numeral-char }**

Boolean constants

Boolean-cst ::= $\boxed{\textbf{true}}$ | $\boxed{\textbf{false}}$

Integer constants

Integer-cst ::= **{ numeral-char }$^+$**

Real constants

Real-cst ::= **Simple-precision-real-cst**
| **Double-precision-real-cst**

Simple-precision-real-cst ::=
Integer-cst Simple-precision-exponent
| **Integer-cst** $\boxed{.}$ **Integer-cst [Simple-precision-exponent]**

Double-precision-real-cst ::=
Integer-cst Double-precision-exponent
| **Integer-cst** $\boxed{.}$ **Integer-cst Double-precision-exponent**

Simple-precision-exponent ::= $\boxed{\text{e}}$ **Relative-cst** | $\boxed{\text{E}}$ **Relative-cst**

Double-precision-exponent ::= $\boxed{\text{d}}$ **Relative-cst** | $\boxed{\text{D}}$ **Relative-cst**

Relative-cst ::= **Integer-cst**
| $\boxed{+}$ **Integer-cst**
| $\boxed{-}$ **Integer-cst**

Character constants

Character-cst ::= $\boxed{\text{'}}$ **Character-cstCharacter** $\boxed{\text{'}}$

Character-cstCharacter ::= **{ Character** $\setminus$ **character-spec-char }**

character-spec-char ::= $\boxed{\text{'}}$
| **long-separator**

String constants

String-cst ::= $\boxed{\text{"}}$ **[{ String-cstCharacter }**$^{+}$ **]** $\boxed{\text{"}}$

String-cstCharacter ::= **{ Character** $\setminus$ **string-spec-char }**

string-spec-char ::= $\boxed{\text{"}}$
| **long-separator**

Comments

Comment ::= $\boxed{\textbf{\%}}$ **[{ CommentCharacter }$^+$]** $\boxed{\textbf{\%}}$

CommentCharacter ::= **{ Character $\setminus$ comment-spec-char }**

comment-spec-char ::= $\boxed{\textbf{\%}}$

XIV–2 Domains of values of the signals

SIGNAL-TYPE ::= **Scalar-type**
| **External-type**
| **ENUMERATED-TYPE**
| **ARRAY-TYPE**
| **TUPLE-TYPE**

XIV–2.1 Scalar types

Scalar-type ::= **Synchronization-type**
| **Numeric-type**
| **Alphabetic-type**

Numeric-type ::= **Integer-type**
| **Real-type**
| **Complex-type**

Alphabetic-type ::= $\boxed{\textbf{char}}$
| $\boxed{\textbf{string}}$

Synchronization types

Synchronization-type ::= $\boxed{\textbf{event}}$
| $\boxed{\textbf{boolean}}$

Integer types

Integer-type ::= $\boxed{\textbf{short}}$
| $\boxed{\textbf{integer}}$
| $\boxed{\textbf{long}}$

Real types

Real-type ::= $\boxed{\textbf{real}}$
| $\boxed{\textbf{dreal}}$

Complex types

Complex-type ::= $\boxed{\textbf{complex}}$
| $\boxed{\textbf{dcomplex}}$

XIV–2.2 External types

External-type ::= **Name-***type*

XIV–2.3 Enumerated types

ENUMERATED-TYPE ::=
  $\boxed{\textbf{enum}}\ \boxed{\textbf{(}}$ **Name-***enum-value* **{** $\boxed{\textbf{,}}$ **Name-***enum-value* **}*** $\boxed{\textbf{)}}$

ENUM-CST ::=
  $\boxed{\#}$ **Name-***enum-value*
| **Name-***type* $\boxed{\#}$ **Name-***enum-value*

XIV–2.4 Array types

ARRAY-TYPE ::=
  $\boxed{\textbf{[}}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }*** $\boxed{\textbf{]}}$ **SIGNAL-TYPE**

XIV–2.5 Tuple types

TUPLE-TYPE ::=
  $\boxed{\textbf{struct}}\ \boxed{\textbf{(}}$ **NAMED-FIELDS** $\boxed{\textbf{)}}$
| $\boxed{\textbf{bundle}}\ \boxed{\textbf{(}}$ **NAMED-FIELDS** $\boxed{\textbf{)}}$
    [SPECIFICATION-OF-PROPERTIES]

