GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems (Syntax and Formal Semantics)

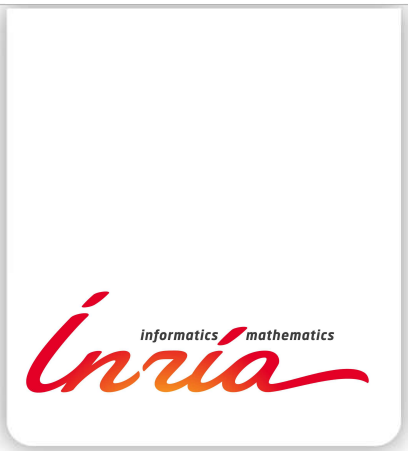
Fatma Jebali, Frédéric Lang, Radu Mateescu

GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems (Extended Version)

Fatma Jebali, Frédéric Lang, Radu Mateescu

RESEARCH REPORT
N° 8527
Avril 2014
Project-Team Convecs

ISSN 0249-6399 ISRN INRIA/RR--8527--FR+ENG



GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems (Extended Version)

Fatma Jebali, Frédéric Lang, Radu Mateescu

Project-Team Convecs

Research Report n° 8527 — Avril 2014 — 59 pages

Abstract: A GALS (Globally Asynchronous, Locally Synchronous) system consists of a set of synchronous subsystems executing concurrently and interacting using an asynchronous communication scheme. Such systems involve a high degree of synchronous and asynchronous concurrency which makes challenging the design and debugging of applications. The use of formal methods in the design process helps designers to master that complexity and to build strong confidence in the correctness of these, usually safety-critical, systems.
This report presents GRL (*GALS Representation Language*), a new formal language to specify GALS system for the purpose of formal verification, to provide design process with efficiency and correctness. GRL has a user-friendly syntax close to classical programming languages and an operational semantics combining the synchronous reactive paradigm inspired by classical data-flow languages and the asynchronous paradigm inspired by value-passing process algebras.

Key-words: concurrency, specification languages, compilation, modelling, formal verification, GALS systems

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Langage GRL pour la Spécification des Systèmes Globalement Asynchrones Localement Synchrones

Résumé : Un système GALS (Globalement Asynchrone, Localement Synchrone) est composé d'un ensemble de sous-systèmes synchrones qui s'exécutent de manière concurrente suivant un schéma de communication asynchrone. De tels systèmes impliquent un haut degrde concurrence synchrone et asynchrone, ce qui rend difficile la conception et le débogage des applications à cause du non-déterminisme des communications. L'intégration des méthodes formelles dans la procédure de conception aide les concepteurs à maîtriser cette complexité et à garantir la sûreté de ces systèmes, souvent critiques.

Ce rapport présente GRL(*GALS Representation Language*), un nouveau langage formel pour la spécification des systèmes GALS afin de les vérifier formellement, pour rendre le processus de conception sûr et efficace. GRL possède une syntaxe conviviale, proche des langages de programmation classiques, et une sémantique opérationnelle qui combine le modèle synchrone inspiré de la programmation par flot de données et le modèle asynchrone inspiré des algèbres de processus.

Mots-clés : concurrence, languages de spécification, compilation, modélisation, vérification formelle , systèmes GALS

1 Introduction

A wide range of industrial applications represent instances of GALS (*Globally Asynchronous, Locally Synchronous*) systems composed of several synchronous systems that execute concurrently and communicate asynchronously. Whatever their target platform (software, hardware, or heterogeneous) and their architecture (parallel or distributed), these systems are complex and are confronted to severe challenges in the design process, which makes the system integration critical. A formal analysis is then required to prove the efficiency and the correctness of such systems, usually safety-critical. Nevertheless, most available design tools in industry do not use rigorous verification methods because of a lack of support for formal modeling. Industrial companies have for long built their tools on classical synchronous frameworks, which prevent them from fully switching to new production methods supporting the GALS model.

Modeling and verification of GALS systems have gained remarkable research interest over the last decades in computer software. Most of the existing approaches are based on ad hoc modeling and fall into two main classes. The first class, aims to describe GALS systems by enhancing synchronous languages and frameworks [20, 5, 21, 15] to fit asynchronous concurrency and nondeterminism that such systems involve. The second class consists in expressing synchronous programs in fully asynchronous languages to encompass synchronous features [14, 10, 6]. A common limitation of these two trends is that either the asynchronous concept (in the case of the first class) or the synchronous concept (in the case of the second class) is disadvantaged since not recognized as a main paradigm, which narrows down verification results accuracy.

A more recent trend has emerged to preserve GALS systems features as much as possible by designing a new generation of languages that couple both the synchronous and the asynchronous paradigms. We cite the CRP model [3] that combines the Esterel [4] synchronous language and the CSP [16] asynchronous language. CRP is suitable for formal verification since it has been translated into classical process calculi. However, it is still very rarely used in industry since it requires a solid expertise in both synchronous and asynchronous concepts. In the same vein, SystemJ [19] extends the Java language with Esterel-like synchronous model and CSP-like asynchronous model. To our knowledge, no formal verification based on SystemJ has been investigated.

This report addresses the issue of designing a new modelling language with formal semantics, named *GRL* (*GALS Representation Language*), to bridge the gap between industrial design tools and automatic verification tools in order to enhance the design process of GALS systems and ensure the correctness of system construction. Such a language has to be general-purpose enough to cover a large scope of industrial systems and user-friendly to allow a degree of comfort for designers without the need of background in theoretical concurrency and formal verification.

The synchronous model of GRL is inspired from the data-flow model, a synchronous programming paradigm widely used in industry to design GALS systems. The asynchronous model of GRL is inspired from the LNT [8] model, a formal specification language based on process algebra and functional programming, instead of classical formal specification languages. LNT has been successfully used for the analysis and verification of industrial applications, in particular those based on the GALS scheme [9, 14, 13, 18].

The report is organized as follows. Section 2 defines some notational conventions used in the remainder of this report. Section 3 describes the lexical structure of the language. Section 4 presents the grammatical structure of the language. Sections 5 and 6 cover the static semantics and the operational semantics rules, respectively. Section 7 introduces some examples of GRL programs. Section 8 concludes the report. Finally, Appendices A and B give respectively the description of GRL lexical implementation using the Syntax LECL tool and the description of GRL concrete grammar using the Syntax TABC tool.

2 General Mathematical Notations

The following tables introduce some mathematical concepts and conventions used in this report.

Logical Operators	
Symbol	Meaning
$\neg$	negation
$\wedge$	conjunction
$\vee$	disjunction
$\Rightarrow$	implication

Sets and Sequences	
Symbol	Meaning
$a_1, \ldots, a_n$	possibly empty finite sequence of elements
$a_0, \ldots, a_n$	non-empty finite sequence of elements
$\{a_1, \ldots, a_n\}$	possibly empty set of elements $a_1, \ldots, a_n$
$\{a_0, \ldots, a_n\}$	non-empty set of elements $a_0, \ldots, a_n$
$a \in A$	a is an element of the set A
$A \subseteq B$	A is a subset of B
$\{a \in A \mid P(a)\}$	the set which contains only elements of A satisfying property P
$A \times B$	the Cartesian product, *i.e.*, the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$

3 Lexical Elements

This section presents the lexical conventions of GRL. The lexical structure is defined using a set of regular expressions in order to specify how characters are combined to form tokens and comments. Tokens consist of: identifiers, keywords, literals, and operators of the GRL syntactic grammar.

