Collaboration "CAO-Automatique" (C2A)
PRC "Communication, Coopération, Concurrence" (C^3)
GRECO Automatique
Cercle pour les Projets Innovants en Informatique (CP2I)

SYNCHRON PROJECT

Common Formats of the Synchronous Languages

Version 0 (english)

March 17, 1994

Collaboration "CAO-Automatique" (C2A)
PRC "Communication, Coopération, Concurrence" (C^3)
GRECO Automatique
Cercle pour les Projets Innovants en Informatique (CP2I)

PROJECT SYNCHRONE

Common Formats for the Synchronous Languages

Version 0 (english)

This document was written by the authors shown below. Each is cited with their sponsoring organization.

Jean-Pierre PARIS	CISI Ingénierie / Agence Provence-Est Les Cardoulines - Bât. B1 – Route des Dolines Sophia Antipolis – 06560 Valbonne
Gérard BERRY, Frédéric MIGNARD	École des Mines de Paris / C.M.A. - INRIA B.P. 207 – 06904 Sophia-Antipolis Cedex
Philippe COURONNÉ	ILOG S.A. – 2, Av. Galiéni – 94253 Gentilly
Paul CASPI, Nicholas HALBWACHS	IMAG – B.P. 53 – 38041 Grenoble Cedex
Yves SOREL	INRIA – Domaine de Voluceau Rocquencourt – B.P. 105 – 78150 Le Chesnay
Albert BENVENISTE, Thierry GAUTIER, Paul LE GUERNIC	IRISA/INRIA – Campus de Beaulieu 35042 Rennes Cedex
François DUPONT	TNI – ZI du Vernis – 29608 Brest
Claude LE MAIRE	VERILOG – Centre d'Études Rhône-Alpes ZAC du Pré Millet – 38330 Montbonnot

This document on the common formats of the synchronous languages has been described by the authors with the support of research conducted at CMA, IMAG and at IRISA. The authors thank the participants for their various contributions.

The editing duties in the first phase were undertaken by Nicholas HALBWACHS, with Thierry GAUTIER with Paul LE GUERNIC assuming them in the final phase.

SYNCHRON PROJECT
The common formats of synchronous languages

Abstract

This report defines the syntax and semantics of the three common formats for the synchronous languages. These formats will support the representation of most of the compiling process for these languages. There are three formats and this report defines them and the translators between them:

- IC is a parallel format for *imperative* style;
- GC is a parallel format for *data-flow* style;
- OC is a sequential format used to describe automata.

Warning : This document defines the first version, called Version 0, of the common formats for synchronous languages. Some changes are forseen for future versions : they will result, in particular, from experiments with the formats. Other changes are already planned : ones concerning for example inter-format communication, more powerful data structures and the like.

Contents

Chapter 1

General Organization

1.1 Introduction

In the future, many objects in everyday life will be controlled by computers : factories, airplanes, trains, cars, televisions . . .

Using signals received from the outside world they must react in a constrained amount of time. Traditionally the applications area called "real-time" uses analog systems or relays. Today computers and intelligent electronics are installed in all sorts of products. Now, its service is the only security : one single error, in software or hardware, at design time or at fabrication time, can have dramatic consequences. The reliability of embedded software is critical : it requires an exact solution to the problem of computing and rigor in the process of writing the programs. These two aspects crucially depend on the manner of the formal expression used for the specifications, the programming language supporting this formalism and the supporting services. The developer of real-time applications must be offered a powerful language with a rigorous underlying formalism and provided with an environment (compilers, verifiers, graphical editors) of high quality : that's the goal of the synchronous programming languages Esterel, Lustre, Signal and Argos. The synchronous character [1] of these languages allows for guarantees of good behavior in the areas of design, realization and maintenance.

The application domains of synchronononous languages and their tools and associated environments cover the gamut of real-time applications, from integrated micro-applications to large systems :

- *hardware :* integrated circuites for complact real-time applications (fast computing, state machines, micro-controllers) ; the need for such is fre-

[1] the synchronous approach rests on a vision of time which is "logical" and which is global to the whole system; in contrast, the asynchronous permits formal reasoning on aspects such as synchronization, logic and ordering of the computation.

quent (robots, automobiles, interfaces, signal processing. . .) ; synchronous languages can also interface between different circuit CAD tools ;

- *multi-, micro- or heterogeneous architectures :* these are found in military applications, industrial process control, real-time information processing. . . ; synchronous languages allow one to have a global view of the application (by virtue of ease of specification) and the production of code for these architectures ;

- *complex applications* or "systems" which is a domain where the French companies are well-placed among the industrial firms undertaking large projects (THOMSON, ALSTHOM, MATRA, MERLIN-GERIN. . .) ; synchronous languages form the heart of an uninterrupted chain: *specification → implementation*.

To address the grand diversity of professions through a unique concept necessitates the development of tools and environments adapted to each particular culture. Synchronous programming ought then to cease to be a juxtaposition of tools and to become a *concept* permitting many ways of speaking about the same coherent object. This justifies the goals of the SYNCHRONE project : *the definition and realization by multiple examples in a common base*, whose general specification is the object of this document. This base is the ideal candidate for the integration in a true workshop for the development of real-time applications, constituting an essential piece of it. It will permit the evolution of *synchronous* languages in many ways simultaneously and concurrently, with all the languages leaning on a common interchange format : synchronous languages will become mutually reinforcing. The long-awaited result of this project is depicted in Figure 1.1 of Section 1.2.

1.2 Synchronous Formats

The four languages ESTEREL, LUSTRE, SIGNAL et ARGOS use different styles : ESTEREL and ARGOS are imperative, LUSTRE is declarative and functional while SIGNAL is a declarative and relational language. This is a richness, more from the point of view of theoretical exploration of the possibilities of the synchronous approach than the cultural diversity of potential users. This diversity is also a handicap for the dissemenation of the concept. The common base for synchronous programming allows the filling in of this last handicap, all the while preserving the richness and the diversity.

The common base of synchronous programming (cf. Figure 1.1) consists of a set of formats to which various tools are applicable. The necessary compromise between the facility for translation between source languages through the formats

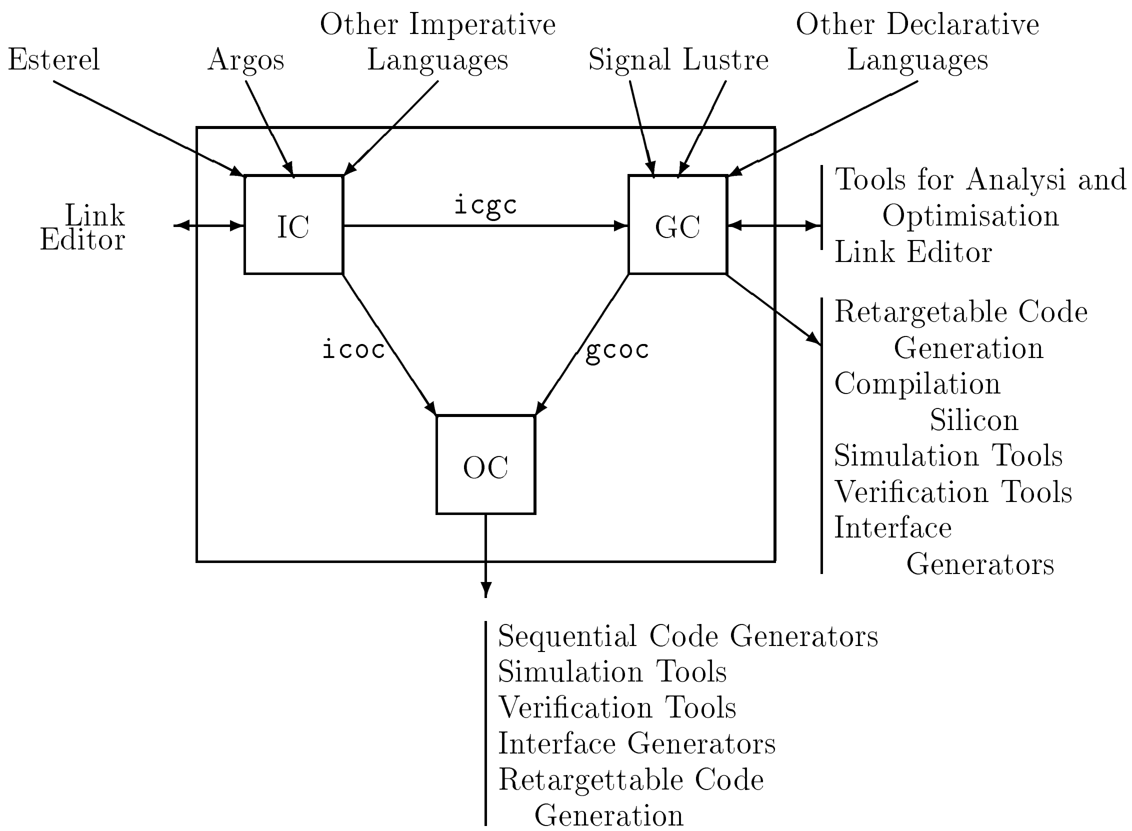


Figure 1.1: The Common Base for Synchronous Programming

and the appropriateness of the formats for effective treatment by the relevant tools distinguishes two levels :

- one level where the parallelism present in the program source is preserved
- one level of description which is purely sequential;

and in the first level, two descriptive styles : imperative and declarative.

The base is constituted thus of three formats between which there exists translation "bridges :"

IC : is an *imperative* synchronous intermediate format : it is a parallel format which allows exceptions, sequencing, interrupts, and concurrent execution of actions. This format was derived from the actual intermediate code of the Esterel compiler which also as used as the target of the Argos compiler. Besides Esterel and Argos, this format can serve as the intermediate format between any two imperative formalisms.

GC : this is a *graph*-based synchronous intermediate format : it is a parallel format written with graphical symbols of schematic block-diagrams with links between operations, these links being labeled by the clocks that activate them. It is a hierarchical format (a box can be a predefined function or contain a complete schematic). Besides SIGNAL and LUSTRE, the format can be the target between any declarative formalism. It is, moreover, a natural candidate for separate compilation, for performance evaluation and placement and routing on a distributed architecture.

OC : this is a sequential "automaton" format which is the result of work between the languages ESTEREL and LUSTRE. This format is a interesting tool for the production of sequential (automatons in languages such as C, ADA, FORTRAN, ...) and is the favored path for processing and using automatons (simulations, animations or proofs).

1.3 Organisation of the Document

The common formats are internal formats designed to be produced and used by computer programs. Unlike languages, their syntax is thus designed more in light of automatic analysis.

Although designed to be written by very different sources, the three formats have been defined so as to share the best notation and basic ideas. One essential part common to the three formats allows the description of the low-level objects of system specification in the form of tables. These imported objects are described in other langauges called *host languages* : a host language must be a language filling in between synchronous language compilers and any other language which can be interfaced to the code produced by it.

Chapter 2 decribes the general conventions and common tables of the three formats. Chapter 3 describes the common parts of the two imperative intermediate formats. The following chapters. 4, 5, and 6 then describe the elements specific to each of the formats.

The Form and Presentation of the Syntax

There are three classes of terms in the description of the syntax of the formats :

- vocabulary at the lexical levels : **terminals** describe an indivisible succession of characters .
- lexical structures : **Terminals** at the syntactic level are defined, at a lexical level, by rules in a grammar whose vocabulary is a union of **terminal** sets ; each implicit character (for example separators) are not given directly in the rules ;

- syntactic structures : **NON-TERMINALS** are defined, at the syntactic level, by rules in the grammar whose vocabulary is formed by sets of **Terminals**; between two **Terminals**, one can insert a number of separators.

Each format unit is introduced then described with the aid of a grammar. The grammar gives the context-free syntax of the structure under consideration in one of the following forms :

♠ **STRUCTURE** ::=

- • DERIVATION1 •
- • DERIVATION2 •
- • ... •

♠ **Terminal** ::=

- • DERIVATION1 •
- • DERIVATION2 •
- • ... •

♠ **terminal** ::=

- • SET1 •
- • SET2 •
- • ... •

DERIVATION1, DERIVATION2 are rewritings of the variable **STRUCTURE** (respectively the variable **Terminal**).

SET1, SET2 are rewritings of the variable **terminal**; they are **Derivations** reduced to a single element (cf below).

Each DERIVATION is a succession of elements each of which can be :

- a **set** of characters, written in this type (unique to the lexical level).
- a $\boxed{\textbf{keyword}}$ (formed from letters), or a terminal $\boxed{\textbf{symbol}}$ (formed of other admissible characters) in that type.
- a **Terminal**, in its type.
- a syntactic **STRUCTURE** in its type (unique to the syntactic level).
- the succession of *elements* with or without separators respectively in the one of the following forms :
  - { *element* $\boxed{\textbf{symbol}}$ ... }
  - { *element* ... }
- an optional *element*, written [*element*],

- a set difference, written { *element1* $\ominus$ *element2* }, allowing one to describe text of *element1* which does not contain any text of *element2*.

The syntactic or lexical structures can appear either in singular or the plural according to their context. They can be completed with *conceputal information* given in the type used. Finally, many descriptions can be placed on the same line; in this case the terminator for each will be indicated implicitly by the beginning of the following one.

Chapter 2

Common Elements

This chapter describes the common areas of the three intermediate formats for synchronous languages. Thus, the four following sections present respectively :

- the lexical units,
- the general organization of the formats, in packages, entities and tables,
- the pragmas,
- finally, the common organization of the data in the formats.

2.1 Lexical Units

This section presents the characters used to form lexical units then the commentary, numbers, strings, identifiers and keywords and predefined identifiers. Indices which are equally lexical units are also presented in this section.

2.1.1 Lexical Conventions

A text written in one common formats of the synchronous languages is a succession of elements in the **Terminal** vocabulary (cf. Section 1.3, page 4) of the formats, between which there can appear some number of **separator**s (none in the limit). A **Terminal** in the common formats is the longest string of contiguous **terminals** and a **terminal** is the longest string of continguous characters that can be formed by left-to-right anlaysis with respect to the rules of that section. A terminal can consist of punctuation; the following symbols are not in the set of **characters** :

♠ **prefix-mark** ::= • $\boxed{\backslash}$ *start of* **CharacterCode** •
• $\boxed{\textbf{\$I}}$ *start of imperative predefined* **object** *identifiers* •
• $\boxed{\textbf{\$g}}$ *start of predefined* GC *identifiers* •
• $\boxed{\texttt{--}}$ *start of single-line comment* •

Characters used in the formats are described in the present section (the set **Character**). They can be given in a coded form which is not usable in comments, string constants or pragmas.

♠ **Character** ::= • **character** • **CharacterCode** •

Characters

The set of characters (written **character**) used in the formats contains the following subsets :

♠ **character** ::= • **char-ident** • **mark** • **delimiter** • **separator** • **other-character** •

1. The set **char-ident** of characters is used to build identifiers :

   ♠ **char-ident** ::= • **char-letter** • **char-digit** • $\boxed{\_}$ •

   ♠ **char-lettre** ::= • **char-uppercase-letter** • **car-lowercase-letter** •

   ♠ **char-uppercase-lettre** ::=
   • $\boxed{\textbf{A}}$ • $\boxed{\textbf{B}}$ • $\boxed{\textbf{C}}$ • $\boxed{\textbf{D}}$ • $\boxed{\textbf{E}}$ • $\boxed{\textbf{F}}$ • $\boxed{\textbf{G}}$ • $\boxed{\textbf{H}}$ • $\boxed{\textbf{I}}$ •
   • $\boxed{\textbf{J}}$ • $\boxed{\textbf{K}}$ • $\boxed{\textbf{L}}$ • $\boxed{\textbf{M}}$ • $\boxed{\textbf{N}}$ • $\boxed{\textbf{O}}$ • $\boxed{\textbf{P}}$ • $\boxed{\textbf{Q}}$ • $\boxed{\textbf{R}}$ •
   • $\boxed{\textbf{S}}$ • $\boxed{\textbf{T}}$ • $\boxed{\textbf{U}}$ • $\boxed{\textbf{V}}$ • $\boxed{\textbf{W}}$ • $\boxed{\textbf{X}}$ • $\boxed{\textbf{Y}}$ • $\boxed{\textbf{Z}}$ •

   ♠ **char-lowercase-letter** ::=
   • $\boxed{\text{a}}$ • $\boxed{\text{b}}$ • $\boxed{\text{c}}$ • $\boxed{\text{d}}$ • $\boxed{\text{e}}$ • $\boxed{\text{f}}$ • $\boxed{\text{g}}$ • $\boxed{\text{h}}$ • $\boxed{\text{i}}$ •
   • $\boxed{\text{j}}$ • $\boxed{\text{k}}$ • $\boxed{\text{l}}$ • $\boxed{\text{m}}$ • $\boxed{\text{n}}$ • $\boxed{\text{o}}$ • $\boxed{\text{p}}$ • $\boxed{\text{q}}$ • $\boxed{\text{r}}$ •
   • $\boxed{\text{s}}$ • $\boxed{\text{t}}$ • $\boxed{\text{u}}$ • $\boxed{\text{v}}$ • $\boxed{\text{w}}$ • $\boxed{\text{x}}$ • $\boxed{\text{y}}$ • $\boxed{\text{z}}$ •

   ♠ **char-digit** ::= • $\boxed{0}$ • $\boxed{1}$ • $\boxed{2}$ • $\boxed{3}$ • $\boxed{4}$ • $\boxed{5}$ • $\boxed{6}$ • $\boxed{7}$ • $\boxed{8}$ • $\boxed{9}$ •

   Uppercase and lower case are distinct.

2. The set **mark** of punctuation characters as lexical units are :

♠ **marque** ::= • `$` *start of common predefined* **object** *identifiers* •
• `:` *end of keywords* •
• `.` *separator character in floating-point constants and indices* •
• `+` *positive exponent in a floating-point constant* •
• `−` *negative exponent in a floating-point constant* •
• `#` *start of a literal constant* •
• `@` *start of an index to a constant* •
• `!` *start of a test for the presence of a signal* •
• `?` *start of an access to the value of a signal* •
• `"` *start and end of a string* •
• `%` *start and end of a pragma* •

3. Delimiters are terminals at the syntactic level constructed with characters other than letters or digits :

♠ **delimiter** ::= • **icgcocsymb** • **icocsymb** •

♠ **icgcocsymb** ::= • `(` • `)` • *parentheses*
• `,` • *field separator*
• `/` • *substitution*
• `:` • *function result indicator and number suffix*
• `;` • *end of structure*

♠ **icocsymb** ::= • `<` • `>` • *argument grouping*
• `[` • `]` • *argument grouping*
• `{` • `}` • *argument grouping*

4. The set of **separator** characters is given here by their hexidecimal ASCII code :

♠ **separator** ::= • `\x9` *horizontal tab* •
• `\xA` *new line* •
• `\xC` *new page* •
• `\xD` *carriage return* •
• `\x20` *space* •

5. Other printable characters are allowed in comments, pragmas and constant denotations. This subset, **other-character** is not defined by the format.

Coded Characters

All the characters (printable or non-printable) can be given in a coded form (**CharacterCode**) in comments, string constants and pragmas. The authorized codes are the ones in the ANSI C language to which has been added escape character $\boxed{\backslash\%}$ used in pragmas. A character in coded form must be a special character (**escape-code**), or an octal character code (**OctalCode**), or a hexidecimal character code (**HexidecimalCode**).

♠ **CharacterCode** ::= • **OctalCode** • **HexiadecimalCode** •
• **escape-code** •

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

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

♠ **HexidecimalCode** ::= • $\boxed{\backslash x}$ **char-hexidecimal** [**char-hexidecimal**] •

♠ **char-hexidecimal** ::= • **char-digit** •
• $\boxed{A}$ • $\boxed{B}$ • $\boxed{C}$ • $\boxed{D}$ • $\boxed{E}$ • $\boxed{F}$ •
• $\boxed{a}$ • $\boxed{b}$ • $\boxed{c}$ • $\boxed{d}$ • $\boxed{e}$ • $\boxed{f}$ •

♠ **escape-code** ::= • $\boxed{\backslash a}$ *beep* •
• $\boxed{\backslash b}$ *backspace* •
• $\boxed{\backslash f}$ *new page* •
• $\boxed{\backslash n}$ *new line* •
• $\boxed{\backslash r}$ *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* •

2.1.2 Comments

All fragments of text between a $\boxed{\texttt{--}}$ and the end of the line are a single-line comment.

On the other hand, one can equivalently put comments in the form of a pragma (cf. Section 2.3, page 21).

2.1.3 Numbers

Two types of numbers are found in the common formats : positive natural numbers and positive floating point (single precision or dobule precision). The negative numbers are obtained by the application of the `$minus` to the type under consideratino (cf. Section ??, page ??).

Integers

An integer is a lexical unit exclusively composed of a series of **char-digit**s interpreted in base 10.

An integer value is given by a **Integer-const**.

♠ **Integer-const** ::= • { **char-digit** ... } •

Reals

A single precision floating point number is a lexical unit composed of a succession of :

- an integer ;
- a decimal part consisting of by a period ($\boxed{.}$) followed by an integer ;
- an exponent, consisting of the character $\boxed{\mathbf{E}}$ or $\boxed{\mathbf{e}}$, followed by a sign (with optional $\boxed{+}$ or $\boxed{-}$) and an integer.

A double precision floating point number is a lexical unit whose definition is identical to the single precision case with the exception of the exponent. The difference is that the exponent is given by the character $\boxed{\mathbf{D}}$ or $\boxed{\mathbf{d}}$ in place of $\boxed{\mathbf{E}}$ or $\boxed{\mathbf{e}}$.

) A **Real-const** denotes a value approximating a real number.

♠ **Real-const** ::= • **Real-const-single-precision** •
• **Real-const-double-precision** •

♠ **Real-const-single-precision** ::= • **Fraction-part Single-precision-exponent** •

♠ **Real-const-double-precision** ::= • **Fraction-part Double-precision-exponent** •

♠ **Fraction-part** ::= • **Integer-const** $\boxed{.}$ **Integer-const** •

♠ **Single-precision-exponent** ::= • $\boxed{\text{e}}$ **Relative-const** • $\boxed{\text{E}}$ **Relative-const** •

♠ **Double-precision-exponent** ::= • $\boxed{\text{d}}$ **Relative-const** • $\boxed{\text{D}}$ **Relative-const** •

♠ **Relative-const** ::= • **Integer-const** • $\boxed{+}$ **Integer-const** •
• $\boxed{-}$ **Integer-const** •

2.1.4 Character Strings

Character strings are strings of characters as defined by the ANSI standard for the C language.

A string value is given by a **String-const**.

♠ **String-const** ::= • $\boxed{\text{"}}$ [{ **CharacterString** ... }] $\boxed{\text{"}}$ •

♠ **CharacterString** ::= • { **Character** ⊖ **char-spec-string** } •

♠ **char-spec-string** ::= • $\boxed{\text{"}}$ • $\boxed{\backslash}$ •

2.1.5 Identifiers

Identifiers name objects. Among all identifiers, there are distinguished **object** identifiers that are predefined.

Identifiers are lexical units composed of characters taken from among the set formed from **char-letter**s, the character $\boxed{\_}$ and the **char-digit**s ; an identifier can not start with a **char-digit**. The identifiers correspond, for example, to programmer-assigned names in the source code.

Predefined **object** identifiers are lexical unit starting with the character $\boxed{\$}$ and followed by an identifier as described below.

♠ **Identifier** ::= • **char-start-ident** [{ **char-ident** ... }] •
• $\boxed{\$}$ **char-start-ident** [{ **char-ident** ... }] •

♠ **char-start-ident** ::= • { **char-ident** ⊖ **char-digit** } •

One can, in this document, define an **Identifier** naming a particular category of **object** such as an *X* with the name *X*-**identifier**.

2.1.6 Keywords and Predefined Identifiers

Keywords

Keywords are lexical units starting with a **char-letter**, followed by **char-letter**s, **char-digit**s or $\boxed{\_}$ ending with a $\boxed{:}$ character.

Keywords used in the formats are the following :

Common to IC, GC and OC

♠ **icgcockw** ::=

• $\boxed{\textbf{call:}}$ • $\boxed{\textbf{computed:}}$ • $\boxed{\textbf{constants:}}$ • $\boxed{\textbf{data:}}$ • $\boxed{\textbf{dc:}}$ •
• $\boxed{\textbf{end:}}$ • $\boxed{\textbf{enddata:}}$ • $\boxed{\textbf{endpackage:}}$ • $\boxed{\textbf{flat:}}$ • $\boxed{\textbf{functions:}}$ •
• $\boxed{\textbf{gc:}}$ • $\boxed{\textbf{i:}}$ • $\boxed{\textbf{ic:}}$ • $\boxed{\textbf{io:}}$ • $\boxed{\textbf{import:}}$ • $\boxed{\textbf{memorysafe:}}$ •
• $\boxed{\textbf{o:}}$ • $\boxed{\textbf{oc:}}$ • $\boxed{\textbf{package:}}$ • $\boxed{\textbf{pragmas:}}$ • $\boxed{\textbf{procedures:}}$ •
• $\boxed{\textbf{safe:}}$ • $\boxed{\textbf{types:}}$ • $\boxed{\textbf{unsafe:}}$ • $\boxed{\textbf{value:}}$ •

Common to IC and OC

♠ **icockw** ::=

• $\boxed{\textbf{act:}}$ • $\boxed{\textbf{actions:}}$ • $\boxed{\textbf{bool:}}$ • $\boxed{\textbf{combine:}}$ •
• $\boxed{\textbf{endmodule:}}$ • $\boxed{\textbf{execs:}}$ • $\boxed{\textbf{if:}}$ • $\boxed{\textbf{input:}}$ • $\boxed{\textbf{inputoutput:}}$ •
$\boxed{\textbf{instances:}}$ •
• $\boxed{\textbf{kill:}}$ • $\boxed{\textbf{local:}}$ • $\boxed{\textbf{module:}}$ • $\boxed{\textbf{multiple:}}$ •
• $\boxed{\textbf{output:}}$ • $\boxed{\textbf{relations:}}$ • $\boxed{\textbf{reset:}}$ • $\boxed{\textbf{resume:}}$ • $\boxed{\textbf{return:}}$ • $\boxed{\textbf{root:}}$
•
• $\boxed{\textbf{signals:}}$ • $\boxed{\textbf{single:}}$ • $\boxed{\textbf{start:}}$ • $\boxed{\textbf{startpoint:}}$ • $\boxed{\textbf{suspend:}}$ •
• $\boxed{\textbf{tasks:}}$ • $\boxed{\textbf{variables:}}$ •

Specific to GC

♠ **gckw** ::=

• $\boxed{\textbf{assertions:}}$ • $\boxed{\textbf{define:}}$ • $\boxed{\textbf{definitions:}}$ • $\boxed{\textbf{dependences:}}$ • $\boxed{\textbf{do:}}$ •
• $\boxed{\textbf{endinterface:}}$ • $\boxed{\textbf{endnode:}}$ • $\boxed{\textbf{extern:}}$ • $\boxed{\textbf{flows:}}$ •
• $\boxed{\textbf{interface:}}$ • $\boxed{\textbf{node:}}$ • $\boxed{\textbf{set:}}$ • $\boxed{\textbf{synchronizations:}}$ •

Specific to IC

♠ **ickw** ::=

• $\boxed{\textbf{Access:}}$ • $\boxed{\textbf{Action:}}$ • $\boxed{\textbf{Emit:}}$ • $\boxed{\textbf{Exit:}}$ • $\boxed{\textbf{Fork:}}$ •
• $\boxed{\textbf{Goloop:}}$ • $\boxed{\textbf{Goto:}}$ • $\boxed{\textbf{Halt:}}$ • $\boxed{\textbf{Parallel:}}$ •
• $\boxed{\textbf{Reset:}}$ • $\boxed{\textbf{Return:}}$ • $\boxed{\textbf{Run:}}$ • $\boxed{\textbf{Stay:}}$ • $\boxed{\textbf{Test:}}$ • $\boxed{\textbf{Watchdog:}}$
•
• $\boxed{\textbf{exitlevels:}}$ • $\boxed{\textbf{goloops:}}$ • $\boxed{\textbf{instructions:}}$ • $\boxed{\textbf{linked:}}$ •

Specific to OC

♠ **ockw** ::=

• $\boxed{\textbf{calls:}}$ • $\boxed{\textbf{dags:}}$ • $\boxed{\textbf{sink:}}$ • $\boxed{\textbf{states:}}$ •

Pragma Names

♠ **pragmaskw** ::=

• $\boxed{\textbf{alias:}}$ • $\boxed{\textbf{awaited:}}$ • $\boxed{\textbf{comment:}}$ • $\boxed{\textbf{count:}}$ • $\boxed{\textbf{dir:}}$ •
• $\boxed{\textbf{emitted:}}$ • $\boxed{\textbf{file:}}$ • $\boxed{\textbf{haltset:}}$ • $\boxed{\textbf{instance:}}$ • $\boxed{\textbf{killed:}}$
•
• $\boxed{\textbf{lc:}}$ • $\boxed{\textbf{main:}}$ • $\boxed{\textbf{name:}}$ • $\boxed{\textbf{previous:}}$ • $\boxed{\textbf{register:}}$
•
• $\boxed{\textbf{sigbool:}}$ • $\boxed{\textbf{sigval:}}$ • $\boxed{\textbf{started:}}$ • $\boxed{\textbf{wire:}}$ •

Predefined Identifiers

The **predefined-identifiers** used in the formats are the following :
Common to IC, GC et OC

♠ **icgcocui** ::=

• `$and` • `$boolean` • `$cond` • `$conv` • `$div` • `$double` •
• `$eq` • `$false` • `$float` • `$ge` • `$gt` • `$integer` •
• `$le` • `$lt` • `$minus` • `$mod` •
• `$ne` • `$not` • `$or` • `$plus` • `$pure` •
• `$string` • `$times` • `$top` • `$true` • `$uminus` •

Common to IC and OC

♠ **icocui** ::=

• `$assign` • `$dsz` • `$left_and` • `$left_or` •

Specific to GC

♠ **gcui** ::=

• `$any` • `$base` •
• `$clkadd` • `$clkdiff` • `$clkeq` • `$clkimplies` • `$clkmult` • `$clkmutex` •
• `$clock` • `$default` • `$fby` • `$pre` • `$present` •
• `$select` • `$tt` • `$when` • `$win` • `$window` •

2.2 Structure Common to the Formats

Common formats allow the description of *entities*. These entities are organized in *packages*, anything outside of which must have a documented function.

A package is thus a synchronous domain, a hybrid between IC, GC, and OC, consisting of a set of numbered *entities*. The entities contained in a package can be data tables, IC modules, OC modules, or GC nodes or interfaces.

An entity is formed by *tables* which define the objects of the formats

2.2.1 Packets

A package contains, in order, the following structures :

1. the keyword **package:** ;
2. the number of entities defined in the package ;

3. the identifier naming the package;

4. the list of entities. Each entity is preceded by a number (followed by the **:** character) which names it within the package. Entities forming the package are listed one after another in numerical order. This numbering, without exception, starts at `0`. The order of the entities in a package is such that in the absence of cross references, there are no forward references (i.e., to an object in an entity with a greater number) for format objects other than pragmas;

5. the keyword **endpackage:** which ends the package.

There can not be two distinct packages with the same name in a common format text body.

♠ **PACKAGE** ::= • **package:** **PACKAGE-HEAD**
{ **PACKAGE-BODY** ... } **endpackage:** •

♠ **PACKAGE-HEAD** ::= • **Number-of-entities** *package*-**identifier** •

♠ **Number-of-entities** ::= • **Integer-const** •

♠ **PACKAGE-BODY** ::= • **Entity-number** **:** **ENTITY** •

♠ **Entity-number** ::= • **Integer-const** •

2.2.2 Entities

Entities are the contents of a package. An entity is formed by a succession of tables in an order determied by the grammer of each entity *(there is more than one table per category of objects in each table).*

The following common information is associated with each entity :

- its format followed by a version number then the name of the entity. The formats actually allowed (and described in this document) are :
  - **dc:**, for data tables (cf. Section 2.4.6, page 30);
  - **ic:**, for IC modules (cf. Section 4, page 45);
  - **gc:**, for GC nodes and interfaces (cf. Section 5, page 69);
  - **oc:**, for OC modules (cf. Section 6, page 97);

- an optional attribute (that doesn't relate to external nodes) denoted by the keyword **flat:** ; this attribute describes the form of expressions (cf. Section 2.2.5, page 20), and thus the *DAGs* (cf. Section 6.5, page 102), given in the entity ; the attribute **flat:** indicates that the expressions and the dags of the entity are flat, which is to say they can't be broken up. By default the attribute **flat:** is absent and the expressions and *DAGs* are allowed in all their generality ;

- optional attribute (which is irrelevant for data tables) gives the behavioral properties of the entity (cf. Section ??, page ??).

In the following are described entities common to the formats (data blocks) and the entities specific (IC modules, GC interfaces, GC nodes, and OC modules). The syntax of each of the entities is given in description of the entity.

♠ **ENTITY** ::= • **DATA-BLOCK** •
• **IC-MODULE** •
• **INTERFACE** •
• **NODE** •
• **OC-MODULE** •

There cannot be two distinct objects with the same name in an entity.

2.2.3 Tables

The table is the universal means to describe objects in the format. All the tables have the same structure. A table contains in order :

1. a keyword identifying the category of objects which are contained in it (for example **functions:**, **signals:**, etc.) ;
2. the number of the objects that it contains (this can be zero if the table contains no entries) ;
3. information specific to each category of table ;
4. the list of objects. Each object is preceded by a number (followed by the **:** character) which names it; it is followed by the **;**. The objects in the table are listed one after the other in the order of their numbers. This numbering without exception starts at `0` ;
5. the keyword **end:** that terminates the table.

In order to give a systematic rewriting, we give below a generic syntax for tables. Here the suffix **X** defines different categories possible for tables (it can appear either single or plural, upper case or lower case).

♠ **X-TABLE** ::= • $\boxed{\textbf{X-table-name}}$ **X-TABLE-HEAD**
[{ **X-TABLE-HEAD** ... }] $\boxed{\textbf{end:}}$ •

♠ **X-TABLE-HEAD** ::= • **Nombre-of-entries** [**X-TABLE-INFO**] •

♠ **Number-of-entries** ::= • **Integer-const** •

♠ **X-TABLE-HEAD** ::= • **Entry-number** $\boxed{\textbf{:}}$ **OBJECT-X** $\boxed{\textbf{;}}$ •

♠ **Entry-number** ::= • **Integer-const** •

In in the tables of predefined objects, each entry number is preceded by a prefix giving the name of the predefined objects in the table (cf. Section **??**, page **??**) :

♠ **X-TABLE-BODY** ::= • **Predefined-object-index** $\boxed{\textbf{:}}$ **OBJECT-X** $\boxed{\textbf{;}}$ •

Tables whose names are given below are "**X** tables" common to all the formats :

♠ **Import-table-name** ::= • $\boxed{\textbf{import:}}$ •

♠ **Type-table-name** ::= • $\boxed{\textbf{types:}}$ •

♠ **Constant-table-name** ::= • $\boxed{\textbf{constants:}}$ •

♠ **Function-table-name** ::= • $\boxed{\textbf{functions:}}$ •

♠ **Procedure-table-name** ::= • $\boxed{\textbf{procedures:}}$ •

♠ **Pragma-table-name** ::= • $\boxed{\textbf{pragmas:}}$ •

For each category of "**X** objects," there are described the following :

- The **X-TABLE-INFO**. (when they are not described for a class of tables **X**, the **X-TABLE-INFO** is empty; when they are present, each piece of the supplementary information out to the end with a $\boxed{\textbf{;}}$ character);
- **OBJECT-X**s.

2.2.4 Indexing and Address Mechanisms

Index of an Object The address of an **object** in a package is given by an index. An index can be an index of a predefined **object** a local index or a compound index.

- An *predefined object index* is composed of :
  - a $\boxed{\textbf{\$}}$ character followed by the number of the designated object belonging to a table of predefined objects common to the three formats ;
  - the characters $\boxed{\textbf{\$I}}$ followed by a specific index for predefined objects common to the imperative formats IC and OC ;
  - the characters $\boxed{\textbf{\$g}}$ followed by an index specific to the predefined objects of the GC format.
- A *local index* is formed with the number of the designated object in a table. It allows for the reference, inside an entity, of an object defined in one of the tables of the entity.
- A *compound index* is formed with :
  - an index into the importing table of the current entity in which there is the number of the entity – an *external index* – containing the designated object,
  - followed by a $\boxed{\textbf{.}}$ character or a local index in an entity table named by an external index.

  In a packet, these allow, in the interior of an entity, for an object in another entity to be referenced by “virtual address.”
- An *undefined index* is denoted by the character $\boxed{-}$. This allows the omission of an index in a syntactic position where one is expected.

The category of **object** that is referenced by through an index is determined by the context of Duhse of the index.

External Indices

- An external index is :
  - an entity number in the current package,
  - a package identifier containing the designated entity, followed by the character $\boxed{\textbf{.}}$ then the number of the entity in the packet.

Index Syntax

♠ **Index** ::= • **Local-index** •
• **Predefined-object-index** •
• **Compound-index** •
• **Undefined-Index** •

♠ **Local-index** ::= • **Body-number** •

♠ **Predefined-object-index** ::= • $\boxed{\$}$ **Integer-const** •
• $\boxed{\$I}$ **Integer-const** •
• $\boxed{\$g}$ **Integer-const** •

♠ **Compound-index** ::= • **Import-index** $\boxed{.}$ **Local-index** •

♠ **Import-index** ::= • **Index-const** •

♠ **Undefined-index** ::= • $\boxed{-}$ •

Indices in this document are associated with a particular category of **object X** by means of the notation X-**index**.

Import Table The import table present in an entity `E` offers a mechanism for associative access to other entities. The list of external entities used is found in the entity `E`; these uses occur in the form of of an index into the import table.

Recal that the general syntax of tables is described in 2.2.3.

♠ **Imported-object** ::= • **Entity-number** •
• *packet*-**identifier** $\boxed{.}$ **Entity-number** •

2.2.5 Common Syntax of Expressions

An expression is a combination of predefined atoms (cf. Section 2.4.1, page 25), references to constants (cf. Section 2.4.2, page 26), function calls (cf. Section 2.4.3, page 27), and, according to the context of use, expressions specific to the imperative formats (cf. Section 3.1.3, page 36) or expressions specific to GC (cf. Section 5.2.2, page 72).

Atoms are made up of two lexical units : the $\boxed{\#}$ character immediately followed by a string of characters or a number. For example example `#12` denotes the integer 12 and `#"Hello"` denotes the string `"Hello"`.

A reference to a constant (**ConstantRef**) is formed by the $\boxed{@}$ character immediately followed by the index of the constant.

The names of functions consist of the index of the function followed by parenthesis, a list of expressions (separated by commas) corresponding to the parameters of the function.