NAMED-FIELDS ::=
  { S-DECLARATION }$^{+}$

XIV–2.6 Denotation of types

SIGNAL-TYPE ::=

Name-*type*

DECLARATION-OF-TYPES ::=

$\boxed{\textbf{type}}$ **DEFINITION-OF-TYPE {** $\boxed{\textbf{,}}$ **DEFINITION-OF-TYPE }*** $\boxed{\textbf{;}}$

DEFINITION-OF-TYPE ::=

Name-*type*
| **Name-***type* $\boxed{=}$ **DESCRIPTION-OF-TYPE**

DESCRIPTION-OF-TYPE ::=

SIGNAL-TYPE
| **EXTERNAL-NOTATION [TYPE-INITIAL-VALUE]**

XIV–2.7 Declarations of constant identifiers

DECLARATION-OF-CONSTANTS ::=

$\boxed{\textbf{constant}}$ **SIGNAL-TYPE**
DEFINITION-OF-CONSTANT { $\boxed{\textbf{,}}$ **DEFINITION-OF-CONSTANT }*** $\boxed{\textbf{;}}$

DEFINITION-OF-CONSTANT ::=

Name-*constant*
| **Name-***constant* $\boxed{=}$ **DESCRIPTION-OF-CONSTANT**

DESCRIPTION-OF-CONSTANT ::=

S-EXPR
| **EXTERNAL-NOTATION**

XIV–2.8 Declarations of sequence identifiers

S-DECLARATION ::=

SIGNAL-TYPE
DEFINITION-OF-SEQUENCE { $\boxed{,}$ **DEFINITION-OF-SEQUENCE }*** $\boxed{;}$

DEFINITION-OF-SEQUENCE ::=

Name-*signal*
| **Name-***signal* $\boxed{\textbf{init}}$ **S-EXPR**

XIV–2.9 Declarations of state variables

DECLARATION-OF-STATE-VARIABLES ::=

$\boxed{\textbf{statevar}}$ **SIGNAL-TYPE**
DEFINITION-OF-SEQUENCE { $\boxed{,}$ **DEFINITION-OF-SEQUENCE }*** $\boxed{;}$

XIV–3 Expressions on signals

XIV–3.1 Systems of equations on signals

Elementary equations

ELEMENTARY-PROCESS ::=

DEFINITION-OF-SIGNALS

DEFINITION-OF-SIGNALS ::=

Name-*signal* $\boxed{:=}$ **S-EXPR**

DEFINITION-OF-SIGNALS ::=

$\boxed{(}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}*** $\boxed{)}$ $\boxed{:=}$ **S-EXPR**

DEFINITION-OF-SIGNALS ::=

Name-*signal* $\boxed{\textbf{::=}}$ **S-EXPR**
| **Name-***signal* $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

DEFINITION-OF-SIGNALS ::=

$\boxed{\textbf{(}}$ **Name-***signal* **{** $\boxed{\textbf{,}}$ **Name-***signal* **}**** $\boxed{\textbf{)}}$ $\boxed{\textbf{::=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **Name-***signal* **{** $\boxed{\textbf{,}}$ **Name-***signal* **}**** $\boxed{\textbf{)}}$ $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

Call of model

ELEMENTARY-PROCESS ::=

INSTANCE-OF-PROCESS

INSTANCE-OF-PROCESS ::=

EXPANSION
| **Name-***model* $\boxed{\textbf{(}}$ $\boxed{\textbf{)}}$

EXPANSION ::=

Name-*model*
$\boxed{\textbf{\{}}$ **S-EXPR-PARAMETER {** $\boxed{\textbf{,}}$ **S-EXPR-PARAMETER }**** $\boxed{\textbf{\}}}$

S-EXPR-PARAMETER ::=

S-EXPR
| **SIGNAL-TYPE**

INSTANCE-OF-PROCESS ::=

PRODUCTION

PRODUCTION ::=

MODEL-REFERENCE $\boxed{\textbf{(}}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }**** $\boxed{\textbf{)}}$

MODEL-REFERENCE ::=

EXPANSION
| **Name-***model*

S-EXPR ::=

  INSTANCE-OF-PROCESS

S-EXPR ::=

  CONVERSION

CONVERSION ::=

  Type-conversion $\boxed{\textbf{(}}$ **S-EXPR** $\boxed{\textbf{)}}$

Type-conversion ::=

  Scalar-type
  | **Name-***type*

Nesting of expressions on signals

S-EXPR ::=

  $\boxed{\textbf{(}}$ **S-EXPR** $\boxed{\textbf{)}}$

XIV–3.2 Elementary expressions

S-EXPR-*ELEMENTARY* ::=

  CONSTANT
  | **Name-***signal*
  | **Label**
  | **Name-***state-variable* $\boxed{\textbf{?}}$