3.1 BNF notations

GRL grammar is described using the metalanguage BNF (Backus-Naur Form). A BNF specification is given by a set of derivation rules of the form "$A ::= B$" called productions, where A is a non-terminal symbol and B is a meta-expression built upon terminal symbols, non-terminal symbols, and BNF operators defined below.

Terminal symbols are written literally in the grammar in bold face and non-terminal symbols are written in italics.

The following table lists BNF meta-expression operators used to describe the grammar of GRL.

notation	description
$B_1 \mid B_2$	alternative: B_1 or B_2
$B_1\, B_2$	sequence: B_1 followed by B_2
$[B_0]$	option: B_0 or nothing
B_0*	possibly empty repetition: B_0, zero, one or several times
B_0+	non-empty repetition: B_0, one or several times

3.2 Boolean literals

Boolean literals can be either **true** or **false**.

$$\mathit{bool} \;::=\; \textbf{true} \mid \textbf{false}$$

3.3 Natural number literals

Natural number literals can be expressed in either decimal, hexadecimal, octal, or binary notation.

$$\begin{array}{rcl} \mathit{nat} & ::= & \mathit{digit}+ \\ & \mid & \textbf{0}\ (\textbf{x}\mid\textbf{X})\ \mathit{hexdigit}+ \\ & \mid & \textbf{0}\ (\textbf{o}\mid\textbf{O})\ \mathit{octaldigit}+ \\ & \mid & \textbf{0}\ (\textbf{b}\mid\textbf{B})\ \mathit{bitdigit}+ \end{array}$$

where *digit*, *hexdigit*, *octaldigit*, and *bitdigit* are defined by the following regular expressions:

$$\begin{array}{rcl} \mathit{bitdigit} & ::= & \textbf{0} \mid \textbf{1} \\ \mathit{octaldigit} & ::= & \mathit{bitdigit} \mid \textbf{2} \mid \textbf{3} \mid \textbf{4} \mid \textbf{5} \mid \textbf{6} \mid \textbf{7} \\ \mathit{digit} & ::= & \mathit{octaldigit} \mid \textbf{8} \mid \textbf{9} \\ \mathit{hexdigit} & ::= & \mathit{digit} \mid \textbf{a} \mid \textbf{b} \mid \textbf{c} \mid \textbf{d} \mid \textbf{e} \mid \textbf{f} \mid \textbf{A} \mid \textbf{B} \mid \textbf{C} \mid \textbf{D} \mid \textbf{E} \mid \textbf{F} \end{array}$$

3.4 Integer number literals

Integer number literals can be either zero, positive, or negative.

$$\mathit{int} \;::=\; [\textbf{-}\mid\textbf{+}]\ \mathit{nat}$$

3.5 Character and string literals

Characters and strings follow the same convention as the language LNT. In particular, strings are sequences of characters enclosed between double quotes (‘"’). For further details, see Subsection 3.1.5 of the Lnt2Lotos reference manual [8].

3.6 Operators

Operators can be classified as follows:

unary_operator ::= + | - | **not** | **abs** | **nat** | **nat16** | **nat32** | **int** | **int16** | **int32**
binary_operator ::= **and** | **or** | **xor** | **implies** | **equ** | + | - | % | ^ | * | /
| != | == | < | > | <= | >=

3.7 Comments

Comments can be either single line “- text” or multi-line “ (* *text* *)”. Therefore:

- All the text from the characters “-” to the end of the line is ignored.
- All the text enclosed by the characters “(*” and “*)” is ignored.

3.8 Identifiers

An identifier is defined by a letter followed by a possibly empty series of letters, digits and underscores. In order to prevent confusion and avoid conflicts with LNT infix declaration, we prohibit that an identifier starts or ends with an underscore.

letter ::= **a** | ... | **z** | **A** | ... | **Z**
identifier ::= *letter*(_ * (*digit* | *letter*))*

3.9 Reserved words

Reserved words can not be used as identifiers in GRL programs. The following table lists GRL reserved words.

abs	**allocate**	**and**	**any**	**as**	**array**	**block**
bool	**by**	**case**	**char**	**connectedby**	**const**	**constrainedby**
constant	**else**	**elsif**	**end**	**environment**	**enum**	**equ**
false	**for**	**if**	**implies**	**in**	**int**	**int8**
int16	**int32**	**is**	**loop**	**medium**	**nat**	**nat8**
nat16	**nat32**	**network**	**not**	**null**	**of**	**on**
or	**out**	**perm**	**program**	**record**	**select**	**send**
string	**system**	**temp**	**then**	**true**	**type**	**range**
receive	**where**	**while**	**xor**			

4 Abstract Syntax

This section covers the grammatical structure of GRL programs using BNF notation. The syntax of GRL is described by a context-free grammar, *i.e.*, a set of productions used to generate patterns of strings. BNF notations are described in Section 2.1.

4.1 Notational conventions

The following table summarizes terminal symbol identifiers and non-terminal symbols used to specify the grammar of GRL.

Symbols	
Identifiers (terminal symbols)	Meaning
P	*program*
S	*system*
B	*block*
Bi	*block instance*
N	*environment*
Ni	*environment instance*
M	*medium*
Mi	*medium instance*
F	*function*
X, Y	*variables*
T	*user-defined type identifier*
type	*type identifier*
C	*type constructor*
f	*record field*
Non-terminal symbols	**Meaning**
I	*statement*
E	*expression*
K	*literal constant*

4.2 Program definition

A program is the highest level syntactic construct in a GRL specification. It contains the definition of lower level constructs. A GRL file contains exactly one program definition. This program inherits the name of the file containing it.

A program P can import other programs P_0, ..., P_n. Therefore, the definitions of the imported programs P_0, ..., P_n are visible in the program P. Note that circular definitions are not allowed. For instance, if P_0 imports P_1, P_1 cannot import P_0.

The specification of a program is composed of the keyword **program**, the identifier of the program, an optional list of imported programs, the keyword **is**, a possibly empty list of definitions, and the keywords **end program**.

program_definition ::= **program** P [(*P_0* , ... , *P_n*)] **is**
(*type_definition*
| *constant_definition*
| *block_definition*
| *environment_definition*
| *medium_definition*
| *system_definition*)*
end program

4.3 Types definition

The following syntax lists the types recognized in a GRL specification. Typing rules are discussed in Section 4.3. Predefined types consist of **bool**, **nat**, **nat16**, **nat32**, **int**, **int16**, **int32**, **char**, and **string**. T denotes a user-defined type identifier.

$$
\begin{array}{rcl}
\textit{type} & ::= & \textbf{bool} \\
& | & \textbf{nat} \\
& | & \textbf{nat16} \\
& | & \textbf{nat32} \\
& | & \textbf{int} \\
& | & \textbf{int16} \\
& | & \textbf{int32} \\
& | & \textbf{char} \\
& | & \textbf{string} \\
& | & T
\end{array}
$$

User-defined types are the following:

- The array type, defined using the keyword **array**, denotes a fixed-size set of elements indexed by natural numbers.
- The record type, defined using the keyword **record**, denotes a fixed-size tuple of elements indexed by field names.
- The range type, defined using the keyword **range**, denotes a finite interval of integer numbers.
- The enumerated type, defined using the keyword **enum**, denotes a finite and ordered set of values.

$$
\begin{array}{rcl}
\textit{type\_definition} & ::= & \textbf{type } T \textbf{ is} \\
& & \quad \textit{type\_expression} \\
& & \textbf{end type} \\
\\
\textit{type\_expression} & ::= & \textbf{array } [m\,..\,n] \textbf{ of } \textit{type} \\
& | & \textbf{range } m \ .. \ n \textbf{ of } \textit{type} \\
& | & \textbf{record } f_0 : \textit{type}_0 \textbf{ , } \ \ldots \textbf{ , } f_n : \textit{type}_n \\
& | & \textbf{enum } C_0 \textbf{ , } \ \ldots \textbf{ , } C_n
\end{array}
$$

Note that:

- In array type, the bounds m and n must be literal naturals.
- In range type:
  - The bounds m and n must be literal integers.
  - The type *type* must be one of the following types: **nat**, **nat16**, **nat32**, **int**, **int16**, or **int32**.
- In an enumerated type, C_0, ..., C_n are identifiers representing symbolic values.