♠ **EXPRESSION** ::= • **Term** •
• **FUNCTION-CALL** •

♠ **Term** ::= • **Atom** •
• **ConstantRef** •
• **IC-OC-Term** •
• **GC-Term** •

♠ **Atom** ::= • $\boxed{\#}$ **String-const** •
• $\boxed{\#}$ **Integer-const** •
• $\boxed{\#}$ **Real-const** •

♠ **ConstantRef** ::= • $\boxed{@}$ *constant*-**index** •

♠ **FUNCTION-Call** ::=
• *function*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •

Expressions cannot be broken (**FLAT-EXPRESSION**) in a package which has the $\boxed{\textbf{flat:}}$ attribute associated with it.

♠ **FLAT-EXPRESSION** ::= • **Term** •
• **FLAT-FUNCTION-CALL** •

♠ **FLAT-FUNCTION-CALL** ::= • *function*-**index** $\boxed{(}$ [{ **Term** $\boxed{,}$ ... }] $\boxed{)}$ •

To understand expressions given below by example, refer to the predefined functions listed on page 28.

For an object `X` of type integer with index value 3, which is available (in a module), or a stream (in a node) :

- `$14(3,#1)` evaluates to the result of the integer addition of the integer constant `1` and `X`;
- `$10(@2,3)` evalutes to *true* if the integer constant with index 2 is strictly smaller than `X`, *false* otherwise.

2.3 Pragmas

Pragmas are "executable comments," specific to certain languages or certain tools.

2.3.1 Syntax of Pragmas

All the text between two **marks** $\boxed{\%}$ is a pragma. A pragma is composed of a succession of **Characters** and strings, or an index into a table of pragmas (a local index for the current entity, compound for an importing entity). To write the character $\boxed{\%}$ in a pragma (this does not relate to those $\boxed{\%}$ characters in character strings in inside pragmas), one writes $\boxed{\backslash\%}$. To write the $\boxed{"}$ character in a pragma one uses $\boxed{\backslash"}$. Pragmas cannot be broken.

There are two types of pragmas :

- pragma "referenced in the table of pragmas" consists of an index between $\boxed{\%}$ characters;
- the other pragmas are identified by a keyword following immediately after the first $\boxed{\%}$: this form allows each tool tool to determine if it needs to be concerned with the pragma, or if it ought to treat it as a simple comment. The "value" of the pragma composed of different pragma lexical units is given by the text that follows the keywords up until the closing $\boxed{\%}$.

♠ **Pragma** ::= • **Reference-pragma** • **Standard-pragma** •

♠ **Reference-pragma** ::= • $\boxed{\%}$ *pragmas*-**index** $\boxed{\%}$ •

♠ **Standard-pragma** ::= • $\boxed{\%}$ **Pragma-name pragma-value** $\boxed{\%}$ •

♠ **Pragma-value** ::= • [{ **Pragma-unit** … }] •

♠ **Pragma-unit** ::= • **PragmaCharacter** • **String-const** •

♠ **PragmaCharacter** ::= • { **Charactere** ⊖ **char-spec-pragma** } •

♠ **char-spec-pragma** ::= • $\boxed{\%}$ • **char-spec-string** •

2.3.2 Pragma Tables and Computed Pragmas

A table of pragmas starts with the keyword $\boxed{\textbf{pragmas:}}$ followed by the number of entries in the table. Then, at the beginning the keyword $\boxed{\textbf{computed:}}$ follwed by a list (possibley empty) of identifier keywords and the $\boxed{;}$ character. This list indicates the pragmas that a code generation tool must have computed.

A syntactic structure that can be associated (after the definition of the pragma) with the listed pragmas is automatically defined, being an explicit value (a pragma in the code) or with the default value specified in the definition of the pragma (for example, an empty list for pragmas whose value is a list).

The table contains next the list of of entries and ends with $\boxed{\textbf{end:}}$. Each entry in the table is made up of its index, followed by a $\boxed{:}$, a succession of pragmas (with their $\boxed{\%}$ delimiters) and the $\boxed{;}$ character.

♠ **PRAGMA-INFO-TABLE** ::=

• $\boxed{\textbf{computed:}}$ [{ **Pragma-name** ... }] $\boxed{;}$ •

♠ **PRAGMA-OBJECTS** ::= • { **Pragma** ... } •

Tools that translate one format to another or which manipulate the code of a data format (for example, compilers, optimizers) ought, if at all possible, to transmit the unrecognized pragmas as they are and remove the names of unrecognized pragmas in the list of calculated pragmas. In effect if a tool does not understand a data pragma, it should maintain that the pragma had been correctly calculated for all the syntactic structures to which it could be associated, and in particular for the additions or modifications of the tool.

2.3.3 Rules of Association for Pragmas

Pragmas can appear any place in a package.

The rules of association for pragmas are different according to whether the pragma is found in a table or not.

- A pragma which is not found in a table is associated with the entity (data block, module, node or interface) that contains it directly, or, if such an entity does not contain it to the package in which it is found.

- A pragma in a table is associated with the element that immediately precedes it in the table according to the following rules :

  - a pragma placed between the keyword identifying the table and the keyword identifying the first piece of information that it contains, by failing that the first entry of the table, or the end of the table (if the table contains nothing), is associated with the table;
  - a pragma placed immediately after the $\boxed{;}$ character, ending a piece of information or an entry in the table is associated with that bit of information or entry;

– a pragma immediately following an element (the abstraction made by the separators) of an entry or a piece of information in the table, is associated with that element.

2.3.4 Definition Procedure for a Pragma

The definition of a pragma consists in giving the following six elements :

- the keyword identifying the pragma (or equivalently the "name of the pragma") ;
- the list of objects to which the the pragma can be associated ;
- the precise syntax of the value of the pragma (**Pragma-value**) ;
- the meaning of the pragma, possibly the funtion of the object with which it is associated ;
- the authority or not to create the name of this pragma (its keyword) in the list of computed pragmas of the entity ;
- if the name of the pragma can appear in the list of computed pragmas of an entity, the default value of the pragma should be given.

Finally, in the event of name collisions and to define the contents of pragmas a pragma registry service ("Bureau d'Enregistrement des Pragmas Standards" – BEPS) will have the duty to adjudicate requests. The requests ought to consist of the six (or five if there is no default value) elements described above for each pragma.

Appendix A of this documents defines a certin number of pragmas, in particular the comment pragma, and consititutes the first request for registry with the BEPS.

2.4 Organization of Common Data

The description of value domains (the elements of synchronus effects) and the operations that apply to them are organized by tables in data blocks. We describe in order the common types, constants, functions and procedures on these types, and the data blocks that contain them.

Used in particular to construct interfaces with friendly environments that are not necessarily synchronous, procedures and functions given in the data blocks ought to respect the rules permitting a usage consistent with the synchronous assumptions. Finally for the characteriation the effects possible in their invocation, a semantic attribute (cf. Section 2.4.5, page 30) is associated with their declaration. This attribute can equally be used to characterize nodes and modules.

2.4.1 Types

A type descriptor is essentially an identifier giving its name. Each type can be associated with procedures and functions providing operations on that type. Among these functions and procedures one distinguishes :

- a procedure copying the type. This procedure takes one input parameter and one return parameter of the given type ; it moves the parameter value between the input parameter and the return parameter. this procedure will have the es predefined name `$assign` ;
- an equality function for the type. This function takes two arguments of the given type and returns a boolean truth value (the constant `$2`) only if the arguments are equal and false (the constant `$1`) otherwise. This function will have the predefined name `$eq` ; this function ought to be *safe* (cf. Section 2.4.5, page 30) ;
- an inequality function for the type. This function takes two arguments of the given type and returns as a result the boolean negation of the `$eq` function. This function will have the predefined name `$ne` ; this function ought to be *safe* ;
- a conditional function of the type. This function takes an argument of boolean type (`$1`) and two arguments of the given type ; it returns the first of the two objects if the boolean argument is true (the constant `$true`) and the second object otherwise. This function will have the predefined name `$cond` ; this function ought to be *safe*.

Remark : If for a given type, the assignment procedure or one of the functions is not used, then it is not necessaryily provided in the corresponding table.

♠ **TYPE-OBJECT** ::= • *type*-**identifier** •

The common predefined types are given in Table 2.1 ; the assignment procedure and the equality, inequality and conditional functions associated with each appear in comments in this table. The predefined functions and procedures associates with common predefined types are detailed in Tables 2.5 and 3.1.

The boolean (`$1`) possesses all the standard algebraic properties of booleans. The boolean constants `$true` (`$2`) and `$false` (`$1`) are predefined in the constants table.

The type `$pure` (`$0`) posses a unique value denoted by the predefined constant `$top` whose index is `$0` (cf. Section 2.4.2, page 26).

The equality and inequality functions are supposed to be coherent (which is to say that for any two objects, one and only one of the two functions ought

```
types: 6
$0: $pure;    --                   $0  code $eq
$1: $boolean; -- $0 code $assign, $1  code $eq, $2  code $ne, $3  code $cond
$2: $integer; -- $1 code $assign, $7  code $eq, $8  code $ne, $9  code $cond
$3: $string;  -- $2 code $assign, $20 code $eq, $21 code $ne, $22 code $cond
$4: $float;   -- $3 code $assign, $23 code $eq, $24 code $ne, $25 code $cond
$5: $double;  -- $4 code $assign, $35 code $eq, $36 code $ne, $37 code $cond
end:
```

Table 2.1: Common Predefined Types

```
types: 1
0: A_TYPE;  -- 0 code $assign, 0 code $eq, 1 code $ne, 2 code $cond
end:
```

Table 2.2: One Type Table

to return the boolean constant true $2). The replacement of the negation of an equality by an inequality must be invariant semantically.

2.4.2 Constants

A constant descriptor is the name of the constant followed by an index denoting its type. Next comes an optional field indicated by the keyword $\boxed{\textbf{value:}}$ defining, with an expression, the value of the constant.

♠ **CONSTANT-OBJECT** ::=

• *constant*-**identifier** *type*-**index** [**CONSTANT-VALUE**] •

♠ **CONSTANT-VALUE** ::= • $\boxed{\text{value:}}$ **EXPRESSION** •

```
constants: 3
$0: $top $0;
$1: $false $1;
$2: $true $1;
end:
```

Table 2.3: The Common Predefined Constants

```
constants: 3
0: LINESIZE $2;
1: COLUMNSIZE $2;
2: ARRAYSIZE $2 value: $16(@0,@1);
end:
```

Table 2.4: A Table of Constants

2.4.3 Functions

A function is a syntactic structure that allows, within expressions, the positional invocation of functions in the usual sense of host languages. The description of a function contains in order :

- an identifier giving the name of the function ;
- the list of indices of the argument types (between parentheses and separated by commas) ;
- the index of the type of the return value preceded by a `:` ;
- possibly the keywords `memorysafe:` or `unsafe:` (cf. Section 2.4.5, page 30), in the absence of which the function is deemed to be *safe*.

♠ **FUNCTION-OBJECT** ::=

- *function*-**identifier** `(` [{ *type*-**index** `,` ... }] `)` `:` *type*-**index** [**computational-property**] •

Recall that for an arbitrary type μ, there can exist predefined functions with the names `$eq`, `$ne` and `$cond`.
Example of a table of functions :

```
functions: 5
0: $eq (0, 0): $1;
1: $ne (0, 0): $1;
2: $cond ($1, 0, 0): 0;
3: MODULO ($2, $2): $2;
4: RANDOM (): $4 unsafe:;
end:
```

In the preceding description, the name of the function used (such as `MODULO`) is an identifier. Type manipulation functions whose indices appear in the type table have predefined names. The function `RANDOM`, of no arguments, returns a predefined floating point type `$4` which due to the keyword `unsafe:` can return different values with successive calls.

Table 2.5 gives a list of some common predefined functions. A refernece to one of these functions is a *predefined-object*-**index** formed by the `$` symbol, followed by its number in that table

These functions are strict. Besides, the functions `$eq`, `$ne` and `$cond`, there are :

- comparison functions for numeric values (the types `$integer`, `$float` and `$double`)

```
functions: 50
-- pure
 $0: $eq ($0, $0): $1;
-- boolean
 $1: $eq ($1, $1): $1;
 $2: $ne ($1, $1): $1;
 $3: $cond ($1, $1, $1): $1;
 $4: $or ($1, $1): $1;
 $5: $and ($1, $1): $1;
 $6: $not ($1): $1;
-- integer
 $7: $eq ($2, $2): $1;
 $8: $ne ($2, $2): $1;
 $9: $cond ($1, $2, $2): $2;
$10: $lt ($2, $2): $1;
$11: $le ($2, $2): $1;
$12: $gt ($2, $2): $1;
$13: $ge ($2, $2): $1;
$14: $plus ($2, $2): $2;
$15: $minus ($2, $2): $2;
$16: $times ($2, $2): $2;
$17: $mod ($2, $2): $2;
$18: $div ($2, $2): $2;
$19: $uminus ($2): $2;
-- string
$20: $eq ($3, $3): $1;
$21: $ne ($3, $3): $1;
$22: $cond ($1, $3, $3): $3;
-- float
$23: $eq ($4, $4): $1;
$24: $ne ($4, $4): $1;
$25: $cond ($1, $4, $4): $4;
$26: $lt ($4, $4): $1;
$27: $le ($4, $4): $1;
$28: $gt ($4, $4): $1;
$29: $ge ($4, $4): $1;
$30: $plus ($4, $4): $4;
$31: $minus ($4, $4): $4;
$32: $times ($4, $4): $4;
$33: $div ($4, $4): $4;
$34: $uminus ($4): $4;
-- double
$35: $eq ($5, $5): $1;
$36: $ne ($5, $5): $1;
$37: $cond ($1, $5, $5): $5;
$38: $lt ($5, $5): $1;
$39: $le ($5, $5): $1;
$40: $gt ($5, $5): $1;
$41: $ge ($5, $5): $1;
$42: $plus ($5, $5): $5;
$43: $minus ($5, $5): $5;
$44: $times ($5, $5): $5;
$45: $div ($5, $5): $5;
$46: $uminus ($5): $5;
-- conversions
$47: $conv ($2): $4;
$48: $conv ($2): $5;
$49: $conv ($4): $5;
end:
```

Table 2.5: Common Predefined Functions

- `$lt` whose result is true if the first argument is strictly smaller than the second, false otherwise.
- `$le` whose result is true if the first argument is less than or or equal to the second, false otherwise.
- `$gt` whose result is true if the first argument is strictly greater than the second, false otherwise.

  - `$ge` whose result is true if the first argument is greater than or equal to the second, false otherwise.

- Functions of addition (`$plus`), subtraction (`$minus`), multiplication (`$times`), division (`$div`) and of negation (`$minus`) for numeric values ; numériques ;
- the modulo function (`$mod`) for integer values ;
- conversion functions (`$conv`) between `$integer` and `$float`, between `$integer` and `$double` and `$float` and `$double`.

2.4.4 Procedures

A procedure is a syntactic structure allowing the positional invocation of procedures in the usual sense of host languages. A procedure is invoked by a call to the procedure ($\boxed{\textbf{call:}}$).

The description of a procedure contains, in order :

- an identifier giving the name of the procedure ;
- the list of indices of the argument types, each of them has preceding a keyword indicating is mode :
  - an input parameter (keyword $\boxed{\textbf{i:}}$), where the procedure receives the value ;
  - an output parameter (keyword $\boxed{\textbf{o:}}$), where the procedure defines the value ;
  - or an input-output parameter (keyword $\boxed{\textbf{io:}}$), where the procedure receives the value before redefining it.
- possibly the keywords $\boxed{\texttt{memorysafe:}}$ or $\boxed{\texttt{unsafe:}}$ (cf. Section 2.4.5, page 30), in the absence of which the procedure is presumed to be *safe* (cf. Section 2.4.5, page 30).

♠ **PROCEDURE-OBJECT** ::=

• *procedure*-**identifier** $\boxed{(}$ [{ **PARAMETER** $\boxed{,}$ ... }] $\boxed{)}$
[**Computational-property**] •

♠ **PARAMETER** ::= • **INPUT-PARAMETER** •
• **OUTPUT-PARAMETRE** •
• **INPUT-OUTPUT-PARAMETER** •

♠ **INPUT-PARAMETER** ::= • $\boxed{\textbf{i:}}$ *type*-**index** •

♠ **OUTPUT-PARAMETER** ::= • $\boxed{\textbf{o:}}$ *type*-**index** •

♠ **INPUT-OUTPUT-PARAMETER** ::= • $\boxed{\textbf{io:}}$ *type*-**index** •

2.4.5 Common Semantic Properties of Processes

Whatever their syntactic form, modules, nodes, functions, procedures (called here generically *processes*) enjoy the following semantic properties :

safe **:** a process is said to be "*safe*" if it is a function in the mathematical sense of the term, without memory (i.e. constant over time) ; this property is denoted by the $\boxed{\texttt{safe:}}$ attribute, which is the default attribute for functions and procedures ;

memorysafe **:** a process is said to be "*memorysafe*" if it performs a function of initial states, entry paths, paths of exit clocks in the exit paths of exits ; this property is denoted by the $\boxed{\texttt{memorysafe:}}$ attribute ; for these processes the principle of substitution applies.

unsafe **:** a process is said to be "*unsafe*" in all the other cases ; this property is denoted by the $\boxed{\texttt{unsafe:}}$ attribute.

These properties are ordered as follows :

- all processes that are *safe* are *memorysafe* ;
- all processes that are *memorysafe* are *unsafe.*

♠ **Process-property** ::= • $\boxed{\textbf{safe:}}$ • $\boxed{\textbf{memorysafe:}}$ • $\boxed{\textbf{unsafe:}}$ •

2.4.6 Data Blocks

Data blocks provide a syntactic descriptions of external objects in the format (types, constants, functions and procedures). Many entities can import such blocks to access the objects described there. A data block can itself import other data blocks by giving their external indices in the importation list. This list can not be made of other entities than these data blocks.

Each part is optional except for the indication of the format used and the name.

♠ **DATA-BLOCKSBLOC** ::= • $\boxed{\textbf{data:}}$ **DATA-BLOCK-INFO**
[**IMPORTATION-TABLE**]
[**TYPE-TABLE**]
[**CONSTANT-TABLE**]
[**FUNCTIONS-TABLE**]
[**PROCEDURE-TABLE**]
[**PRAGMA-TABLE**]
$\boxed{\textbf{enddata:}}$ •

♠ **DATA-BLOCK-INFO** ::=

• **Dc-format** *data-block*-**identifier** [$\boxed{\textbf{flat:}}$] •

♠ **Dc-format** ::= • $\boxed{\textbf{dc:}}$ **Version-number** •

Example of data blocks This example is presented in two columns.

```
data: dc:0 ARRAY
   types: 2
     0: ELEM_TYPE;
     1: ARRAY_TYPE;
   end:

   constants: 2
     0: LINESIZE $2;
     1: COLUMNSIZE $2;
   end:
```

```
   functions: 7
     0: GET (1, $2, $2): 0 safe:;
     1: $eq (0, 0): $1;
     2: $ne (0, 0): $1;
     3: $cond ($1, 0, 0): 0;
     4: $eq (1, 1): $1;
     5: $ne (1, 1): $1;
     6: $cond ($1, 1, 1): 1;
   end:

   procedures: 4
     0: $assign (o: 0, i: 0) memorysafe:;
     1: $assign (o: 1, i: 1) memorysafe:;
     2: SET (io: 1, i: $2, i: $2, i:0) memorysafe:;
     3: PROCGET (i: 1, i: $2, i: $2, o: 0) safe:;
   end:
enddata:
```

Chapter 3

Common Elements Between the IC and OC Formats

This chapter describes the common parts which are specific to the two imperative intermediate formats of the synchronous languages.

3.1 Organization of IC and OC Data

3.1.1 Variables

The table of variables is named by the keyword $\boxed{\textbf{variables:}}$. Recall that the general syntax of tables was described in 2.2.3.

♠ **Variable-table-name** ::= • $\boxed{\textbf{variables:}}$ •

This table collects the variables declared in the program and the compiler-generated variables (the status and value of signals, counters etc.). An entry in this table contains two fields, the second being optional. The information given on a variable must be :

- the index of the type of the variable ;
- possibly the initial value of the variable preceded by the keyword $\boxed{\textbf{value:}}$. This value is an expression of the same type as the variable and its syntax was given in Section 2.2.5.

♠ **VARIABLE-OBJECT** ::= • *type*-**index** [**INITIAL-VALUE**] •

♠ **INITIAL-VALUE** ::= • $\boxed{\textbf{value:}}$ **EXPRESSION** •

3.1.2 Signals

The table of signals is named by the keyword $\boxed{\textbf{signals:}}$. Recall that the general syntax of these tables was described in Section 2.2.3.

♠ **Signal-table-name** ::= • $\boxed{\text{signals:}}$ •

There are two kinds of signals : interface signals between the program and its environment on the one hand, and internal signals of the program on the other hand. Interface signals are of one of four types : input, normal termination of a task, output or input/output.
The various objects that can be associated with a signal are as follows :

- with each interface signal is associated a name identifying the signal to the exterior of the module;
- with each signal value is associated a value preserving the current value of the signal;
- with each signal value that was declared multiply emittable in an instant is associated a combiner function that merges two values; this function must take two arguments of the type of the signal value and produce a result of the same type; the operator associated with this function ought to be associative and commutative.

An entry in the signal table collects, according to the type and the value of the signal, the references to objects listed above. An entry in the table consists therefore of the following :

1. a **SIGNAL-KIND**;
2. a **SIGNAL-CHANNEL**;
3. an optional boolean (**SIGNAL-BOOL**).

Described below are each of these three parts :

- **SIGNAL-KIND** represents the various sorts of signals :
  - for input signals, the keyword $\boxed{\textbf{input:}}$ follows the name of the signal;
  - for signals of norrmal task termination, the keyword $\boxed{\textbf{return:}}$ follows the name of the signal; these signals are a unique sort of input signal.
  - for output signals, the keyword $\boxed{\textbf{output:}}$ follows the name of the signal;

  - for input/output signals the keyword $\boxed{\textbf{inputoutput:}}$ follows the name of the signal ;
  - for local signals, the keyword $\boxed{\textbf{local:}}$ is used ;

- **SIGNAL-CHANNEL** represents the value transported by the sigal. A channel is composed of two or three fields :
  - the first field indicates the type of the signal ;
  - if the signal is not of type `$pure`, this information is followed by the index of the variable containing the value ; if the signal is of `$pure`, the second field is omitted ;
  - the second field indicates if the signal can be multiply emitted in the same instant : the field must be $\boxed{\textbf{single:}}$ if the signal can not be emitted more than once, otherwise $\boxed{\textbf{multiple:}}$ followed by the index of the combiner function.

  Warning : a $\boxed{\textbf{return:}}$ signal can not be $\boxed{\textbf{multiple:}}$.

- **SIGNAL-BOOL** consists of the keyword $\boxed{\textbf{bool:}}$ followed by an index of a boolean variable. This field exists :
  - for the signal kinds $\boxed{\textbf{input:}}$, $\boxed{\textbf{return:}}$ and $\boxed{\textbf{inputoutput:}}$. It is used to test for the presence of the signal.
  - For signal channels which are $\boxed{\textbf{multiple:}}$. It is used to mark if the signal has already been emitted in the instant and so to know to whether or not to apply the combiner function for emissions following the first.

♠ SIGNAL-OBJECT ::= • **SIGNAL-KIND SIGNAL-CHANNEL [SIGNAL-BOOL]** •

♠ SIGNAL-KIND ::= • $\boxed{\textbf{input:}}$ *signal*-**identifier** •
• $\boxed{\textbf{return:}}$ *signal*-**identifier** •
• $\boxed{\textbf{output:}}$ *signal*-**identifier** •
• $\boxed{\textbf{inputoutput:}}$ *signal*-**identifier** •
• $\boxed{\textbf{local:}}$ •

♠ SIGNAL-CHANNEL ::=

• *type*-**index** [**ASSOCIATED-VARIABLE**] **CARDINALITY** •

♠ ASSOCIATED-VARIABLE ::= • $\boxed{\textbf{value:}}$ *variable*-**index** •

♠ **CARDINALITY** ::= • **single:** •
• **multiple:** *function*-**index** •

♠ **SIGNAL-BOOL** ::= • **bool:** *variable*-**index** •

3.1.3 Expressions Specific to IC and OC

The expressions specific to the imperative formats are the expressions on signals and variables which are completed with expressions described in Section 2.2.5 :

- the presence status of a signal is represented by the ! character followed by the signal index ;
- the current value of a signal is represented by the ? character followed by the signal idnex ;
- a variable is simply referenced by its index.

♠ **IC-OC-Term** ::= • **Signal-presence** •
• **Signal-value** •
• *variable*-**index** •

♠ **Signal-presence** ::= • ! *signal*-**index** •

♠ **Signal-value** ::= • ? *signal*-**index** •

In an IC module (cf. Section 4.2, page 45) and in an OC module (cf. Section 6.2, page 97) :

- the indices of signals and variables appearing in in an expression are local indices ;
- the indices of constants appearing in an expression can be indices of predefined common constants or compound indices naming constants in imported data blocks ;
- the indices of functions appearing in an expression can be indices of common predefined functions or compound indices naming functions from an imported or predefined data block

3.1.4 Types Specific to IC et OC

There are no types specific to the imperative formats.

3.1.5 Functions and Procedures

The parameters of type $\boxed{\textbf{i:}}$ in procedures and functions are expressions on variables and signals; the parameters $\boxed{\textbf{io:}}$, $\boxed{\textbf{o:}}$ must be variables.

3.1.6 Predefined Functions Specific to IC and OC

The imparative formats use the following three supplementary functions :

- `left_and` and `left_or` are the two classical boolean functions *and* and *or* that do not evaluate both their arguments at once.
- the function `dsz` decrements the variable passed to it and returns *true* (constant `$2`) if and only if the new value of the variable is `0`.

```
functions: 3
  -- boolean
  $I0: $left_and ($1, $1): $1;
  $I1: $left_or  ($1, $1): $1;
  -- integer
  $I2: $dsz ($2): $1 unsafe:;
end:
```

3.1.7 Predefined Procedures Specific to IC and OC

Recall that for any type μ, there can exist an assignment procedure with the predefined name `$assign`.

Table 3.1 gives the predefined procedures, they are assignment procedures associated with the predefined types.

```
procedures: 5
  $I0: $assign (o: $1, i: $1);
  $I1: $assign (o: $2, i: $2);
  $I2: $assign (o: $3, i: $3);
  $I3: $assign (o: $4, i: $4);
  $I4: $assign (o: $5, i: $5);
end:
```

Table 3.1: Predefined Procedures

3.1.8 Example

The ESTEREL declarations are as follows :

```
type USER_TYPE;
function PLUS(integer, integer) : integer;
output O : combine integer with PLUS;
input T : USER_TYPE,
      S;
```

producing in one package for example :

- a data block containing the three following tables :

```
types: 1             functions: 4              procedures: 1
   0: USER_TYPE;        0: PLUS ($2,$2):$2;       0: $assign (o: 0, i: 0);
end:                    1: $eq (0,0):$1;       end:
                        2: $ne (0,0):$1;
                        3: $cond ($1,0,0):0;
                     end:
```

- a module containing an import table containing (for example) in position one, the index of a data block 3 under it and the following tables :

```
variables: 5         signals: 3
   0: $2;               0: output: O $2 value: 0 multiple: 1.0 bool: 1;
   1: $1;               1: input: T 1.0 value: 2 single: bool: 3;
   2: 1.0;              2: input: S $0           single: bool: 4;
   3: $1;            end:
   4: $1;
end:
```

3.2 Tasks

Tasks are external processes which are executed by the controlling system in parallel with the synchronous program ; this is accomplished by specific action s (`start`, `kill`, `suspend` et `resume`).

3.2.1 Task Description

The task table is named by the keyword **tasks:**. Recall that the general syntax of tables was described in Section 2.2.3.

♠ **Task-table-name** ::= • **tasks:** •

The description of a task contains, in order :

- an identifier giving the name of the task ;
- the list of indices of argument types, each of these being preceded by a keyword indicating its mode :
  - an input parameter (the **i:** keyword), that the task can read but cannot modify ;
  - an output parameter (the **o:** keyword), to which the task can write a value but from which it can not read at all ;
  - or an input/output parameter (the **io:** keyword) that the task can both read and write.

The syntax is thus given as follows :

♠ **TASK-OBJECT** ::=

• *task*-**identifier** ([{ **PARAMETER** , ... }]) [**Computational-property**] •

3.2.2 Description of a Task Invocation

The table of task invocations is named by the keyword **execs:**. Recall that the general syntax of tables was described in Section 2.2.3.

♠ **Task-table-name** ::= • **execs:** •

The task invocation table gathers the various task calls that appear in the source code. A task call in the source corresponds to one and only one entry in the table of task invocations. There can be many invocations for the same task with the same arguments in the same way that there can be such for a procedure.

A task invocation gathers the information necessary to control the execution of the task. This is thus in order :

- the index of the associated task in the task table.
- the index of the normal termination signal (a **return:** signal) of the task (cf. Section 3.1.2, page 34),
- A list of expressions between parenthesis and separated by commas which are the arguments to the task when it is called. For all formal parameters declared with the mode **o:** or **io:** there must be an actual parameter that is a variable index.

♠ **TASK-CALL-OBJECT** ::=

• *task*-**index** *signal*-**index** ([{ **EXPRESSION** , ... }]) •

3.2.3 Example

The Esterel code is as follows : gives the following tables :

```
type POSITION;

task MOVE_ARM(POSITION)(),
     CLOSE_HAND()(),
     OPEN_HAND()();

...

   exec OPEN_HAND()();
   exec MOVE_ARM(POS)();
   exec CLOSE_HAND()();
...
   exec OPEN_HAND()();
```

```
tasks: 3
   0: MOVE_ARM (io: 0);
   1: CLOSE_HAND ();
   2: OPEN_HAND ();
end:

execs: 4
   0: 2 0 ();
   1: 0 1 (0);
   2: 1 2 ();

   3: 2 3 ();
end:
```

3.3 Actions

The actions table is named by the keyword **actions:**. Recall that the general syntax of tables was described in Section 2.2.3.

♠ **Action-table-name** ::= • **actions:** •

An action description is a non-empty list of elementary actions (without separators). These elementary actions can be classified into 12 categories :

- `act:` ⟨*action-index*⟩ executes the action of the index ⟨*action-index*⟩. *Links among the elementary actions must not create a cycle.*
- `if:` ⟨*expression*⟩ tests the boolean expression (of type `$1`) ⟨*expression*⟩. This elementary action cannot be used as the last action in a list because the choice of the action following it depends on the result of the test. For example in "`if: $10(@3, 2)`" which is the translation of "`if LIMIT_TIME < TIME`" if `LIMIT_TIME` is the constant 3 and `TIME` is the variable 2; then `if: !3` tests for the presence of the signal with index 3.
- `call:` ⟨*procedure-index*⟩`(`⟨*expression-list*⟩`)` calls the procedure referenced by ⟨*procedure-index*⟩ with the arguments given in the expression list. All formal parameters declared with the prefix **o:** or **io:** must correspond to an actual parameter which is the variable index.
- `input:` ⟨*signal-index*⟩ acquires the presence and the value of the signal with index ⟨*signal-index*⟩ from the exterior.

- `output:` ⟨*signal-index*⟩ emits to the exterior the output signal with index ⟨*signal-index*⟩.
- `reset:` ⟨*variable-index*⟩ resets the variable with index ⟨*variable-index*⟩ to its initial value. If the variable does not have an initial value then it is put in the *uninitialized* state.
- `combine:` ⟨*signal-index*⟩⟨*expression*⟩ computes the value of a multiple signal. If the signal has never been emitted the value of the signal becomes ⟨*expression*⟩. If the signal has already been emitted, it has a value ; its value and ⟨*expression*⟩ are passed as arguments to the combiner function for the signal and the result becomes the new value of the signal. Recall that the conbiner functions take two arguments and return a result of the same type : they give the value of the signal. For example, supposing that the constant 5 names `C`,

  `combine: 3 $14(#3, @5)` causes `C+3` to become the value of the signal 3 if it has not yet been emitted. If the signal has already been emitted, the new value is the result of the application fo the function to the combination of the old value and `C+3`.
- `start:` ⟨*exec-index*⟩ starts the execution of a new instance of the task given by ⟨*exec-index*⟩.
- `kill:` ⟨*exec-index*⟩ kills an instance of the task given ⟨*exec-index*⟩.
- `return:` ⟨*exec-index*⟩ writes the pass-by-reference parameters (prefixed by **o:** or **io:**) of the task instance given by the ⟨*exec-index*⟩.
- `suspend:` ⟨*exec-index*⟩ suspends the execution of the task instance given by ⟨*exec-index*⟩.
- `resume:` ⟨*exec-index*⟩ restarts the execution of the task instance given by ⟨*exec-index*⟩.

The actions themselves are classified into two categories. On can distinguish in effect between “pure” actions and test actions :

- a pure action is an action that does not contain any elementary actions of the types **input:**, **output:**, **call:**, **reset:**, **combine:**, **start:**, **kill:**, **return:**, **suspend:** ou **resume:** or any elementary actions of type **act:** referencing pure actions ;
- a test action is a pure action of an elementary **if:** or an action of type **act:** referencing a test action.

♠ **ACTION-OBJECT** ::= • { **ELEMENTARY-ACTION** ... } •

♠ **ELEMENTARY-ACTION** ::=

- • $\boxed{\textbf{act:}}$ *action*-**index** •
- • $\boxed{\textbf{if:}}$ **EXPRESSION** •
- • $\boxed{\textbf{call:}}$ *procedure*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •
- • $\boxed{\textbf{input:}}$ *signal*-**index** •
- • $\boxed{\textbf{output:}}$ *signal*-**index** •
- • $\boxed{\textbf{reset:}}$ *variable*-**index** •
- • $\boxed{\textbf{combine:}}$ *signal*-**index** **EXPRESSION** •
- • $\boxed{\textbf{start:}}$ *exec*-**index** •
- • $\boxed{\textbf{kill:}}$ *exec*-**index** •
- • $\boxed{\textbf{return:}}$ *exec*-**index** •
- • $\boxed{\textbf{suspend:}}$ *exec*-**index** •
- • $\boxed{\textbf{resume:}}$ *exec*-**index** •

By way of example, here is the table of actions for a small ESTEREL program :

```
actions: 26
0: if:!0;
1: if:!1;
2: if:!2;
3: if:!3;
4: output:4;
5: output:5
    call:$1 (11) (@0)
    call:$1 (10) (#0);
6: output:6;
7: output:7;
8: output:8
    act:10;
9: output:9;
10: output:10;
11: output:11;
12: call:$1 (4) (#0)
    act:25;
13: call:$1 (4) (10)
    call:$1 (5) (10)
    call:$1 (13) ($14 (13,5))
    call:$1 (14) ($14 (14,#1))
    call:$1 (6) ($18 (13,14))
    act:25;
14: call:$1 (4) (6)
    act:4;
15: call:$1 (8) (@0);
16: call:$1 (9) (0 ());
17: call:$1 (10) ($14 (10,#1));
18: call:$1 (12) (@2);
19: call:$1 (13) (#0)
    call:$1 (14) (#0)
    call:$1 (7) (@1)
    if:$12 (7,#0);
20: if:$19 (7);
21: if:$19 (8);
22: if:$19 (9);
23: if:$19 (11);
24: if:$19 (12);
25: act:4
    act:6;
end:
```

3.4 Instances

The table of instances is named by the keyword $\boxed{\textbf{instances:}}$. Recall that the general syntax of tables was described in Section 2.2.3.

This table is used to relate generic source code and the corresponding program source. The module to which this table belongs is called the *root module*. In the instance table, there is an entry for each instance of a submodule used (in a `Run` instruction) within the root module. The index of the root moodule is given by the supplementary information $\boxed{\textbf{root:}}$.

♠ **Instance-table-name** ::= • $\boxed{\textbf{instances:}}$ •

♠ **INSTANCE-TABLE-INFO** ::= • $\boxed{\textbf{root:}}$ *instance*-**index** $\boxed{\;;\;}$ •

Each entry in the instance table contains :

- an identifier giving the name of the module ;

- the index of the module (called the *parent module*) using the module corresponding to that entry. For the root module the undefined index $\boxed{-}$ is used.

♠ **INSTANCE-OBJECT** ::= • *module*-**identifier** *parent-module*-**index** •

3.5 Relations

The relations table is identified by the keyword $\boxed{\textbf{relations:}}$. Recall that the general syntax of tables was described in Section 2.2.3.

♠ **Relations-table-name** ::= • $\boxed{\textbf{relations:}}$ •

This table gathers the relations between signals given in the program source code. The information is :

- a list of signal indices giving the signals that can not be present simulataneously in an instant ;
- two indices of signals where the presnce of the first signal implies the presence of the second.

♠ **RELATION-OBJECT** ::= • **EXCLUSION** • **IMPLICATION** •

♠ **EXCLUSION** ::= • $\boxed{(}$ *signal*-**index** $\boxed{,}$ { *signal*-**index** $\boxed{,}$ ... } $\boxed{)}$ •

♠ **IMPLICATION** ::= • *signal*-**index** *signal*-**index** •

Chapter 4

The Parallel Imperative Format : IC

4.1 Introduction

The IC format is an imperative and parallel representation of a synchronous program. An IC code is composed of two parts :

- one declarative part defines objects such as signals, variables, ...
- the other imperative part encodes the behavior of the program.

Structure of an IC Code

An IC code consists of a list of declarations of two types of entities : *data blocks* and *modules*.

The low level entity of an IC code is called a *module*. A module described in the IC format is composed of tables. The tables are of two types :

- data tables (signals, variables, tasks, ...)
- a table of instructions forming the body of a module using the objects defined in the other tables.

4.2 Modules

An IC module defines the behavior and the interface of a synchronous program. A module described in the IC format consists of :

- the keyword $\boxed{\textbf{module:}}$ following the IC format then the name of the module ;

- the optional attributes of a module among which the attribute `linked:` indicates that the module does not contain any **Run:** instructions;
- the optional tables common to the imperative formats;
- the table of instructions;
- the keyword **endmodule:**.

♠ **IC-MODULE** ::= • **module:** **IC-MODULE-INFO**
COMMON-IMPERATIVE-PART
INSTRUCTIONS-TABLE
endmodule: •

♠ **IC-MODULE-INFO** ::= • **Ic-format** *module*-**identifier**
[**flat:**] [**linked:**]
[**Computational-property**] •

♠ **COMMON-IMPERATIVE-PART** ::= • [**IMPORTATION-TABLE**]
[**INSTANCE-TABLE**]
[**VARIABLE-TABLE**]
[**SIGNAL-TABLE**]
[**RELATION-TABLE**]
[**TASK-TABLE**]
[**EXEC-TABLE**]
[**ACTION-TABLE**]
[**ic-PRAGMAS-TABLE**] •

♠ **Format** ::= • **ic:** **Version-number** •

4.3 Instructions

The instructions table is identified by the keyword **instructions:**. Recall that the general syntax of tables was described in Section 2.2.3.

♠ **Instructions-name** ::= • **instructions:** •

The information specific to instruction tables is the following :

- the instruction index of the starting point is preceded by the keyword **startpoint:**; this instruction must be a control instruction (c.f. below);
- the number of `Goloop` instructions preceded by the keyword **goloops:**;

- the maximum number of levels of `Exit` instructions preceded by the keyword **exitlevels:**.