Constant expressions

CONSTANT ::=

Boolean-cst
| **Integer-cst**
| **Real-cst**
| **Character-cst**
| **String-cst**
| **ENUM-CST**

XIV–3.3 Dynamic expressions

S-EXPR-*DYNAMIC* ::=

SIMPLE-DELAY
| **WINDOW**
| **GENERALIZED-DELAY**

Simple delay

SIMPLE-DELAY ::=

S-EXPR $\boxed{\textbf{\$}}$ **[** $\boxed{\textbf{init}}$ **S-EXPR]**

Sliding window

WINDOW ::=

S-EXPR $\boxed{\textbf{window}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

Generalized delay

GENERALIZED-DELAY ::=

S-EXPR $\boxed{\textbf{\$}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

XIV–3.4 Polychronous expressions

S-EXPR-*TEMPORAL* ::=

MERGING
| **EXTRACTION**
| **MEMORIZATION**
| **VARIABLE**
| **COUNTER**

Merging

MERGING ::=

S-EXPR $\boxed{\textbf{default}}$ **S-EXPR**

Extraction

EXTRACTION ::=

S-EXPR $\boxed{\textbf{when}}$ **S-EXPR**

Memorization

MEMORIZATION ::=

S-EXPR $\boxed{\textbf{cell}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

Variable clock signal

VARIABLE ::=

$\boxed{\textbf{var}}$ **S-EXPR [** $\boxed{\textbf{init}}$ **S-EXPR]**

Counters

COUNTER ::=

  S-EXPR $\boxed{\textbf{after}}$ **S-EXPR**
| **S-EXPR** $\boxed{\textbf{from}}$ **S-EXPR**
| **S-EXPR** $\boxed{\textbf{count}}$ **S-EXPR**

XIV–3.5 Constraints and expressions on clocks

ELEMENTARY-PROCESS ::=

  CONSTRAINT

Expressions on clock signals

S-EXPR-*CLOCK* ::=

  SIGNAL-CLOCK

SIGNAL-CLOCK ::=

  $\boxed{\hat{}\ }$ **S-EXPR**

S-EXPR-*CLOCK* ::=

  CLOCK-EXTRACTION

CLOCK-EXTRACTION ::=

  $\boxed{\textbf{when}}$ **S-EXPR**

S-EXPR-*CLOCK* ::=

  $\boxed{\hat{}0}$

Operators of clock lattice

S-EXPR-*CLOCK* ::=
    S-EXPR $\boxed{\hat{}+}$ **S-EXPR**
  | **S-EXPR** $\boxed{\hat{}-}$ **S-EXPR**
  | **S-EXPR** $\boxed{\hat{}*}$ **S-EXPR**

Relations on clocks

CONSTRAINT ::=
    S-EXPR { $\boxed{\hat{}=}$ **S-EXPR }***
  | **S-EXPR {** $\boxed{\hat{}<}$ **S-EXPR }***
  | **S-EXPR {** $\boxed{\hat{}>}$ **S-EXPR }***
  | **S-EXPR {** $\boxed{\hat{}\#}$ **S-EXPR }***

XIV–3.6 Constraints on signals

CONSTRAINT ::=
    S-EXPR $\boxed{\textbf{:=:}}$ **S-EXPR**

XIV–3.7 Boolean synchronous expressions

Expressions on Booleans

S-EXPR-*BOOLEAN* ::=
    $\boxed{\textbf{not}}$ **S-EXPR**

S-EXPR-*BOOLEAN* ::=
    S-EXPR $\boxed{\textbf{or}}$ **S-EXPR**
  | **S-EXPR** $\boxed{\textbf{and}}$ **S-EXPR**
  | **S-EXPR** $\boxed{\textbf{xor}}$ **S-EXPR**