4.4 Literal constants

Literal constants may be either integer numbers, boolean constants, string constants, or elements of enumerated types.

$$
\begin{array}{rcl}
K & ::= & \mathit{int} \\
 & | & \mathit{bool} \\
 & | & \mathit{string} \\
 & | & C
\end{array}
$$

4.5 Expressions

An expression can be either a variable, a parenthesized expression, a record field access, an array element access, a unary operation, a binary operation, a literal constant, or a predefined function call. The syntax of expressions is given by the following grammar.

$$
\begin{array}{rcl}
E & ::= & X \\
 & | & \texttt{(}E_0\texttt{)} \\
 & | & E_0\,\texttt{.}f \\
 & | & E_1\,\texttt{[}E_0\texttt{]} \\
 & | & \mathit{unary\_operator}\ E_0 \\
 & | & E_1\ \mathit{binary\_operator}\ E_2 \\
 & | & K\ [\ \textbf{of}\ \mathit{type}\ ] \\
 & | & F\ \texttt{(}E_0\texttt{,}\ldots\texttt{,}E_n\texttt{)}
\end{array}
$$

4.6 Predefined functions

Predefined functions that can be used in a GRL program are unary operations, binary operations, type conversion functions, functions on arrays, and functions on records. Unary and binary operations are described in Section 3.6.

4.6.1 Type conversion

Type conversion functions, denoted "*type*`(`E`)`", convert an expression E from one numerical data type to another numerical data type "*type*". Numerical data types are: **nat**, **nat16**, **nat32**, **int**, **int16**, **int32**, and all the range types.

4.6.2 Functions on arrays

Given an array type defined by "**type** T **is array**`[`$m..n$`]` **of** *type*", two predefined functions $T : type^{n-m+1} -> T$ and $T : type -> T$ are automatically generated (those two functions coincide into one single function if $m{=}n$). The call $T\texttt{(}E_m\texttt{,}\ldots\texttt{,}E_n\texttt{)}$ builds an array X in which each element $X\texttt{[}i\texttt{]}$ $(i \in m..n)$ is set to the value of expression E_i. The call $T\texttt{(}E_0\texttt{)}$ builds an array X in which all elements $X\texttt{[}i\texttt{]}$ $(i \in m..n)$ are set to the value of expression E_0.

4.6.3 Functions on records

Given a record type defined by "**type** T **is record** $f_0 : type_0\texttt{,}\ldots\texttt{,}f_n : type_n$", a predefined function $T : type_0 \times \ldots \times type_n -> T$ is automatically generated. The call $T\texttt{(}E_0\texttt{,}\ldots\texttt{,}E_n\texttt{)}$ returns a record in which each field f_i $(i \in 0..n)$ is set to the value of expression E_i.

4.7 Statements

Below is the list of statements that can be used in GRL programs. For further details on static and dynamic semantics of statements, see Sections 5.5 and 6.3.

I	::=	**null**	no effect
	|	X `:=` E_0	assignment
	|	X `[`E_0`]` `:=` E_1	array element assignment
	|	X`.`f `:=` E_0	record field assignment
	|	I_1 `;` I_2	sequential composition
	|	**if** E_0 **then** I_0	conditional
		[**elsif** E_1 **then** I_1	
		...	
		elsif E_n **then** I_n]	
		[**else** I_{n+1}]	
		end if	
	|	**while** E_0 **loop**	while loop
		I_0	
		end loop	
	|	**for** I_0 **while** E_0 **by** I_1 **loop**	for loop
		I_2	
		end loop	
	|	**case** E_0 **is**	selection
		K_0 `->` I_0	
		`|` ...	
		`|` K_n `->` I_n	
		[`|` **any** `->` I_{n+1}]	
		end case	
	|	**select**	nondeterministic choice
		I_0 `[]` ... `[]` I_n	
		end select	
	|	X `:=` **any** *type* [**where** E_0]	nondeterministic assignment
	|	**on** [`?`]X_0`,` ... `,`[`?`]X_n `->` I_0	signal
	|	Bi`(`arg_0`,` ... `,`arg_n`)`	block invocation

4.8 Constant definition

The keyword **constant** is used to define a variable whose value, once initialized, can not be changed whatsoever. A constant, formally described below, is visible and can be called by all other entities defined in the program enclosing it and all programs that import the program enclosing it. Constant definition is composed of the keyword **constant**, an identifier of the constant, a type, the keyword **is**, an expression, and the keywords **end constant**.

constant_definition ::= **constant** X : *type* **is** E_0 **end constant**

4.9 Variable declaration

Variables in GRL are declared as follows:

$$
\begin{array}{rcl}
\mathit{declaration\_list} & ::= & \mathit{variable\_declaration}_0 \,,\, \ldots \,,\, \mathit{variable\_declaration}_n \\
\mathit{variable\_declaration} & ::= & X_0 \,,\, \ldots \,,\, X_m \ \texttt{:}\ \mathit{type}\ [\ \texttt{:=}\ E_0\ ] \\
\mathit{declaration\_list\_non\_init} & ::= & \mathit{variable\_declaration\_non\_init}_0 \,,\, \ldots \,,\, \mathit{variable\_declaration\_non\_init}_n \\
\mathit{variable\_declaration\_non\_init} & ::= & X_0 \,,\, \ldots \,,\, X_m \ \texttt{:}\ \mathit{type}
\end{array}
$$

Local variables, formally defined below, are visible only inside the caller body. Permanent variables, defined after the keyword **perm**, have a lifetime extending across the entire execution of the program. Temporary variables, defined after the keyword **temp**, have the same lifetime as one cycle of the caller execution.

$$
\begin{array}{rcl}
\mathit{local\_variables} & ::= & \textbf{temp}\ \mathit{declaration\_list} \\
& | & \textbf{perm}\ \mathit{declaration\_list}
\end{array}
$$

4.10 Formal parameters

Formal parameters are classified as follows:

– Constant parameters are user-fixed parameters defined as a set of variable declaration preceded by the keyword **const**. Such parameters should not be assigned a value in the body of the caller.

$$
\mathit{const\_parameters} \quad ::= \quad \textbf{const}\ \mathit{declaration\_list}
$$

– Input (resp., output) parameters defined as a set of variable declaration preceded by the keyword **in** (**out**).

$$
\begin{array}{rcl}
\mathit{inout\_parameters} & ::= & \textbf{in}\ \mathit{declaration\_list} \\
& | & \textbf{out}\ \mathit{declaration\_list\_non\_init} \\
\mathit{inout\_parameters\_non\_init} & ::= & \textbf{in}\ \mathit{declaration\_list\_non\_init} \\
& | & \textbf{out}\ \mathit{declaration\_list\_non\_init}
\end{array}
$$

– Receive (resp., send) parameters defined as a set of variable declaration preceded by the keyword **receive** (**send**).

$$
\begin{array}{rcl}
\mathit{com\_parameters} & ::= & \textbf{receive}\ \mathit{declaration\_list\_non\_init} \\
& | & \textbf{send}\ \mathit{declaration\_list\_non\_init}
\end{array}
$$

Input and receive parameters may be assigned a value in the body of the caller whereas output send parameters must be assigned a value in the body of the caller.