♠ **INSTRUCTION-TABLE-INFO** ::= • **startpoint:** *instruction*-**index** **;**
goloops: **Integer-const** **;**
exitlevels: **Integer-const** **;** •

The following sorts of instructions exist :

- *control instructonis* whose *arity* is defined by the number of continuations in them :
  - `Exit`, `Halt`, `Watchdog`, `Stay` and `Return` are of arity 0.
  - `Action`, `Reset`, `Emit`, `Access`, `Goto`, `Goloop` and `Run` are of arity 1,
  - `Test` has arity 2,
  - `Fork` is of arity greater than 0 ;
- *reconstruction instructions*
  - `Halt`, `Watchdog`, `Stay` and `Return` are also control instructions (of arity 0).
  - `Parallel`.

♠ **INSTRUCTION-OBJECT** ::=

- $\boxed{\textbf{Action:}}$ $\boxed{[}$ *action*-**index** $\boxed{]}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Test:}}$ $\boxed{[}$ *action*-**index** $\boxed{]}$
  $\boxed{(}$ *instruction*-**index** $\boxed{,}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Reset:}}$ $\boxed{[}$ *signal*-**index** $\boxed{]}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Emit:}}$ $\boxed{[}$ *signal*-**index** $\boxed{]}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Access:}}$ $\boxed{[}$ *signal*-**index** $\boxed{]}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Goto:}}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Goloop:}}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Fork:}}$ $\boxed{\{}$ *instruction*-**index** $\boxed{\}}$ $\boxed{(}$ { *instruction*-**index** $\boxed{,}$ … } $\boxed{)}$ •
- $\boxed{\textbf{Parallel:}}$ $\boxed{(}$ { *instruction*-**index** $\boxed{,}$ … } $\boxed{)}$ $\boxed{<}$ *instruction*-**index** $\boxed{>}$ •
- $\boxed{\textbf{Exit:}}$ $\boxed{\{}$ *instruction*-**index** $\boxed{\}}$ **Integer-const** •
- $\boxed{\textbf{Halt:}}$ $\boxed{<}$ *instruction*-**index** $\boxed{>}$
  [$\boxed{\{}$ { *task-call*-**index** $\boxed{,}$ … } $\boxed{\}}$]**Integer-const** •
- $\boxed{\textbf{Watchdog:}}$ $\boxed{\{}$ *instruction*-**index** $\boxed{\}}$ $\boxed{<}$ *instruction*-**index** $\boxed{>}$ •
- $\boxed{\textbf{Stay:}}$ $\boxed{<}$ *instruction*-**index** $\boxed{>}$ •
- $\boxed{\textbf{Run:}}$ *module*-**identifier** $\boxed{[}$ **IC-RENAMINGS** $\boxed{]}$
  $\boxed{<}$ *instruction*-**index** $\boxed{>}$ $\boxed{(}$ *instruction*-**index** $\boxed{)}$ •
- $\boxed{\textbf{Return:}}$ **Integer-const** •

`Action:` [a] (c)

executes the action a and passes control to the instruction with index c.

The integer a must be the index of an pure action in the action table (cf. Section 3.3, page 40). The integer c must be the index of a control instruction in the instruction table.

`Test:` [a] (c_1,c_2)

executes the test action with index a and passes control to the instruction with index c_1 if a returns true or to the instruction with index c_2 if a returns false.

The integer a must be the index of a test action in the action table (cf. Section 3.3, page 40). The integers c_1 and c_2 must be the indices of control instructions in the instruction table.

`Reset:` [s] (c)

declares that the signal with index s must have its state reset to *not emitted* and that the value must be reset to *undefined* if it is valued; then control is passed to the instruction with index c.

The integer s is an index into the signal table. The integer c must be the index of a control instruction in the instruction table.

A `Reset` instruction is associated with each declaration of a local signal.

`Emit:` [s] (c)

emits the signal with index s and passes control to the instruction with index c.

The integer s is an index into the signal table. The integer c must be the index of a control instruction in the instruction table.

If the signal is valued, the computation of the new value is not changed by the `Emit` instruction; it must be changed by a separately generated `Action` instruction that changes the value of the variable of the signal or combines its current value in case of a multiple signal.

The emission of an output signal to the environment is not done by the `Emit` instruction. This must be done by a call to an `Output` action, and must not take place until there will be no further `Emit` instructions on this signal. In the IC code, there is no explicit instruction to call this `Output` action; its call is the responsibility of the interpreter which must, for example, call it at the end of the reaction if the signal has been emitted fewer than one times. In the OC code, the calls to the `Output` action are explicit and ought therefore to be produced by the IC-OC compiler.

`Access:` [s] (c)

manages the synchronization of signal emissions and receptions; control remains blocked so long as the emission of the referenced signal is possible; when the signal can no longer be emitted, control is passed to the instruction with index c.

The integer s is an index into the signal table. The integer c must be the index of a control instruction in the instrution table.

`Goto:` (c)

passes control to the instruction with index c.

The integer c must be the index of a control instruction in the instruction table.

A `Goto` instruction replaces the `Return` instruction in an instantiated module (cf. Section 4.3, page 52).

`Goloop:` (c)

indicates the end of a loop body and passes control to the instrution with index c; this instruction allows the detection of intantaneous loops.

The integer c must be the index of a control instruction in the instruction table.

`Fork:` $\{p\}$ $(c_1, c_2, \ldots, c_n)$

instantaneously distributes control to the n instructions with index $c_1, c_2, \ldots, c_n$; the execution of these instructions is synchronized by the `Parallel` with index p.

The integer p must be the index of a `Parallel` instruction in the instruction table. The value of n must be positive and the integers $c_1, c_2, \ldots, c_n$ must reference control instructions in the table of instructions.

`Parallel:` $(c_0, c_2, \ldots, c_n)$ $< r >$

synchronizes, at each instant, the execution of the branches started by the associated `Fork`.

The integer r must be the index of a reconstruction instruction such as `Parallel`, `Watchdog`, `Stay` or `Return` in the instruction table. It must be the case that either $n = 0$, or $n \geq 2$, and the integers $c_0, c_2, \ldots, c_n$ forming the *continuation list* must be control instructions in the instruction table (warning: there is no index 1 as is explained presently). The index of the `Parallel` instruction must be referenced by only one `Fork`.

At each instant, each branch that is still active sends a *termination code* by the execution of a `Halt` or `Exit` instruction; the `Halt` instruction sends the code 1 and an instruction of the form "`Exit` $\{p\}$ l" sends the code l. The `Parallel` instruction computes the maximum m of the termination codes of the active branches. In the list of continuations for a `Parallel`, the first continuation corresponds to the level 0, the second to level 2, the third to level 3 and so on. The level 1 being handled with the aid of the reconstruction tree and therefore through the field $< r >$. This is the reason that there is no instruction c_1 in the continuation list.

- If $m = 0$, all the branches are terminated. In this case, the `Parallel` is terminated also and passes control to the instruction with index c_0.
- If $m = 1$, some branches may have finished, at least one branch has executed a `Halt` instruction, and no branch has executed an `Exit` instruction with $l > 1$. In this case, the `Parallel` instruction is not terminated yet, and it sends the termination code 1 to its possible parent parallel in the reconstruction tree; it behaves exactly like a `Halt`.

- If $m > 1$, one of the branches has has asked to exit the parallel immediately. All the branches are therefore killed.
  - If $m \leq n$, the exit is handled locally and passes control to the instruction with index c_m.
  - If $m > n$, the exit is not handled locally. The code is sent to the exit level $m-n+1$ to the parent `Parallel` or `Return` in the reconstruction tree, the difference proving the fact that the level 1 does not correspond to any explicit continuation.

`Exit:` $\{p\}$ l

signals to a parallel with index p a termination at level l.

The integer p must be the index of a `Parallel` instruction. The integer l must be different than 1.

`Halt:` $<r>$ i `or Halt:` $<r>$ $\{e\}$ i

blocks control and sends the termination code 1 to its parent `Parallel` or `Return` in the reconstruction tree.

The integer r must be the index of a reconstruction instruction such as `Parallel`, `Watchdog`, `Stay` or `Return` in the instruction table. If it is present, the integer 4 must be the index of an `exec` in the exec table. *An integer e must not appear more than one time in the instruction table.*

The integer $i \geq 0$ is called the *rank* of the `Halt`. The rank of all `Halt` and `Return` instructions must be unique and be numbered from the set $\{0, 1, ..., n\}$ if there are n such instructions in the table. The field *halt-rank* is a unique number identifying the `Halt` used to encode the states in the form of a set of integers implemented by bit strings (a state is simply a set of `Halt` instructions). In the case of a `Halt` that has an `exec`, the number of the `exec` allows the management of external tasks referenced by the `exec` as explained in Section 3.2.2.

`Watchdog:` $\{p\}$ $<r>$

controls the execution of the part of the tree that it dominates.

The integer p must be the index of a control instruction in the instruction table. The integer r must be the index of a reconstruction instruction such as `Parallel`, `Watchdog`, `Stay` or `Return` in the reconstruction table.

`Stay:` $<r>$

freezes in all the `Halt` instructions that it dominates in the reconstruction tree in their current state.

The integer r must be the index of a reconstruction instruction such as `Parallel`, `Watchdog`, `Stay` or `Return` in the instruction table.

`Run:` *module-name* `[`*renamings*`]` $< r >$ `(`c`)`

executes the module *module-name*; if the module terminates, then control is passed to the instruction with index c.

The *renaming* field gives the following information in order :

- a list of renaming of types separated by commas and followed by a '`;`.'
- a list of renamings of constants separated by commands and followed by a '`;`.'
- a list of renamings of functions separated by commas and followed by a '`;`.'
- a list of renamings of procedures separated by commas and followed by a '`;`.'
- a list of renamings of tasks separated by commas and followed by a '`;`.'
- a list of renaming of signals separated by commas and followed by '`;`.'
- the index of the last signal declared at the level of the instruction expanding the module's source.

Each renaming is of the following form :

$$object - name/object - index$$

where the *object-name* is the name of the objects in the instantiated module. This object is renamed as the object referenced by *object-index* in the current module.

♠ **IC-RENAMINGS** ::= • **IC-TYPE-RENAMINGS** $\boxed{;}$
IC-CONSTANT-RENAMINGS $\boxed{;}$
IC-FUNCTION-RENAMINGS $\boxed{;}$
IC-PROCEDURE-RENAMINGS $\boxed{;}$
IC-TASK-RENAMINGS $\boxed{;}$
IC-SIGNALS-RENAMINGS $\boxed{;}$
last-signal-**index** •

♠ **IC-TYPE-RENAMINGS** ::=

• { **IC-TYPE-RENAMING** $\boxed{,}$ ... } •

♠ **IC-CONSTANT-RENAMINGS** ::=
• { **IC-CONSTANT-RENAMING** $\boxed{,}$ ... } •

♠ **IC-FUNCTION-RENAMINGS** ::=
• { **IC-FUNCTION-RENAMING** $\boxed{,}$ ... } •

♠ **IC-PROCEDURE-RENAMINGS** ::=
• { **IC-PROCEDURE-RENAMING** $\boxed{,}$ ... } •

♠ **IC-TASK-RENAMINGS** ::=
• { **IC-TASK-RENAMING** $\boxed{,}$ ... } •

♠ **IC-SIGNAL-RENAMINGS** ::=
• { **IC-SIGNAL-RENAMING** $\boxed{,}$ ... } •

♠ **IC-TYPE-RENAMING** ::= • *type*-**identifier** $\boxed{/}$ *type*-**index** •

♠ **IC–CONSTANT-RENAMING** ::=
• *constant*-**identifier** $\boxed{/}$ *constant*-**index** •

♠ **IC-FUNCTION-RENAMING** ::=
• *function*-**identifier** $\boxed{/}$ *function*-**index** •

♠ **IC-PROCEDURE-RENAMING** ::=
• *procedure*-**identifier** $\boxed{/}$ *procedure*-**index** •

♠ **IC-TASK-RENAMING** ::= • *task*-**identifier** $\boxed{/}$ *task*-**index** •

♠ **IC-SIGNAL-RENAMING** ::= • *signal*-**identifier** $\boxed{/}$ *signal*-**index** •

`Return:` i

indicates the global terimination of the module.

The integer i, called the *rank*, must be non-negative.

The `Return` has a rank just like a `Halt` because it appears in the state encoding. Intuitively, the `Return` can thought of in two ways :

- When one compiles a full module, the `Return` instruction is considered the same as a `Halt` instruction.

- When a module is instantiated in another module, the `Return` is replaced by a `Goto` instruction pointing to the continuation which was indicated in the instantiating `Run` instruction.

4.4 Correctness of IC Programs

4.4.1 The Reconstruction Tree

All the reconstruction instructions, save for `Return` have a reconstruction field $< r >$. The rules governing the correctness of reconstruction are as follows :

> *There must be one and only one* `Return` *instruction in the table. The reconstruction field* $< r >$ *of all reconstruction instructions except* `Return` *must point to another reconstruction instruction. If one builds a graph whose nodes are the indices of reconstruction instructions and having an arc between* i *and* j *if the reconstruction field of* j *is* $< i >$*, so this graph must be a tree having for the root, the index of the* `Return` *instruction, and for nodes, the indices of all the* `Watchdog`*,* `Parallel`*, and* `Stay` *instructions, and for leaves, the indices of all the* `Halt` *instructions.*

This defines the *reconstruction tree* of an IC program.

4.4.2 Correctness of Exit Levels

In an instruction of the form "`Exit` $\{p\}$ l,", the integer p must be the index of a `Parallel` instruction, and the integer l, which represents the exit level, must be different from 1 ; recall that the exit level 1 is not used because it corresponds to the implicit exit level of the `Halt` instruction.

The exit level cannot be arbitrary because it must correspond to the continuation of the enclosing parallel. As explained in the decription of the `Parallel` instruction, this continuation can be directly handled by the parallel with index p, or it can be handled by a parent parallel of p in the reconstruction tree. The n must be the number of continuations of the parallel with index p. The exit level l is correct if it it is the case that $l \leq n$ (in this case the exit is handled directly by the parallel p). If $l > n$, it must be the case that p' is the index of the first parallel encountered when building the reconstruction tree above p, called the parent parallel of p. So the exit level l is correct for the `Parallel` with index p if and only if the exit level $l - n + 1$ is correct for the `Parallel` of index p'.

4.4.3 Correctness of Continuations

The continuation fields of IC instructions can not be arbitrary. In particular, in one branch of a parallel, it is not legal for a continuation to reference an instruction that is used in another of its branches. The correctness condition for continuations in the IC instruction table is described in this section. The following describes the procedure constructing the graph formed by the instructions, using the continuation fields c and the reconstruction fields r. The procedure marks each reachable instruction and appends a marking of the form $M = (p, b)$ where p is the index of a `Parallel` or `Return`, and b is the number of the parallel's branch or 0 for the `Return` instruction.

At the start, no instruction is marked. $Mark(i, M)$ where i is the index of the current instruction being exlored and M is its mark. This procedure finishes normally if the IC code is correct, otherwise it declares the IC code incorrect and stops immediately. The global correctness is tested by calculating $Mark(s, (r, 0))$ where s is the index given in the `startpoint:` field. at the beginning of the table of instructions and where r is the index of the `Return` instruction.

The $Mark(i, M)$ is computed as follows :

- if i is a unary control instruction of the form `Action`, `Reset`, `Emit`, `Access`, `Goto` or `Run` having a continuation c, there are three cases :
  - If the instruction i is not yet marked, it is marked with M, $Mark(c, M)$, is called, and then control is returned to the caller.
  - If the instruction is i is already marked with the mark M, control is returned to the caller directly.
  - If the instruction i is already marked with a mark different than M then the IC code is incorrect.
- If i is a `Goloop` instruction with continuation c, there are three cases :
  - If the instruction i is not yet marked, it is marked with M. There are two subcases :
    * if c is not already marked with M, the IC code is incorrect : a `Goloop` can not loop back on instructions that have already been explored and marked ;
    * if c is already marked with M, control is returned to the caller.
  - If the instruction i is already marked with M, control is returned to the caller directly.
  - If the instruction i is already marked with a mark different than M, the IC code is declared to be incorrect.
- If i is a `Test` instruction with continuations c_1 and c_2, there are three cases :
  - If the instruction is not yet mrked, it is marked with M, $Mark(c_1, M)$ and $Mark(c_2, M)$ are called, then control is returned to the caller.

- If the instruction is already marked with M, control is returned to the caller directly.
- If the instruction is already marked with a mark different than M, the IC code is declared to be incorrect.

- If i is an `Exit` instruction with arguments p' and l, there are two cases :
  - If the mark M has the form $M = (p, b)$ with $p' \neq p$, the IC code is declared to be incorrect. Intuitively, the parallel mentioned in the `Exit` is not the one reached by the execution of IC.
  - Otherwise, the instruction is marked with M and control is returned to the caller (independent of whether the instruction has already been marked or not).

  Note that the number of the branch was not tested for `Exit` instructions. Then `Exit` is sharable between branches of a `Parallel`, and it is the only instruction having this property

- If i is a `Fork` instruction associated with a parallel p' and with continuations $c_1, c_2, \ldots, c_n$, there are three cases :
  - if the instruction is not yet marked, it is marked with M, and $Mark(c_1, (p', 1))$, $Mark(c_2, (p', 2))$, …, $Mark(c_n, (p', n))$ are called in order to verify the body of the new `Fork`-`Parallel` pair, then $Mark(p', M)$ is called to verify the continuations of the parallel p', and finally control is returned to the caller.
  - If the instruction is already marked with M, control is returned to the caller directly.
  - If the instructin is already marked with a mark different than M the IC code is declared to be incorrect.

- If i is a `Parallel` instruction with continuations c_0,c_2,…,c_n and reconstruction index r and if the mark M is $M = (p, b)$, there are two cases because there are two ways to receive control for *Mark* : by the unique `Fork` associated with the parallel, this can't be produced but once, or from the child instructions in the reconstruction tree at the time of returning to that tree by `Halt` instructions (see the instruction furthest away) :
  - If $p \neq i$, and if p is not the parent of i in the reconstruction tree, then declare the IC code incorrect. If p is the parent of i, this is the one called from the `Fork`. In this case, call $Mark(c_0, M)$, $Mark(c_1, M), \ldots, Mark(c_n, M)$, and finally $Mark(r, M)$. There is no need to mark the instruction itself with M since this call will not happen another time.

  - If p is exactly the index of a `Parallel`, return control to the caller directly. This stops the recursion in the reconstruction tree.

- If i is a `Halt` instruction with arguments r and j or a `Stay` instruction with arguments r, there are three cases :

  - If the instruction is not already marked, then mark it with M, call $Mark(r, M)$, then return control to the caller. Note that the call $Mark(r, M)$ implies going back up the reconstruction tree initialized by the `Halt` instructions and going up the to the first parent `Parallel` which must just be the one mentioned in M.
  - If the instruction is marked with M return control to the caller directly.
  - If the instruction is already marked with a mark different than M the IC code is declared to be incorrect.

- If i is a `Watchdog` instruction with arguments p and r there are three cases :

  - If the instruction is not marked then mark it with M and call $Mark(p, M)$ and then $Mark(r, M)$, and return control to the caller. Note that the call $Mark(r, M)$ causes a return up the reconstruction tree.
  - If the instruction is marked with M then return control to the caller directly
  - If the instruction is already marked with a mark different than M then declare the IC code to be incorrect.

- If i is a `Return` instruction and if the mark is $M = (p, b)$ with $p \neq i$, then declare the IC code incorrect ; otherwise return control to the caller.

4.4.4 Examples of Correctness Checking for IC Programs

Example 1

The following ESTEREL program :

```
input I1, I2, I3;
output O, O1, O2;

trap T1 in
   [
      await I1
   ||
      trap T2 in
         do
            [
```

```
               await I1; exit T2
            ||
               await I2; exit T1
            ]
         watching I3
      handle T2 do
         emit O2
      end trap;
   ]
handle T1 do
   emit O1
end trap;
emit O
```

is translated into IC in the following fashion :

```
...
signals: 8
0: input: I1 $0 single: bool: 0 ;
1: input: I2 $0 single: bool: 1 ;
2: input: I3 $0 single: bool: 2 ;
3: output: O $0 single: ;
4: output: O1 $0 single: ;
5: ouptut: O2 $0 single: ;
6: local: $0 single: ;
7: local: $0 single: ;
end:
...
statements: 27 startpoint: 1 ;
goloops: 0 ;
exitlevels: 3 ;
0: Return: 0 ;
1: Reset: [6] (2) ;
2: Fork: {3} (4, 8) ;
3: Parallel: (26, 25) <0> ;
4: Halt: <5> 1 ;
5: Watchdog: {6} <3> ;
6: Test: [0] (7,5) ;
7: Exit: {3} 0 ;
8: Reset: [7] (9) ;
9: Fork: {10} (11, 17) ;
10: Parallel: (7, 24) <22> ;
11: Halt: <12> 2 ;
12: Watchdog: {13} <10> ;
13: Test: [0] (14,12) ;
14: Emit: [7] (15) ;
15: Exit: {10} 2 ;
16: Exit: {10} 0 ;
17: Halt: <18> 3 ;
18: Watchdog: {19} <10> ;
19: Test: [1] (20,18) ;
20: Emit: [6] (21) ;
21: Exit: {10} 3 ;
22: Watchdog: {23} <3> ;
23: Test: [2] (7,22) ;
24: Emit: [5] (7) ;
25: Emit: [4] (26) ;
26: Emit: [3] (0) ;
end:
```

The correctness of this IC program will be verified using the preceding correctness criteria.

Correctness of Auxiliary Values The information used for the verification are the following :

```
startpoint: 1 ;
goloops: 0 ;
exitlevels: 3 ;
...
1: Reset: [6] (2) ;
7: Exit: {3} 0 ;
15: Exit: {10} 2 ;
16: Exit: {10} 0 ;
21: Exit: {10} 3 ;
```

The `startpoint` identifies a `Reset` control instruction.

There are no `Goloop` instructions in the IC program

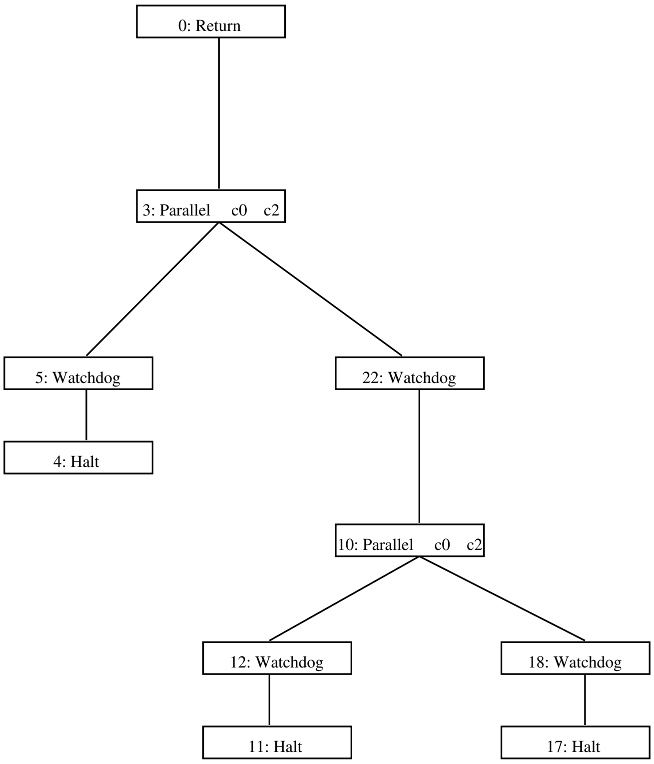
The `exitlevels` field indicates that there are maximum of 3 levels of termination for the 4 `Exit` instructions in the program.

Correctness of the Reconstruction Tree The instructions used are :

```
0: Return: 0 ;
3: Parallel: (26, 25) <0> ;
5: Watchdog: {6} <3> ;
10: Parallel: (7, 24) <22> ;
11: Halt: <12> 2 ;
12: Watchdog: {13} <10> ;
17: Halt: <18> 3 ;
18: Watchdog: {19} <10> ;
22: Watchdog: {23} <3> ;
```

There is one and only one `Return` instruction.

The graph constructed from the reconstruction information is given by the following figure :



is a tree having for the root index a `Return` instruction, for nodes the indices of all the `Watchdog`, `Parallel`, and `Stay` instructions, and or leaves the indices of all the `Halt` instructions.

Correctness of `Exit` Levels The only information used to verify the correctness here is the following :

```
3: Parallel: (26, 25) <0> ;
7: Exit: {3} 0 ;
10: Parallel: (7, 24) <22> ;
15: Exit: {10} 2 ;
16: Exit: {10} 0 ;
21: Exit: {10} 3 ;
22: Watchdog: {23} <3> ;
```

One can see that the `Exit` levels (`0, 2, 0, 3`) are different than 1.

Next the `Exit` make reference to two instructions (3 and 10) which are in the same `Parallel`.

Finally, it can be seen that for each `Exit`, its termination level is handled by a parent `Parallel`.

- `7: Exit: {3} 0 ;`

  The `Parallel` 3 handles the termination level 0 with the continuation 26. 26.

- `15: Exit:` $\{$`10`$\}$ `2 ;`

  The `Parallel` 10 is handles the termination level 3 with the continuation 24.

- `16: Exit:` $\{$`10`$\}$ `0 ;`

  The `Parallel` 10 handles the termination level 0 with the continuation 7.

- `21: Exit:` $\{$`10`$\}$ `3 ;`

  The `Parallel` 10 does not handle the termination level 4. It transmits therefore to its parent `Parallel` the termination level $3 - 2 + 1 = 2$. The parent `Parallel` of `Parallel` 10 is found above `Watchdog` 22 and at last on `Parallel` 3. `Parallel`3 handles the termination level 2 with the continuation 25.

The `Exit` levels are correct.

Correctness of Continuations The function *Mark* is called. In the table that follows, successive calls to *Mark* are separated by a double line and are done in parallel.

Instructions	M	Calls
startpoint: 1		$Mark(1, (0, 0))$
0: Return: 0		
startpoint: 1		
0: Return: 0		
1: Reset: [6] (2)	(0, 0)	$Mark(2, (0, 0))$
0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	$Mark(4, (3, 1))$ $Mark(8, (3, 2))$
		$Mark(3, (0, 0))$
0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	0 is the parent of 3
		$Mark(26, (0, 0))$ $Mark(25, (0, 0))$
4: Halt: <5> 1	(3, 1)	$Mark(5, (3, 1))$
8: Reset: [7] (9)	(3, 2)	$Mark(9, (3, 2))$
0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: {6} <3>	(3, 1)	$Mark(6, (3, 1))$ $Mark(3, (3, 1))$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	$Mark(11, (10, 1))$ $Mark(17, (10, 2))$
		$Mark(10, (3, 2))$
25: Emit: [4] (26)	(0, 0)	$Mark(26, (0, 0))$
26: Emit: [3] (0)	(0, 0)	$Mark(0, (0, 0))$

0: Return: 0		Ok : $0 = 0$
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	Ok : $M = (3, 1)$
4: Halt: <5> 1	(3, 1)	
5: Watchdog: {6} <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	$Mark(7, (3, 1))$ $Mark(5, (3, 1))$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	3 is the parent of 10
		$Mark(7, (3, 2))$ $Mark(24, (3, 2))$
		$Mark(22, (3, 2))$
11: Halt: <12> 2	(10, 1)	$Mark(12, (10, 1))$
17: Halt: <18> 3	(10, 2)	$Mark(18, (10, 2))$
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	Ok : $M = (0, 0)$

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: {6} <3>	(3, 1)	Ok : $M = (3, 1)$
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	Ok : $M = (3, 1)$, $3 = 3$
		Ok : $M = (3, 2)$, $3 = 3$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	
11: Halt: <12> 2	(10, 1)	
12: Watchdog: {13} <10>	(10, 1)	$Mark(13, (10, 1))$ $Mark(10, (10, 1))$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: {19} <10>	(10, 2)	$Mark(19, (10, 2))$ $Mark(10, (10, 2))$
22: Watchdog: {23} <3>	(3, 2)	$Mark(23, (3, 2))$ $Mark(3, (3, 2))$
24: Emit: [5] (7)	(3, 2)	$Mark(7, (3, 2))$
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	Ok : $M = (3, 2)$, $3 = 3$
4: Halt: <5> 1	(3, 1)	
5: Watchdog: {6} <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	Ok : $M = (3, 2)$, $3 = 3$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	Ok : $M = (10, 1)$, $10 = 10$
		Ok : $M = (10, 2)$, $10 = 10$
11: Halt: <12> 2	(10, 1)	
12: Watchdog: {13} <10>	(10, 1)	
13: Test: [0] (14,12)	(10, 1)	$\mathit{Mark}(14, (10, 1))$ $\mathit{Mark}(12, (10, 1))$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: {19} <10>	(10, 2)	
19: Test: [1] (20,18)	(10, 2)	$\mathit{Mark}(20, (10, 2))$ $\mathit{Mark}(18, (10, 2))$
22: Watchdog: {23} <3>	(3, 2)	
23: Test: [2] (7,22)	(3, 2)	$\mathit{Mark}(7, (3, 2))$ $\mathit{Mark}(22, (3, 2))$
24: Emit: [5] (7)	(3, 2)	
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: {6} <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	Ok : $M = (3, 2)$, $3 = 3$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	
11: Halt: <12> 2	(10, 1)	
12: Watchdog: {13} <10>	(10, 1)	Ok : $M = (10, 1)$
13: Test: [0] (14,12)	(10, 1)	
14: Emit: [7] (15)	(10, 1)	$\mathit{Mark}(15, (10, 1))$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: {19} <10>	(10, 2)	Ok : $M = (10, 2)$
19: Test: [1] (20,18)	(10, 2)	
20: Emit: [6] (21)	(10, 2)	$\mathit{Mark}(21, (10, 2))$
22: Watchdog: {23} <3>	(3, 2)	Ok : $M = (3, 2)$
23: Test: [2] (7,22)	(3, 2)	
24: Emit: [5] (7)	(3, 2)	
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: {6} <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	
11: Halt: <12> 2	(10, 1)	
12: Watchdog: {13} <10>	(10, 1)	
13: Test: [0] (14,12)	(10, 1)	
14: Emit: [7] (15)	(10, 1)	
15: Exit: {10} 2	(10, 1)	$\mathrm{Ok} : 10 = 10$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: {19} <10>	(10, 2)	
19: Test: [1] (20,18)	(10, 2)	
20: Emit: [6] (21)	(10, 2)	
21: Exit: {10} 3	(10, 2)	$\mathrm{Ok} : 10 = 10$
22: Watchdog: {23} <3>	(3, 2)	
23: Test: [2] (7,22)	(3, 2)	
24: Emit: [5] (7)	(3, 2)	
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

The IC code is correct. Note that instruction 16 does not appear in the last table as it is not reachable.

Example 2

The correctness of the following IC code will be verified :

```
...
signals: 2
0: output: O1 $0 single: ;
1: output: O2 $0 single: ;
end:
...
statements: 4 startpoint: 1 ;
goloops: 0 ;
exitlevels: 0 ;
0: Return: 0 ;
1: Fork: {2} (3, 3) ;
2: Parallel: (0) <0> ;
3: Emit: [0] (4) ;
4: Exit: {2} 0 ;
end:
```

Instructions	M	Calls
`startpoint: 1` `0: Return: 0`		$Mark(1,(0,0))$
`0: Return: 0` `1: Fork: {2} (3, 3)`	 (0, 0)	 $Mark(3,(2,1))$, $Mark(3,(2,2))$

`0: Return: 0`
`1: Fork: {2} (3, 3)`
`3: Emit: [0] (4)`
	
	(0, 0)
	(2, 1)
		
		Incorrect : $(2,2) \neq (2,1)$

The same instruction is executed by two branches of the same `Parallel` : the code is rejected.

Example 3

In the IC code that follows, the `Exit` instruction 5 is not close to the required `Parallel`.

```
signals: 3
0: output: O1 $0 single: ;
1: output: O2 $0 single: ;
2: local: $0 single: ;
end:
...
statements: 13 startpoint: 1 ;
```

```
goloops: 0 ;
exitlevels: 2 ;
0: Return: 0 ;
1: Reset: [2] (2) ;
2: Fork: {3} (4, 6) ;
3: Parallel: (8, 0) <0> ;
4: Emit: [2] (5) ;
5: Exit: {9} 2 ;
6: Emit: [0] (7) ;
7: Exit: {3} 0 ;
8: Fork: {9} (10, 11) ;
9: Parallel: (0, 0) <0> ;
10: Emit: [0] (12) ;
11: Emit: [1] (12) ;
12: Exit: {9} 0 ;
end:
```

Instructions	M	Calls
`startpoint: 1` `0: Return: 0`		$Mark(1,(0,0))$
`0: Return: 0` `1: Reset: [2] (2)`	 (0, 0)	 $Mark(2,(0,0))$

`0: Return: 0`
`1: Reset: [2] (2)`
`2: Fork: {3} (4, 6)`
	
	(0, 0)
	(0, 0)
		
		$Mark(4,(3,1))$ $Mark(6,(3,2))$

`0: Return: 0`
`1: Reset: [2] (2)`
`2: Fork: {3} (4, 6)`
`4: Emit: [2] (5)`
`6: Emit: [0] (7)`
	
	(0, 0)
	(0, 0)
	(3, 1)
	(3, 2)
		
		$Mark(5,(3,1))$
		$Mark(7,(3,2))$

`0: Return: 0`
`1: Reset: [2] (2)`
`2: Fork: {3} (4, 6)`
`4: Emit: [2] (5)`
`5: Exit: {9} 2`
`6: Emit: [0] (7)`
`7: Exit: {3} 0`
	
	(0, 0)
	(0, 0)
	(3, 1)
	
	(3, 2)
		
		Incorrect : $3 \neq 9$

Chapter 5

The Data-Flow Graph Format : GC

5.1 Introduction

GC is a hierarchical representation in the data flow style or synchronous program block diagrams, augmented with explicit control oriented especially at handling program data translated from IC. The declarative part of a GC code has a synchronous semantics which is a *specification* in the sense of a semantic model described in Section 5.7. The notions of clock and flows used following in the sequel make reference to this model. The complete semantics of GC is presented in Appendix D.

5.1.1 Structure of a GC Code

A GC code is made up of a list of declarations of three kinds of entities : the *data blocks*, the *interfaces* and the *nodes*.

5.1.2 Nodes

The low-level entity of a GC code is called a *node*. A node described in the GC format is given by an interfaces, some data and a body.

The interface of a node describes the properties which are visible to the outside.

The data describes, in the form of data block imports, the types, constants, functions and procedures used by the node.

The body of a node is made up of data composed of the following, described in tables specific to a node :

Definition of Values of Streams : The definition expressions express the computation of values at the exit flows in terms of functions of entry flows. They are *deterministic input/output functions*, they obey the principle of substitution (their lefthand side can be substituted for the righthand side in all cases).

The definition expressions are made up of equations, node instantiations and calls to procedures and activations of other external nodes which can be actions in imperative modules for example.

A set of definition expressions can be seen as a data flow graph.

It is possible to add to this graph some explicit control dependencies between the definitions.

Synchronisation : Synchronizations specify *constraints* governing the semantics of the program. They are relations and not functions. They must be taken into account by the set of tools such as provers or for code generators.

Assertions : Assertions are properties expressed on the flow nodes under consideration (they are therefore relations and not functions). Assertions can be used as *comments* and ignored by formal tools or code generators :they serve also to describe the hypothetical behavior of the environment. These hypotheses must eventually be verified under operating conditions. They can also be used at the end of the tool chain by code optimizers.

Dependancies : The dependancies allow the specification of a partial execution order in the same instant. They are specifically necessary for the specification of interfaces. Dependancies can change from one instant to the other and this is encoded in the label which is a *clock* assigned to all dependancies : the dependancy under consideration is in force when the flows indicated by the dependence are active under the clock; during instants of its presence.

Certain dependencies are implicit (for example the outputs of most of the operators are implicitly dependant on their inputs), but can be made explicit during compilation.

5.1.3 Interface of a Node or External Node

The interface of a node is the summary required for use of the node as a "*black box*" in its instantiation context. The interface of a node is made up of :

a data part which describes specifically in the form of importation of data blocks, the types of flows entering and exiting the node;

dependencies which permit the specification of a partial communication order among the interface flows in the interior within the same instant;

assertions and synchronizations which are the properties expressed on the interface flows of the node under consideration.

An external node is a node where the GC description is not accessible. In particular, this can be the case from the perspective of GC if the action is an imperative module. Like all nodes, an external node is matched by an interface that describes the properties of it that are visible to the outside.

5.2 Organisation of GC Data

5.2.1 Flows

A table of flows is identified by the keyword **flows:**. Recall that the general syntax of tables was described in Section 2.2.3.

♠ **Flow-table-name** ::= • **flows:** •

An entry in this table contains the kind of flow (input **i:**, output **o:**, or implicitly a local flow), the name of the flow, its type, possibly the initial value and its clock. The clock is a flow represented by a flow index which must be of type `$pure`.

♠ **STREAM-OBJECT** ::= • **STREAM-KIND STREAM-CHANNEL Clock** •

♠ **STREAM-KIND** ::= • **i:** *flow*-**identifier** • [**o:**] *flow*-**identifier** •

♠ **STREAM-CHANNEL** ::= • *type*-**index** [**INITIAL-STREAM-VALUE**] •

♠ **INITIAL-STREAM-VALUE** ::= • **value:** **EXPRESSION** •

♠ **Clock** ::= • *flow*-**index** •

Associated with each node, there is a predefined flow at least as fast as all the other flows visible in the node called `$base` and written `$g0`, of type `$pure`. This flow is its own clock.

the flows of a node are :

- flows of the interface and in this case are described in the table of flows of the interface ; these flows must have an attribute indicating the kind **i:** or **o:** ;
- local flows, which are also described in the table of flows of the node ; these local flows have no flow kind attribute

5.2.2 Expressions Specific to GC

Expressions specific to the GC format are flows, which complete thus the list of possible expressions defined in Section 2.2.5.

A flow is referenced simply by its index.

♠ **GC-Term** ::= • *flot*-**index** •

In a node (cf. Section 5.4, page 81) :

- the indices of flows appearing in an expression can be local indices designating external or local flows of the node, or compound indices designating external flows of imported nodes ;
- The indices of constants appearing in an expression can be indices of predefined common constants, or compound indices designating the constants of imported data blocks ;
- the indices of functions appearing in an expression can be the indices of predefined common functions, or compound indices designating functions of imported or predefined data blocks.

5.2.3 Types Specific to GC

New Types

For each type μ different from the type `$pure`, and different from the type ν (for all types ν), a type `$win`(μ) can be defined. The grammar given in Section 2.4.1 is completed with the following rule :

♠ **TYPE-OBJET** ::= • $\boxed{\texttt{\$win}}$ $\boxed{(}$ *type*-**index** $\boxed{)}$ •

The associated operators are defined in the GC predefined data block.