Boolean relations

S-EXPR-*BOOLEAN* ::=
RELATION

RELATION ::=
S-EXPR $\boxed{=}$ **S-EXPR**
| **S-EXPR** $\boxed{/=}$ **S-EXPR**
| **S-EXPR** $\boxed{>}$ **S-EXPR**
| **S-EXPR** $\boxed{>=}$ **S-EXPR**
| **S-EXPR** $\boxed{<}$ **S-EXPR**
| **S-EXPR** $\boxed{<=}$ **S-EXPR**
| **S-EXPR** $\boxed{==}$ **S-EXPR**
| **S-EXPR** $\boxed{<<=}$ **S-EXPR**

XIV–3.8 Synchronous expressions on numeric signals

Binary expressions on numeric signals

S-EXPR-*ARITHMETIC* ::=
S-EXPR $\boxed{+}$ **S-EXPR**
| **S-EXPR** $\boxed{-}$ **S-EXPR**
| **S-EXPR** $\boxed{*}$ **S-EXPR**
| **S-EXPR** $\boxed{/}$ **S-EXPR**
| **S-EXPR** $\boxed{\textbf{modulo}}$ **S-EXPR**
| **S-EXPR** $\boxed{**}$ **S-EXPR**
| **DENOTATION-OF-COMPLEX**

DENOTATION-OF-COMPLEX ::=
S-EXPR $\boxed{@}$ **S-EXPR**

Unary operators

S-EXPR-*ARITHMETIC* ::=
$\boxed{+}$ **S-EXPR**
| $\boxed{-}$ **S-EXPR**

XIV–3.9 Synchronous condition

S-EXPR-*CONDITION* ::=
$\boxed{\textbf{if}}$ **S-EXPR** $\boxed{\textbf{then}}$ **S-EXPR** $\boxed{\textbf{else}}$ **S-EXPR**

XIV–4 Expressions on processes

P-EXPR ::=
ELEMENTARY-PROCESS
| **HIDING**
| **LABELLED-PROCESS**
| **GENERAL-PROCESS**

GENERAL-PROCESS ::=
COMPOSITION
| **CONFINED-PROCESS**
| **CHOICE-PROCESS**

XIV–4.1 Composition

COMPOSITION ::=
$\boxed{(|}$ **[P-EXPR {** $\boxed{|}$ **P-EXPR }*** **]** $\boxed{|)}$

XIV–4.2 Hiding

HIDING ::=

GENERAL-PROCESS $\boxed{/}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}***

| **HIDING** $\boxed{/}$ **Name-***signal* **{** $\boxed{,}$ **Name-***signal* **}***

XIV–4.3 Confining with local declarations

CONFINED-PROCESS ::=

GENERAL-PROCESS DECLARATION-BLOCK

DECLARATION-BLOCK ::=

$\boxed{\textbf{where}}$ **{ DECLARATION }+** $\boxed{\textbf{end}}$

XIV–4.4 Labelled processes

LABELLED-PROCESS ::=

Label $\boxed{::}$ **P-EXPR**

Label ::=

Name

XIV–4.5 Choice processes

CHOICE-PROCESS ::=

$\boxed{\textbf{case}}$ **Name-***signal* $\boxed{\textbf{in}}$ **{ CASE }+ [ELSE-CASE]** $\boxed{\textbf{end}}$

CASE ::=

ENUMERATION-OF-VALUES $\boxed{:}$ **GENERAL-PROCESS**

ELSE-CASE ::=

$\boxed{\textbf{else}}$ **GENERAL-PROCESS**

ENUMERATION-OF-VALUES ::=

$\boxed{\{}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{\}}$
| $\boxed{[.}$ **[S-EXPR]** $\boxed{,}$ **[S-EXPR]** $\boxed{.]}$
| $\boxed{[.}$ **[S-EXPR]** $\boxed{,}$ **[S-EXPR]** $\boxed{[.}$
| $\boxed{.]}$ **[S-EXPR]** $\boxed{,}$ **[S-EXPR]** $\boxed{.]}$
| $\boxed{.]}$ **[S-EXPR]** $\boxed{,}$ **[S-EXPR]** $\boxed{[.}$