4.11 Blocks

Blocks represent the synchronous part of GRL, which is inspired by synchronous dataflow languages based on the block-diagram model. Following the definition of synchronous programs, a block is the synchronous composition of one or several subblocks, all governed by the clock of the highest level block.

A block specification consists of: the keyword **block**, the identifier of the block, an optional list of const parameters enclosed in square brackets, an optional list of input and output parameters enclosed in parenthesis and delimited by the symbol "**;**", an optional list of receive and send parameters enclosed in braces and delimited by the symbol "**;**", the keyword **is**, an optional list of block instance declarations defined after the keyword **allocate**, an optional list of local variables lists, a sequential deterministic behaviour described as statement I_0, and the keywords **end block**.

A block specification can also be included from an external code written in another language, in which case the block is called *external*. Such a block should not have receive and send parameters and its body consists of a *pragma* denoting the language in which the external function implementing the block is written, followed by the name of the function in the external program. Up to now, the supported external languages are C and LNT. An external C code identifier is preceded by the pragma `!c` and an external LNT code identifier is preceded by the pragma `!lnt`.

block_definition ::= **block** *B* [`[`*const_parameters*`]`]
[`(`*inout_parameters*$_0$ `;` ... `;` *inout_parameters*$_m$`)`]
[`{`*com_parameters*$_0$ `;` ... `;` *com_parameters*$_n$`}`] **is**
[**allocate** *B*$_0$`[`[*arg*$_{(0,0)}$ `,` ... `,` *arg*$_{(0,p_0)}$`]`] **as** *Bi*$_0$ `,`
...,
B$_q$`[`[*arg*$_{(q,0)}$ `,` ... `,` *arg*$_{(q,p_q)}$`]`] **as** *Bi*$_q$]
[*local_variables*$_0$ `,` ... `,` *local_variables*$_l$]
I$_0$
end block

| **block** *B* [`[`*const_parameters*`]`]
[`(`*inout_parameters*$_0$ `;` ... `;` *inout_parameters*$_m$`)`] **is**
!c *string* | **!lnt** *string*
end block

Blocks are allocated using the clause "**allocate** *B*$_0$ `[`*arg*$_0$ `,` ... `,` *arg*$_n$`]` **as** *Bi*$_0$", which creates an instance *Bi*$_0$ of block *B*$_0$ that has as const parameters the list *arg*$_0$, ..., *arg*$_n$.

The behaviour of a block is the following. First, it consumes its input and receive parameters. Second, it computes in zero time. Finally, it produces its output and send parameters. All aforementioned steps are assumed to be performed in zero-delay following the standard abstraction in synchronous programming. From one execution cycle to another, the set of permanent variables preserve their value constituting the *memory* of the block.

Inside a block, the scheduling of nested subblocks and interconnections between subblocks are inherently specified by the order in which the subblocks are invoked. For instance, if the body I_0 of a block B contains the call sequence "*Bi*$_0$ `;` *Bi*$_1$ `;` *Bi*$_2$" (where *Bi*$_0$, *Bi*$_1$, and *Bi*$_2$ are block instances), *Bi*$_0$, *Bi*$_1$, and *Bi*$_2$ are executed in this specific order. To preserve the synchronous behaviour of a block, static semantics prohibit the use of nondeterministic assignment and signal statement inside its body (See rule BB7 in Section 5.8.1).

4.12 Environments

Environments are GRL structures representing the behaviour of the environment surrounding a network of blocks. They allow to constrain either inputs of separate blocks or the relative order and frequency of block executions within a network of blocks. An environment definition consists of: the keyword **environment**, the identifier of the environment, an optional list of const parameters, an optional list of non-initialized input and output parameters delimited by the symbol "|", the keyword **is**, an optional list of block instance declarations, an optional list of local variables lists, a possibly nondeterministic behaviour described as statement I_0, and the keywords **end environment**. Such a representation is flexible since it makes possible the description of the common environment in the case of parallel systems distributed on a single platform as well as a set of separate environments in the case of geographically distributed systems, namely.

environment_definition ::= **environment** N [[*const_parameters*]]
[(*inout_parameters_non_init*$_0$ | ... | *inout_parameters_non_init*$_m$)] **is**
[**allocate** B_0[[$arg_{(0,0)}$, ... , $arg_{(0,p_0)}$]] **as** Bi_0 ,
... ,
B_q[[$arg_{(q,0)}$, ... , $arg_{(q,p_q)}$]] **as** Bi_q]
[*local_variables*$_0$, ... , *local_variables*$_l$]
I_0
end environment

Environments interact with blocks by connecting their input (resp., output) parameters to output (resp., input) parameters of blocks. They are invoked each time a connected block needs to consume its input parameters, or to produce its output parameters, in which case environments are said to be *activated*. As such, the activation of an environment is driven by operations over parameters arising from different blocks at different instants. Interactions between a block and an environment are as follows. During one execution cycle of the block, the environment can be activated at most once to provide the block with needed information and at most once to consume information received from the block. Consequently, since different code may have to be executed on the activating of an environment, signal statements enable the guard of parts of the environments code that need to be executed at each activation.

More specifically, let **N** be an environment having "**in** X_0 , ... , X_n" (resp., "**out** Y_0 , ... , Y_m") as input (resp., output) parameter list, the body I_0 of N should contain the statement "**on** X_0 , ... , X_n -> I_x" (resp., "**on** ? Y_0 , ... , ? Y_m -> I_y") to enable communication over this parameter list. In this respect, if N interacts with a block by receiving X_0 , ... , X_n (resp., sending Y_0 , ... , Y_m), only the statement "**on** X_0 , ... , X_n -> I" (resp., "**on** ? Y_0 , ... , ? Y_m -> I_y"), if any, is executed among all signal statements in the environment specification, all other signal statements being ignored (not executed). In other words, only one signal statement execution is permitted in one execution cycle of an environment and to this aim static semantics prohibit sequential composition of signals, signals in loops, and signal nesting (See rules IB3, IB5, IB6, and IB7 respectively in Section 5.5.1).

4.13 Mediums

Mediums are GRL structures representing the behaviour of communication mediums and enables a clean description of asynchronous interactions within a network of blocks. They enable the explicit description of the communication protocol and the rigorous design of networks whatever their topologies (star, bus, ring, etc.) and their means of communication (point-to-point, multipoint, etc.).