Bounded integers are of type integer and bound to a specific constraint by an assertion.

Form of the Presentation

For this presentation, a description of data blocks is given, then one for each of the operators. For each operator the following is given :

- the clock equations are implicitly understood (cf. Section 5.7, page 92) ;
- the dependencies are implicitly understood **when they are used in a definition** ; these dependencies have no counterpart in the semantics of

continuations. The dependencies are indications for the order of evaluation in the course of a reaction (therefore *in an instant*) : note that in `X -h-> Y` the fact that in all reactions where

- the clock of the flow `X` is present,
- the clock of the flow `Y` is present,
- the clock `h` is present (it is described implicitly by the clock `$base` when it is not mentioned).

the evaluation of `Y` can not precede (inside an instant!) the evaluation of `X`. For notational variants concerning the dependencies, see Section **??**.

- its semantics of continuations (cf. Section 5.7, page 92).

Predefined Data Blocks in GC

The predefined GC data blocks contain specifically the definition of the generic types `$any` and `$win($g0)`. The functions defined for these types are overloaded : for all types μ, all functions defined for the type μ (each of the references to the type `$any` in the definition of the function must be replaced by a reference to the same type μ). In the same fashion, for all types μ for which there exists a type `$win(`μ`)`, all functions defined for the types `$any` and `$win($g0)` are impicitly defined for the types μ and `$win(`μ`)`.

```
data: dc:0 GC
  types: 7
    $g0: $any;      -- no functions associated
    $g1: $win($g0); -- no functions associated
    $g2: $win($1);  -- $boolean   $g15 code $eq
    $g3: $win($2);  -- $integer   $g16 code $eq
    $g4: $win($3);  -- $string    $g17 code $eq
    $g5: $win($4);  -- $float     $g18 code $eq
    $g6: $win($5);  -- $double    $g19 code $eq
  end:
  functions: 20
    -- pure
    $g0: $tt ($1): $0;                    -- polychrone
    $g1: $clkadd ($0, $0): $0;            -- polychrone
    $g2: $clkdiff ($0, $0): $0;           -- polychrone
    $g3: $clkmult ($0, $0): $0;           -- polychrone
    $g4: $present ($0): $1;
    $g5: $clkeq ($0, $0): $1;             -- polychrone
    $g6: $clkimplies ($0, $0): $1;        -- polychrone
    $g7: $clkmutex ($0, $0): $1;          -- polychrone
    -- any
```

```
    $g8: $clock ($g0): $0;
    $g9: $default ($g0, $g0): $g0;        -- polychrone
   $g10: $when ($g0, $0): $g0;            -- polychrone
   $g11: $pre ($g0): $g0;
   $g12: $fby ($g0, $g0): $g0;
   $g13: $window ($g0, $2, $g1): $g1;
   $g14: $select ($g1, $2): $g0;
    --
   $g15: $eq ($g2, $g2): $1;
   $g16: $eq ($g3, $g3): $1;
   $g17: $eq ($g4, $g4): $1;
   $g18: $eq ($g5, $g5): $1;
   $g19: $eq ($g6, $g6): $1;
end:
enddata:
```

The predefined GC data blocks are imported implicitly by all entities which use one of the objects that they define.

The references to types `$win($1)`, `$win($2)`, `$win($3)`, `$win($4)`, `$win($5)` have respectively the form "`$g2`", "`$g3`", "`$g4`", "`$g5`", "`$g6`". A reference to a function of index i in the GC data block has the form `GC`.

In the following a literal form of notation is used which is a bit more readable for the predefined objects (although this form is syntactically incorrect).

5.2.4 Functions andt Procedures

The parameters of procedures and functions are flows. The result of a function is also a flow. If one calls the input flows $\boxed{\textbf{i:}}$ of a procedure and the parameters of a function, output flows $\boxed{\textbf{o:}}$ of a procedure and the result of a function, the inputs and the outputs of each call are synchronous (have the same clock) ; whats more, all inputs implicitly precede all outputs with each call.

Warning, the implicit relations on clocks and dependencies do not apply to predefined functions on the pure type (cf. Section 5.2.5, page 74) and to predefined generic functions (cf. Section 5.2.6, page 78). For these functions, the implicit relations on the clocks and the dependencies are specified in their definition.

5.2.5 Predefined Functions on the Pure Type

The flows of type `$pure` are also called clocks.

The function `$tt`

- Clock

$$h^{\texttt{\$tt(X)}} = \{h_n^{\texttt{X}} \mid v_n^{\texttt{X}} = true\}$$

- Dependencies
  The following implicit dependancy is generated : `X --> $tt(X)`

- Semantics
  `$tt(X)` is the flow of type `$pure` present if and only if the boolean flow `X` is present and equal to *true*.

The function `$clkadd`

- Clock

$$h^{\texttt{\$clkadd(X,Y)}} = h^{\texttt{X}} \cup h^{\texttt{Y}}$$

- Dependencies
  The following implicit dependencies are generated : `(X,Y) --> $clkadd(X,Y)`

- Semantics
  `$clkadd(X,Y)` is the flow of type `$pure` present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present.

The function `$clkdiff`

- Clock

$$h^{\texttt{\$clkdiff(X,Y)}} = h^{\texttt{X}} \ominus h^{\texttt{Y}}$$

- Dependencies
  The following implicit dependencies are generated : `(X,Y) --> $clkdiff(X,Y)`

- Semantics
  `$clkdiff(X,Y)` is the flow of type `$pure` present if and only if the flow `X` of type `$pure` is present and the flow of type `X $pure` is not present.

The function `$clkmult`

- Clock

$$h^{\texttt{\$clkmult(X,Y)}} = h^{\texttt{X}} \ \cap \ h^{\texttt{Y}}$$

- Dependencies
The following implicit dependencies are generated : `(X,Y) --> $clkmult(X,Y)`

- Semantics
`$clkmult(X,Y)` is the flow of type `$pure` present if and only if the flow `X` of type `$pure` and the flow `Y` of type `$pure` are present.

The function `$present`

- Clock
`$present(X)` et `X` have the same clock.

$$h^{\texttt{\$present(X)}} = h^{\texttt{X}}$$

- Dependencies
The following implicit dependencies are generated : `X --> $present(X)`

- Semantics
`$present(X)` is the boolean flow on the clock `X`, where the value is always equal to *true*.

$$\forall n \leq |h^{\texttt{X}}|,\ v_n^{\texttt{\$present(X)}} = true$$

The function `$clkeq`

- `$clkeq` is not a primitive function :
`$clkeq(X,Y)` is equivalent to the following expression :
`$default($present($clkmult(X,Y)),$not($present($clkadd(X,Y))))`

- Clock

$$h^{\texttt{\$clkeq(X,Y)}} = h^{\texttt{X}} \ \cup \ h^{\texttt{Y}}$$

- Dependencies
The implicit dependencies generated are deduced from the expansion of the function : `(X,Y) --> $clkeq(X,Y)`

- Semantics
  `$clkeq(X,Y)` is the boolean flow present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present ; its value is equal to *true* if and only if the flows `X` and `Y` are both present :

$$\forall n \leq |h^{\mathtt{X}} \cup h^{\mathtt{Y}}|,$$
$$v_n^{\mathtt{\$clkeq(X,Y)}} = \begin{cases} true & \text{if } h_n^{\mathtt{\$clkeq(X,Y)}} \in h^{\mathtt{X}} \cap h^{\mathtt{Y}} \\ false & \text{else} \end{cases}$$

The function `$clkimplies`

- `$clkimplies` is not a primitive function :
  `$clkimplies(X,Y)` is equivalent to the following expression :
  `$default($present(Y),$not($present(X)))`

- Clock

$$h^{\mathtt{\$clkimplies(X,Y)}} = h^{\mathtt{X}} \;\cup\; h^{\mathtt{Y}}$$

- Dependencies
  The implicit dependencies generated are deduced from the expansion of the function : `(X,Y) --> $clkimplies(X,Y)`

- Semantics
  `$clkimplies(X,Y)` is the boolean flow present if and only if the flow `X` of type `$pure` or the flow `Y` of type `$pure` is present ; its value is equal to *true* if and only if the flow `X` is present more often than the flow `Y` :

$$\forall n \leq |h^{\mathtt{X}} \cup h^{\mathtt{Y}}|,$$
$$v_n^{\mathtt{\$clkimplies(X,Y)}} = \begin{cases} true & \text{if } h_n^{\mathtt{\$clkimplies(X,Y)}} \in h^{\mathtt{Y}} \\ false & \text{else} \end{cases}$$

The function `$clkmutex`

- `$clkmutex` is not is not a primitive function :
  `$clkmutex(X,Y)` is equivalent to the following expression :
  `$not($clkeq(X,Y))`
  Clocks and dependancies are deducible from it immediately.
  `$clkmutex(X,Y)` is the boolean flow present if and only if the flow `X` of type `$pure` or the flow of type `Y` `$pure` is present ; its value is equal to *true*if and only if more than one of the flows `X` or `Y` is present.

5.2.6 Predefined Generic Functions

The function `$clock`

- Clock
  `$clock(X)` and `X` have the same clock.

  $$h^{\texttt{\$clock(X)}} = h^{\texttt{X}}$$

- Dependencies
  `$clock(X)` generates no implicit dependencies.

- Semantics
  `$clock(X)` is the flow of type `$pure` which is present if and only if the flow `X` is present.

The function `$default`

- Clock

  $$h^{\texttt{\$default(X,Y)}} = h^{\texttt{X}} \ \cup \ h^{\texttt{Y}}$$

- Dependencies
  The pure flow h must be of a clock $h^{\texttt{Y}} \ominus h^{\texttt{X}}$, the following implicit dependencies are generated :
  `X -`$h^{\texttt{X}}$`-> $default(X,Y)`
  `Y -`h`-> $default(X,Y)`

- Semantics
  `$default(X,Y)` is a flow of the same type as the flows `X` and `Y` and is present if and only if the flow `X` or the flow `Y` is present; its value is equal to the one of `X` when that is present, and to the one in `Y` in the other case :

  $$\forall n \leq |h^{\texttt{\$default(X,Y)}}|,$$

  $$v_n^{\texttt{\$default(X,Y)}} = \begin{cases} v_{m_n}^{\texttt{X}} & \text{if } h_n^{\texttt{\$default(X,Y)}} = h_{m_n}^{\texttt{X}} \in h^{\texttt{X}} \\ v_{k_n}^{\texttt{Y}} & \text{if } h_n^{\texttt{\$default(X,Y)}} = h_{k_n}^{\texttt{Y}} \in h^{\texttt{Y}} \ominus h^{\texttt{X}} \end{cases}$$

The function `$when`

- Clock

  $$h^{\texttt{\$when(X,Y)}} = h^{\texttt{X}} \ \cap \ h^{\texttt{Y}}$$

- Dependencies
  The only implicit dependancy generated is the following :
  `X -`$h^{\texttt{Y}}$`-> $when(X,Y)`

- Semantics
  `$when(X,Y)` is a flow of the same type as the flow `X`, present if and only if the flow `X` and the flow `Y` (this last must have the type `$pure`) are present ; its value is thus equal to the one in `X` :

$$\forall n \leq |h^{\texttt{\$when(X,Y)}}|,$$
$$v_n^{\texttt{\$when(X,Y)}} = v_{m_n}^{\texttt{X}} \text{ si } h_n^{\texttt{\$when(X,Y)}} = h_{m_n}^{\texttt{X}}$$

The function `$pre`

- Clock

$$h^{\texttt{\$pre(X)}} = h^{\texttt{X}}$$

- Dependencies
  `$pre(X)` does not generate implicit dependencies.

- Semantics
  `$pre(X)` is a flow equal to flow `X`, translated into the past by one index:

$$\forall n \leq |h^{\texttt{\$pre(X)}}|,$$
$$v_n^{\texttt{\$pre(X)}} = \begin{cases} \mathit{nil} & \text{if } n = 1 \\ v_{n-1}^{\texttt{X}} & \text{if } n > 1 \end{cases}$$

  or *nil* denoting the undefined value (non-initialized).

The function `$fby`

- Clock

$$h^{\texttt{\$fby(X,Y)}} = h^{\texttt{X}} = h^{\texttt{Y}}$$

- Dependencies
  The flow h must be pure where the only instant of presence is the first instant of presence of the flows `X` and `Y`, and K is a pure clock flow $h^{\texttt{X}} \ominus h$, the following implicit dependencies are generated :
  `X -`h`-> $fby(X,Y)`
  `Y -`K`-> $fby(X,Y)`

- Semantics
  `$fby(X,Y)` is a flow, of the same type as that of `X` and `Y`, except at the first instant of their common clock where the value is the one of `X` :

$$\forall n \leq |h^{\texttt{\$fby(X,Y)}}|,$$
$$v_n^{\texttt{\$fby(X,Y)}} = \begin{cases} v_1^{\texttt{X}} & \text{if } n = 1 \\ v_n^{\texttt{Y}} & \text{if } n > 1 \end{cases}$$

The function `$window`

- Clock

$$h^{\texttt{\$window(X,N,W0)}} = h^{\texttt{X}}$$

  `N` is an integer constant which is strictly positive,
  `W0` is an external vector of length equal to `N`.

- Dependencies
  The following implicit dependencies are generated :
  `X --> $window(X,N,W0)`

- Semanticsn
  The expression "`$window(X,N,W0)`", where `X` is a flow of type `T`, is a flow `W`, of type `$win(T)` ; its value is at each instant a set of `N` elements (one says that it *cuts* `N`), written respectively `W[1]`...`W[N]` ; at the initial instant of its clock, it takes on the value `W0` ; at each distinct instant after this initial instant, the following property is verifiable :

  ```
  W[N] = X      W[k-1] = pre W[k]
  ```

  Formally, using the preceding notations, this is written as :

$$\forall n \leq |h^{\texttt{\$window(X,N,W0)}}|,$$

$$v_n^{\texttt{W}}[k] = \begin{cases} \text{if } k = \texttt{N} \text{ so} & v_n^{\texttt{X}} \\ \text{if } 1 \leq k < \texttt{N} \text{ so} & \begin{cases} \texttt{W0}[k] & \text{if } n = 1 \\ v_{n-1}^{\texttt{W}}[k+1] & \text{if } n > 1 \end{cases} \end{cases}$$

The function `$select`

- Clock

$$h^{\texttt{\$select(WX,I)}} = h^{\texttt{WX}} = h^{\texttt{I}}$$

- Dependencies
  Considering `N`, the *cut* of `WX`, and h the clock of the instants of `I` where `I` is equal to `N`, the implicit dependencies are the following :
  `I --> $select(WX,I)`
  `WX -`h`-> $select(WX,I)`

- Semantics
  The expression "`$select(WX,I)`", where `WX` is a flow of type `$win(T)` and the *cut* `N`, and where `I` is a flow of strictly positive integer values at least equal to `N`, is a flow of type `T`, having at each instant the value `WX[I]`.

Formally, using the preceding notation, this is written as :

$$\forall n \leq |h^{\texttt{\$select(WX,I)}}|,$$
$$v_n^{\texttt{\$select(WX,I)}} = v_n^{\texttt{WX}}[v_n^{\texttt{I}}]$$

5.3 Interfaces

The interface of a node specifies a data part (the importation of data blocks), its input and output flows (external flows) and the dependancies among them. Finally the assertions and synchronizations relating to the external flows and using as needed the intermediate flows that appear at the end of the flows table in the interface without any kind attribute. Like the data blocks, interfaces are declared and can be common to many nodes.

An interface can import data blocks by giving it the external index of the importation table. This table can not consist of entities other than data blocks.

♠ **INTERFACE** ::= • $\boxed{\textbf{interface:}}$ **INTERFACE-INFO** [**IMPORTATION-TABLE**] **DECLARATIVE-PART** $\boxed{\textbf{endinterface:}}$ •

♠ **INTERFACE-INFO** ::= • **Gc-format** *interface*-**identifier** [$\boxed{\textbf{flat:}}$] •

♠ **Gc-format** ::= • $\boxed{\textbf{gc:}}$ **Version-number** •

♠ **DECLARATIVE-PART** ::= • [**SIGNAL-TABLE**]
[**ASSERTION-TABLE**]
[**SYNCHRONISATION-TABLE**]
[**DEPENDANCY-TABLE**]
[**PRAGMA-TABLE**] •

The absence of one of the optional tables is interpreted as the absence of the its constituent elements (not the default value).

An interface without a flow table contains no other table. This can occur with the interface of an action that posesses an implicit clock `$base` as the only clock.

5.4 Nodes

A node is a local node, described in the GC format, or is an external node which means that it has a GC description that is not accessible.

♠ **NODE** ::= • **LOCAL-NODE** •
• **EXTERNAL-NODE** •

5.4.1 Local Nodes

The declaration of a local node consists of :

- the indication of the format ($\boxed{\textbf{gc:}}$), with its version number ;
- the name of the node ;
- the optional attribute $\boxed{\textbf{flat:}}$;
- a **Computational-property** associated to the node, specified by the keywords $\boxed{\texttt{safe:}}$, $\boxed{\texttt{memorysafe:}}$ or $\boxed{\texttt{unsafe:}}$ (cf. Section 2.4.5, page 30). In the absence of these specifications, the node is presumed to be *memorysafe* ;
- the table of references to entities imported containing at least the index `0`, the external index of the interface that describes the external properties of the node (a node can import but one interface). And also of its interface, the importation table of the node contains the nodes (local or external) used by the node under consideration, in the same way that possible data blocks used by the node (in poarticular data blocks imported by the interface must be also imported by the node) ;
- a table of local flows of th node ;
- a table of assertions stated on the local flows ;
- a table of internal synchronizations ;
- a table of definitions of node flows ;
- a table of dependancies ;
- a table of pragmas.

Except for the name, the indication of the format and the table if imports, all the parts are optional.

♠ **LOCAL-NODE** ::= • $\boxed{\textbf{node:}}$ **LOCAL-NODE-INFO**
IMPORTATION-TABLE
DECLARATIVE-PART
[**DEFINITIONS-TABLE**]
$\boxed{\textbf{endnode:}}$ •

♠ **LOCAL-NODE-INFOS** ::= • **Gc-format** *node*-**identifier** [$\boxed{\textbf{flat:}}$]
[**Computational-properties**] •

5.4.2 Definitions

The table of definitions is identified by the keyword **definitions:**.

♠ **Definitions-table-name** ::= • **definitions:** •

Kinds of Definitions

One finds in the table of definitions, expressions defining the flows given directly in the form of explicit equations, and given by node instantiations ; there is equally, function calls and node activations and explicit dependancies between definitions.

♠ **DEFINITION-OBJECT** ::= • **EQUATION-PART** •
• **ACTIVATION** •
• **DEFINITIONS-DEPENDANCIES** •

♠ **EQUATION-PART** ::= • **EQUATION** •
• **INSTANTIATION** •
• **PROCEDURE-CALL** •

Explicit Equation

An equation defines the flow whose index appears on the lefthand side by the expression appearing on the righthand side. Teh flow and the expression are identical : they have the same clock, the predecessors are those of the expression, the clock dependancies are preserved.

♠ **EQUATION** ::= • **define:** *flot*-**index** **EXPRESSION** •

The index of flows appearing on the lefthand side of an equation can be given the local index designating the external or local flows of the node, given the compound index designating the external flows of imported nodes.

Example : To the index `10` of the table of definitions, the definition of the value of `X`, local flow with index `41`, by integer addition with the constant one to the value of `Y`, local flow of index `4`.

```
10: define: 41 $14(#1,4)
```

Node Instantiation

An instantiation of a node defines the exit flows of a node instantiation ; the dependancies induced are therefore the ones specified in the interface of the node. *Many syntactic occurrences of instantitation of the same node produce distinct objects.*

♠ **INSTANTIATION** ::=

• $\boxed{\textbf{set:}}$ [**Clock**] *node*-**index** $\boxed{(}$ **Renamings** $\boxed{)}$ •

The index of an instantiated node is a local index of hte importation table designating the imported node.

The clock following the $\boxed{\texttt{set:}}$ symbol is the base clock of the model produced. It must be less than or equal to the upper limit of clocks of the interface flows. In the absence of this clock, this is implicitly equal to the upper limit, but it is not explicitly and directly accessible.

Example : To the index `11` of the table of definitions, model of mode whose local rference is index tt 7 without renaming. `11: set: 7 ()`

Procedure Call

A procedure call associated with a clock (or the *activation clock*) and the flows have an occurrence of syntactic use of the procedure. Many syntactic occurrences of a call to teh same procedure make reference to the smae object, possibly with different activation clocks and with different flows : the effects to the environment be produced if the procedure is *unsafe*.

♠ **PROCEDURE-CALL** ::=

• $\boxed{\textbf{call:}}$ [**Clock**] *procedure*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •

The index of the procedure call can be given an index of a predefined common procedure, or a compound index designating a procedure in an imported data block.

The occurrence of the clock following the $\boxed{\texttt{call:}}$ symbol is the activation clock of the procedure. It must be less than or equal to the upper limit of clocks of flows in the interface. In the absence of this occurrence, the list of expressions cannot be empty, and the activation clock of the procedure is implicitly equal to the upper limit of the flows in the interface.

Example : To the index `12` of the table of definitions, call, by the clock represented by the pure flow with local index `1414`, a procedure whose reference is the index `1` in teh table respectively of `41` and `5927` (outside). `12: call: 1414 3.1 (41, 5`

Activation of a Node

An activation of a node is a call to a particular category of procedures (actions) represented by the external nodes (cf. Section 5.4.5, page 88) : that represent the GC perspective of an action of an imperative module. *On the other hand, it happens that for an instantiation, many syntactic occurrences of an activation of* *the same node make reference to the same object.* The base clock of an external node is less than or equal to the upper limit of its activation clocks.

♠ **ACTIVATION** ::=

• $\boxed{\textbf{do:}}$ [**Clock**] *node*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •

The index of a node activated is a local index in the importation table, designating an imported node.

The expressions associate the flows in a positional way to the external flows of the activated node, by respecting the order of the flows as described in **??**.

The occurrence of the clock following the $\boxed{\texttt{do:}}$ symbol is the activation clock of the node. It must be less than or equal to the upper limit of clocks of its interface flows. In the absence of this occurrence, the activation clock of the node is implicitly equal to the base clock of the node that contains the activation.

Example : To the index `13` the table of definitions, activate, to the clock represented by the pure flow with local index `1414`, an action without parameters designated (cf. Section 5.4.5, page 88) in the external node whose refernce is the index `3` in the table of local imports.

```
13: do: 1414 3 ()
```

Dependancies on Definitions

A dependancy on definition adds (possibly) to the dependancies associated to the information, the control precedences between the computations expressed in the table, and all execution will respect it.

♠ **DEPENDANCY-DEFINITIONS** ::=

• **DEFINITION-INDEX-LIST DEFINITION-INDEX-LIST** [**Clock**] •

♠ **DEFINITION-INDEX-LIST** ::= • *definition*-**index** •

• $\boxed{(}$ { *definition*-**index** $\boxed{,}$ ... } $\boxed{)}$ •

Indices of definition are local indices designating definitions in the table.

Les index de définition sont des index locaux désignant des définitions de la table.

Exemple : To the index `14` of the table of definitions, specify, to the clock represented by the pure flow with index `732` in the entity with local index `0`, the precedence of the definition of the `X` above on the activation of index `13` above.

```
14: 10 13 0.732
```

5.4.3 Renamings

The objets of data blocks (types, constants, functions and procedures) and the external flows declared in its interface can be renamed during the instantiation of a node. By deault they are not. Each renaming during instantiation of a node P has the following form :

$$\textit{categoriy: } NI_1/AI_1, \ldots, NI_n/AI_n$$

All the AI_j are indices of objects of the category under consideration, belonging to the interface or to a table cited in the node P; all the NI_j are the indices of objects of the category under consideration, appearing in the interface or in a table cited in the node P; each of the objects AI_j is renamed by NI_j.

The indices AI_j relative indices to the instantiated node : the indices of flows are simple indices designating directly the flows of the interface of the instantiated node; the indices o f types, constants, functions and procedures are compound indices designating the corresponding object in an imported data block by the instantiated node.

the indices NI_j are relative indices to the current node (in which the instantiation is found) : the indices of flows are local indices designating local flows of the curent node or compound indices naming flows of its interface; the indices of types, constants, functions and procedures are compound indices designating the corresponding object in an imported data block of the current node, or the indices of predefined objects.

For the flows, the NI_j can be expressions.

♠ **RENAMINGS** ::= • [**TYPES-RENAMINGS**]
[**CONSTANT-RENAMINGS**]
[**FUNCTION-RENAMING**]
[**PROCEDURE-RENAMING**]
STREAM-RENAMING •

♠ **TYPE-RENAMINGS** ::=

• $\boxed{\textbf{types:}}$ { *type*-**index** $\boxed{/}$ *type*-**index** $\boxed{,}$ . . . } •

♠ **CONSTANT-RENAMINGS** ::=

• $\boxed{\textbf{constants:}}$ { *constant*-**index** $\boxed{/}$ *constant*-**index** $\boxed{,}$ . . . } •

♠ **FUNCTION-RENAMINGS** ::=

• $\boxed{\textbf{functions:}}$ { *function*-**index** $\boxed{/}$ *function*-**index** $\boxed{,}$ . . . } •

♠ **PROCEDURE-RENAMINGS** ::=

• $\boxed{\textbf{procedures:}}$ { *procedure*-**index** $\boxed{/}$ *procedure*-**index** $\boxed{,}$ . . . } •

♠ **STREAM-RENAMINGS** ::=

• $\boxed{\textbf{flows:}}$ { **EXPRESSION** $\boxed{/}$ *flow*-**index** $\boxed{,}$ ... } •

5.4.4 Flow Dependancies

The implicit dependancies induced by the operators have been defined at the same time as those here.

The implicit dependancies following apply equally :

- all flows depend implicitly upon their clock ;
- all flows `X` dependant on a flow `Y` to a clock h, depend implicitly on the clock h.

Finally, it is possible to define the explicit dependancies between flows by means of the table of dependancies. In particular, these explicit dependancies are used in teh specification of interfaces. The table of dependancies is identified by the keyword $\boxed{\textbf{dependances:}}$

♠ **Dependancies-table-name** ::= • $\boxed{\textbf{dependences:}}$ •

♠ **DEPENDANCY-OBJECT** ::=

• **STREAM-INDEX-LIST STREAM-INDEX-LIST** [**Clock**] •

♠ **STREAM-INDEX-LIST** ::= • *flow*-**index** •
• $\boxed{(}$ { *flow*-**index** $\boxed{,}$ ... } $\boxed{)}$ •

The semantics of `X Y h` is that at each instant for which :

- the flow `X` is present,
- the flow `Y` is present,
- the clock `h` is presente (it is the implicit action of the clock `$base` when it is not mentioned),

the evaluation (knowing its value) of `X` must be preceded (in the instant !) by the evaluation of `Y`.

The notation $(\mathtt{X}_1,\ldots,\mathtt{X}_n)\ (\mathtt{Y}_1,\ldots,\mathtt{Y}_p)\ \mathtt{h}$
is equivalent to $\mathtt{X}_i\ \mathtt{Y}_j\ \mathtt{h}$
pour tout couple (i, j).

5.4.5 External Nodes

An external node is a node whose GC description is not accessible ; the external index of its interface is furnished by the index `0` in the importation table.

Such an external node can represent from the GC perspective of the action of an imperative module ($\boxed{\textbf{ic:}}$; the external index of the module is in this cased provided by the index `1` in the importation table. Th index of the IC action follows immediately in the importation table.

An external node can also represent the GC perspective of a completely external object whose description in the format is not accessible ($\boxed{\textbf{extern:}}$).

♠ **EXTERNAL-NODE** ::= • $\boxed{\textbf{node:}}$ **EXTERNAL-IMPERATIVE-NODE-INFOS**
IMPORTATOIN-TABLE
action-**index**
[**PRAGMAS-TABLE**]
$\boxed{\textbf{endnode:}}$ •
• $\boxed{\textbf{node:}}$ **EXTERNAL-NODE-INFOS**
IMPORTATION-TABLE
[**PRAGMAS-TABLE**]
$\boxed{\textbf{endnode:}}$ •

♠ **IMPERATIVE-EXTERNAL-NODE-INFOS** ::=

• **Gc-format** *node*-**identifier** $\boxed{\textbf{ic:}}$ [**Computational-properties**] •

♠ **EXTERNAL-NODE-INFOS** ::=

• **Gc-format** *nœud*-**identifier** $\boxed{\textbf{extern:}}$ [**Computational-properties**] •

5.5 Properties

5.5.1 Synchronisation

The table of synchronizations is identified by the keyword $\boxed{\textbf{synchronizations:}}$.

♠ **Synchronisation-table-name** ::= • $\boxed{\textbf{synchronizations:}}$ •

Synchronizations specify constraints governing the semantics of the program. They express relations on the flows, in the form :

- **EXPRESSIONS** with a boolean result always being true when it is present ;

- **ÉQUATIONS**, **INSTANCIATIONS** or d'**PROCEDURE-CALLS**, which specify equality constraints of clocks and their values spécifient des contraintes d'égalité d'horloges et de valeurs (**EQUATION-PART**).

An equation

```
define: E X
```

in the table of synchronizations is equivalent in this table to the expression

```
$clkeq($tt($eq(X,E)),$clock(X))
```

which must always be true.
The same relation is defined between each output flow `X` of an instantiation or procedure call in the synchronization table and expression `E` which defines it.

Synchronizations do not generate dependancies.

♠ **SYNCHRONISATION-OBJECT** ::= • **ASSERTION-OBJECT** •

5.5.2 Assertions

The table of assertions is identified by the keyword **assertions:**.

♠ **Assertion-table-name** ::= • **assertions:** •

Like synchronizations, assertions express relations on flows in the form :

- **EXPRESSIONS** with boolean result being always true when they are present ;
- **ÉQUATIONS**, **INSTANCIATIONS** or **APPELS-PROCÉDURES**, which specify the equality relations of clocks and values (cf. Section 5.5.1, page 88)

As distinct from synchronizations, the relations specifyed by assertions do not govern the semantics of programs, but describe functinoal hypotheses. If Σ_{node} and Σ_{assert} name respectively the semantics of sets of nodes and their assertion part all working satisfactorily then Σ_{node} unde the hypothesis of Σ_{assert} must be respected if it is to be considered correct.

The same as for synchronizations, assertions do not generate dependencies.

♠ **ASSERTION-OBJECT** ::= • **EXPRESSION** •
• **EQUATION-PART** •

5.6 Example

An example of a GC package is given.

```
package: 5 STOPWATCH

0: data: dc:0 STOPWATCH_DATA
   types: 1
   0: STOPWATCH_TIME_TYPE; -- 0 code $eq, 1 code $ne, 2 code $cond
   end:

   constants: 1
   0: INITIAL_STOPWATCH_TIME : 0;
   end:

   functions: 5
   0: $eq (0,0) : $1;
   1: $ne (0,0) : $1;
   2: $cond ($1,0,0) : 0;
   3: INCREMENT_STOPWATCH_TIME (0) : 0;
   4: IS_ZERO_MOD_10_MN (0) : $1;
   end:

enddata: -- STOPWATCH_DATA

1: interface: gc:0 TWO_STATES_INTF
   flows: 4
   0: i: init  $1 $g0;
   1: i: set   $1 $g0;
   2: i: reset $1 $g0;
   3: o: state $1 $g0;
   end:
   dependences: 3
   0: 0 3 $g0;
   1: 1 3 $g0;
   2: 2 3 $g0;
   end:
endinterface: -- TWO_STATES_INTF

2: node: gc:0 TWO_STATES
   import: 1
   0: 1;
   end:
   definitions: 1
   0: define:
       0.3 $g12(0.0,$3($5(0.1,$6($g11(0.3))),@$2,$3($5(0.2,$g11(0.3)),@$1,$g11(0.3))));
   end:
endnode: -- TWO_STATES
```

```
3: interface: gc:0 STOPWATCH_INTF
   import: 1
   0: 0;
   end:
   flows:7
   0: i: hs $1 $g0;
   1: i: start_stop $1 $g0;
   2: i: lap $1 $g0;
   3: o: time 0.0 $g0;
   4: o: run_state $1 $g0;
   5: o: lap_state $1 $g0;
   6: o: beep $2 $g0;
   end;
   assertions: 1
   0: $g7(0,1,2)
   end;
   dependences: 3
   0: 1 4;
   1: 2 5;
   2: (1,2,3) (3,6);
   end;
endinterface: -- STOPWATCH_INTF

4: node: gc:0 STOPWATCH
   import: 3
   0: 3;
   1: 0;
   2: 2;
   end:
   flows: 3
   0: reset $1 $g0;
   1: must_beep $1 $g0;
   2: internal_time 1.0 $g0;
   end:
   definitions: 7
   0: define: 0 $g12(@$1, $5(0.2,$g11($5($6(0.4),$6(0.5)))));
   1: set: 2 (flows: @$1/0,0.1/1,0.1/2,0.4/3);
   2: set: 2 (flows: @$1/0,$5(0.2,0.4)/1,0.2/2,0.5/3);
   3: define: 0.3 $g9($g10(2,$g8($6(0.5))),$g11(0.3));
   4: define: 2 1.2($g12(@$2,0), @1.0, 1.2($5(0.4,0.0), 1.3($g11(2)), $g11(2)));
   5: define: 1 $3(0.1,@$2,$3($5(0.0,0.4),1.4(2),@$1));
   6: define: 0.6 $9(1,#1,#0);
   end:
endnode: -- STOPWATCH
```

```
endpackage: -- STOPWATCH
```

5.7 The Semantic Model of the GC Format

The GC format allows for the descrition of dynamic systems. A GC program behaves consists essentially of a declarative part (which describes a hierarchical graph of schematic blocks), augmented with explicit control reserved espcially to translate the action sequences of imperative porgrams. This part describes the model underlying the declarative part. This consists in the *specification* of the set of behaviors equal to the *signals* under consideration. These notions are introduced and formalized in the two following sections. The complete semantics of GC is described in Appendix ??.

5.7.1 Objects

Clocks, Flows, Traces : A clock is a strictly increasing set, finite or divergent, of instants in $\mathbb{R}^+$ where $\mathbb{R}^+$ designates the set of real numbers positive or zero. The set operators — union ($\cup$), intersectin ($\cap$)), difference ($\ominus$) — are available on clocks. Note that $|h|$ denotes teh number of elements (finite or denumerable) of the clock h.

A *signal* is a set of values synchronized ith a clock : all signals X are therefore a pair (h^X, v^X) where h^X is a clock and v^X is a set of vaules of type X of the same length as h^X. Intuitively, X posesses the value v_n^X at the instant h_n^X. A *trace* is a vector of signals $T = (X_1, \ldots, X_p)$.

Commentary : The particular values of instants of a clock are without significance, they are used only for the total ordering property. As a consequence, if one considers a signal X taken in isolation, only the followling are pertinent

- the length $|h^X|$ of its clock,

- the set $v_1^X, \ldots, v_n^X, \ldots$ of its values.

The notion of clock becomes useful when one considers more multiple signals in conjunction : one can so define the *constraints* on their clocks, as indicated in Table 5.1. This degree of freedom is formalized in the following sections.

Time Changes : *Time changing* is a bijection of $\mathbb{R}^+$ on itself, preserving its order and leaving 0 and $+\infty$ invariant. By consequent, the time change transforms clocks to clocks of the same length.

contraints clock	significance
$h = k$	equal clock
$h \# k$	clocks without a common instant
$h \subset k$	the instants of h are the instants of k

Table 5.1: Constraints on Clocks

5.7.2 Specifications

A *specification* is simple given as,

1. a finite set A of typed signal names, or "ports." For each $a \in A$, $\mathcal{D}_a$ names the domain (or type) of values of the signal X_a with name a.

2. the subset Σ of the set $\mathcal{T}_A$ of all the traces of the form $(X_a)_{a \in A}$, where X_a is a signal of type D_a; this subset Σ satisfying the following property :

$$\Sigma \quad \textit{is invariant by all time changes} \tag{5.1}$$

This is to say, only the global interlacing of clocks of the set of signals is significant, the particular valuevalues taken by the instants are not. Two operators will be described in the specifications : the *restriction* and *composition.*

Restriction : Given a specificatoin Σ bearing on teh set of signals A, and considering a subset of ports $A' \subset A$. The restriction of

Σ to A' is written $\Sigma_{!!A'}$; it is composed of the set of traces $T' = (X_a)_{a \in A'}$ where $T = (X_a)_{a \in A}$ appears in Σ. Note that $\Sigma_{!!A'}$ satisfies the the property as well (5.1).

Composition : Consider two specifications Σ_1 and Σ_2, respectively defined on the set of ports A_1 and A_2. So $\Sigma_1|\Sigma_2$ designates the maximal specification [1] Σ define on the set of ports $A = A_1 \cup A_2$, satisfies the following conditions :

$$\begin{aligned} \Sigma_{!!A_1} &\subseteq \Sigma_1 \\ \Sigma_{!!A_2} &\subseteq \Sigma_2 \end{aligned}$$

This is to say, composition constrains the signals of Σ_1 and Σ_2 with the same name to be equal.

[1]for the inclusion $\Sigma' \subseteq \Sigma$ defined in the specifications.

Example : An illustration of the notion of composition in a case where the specifications Σ_1 and Σ_2 are, each, made explicity by a small simple program such that one can write it in GC. Σ is given by the GC equation

```
Y := X when tt(X>0)
```

which signifies that `Y` is present and has the same value as `X` when the condition `X>0` is satisfied, and is absent in the other case (the GC operator "`tt(b)`" delivers a clock whose instants of presence are ones where the boolean signal `b` is present and has the value *"vrai"*, whereas "`X when h`" transmits the values of `X` when the clock `h` is present). This instruction corresponds to the specification Σ_1 following, bearing on the signals `X, Y` :

$$h^Y = \{h_n^X \mid v_n^X > 0\}$$
$$\forall n \leq |h^Y|,\ v_n^Y = v_{m_n}^X, \text{ where } m_n \text{ is the index such that } h_{m_n}^X = h_n^Y$$

The other part, Σ_2 is given by the equation

```
Z := Y + U
```

whose specification is that the signals `U`, `Y` et `Z` are present simulataneously (they have therefore the same clock) and that, when they are present the third is the sum of the first two. The specification $\Sigma_1|\Sigma_2$ results from the composition of two specifications Σ_1 and Σ_2, which will be simply represented by the GC program

```
Y := X when tt(X>0)
Z := Y + U
```

is therefore given by the equations

$$h^Y = h^Z = h^U = \{h_n^X \mid v_n^X > 0\}$$
$$\forall n \leq |h^Y|,\ \left\{\begin{array}{lcl} v_n^Y & = & v_{m_n}^X \\ v_n^Z & = & v_n^Y + v_n^U \end{array}\right.$$

and can tehrefore be interpreted intuitively as the following :

> `Y`, `Z` et `U` are not present in the instants where `X` is present and has a positive value, and on these instants it is the case that `Y = X` and `Z = Y + U`.