XIV–5 Tuples of signals

S-EXPR-*TUPLE* ::=

TUPLE-ENUMERATION
| **TUPLE-FIELD**

XIV–5.1 Enumeration of tuple elements

TUPLE-ENUMERATION ::=

$\boxed{(}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{)}$

XIV–5.2 Denotation of field

TUPLE-FIELD ::=

S-EXPR $\boxed{.}$ **Name-***field*

XIV–5.3 Equation of definition of tuple component

DEFINITION-OF-SIGNALS ::=
  COMPONENT $\boxed{\textbf{:=}}$ **S-EXPR**
| **COMPONENT** $\boxed{\textbf{::=}}$ **S-EXPR**
| **COMPONENT** $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **COMPONENT** { $\boxed{\textbf{,}}$ **COMPONENT** }* $\boxed{\textbf{)}}$ $\boxed{\textbf{:=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **COMPONENT** { $\boxed{\textbf{,}}$ **COMPONENT** }* $\boxed{\textbf{)}}$ $\boxed{\textbf{::=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **COMPONENT** { $\boxed{\textbf{,}}$ **COMPONENT** }* $\boxed{\textbf{)}}$ $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

COMPONENT ::=
  Name-*signal*
| **Name-***signal* $\boxed{\textbf{.}}$ **COMPONENT**

XIV–6 Spatial processing

S-EXPR-*ARRAY* ::=
  ARRAY-ENUMERATION
| **CONCATENATION**
| **ITERATIVE-ENUMERATION**
| **INDEX**
| **ARRAY-ELEMENT**
| **SUB-ARRAY**
| **ARRAY-RESTRUCTURATION**
| **MULTI-INDEX**
| **SEQUENTIAL-DEFINITION**
| **TRANSPOSITION**
| **ARRAY-PRODUCT**
| **REFERENCE-SEQUENCE**

XIV–6.1 Enumeration

ARRAY-ENUMERATION ::=
$\boxed{[}$ **S-EXPR {** $\boxed{,}$ **S-EXPR }*** $\boxed{]}$

XIV–6.2 Concatenation

CONCATENATION ::=
S-EXPR $\boxed{|+}$ **S-EXPR**

XIV–6.3 Repetition

ITERATIVE-ENUMERATION ::=
S-EXPR $\boxed{|*}$ **S-EXPR**

XIV–6.4 Definition of index

INDEX ::=
S-EXPR $\boxed{..}$ **S-EXPR [** $\boxed{\textbf{step}}$ **S-EXPR]**

XIV–6.5 Array element

ARRAY-ELEMENT ::=
S-EXPR $\boxed{[}$ **S-EXPR** { $\boxed{,}$ **S-EXPR** }* $\boxed{]}$
| **S-EXPR** $\boxed{[}$ **S-EXPR** { $\boxed{,}$ **S-EXPR** }* $\boxed{]}$ **ARRAY-RECOVERY**

ARRAY-RECOVERY ::=
$\boxed{\backslash\backslash}$ **S-EXPR**

XIV–6.6 Extraction of sub-array

SUB-ARRAY ::=
S-EXPR $\boxed{[}$ **S-EXPR** { $\boxed{,}$ **S-EXPR** }* $\boxed{]}$

XIV–6.7 Array restructuration

ARRAY-RESTRUCTURATION ::=
S-EXPR $\boxed{:}$ **S-EXPR**

XIV–6.8 Extended syntax of equations of definition

DEFINITION-OF-SIGNALS ::=

DEFINED-ELEMENT $\boxed{\textbf{:=}}$ **S-EXPR**
| **DEFINED-ELEMENT** $\boxed{\textbf{::=}}$ **S-EXPR**
| **DEFINED-ELEMENT** $\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **DEFINED-ELEMENT {** $\boxed{\textbf{,}}$ **DEFINED-ELEMENT }*** $\boxed{\textbf{)}}$
$\boxed{\textbf{:=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **DEFINED-ELEMENT {** $\boxed{\textbf{,}}$ **DEFINED-ELEMENT }*** $\boxed{\textbf{)}}$
$\boxed{\textbf{::=}}$ **S-EXPR**
| $\boxed{\textbf{(}}$ **DEFINED-ELEMENT {** $\boxed{\textbf{,}}$ **DEFINED-ELEMENT }*** $\boxed{\textbf{)}}$
$\boxed{\textbf{::=}}$ $\boxed{\textbf{defaultvalue}}$ **S-EXPR**

DEFINED-ELEMENT ::=

COMPONENT
| **COMPONENT** $\boxed{\textbf{[}}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }*** $\boxed{\textbf{]}}$