A medium definition consists of: the keyword **medium**, the identifier of the medium, an optional list of const parameters, an optional list of receive and send parameters delimited by the symbol "|", the keyword **is**, an optional list of block instance declarations, an optional list of local variables lists, a possibly nondeterministic behaviour described as a statement I_0, and the keywords **end medium**.

$$
\begin{array}{lll}
\textit{medium\_definition} & ::= & \textbf{medium}\ M\ [\ [\textit{const\_parameters}]\ ] \\
 & & \quad [\ \{\textit{com\_parameters}_0 \mid \ldots \mid \textit{com\_parameters}_m\}\ ]\ \textbf{is} \\
 & & [\ \textbf{allocate}\ B_0[[\textit{arg}_{(0,0)}, \ldots, \textit{arg}_{(0,p_0)}]]\ \textbf{as}\ Bi_0, \\
 & & \quad \ldots, \\
 & & \quad B_q[[\textit{arg}_{(q,0)}, \ldots, \textit{arg}_{(q,p_q)}]]\ \textbf{as}\ Bi_q\ ] \\
 & & [\ \textit{local\_variables}_0, \ldots, \textit{local\_variables}_l\ ] \\
 & & \quad I_0 \\
 & & \textbf{end medium}
\end{array}
$$

Mediums interact with blocks by connecting their receive (resp., send) parameters to send (resp., receive) parameters of blocks and exhibit the same behaviour as environments. Blocks, environments, and mediums are referred to as *actors* in the sequel.

4.14 Actors invocation

Actual parameters, formally defined below, denote parameters passed to an actor at invocation time. In the remainder of the document, we consider the following definitions:

- The *corresponding formal parameter* of an actual parameter denotes the formal parameter that has the same position (as the actual parameter) in the actor definition.
- An actual parameter is *at constant position* (respectively *at input position*, *at output position*, *at receive position*, and *at send position*) if its corresponding formal parameter is defined after the keyword **const** (respectively **in**, **out**, **receive**, and **send**).
- A *formal parameter list* denotes either *inout_parameters*, *inout_parameters_non_init*, *com_parameters*, or *const_parameters*.
- An *actual channel* denotes a set of actual parameters of the form "$\textit{arg}_0, \ldots, \textit{arg}_n$" used to connect actors inside systems.
- The *corresponding formal parameter list* of an actual channel when calling an actor denotes the formal parameter list that have the same position (as the formal parameter list) in the actor definition.

An actual parameter can have different forms according to its position (for further details, see rules ACB6 to ACB10 in Section 5.6.1). A question mark precedes both actual parameters at output and send positions. An underscore is used for unconnected parameters (*i.e.* unusable parameters). An actual parameter of the form "**any** *type*" assigns the corresponding formal parameters an arbitrary value of type "*type*".

$$\textit{arg} \ ::= \ ?X \mid [?]\_ \mid E \mid \textbf{any}\ \textit{type}$$

Actor instances must be allocated before being invoked. Actual parameters at constant position should be fixed at allocation time as follows:

$$
\begin{array}{lll}
\textit{allocation} & ::= & B[[\textit{arg}_0, \ldots, \textit{arg}_n]]\ \textbf{as}\ Bi \\
 & \mid & N[[\textit{arg}_0, \ldots, \textit{arg}_n]]\ \textbf{as}\ Ni \\
 & \mid & M[[\textit{arg}_0, \ldots, \textit{arg}_n]]\ \textbf{as}\ Mi
\end{array}
$$

Once allocated, actor instances can be invoked as given in the following grammar:

$$
\begin{array}{rcl}
\mathit{block\_instance} & ::= & \mathit{Bi}\ (\mathit{arg}_{(0,0)}, \ldots, \mathit{arg}_{(0,m_0)}, \ldots, \mathit{arg}_{(n,0)}, \ldots, \mathit{arg}_{(n,m_n)}) \\
\mathit{block\_instance} & ::= & \mathit{Bi}\ [(\mathit{arg}_{(0,0)}, \ldots, \mathit{arg}_{(0,m_0)}; \ldots; \mathit{arg}_{(n,0)}, \ldots, \mathit{arg}_{(n,m_n)})][\{\mathit{arg}'_{(0,0)}, \ldots, \mathit{arg}'_{(0,p_0)}; \ldots; \mathit{arg}'_{(q,0)}, \\
\mathit{environment\_instance} & ::= & \mathit{Ni}\ (\mathit{arg}_{(0,0)}, \ldots, \mathit{arg}_{(0,m_0)} \mid \ldots \mid \mathit{arg}_{(n,0)}, \ldots, \mathit{arg}_{(n,m_n)}) \\
\mathit{medium\_instance} & ::= & \mathit{Mi}\ \{\mathit{arg}_{(0,0)}, \ldots, \mathit{arg}_{(0,m_0)} \mid \ldots \mid \mathit{arg}_{(n,0)}, \ldots, \mathit{arg}_{(n,m_n)}\}
\end{array}
$$

4.15 System definition

Actors can be composed to form a higher level construct, called system. A system specifies a network of synchronous systems represented by a set of blocks. Those blocks are constrained by a set of environments and interact asynchronously via a set of mediums. A system definition consists of: the keyword **system**, the identifier of the system, an optional list of non-initialized parameters denoting the visible parameters of the system, the keyword **is**, a list of allocations, an optional list of temporary variables denoting the invisible parameters of the system, a list of block invocations defined after the keyword "**network**", an optional list of environment invocations defined after the keyword "**constrainedby**", and an optional list of medium invocations defined after the keyword "**connectedby**". Formally, a system is defined as follows:

$$
\begin{array}{rcl}
\mathit{system\_definition} & ::= & \textbf{system}\ S\ [\ (\mathit{declaration\_list\_non\_init}_0)\ ]\ \textbf{is} \\
& & \quad \textbf{allocate}\ \mathit{allocation}_0, \ldots, \mathit{allocation}_n \\
& & \quad [\ \textbf{temp}\ \mathit{declaration\_list\_non\_init}_1\ ] \\
& & \quad\ \textbf{network} \\
& & \quad\quad \mathit{block\_instance}_0, \\
& & \quad\quad \ldots, \\
& & \quad\quad \mathit{block\_instance}_p \\
& & \quad [\ \textbf{constrainedby} \\
& & \quad\quad \mathit{environment\_instance}_0, \\
& & \quad\quad \ldots, \\
& & \quad\quad \mathit{environment\_instance}_q\ ] \\
& & \quad [\ \textbf{connectedby} \\
& & \quad\quad \mathit{medium\_instance}_0, \\
& & \quad\quad \ldots, \\
& & \quad\quad \mathit{medium\_instance}_r\ ] \\
& & \textbf{end system}
\end{array}
$$

Interactions and communications between actors in a system are governed by the following rules:

- An environment can constrain more than one block. Conversely, a block can be constrained by more than one environment.
- A block can be connected to one or more mediums. However, a medium should be connected to more than one block (not prohibited but meaningless).
- An interaction between a block and an environment (respectively, between a block and a medium) is performed via exactly one actual channel.
- To preserve an independent synchronous behaviour of each block and an asynchronous behaviour of the network, direct (synchronous) communication between blocks is not permitted. Blocks communicate only across mediums.
- There is no interaction between environments and mediums.

5 Static Semantics

This section deals with the well-formedness of each of the syntactic construct of the language. We define the static semantics of GRL programs by specifying additional constraints that can not be captured by the grammar and are checked at compilation time. There are three classes of static semantics rules: binding rules, typing rules, and initialization rules.