In particular, the effect of this composition on the input `U` of this prgram can be interpreted as a "read suspension" of `U` while waiting for the next positive occurrence of `X`, and this without loss of the present occurrence of the signal `U`.

In the presentation of GC, the semantics of GC objects is given in terms of specifications in the above sense. A specification is defined by means of the following information :

1. **List and types of signals under consideration :** the specification under consideration is therefore a subset of the value traces in domain thus introduced.

2. **Clock Equations :** they express the clocks associated with various signals, in the same way as relations between clocks.

3. **Signal Equations :** they give the equations being satisfied between the present values of signals under consideration, with mention of instants where the equations are in force.

   The equations of signals are each preceded by a clock expression which specifies the instants where this equation is valid.

   For example, one writes écrit

   $$h_X = h_Y = h_Z \ , \ \forall n \leq |h_X| : \ v_n^Z = v_n^X + v_n^Y$$

   to indicate that `X`, `Y`, and `Z` have the same clock and that on the instants of this clock, the value of `Z` is the sum of the ones of `X` and `Y`.

   As well, one writes

   $$h_Z = h_X \cup h_Y \ , \ \forall n \leq |h_Z|, v_n^Z = \begin{cases} v_{m_n}^X, & \text{si } h_n^Z = h_{m_n}^X \in h^X \\ v_{k_n}^Y, & \text{si } h_n^Z = h_{k_n}^Y \in h^Y \ominus h^X \end{cases}$$

   to indicate that `Z` is present when `X` or `Y` it is, and that `Z=X` when `X` is present, and that `Z=Y` if progX is absent and `Y` present.

The composition of specifications (the | operator must be evidently associative and commutative, on can derive the semantic of the body of a GC node, by following the indicated above globally for the node under consideration, and by constructing first the semantics of each of the clock or signal equations, then by composing then semantics. The semantics of a GC node is so obtained in applying the corresponding specification of the body of the operator with the restriction that we have defined on specifications, to not allow visibility except for the interface ports of the node.

Chapter 6

The Sequential Imperative Format : OC

6.1 Introduction

In this part, we present the two tables specific to the OC format : the *DAGs* table (*Directed Acyclic Graphs*) and the one of states. These two tables appear in this order. Without these two tables, which replace the table of IC instructions, and represent the result of the compilation of these instructions into automata, the IC and OC formats are identical

The first section presents the context-free syntax of an OC module. The second section describes next the encoding of automata in OC and presents differents representations which are more or less compact. The following section gives the complete grammer for encoding the *DAGs* used by this representation, whereas the two last sections describe respectively the exact syntax of the *DAGs* table and the table of states.

6.2 Structure of the OC Format

The OC format has a structure much like the IC format and its context-free sy ntax is the following :

♠ **OC-MODULE** ::= • $\boxed{\textbf{module:}}$ **OC-MODULE-INFO**
COMMON-IMPERATIVE-PART
[**DAGS-TABLE**]
STATES-TABLE
$\boxed{\textbf{endmodule:}}$ •

♠ **OC-MODULE-INFO** ::=

• **OC-format** *module*-**identifier** [$\boxed{\textbf{flat:}}$] [**Computational-properties**] •

♠ **Format-oc** ::= • $\boxed{\textbf{oc:}}$ **Numéro-version** •

Recall that the **COMMON-IMPERATIVE-PART** was defined in teh chapter describing the IC format (cf. Section 4.2, page 45).

6.3 Encoding of Automata in OC

6.3.1 First Encoding

An automata consists of a set of states and a set of transitions between the states. Each transition is itself composed of a set of actions complete and an instruction changing the state. All the transitions emanating from the same state are encoded by a decision tree whose nodes are the actions to execute (represented by their index in the actins table) and the leaves are the state change instructions. The nodes are of two types, distinguished by the number of children :

- nodes having one child are pure actions (cf. Section 3.3, page 40);
- nodes having two children are test actions (cf. Section **??**, page **??**), each child representing the continuation of the transition depending on whether the test is true or false.

The encoding of these decision trees in OC is made in the following manner :

- actions are encoded by their index in the actions table;
- a state change instruction is encoded by $\boxed{<}$ *state*-**index** $\boxed{>}$;

Nodes not having only child are simply written one after another whereas with the test nodes, one writes first the index of the test action then, between parenthesis, the branch to execute of the test is true, and also between parenthesis, the branch to execute if the test is false.

To give a title to the example, see Figure 6.1 which shows an extract of an ESTEREL program and the part corresponding to the decision tree generated by this program. Finally to facilitate reading, the actions in the decision tree contain at the same time their index and the complete labeling. Also, the references to signals are made with the aid of signal names and not by their indices into the signal table. This decision tree is encoded in OC with the aid of the action indices in the following manner:

```
1 (2 (3 5 <3>) (<2>)) (4 5 <3>)
```

The decision trees are regrouped in the state table where each state is represented directly by the corresponding decision tree.

```
present S1 then
   present S2 then
      emit O1
   else
      await I1
   end present
else
   emit O2
end present;
emit O3;
await I2
```

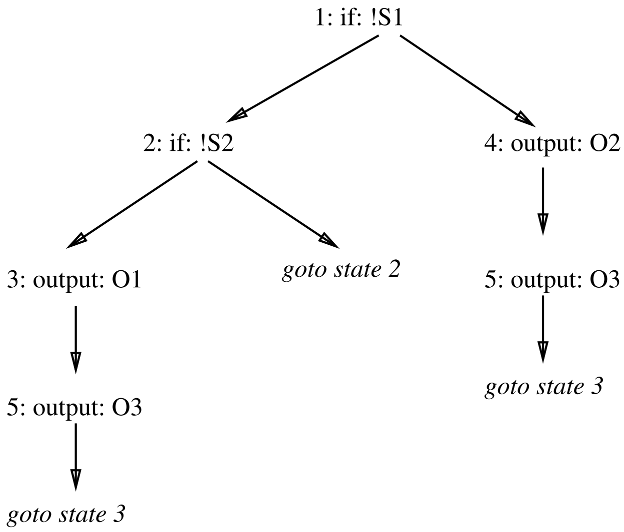


Figure 6.1: An Extract From an ESTEREL Program and its Associated Decision Tree

6.3.2 Transformation of Decision Trees into *DAGs*

A first modification to this representation allows for a more compact encoding. When the end of two branches of a test node of a tree are identicial one can share them and transforme the tree into a directed acyclic graph (DAG). The state change instructions stop the execution, one can share them also at the end of the branches not belonging directly to the same test (like for example the part "`5: output: O3; goto state 3`" in the preceding example). This, however, cannot be done for two branches of the same test sharing each a different continuation with other branches, in order to preserve the simple *DAGs*.

For example, the decision tree of the extract from the preceding ESTEREL program can be transformed into the dag of Figure 6.2.

Due to their particular structure, the *DAGs* obtained by transforming the decision trees can still be represented by trees. In effect, each part shared by two or more branches of a decision tree can be associated with the test on which the branches that share that part depend. In the example above, the continuation "`5: output: O3; goto state 3`" can be associated with the test "`1: if: !S1`." Such tests are so represented no longer by nodes having two culidren, but by nodes having three children : the third child is the continuation common to all the branchies which don't terminate with a state change instruction. To encode these three-child nodes in OC, one writes firs the index of the test action, then between parenthesis, the branch to execute if the test is true, then always between parenthesis, the branch to execute if the test is false and finally, directly following it, the possible continuation. thus, the *DAG* of the preceding example

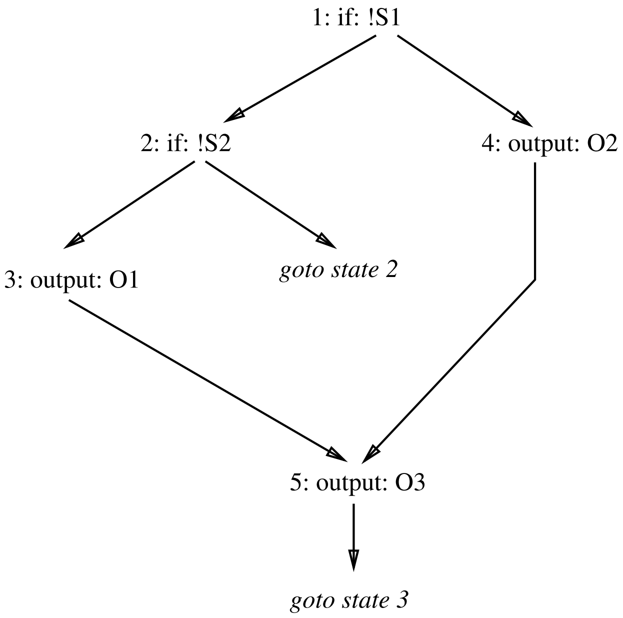


Figure 6.2: The *DAG* Associated With the Extract from the ESTEREL Program

is encoded in OCin teh following manner :

```
1 (2 (3) (<2>)) (4) 5 <3>
```

OV course when there are multiple tests broken up in a *DAG*, the same transition can use the successive continuations of multiple tests.

6.3.3 Sharing of Sub-DAGS

Being given a node of a *DAG*, one defines recursively the "sub-*DAG*" having this node for the root in the following fashion : its the set composed of the root and the nodes all of whose parents are in the sub-*DAGs*. For example, the part "`2: if: !S2, 3: output:O1, goto state 2`" is a sub-*DAG* (whose root is "`2: if: !S2`") of the dag of the preceding example, but not the part "`1: if: !S1, 4: output: O2, 5: output: O3`." A sub-*DAG* is said to be "closed" if and only if all the leaves are state change instructions. In the other case the sub-*DAG* is said to be "open." Recall in passing that all the *DAGs* associated with automata statse are closed.

A second modification to the representation of *DAGs* allows for an encoding even more compact. when a sub-*DAG* (open or closed) appears multiple times in one or more *DAGs*, it can be shared by replacing all the occurrences of it by a reference to a unique instance of the sub-*DAG*. This unique intance is placed in the *DAG* table and the reference is called a "DAG *change instruction*" according to whether the referenced *DAG* is open or closed (this allows for the local verification of the closure of a *DAG*). The instructions indicating a change to an open *DAG* are encoded in OC by { *dag*-**index** }, whereas the ones that indicate a change to a closed *DAG* are encoded by [*dag*-**index**].

Thus the *DAG* of the preceding example can be represented by the two *DAGs* of Figure 6.3. This *DAG* is encoded in OC in the following way :

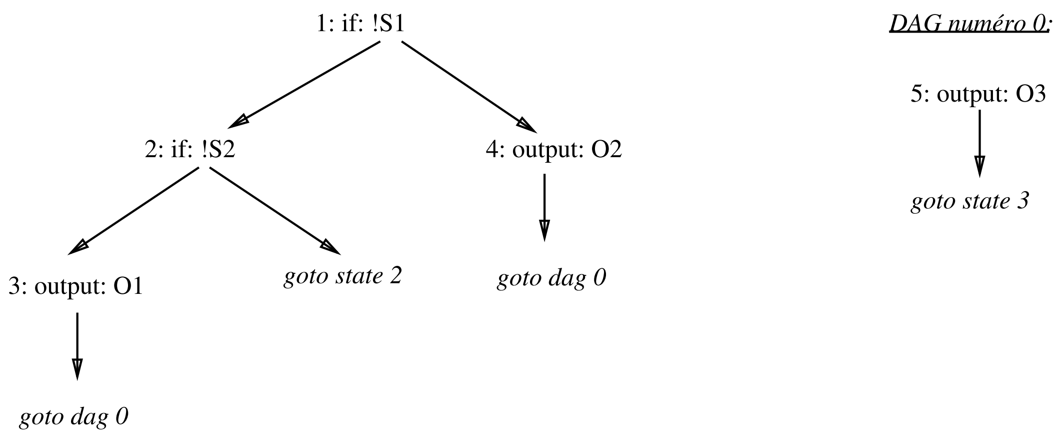


Figure 6.3: *DAGs* Associated with the Extract of the ESTEREL Program

```
1 (2 (3 [0]) (<2>)) (4 [0])
```

where the (closed) *DAG* with index 0 is the following :

```
5 <3>
```

6.4 The Dags Table

The table of *DAGs* is made up of :

- the keyword **dags:** identifying the table followed by the number of *DAGs* in the table,;
- some supplementary information indicating the totoal number of action calls made in the *DAGs* of the table (the keyword **calls:** followed by an integer and the ; character);
- the list of dags of the table, each *DAG* being preceded by its number and the : character and followed by a ; character;
- the keyword **end:**.

♠ **Dags-table-name** ::= • $\boxed{\textbf{dags:}}$ •

♠ **DAGS-TABLE-INFO** ::= • $\boxed{\textbf{calls:}}$ **Integer-const** $\boxed{;}$ •

6.5 The Grammar of *DAGs*

To continue, this section contains the complete syntax of the *DAGs*, that can be found in the table of statse or the table of *DAGs* :

♠ **DAG-OBJECT** ::= • **CLOSED-DAG**
• **OPEN-DAG** • •
CLOSED-DAG
• **OPEN-DAG DAG-CLOSURE** •

OPEN-DAG
• [**LIST-OF-DAGS**] [{ *pure-action*-**index** ... }] •

LIST-OF-DAGS
• { **OPEN-ACTIONS-AND-TEST** ... } •

OPEN-ACTIONS-AND-TEST
• { *pure-action*-**index** ... } **OPEN-TEST** •

DAG-CLOSURE
• $\boxed{<}$ *état*-**index** $\boxed{>}$ •
• $\boxed{[}$ *dag*-**index** $\boxed{]}$ •
• **CLOSED-TEST** •
CLOSED-TEST ::=

• *test-action*-**index** $\boxed{(}$ **CLOSED-DAGE** $\boxed{)}$ $\boxed{(}$ **CLOSED-DA**
•

OPEN-TEST ::=

• $\boxed{\{}$ *dag*-**index** $\boxed{\}}$ •
• *test-action*-**index** $\boxed{(}$ **OPEN-DAG** $\boxed{)}$ $\boxed{(}$ **CLOSED-DAG**
• *test-action*-**index** $\boxed{(}$ **CLOSED-DAG** $\boxed{)}$ $\boxed{(}$ **OPEN-DAG**
• *test-action*-**index** $\boxed{(}$ **OPEN-DAG** $\boxed{)}$ $\boxed{(}$ **OPEN-DAG**

6.6 The State Table

The state table is composed of :

- the keyword $\boxed{\textbf{states:}}$ identifying the table, followed by the number of states in the table,;
- some supplementary information indicating :
  - the index of the start state (the keyword $\boxed{\textbf{startpoint:}}$ followed by an index and the $\boxed{;}$ character) ;
  - possibly the index of the sink state (the keyword $\boxed{\textbf{sink:}}$ followed by an index and teh $\boxed{;}$ character) ;
  - the total number of action calls made in the *DAGs* of the able (the keyword $\boxed{\textbf{calls:}}$ followed by an integer and the $\boxed{;}$ character) ;
- the list of states in the table, each state being composed of its number followed by the $\boxed{:}$ character, the *DAG* encoding the transitions leaving this state and the $\boxed{;}$ character ;
- the keyword $\boxed{\textbf{end:}}$.

♠ **State-table-name** ::= • $\boxed{\textbf{states:}}$ •

♠ **STATE-TABLE-INFO** ::= • $\boxed{\textbf{startpoint:}}$ *start-state-index*-**index** $\boxed{;}$
[$\boxed{\textbf{sink:}}$ *sink-state-index*-**index** $\boxed{;}$]
$\boxed{\textbf{calls:}}$ **Integer-const** $\boxed{;}$ •

♠ **STATE-OBJET** ::= • **CLOSED-DAG** •

Appendix A

List of Pragmas

The terminal **Pragma-value** in the format is defined in terms of a syntax which is defined in terms of the name of the pragma ; this syntax can be defined with the aid of the non-terminals and terminals of the foramt. For its definition, the terminal **Pragma-value** is here considered as a non-terminal **PRAGMA-VALUE**.

A.1 Common Pragmas

A.1.1 Entity Pragma "`main`"

The **main:** pragma is a compilation directive that can be associated with an entity thereby stating that it is the principal entity of the program.

♠ **Pragma-name** ::= • **main:** •

♠ **PRAGMA-VALUE** ::= • •

A.1.2 Commentary Pragma "`comment`"

The comment pragma is especially used to transport source code comments between intermediate codes. Its syntax is the following :

♠ **Pragma-name** ::= • **comment:** •

♠ **PRAGMA-VALUE** ::= • **String-const** •

The constant string of characters is the comment.

This pragma can be found in any arbtrary place in a package. It does not have a default value ; its name can therefore not be found in teh list of computed pragmas of a module.

A.2 Pragmas Appearing in IC and OC Modules

A.2.1 File Pragma "`file`"

The file pragma can not appear but in a table of instances and is used to name the source file in which the given instance can be found. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{file:}}$ •

♠ **PRAGMA-VALUE** ::= • **String-const** •

The constant string of characters is the name of the file.

This pragma can not be associated with anything except entries in an instance table, and then only once per entry. It does not have a default value; its name can therefore not appear in the list of computed pragmas of the module.

A.2.2 Directory Pragma "`dir`"

The directory pragma can not appear except in a table of instances and is used to designate the directory which the source file of the given instance is found. Its syntax is the following:

♠ **Pragma-name** ::= • $\boxed{\textbf{dir:}}$ •

♠ **PRAGMA-VALUE** ::= • **String-const** •

The constant string of characters is the name of the directory.

This pragma can not be associated with anything except an entry in a table of instances, and then there can only be one per entry. It does not have a default value; its name can not therefore appear in the list of computed pragmas of a module.

A.2.3 Source Pragma "`lc`"

The source pragma "`lc`" is used to point to a place in teh source-text ("lc" signifies "line-character" or "line-column" if al chararacters have the same width). Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{lc:}}$ •

♠ **PRAGMA-VALUE** ::= • **Integer-const** **Integer-const** **Index** •

The two integer constants and the index represent respectively :

- the number of the line in the text ;
- the number of the character (or the coumn) on in line ;
- the index of the instance in teh table of instances.

The lines and columns are numbered starting at 1.

This pragma can be found in arbitrary examples in the module. It designates the spot of the source text that produced or contrubuted to the production of the object to which it is associated. It does not have a default value ; its name can therefore not appear in the list of computed pragmas of a module.

A.2.4 Source Pragma "`instance`"

This pragma is identical to the source pragma "`lc`" but it can not be associated with anything except an entry in the table of instances. It indicates the spot where the instance is made, which is to say the spot in the parent module where the instance to which the pragma is associated is used. The syntax of this pragma is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{instance:}}$ •

♠ **PRAGMA-VALUE** ::= • **Integer-const Integer-const Index** •

A.2.5 Pragma "`name`"

The name pragma is used to associate a name with an object. Its syntax is the following :

♠ **Pragma-nom** ::= • $\boxed{\textbf{name:}}$ •

♠ **PRAGMA-VALUE** ::= • **Identifier** •

This pragma can be associated with any arbitrary object in a package but not all obbject can have more than one name. If manu names must be associated with an object, the "`alias`" pragma, described below, must be used.

This pragma does not have a default value ; its name can therefore not appear in the list of computed pragmas of a module.

A.2.6 Pragma "`alias`"

The alias pragma is used to assicate a list of names with an object. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{alias:}}$ •

♠ **PRAGMA-VALUE** ::= • **IDENTIFIER** [{ **IDENTIFIER** ... }] •

This pragma can be associated with any object in a package. It does not have a default value ; its name can therefore not appear in the list of computed pragmas of a modue. It is important to note that no list in the name is distinguished. If a particular name must be associated with an object, the "`name`" pragma described above must be used.

Warning : this pragma can be very long (more than a thousand characters!) in the case of large examples. As a consequence fixed sized buffers in lexical analyzers is not reccomended.

A.2.7 Pragma "`previous`"

The "`previous`" pragma is used to indicate the "parent" of a signal in the signal declaration tree. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{previous:}}$ •

♠ **PRAGMA-VALUE** ::= • **Index** •

The value "`-`" signifies that the signal is the root of the signal declaration tree. Otherwise the index designates the parent signal in the signal table. For the moment, this pragma does not have the sense that in the signal table where it is associated with one entry in the table. It does not have a default value ; its name therefore cannot appear in the list of computed pragmas of the module.

A.2.8 Pragma "`sigbool`"

The "`sigbool`" pragma associated with an entry in the table of variables, is used to indicate that this variable is the boolean presence indicator of a signal. Its syntax is the following:

♠ **Pragma-name** ::= • $\boxed{\textbf{sigbool:}}$ •

♠ **PRAGMA-VALUE** ::= • **Index** •

The index designates the signal to which the variable (necessarily of type boolean `$1`) is associated in the table of signals. The pragma does not have a default value; its name can therefore not appear in the list of computed pragmas of a module.

A.2.9 Pragma "`sigval`"

The "`sigval`" pragma associated with an entry in the table of variables, is used to indicate that this variable contains the current value of a signal. Its syntax is the following :

♠ **Pragma0name** ::= • **sigval:** •

♠ **PRAGMA-VALUE** ::= • **Index** •

The index designates the signal to which the value (of the same type as the signal) is associated in the table of signals. This pragma does not have a default value; its name can therefore not appear in the list of computed pragmas of a module.

A.2.10 Pragma "`count`"

The "`count`" associated with an entry in the table of variables is used to indicate that this variable had been created by the compiler and implements a counter. Its syntax is the following :

♠ **Pragma-name** ::= • **count:** •

♠ **PRAGMA-VALUE** ::= • •

This pragma does not have a default value; its name can therefore not appear in the list of computed pragmas of a module.

A.3 Pragmas Appearing only in OC Modules

A.3.1 Pragma "`wire`"

The "`wire`" pragma associated with an entry in the table of variables is used to indicate that this variable has been created by the compiler and that it contains the value of a wire in the translation of a system of equations in OC-monoloop. This signifies in particular that the assignment of a value to this variable is valid in the present instant. The syntax of this pragma is the following :

♠ **Pragma-name** ::= • **wire:** •

♠ **PRAGMA-VALUE** ::= • •

This pragma does not have a default value ; its name can theferore not appear in the list of computed pragmas of a module.

A.3.2 Pragma "`register`"

The "`register`" pragma, associated with an entry in the table of variables is used to indicate that this variable has been created by the compiler and that it contains the value of a register in the translation of a system of equations in OC-monoloop. This signifies in particular that the assignment of a value to this variable is not value until the following instant. The syntax of this pragma is the following :

♠ **Nom-pragma** ::= • $\boxed{\textbf{register:}}$ •

♠ **VALEUR-PRAGMA** ::= • •

This pragma does not have a default value ; its name can therefore not appear in the list of computed pragmas of a module.

A.3.3 Pragma "`haltset`"

The "`haltset`" pragma is used to designate a set of `halts` (stopping points). Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{haltset:}}$ •

♠ **PRAGMA-VALUE** ::= • { **Index** ... } •

The value of this pragma is a list of indices naming the ragmas in some table of pragmas. These pragmas give some information (ex. : source reference) associated to each `halt`. The " haltset" can be associated with two types of objects :

- with a *DAG* in the table of states : it indicates the set of `halts` corresponding to that state ;
- to an entry in the table of variables : it indicats thus a set of active `halts` in all the states in which the variable, necessarily of type boolean `$1`, has the value true `$2` (in the case of OC-monoloop).

The name of the pragma can appar in the list of computed pragmas in a module and the default value associated is the empty list.

A.3.4 Pragma "`emitted`"

The "`emitted`" pragma is used to indicate a set of signals emitted by the automaton in a given transition. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{emitted:}}$ •

♠ **PRAGMA-VALUE** ::= • { **Index** ... } •

The value of this pragma is a list of indices naming the signals in the signal table. This pragma can be associated with two types of objects :

- with an OC state change instruction : it indicates the signals emitted by the automaton during the transition from the current state to the new state in following that branch of the *DAG*;
- with an entry in the variables table : it indicates the signal emitted by the automaton since the when the variable, necessarily of type boolean `$1`, has the value true `$2` (in the case of OC-monoloop).

THe name of this pragma can appear in the list of computed pragmas of a module and the default value associated is the empty list.

A.3.5 Pragma "`started`"

The "`started`" pragma is used to indicate a set of tasks forked off by the automaton in a given situation. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{started:}}$ •

♠ **PRAGMA-VALUE** ::= • { **Index** ... } •

The value of this pragma is a list of indices naming the calls of the tasks in the `execs` table. This pragma can be associated with two types of objects :

- with an OC state change instruction : it indicates the tasks forked off by the automaton during th transition between the current state and the new state in following this branch of the *DAG*;
- with an entry in the variables table : it indicates the tasks forked off by the automaton when the variable, necessarily of type boolean `$1`, has the value true `$2` (in the case OC-monoloop).

The name of this pragma can appear in th list of computed pragmas of a module and the default value is the empty list.

A.3.6 Pragma "`killed`"

The "`killed`" pragma is used to indicate a soet of tasks killed by the automaton in a given situation. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{killed:}}$ •

♠ **PRAGMA-VALUE** ::= • { **Index** ... } •

The value of this pragma is a list of indices naming some task calls in the `execs` table. This pragma can be associated with two types of objects.

- with an OC state change instruction : it indicates that tasks killed by the automaton during the transition from the current state to the new state in following this branch of the *DAG*;
- with an entry in the variables table : it idnicates that the tasks killed by the automaton when teh variable, necessarily of type boolean `$1`, has the value `$2` (in the case of OC-monoloop).

The name of this pragma can appear in the list of computed pragmas of a module and the default value associated is the empty list.

A.3.7 Pragma "`awaited`"

The "`awaited`" pragma, associated with a *DAG* in the table of states, is used to indicate a set of signals waited for by the automat in this state. Its syntax is the following :

♠ **Pragma-name** ::= • $\boxed{\textbf{awaited:}}$ •

♠ **PRAGMA-VALUE** ::= • { **Index** ... } •

The value of this pragma is the list of indices naming the signal entries in the signal tables. These signals are such that if any of them are not present in the input event, then the automaton will not change its state. More exactly, this signifies that all the input signals that don't appear in the "`awaited`" pragma can not change the state of the automaton by them selves.

The of this pragma can appar in the list of computed pragmas of a module and the default value associated is the list of .indices of .all the input signals (this is to say of type $\boxed{\textbf{input:}}$, $\boxed{\textbf{inputoutput:}}$ or $\boxed{\textbf{return:}}$) of the automaton.

Appendix B

Grammar of the Formats

B.1 CHARACTERS

♠ **Character** ::= • **character** • **CaracterCode** •

♠ **character** ::= • **mark** • **delimiter** • **separator** • **char-ident** • **autre-character** •

♠ **mark** ::= • $\boxed{\$}$ • $\boxed{:}$ • $\boxed{.}$ • $\boxed{+}$ • $\boxed{-}$ • $\boxed{\#}$ • $\boxed{@}$ • $\boxed{!}$ • $\boxed{?}$ • $\boxed{''}$ • $\boxed{\%}$ •

♠ **delimiter** ::=

• $\boxed{(}$ • $\boxed{)}$ • $\boxed{,}$ • $\boxed{/}$ • $\boxed{:}$ • $\boxed{;}$ •
• $\boxed{<}$ • $\boxed{>}$ • $\boxed{[}$ • $\boxed{]}$ • $\boxed{\{}$ • $\boxed{\}}$ •

♠ **séparateur** ::= • *horizontal tab* • *new line* • *new page* •
• *carriage return* • *space* •

Identifier Characters

♠ **char-ident** ::= • **char-letter** • **char-digit** • $\boxed{\_}$ •

♠ **char-letter** ::= • **char-uppercase-letter** • **char-lowercase-letter** •

♠ **char-upercase-letter** ::=

• $\boxed{\mathbf{A}}$ • $\boxed{\mathbf{B}}$ • $\boxed{\mathbf{C}}$ • $\boxed{\mathbf{D}}$ • $\boxed{\mathbf{E}}$ • $\boxed{\mathbf{F}}$ • $\boxed{\mathbf{G}}$ • $\boxed{\mathbf{H}}$ • $\boxed{\mathbf{I}}$ •
• $\boxed{\mathbf{J}}$ • $\boxed{\mathbf{K}}$ • $\boxed{\mathbf{L}}$ • $\boxed{\mathbf{M}}$ • $\boxed{\mathbf{N}}$ • $\boxed{\mathbf{O}}$ • $\boxed{\mathbf{P}}$ • $\boxed{\mathbf{Q}}$ • $\boxed{\mathbf{R}}$ •
• $\boxed{\mathbf{S}}$ • $\boxed{\mathbf{T}}$ • $\boxed{\mathbf{U}}$ • $\boxed{\mathbf{V}}$ • $\boxed{\mathbf{W}}$ • $\boxed{\mathbf{X}}$ • $\boxed{\mathbf{Y}}$ • $\boxed{\mathbf{Z}}$ •

♠ **char-lowercase-letter** ::=

• $\boxed{\text{a}}$ • $\boxed{\text{b}}$ • $\boxed{\text{c}}$ • $\boxed{\text{d}}$ • $\boxed{\text{e}}$ • $\boxed{\text{f}}$ • $\boxed{\text{g}}$ • $\boxed{\text{h}}$ • $\boxed{\text{i}}$ •
• $\boxed{\text{j}}$ • $\boxed{\text{k}}$ • $\boxed{\text{l}}$ • $\boxed{\text{m}}$ • $\boxed{\text{n}}$ • $\boxed{\text{o}}$ • $\boxed{\text{p}}$ • $\boxed{\text{q}}$ • $\boxed{\text{r}}$ •
• $\boxed{\text{s}}$ • $\boxed{\text{t}}$ • $\boxed{\text{u}}$ • $\boxed{\text{v}}$ • $\boxed{\text{w}}$ • $\boxed{\text{x}}$ • $\boxed{\text{y}}$ • $\boxed{\text{z}}$ •

♠ **char-digit** ::= • $\boxed{0}$ • $\boxed{1}$ • $\boxed{2}$ • $\boxed{3}$ • $\boxed{4}$ • $\boxed{5}$ • $\boxed{6}$ • $\boxed{7}$ • $\boxed{8}$ • $\boxed{9}$ •

Coded Characters

♠ **CaractereCode** ::= • **OctalCode** • **HexidecimalCode** • **escape-code** •

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

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

♠ **HexidecimalCode** ::= • $\boxed{\backslash\text{x}}$ **char-hexidecimal** [**char-hexidecimal**] •

♠ **char-hexidecimal** ::=

• **char-digit** •
• $\boxed{\text{A}}$ • $\boxed{\text{B}}$ • $\boxed{\text{C}}$ • $\boxed{\text{D}}$ • $\boxed{\text{E}}$ • $\boxed{\text{F}}$ • $\boxed{\text{a}}$ • $\boxed{\text{b}}$ • $\boxed{\text{c}}$ • $\boxed{\text{d}}$ • $\boxed{\text{e}}$ •
$\boxed{\text{f}}$ •

♠ **escape-code** ::=

• $\boxed{\backslash\text{a}}$ • $\boxed{\backslash\text{b}}$ • $\boxed{\backslash\text{f}}$ • $\boxed{\backslash\text{n}}$ • $\boxed{\backslash\text{r}}$ • $\boxed{\backslash\text{t}}$ • $\boxed{\backslash\text{v}}$ • $\boxed{\backslash\backslash}$ • $\boxed{\backslash\text{"}}$ • $\boxed{\backslash\text{'}}$ • $\boxed{\backslash\text{?}}$ •

B.2 LEXICAL UNITS

Integer Constants

♠ **Integer-const** ::= • { **car-digit** ... } •

Real Constants

♠ **Real-const** ::= • **Real-const-single-precision** • **Real-const-double-precision** •

♠ **Real-const-single-precision** ::= • **Fraction-Part Single-precision-exponent** •

♠ **Real-const-double-precision** ::= • **Fraction-part Double-precision-exponent** •

♠ **Fraction-part** ::= • **Integer-const** $\boxed{.}$ **Integer-const** •

♠ **Single-precision-exponent** ::= • $\boxed{e}$ **Relative-const** • $\boxed{E}$ **Relative-const** •

♠ **Double-precision-exponent** ::= • $\boxed{d}$ **Relative-const** • $\boxed{D}$ **Relative-const** •

♠ **Relative-const** ::= • **Integer-const** • $\boxed{+}$ **Integer-const** • $\boxed{-}$ **Integer-const** •

String Constants

♠ **String-const** ::= • $\boxed{"}$ [{ **CharactereString** ... }] $\boxed{"}$ •

♠ **CharacterString** ::= • { **Charactere** ⊖ **char-spec-string** } •

♠ **char-spec-sttring** ::= • $\boxed{"}$ • $\boxed{\backslash}$ •

Identifiers

♠ **Identifier** ::= • **char-begin-ident** [{ **char-ident** ... }] •
• $\boxed{\$}$ **char-begin-ident** [{ **char-ident** ... }] •

♠ **char-begin-ident** ::= • { **char-ident** ⊖ **char-digit** } •

Index

♠ **Index** ::= • **Local-index** • **Predefined-object-index** • **Compound-index** • **Undefined-index** •

♠ **Local-index** ::= • **Entry-number** •

♠ **Entry-number** ::= • **Integer-const** •

♠ **Predefined-object-index** ::= • $\boxed{\$}$ **Integer-const** • $\boxed{\$I}$ **Integer-const** • $\boxed{\$g}$ **Integer-const**

♠ **Compound-index** ::= • **Importation-index** $\boxed{.}$ **Local-index** •

♠ **Importation-index** ::= • **Integer-const** •

♠ **Undefined-index** ::= • $\boxed{-}$ •

B.3 PACKAGES

♠ PACKAGE ::=
• **package:** **PACKAGE-HEADER** { **PACKAGE-BODY** ... } **endpackage:** •

PACKAGE-HEADER

♠ PACKAGE-HEADER ::= • **Entity-quantity** *package*-**identifier** •

♠ Entry-count ::= • **Integer-const** •

PACKAGE-BODY

♠ PACKAGE-BODY ::= • **Entity-number** **:** **ENTITY** •

♠ Entity-number ::= • **Integer-const** •

♠ ENTITY ::= • **DATA-BLOCK** •
• **IC-MODULE** •
• **INTERFACE** • **NODE** •
• **OC-MODULE** •

Table Meta-Description

♠ X-TABLE ::= • **Table-X-Name** **TABLE-X-HEADER**
[{ **TABLE-X-BODY** ... }] **end:** •

♠ TABLE-X-HEADER ::= • **Entry-count** [**TABLE-X-INFO**] •

♠ Entry-count ::= • **Integer-const** •

♠ TABLE-X-ENTRY ::= • **Entry-number** **:** **OBJECT-X** **;** •

B.3.1 DATA BLOCKS

♠ **DATA BLOCKS** ::= • $\boxed{\text{data:}}$ **DATA-BLOCK-INFO**
[**IMPORTATIONS-TABLE**]
[**TYPES-TABLE**]
[**CONSTANTS-TABLE**]
[**FUNCTIONS-TABLE**]
[**PROCEDURES-TABLE**]
[**PRAGMAS-TABLE**]
$\boxed{\text{enddata:}}$ •

♠ **DATA-BLOCK-INFO** ::=
• **Dc-format** *data-block*-**identifier** [$\boxed{\text{flat:}}$] •

♠ **Dc-format** ::= • $\boxed{\text{dc:}}$ **Version-number** •

♠ **Version-number** ::= • **Integer-const** •

B.3.2 IC MODULES

♠ **IC-MODULE** ::= • $\boxed{\text{module:}}$ **IC-MODULE-INFO**
COMMON-IMPERATIVE-PART
INSTRUCTIONS-TABLE
$\boxed{\text{endmodule:}}$ •

♠ **MODULE-IC-INFOS** ::= • **Ic-format** *module*-**identifier**
[$\boxed{\text{flat:}}$] [$\boxed{\text{linked:}}$]
[**Computational-properties**] •

♠ **Computational-properties** ::= • $\boxed{\text{safe:}}$ • $\boxed{\text{memorysafe:}}$ • $\boxed{\text{unsafe:}}$ •

♠ **Ic-format** ::= • $\boxed{\text{ic:}}$ **Version-number** •

♠ **COMMON-IMPERATIVE-PART** ::= • [**IMPORTATION-TABLE**]
[**INSTANCES-TABLE**]
[**VARIABLES-TABLE**]
[**SIGNALS-TABLE**]
[**RELATIONS-TABLE**]
[**TASKS-TABLE**]
[**TASK-CALLS-TABLE**]
[**ACTIONS-TABLE**]
[**PRAGMAS-TABLE**] •

B.3.3 INTERFACES

♠ INTERFACE ::= • $\boxed{\text{interface:}}$ **INTERFACE-INFO**
[IMPORTATION-TABLE]
DECLARATIVE-PARTIE
$\boxed{\text{endinterface:}}$ •

♠ INTERFACE-INFOS ::= • [**Gc-format**] *interface*-**identifier** [$\boxed{\text{flat:}}$] •

♠ Gc-format ::= • $\boxed{\text{gc:}}$ **Version-number** •

♠ DECLARATIVE-PART ::= • [**FLOWS-TABLE**]
[**ASSERTIONS-TABLE**]
[**SYNCHRONISATIONS-TABLE**]
[**DEPENDANCES-TABLE**]
[**PRAGMAS-TABLE**] •

B.3.4 NODES

♠ NODE ::= • **LOCAL-NODE** • **EXTERNAL-NODE** •

LOCAL-NODE

♠ LOCAL-NODE ::= • $\boxed{\text{node:}}$ **LOCAL-NODE-INFO**
IMPORTATION-TABLE
DECLARATIVE-PART
[DEFINITIONS-TABLE]
$\boxed{\text{endnode:}}$ •

♠ LOCAL-NODE-INFOS ::= • **Gc-format** *node*-**identifier** [$\boxed{\text{flat:}}$]
[**Computational-properties**] •

EXTERNAL-NODE

♠ EXTERANL-NODE ::= • $\boxed{\text{node:}}$ **IMPERATIVE-EXTERANL-NODE-INFOS**
IMPORTATION-TABLE
action-**index**
[PRAGMAS-TABLE]
$\boxed{\text{endnode:}}$ •
• $\boxed{\text{node:}}$ **EXTERANL-NODE-INFOS**
IMPORTATIONS-TABLE
[PRAGMAS-TABLE]
$\boxed{\text{endnode:}}$ •

♠ **IMPERATIVE-EXTERNAL-NODE-INFOS** ::=

• **Gc-format** *node*-**identifier** $\boxed{\textbf{ic:}}$ [**Computational-properties**] •

♠ **EXTERNAL-NODE-INFOS** ::=

• **Gc-format** *node*-**identifier** $\boxed{\textbf{extern:}}$ [**Computational-properties**] •

B.3.5 OC MODULES

♠ **OC-MODULE** ::= • $\boxed{\textbf{module:}}$ **OC-MODULE-INFOS**
COMMON-IMPERATIVE-PART
[**DAGS-TABLE**]
STATES-TABLE
$\boxed{\textbf{endmodule:}}$ •

♠ **OC-MODULE-INFOS** ::= • **Oc-format** *module*-**identifier** [$\boxed{\textbf{flat:}}$]
[**Computational-properties**] •