XIV–6.9 Cartesian product

MULTI-INDEX ::=

$\boxed{<<}$ **S-EXPR {** $\boxed{\textbf{,}}$ **S-EXPR }*** $\boxed{>>}$

XIV–6.10 Iterations of processes

GENERAL-PROCESS ::=

ITERATION-OF-PROCESSES

ITERATION-OF-PROCESSES ::=

$\boxed{\textbf{array}}$ **ARRAY-INDEX** $\boxed{\textbf{of}}$ **P-EXPR [ITERATION-INIT]** $\boxed{\textbf{end}}$
$\boxed{\textbf{iterate}}$ **ITERATION-INDEX** $\boxed{\textbf{of}}$ **P-EXPR [ITERATION-INIT]** $\boxed{\textbf{end}}$

ARRAY-INDEX ::=

| **Name** $\boxed{\textbf{to}}$ **S-EXPR**

ITERATION-INDEX ::=

DEFINED-ELEMENT
| **(DEFINED-ELEMENT { , DEFINED-ELEMENT }*)**
| **S-EXPR**

ITERATION-INIT ::=

with P-EXPR

REFERENCE-SEQUENCE ::=

S-EXPR [?]

XIV–6.11 Sequential definition

SEQUENTIAL-DEFINITION ::=

S-EXPR next S-EXPR

XIV–6.12 Sequential enumeration

ITERATIVE-ENUMERATION ::=

[ITERATION { , PARTIAL-DEFINITION }*]

PARTIAL-DEFINITION ::=

DEFINITION-OF-ELEMENT
| **ITERATION**

DEFINITION-OF-ELEMENT ::=

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

ITERATION ::=

{ PARTIAL-ITERATION { , PARTIAL-ITERATION }*
: DEFINITION-OF-ELEMENT
| **{ PARTIAL-ITERATION { , PARTIAL-ITERATION }***
: S-EXPR

PARTIAL-ITERATION ::=
[Name] [$\boxed{\textbf{in}}$ S-EXPR] [$\boxed{\textbf{to}}$ S-EXPR] [$\boxed{\textbf{step}}$ S-EXPR]

XIV–6.13 Operators on matrices

Transposition

TRANSPOSITION ::=
$\boxed{\textbf{tr}}$ **S-EXPR**

Matrix products

ARRAY-PRODUCT ::=
S-EXPR $\boxed{*.}$ **S-EXPR**

XIV–7 Models of processes

XIV–7.1 Classes of process models

MODEL ::=
PROCESS
| **ACTION**
| **NODE**
| **FUNCTION**

PROCESS ::=
$\boxed{\textbf{process}}$ **Name-***model* $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

ACTION ::=
$\boxed{\textbf{action}}$ **Name-***model* $\boxed{=}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

NODE ::=

$\boxed{\textbf{node}}$ **Name-***model* $\boxed{\textbf{=}}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{\textbf{;}}$

FUNCTION ::=

$\boxed{\textbf{function}}$ **Name-***model* $\boxed{\textbf{=}}$
DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{\textbf{;}}$

BODY ::=

DESCRIPTION-OF-MODEL

DESCRIPTION-OF-MODEL ::=

GENERAL-PROCESS
| **EXTERNAL-NOTATION**

XIV–7.2 Local declarations of a process model

DECLARATION ::=

S-DECLARATION
| **DECLARATION-OF-STATE-VARIABLES**
| **DECLARATION-OF-CONSTANTS**
| **DECLARATION-OF-TYPES**
| **DECLARATION-OF-LABELS**
| **REFERENCES**
| **MODEL**

XIV–7.3 Declarations of labels

DECLARATION-OF-LABELS ::=

$\boxed{\textbf{label}}$ **Name-***label* **{** $\boxed{\textbf{,}}$ **Name-***label* **}*** $\boxed{\textbf{;}}$

XIV–7.4 References to signals with extended visibility

REFERENCES ::=
$\boxed{\textbf{ref}}$ **Name-***signal* **{** $\boxed{\textbf{,}}$ **Name-***signal* **}*** $\boxed{\textbf{;}}$

XIV–7.5 Interface of a model

DEFINITION-OF-INTERFACE ::=
INTERFACE

INTERFACE ::=
[PARAMETERS] $\boxed{\textbf{(}}$ **INPUTS OUTPUTS** $\boxed{\textbf{)}}$ **EXTERNAL-GRAPH**

PARAMETERS ::=
$\boxed{\textbf{\{}}$ **[{ FORMAL-PARAMETER }**$^+$ **]** $\boxed{\textbf{\}}}$