5.1 Conventions

The following conventions are used:

1. "*Entity*" denotes an instance of: system, medium, environment, block, constant, or type.
2. "*vars(E)*" (where E is an expression) denotes the set of variables in the expression E. Formally:

$$\begin{aligned} vars(X) &= \{X\} \\ vars(E\texttt{.}f) &= vars(E) \\ vars(\texttt{(}E\texttt{)}) &= vars(E) \\ vars(F\texttt{(}E_0\texttt{,}\ldots\texttt{,}E_n\texttt{)}) &= vars(E_0) \cup \ldots \cup vars(E_n) \\ vars(E_0\texttt{[}E_1\texttt{]}) &= vars(E_0) \cup vars(E_1) \\ vars(unary\_operator\ E_0) &= vars(E_0) \\ vars(E_0\ binary\_operator\ E_1) &= vars(E_0) \cup vars(E_1) \\ vars(\texttt{(}E\texttt{)}) &= vars(E) \\ vars(K) &= \emptyset \end{aligned}$$

3. "*Constant variable*" is used to denote:
   - A global constant variable defined as **constant** entity.
   - A local constant parameter defined inside an entity after the keyword **const**.
4. "*Constant expression*" is used to denote an expression E such that $vars(E)$ contains only constant variables.
5. "*the current scope*" of an entity denotes the current program or an imported program.
6. "*Scope*" of a variable is used to denote the region of GRL code within which the variable is visible and usable.

5.2 Identifiers

(ID1) GRL reserved words must not be used as identifiers (see Section 3.9).

5.3 Types

In the sequel, we call "existing type" a type that is either a predefined type or a type defined in the current scope.

5.3.1 Binding rules

- **Enumerated type** "**type** T **is enum** C_0 **,** $\ldots$ **,** C_n **end type**"

  (TB1) Identifiers C_0, ..., C_n must be pairwise distinct.

- **Record type** "**type** T **is record** f_0 **:** *type*$_0$ **,** $\ldots$ **,**f_n **:** *type*$_n$ **end type**"

  (TB2) Fields f_0, ..., f_n must be pairwise distinct.

  (TB3) Types *type*$_0, \ldots,$ *type*$_n$ must be existing types.

  (TB4) If *type*$_i$ is a record type, it must not contain a field of type T. This prohibits circular dependencies between record types.

- **Array type** "**type** T **is array** `[`m`..`n`]` **of** *type* **end type**"

  (TB5) *type* must be an existing type.

  (TB6) The bounds m and n must be natural numbers such that $m \leq n$.

- **Range type** "**type** T **is range** m .. n **of** *type* **end type**"

  (TB7) *type* must be either **nat**, **nat16**, **nat32**, **int**, **int16**, or **int32**.

  (TB8) The bounds m and n must be integer numbers such that $m \leq n$.

5.4 Expressions

5.4.1 Binding rules

(EB1) Each variable must be declared in the scope where it is used.

5.4.2 Typing rules

- **Literal constant** "K **of** *type*"

  (ET1) *type* must be an existing type.

  (ET2) Each literal constant *int* of the form "*nat*" or "`+`*nat*" may have one of the types: **nat**, **nat16**, **nat32**, **int**, **int16**, or **int32** and each literal constant *int* of the form "`-`*nat*" may have one of the types: **int**, **int16**, or **int32**.

  (ET3) Each literal constant *bool* has the type **bool**.

  (ET4) Each literal constant *string* has the type **string**.

  (ET5) Each literal constant C_i ($i \in 1..n$) defined in "**type** T **is enum** C_0 **,** $\ldots$ **,** C_n **end type**" has the type T.

- **Variable** "X"

  (ET6) The type of X is the type with which X has been declared in the current scope.

- **Parenthesized expression** "`(`E`)`"

  (ET7) The type of `(`E`)` is the type of E.

- **Function call** "F`(`E_0 **,** $\ldots$ **,**E_n`)`"

(ET8) Each expression E_i ($i \in 0..n$) must have the same type as the corresponding parameter in the definition of F.

(ET9) The type of $F\texttt{(}E_0\texttt{,}\ldots\texttt{,}E_n\texttt{)}$ is the same type as the return type in the definition of F.

- **Array element access** "$E_0\texttt{[}E_1\texttt{]}$"

  (ET10) Expression E_0 must have a type T of the form **array** $\texttt{[}m\texttt{..}n\texttt{]}$ **of** *type*.

  (ET11) Expression E_1 must have either the type **nat**, the type **nat16**, or the type **nat32**.

  (ET12) The type of $E_0\texttt{[}E_1\texttt{]}$ is *type*.

- **Record field access** "$E\texttt{.}f$"

  (ET13) Expression E must have a type T of the form "**record** $f_0 : type_0\texttt{ , }\ldots\texttt{ ,}f_n : type_n$", such that $f = f_i$ for some $i \in 1..n$.

  (ET14) The type of $E\texttt{.}f$ is $type_i$.

- **Unary operation**

  The following table lists the possible types of the expression "$unary\_operator\ E_0$" given the type of E_0 for each unary operator.

Operator	Type of E_0	Type of "*unary_operator* E_0"
-, +	**int**	**int**
	int16	**int16**
	int32	**int32**
not	**bool**	**bool**
abs	**int**	**nat**
	int16	**nat16**
	int32	**nat32**
nat	**nat**	**nat**
	nat16	**nat**
	nat32	**nat**
	int	**nat**
	int16	**nat**
	int32	**nat**
nat16	**nat**	**nat16**
	nat16	**nat16**
	nat32	**nat16**
	int	**nat16**
	int16	**nat16**
	int32	**nat16**
nat32	**nat**	**nat32**
	nat16	**nat32**
	nat32	**nat32**
	int	**nat32**
	int16	**nat32**
	int32	**nat32**
int	**nat**	**int**
	nat16	**int**
	nat32	**int**
	int	**int**
	int16	**int**
	int32	**int**
int16	**nat**	**int16**
	nat16	**int16**
	nat32	**int16**
	int	**int16**
	int16	**int16**
	int32	**int16**
int32	**nat**	**int32**
	nat16	**int32**
	nat32	**int32**
	int	**int32**
	int16	**int32**
	int32	**int32**

Note that in the case of overflow (*e.g.*, "**nat16** (2 ^17)"), an error will be issued at run-time.

- **Binary operation**

  The following table shows the types of the operands E_0 and E_1 that can be used in an

expression of the form "E_0 *binary_operator* E_1", as well as the corresponding return type of the predefined function. Note that E_0 and E_1 must have the same type (otherwise, explicit type conversion using "**of**" must be used).

Operators on booleans		
Binary operator	Operands type	Return type
and, **or**, **xor**, **implies**, **equ**, ==, !=	**bool**	**bool**
Operators on natural numbers		
Binary operator	Operands type	Return type
+, -, *, /, ^, %	**nat16** **nat32**	**nat16** **nat32**
==, !=, <, >, <=, >=	**nat16** **nat32**	**bool** **bool**
Operators on integers		
Binary operator	Operands type	Return type
+, -, *, /, ^	**int16** **int32**	**int16** **int32**
==, !=, <, >, <=, >=	**int16** **int32**	**bool** **bool**
Operators on characters and strings		
Binary operator	Operands type	Return type
==, !=, <, >, <=, >=	**char**	**bool**
==, !=	**string**	**bool**

5.5 Statements

5.5.1 Binding rules

- **Deterministic assignment** "X :=E"

  (IB1) X must be declared in the scope where it is used.