♠ **Format-oc** ::= • $\boxed{\textbf{oc:}}$ **Version-number** •

B.4 TABLES

B.4.1 IMPORTATION-TABLE

♠ **Importation-table-name** ::= • $\boxed{\textbf{import:}}$ •

♠ **Object-importation** ::= • **Entry-number** •
• *paquet*-**identifier** $\boxed{\;.\;}$ **Entry-number** •

B.4.2 TYPES-TABLE

♠ **Types-table-name** ::= • $\boxed{\textbf{types:}}$ •

♠ **TYPE-OBJECT** ::= • *type*-**identifier** • $\boxed{\texttt{\$win}}$ $\boxed{\;(\;}$ *type*-**index** $\boxed{\;)\;}$ •

B.4.3 CONSTANTS-TABLE

♠ **Constants-table-name** ::= • $\boxed{\textbf{constants:}}$ •

♠ **CONSTANT-OBJECT** ::=

• *constante*-**identifier** *type*-**index** [**CONSTANT-VALUE**] •

♠ **CONSTANT-VALUE** ::= • $\boxed{\textbf{value:}}$ **EXPRESSION** •

B.4.4 FUNCTIONS-TABLE

♠ **Functions-table-name** ::= • $\boxed{\textbf{functions:}}$ •

♠ **FUNCTION-OBJECT** ::= • *function*-**identifier** $\boxed{(}$ [{ *type*-**index** $\boxed{,}$ ... }] $\boxed{)}$ $\boxed{:}$ *type*-**index** [**Computational-properties**] •

B.4.5 PROCEDURES-TABLE

♠ **Procedures-table-name** ::= • $\boxed{\textbf{procedures:}}$ •

♠ **PROCEDURE-OBJECT** ::=

• *procedure*-**identifier** $\boxed{(}$ [{ **PARAMETER** $\boxed{,}$ ... }] $\boxed{)}$ [**Computational-processes**] •

♠ **PARAMETER** ::= • **INPUT-PARAMETER** •
• **OUTPUT-PARAMETER** •
• **INPUT-OUTPUT-PARAMETRE** •

♠ **INPUT-PARAMETER** ::= • $\boxed{\textbf{i:}}$ *type*-**index** •

♠ **OUTPUT-PARAMETER** ::= • $\boxed{\textbf{o:}}$ *type*-**index** •

♠ **INPUT-OUTPUT-PARAMETER** ::= • $\boxed{\textbf{io:}}$ *type*-**index** •

B.4.6 PRAGMAS-TABLE

♠ **Pragmas-table-name** ::= • $\boxed{\textbf{pragmas:}}$ •

♠ **PRAGMAS-TABLE-INFOS** ::= • $\boxed{\textbf{computed:}}$ [{ **Pragma-name** ... }] $\boxed{;}$ •

♠ **PRAGMAS-OBJECT** ::= • { **Pragma** ... } •

♠ Pragma ::= • Reference-pragma • Standard-standard •

♠ Reference-pragma ::= • $\boxed{\%}$ *pragmas*-index $\boxed{\%}$ •

♠ Standard-pragma ::= • $\boxed{\%}$ Pragma-name pragma-value $\boxed{\%}$ •

♠ Pragma-value ::= • [{ Pragma-unit ... }] •

♠ Pragma-unit ::= • PragmaCharacter • String-const •

♠ PragmaCharacter ::= • { Character ⊖ char-spec-pragma } •

♠ char-spec-pragma ::= • $\boxed{\%}$ • char-spec-string •

B.4.7 INSTANCES-TABLE

♠ Instances-table-name ::= • $\boxed{\text{instances:}}$ •

♠ INSTANCES-TABLE-INFOS ::= • $\boxed{\text{root:}}$ *instance*-index $\boxed{;}$ •

♠ INSTANCE-OBJECT ::= • *module*-identifier *module-parent*-index •

B.4.8 VARIABLES-TABLE

♠ Variables-table-name ::= • $\boxed{\text{variables:}}$ •

♠ VARIABLE-OBJECT ::= • *type*-index [INITIAL-VALUE] •

♠ INITIAL-VALUE ::= • $\boxed{\text{value:}}$ EXPRESSION •

B.4.9 SIGNAL-TABLE

♠ Signal-table-name ::= • $\boxed{\text{signals:}}$ •

♠ **OBJET-SIGNAL** ::= • **NATURE-DE-SIGNAL CANAL-DE-SIGNAL [BOOL-DE-SIGNAL]** •

♠ **SIGNAL-KIND** ::= • $\boxed{\textbf{input:}}$ *signal*-**identifier** •
• $\boxed{\textbf{return:}}$ *signal*-**identifier** •
• $\boxed{\textbf{output:}}$ *signal*-**identifier** •
• $\boxed{\textbf{inputoutput:}}$ *signal*-**identifier** •
• $\boxed{\textbf{local:}}$ •

♠ **SIGNAL-CHANNEL** ::=
• *type*-**index** [**ASSOCIATED-VARIABLE**] **CARDINALITY** •

♠ **ASSOCIATED-VARIABLE** ::= • $\boxed{\textbf{value:}}$ *variable*-**index** •

♠ **CARDINALITY** ::= • $\boxed{\textbf{single:}}$ • $\boxed{\textbf{multiple:}}$ *function*-**index** •

♠ **SIGNAL-BOOL** ::= • $\boxed{\textbf{bool:}}$ *variable*-**index** •

B.4.10 RELATIONS-TABLE

♠ **Relations-table-name** ::= • $\boxed{\textbf{relations:}}$ •

♠ **RELATION-OBJECT** ::= • **EXCLUSION** • **IMPLICATION** •

♠ **EXCLUSION** ::= • $\boxed{(}$ *signal*-**index** $\boxed{,}$ { *signal*-**index** $\boxed{,}$ ... } $\boxed{)}$ •

♠ **IMPLICATION** ::= • *signal*-**index** *signal*-**index** •

B.4.11 TASKS-TABLE

♠ **Tasks-table-name** ::= • $\boxed{\textbf{tasks:}}$ •

♠ **TASK-OBJECT** ::= • *task*-**identifier** $\boxed{(}$ [{ **PARAMETER** $\boxed{,}$ ... }] $\boxed{)}$ [**Computational-properties**] •

B.4.12 TASK-CALLS-TABLE

♠ **Task-calls-table-name** ::= • $\boxed{\textbf{execs:}}$ •

♠ **TASK-CALL-OBJECT** ::=

• *task*-**index** *signal*-**index** $\boxed{\ (\ }$ [{ **EXPRESSION** $\boxed{\ ,\ }$ . . . }] $\boxed{\ )\ }$ •

B.4.13 ACTIONS-TABLE

♠ **Actions-table-name** ::= • $\boxed{\textbf{actions:}}$ •

♠ **ACTION-OBJECT** ::= • { **ELEMENTARY-ACTION** . . . } •

♠ **ELEMENTARY-ACTION** ::=

- $\boxed{\textbf{act:}}$ *action*-**index** •
- $\boxed{\textbf{if:}}$ **EXPRESSION** •
- $\boxed{\textbf{call:}}$ *procedure*-**index** $\boxed{\ (\ }$ [{ **EXPRESSION** $\boxed{\ ,\ }$ . . . }] $\boxed{\ )\ }$ •
- $\boxed{\textbf{input:}}$ *signal*-**index** •
- $\boxed{\textbf{output:}}$ *signal*-**index** •
- $\boxed{\textbf{reset:}}$ *variable*-**index** •
- $\boxed{\textbf{combine:}}$ *signal*-**index** **EXPRESSION** •
- $\boxed{\textbf{start:}}$ *exec*-**index** •
- $\boxed{\textbf{kill:}}$ *exec*-**index** •
- $\boxed{\textbf{return:}}$ *exec*-**index** •
- $\boxed{\textbf{suspend:}}$ *exec*-**index** •
- $\boxed{\textbf{resume:}}$ *exec*-**index** •

B.4.14 INSTRUCTIONS-TABLE

♠ **Instructions-table-name** ::= • $\boxed{\textbf{instructions:}}$ •

♠ **INSTRUCTIONS-TABLE-INFOS** ::= • $\boxed{\textbf{startpoint:}}$ *instruction*-**index** $\boxed{\ ;\ }$
$\boxed{\textbf{goloops:}}$ **Integer-const** $\boxed{\ ;\ }$
$\boxed{\textbf{exitlevels:}}$ **Integer-const** $\boxed{\ ;\ }$ •

♠ **INSTRUCTION-OBJET** ::=

- **Action:** [*action*-**index**] (*instruction*-**index**) •
- **Test:** [*action*-**index**]
  (*instruction*-**index** , *instruction*-**index**) •
- **Reset:** [*signal*-**index**] (*instruction*-**index**) •
- **Emit:** [*signal*-**index**] (*instruction*-**index**) •
- **Access:** [*signal*-**index**] (*instruction*-**index**) •
- **Goto:** (*instruction*-**index**) •
- **Goloop:** (*instruction*-**index**) •
- **Fork:** { *instruction*-**index** } ({ *instruction*-**index** , . . . }) •
- **Parallel:** ({ *instruction*-**index** , . . . }) < *instruction*-**index** > •
- **Exit:** { *instruction*-**index** } **Integer-const** •
- **Halt:** < *instruction*-**index** >
  [{ { *appel-tâche*-**index** , . . . } }] **Integer-const** •
- **Watchdog:** { *instruction*-**index** } < *instruction*-**index** > •
- **Stay:** < *instruction*-**index** > •
- **Run:** *module*-**identifier** [**IC-RENAMINGS**]
  < *instruction*-**index** > (*instruction*-**index**) •
- **Return:** **Integer-const** •

♠ **IC-RENAMINGS** ::= • **IC-TYPE-RENAMINGS** ;
IC-CONSTANT-RENAMINGS ;
IC-FUNCTION-RENAMINGS ;
IC-PROCEDURE-RENAMINGS ;
IC-TASK-RENAMINGS ;
IC-SIGNAL-RENAMINGS ;
last-signal-**index** •

♠ **IC-TYPE-RENAMINGS** ::= • { **IC-TYPE-RENAMING** , . . . } •

♠ **IC-CONSTANT-RENAMINGS** ::=

- { **IC-CONTANT-RENAMING** , . . . } •

♠ **IC-FUNCTION-RENAMINGS** ::=
• { **IC-FUNCTION-RENAMING** $\boxed{,}$ ... } •

♠ **IC-PROCEDURE-RENAMINGS** ::=
• { **IC-PROCEDURE-RENAMING** $\boxed{,}$ ... } •

♠ **IC-TASK-RENAMINGS** ::= • { **IC-TASK-RENAMING** $\boxed{,}$ ... } •

♠ **IC-SIGNAL-RENAMINGS** ::= • { **IC-SIGNAL-RENAMING** $\boxed{,}$ ... } •

♠ **IC-TYPE-RENAMING** ::= • *type*-**identifier** $\boxed{/}$ *type*-**index** •

♠ **IC-CONSTANT-RENAMING** ::=
• *constant*-**identifier** $\boxed{/}$ *constant*-**index** •

♠ **IC-FUNCTION-RENAMING** ::= • *fonction*-**identifier** $\boxed{/}$ *fonction*-**index** •

♠ **IC-PROCEDURE-RENAMING** ::=
• *procedure*-**identifier** $\boxed{/}$ *procedure*-**index** •

♠ **IC-TASK-RENAMING** ::= • *task*-**identifier** $\boxed{/}$ *task*-**index** •

♠ **IC-SIGNAL-RENAMING** ::= • *signal*-**identifier** $\boxed{/}$ *signal*-**index** •

B.4.15 FLOWS-TABLE

♠ **Flows-table-name** ::= • $\boxed{\text{flows:}}$ •

♠ **FLOW-OBJECT** ::= • **FLOW-KIND FLOW-CHANNEL Clock** •

♠ **FLOW-KIND** ::= • $\boxed{\text{i:}}$ *flow*-**identifier** • [$\boxed{\text{o:}}$] *flow*-**identifier** •

♠ **FLOW-CHANNEL** ::= • *type*-**index** [**INITIAL-FLOW-VALUE**] •

♠ **INITIAL-FLOW-VALUE** ::= • $\boxed{\text{value:}}$ **EXPRESSION** •

♠ **Clock** ::= • *flow*-**index** •

B.4.16 ASSERTION-TABLE

♠ **Assertions-table-name** ::= • $\boxed{\textbf{assertions:}}$ •

♠ **ASSERTION-OBJECT** ::= • **EXPRESSION** • **EQUATION-PART** •

♠ **EQUATION-PART** ::= • **EQUATION** • **INSTANTIATION** •
• **PROCEDURE-CALL** •

EQUATION

♠ **EQUATION** ::= • $\boxed{\textbf{define:}}$ *signal*-**index** **EXPRESSION** •

INSTANTIATION

♠ **INSTANTIATION** ::=
• $\boxed{\textbf{set:}}$ [**Clock**] *node*-**index** $\boxed{\textbf{(}}$ **RENAMINGS** $\boxed{\textbf{)}}$ •

♠ **RENAMINGS** ::= • [**TYPE-RENAMINGS**]
[**CONSTANT-RENAMINGS**]
[**FUNCTION-RENAMINGS**]
[**PROCEDURE-RENAMINGS**]
FLOW-RENAMINGS •

♠ **TYPE-RENAMINGS** ::= • $\boxed{\textbf{types:}}$ { *type*-**index** $\boxed{/}$ *type*-**index** $\boxed{,}$ ... } •

♠ **CONSTANT-RENAMINGS** ::=
• $\boxed{\textbf{constants:}}$ { *constant*-**index** $\boxed{/}$ *constant*-**index** $\boxed{,}$ ... } •

♠ **FUNCTION-RENAMINGS** ::=
• $\boxed{\textbf{functions:}}$ { *function*-**index** $\boxed{/}$ *function*-**index** $\boxed{,}$ ... } •

♠ **PROCEDURE-RENAMINGS** ::=
• $\boxed{\textbf{procedures:}}$ { *procédure*-**index** $\boxed{/}$ *procédure*-**index** $\boxed{,}$ ... } •

♠ **FLOW-RENAMINGS** ::= • $\boxed{\textbf{flows:}}$ { **EXPRESSION** $\boxed{/}$ *flow*-**index** $\boxed{,}$ ... } •

PROCEDURE-CALL

♠ **PROCEDURE-CALL** ::=

• $\boxed{\textbf{call:}}$ [**Clock**] *procedure*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •

B.4.17 SYNCHRONISATIONS-TABLE

♠ **Synchronizations-table-name** ::= • $\boxed{\text{synchronizations:}}$ •

♠ **SYNCHRONISATION-OBJECT** ::= • **ASSERTION-OBJECT** •

B.4.18 DEPENDANCES-TABLE

♠ **Dependancies-table-name** ::= • $\boxed{\text{dependences:}}$ •

♠ **DEPENDANCY-OBJECT** ::=

• **FLOW-INDEX-LIST FLOW-INDEX-LIST** [**Clock**] •

♠ **FLOW-INDEX-LIST** ::= • *FLOW*-**index** • $\boxed{(}$ { *flow*-**index** $\boxed{,}$ ... } $\boxed{)}$ •

B.4.19 DEFINITIONS-TABLE

♠ **Definitions-table-name** ::= • $\boxed{\text{definitions:}}$ •

♠ **DEFINITION-OBJECT** ::= • **EQUATION-PART** • **ACTIVATION** •
• **DEFINITION-DEPENDANCIES** •

♠ **ACTIVATION** ::=

• $\boxed{\textbf{do:}}$ [**Clock**] *node*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •

♠ **DEFINITION-DEPENDANCIES** ::=

• **DEFINITION-INDEX-LIST DEFINITION-INDEX-LIST**
[**Clock**] •

♠ **DEFINITION-INDEX-LIST** ::= • *definition*-**index** •
• $\boxed{(}$ { *definition*-**index** $\boxed{,}$ ... } $\boxed{)}$ •

B.4.20 DAGS-TABLE

♠ **Dags-table-name** ::= • $\boxed{\textbf{dags:}}$ •

♠ **DAGS-TABLE-INFOS** ::= • $\boxed{\textbf{calls:}}$ **Integer-const** $\boxed{;}$ •

♠ **DAG-OBJECT** ::= • **CLOSED-DAG** •
• **OPEN-DAG** •

♠ **CLOSED-DAG** ::= • **OPEN-DAG DAG-CLOSURE** •

♠ **OPEN-DAG** ::= • [**DAG-LIST**] [{ *pure-action*-**index** ... }] •

♠ **DAG-LIST** ::= • { **OPEN-ACTIONS-AND-TEST** ... } •

♠ **OPEN-ACTIONS-AND-TEST** ::= • { *pure-action*-**index** ... } **OPEN-TEST** •

♠ **DAG-CLOSURE** ::= • $\boxed{<}$ *state*-**index** $\boxed{>}$ •
• $\boxed{[}$ *dag*-**index** $\boxed{]}$ •
• **CLOSED-TEST** •

♠ **CLOSED-TEST** ::=
• *test-action*-**index** $\boxed{(}$ **CLOSED-DAG** $\boxed{)}$ $\boxed{(}$ **CLOSED-DAG** $\boxed{)}$
•

♠ **OPEN-TEST** ::=
• $\boxed{\{}$ *dag*-**index** $\boxed{\}}$ •
• *test-action*-**index** $\boxed{(}$ **OPEN-DAG** $\boxed{)}$ $\boxed{(}$ **CLOSED-DAG** $\boxed{)}$ •
• *test-action*-**index** $\boxed{(}$ **CLOSED-DAG** $\boxed{)}$ $\boxed{(}$ **OPEN-DAG** $\boxed{)}$ •
• *test-action*-**index** $\boxed{(}$ **OPEN-DAG** $\boxed{)}$ $\boxed{(}$ **OPEN-DAG** $\boxed{)}$ •

B.4.21 STATES-TABLE

♠ **States-table-name** ::= • $\boxed{\textbf{states:}}$ •

♠ **STATES-TABLE-INFOS** ::= • $\boxed{\textbf{startpoint:}}$ *start-statet*-**index** $\boxed{;}$
[$\boxed{\textbf{sink:}}$ *final-state*-**index** $\boxed{;}$]
$\boxed{\textbf{calls:}}$ **Integer-const** $\boxed{;}$ •

♠ **STATE-OBJECT** ::= • **CLOSED-DAG** •

B.5 EXPRESSIONS

♠ **EXPRESSION** ::= • **FUNCTION-CALL** • **Term** •

♠ **FUNCTION-CALL** ::=
• *function*-**index** $\boxed{(}$ [{ **EXPRESSION** $\boxed{,}$ ... }] $\boxed{)}$ •

♠ **Term** ::= • **Atom** • **ConstantRef** • **IC-OC-Term** • **GC-Term** •

♠ **Atom** ::= • $\boxed{\#}$ **String-const** • $\boxed{\#}$ **Integer-const** • $\boxed{\#}$ **Real-const** •

♠ **ConstantRef** ::= • $\boxed{@}$ *constant*-**index** •

IC-OC-Term

♠ **IC-OC-Term** ::= • **Signal-presence** • **Signal-value** • *variable*-**index** •

♠ **Signal-presence** ::= • $\boxed{!}$ *signal*-**index** •

♠ **Signal-value** ::= • $\boxed{?}$ *signal*-**index** •

GC-Term

♠ **GC-Term** ::= • *flow*-**index** •

Appendix C

An Operational Semantics for the IC Code

C.1 Execution Examples of IC Instructions by the LcOc Processor

The LcOc processor of the Esterelcompiler translates IC programs with the `linked:` attribute into deterministic finite automata. Recall that the IC code with the attribute `linked:` is IC code that does not contain any `Run` instructions. The examples presented in this section are based on the algorithms governing the execution of the LcOc processor version 3. Other executions however, are possible that do not make reference to dynamic control paths.

We present in this section some examples of execution of IC programs. The execution of IC relies on the notion of an instantaneous transition of an Esterel program : an instant is a *maximal execution* of IC instructions. An execution is said to be maximal when control stops on `Halt` or `Return` instructions (called *stopping points*). After an executions we have a *reconstruction* which prepares the IC program for the following execution. The reconstruction orders the instructions which will be executed in the next instant and computes the new starting point of the program. This phase necessitates the introduction of a memory associated with each `Watchdog` and `Parallel` and which will be materialized in our examples by a box next to the affected instruction. An arror next to a stopping point indicates that control is blocked on that instruction. Next to the IC code, we give a graphical representation of it. We find the following connections :

Reconstruction

Contrôle statique

Contrôle dynamique

The reconstruction connections are static and represent fields between brackets (“<” et “>”) of IC instructions. Static control paths represent the continuation of instructions and dynamic control paths represent the memories introduced beforehand. The associated instructions like `Watchdog/Test` or `Parallel/Fork` for example are represented by two boxes next to each other.

We give here an abstract execution of IC instructions that demonstrates two stopping points of the execution according to the semantics : the order of of instructions in two branches of a `parallel` is arbitrary. The semantics does not regulate therefore any of the problems such as instantaneous dialogs or causality cycles. A computation of proper execution order is given in the thesis of G. Gonthier ; it is based on the computation of signals potentially emitted by an IC instruction.

Example 1 : The following program :

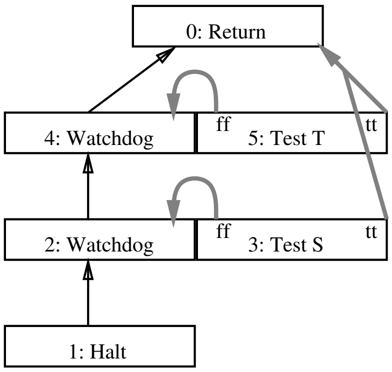
```
do
   await S
watching T
```

is translated into IC in the following fashion :

```
...
signals: 2
0: input: T $0 multiple: - bool: 0;
1: input: S $0 multiple: - bool: 1;
end:
...
statements: 6 startpoint: 1;
goloops: 0;
exitlevels: 0;
0: Return: 0;
1: Halt: <2> 1;
2: Watchdog: {3} <4>;
3: Test: [1] (0,2);
4: Watchdog: {5} <0>;
5: Test: [0] (0,4);
end:
endmodule:
```

The following recaptures the necessary information for the execution of this module and replaces the signal indices with their names. In the graphical representations, only the static connections are drawn (reconstruction, associated instructions and static control).

```
      startpoint: 1
      0: Return: 0
      1: Halt: <2> 1
[ - ] 2: Watchdog: {3} <4>
      3: Test: S (0,2)
[ - ] 4: Watchdog: {5} <0>
      5: Test: T (0,4)
```



Execution starts with the instruction whose index is given by the `startpoint` : in our example, we find the `Halt` with index 1. The control is therefore blocked and the execution is terminated as maximal.

The reconstruction begins at the end of the `Halt` by applying to it the following rules :

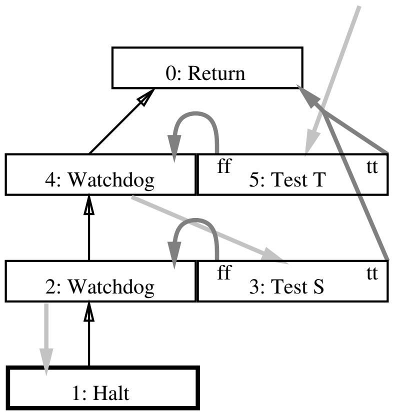
- the index of a `Halt` is put in the memory of its parent,
- the index of a `Parallel` is put in memory of its parent,
- the *guard* field of a `Watchdog` is put in the memory if its parent,
- when the parent is a `Return`, the reconstruction terminates and the index becomes the new entry point.

For the graphical representation, the rules become :

- add a dynamic control path between the parent instruction (follow the reconstruction path) of a `Halt` and this `Halt`,
- add a dynamic control path between the parent instruction of a `Parallel` and this `Parallel`,
- add a dynamic control path between the parent instruction of a `Watchdog` and the instruction associated with this `Watchdog`.
- when the parent is a `Return`, the reconstruction is termianted and the instruction pointed to by the last dynamic control path becomes the new entry point.

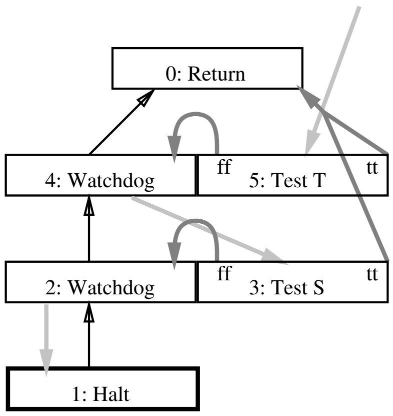
A new program is thus obtained with the control blocked at `Halt` 1. In the graphical representation, the dynamic control paths are made and the `Halt` 1 is framed in bold.

```
        startpoint: 5
        0: Return: 0
  →     1: Halt: <2> 1
[ 1 ]   2: Watchdog: {3} <4>
        3: Test: S (0,2)
[ 3 ]   4: Watchdog: {5} <0>
        5: Test: T (0,4)
```



The execution of in the second instant begins with the instruction given by the reconstruction that preceded : the `Test` with index 5. If we supposed that no inputs are present, the execution continues with the `Watchdog` 4. In executing a `Watchdog`, the instruction contained in its memory is executed : here the `Test` 3. Then finally, the `Watchdog` 2 and the `Halt`. The execution on the graphical representation is made simply by following the dynamic or static control paths. The appropriate choice must be made in the test instructions (like the `Test` here). The reconstruction is the same as for the first instant :

```
        startpoint: 5
        0: Return: 0
  →     1: Halt: <2> 1
[ 1 ]   2: Watchdog: {3} <4>
        3: Test: S (0,2)
[ 3 ]   4: Watchdog: {5} <0>
        5: Test: T (0,4)
```



If on the other hand, S or T are present, the execution stops on the `Return` and the program is terminated.

Example 2 : The following program :

```
   await S
||
   await T
```

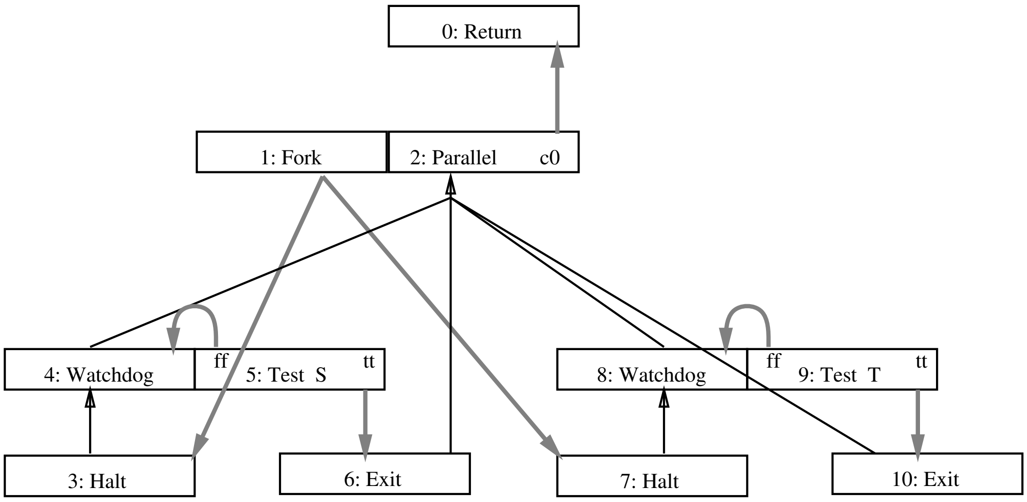
is translated into IC in the following fashion :

```
...
signals: 2
```

```
0: input: T $0 multiple: - bool: 0;
1: input: S $0 multiple: - bool: 1;
end:
...
statements: 11 startpoint: 1;
goloops: 0;
exitlevels: 0;
0: Return: 0;
1: Fork: {2} (3, 7);
2: Parallel: (0) <0>;
3: Halt: <4> 1;
4: Watchdog: {5} <2>;
5: Test: [1] (6,4);
6: Exit: {2} 0;
7: Halt: <8> 2;
8: Watchdog: {9} <2>l;
9: Test: [0] (10,8);
10: Exit {2} 10;
end:
endmodule:
```

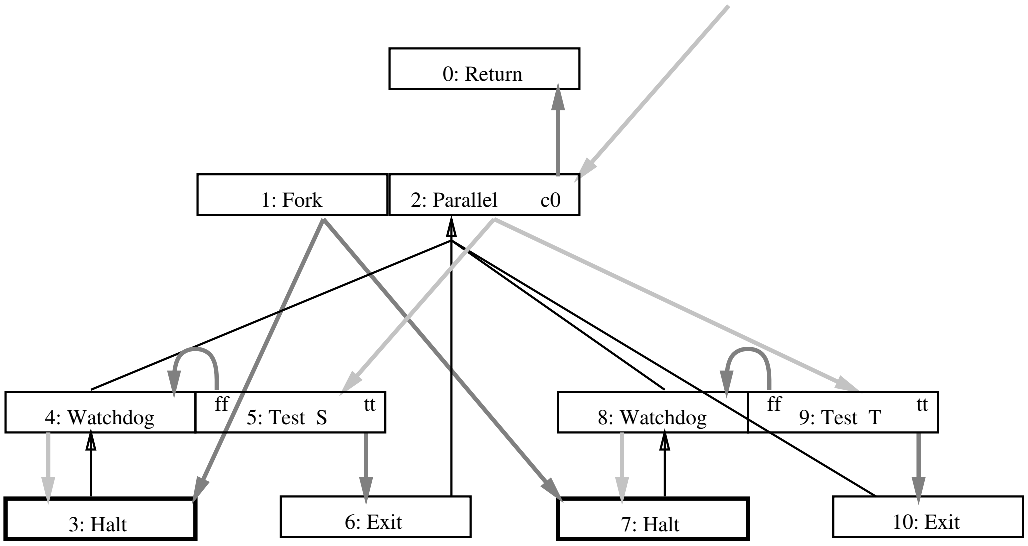
The initial program is the following :

```
        startpoint: 1
        0: Return: 0
        1: Fork: {2} (3, 7)
[ - ]   2: Parallel: (0) <0>
        3: Halt: <4> 1
[ - ]   4: Watchdog: {5} <2>
        5: Test: S (6,4)
        6: Exit: {2} 0
        7: Halt: <8> 2
[ - ]   8: Watchdog: {9} <2>
        9: Test: T (10,8)
        10: Exit: {2} 0
```



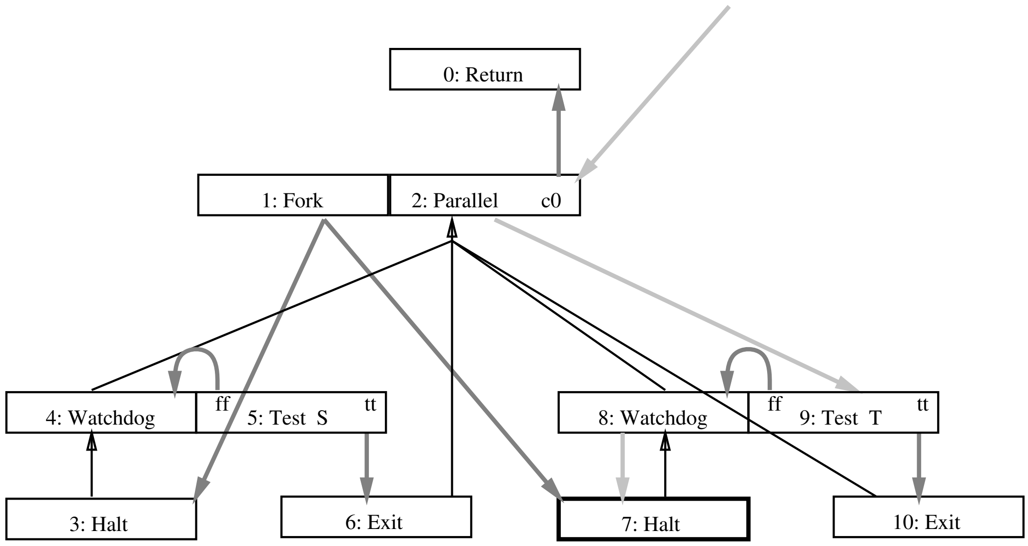
In the first instant, the `Fork` 1 then in parallel the two `Halt` 3 and 7. These two `Halt` form the basis for the reconstruction which gives :

```
            startpoint: 2
            0: Return: 0
            1: Fork: {2} (3, 7)
[5, 9]      2: Parallel: (0) <0>
     →      3: Halt: <4> 1
[3]         4: Watchdog: {5} <2>
            5: Test: S (6,4)
            6: Exit: {2} 0
     →      7: Halt: <8> 2
[7]         8: Watchdog: {9} <2>
            9: Test: T (10,8)
            10: Exit: {2} 0
```



In the second instant if `S` is present : to execute the `Parallel` 2, the instructions contained in its memory are executed. The execution ends at `Exit` 6 and `Halt` 7. The exit level of the `Parallel` is therefore $1 = max(0, 1)$. The reconstruction starts from only one `Halt` and gives :

```
        startpoint: 2
        0: Return: 0
        1: Fork: {2} (3, 7)
 [ 9 ]  2: Parallel: (0) <0>
        3: Halt: <4> 1
 [ - ]  4: Watchdog: {5} <2>
        5: Test: S (6,4)
        6: Exit: {2} 0
    →   7: Halt: <8> 2
 [ 7 ]  8: Watchdog: {9} <2>
        9: Test: T (10,8)
        10: Exit: {2} 0
```



Given that `T` is not present, the same program is always obtained. If `T` is present, the execution of the `Parallel` 2 then the `Test` 9 and therefore the `Exit` 10 are obtained. The exit level of the `Parallel` is 0 : it continues through to the `Return` and the execution is terminated :

```
      startpoint: 0
      0: Return: 0
      1: Fork: {2} (3, 7)
  -   2: Parallel: (0) <0>
      3: Halt: <4> 1
  -   4: Watchdog: {5} <2>
      5: Test: S (6,4)
      6: Exit: {2} 0
   →  7: Halt: <8> 2
  -   8: Watchdog: {9} <2>
      9: Test: T (10,8)
      10: Exit: {2} 0
```

Example 3 : The following program :

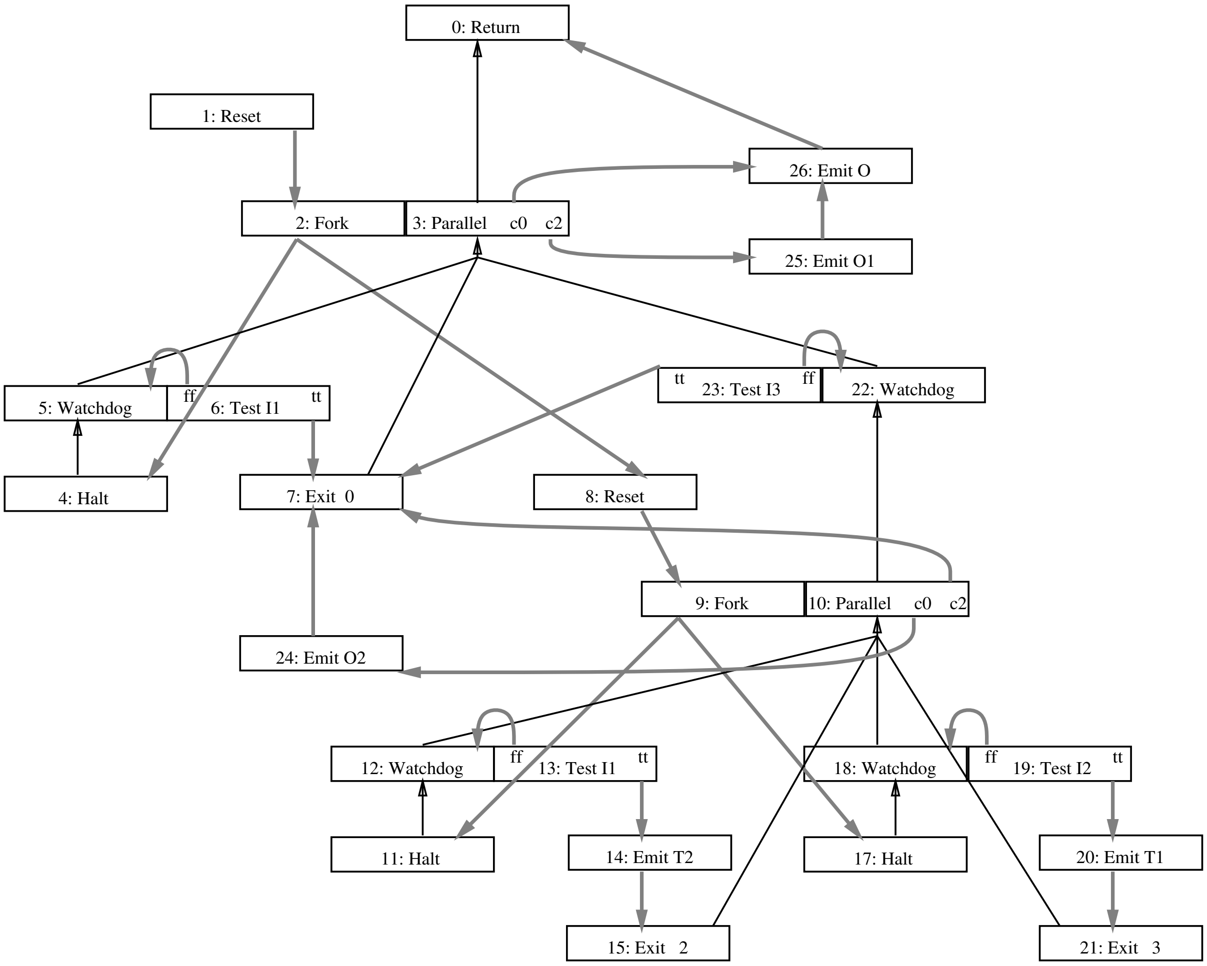
```
input I1, I2, I3;
output O, O1, O2;

trap T1 in
   [
      await I1
   ||
      trap T2 in
         do
            [
               await I1; exit T2
            ||
               await I2; exit T1
            ]
         watching I3
      handle T2 do
         emit O2
      end trap;
   ]
handle T1 do
   emit O1
end trap;
emit O
```

is translated into IC in the following fashion :

```
...
signals: 8
0: input: I1 $0 multiple: - bool: 0;
1: input: I2 $0 multiple: - bool: 1;
2: input: I3 $0 multiple: - bool: 2;
3: output: O $0 multiple: - ;
4: output: O1 $0 multiple: - ;
5: output: O2 $0 multiple: - ;
6: local: $0 multiple: - ;
7: local: $0 multiple: - ;
end:
...
statements: 27 startpoint: 1;
goloops: 0;
exitlevels: 3;
0: Return: 0;
1: Reset: [6] (2);
2: Fork: {3} (4, 8);
3: Parallel: (26, 25) <0>;
4: Halt: <5> 1;
5: Watchdog: {6} <3>;
6: Test: [0] (7,5);
7: Exit: {3} 0;
8: Reset: [7] (9);
9: Fork: {10} (11, 17);
10: Parallel: (7, 24) <22>;
11: Halt: <12> 2;
12: Watchdog: {13} <10>;
13: Test: [0] (14,12);
14: Emit: [7] (15);
15: Exit: {10} 2;
16: Exit: {10} 0;
17: Halt: <18> 3;
18: Watchdog: {19} <10>;
19: Test: [1] (20,18);
20: Emit: [6] (21);
21: Exit: {10} 3;
22: Watchdog: {23} <3>;
23: Test: [2] (7,22);
24: Emit: [5] (7);
25: Emit: [4] (26);
26: Emit: [3] (0);
end:
```