FORMAL-PARAMETER ::=
S-DECLARATION
| **DECLARATION-OF-TYPES**

INPUTS ::=
$\boxed{\textbf{?}}$ **[{ S-DECLARATION }**$^+$ **]**

OUTPUTS ::=
$\boxed{\textbf{!}}$ **[{ S-DECLARATION }**$^+$ **]**

XIV–7.6 Graph of a model

EXTERNAL-GRAPH ::=
[PROCESS-ATTRIBUTE] [SPECIFICATION-OF-PROPERTIES]

PROCESS-ATTRIBUTE ::=

safe
| **deterministic**
| **unsafe**

SPECIFICATION-OF-PROPERTIES ::=

spec **GENERAL-PROCESS**

Dependences

ELEMENTARY-PROCESS ::=

DEPENDENCES

DEPENDENCES ::=

SIGNALS { --> SIGNALS }*
| **{ SIGNALS --> SIGNALS } when S-EXPR**

SIGNALS ::=

ELEMENTARY-SIGNAL
| **{ ELEMENTARY-SIGNAL { , ELEMENTARY-SIGNAL }* }**

ELEMENTARY-SIGNAL ::=

DEFINED-ELEMENT
| **Label**

XIV–7.7 Directives

DIRECTIVES ::=

pragmas { PRAGMA }+ end pragmas

PRAGMA ::=

Name-*pragma* **[{ PRAGMA-OBJECT { , PRAGMA-OBJECT }* }]**
[Pragma-statement]

PRAGMA-OBJECT ::=

Label
| **Name**

Pragma-statement ::=
  String-cst

XIV–7.8 Models as types and parameters

DEFINITION-OF-TYPE ::=
  $\boxed{\textbf{process}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**
| $\boxed{\textbf{action}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**
| $\boxed{\textbf{node}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**
| $\boxed{\textbf{function}}$ **Name-***model-type* $\boxed{=}$ **DEFINITION-OF-INTERFACE**

DEFINITION-OF-INTERFACE ::=
  Name-*model-type*

FORMAL-PARAMETER ::=
  FORMAL-MODEL

FORMAL-MODEL ::=
  $\boxed{\textbf{process}}$ **Name-***model-type* **Name-***model*
| $\boxed{\textbf{action}}$ **Name-***model-type* **Name-***model*
| $\boxed{\textbf{node}}$ **Name-***model-type* **Name-***model*
| $\boxed{\textbf{function}}$ **Name-***model-type* **Name-***model*

S-EXPR-PARAMETER ::=
  Name-*model*

XIV–8 Modules

XIV–8.1 Declaration and use of modules

MODULE ::=

$\boxed{\textbf{module}}$ **Name-***module* $\boxed{=}$

[DIRECTIVES] { DECLARATION }$^{+}$ $\boxed{\textbf{end}}$ $\boxed{;}$

DECLARATION-OF-CONSTANTS ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{constant}}$ **SIGNAL-TYPE**

DEFINITION-OF-CONSTANT { $\boxed{,}$ **DEFINITION-OF-CONSTANT }*** $\boxed{;}$

DECLARATION-OF-TYPES ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{type}}$

DEFINITION-OF-TYPE { $\boxed{,}$ **DEFINITION-OF-TYPE }*** $\boxed{;}$

PROCESS ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{process}}$ **Name-***model* $\boxed{=}$

DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

ACTION ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{action}}$ **Name-***model* $\boxed{=}$

DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

NODE ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{node}}$ **Name-***model* $\boxed{=}$

DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

FUNCTION ::=

$\boxed{\textbf{private}}$ $\boxed{\textbf{function}}$ **Name-***model* $\boxed{=}$

DEFINITION-OF-INTERFACE [DIRECTIVES] [BODY] $\boxed{;}$

EXTERNAL-NOTATION ::=

$\boxed{\textbf{external}}$ **[String-cst]**

DECLARATION ::=

IMPORT-OF-MODULES

IMPORT-OF-MODULES ::=

$\boxed{\textbf{use}}$ **IMPORTED-OBJECTS {** $\boxed{\textbf{,}}$ **IMPORTED-OBJECTS }*** $\boxed{\textbf{;}}$

IMPORTED-OBJECTS ::=

Name-*module*

List of figures

List of tables

Index