  (IB2) X must be neither an input parameter, nor a constant variable.

- **Sequential composition** "I_1 **;** I_2"

  (IB3) At most one of I_1 and I_2 may contain a signal assignment.

- **Selection** "**case** E **is** K_0 -> I_0 | ... | K_n -> I_n [| **any** -> I_{n+1}] **end case**"

  (IB4) Constants $K_0, \ldots, K_n$ should cover all possible values of the type of E. Otherwise, the clause **any** -> I_{n+1} is mandatory.

- **While loop** "**while** E **loop** I **end loop**"

  (IB5) The statement I must not contain signal statements.

- **For loop** "**for** I_0 **while** E **by** I_1 **loop** I_2 **end loop**"

  (IB6) The statements I_1 and I_2 must not contain signal statements.

- **Signal** "**on** [?]X_0 **,** ... **,** [?]X_n -> I_0"

(IB7) I_0 must not contain a signal statement.

(IB8) A signal statement must have one of the following forms:

- "**on** X_0 **,** ... **,** X_n `->` I_0". In this case:
  - (IB8.1) Inside an environment definition, X_0, ..., X_n must have been defined as input parameters.
  - (IB8.2) Inside a medium definition, X_0, ..., X_n must have been defined as receive parameters.
- "**on** `?`X_0 **,** ... **,**`?`X_n `->` I_0". In this case:
  - (IB8.3) Inside an environment definition, X_0, ..., X_n must have been defined as output parameters.
  - (IB8.4) Inside a medium definition, X_0, ..., X_n must have been defined as send parameters.

Block invocation "Bi`(` arg_0 **,** ... **,** arg_n `)`"
See Section 4.6.

5.5.2 Typing rules

- **Deterministic assignment** "X `:=` E"

  (IT1) The type of X must be a valid type of E.

- **Nondeterministic assignment** "X `:=` **any** *type* [**where** E]"

  (IT2) *type* must be an existing type.

  (ST3) X must have type *type*.

  (ST4) E must have the type *bool*.

- **Array element assignment** X`[`E_0`]` `:=` E_1

  (IT5) Variable X must have an **array** type T defined as "**array** `[`$m..n$`]` **of** *type*".

  (IT6) Expression E_0 must have either the type **nat**, the type **nat32**, or the type **nat16**.

  (IT7) Expression E_1 must have the type *type*.

- **Record field assignment** "X`.`f `:=` E"

  (IT8) The variable X must have a type T defined as "**record** f_0 **:** $type_0$ **,** ... **,**f_n **:** $type_n$", such that $f = f_i$ for some $i \in 1..n$.

  (IT9) The type $type_i$ must be a type of E.

- **Conditional** "**if** E_0 **then** I_0 **elsif** E_1 **then** I_1 ... **elsif** E_n **then** I_n **else** I_{n+1} **end if**"

  (IT10) All expressions in conditional clauses E_0, ..., E_n must have the type *bool*.

- **While loop** "**while** E **loop** I **end loop**"

  (IT11) The expression E in the while clause must have the type *bool*.

- **For loop** "**for** I_0 **while** E **by** I_1 **loop** I_2 **end loop**"

  (IT12) The expression E in the for clause must have the type *bool*.

5.5.3 Initialization rules

We define triples of the form $S \triangleright I \triangleright S'$, where S and S' are sets of variables. The meaning of $S \triangleright I \triangleright S'$ is that, assuming all variables in S are initialized before executing I, then all variables in S' are necessarily initialized after executing I independently of the current valuation of variables. Hence, S' is a superset of S.

- **Deterministic assignment**

$$(SI1) \quad \frac{vars(E) \subseteq S}{S \ \triangleright \ X\texttt{:=}E \ \triangleright \ S \cup \{X\}}$$

- **Array update**

$$(SI2) \quad \frac{X \in S \qquad vars(E_0) \subseteq S \qquad vars(E_1) \subseteq S}{S \ \triangleright \ X\texttt{[}E_0\texttt{]:=}E_1 \ \triangleright \ S}$$

- **Record field update**

$$(SI3) \quad \frac{X \in S \qquad vars(E) \subseteq S}{S \ \triangleright \ X\texttt{.}f\texttt{:=}E \ \triangleright \ S}$$

- **Sequential composition**

$$(SI4) \quad \frac{S \triangleright I_1 \triangleright S' \qquad S' \triangleright I_2 \triangleright S''}{S \ \triangleright \ I_1\texttt{;}I_2 \ \triangleright \ S''}$$

- **Conditional**

$$(SI5) \quad \frac{\forall i \in 0..n \ vars(E_i) \subseteq S \qquad \forall j \in 0..n+1 \ S \triangleright I_j \triangleright S_j}{S \triangleright \ \textbf{if } E_0 \textbf{ then } I_0 \textbf{ elsif } E_1 \textbf{ then } I_1 \ \ldots \textbf{ elsif } E_n \textbf{ then } I_n \textbf{ else } I_{n+1} \textbf{ end if} \triangleright \ \bigcap_{j \in 0..(n+1)} S_j}$$

- **While loop**

$$(SI6) \quad \frac{vars(E) \subseteq S \qquad S \triangleright I_0 \triangleright S'}{S \ \triangleright \ \textbf{while } E \textbf{ loop } I_0 \textbf{ end loop} \triangleright \ S}$$

- **For loop**

$$(SI7) \quad \frac{vars(E) \subseteq S \qquad S \triangleright I_0 \triangleright S_1 \qquad S_1 \triangleright I_2 \triangleright S_2 \quad S_2 \triangleright I_1 \triangleright S_3}{S \ \triangleright \ \textbf{for } I_0 \textbf{ while } E \textbf{ by } I_1 \textbf{ loop } I_2 \textbf{ end loop} \ \triangleright \ S_1}$$

- **Selection**

$$(SI8) \quad \frac{vars(E) \subseteq S \qquad \forall i \in 0..n+1 \ S \ \triangleright \ I_i \triangleright S_i}{S \ \triangleright \ \textbf{case } E \textbf{ is } K_0 \texttt{ -> } I_0 \texttt{ | } \ldots \texttt{ | } K_n \texttt{ -> } I_n \texttt{ [| any -> } I_{n+1} \texttt{] } \textbf{end case} \ \triangleright \ \bigcap_{i \in 0..(n+1)} S_i}$$

- **Nondeterministic choice**

$$(SI9) \quad \frac{\forall i \in (0..n) \ S \ \triangleright \ I_i \ \triangleright S_i}{S \ \triangleright \ \textbf{select } I_0 \texttt{ [] } \ldots \texttt{ [] } I_n \textbf{ end select} \ \triangleright \ \bigcap_{i \in 0..n} S_i}$$

- **Nondeterministic assignment**

$$(SI10)\quad \frac{vars(E) \subseteq S \cup \{X\}}{S \ \triangleright\ X\ \texttt{:=}\ \textbf{any}\ T\ \textbf{where}\ E\ \triangleright\ S \cup \{X\}}$$

- **Signal**

$$(SI11)\quad \frac{S \triangleright I_0 \triangleright S' \qquad \{X_0, \ldots, X_n\} \subseteq S'}{S\ \triangleright\ \textbf{on}\ \texttt{?}X_0\texttt{,}\ldots\texttt{,?}X_n\ \texttt{->}\ I_0\ \triangleright\ S'}$$

$$(SI12)\quad \frac{S \cup \{X_0, \ldots, X_n\} \triangleright I_0 \triangleright S'}{S\ \triangleright\ \textbf{on}\ X_0\texttt{,}\ldots\texttt{,}X_n\ \texttt{->}\ I_0\ \triangleright\ S'}$$

- **Block invocation**
  See Section 5.6.