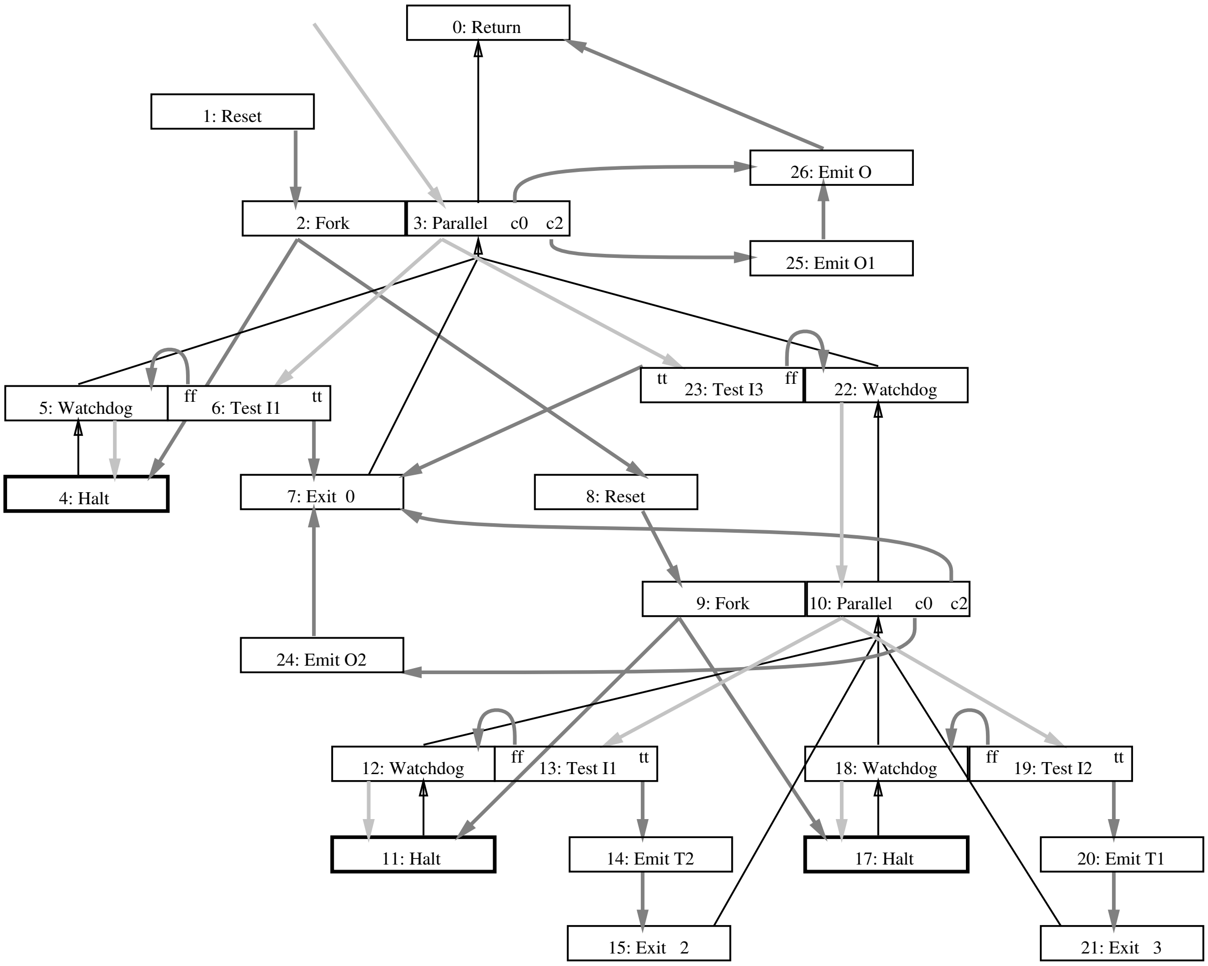
Execution starts from the following state :

```
    startpoint: 1
    0: Return: 0
    1: Reset: T1 (2)
    2: Fork: {3} (4, 8)
 -  3: Parallel: (26, 25) <0>
    4: Halt: <5> 1
 -  5: Watchdog: {6} <3>
    6: Test: I1 (7,5)
    7: Exit: {3} 0
    8: Reset: T2 (9)
    9: Fork: {10} (11, 17)
 -  10: Parallel: (7, 24) <22>
    11: Halt: <12> 2
 -  12: Watchdog: {13} <10>
    13: Test: I1 (14,12)
    14: Emit: T2 (15)
    15: Exit: {10} 2
    16: Exit: {10} 0
    17: Halt: <18> 3
 -  18: Watchdog: {19} <10>
    19: Test: I2 (20,18)
    20: Emit: T1 (21)
    21: Exit: {10} 3
 -  22: Watchdog: {23} <3>
    23: Test: I3 (7,22)
    24: Emit: O2 (7)
    25: Emit: O1 (26)
    26: Emit: O (0)
```



Execution is blocked on the `Halt` instructions 4, 11, and 17 and after the reconstruction the following is obtained :

```
            startpoint: 3
            0: Return: 0
            1: Reset: T1 (2)
            2: Fork: {3} (4, 8)
[6, 23]     3: Parallel: (26, 25) <0>
      →     4: Halt: <5> 1
[4]         5: Watchdog: {6} <3>
            6: Test: I1 (7,5)
            7: Exit: {3} 0
            8: Reset: T2 (9)
            9: Fork: {10} (11, 17)
[13, 19]    10: Parallel: (7, 24) <22>
      →     11: Halt: <12> 2
[11]        12: Watchdog: {13} <10>

            13: Test: I1 (14,12)
            14: Emit: T2 (15)
            15: Exit: {10} 2
            16: Exit: {10} 0
      →     17: Halt: <18> 3
[17]        18: Watchdog: {19} <10>
            19: Test: I2 (20,18)
            20: Emit: T1 (21)
            21: Exit: {10} 3
[10]        22: Watchdog: {23} <3>
            23: Test: I3 (7,22)
            24: Emit: O2 (7)
            25: Emit: O1 (26)
            26: Emit: O (0)
```



For the second instant, the only interesting caseis where I2 is present to give some interaction between `Exit` and `Parallel`. Execution therefore occurs as `3: Parallel`, `23: Test`, `22: Watchdog` and `10: Parallel`. Execution of the first branch and this last `Parallel` start at 13 and end at `11: Halt` ; the second starts at `19: Test` and ends at `21: Exit` $\{10\}$ 3. The exit level of `10: Parallel` is therefore 3. Besides this `Parallel` handles the exit levels up to ‘; it therefore transmits to its parent `Parallel` an exit level of $3 - 1 = 2$. After `10: Parallel`, there is `22: Watchdog` and `3: Parallel` which takes the continuation corresponding to level do which is `25: Emit: O1 (26)`. Next `26: Emit` and `0: Return` are executed and the execution for the instant is finished. All the reconstructions start henceforth from `0: Return` : the programme is therefore finished.

Notes :

The calculation of the exit level of the `Parallel` can be simplified by a calculation during the execution of the `Fork` instructions :

> During the execution of a `Fork`, all the continuations with level greater than or equal to 2 of the enclosing parallel are added to the end of the

list of continuations of the associated `Parallel`. associated

In the preceding example, execution arrives at `2: Fork` with the empty list and therefore `3: Parallel` is not modified. By the rule, the the new list of continuations must be `(25)` (the only continuation level greater than or equal to 2). During the execution of `9: Fork`, this list is added to `10: Parallel` to get `10: Parallel: (7, 24, 25) <22>`. The new list is `(24, 25)` and `10: Parallel` can now handle the `Exit` instructions up to level 3.

C.2 An Operational Semantics of IC Code

In this section we define an operational semantics for the IC code with the aide of an abstract machine. We want to demonstrate the execution mechanics of IC instructions independent of the Esterel language to give IC a status of an entirely independent language and to allow the use of it in other contexts. This mechanical execution is the one used by the LcOc processor, version 3.

The reconstruction presented in the examples and formalized in the following works through *dynamically* a number of examples. In fact, these examples computed dynamically are alwasy the subset of the examples *statically* computable from IC code. Other algorithms for execution of IC code exist, in particular the one used in Esterel V4.

First, we give some definitions of types and functions used next in the semantics.

C.2.1 Definitions

In the following, *Object* (with an uppercase O) designates a set of *objects* (in lower case). We define the following notation :

- Instructions

  instruction denotes an IC instruction.

  instrIndexList denotes a list of instruction indices.

- Stopping points

  haltset is a set of indices of stopping points.

  level is an integer giving the exit level of an instruction.

- Memories

  memInstr is a table associating with an instruction index a list of instructions to execute that we will call in the following *associated instructions* This list is non-empty for `Parallel` and `Watchdog` instructions. They correspond to the associated memory in the preceding examples.

memPar is a table associating with an instruction index a triplet containing :

- a set of indices of stopping points. This set is non-empty for `Parallel` instructions and gives the indices of the `Halt` instructions on which a `Parallel` is blocked. This set will be written $M[p].haltset$.
- a counter giving the number of active branches of a `Parallel` (i.e. the branches for wh ich the control is neither at a `Halt` nor at a `Exit`) written $M[p].active$.
- an integer giving the exit level of a `Parallel` written $M[p].level$.

With the preceding definitions and for a program `P` given, we define the functions :

- $\texttt{Execute}_{\texttt{P}}$:
  $$Instruction \times InstrIndexList \times MemInstr \times MemPar \times Haltset \longrightarrow InstrIndexListe \times MemPar \times Haltset$$
  This function executes its instruction and uses the information contained in *memInstr*. It modifies *memPar* and *instrIndexList* which gives the indices of instructions remaining to be executed.
- $\texttt{FindContinuation}_{\texttt{P}}$:
  $$Instruction \times Level \times Haltset \times InstrIndexList \times MemPar \times Haltset \longrightarrow InstrIndexList \times MemPar \times Haltset$$
  This function is used by $\texttt{Execute}_{\texttt{P}}$ to compute the continuations of `Parallel` instructions. It modifies the memory of `Parallel` instructions and returns a list of indices of instructions remaining to be executed.
- $\texttt{Reconstruct}_{\texttt{P}}$: $Haltset \longrightarrow MemPar \times InstrIndexList$
  This function computes the memory of `Parallel` and `Watchdog` instructions for them or for the new entry points. The reconstruction starts with the stopping points on which control is blocked after execution.
- $\texttt{Reconst}_{\texttt{P}}$:
  $$Instruction \times Index \times MemInstr \times InstrIndexList \longrightarrow MemInstr \times InstrIndexList$$
  This auxiliary function is used by $\texttt{Reconstruct}_{\texttt{P}}$.

Notation :

For a table M, we write $M[p \leftarrow val]$ is a new table equal to M but with the value corresponding to p becoming val.

$M_\perp$ designates a table with all entries undefined.

To finish, $()$ denotes the empty list, $+$ the concatenatio nof two lists and $\perp$ designates the undefined value.

C.2.2 Semantics

We define :

- $R \in InstrIndexList$ is the set of instructions remaining to be executed,
- $M_{instr} \in MemInstr$ is the memory of the instructions,
- $M_{par} \in MemPar$ is the memory of the `Parallel` instructions,
- $H \in Haltset$ is the set of stopping pints of the program,
- $Start \in InstrIndexList$ is the set of entry points for the execution at the following instant.

We define the execution of a program by the following algorithm :

- Initialisation at the first instant :
  - The initial instruction – `startpoint` is given by the IC program.
    $Start = (\texttt{startpoint})$
  - The memory of instructions is non-initialized.
    $M_{instr} = M_\perp$
- At each instant :

  1 Maximal executionex :

  1.1 $R = Start$, $M_{par} = M_\perp$ et $H = ()$

  1.2 as long as $R = (c_1, \ldots, c_n)$ is non-empty, choose i, $1 \leq i \leq n$ and compute
  $R, M_{par}, H = Executer_{\texttt{P}}\ \texttt{P}[c_i], (c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n), M_{instr}, M_{par}, H$

  2 Reconstruction :
  $M_{instr}, Start = Reconstruct_{\texttt{P}}\ H$

The execution in an instant is terminated ans the program is ready for the next instant.

Definition of $\texttt{Execute}_{\texttt{P}}$

We define $\texttt{Execute}_{\texttt{P}}$ as a function of the type of the instruction :

- A `Return` does nothing :
  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Return} : \texttt{r}]\!], R, M_{instr}, M_{par}, H = R, M_{par}, H$

- `Action` adds its continuation to the instructions remaining to be executed (we are not concerned here with the effect of the action on the environment) :

  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Action} : [\texttt{k}]\ \ (\texttt{c})]\!], R, M_{instr}, M_{par}, H = R + (\texttt{c}), M_{par}, H.$

- A `Watchdog` adds its associated instructions to the ones remaining to be executed :

  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} :\ \ \texttt{Watchdog} :\ \ \{\texttt{r}\}\ \ < \texttt{f} >]\!], R, M_{instr}, M_{par}, H =$
  $\quad R + M_{instr}[\texttt{n}], M_{par}, H$

- A `Fork` adds to the remaining instructions to be executed the instructions that it starts in parallel and it initializes the memory of the `Parallel` to which it is associated :

  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Fork} : \{\texttt{p}\}(\texttt{c}_1, \ldots, \texttt{c}_\texttt{m})]\!], R, M_{instr}, M_{par}, H =$
  $\quad R + (c_1, \ldots, c_m), M_{par}[\texttt{p} \leftarrow (\emptyset, \texttt{m}, 0)], H$

  Note : in M_{par} the exit level of the `Parallel` instruction is set to 0. It can also be initialized with a negative integer. In efect the exit levels given by the `Exit` and `Halt` instructions are integers which are positive or zero and calculation of the exit level of a `Parallel` instruction takes the maximum of the levels of its children. This note applies also for the `Parallel` instruction that follows.

- A `Parallel` adds its associated instructions to the ones remaining to be executed and initializes the triplet that corresponds to it in the memory of its `Parallel` :

  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} :\ \ \texttt{Parallel} :\ \ (\texttt{c\_0},\ \ \texttt{\textbackslash ldots},\ \ \texttt{c\_m})\ \ < \texttt{f} >]\!], R, M_{instr}, M_{par}, H =$
  $\quad R + M_{instr}[\texttt{n}], M_{par}[\texttt{n} \leftarrow (\emptyset, k, 0)], H$
  $\quad$ où k est le nombre d'éléments dans $M_{instr}[\texttt{n}]$

  Note : k is the number of instructions executed by the `Parallel`.

- An `Exit` transmits its exit level to the parent `Parallel` :

  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Exit} : \{\texttt{p}\}\ \ \texttt{e}]\!], R, M_{instr}, M_{par}, H =$
  $\quad \texttt{FindContinuation}_{\texttt{P}}\ \texttt{P}[\texttt{p}], \texttt{e}, \emptyset, R, M_{par}, H$

- A `Halt` transmits its exit level and its index. The index of the `Halt` is needed for the calculation of the set of stopping points that is used in the reconstruction :

  $\texttt{Execute}_{\texttt{P}}\ [\![\texttt{n} :\ \ \texttt{Halt} :\ < \texttt{f} >]\!], R, M_{instr}, M_{par}, H =$
  $\quad \texttt{FindContinuation}_{\texttt{P}}\ \texttt{P}[\texttt{f}], 1, \{\texttt{n}\}, R, M_{par}, H$

Note pour Esterel **:**

The definition of execution of other IC instructions is not given here. They add some difficulties that we voluntarily ignore in an effort to simplify. For example in the case of IC generrated by ESTERELwe don't handle the problems of causality. A resolution of these problems which takes away non-determinism of step 1.2 of the execution is proposed in the thesis of G. Gonthier by a calculation of signals potentially emitted and is implemented in the ESTEREL V3 compiler. We give only an idea of the execution of `Emit` and `Test` (of presentce of a signal) in the case of pure signals. The function $\texttt{Execute}_\texttt{P}$ takes and returns a extra argument : a set ofsignals in which a signal `S` that is present is written $\texttt{S}^+$ and `S` that is absent is written $\texttt{S}^-$. In order to simplify the reading, in `Test` instructions, the index of the test action is replaced with the name of the signal tested.

- The emission of a signal is not made if the signal has not already been fixed as being present or absent :

  $\texttt{Execute}_\texttt{P}\ [\![\texttt{n} : \texttt{Emit} : [\texttt{S}]\ \ (\texttt{c})]\!], R, M_{instr}, M_{par}, H, S =$
  $\quad R + (\texttt{c}), M_{par}, H, S \cup \{\texttt{S}^+\}$ si $\texttt{S}^+ \notin S$ et $\texttt{S}^- \notin S$.

- If the signal emitted is fixed, present or absent, the `Emit` instruction is not executed. It stays in R and the execution of step 1.2 will never terminate. We could also return an execution error :

  $\texttt{Execute}_\texttt{P}\ [\![\texttt{n} : \texttt{Emit} : [\texttt{S}]\ \ (\texttt{c})]\!], R, M_{instr}, M_{par}, H, S =$
  $\quad R + (\texttt{n}), M_{par}, H, S$ si $\texttt{S}^+ \in S$ ou $\texttt{S}^- \in S$.

- A `Test` adds the continuation *true* of the signal is fixed as present :

  $\texttt{Execute}_\texttt{P}\ [\![\texttt{n} : \texttt{Test} : [\texttt{S}]\ \ (\texttt{ct}, \texttt{cf})]\!], R, M_{instr}, M_{par}, H, S =$
  $\quad R + (\texttt{ct}), M_{par}, H, S$ si $\texttt{S}^+ \in S$.

- A `Test` adds the continuation *false* if the signal is fixed and absent :

  $\texttt{Execute}_\texttt{P}\ [\![\texttt{n} : \texttt{Test} : [\texttt{S}]\ \ (\texttt{ct}, \texttt{cf})]\!], R, M_{instr}, M_{par}, H, S =$
  $\quad R + (\texttt{cf}), M_{par}, H, S$ si $\texttt{S}^- \in S$.

- The presence test on a non-fixed signal fixes it as absent :

  $\texttt{Execute}_\texttt{P}\ [\![\texttt{n} : \texttt{Test} : [\texttt{S}]\ \ (\texttt{ct}, \texttt{cf})]\!], R, M_{instr}, M_{par}, H, S =$
  $\quad R + (\texttt{cf}), M_{par}, H, S \cup \{\texttt{S}^-\}$ si $\texttt{S}^+ \notin S$ et $\texttt{S}^- \notin S$.

Definition of $\texttt{FindContinuation}_\texttt{P}$

This function is defined on the `Return`, `Watchdog` and `Parallel` instructions and is not called except on `Halt` or `Exit`.

- $\texttt{FindContinuation}_\texttt{P}\ [\![\texttt{p} :\ \texttt{Parallel} :\ (\texttt{c\_0},\ \texttt{\textbackslash ldots},\ \texttt{c\_m})\ < \texttt{f} >]\!], e, h, R, M, H = R', M', H$

  Posons $M'' = M[\texttt{p} \leftarrow (M[\texttt{p}].haltset \cup h, M[\texttt{p}].active - 1, max(M[\texttt{p}].level, e))]$. Three cases can be distinguished :

(1) all the sons of the `Parallel` are not terminated : so it is the case that $M[\texttt{p}].active \neq 1$ which is therefore decremented to 1. The set of `Halt` h given by `FindContinuationP` is added to the set of `Halt` instructions on which the `Parallel` is blocked. The parameter e of `FindContinuationP` allows finding the exit level of the `Parallel`.

(2) the `Parallel`is terminated and is bocked on at least one `Halt` and `Exit` with level 0. The `Parallel` signals therefore to its parent that its termination level is 1 and it transmits ot it the set of `Halt` instructions on which it is blocked (calculated like in case 1) : it calls `FindContinuationP`. This is the only one of the three cases in which there is a recursive call to this function.

(3) the `Parallel`is terminated and is blocked in only one `Exit` with level 0, or on at least one `Exit` with a level greater or equal to 2. The `Parallel` selects therefore a continuation as a function of its exit level : the instruction corresponding to this continuation is added to the remaining instructions R to execute.

Formally :

(1) $R', M' = R, M''$ si $M''[\texttt{p}].active \neq 0$,

(2) $R', M' = \texttt{FindContinuationP}\ \texttt{P}[\texttt{f}], 1, M''[\texttt{p}].haltset, R, M''$
if $M''[\texttt{p}].active = 0$ and $M''[\texttt{p}].level = 1$,

(3) $R', M', H = R + \texttt{c}_{max(M''[\texttt{p}].level-1,0)}, M'', H$
if $M''[\texttt{p}].active = 0$, $M''[\texttt{p}].level \neq 1$ et $M''[\texttt{p}].level \leq \texttt{m} + 1$.

Recall that `m` is the number of continuations minus one of the `Parallel` given in $[\![\texttt{p}: \ \texttt{Parallel}: \ (\texttt{c\_0}, \ \texttt{\textbackslash ldots}, \ \texttt{c\_m}) \ < \texttt{f} >]\!]$.

- When `FindContinuationP` is called on a `Return`, it is known that all `Parallel` instructions are terminated and have no continuation selection (see case 2 of `Parallel`) : the execution is finished. The set o fsetopping points is contained in H :

  $\texttt{FindContinuationP}\ [\![\texttt{n}: \texttt{Return}: \texttt{r}]\!], e, h, R, M, H = (), M, H \cup h$

- The `Watchdog` instructions are simply :

  $\texttt{FindContinuationP}\ [\![\texttt{n}: \ \texttt{Watchdog}: \ \{\texttt{r}\} \ < \texttt{f} >]\!], e, h, R, M, H =$
  $\qquad \texttt{FindContinuationP}\ \texttt{P}[\texttt{f}], e, h, R, M, H$

Note for ESTEREL **:** a `Parallel` handle the continuations possible as a function of the exit levels of its children. In this case it transmits its exit level to the first `Parallel` that encloses it. Using the preceding notation :

$R', M', H = \texttt{FindContinuationP}\ \texttt{P}[\texttt{f}], max(M[\texttt{p}].level, e) - m, \emptyset, R, M'', H$
si $M[\texttt{p}].active = 1$ et $max(M[\texttt{p}].level, e) > m + 1$.

Definition of $\mathtt{Reconstruct_P}$

- $\mathtt{Reconstruct_P}\ \{h_1, \ldots, h_n\} = M_n, Start_n$ computes step by step with the aid of the auxiliary function $\mathtt{Reconst_P}$:

$$\begin{aligned} M_1, Start_1 &= \mathtt{Reconst_P}\ \mathtt{P}[h_1], h_1, M_\perp, () \\ \forall i, 1 < i < n, M_i, Start_i &= \mathtt{Reconst_P}\ \mathtt{P}[h_i], h_i, M_{i-1}, Start_{i-1} \end{aligned}$$

Definition of $\mathtt{Reconst_P}$

The function $\mathtt{Reconst_P}$ encodes precisely the occurrences in the execution examples.

- The index of a `Halt` is added to the instructions associated with its parent :

  $\mathtt{Reconst_P}\ [\![\mathtt{h} :\ \ \mathtt{Halt} :\ <\mathtt{f}>]\!], index, M, Start = \mathtt{Reconst_P}\ \mathtt{P}[\mathtt{f}], \mathtt{h}, M, Start$

  Note that the *index* passed as an argument is not used.

- The index of a `Parallel` is added to the instructions associated with its parent :

  $\mathtt{Reconst_P}\ [\![\mathtt{p} :\ \ \mathtt{Parallel} :\ \ (\mathtt{c\_0},\ \ \backslash\mathtt{ldots},\ \ \mathtt{c\_m})\ \ <\mathtt{f}>]\!], index, M, Start =$
  $\qquad \mathtt{Reconst_P}\ \mathtt{P}[\mathtt{f}], \mathtt{p}, M[\mathtt{p} \leftarrow M[\mathtt{p}] + (index)], Start$

- The *garde* field of a `Watchdog` is added to the instructions associated with its parent :

  $\mathtt{Reconst_P}\ [\![\mathtt{n} :\ \ \mathtt{Watchdog} :\ \ \{\mathtt{r}\}\ \ <\mathtt{f}>]\!], index, M, Start =$
  $\qquad \mathtt{Reconst_P}\ \mathtt{P}[\mathtt{f}], \mathtt{r}, M[\mathtt{n} \leftarrow M[\mathtt{n}] + (index)], Start$

- A `Return` stops the recursion and adds the index that it received into the set of stopping points.

  $\mathtt{Reconst_P}\ [\![\mathtt{n} : \mathtt{Return} : \mathtt{r}]\!], index, M, Start = M, Start + (index)$

C.2.3 With Respect to the Reconstruction

In the preceding, it seems that the reconstruction computes *dynamically* a large amount of information. In fact, in applying $\mathtt{Reconstruct_P}$ to the set of `Halt` instructions of program `P`, one obtains for `Watchdog` and `Parallel` a *static* set of associated instructions. In the reconstruction presented above, the set of associated instructions was calculated dynamically, always being a subset of the corresponding static set. In fact the reconstruction *selects* from a static set ofinstructins which are activated in the following instant.

Appendix D

The Semantics of GC

D.1 Complete Datafow Model of GC

D.1.1 Preorders

Execution Preorders, executions

Here we want to model the following objects ; a finite set of 'ports' where each port is a "flow," which is to day that it is a succession of values, a behavior of this set of ports, and a (pre)ordering that describes the order of availability given to all these values.

Ports Given an enumerable set S of *occurrences* ; each occurence in S is given a tag $X \in \mathbf{X}$, where $\mathbf{X}$ is a *finite* set of *ports*. $X(s)$ designates the port corresponding to the occurrence s, and $S(X)$ designates the set of occurrences of the port X.

Time For all ports X, the set $S(X)$ is ordered in a *succession* that we write as $X_1, X_2, X_3, \ldots$, where $n = 1, 2, 3, \ldots$ are the "time indices." The set of occurrences X_n where X ranges ofer the set of ports, is called the *n-th event.* [1] A *clock* is a subset of the set of natural integers. The set of integers $\mathbf{N}$ itself will be called the "*base* clock. One can therefore define on clocks the usual set of operations:

$$
\begin{array}{ll}
K = G \cup H & \text{union} \\
K = G \cap H & \text{intersection} \\
K = G \ominus H & \text{complement of } H \text{ dans } G
\end{array}
$$

Synchronous Flows We now model the values taken by the ports. At each point $s \in S$ a value $v(s) \in V(X)$, where x is the port associated with s and

[1] Note that these time indices make no reference to any physical notion of time. They describe only an abstract numbering.

$V(X)$ is a domain of values (of type X), and on it is put $V = \coprod_{X \in \mathbf{X}} V(X)$, where $\coprod$ designates the disjoint union. We will often use the notation v_n^X to designate $v(X_n)$. The *synchronous flow* (or more simply *flow* in the following) taken by the port X is the succession of values taken by the occurrences of the port X[2]. All the domains $V(X)$ are supposed to contain a distinguished element, written $\perp$, which is interpreted as a status "absent;" conversely if s take a value different than $\perp$, it must be said to be "present."[3] For $X \in \mathbf{X}$, the *clock* of X written H_X is the subsequence $(n_k)_{k=1,2,\ldots}$ numbering the present occurrences on the port X. It will be easy to identify clocks and "pure" flows, which is to say the flows whose domain is reduced, except for $\perp$, having a unique value that we can also write as $\top$, and which symbolizes presence. A clock defined by a pure flow is one defined by its instants of presence.

(Pre)orderings We define a *preorder* on S as having the information of a relation $\preceq$, reflexive, and transitive on a set of occurrences S. The information of a directed graph (admittedly possibly of circuits) whose ends belong to S induces a unique preorder, by a reflexive and transitive closure. It must be determined that $\{S, \mathbf{X}, \preceq\}$ satisfies the following conditions of *causality* :

$$\forall X_n, Y_m \in S \ : \ X_n \preceq Y_m \ \Rightarrow \ n \leq m \tag{D.1}$$

(dependencies cannot go back in time) and of *succession* :

$$\forall X \in \mathbf{X}, \ n \leq m \ \Rightarrow \ X_n \preceq X_m \tag{D.2}$$

On the other hand, it could be that $s \preceq s'$ and $s' \preceq s$ on the condition that s and s' are the same event.

Executions We will call an *execution* a tuple $\{S, \mathbf{X}, \preceq, v\}$ satisfying the conditions (D.1,D.2) introduced above. The executions will be generically designated by the notation $\mathcal{E}$.

INTERPRETATION : an execution describes at the same time a behavior (a succession of events consisting of the assignment of values to the set of ports participating in the event) and some directives for the order of evaluation of these values (the relation $s \preceq s'$ interpreted as "s' can not be evaluated before s"; in particular, $\{s \preceq s'$ and $s' \preceq s\}$ described the evaluatin of $\{s, s'\}$ as an atomic action — non-interruptible –).

[2]The notion of "port" does not appear in GC, this role is taken by the name of the flow.

[3]The notion of synchronous flows is different that that the classically accepted data flow model by virtue of this particular value $\perp$ denoting the absence; a notion that is unknown in dataflow models. The term "trace" could be a more appropate one for synchronous flows, but we take the terminiology already in use.

Relation of the Order $\leq_{\mathcal{E}}$ On Executions and the Set of Executions

We say that

$$\mathcal{E}_1 = \{S_1, \mathbf{X}_1, \preceq_1, v_1\} \quad \leq_{\mathcal{E}} \quad \{S_2, \mathbf{X}_2, \preceq_2, v_2\} = \mathcal{E}_2$$

($\mathcal{E}_1$ is a preorder tagged "less partial" than $\mathcal{E}_2$) if the following conditions are satisfied ;

1. $S_1 = S_2$; $\mathbf{X}_1 = \mathbf{X}_2$; $v_1 = v_2$ [4]
2. $\preceq_2 \subseteq \preceq_1$, in other words $\preceq_1$ is a reinforcement of $\preceq_2$.

In other words said, the two execution have the same set of occurrences, same ports, same tagging functions, and only differ by their preorders ; and the preorder of the second contains *fewer* branches than the first one (and, note, not the inverse). It can be easily verified that $\leq_{\mathcal{E}}$ is an order.

This ordering relation on the execution is defined for *sets* in the following fashion : if Π_1 and Π_2 designate two such sets, one writes

$$\Pi_1 \ \leq_{\mathcal{E}} \ \Pi_2 \tag{D.3}$$

if

$$\exists\ \varphi \ : \ \Pi_1 \mapsto \Pi_2\ ,\ \varphi \text{ injective, such that, } \forall \mathcal{E}_1 \in \Pi_1,\ \mathcal{E}_1 \leq_{\mathcal{E}} \varphi(\mathcal{E}_1)$$

It can be verified again that (D.3) a well order on the sets of executions. More, it can be easily verified that all families of well ordered executions admitting a greater-than has a largest element. This we use in the definition of program composition.

D.1.2 Programs and Their Combinators

Given two executions $\mathcal{E}$ and $\mathcal{E}'$, we say that $\mathcal{E}'$ is a *dilatation* of $\mathcal{E}$, or, in an equivalent manner, that $\mathcal{E}$ is a *compression* of $\mathcal{E}'$, if $\mathcal{E}$ is obtained by deleting in $\mathcal{E}'$ a finite set of *silent* events. This is to say the events where all the ports are absent, and taking in it the restriction to $\mathcal{E}$ the orders and preorders defined on $\mathcal{E}'$. We say that $\mathcal{E}'$ is a *strict* dilation of $\mathcal{E}$ if even stronger, $\mathcal{E}'$ is different than $\mathcal{E}$. We say that a set of executions is *saturated by compression* if all executions contain no silent instant in the strictly compressed set. Conversely, we say that a set of executions is *saturated by dilation* if all executions allow, for all n, a strict dilation obtained by insertion a silent event between the n-th and the $(n+1)$th events.

[4] strictly, it is not necessary that $S_1 = S_2$, in the measure where the set of occurrences S is defined by a bijection, is an abuse of the language.

Programs

Definition 1 (programs) *A program on* $\mathbf{X}$ *is given by a set of executions on* $\mathbf{X}$*, saturated by dilation and compression. Programs will be generically designated by* Π*. The ordering relation* $\leq_{\mathcal{E}}$ *defined on the set of executions defining therefore an ordering relation on programs.*

Restriction and Composition

Given $\mathcal{E} = \{S, \mathbf{X}, \preceq, v\}$ an execution on $\mathbf{X}$, and given $\mathbf{Y} \subset \mathbf{X}$. The *restriction* of $\mathcal{E}$ to $\mathbf{Y}$ will be an execution $\mathcal{E}' = \{S', \mathbf{Y}, \preceq', v'\}$ on $\mathbf{Y}$, where

- S' is the restriction of S to occurrences of port $Y \in \mathbf{Y}$;

- $\preceq', v'$ are the restrictions of $\preceq, v$ à S'.

Given Π a program on $\mathbf{X}$, and given $\mathbf{Y} \subset \mathbf{X}$. The set of restrictions to $\mathbf{Y}$ of executions of Π definThats, after saturation by compression, a new program on $\mathbf{Y}$, that we will call the *restriction* of Π to $\mathbf{Y}$ and that we will write as

$$\Pi_{!!\mathbf{Y}}$$

Given now, for $i = 1, 2$, Π_i a program on $\mathbf{X}_i$. We will call a *composition* of Π_1 and Π_2, written

$$\Pi_1|\Pi_2$$

the largest (for the order defined on programs) program on $\mathbf{X_1} \cup \mathbf{X_2}$ whose restriction to $\mathbf{X}_1$ (respectively $\mathbf{X}_2$) is $\leq_{\mathcal{E}} \Pi_1$ (respectively $\leq_{\mathcal{E}} \Pi_2$), where $\leq_{\mathcal{E}}$ is the partial order defined on programs.

INTERPRETATION That which concerns the "behaviors," which is to say, the legal sets of values of a program, the order defined on programs coincides with inclusion; that which concerns ordering, it coincides with weakening of themselves. By consequence, composing two programs becomes :

- with respect to the behavior, take the intersections of the two programs restricted to common ports;

- with respect to the ordering, impose the ordering which results from one or the other of the two programs.

D.1.3 Behavioral Specifications, Synchronisation Specifications, Execution Schemas

Behavioral Specifications

Given $\mathcal{E} = \{S, \mathbf{X}, \preceq, v\}$ an execution on $\mathbf{X}$. we associate it with a *behavior* $\{S, \mathbf{X}, v\}$ obtained simply by suppressing the relation $\preceq$. In the general style, we will call *behavioral* on $\mathbf{X}$ a t-uple $T = \{S, \mathbf{X}, v\}$ satisfying the conditions defined above for these objects.

Definition 2 (behavioral specification) *A behavioral specification (or just specification) on* $\mathbf{X}$ *is a set of behaviors on* $\mathbf{X}$*, saturated by dilation and compression.*

We will write specifications generically as Σ, and we would define, just as for programs, *restriction*

$$\Sigma_{!!\mathbf{Y}}$$

where $\mathbf{Y} \subset \mathbf{X}$ and the *composition* [5]

$$\Sigma_1 | \Sigma_2$$

In this style, all programs Π are associated with a unique specifications that we will write

$$\Sigma(\Pi) \tag{D.4}$$

and such that

$$\begin{aligned} \Sigma\left(\Pi_{!!\mathbf{Y}}\right) &= \Sigma(\Pi)_{!!\mathbf{Y}} \\ \Sigma\left(\Pi_1 | \Pi_2\right) &= \Sigma\left(\Pi_1\right) | \Sigma\left(\Pi_2\right) \end{aligned}$$

In other words $\Pi \mapsto \Sigma(\Pi)$ is a homomorphism.

Synchronization Specifications

Given $T = \{S, \mathbf{X}, v\}$ a behavior on $\mathbf{X}$. We associate it with a *synchronization* $\{S, \mathbf{X}, h\}$ which is obtained by composing the tagging function v with the application that leaves $\perp$ invariant and which, for all values present, associates the symbol $\top$ (the values are lost, but the information concerning the clock is preserved). In this style, we will call a *synchronization* on $\mathbf{X}$ a t-tuple $\{S, \mathbf{X}, h\}$ satisfying the conditions defined above for these objects.

Definition 3 (synchronization specification) *A synchronization specification on* $\mathbf{X}$ *is a set of synchronizations on* $\mathbf{X}$ *saturated by dilation and compression.*

[5] so called because for specifications, the order retained is the one of inclusion.

We will in general write synchronization specifications as $\mathcal{H}$ and we would define, as for programs, *restriction*

$$\mathcal{H}_{!!\mathbf{Y}}$$

where $\mathbf{Y} \subset \mathbf{X}$, and *composition* as

$$\mathcal{H}_1 | \mathcal{H}_2$$

In this way, all programs Π are associated with a unique synchronization specification so that we will write

$$\mathcal{H}(\Pi)$$

and have for it

$$\begin{array}{rcl} \mathcal{H}\left(\Pi_{!!\mathbf{Y}}\right) & = & \mathcal{H}(\Pi)_{!!\mathbf{Y}} \\ \mathcal{H}\left(\Pi_1 | \Pi_2\right) & = & \mathcal{H}\left(\Pi_1\right) | \mathcal{H}\left(\Pi_2\right) \end{array}$$

In other words, $\Pi \mapsto \mathcal{H}(\Pi)$ is a homomorphism.

Execution Schemas

Given $\mathcal{E} = \{S, \mathbf{X}, \preceq, v\}$ an execution on $\mathbf{X}$, we associated with it an ordering $\mathcal{G} = \{S_{\text{pres}}, \mathbf{X}, \preceq, v\}$ obtained simply by suppressing the occurrences $s \in S$ having the value $\bot$. It can be noted that one must thus forget the notion of simulteneity which is based on the notion of global execution time and the causality property of $\preceq$ with respect to this global time.

Definition 4 (execution schemas) *We will call an execution schema on* $\mathbf{X}$ *a set* Γ *of orderings on* $\mathbf{X}$.

To all programs Π is therefore associated, as indicated above, a unique execution schema that we will write as

$$\Gamma(\Pi)$$

NOTE: *We do not provide the execution schemes for the combinators* $(.)_{!!}$ *(restriction) or* $|$ *(composition), because the application* $\Pi \mapsto \Gamma(\Pi)$ *would not be a homorphism.* In effect considering the two following executions

$$\begin{array}{lccccccccccccccccc} \mathtt{H:} & \bot & \to & \top & \to & \bot & \to & \top & \to & \bot & \to & \top & \to & \bot & \to & \top & \to & \ldots \\ & & & \downarrow & & & & \downarrow & & & & \downarrow & & & & \downarrow & & \ldots \\ \mathtt{X:} & 1 & \to & 2 & \to & 3 & \to & 4 & \to & 5 & \to & 6 & \to & 7 & \to & 8 & \to & \ldots \end{array}$$

$$\begin{array}{lccccccccccccccccc} \mathtt{H:} & \top & \to & \bot & \to & \top & \to & \bot & \to & \top & \to & \bot & \to & \top & \to & \bot & \to & \ldots \\ & & \searrow & & & & \searrow & & & & \searrow & & & & \searrow & & & \ldots \\ \mathtt{X:} & 1 & \to & 2 & \to & 3 & \to & 4 & \to & 5 & \to & 6 & \to & 7 & \to & 8 & \to & \ldots \end{array}$$

where `X,H` are ports, and where the value of the occurrences have simply been written as the placement of these occurrences. They are different, but they define the same preorder tag after erasing the $\perp$. As a consequent, if one considers the programs Π_1 and Π_2 obtained in saturating these executions by dilation (and accepting for example all the orderings for the silent events), $\Pi_1|\Pi_2$ is the program that does not contain any silent events, therefore $\Gamma(\Pi_1|\Pi_2) = \emptyset$. In return, $\Gamma(\Pi_1) = \Gamma(\Pi_2) = \Gamma(\Pi_1)|\Gamma(\Pi_2)$ is not empty [6].

INTERPRETATION : It can be noted that $\Gamma(\Pi)$ is a description in partial orders of the program Π, and can therefore be interpreted as a *purely asynchronous* execution specification : all asynchronous output (in terms of unbounded memory, *rendez-vous* etc) respecting this specification will be "correct" in a sense that that we will define soon.

D.2 Model For the "Imperative Part"

The model of the preceding section is completely rigorous. The one in this section is simply to furnish a means of easily defining the programs in the sense of the preceding section. The principle used then is the following: we will consider now programs enriched with an "imperative part", which provides for us essentially a way of easily speaking about the past (such as that which concerns values with respect to orderings). This imperative part will be purely local to the program, and does not therefore provide for operations of restriction and communication; we don't redefine thefore how the combinators act on the enriched programs and their imperative part since they don't act on the program in the same sense as the preceding section. Let's see that by taking the objects introduced in Section D.1.1 one by one one.

D.2.1 Execution with The Imperative Part

Variables The set of occurrences was widened: it contains now the occurrences s having a new type of tag $\xi \in \Xi$, where Ξ is a finite set of *variables* and the same notations are used for these variables as for ports: $\xi(s), S(\xi)$. The relative notions of time are understood for variables. The set $S(\xi)$ of occurrences of ξ is ordered in a succession, with this exception that *the same variable can admit a number greater than one (but nevertheless finite) of occurrences inside an event.* As a consequence, the occurrences of variables are written $\xi_{1,1}, \xi_{1,2}, \xi_{1,3}, \xi_{2,1}, \xi_{2,2}, \dots$, according to lexicographic order where the first index indicates the event and the second the place in the event.

[6] One other reason to not supply the execution scheme for composition | is that one can not increase it while preserving rationality.

(Pre)orders Suppose that the preorder $\preceq$ is defined on the set of occurrences, that they must be associated either to ports or variables ; the conditions (Equations D.1 and D.2) are equally defined and taken therefore according to the lexicographic numbering, the respective forms being:

$$\forall X_{n_1,p_1}, Y_{n_2,p_2} \in S \quad : \quad X_{n_1,p_1} \preceq Y_{n_2,p_2} \;\Rightarrow\; n_1 \leq n_2 \tag{D.5}$$
$$\forall \xi \in \Xi \quad , \quad (n_1, p_1) \leq (n_2, p_2) \;\Rightarrow\; \xi_{n_1,p_1} \preceq \xi_{n_2,p_2} \tag{D.6}$$

In the first condition, the occurrences are of any type (port or variable), and the standard numbering in n for ports is identified by the double numbering $(n, 1)$ where the supplementary numbering of the compound is 1. It can be noted that (Equation D.5) signifies that the time can not be rewound, the second part of the compound on the other hand can be.

Values The valuation function v is defined on occurrences of variables and we will write in particular $v^{\xi}_{1,1}, v^{\xi}_{1,2}, v^{\xi}_{1,3}, v^{\xi}_{2,1}, \ldots$ to designate $v(\xi_{1,1}), v(\xi_{1,2}), v(\xi_{1,3}), v(\xi_{2,1}), \ldots$, which are the values of the successive occurrences of ξ. Like the preceding, if s is an occurrence of ξ, $v(s) = \perp$ signifies that the variable ξ is not accessible in the occurrence considered. The set of instances of presence of ξ is the clock of ξ and is written H_ξ. All valuation functions v associated with a variable enjoy the following property (which justifies the name "variable") : *there exists a sub-sequence* $S_{\text{write}}(\xi) \subset S(\xi)$*, such that*

$$\begin{aligned} \forall s \in S_{\text{write}}(\xi) \quad &\Rightarrow \quad v(s) \neq \perp \\ \forall s \in S(\xi) \ , \ v(s) \neq \perp \quad &\Rightarrow \quad v(s) = v\left(\max_{s' \in S_{\text{write}}(\xi)} s'\right) \end{aligned} \tag{D.7}$$

where the *max* makes reference to the preorder $\preceq$ on the execution under consideration. Also by definition, we say that $S_{\text{write}}(\xi)$ constitutes the set of *write occurrences* of the variable ξ. The condition (D.7) signifies that between its written occurences the variable ξ is, always absent, or always non-modified. It can't be helped that an event consists of many occurrences of writing to the same variable.

Intuitive Execution We present now the *executions with the imperative part*, which are tuples of the form $\{S, \mathbf{X} \cup \Xi, \preceq, v\}$ where the new object introduced is the set Ξ of variables. The executions with an imperative part will be generically written as $\mathcal{I}$. By definition, to all $\mathcal{I}$ we can associate an execution $\mathcal{E}$ by simply erasing the occurrences of variables and taking the restriction of the preorder. Leaving to a set of $\mathcal{I}$ one obtains thus the set of $\mathcal{I}$.

D.2.2 Programs with an Imperative Part

Definition 5 (programs with an imperative part) *A program with an imperative part on* $\mathbf{X} \cup \Xi$ *is given by a set of executions with the imperative part on*

prog	::=	$\boxed{\texttt{metagc}}$ Prog_Name $\boxed{\texttt{act}}$ List_Act_Name $\boxed{\texttt{var}}$ List_Var_Name Body $\boxed{\texttt{end}}$ Prog_Name
Body	::=	Body $\boxed{\texttt{\|}}$ Body $\mid$ Clockeq $\mid$ Preceq $\mid$ Floweq $\mid$ Imper $\mid$ $\boxed{\texttt{(}}$ Body $\boxed{\texttt{)}}$
Clockeq	::=	$\boxed{\texttt{clock}}$ $\boxed{\texttt{(}}$ Name $\boxed{\texttt{)}}$ $\boxed{\texttt{=}}$ Clock $\mid$ Clock $\boxed{\texttt{=}}$ Clockexp
Clockexp	::=	Clock Clockop Clock $\mid$ $\boxed{\texttt{tt}}$ $\boxed{\texttt{(}}$ bool $\boxed{\texttt{)}}$
Clockop	::=	$\boxed{\texttt{default}}$ $\mid$ $\boxed{\texttt{when}}$ $\mid$ $\boxed{\texttt{whennot}}$
Preceq	::=	Name $\boxed{\texttt{-->}}$ Name $\boxed{\texttt{at}}$ Clock
Name	::=	Flow_Name $\mid$ Act_Name
Floweq	::=	Flow_Name $\boxed{\texttt{=}}$ Flowexp $\boxed{\texttt{at}}$ Clock
Imper	::=	Act_Name $\boxed{\texttt{:}}$ Act
Act	::=	input $\mid$ output
input	::=	Var_Name $\boxed{\texttt{=}}$ Flow_Name $\boxed{\texttt{at}}$ Clock
output	::=	Flow_Name $\boxed{\texttt{=}}$ Varexp $\boxed{\texttt{at}}$ Clock

Table D.1: *The Core Language* μGC

$\mathbf{X} \cup \Xi$ *which generates, by erasing the actions and variables, a set of executions on* $\mathbf{X}$*, saturated by dilation and compression.*

It should be noted that we don't intend for programs with an imperative part to have either partial order $\leq_{\mathcal{E}}$, the combinators "restriction" or "composition." In effect the imperative parts *play no role in these operations,* and they are applied exclusively to programs inferred by erasing their variables. In fact, as one can see, variables are not there to facilitate the expression of dependences and relation between flows bearing on different instants.

D.3 Use of the Model for the Semantics of GC

D.3.1 μGC

The table D.1 describes the metalanguage following, answering to the simple name μGC, which we will use directly to encode GC. The key-words of the language are encoded. Besides the keywords, the terminals of this language are, in the order seen, "Clock" (a clock), "bool" (a boolean flow), "Flow_Name" (flow name), “Act_Name” (action name), “Var_Name” (variable name), “Flowexp” (flow expression), “Var” (variable), “Varexp” (variable expression).

The language is very informally written, in the way where terminals are in all upper case. For example, small μGC program following is equivalent to `pre` in GC :

```
metagc PRE                            -- name of the program

  act STATE, OUTPUT                   -- declaration of local actions
  var Xi                              -- declaration local variables

    ( ( clock(IN)     = H             -- clock equations
      | clock(OUT)    = H
      | clock(STATE)  = H
      | clock(OUTPUT) = H )
    | ( OUTPUT --> OUT   at H         -- dependenciess
      | OUTPUT --> STATE at H
      | IN     --> STATE at H
      | H      --> IN    at base
      | H      --> OUT   at base )
    | ( STATE  :  Xi = IN at H        -- actions
      | OUTPUT :  OUT = Xi at H ) )

end PRE
```

In order to be more precise here, we give the semantics of μGC equations (the declarations are but a syntax sugar that we allow).

The first question that we examine is the semantics of actions, in the way where this notion does not exist in our abstract model. Consider a μGC program. Regrouping all the actions shared by the same variable ξ (for example the actions `STATE` and `OUTPUT` share the variable `Xi` in the program `PRE`). These actions define the subsets of the set of occurrences of the variable ξ, and for distinct actions these subsets are distinct. the clock equations `clock(STATE) = ...` etc., specify the instance of presence of the referenced action, whereas the clock of presence occurrences of the variable ξ is the union of clocks of actions sharing that variable. By definition, when two such actions share the same instant of presence, that creates *two* occurrences of the variable ξ at that instant, with two possible interlacings (action (a) before action (b) or the invverse). The recourse to orderings on the actions allows, if desired, to fix the numbering desired. Thus in teh program `PRE` , the variable `Xi` is shared by the two actions `STATE` and `OUTPUT` , which have the same clock `H` . The variable `Xi` allows therefore, two occurrences of presence, to each instant of presence of `H` . The dependenacy “ `OUTPUT --> STATE at H` ” orders tehse actions at each instanct of `H` . In the present case, it specifies that the emission of the current value of ξ (action

`OUTPUT`) must be made before the writing on ξ the new value of `IN` (action `STATE`).

Now we verify that we have obtain the definition of `pre` in GC. At this point, it will be easy to introduce the index n numbering the events of presence of the clock `H` , the unique clock of all the flows or actions of the program. The program `PRE` above translates thus in writing x for `IN` and y for `OUT` :

$$\begin{aligned} \forall n > 0 &: x_{n-1} \rightarrow \xi_{n-1,2} \rightarrow \xi_{n,1} \rightarrow y_n \\ \forall n > 0 &: y_n = \xi_{n,1} = \xi_{n-1,2} = x_{n-1} \end{aligned}$$

If the variable ξ is erased, it becomes exactly

$$\begin{aligned} \forall n > 0 &: x_{n-1} \rightarrow y_n \\ \forall n > 0 &: y_n = x_{n-1} \end{aligned}$$

which is exactly the definition of `pre` in GC. We will formalize all this presently

Declaration of Actions and Variables

- The declaration

  ```
  metagc PROG

    act ACT_1,...,ACT_p;...
    var Xi
  ```

  where `ACT_1,...,ACT_p` is teh set of actions shared by the variable `Xi` , creating a variable ξ possessing p occurrences at each event. We will designate by $\xi_{n,\texttt{ACT\_1}}, \ldots, \xi_{n,\texttt{ACT\_p}}$ these occurrences to the n-th event. Warning, the order of these occurrences is not specified (the numbering of the actions used here is not a common notation) without an explicit ordering using the `-->` instruction introduced following.

Clock equations This part of the semantics corresponds to the definitions

Clockeq	::=	$\boxed{\texttt{clock}}\ \boxed{\texttt{(}}$ Name $\boxed{\texttt{)}}\ \boxed{\texttt{=}}$ Clock $\mid$ Clock $\boxed{\texttt{=}}$ Clockexp
Clockexp	::=	Clock Clockop Clock $\mid$ $\boxed{\texttt{tt}}\ \boxed{\texttt{(}}$ bool $\boxed{\texttt{)}}$
Clockop	::=	$\boxed{\texttt{default}}$ $\mid$ $\boxed{\texttt{when}}$ $\mid$ $\boxed{\texttt{whennot}}$

- `H = K default/when/whennot L` corresponds to the clock equations $H = K \cup / \cap / \ominus L$
- `clock(X) = H` corresponds go $H_X = H$
- `H = tt(B)` designates the set of executions on the ports `(H,B)` , where `H` is the pure type and `B` is boolean, such that $v_n^H = \top$ if and only if $v_n^B = tt$.

Dependencies. We give the semantics of

$$\begin{array}{rcl} \text{Preceq} & ::= & \text{Name}\ \boxed{\texttt{-->}}\ \text{Name}\ \boxed{\texttt{at}}\ \text{Clock} \\ \text{Name} & ::= & \text{Flow\_Name} \ \big|\ \text{Act\_Name} \end{array}$$

- `X --> Y at H` is here defined in teh case where, for example, `X` is an action and `Y` is a port. Given $\xi^1, \ldots, \xi^p$ the variables invoked in the action `X` . So, this instruction creates, for all instants N of presence of the clock `H` , and for $\xi^i_{n,\texttt{X}} \preceq Y_n$, where $\xi^i_{n,\texttt{X}}$ are the occurrences of the variable ξ associated with the action `X` in the event n under consideration. The other cases are defined similarly.

- `(X_1,...,X_p) --> (Y_1,...,Y_q) at H` is an abriged notation for `X_i --> Y_j at H` for all pairs of indices `(i,j)` .

Flow Equations These correspond to the following part of μGC:

$$\text{Floweq} \ ::= \ \text{Flow\_Name}\ \boxed{\texttt{=}}\ \text{Flowexp}\ \boxed{\texttt{at}}\ \text{Clock}$$

- `X = f(Y,Z) at H` is the set of executions on `(X,Y,Z,H)` , where `H` is a clock and `X,Y,Z` are ports of the appropriate type satisfying the clocking condition $H \subseteq (H_X \cap H_Y \cap H_Z)$ and the condition $v^X_n = f(v^Y_n, v^Z_n)$ for all instants n of presence of the clock H.

Imperative Part This corresponds to the following part of μGC: Ceci correspond à la partie suivante de

$$\begin{array}{rcl} \text{Imper} & ::= & \text{Act\_Name}\ \boxed{\texttt{:}}\ \text{Act} \\ \text{Act} & ::= & \text{input} \ \big|\ \text{output} \\ \text{input} & ::= & \text{Var\_Name}\ \boxed{\texttt{=}}\ \text{Flow\_Name}\ \boxed{\texttt{at}}\ \text{Clock} \\ \text{output} & ::= & \text{Flow\_Name}\ \boxed{\texttt{=}}\ \text{Varexp}\ \boxed{\texttt{at}}\ \text{Clock} \end{array}$$

We have already studied the declaration of actions. It remains for us thefore to detail the "Act."

- `IN_ACT : Xi = Z at H` creates for all events n of the presence of `H` , and for the occurrence $\xi_{n,\texttt{IN\_ACT}}$ of the variable ξ associated with the action `IN_ACT` , the equality $v^{\xi}_{n,\texttt{IN\_ACT}} = v^Z_n$. As a consequence, `H` must be absent when `Z` is.

- `OUT_ACT : Y = f(Xi_1,...,Xi_p) at H` creates in all events n of the presence of `H` and for the respective occurrences of $\xi^1_{n,\texttt{OUT\_ACT}}, \ldots, \xi^p_{n,\texttt{OUT\_ACT}}$ of variables $\xi^1, \ldots, \xi^p$ which are shared by the action under consideration, the constraint $v^Y_n = f(v^{\xi^1}_{n,\texttt{OUT\_ACT}}, \ldots, v^{\xi^p}_{n,\texttt{OUT\_ACT}})$. As a consequence, `H` must be absent when `Y` is.

Combinators acting on Programs with an Imperative Part Their semantics is exactly the one of combinators acting on programs without an imperative part that they induce.

REMARK: It has been noted that in the case where two different actions `ACT` et `ACT'` share the same variable `Xi` , the order of occurrences $\xi_{n,\mathtt{ACT}}$ and $\xi_{n,\mathtt{ACT'}}$ is *indifferent* (at least that it must be specified explicitly like in the example of `PRE`). In particular, in the case where `ACT` is an action of reading the variable and `ACT'` an action of writing in the variable the non-specification of the order between these two actions results in non-determinism, thus since the two orders are considered as equals, with, by definition, the results being different.

D.3.2 GC and μGC

Behavioral Semantics of GC

We give, for each GC program, the encoding in μGC of teh specification to which it is associated. In other words, we do not mention the ordering except where it is necessary to give a good definition of behavior (ordering of actions). For each execution context, the supplementary orderings will be added to it, we will see this in an example.

Primitives

`clock(X) = H` preserves itself

`X = f(A,B)` corresponds to the μGC equations

```
( ( clock(X) = clock(A) = clock(B) = H )  -- clocks
| ( X = f(A,B) at H ) )                   -- flows
```

`Y = pre(X)` corresponds to the μGC program (that we give in full here)

```
metagc PRE                            -- name of program

  act STATE, OUTPUT                   -- declaration of local actions
  var Xi                              -- declaration of local variabls

   ( ( clock(X)     = clock(Y)      = H     -- clocks
     | clock(STATE) = clock(OUTPUT) = H )
   | ( OUTPUT --> STATE at H )              -- dependencies
   | ( STATE  :  Xi = X at H                -- actions
     | OUTPUT :  Y  = Xi at H ) )

end PRE
```

`H = tt(B)` corresponds to the μGC equations

```
( ( clock(B) = K              -- clocks
  | H = tt(B) ) )
```

`Y = X when H` corresponds to the μGC equations

```
( ( clock(X) = CX                   -- clocks
  | clock(Y) = CY
  | CY = CX when H )
| ( Y = X at CY ) )                 -- flows
```

`Y = U default V` corresponds to the μGC equations

```
( ( clock(U) = CU                   -- clocks
  | clock(V) = CV
  | clock(Y) = CY
  | CY = CU default CV
  | seulCV = CV whennot CU )
| ( Y = U at CU                     -- flows
  | Y = V at seulCV ) )
```

The Combinators If μP generically designates the encoding in μGC of the behavioral semantics of a GC program P , it is simply

$$\begin{aligned} \mu(\texttt{P!!X}) &= (\mu\texttt{P})\texttt{!!X} \\ \mu(\texttt{P} \mid \texttt{Q}) &= (\mu\texttt{P})|(\mu\texttt{Q}) \end{aligned}$$

which signifies that the combinators are exactly the same in μGC and GC.

Dependencies and Operational Semantics

Two sorts of dependencies are taken into account: the ones that result from an explicit dependancy in the GC program, and the ones that are required by the operational semantics of a GC program, and which serve to explictly order its evaluations. For the first, there is:

`X -H-> Y` corresponds to the μGC equations:

```
( ( clock(X) = CX                        -- clocks
  | clock(Y) = CY
  | K = CX when CY when H )
| ( X --> Y at K ) )                     -- dependencies
```

Equation	Induced Dependance
	`clock(X) --> X at base`
`H = K clockop L`	`(K,L) --> H at base`
`X --> Y at H`	`H --> Y at base`
`H = tt(B)` `K = clock(B)`	`B --> H at K`
`Y = f(A,B) at H`	`(A,B) --> Y at H`
`IN-ACT: Xi = X at H`	`X --> IN-ACT at H`
`OUT-ACT: X = exp at H`	`OUT-ACT --> X at H`

Table D.2: *Induced Dependencies*

For the second aspect, to the clock equations, of flows, and to the actions we associate, or "induce," the dependencies according to the rules of Table D.2. These induced dependencies encode the ordering which is adequate for each of the mentioned equations. It can be noted in particular that all flows are preceded by their clock, and that the precedence indication of `X` over `Y` to the clock `H` implies a precedence of `H` on `Y` to the clock `base`, the only clock which, in all contexts can be spoken of in terms of presence or absence of a clock. To obtain the operational semantics of a GC program, one proceeds as follows:

1. the behaviors of the GC program under consideration are encoded in μGC, by using the encoding of the primitives given above. The explicit dependencies in the GC program source must be also translated into μGC.

2. The μGC program can be rewritten thus to obtain another equivalent program (defining the same behaviors), the explicit dependencies in the GC program before being unchanged.

3. The rules of Table D.2 are applied until saturation to obtain the ordering necessary for the sound execution of the program.

We will presently give a criterion allowing determining whether this procedure allows the effective execution of the program (showing that one has constructed a true operational semantics of the GC program under consideration).

D.3.3 An Exemple

The following small GC program will serve as an example :

```
( X = U default (pre(X)-1)
| clock(U) = tt(pre(X) =< 0) )    --  =< : less than or equal
```

A behavior of this program is, for example:

U	:	3	$\perp$	$\perp$	$\perp$	0	2	$\perp$	$\perp$	...
X	:	3	2	1	0	0	2	1	0	...
pre(X)	:	0	3	2	1	0	0	2	1	...
pre(X) $\leq$ 0	:	t	f	f	f	t	t	f	f	...
clock(U)	:	$\top$	$\perp$	$\perp$	$\perp$	$\top$	$\top$	$\perp$	$\perp$	...

In other words, this program is a “multiplexor” or “switch” : on reading the input `U` (of type integer, positive or zero), the program delivers the number of supplementary “ticks” equal to the value read for `U`. We will now construct the semantics of this program. We start with the behavioral semantcs. Calling `MUX_1` the μGC program corresponding to the first equation, we decompose it into

```
metagc MUX_1                            -- name of the programme

DEFAULT_GC | PRE_GC                     -- body
```

corresponding respectively to the GC equations `X = U default (Y-1)` and `Y = pre(X)` . Putting together the μGC encodings of the behaviors of the two GC equations, the result of Figure D.1, 172 is obtained. We can simplfy the clock equations, which gives the result shown in Figure D.2, 173. It should be noted that this simplification affects the “clock” fields, and that to change them, *we have used the properties of computations in the clock algebra.* By definition, have not finished here since it is now necessary to take into account the second equation.

Nevertheless, it is interesting to consider this μGC program in whole. Such as it is, it cannot be executed due to the ordering of the value computations, clocks and between flows and actions. The orderings added must be obtained by application of the rules of Table D.2 until saturation to give the induced dependencies. The complete ordering for this program is given in Figure D.3, 174 (and as such constitutes an operational semantics). In examining the sub-program of the first equation, one can prove that there are no appearances of equations defining:

- clocks `H,CU` ,
- flows `U` .

These objexts are therefore "inputs" of the first equation sub-program. This first equation, isolated from the rest of the body of the main program, speicifes the constraint on which the clocks `H,CU` are bound: we have not associated the induced dependencies since a cycle would have immediately resulted. Referring to various fields of GC programs, the first equation constitutes a *synchronization* (in which case no induced dependancy is generated), the rest of the program being in the *definition* field (where the induced dependencies are taken into account). There is here, therefore a deterministic program wherewhose entries are clocks `H,CU` and flow values `U`, but with constraints on teh input clocks `H,CU`.

The analysis is continued by taking the second GC equation. Its μGC encoding is, in writing `B` the boolean `(Y =< 0)`, shown in Figure D.4, 174. The combination of `MUX_1` such that Figure D.2 and of `MUX_2` of Figure D.4 finally, by suppressing the equations twice, the program of D.5, 175. Still missing in the program is the ordering of the computation of the clocks, we proceed as for the program `MUX_1`. The result is given in Figure D.6, 176. It can easily be proved that this μGC program

1. has no cycles if one examines the `-->` operators,
2. is free of multiple definitions of flows,
3. allows input for the activation clock `H` and for values of the flow `U`,
4. all the other objects being calculatable taken from these inputs with the ordering described by the `-->` operators and taking into account their clocks

In other words, we have here an "executable" deterministic program having for inputs the activation clock `H` and teh values of the flow `U`.

But we can say more: we can show, on this μGC program that this is an execution schema of `MUX`, in the sense of the definition 4. It suffices for us to erase the $\perp$. This is done in the following manner:

1. compute the transitive closure of the graph of `-->`,
2. eliminate the usage of "`clockop`" operators on clocks, which really require a context where one can speak of their presence or absence; this means only having the use of the operator `tt(.)`; in our case the `whennot` can be realized by producing the boolean `C = not B`, and by remarking that it is the case that `seulH = tt(C)`,
3. by filtering the clocks of dependencies to bring them back to the present occurrence of their origin.

There is the result on `MUX` : which is the program `MUX_ASYNCH` given in Figure D.7, 177. It still be verified that this μGCprogram

1. has no cycles of one examines the `-->` operators,
2. has no duplicate definitions of any flow,
3. allows for input activation clocks `H` and values of the flow `U` ,
4. all the other objects being computable from these inputs with the ordering described by the `-->` operators and taking into account their clocks.

In other words, we have a program which is "executable in the asynchronous mode." By definition, the filtering of clocks can be made after the fact.

Conclusions with Respect to This Example The following points points can be noted, that we have organized the following section:

- We have used the formal calculus of clocks to modify and simplify the syntactic form of the μGC programs wintout modifying their behavioral semantics.
- The dependencies necessary for a well ordering of the ultimate program found (`MUX`) were not added in the final phase, when the global μGC program was thrown out after rewriting it. In particular, it can be noticed that the program `MUX` of Figure D.6 orients the dependencies of `X` towards the clock `K` of `U` , so that the GC instruction `default` , considered alone induces, by application of the rules of Table D.2, dependencies in the inverse sense, from `U` to `X` .

  It can be proved that the choice of some dependencies depends on the context.
- We will have following used the mechanism of "clock filtering" to obtain a desynchronization, total or partial.
- We have implicitly used a notion of "executable program."

We will systematize all this in the next subsection.

D.3.4 Executability, Induced Dependencies, Desynchronization

Executability

We say that a μGC program is *correctly ordered* if all equations of the program have their induced dependencies made explicit in the program. In the statement that follows we call " `empty_clock` " the clock which is always absent.

Definition 6 (executable μGC programm) *A μGC program written* `P` *, will be said to be executable if it satisfies the following conditions : Un programme μGC, noté* `P` *, sera dit exécutable s'il satisfait aux conditions suivantes :*

1. `P` *is correctly ordered ;*

2. *for all dependancy cycles appearing in* `P` *, oft hef orm*

```
( X_0     --> X_1 at H_1
| X_1     --> X_2 at H_2
| .......
| X_(p-1) --> X_0 at H_0 )
```

   there is

```
H_1 when H_2 when ... when H_0  =  empty_clock
```

   In other words, the program `P` *does not have a circuit ;*

3. *for all duplicate definitions of flows appearing in* `P` *, of the form*

```
( X = exp_1 at H_1
| X = exp_2 at H_2 )
```

   there is

```
H_1 when H_2 =  empty_clock
```

   In other words, the data flow part of `P` *satisfies single assignment property ;*

4. *in considering the respective actions of writing and reading sharing a particular variable ; for all events, each occurrence present of such a read action possesses, among the write actions under consideration, at least one neighboring predecessor specified in a unique way. In other words, the imperative part of* `P` *satisfies single assignment.*

The following is the result :

Theorem 1 (properties of executable programs) *The restriction of an executable program to its "dependencies" part defines a partial order. For all executable μGC programs, there exists a partition of its input ports and other ports, and for this partition, the program is a function, whose partial order above specifies the operational semantics.*

The theorem constitutes in some way, a justification to the label "executable" that we have confered on programs satifying the conditions of the definition 6. It expresses that this definition furnishes a sufficiant condition that is effective for a μGC program to have an operational semantics. It should be noted that the conditions stated in this definition are not decidable in generat due to the operator " `tt(B)` " transforming a boolean into a clock. The boolean `B` can itself be the result of the evaluation of a predicate on flows of any arbitrary type. As a consequence, only an abstraction of this condition is implemented in practice therefore here are two more advantages:

- It reinforces conditions 2 and 3 of definition 6 as follows: there are no dependancy cycles or multiple definitions; we say therefore that the μGC program is *strongly* acyclic and has single assignment.

- It preserves conditions 2 and 3 of the definition 6, but the verification of the conditions of empty clocks is made in consideration that all boolean flows `B` resulting from the evaluation of a predicate on non-boolean types is an input (which is to say that the two valeus *true* and *false* are considered as possibilities since `B` is present)

The first poitn of view is the one which is followed by the existing compilers of the languages LUSTRE and SIGNAL, whereas the second more refined, is the one that is followed for the operational semantics off SIGNAL.

Proof The first point is immediate, only the rest of the statement must be considered. Among the clocks manipulated, we throw out two constants " `base` " (representing $\top$) and " `empty_clock` " (representing $\perp$). We use the following notations:

`eq` $\in$ `P`	`eq` is an equation of `P`
`P / eq`	suppression of `eq` in `P`
`P [X:x]`	replacement de `X` by the value `x` int `P`
`X source in P`	`X` does not appear on the right of the " `-->` " symbol

We introduce also the three following rules:

```
                                      X source in P
(R1)          P_f  =  ( X --> Y at base              ⊢  (P / P_f) [Y:f(X)]
                      | Y = f(X) at base) )     ∈  P

                                      X source in P
(R2)    P_ACT  =  ( X --> ACT at base                ⊢  (P / P_ACT) [Xi:exp]
                  | ACT : Xi = exp at base )    ∈  P

(R3)                 (... at empty_clock)  ∈  P   ⊢  P / (... at empty_clock)
```

The two first rules express the evaluation condition of flows or variables (`X` is an abriged notation denoting the t-uple `(X_1,...,X_p)` which is the set of prececessors of `Y` in `P`), and `Y` evalutates to the value `y` , equal to one of the constants `base` or `empty_clock` when `Y` is a clock. The third rule where "`...`" designates all μGC expression which can b used below the term "`at H`", expressions that all operations that do not have a present occurrence can be erased. These rules encode the transitions of an automaton ; the notion of a program `P` by defining the initial state. Thus one associates with all programs an automaton that encodes the evaluation ordering for the program. We will state conditions 1 and 2 of definition 6 implying that that this automaton does not have another state *puits* than the desired state, to understand the empty program.

We first handle the case of μGC primitives, then those of programs

Primitives We handle in detail two representative cases, to understand the following primitives :

```
X --> Y at H
Y = f(A,B) at H
```

Of the first, the application of the rules of Table D.2 gives, for `X --> Y at H` the set {primitive + induced dependencies} following :

```
( X --> Y at H
| H --> Y at base )
```

By application of the derivation rules given above, it becomes :

```
                        (R1) , H=base
                             ⊢          X --> Y at base
( X --> Y at H
| H --> Y at base )
                        (R3) , H=empty_clock
                             ⊢          nil
```

In the case where `H` evalues to `base` , the automaton is waiting for the evaluation of `X` ; in the other case the ordering condition disappears. In the case of teh second primitive, the application of the rules of Table D.2 given, for `Y = f(A,B) at H` , the set {primitive + induced dependencies} following:

```
( Y = f(A,B) at H
| clock(A) --> A at base
| clock(B) --> B at base
| clock(Y) --> Y at base
| (A,B) --> Y at H
| H --> Y at base )
```

The beginning of the derivation gives

```
                          ( Y = f(A,B) at H
                          | clock(A) --> A at base
                          | clock(B) --> B at base
                          | clock(Y) --> Y at base
                          | (A,B) --> Y at H
                          | H --> Y at base )
(R1) , H=base                 ( Y = f(A,B) at base
        ⊢                     | (A,B) --> Y at base )


(R3) , H=empty_clock          ( clock(A) --> A at base
        ⊢                     | clock(B) --> B at base
                              | clock(Y) --> Y at base )
```

The first case reduces to the program for executing `Y = f(A,B) at H` under a single clock whereas the second reduces to the program accepting all the behaviors for the ports `A,B,Y` . The final derivation for the first case gives thus

```
( Y = f(A,B) at base      (R1)
| (A,B) --> Y at base )    ⊢    [Y:f(A,B)]
```

The other primitives are handled in the same manner.

Programs. Due to condition 1 of the definition 6, all the primitives of the μGC program under consideration are decorated with their induced dependencies: the execution of one of these primitives is therefore possible since the source nodes of the set {primitive + induced dependencies} are evaluated. Condition 2 of definition 6 guarantees that no blockage appears in the series of these evaluations. Conditions 3 and 4 guarantee finally that the these evaluations are defined without ambiguity and produce a unique result. This proves the theorem □

Remark. In ffact, all this subsection could have been written by replacing the word " `base` " by the word " `tick` " where `tick` designates the upper limit of all the clocks of the program under consideration (contrary to `base` , `tick` is therefor a local notion).

Desynchronisation

Going back to Table D.2, the only part of this table which explicitly necessitates the $\perp$ symbols for absence are the first three lines. Reviewing, the other lines do not require the use of the $\perp$ symbol. A μGC program which never makes use of the $\perp$ symbol is an execution scheme in the sens of Section D.1.3; which is to say a program executable in an unsyncrhonized manner. In other worsd, to desynchronize an execution of a μGC program, one would try to rewrite the clock equations of the program as much as possible without making use of the `tt` operator. This is exactly whwat we have done for the `MUX` example. By definition partial desynchronizations are also possible.

```
metagc MUX_1

  act STATE, OUTPUT
  var Xi

    ( ( clock(U) = CU                          -- clocks
      | clock(Y) = CY
      | clock(X) = CX
      | CX = CU default CY
      | seulCY = CY whennot CU
      | clock(X)     = clock(Y)      = H
      | clock(STATE) = clock(OUTPUT) = H )
    | ( OUTPUT --> STATE at H )                -- dependencies
    | ( STATE  :  Xi = X at H                  -- actions
      | OUTPUT :  Y  = Xi at H ) )
    | ( X = U at CU                            -- flows
      | X = (Y-1) at seulCY ) )

end MUX_1
```

Figure D.1: *Behavioral semantics of the program* `MUX_1`

```
metagc MUX_1

  act STATE, OUTPUT
  var Xi

    ( ( clock(U) = CU                              -- clocks
      | H = CU default H
      | seulH = H whennot CU
      | clock(X)     = clock(Y)      = H
      | clock(STATE) = clock(OUTPUT) = H )
    | ( OUTPUT --> STATE at H )                    -- dependencies
    | ( STATE  :  Xi = X at H                      -- actions
      | OUTPUT :  Y  = Xi at H ) )
    | ( X = U at CU                                -- flows
      | X = (Y-1) at seulH ) )

end MUX_1
```

Figure D.2: *Behavioral semantics of the Simplified* MUX_1 *Program*

```
metagc MUX_1

  act STATE, OUTPUT
  var Xi

  ( ( ( H = CU default H ) )     <== contraint    -- clocks

  | ( ( seulH = H whennot CU                      -- clocks
      | clock(U)     = CU
      | clock(X)     = clock(Y)      = H
      | clock(STATE) = clock(OUTPUT) = H )
    | ( CU         --> U                  at base -- dependencies
      | H          --> (X,Y,STATE,OUTPUT) at base
      | (H,CU)     --> seulH              at base
      | (seulH,CU) --> X                  at base
      | U  --> X at CU
      | Y  --> X at seulH
      | OUTPUT --> Y     at H
      | OUTPUT --> STATE at H
      | X      --> STATE at H )
    | ( STATE  :  Xi = X at H                      -- actions
      | OUTPUT :  Y  = Xi at H ) )
    | ( X = U at CU                                -- flows
      | X = (Y-1) at seulH ) ) )

end MUX_1
```

Figure D.3: *The Program* MUX_1 *Simplified and Ordered*

```
metagc MUX_2

    ( ( clock(Y) = clock(B) = H    -- clocks
      | K = tt(B)
      | clock(U) = K )
    | ( B = (Y =< 0) at H ) )      -- flows

end MUX_2
```

Figure D.4: *The Program* MUX_2

```
metagc MUX

  act STATE, OUTPUT
  var Xi

    ( ( clock(X) = clock(Y) = clock(B) = H   -- clocks
      | clock(STATE) = clock(OUTPUT) = H
      | K = tt(B)
      | clock(U) = K
      | seulH = H whennot K )
    | ( OUTPUT --> STATE at H )              -- dependencies
    | ( STATE  :  Xi = X at H                -- actions
      | OUTPUT :  Y  = Xi at H )
    | ( X = U        at K                    -- flows
      | X = (Y-1)    at seulH
      | B = (Y =< 0) at H ) )

end MUX
```

Figure D.5: *The Program* `MUX`

```
metagc MUX

  act STATE, OUTPUT
  var Xi

    ( ( clock(X) = clock(Y) = clock(B) = H   -- clocks
      | clock(STATE) = clock(OUTPUT) = H
      | K = tt(B)
      | clock(U) = K
      | seulH = H whennot K )
    | ( H --> (X,Y,B,K,STATE,OUTPUT) at base -- dependencies
      | K --> U at base
      | (H,K) --> seulH at base
      | Y --> B at H
      | B --> K at H
      | (K,seulH) --> X at base
      | U  --> X at K
      | Y  --> X at seulH
      | OUTPUT --> Y     at H
      | OUTPUT --> STATE at H
      | X      --> STATE at H )
    | ( STATE  :  Xi = X at H                -- actions
      | OUTPUT :  Y  = Xi at H )
    | ( X = U        at K                    -- flows
      | X = (Y-1)    at seulH
      | B = (Y =< 0) at H ) )

end MUX
```

Figure D.6: *The Program* `MUX` *Ordered*

```
metagc MUX_ASYNCH

  act STATE, OUTPUT
  var Xi

    ( ( clock(X) = clock(Y) = clock(B) = H   -- clocks
      | clock(STATE) = clock(OUTPUT) = H
      | clock(U) = tt(B) )
    | ( H --> (X,Y,B,STATE,OUTPUT) at H      -- dependencies
      | Y --> B at H
      | (B,C) --> X at H
      | U  --> X at tt(B)
      | Y  --> X at tt(C)
      | OUTPUT --> Y     at H
      | OUTPUT --> STATE at H
      | X      --> STATE at H )
    | ( STATE  :  Xi = X at H                -- actions
      | OUTPUT :  Y  = Xi at H )
    | ( X = U        at tt(B)                -- flows
      | X = (Y-1)    at tt(C)
      | B = (Y =< 0) at H
      | C = not B at H ) )

end MUX_ASYNCH
```

Figure D.7: *The Program* MUX_ASYNCH