5.6 Actor allocation and invocation

An actor allocation can have one of the following forms:

- $B\,\texttt{[}arg_0\texttt{,}\ldots\texttt{,}arg_n\texttt{]}$ **as** Bi
- $N\,\texttt{[}arg_0\texttt{,}\ldots\texttt{,}arg_n\texttt{]}$ **as** Ni
- $M\,\texttt{[}arg_0\texttt{,}\ldots\texttt{,}arg_n\texttt{]}$ **as** Mi

An actor invocation can have one of the following forms:

- "$Bi\texttt{(}arg_{(0,0)}\texttt{,}\ldots\texttt{,}arg_{(0,n_0)}\texttt{,}\ldots\texttt{,}arg_{(m,0)}\texttt{,}\ldots\texttt{,}arg_{(m,n_m)}\texttt{)}$" (inside actors)
- "$Bi\texttt{(}arg_{(0,0)}\texttt{,}\ldots\texttt{,}arg_{(0,n_0)}\texttt{;}\ldots\texttt{;}arg_{(m,0)}\texttt{,}\ldots\texttt{,}arg_{(m,n_m)}\texttt{)}$" (inside systems)
- "$Bi\texttt{\{}arg_{(0,0)}\texttt{,}\ldots\texttt{,}arg_{(0,n_0)}\texttt{;}\ldots\texttt{;}arg_{(m,0)}\texttt{,}\ldots\texttt{,}arg_{(m,n_m)}\texttt{\}}$" (inside systems)
- "$Bi\texttt{(}arg_{(0,0)}\texttt{,}\ldots\texttt{,}arg_{(0,n_0)}\texttt{;}\ldots\texttt{;}arg_{(m,0)}\texttt{,}\ldots\texttt{,}arg_{(m,n_m)}\texttt{)\{}arg'_{(0,0)}\texttt{,}\ldots\texttt{,}arg'_{(0,p_0)}\texttt{,}\ldots\texttt{,}arg'_{(q,0)}\texttt{,}\ldots\texttt{,}\ arg'_{(q,p_q)}\texttt{\}}$" (inside systems only)
- "$Ni\texttt{(}arg_{(0,0)}\texttt{,}\ldots\texttt{,}arg_{(0,n_0)}\texttt{|}\ldots\texttt{|}arg_{(m,0)}\texttt{,}\ldots\texttt{,}arg_{(m,n_m)}\texttt{)}$" (inside systems)
- "$Mi\texttt{\{}arg_{(0,0)}\texttt{,}\ldots\texttt{,}arg_{(0,n_0)}\texttt{|}\ldots\texttt{|}arg_{(m,0)}\texttt{,}\ldots\texttt{,}arg_{(m,n_m)}\texttt{\}}$" (within systems)

5.6.1 Binding rules

(ACB1) Each actor must be defined in the current scope.

(ACB2) Each invoked actor must appear in the allocation list of the caller entity.

(ACB3) Inside an actor, the number of actual parameters passed to each block instance must be equal to the number of formal parameters of the corresponding block definition.

(ACB4) Inside a system, the number of actual channels passed to each actor instance must be equal to the number of formal parameter lists of the corresponding actor definition (respectively, medium definition).

(ACB5) The number of actual parameters in each actual channel passed to an actor instance must be equal to the number of parameters in the corresponding formal parameter list in the actor definition.

(ACB6) Actual parameters at constant position can be either the wildcard "_" or a constant expression "E".

(ACB7) Inside an actor, actual parameters passed to a block instance at input position can be either the wildcard "`_`" or an expression "E".

(ACB8) Inside an actor, actual parameters passed to a block instance at output position can be either the wildcard "`?_`" or a variable "`?`X".

(ACB9) Inside a system, input actual channels passed to a block instance can have one of the following forms:

- "X_0 **,** $\ldots$ **,** X_n"
- "**any** T_0 **,** $\ldots$ **,any** T_n"
- "`_`**,** $\ldots$ **,**`_`"

In the first form, the channel is said to be *connected* whereas in the two latter forms, the channel is said to be *unconnected*.

(ACB10) Inside a system, receive actual channels passed to a block instance have the form "X_0 **,** $\ldots$ **,** X_n".

(ACB11) Inside a system, output and send actual channels passed to a block instance can have one of the following forms:

- "`?`X_0 **,** $\ldots$ **,**`?`X_n"
- "`?_`**,** $\ldots$ **,**`?_`"

In the first form, the channel is said to be *connected* whereas in the latter form, the channel is said to be *unconnected*.

(ACB12) Inside a system, input (resp., receive) actual channels passed to a environment (resp., medium) instance can have one of the following forms:

- "X_0 **,** $\ldots$ **,** X_n"
- "`_`**,** $\ldots$ **,**`_`"

(ACB13) Inside a system, output (resp., send) actual channels passed to a environment (resp., medium) instance can have one of the following forms:

- "`?`X_0 **,** $\ldots$ **,**`?`X_n"
- "`?_`**,** $\ldots$ **,**`?_`"

(ACB14) In each actor instance, actual parameters at output position (resp., at send position) must be pairwise distinct, except the case of wildcard "`?_`" which may have several occurrences.

5.6.2 Typing rules

(ECT1) Each actual parameter passed at constant position in an actor invocation must have the same type as the corresponding const parameter in the actor definition.

(ECT2) Each actual parameter passed at input position in an actor invocation must have the same type as the corresponding input parameter in the actor definition.

(ECT3) Each actual parameter passed at output position in an actor invocation must have the same type as the corresponding output parameter in the actor definition.

(ECT4) Each actual parameter passed at receive position in an actor invocation must have the same type as the corresponding receive parameter in the actor definition.

(ECT5) Each actual parameter passed at send position in an actor invocation must have the same type as the corresponding send parameter in the actor definition.

5.6.3 Initialization rules

The following tables summarize possible forms of actual parameters, given the class of the corresponding formal parameter.

(ECI1) **Block invocation**

		class of formal parameter X				
		const	**in**	**out**	**receive**	**send**
actual parameter	_	X must have a default value at definition time. Then, X is assigned its default value.	X must have a default value at definition time. Then, X is assigned its default value.			
	E	X is assigned the value of E.	X is assigned the value of E.		X is assigned the value of E (where E is a variable).	
	?_			X must be assigned a value in the body I_0 of the callee actor.		X must be assigned a value in the body I_0 of the callee actor.
	? Y			X must be assigned a value in the body I_0 of the callee actor. Y is assigned the value of X.		X must be assigned a value in the body I_0 of the callee actor. Y is assigned the value of X.
	any T		X is assigned an arbitrary value of type T.			

(ECI2) **Environment invocation**

actual parameter	class of formal parameter X		
	const	**in**	**out**
_	X must have a default value at definition time. Then, X is assigned its default value.	Statements of the form "**on** X_0 **,** $\ldots$ **,**X_n**->** I_1" (if any) such that $X \in \{X_0, \ldots, X_n\}$ are never executed.	
E	X is assigned the value of E.	X is assigned the value of E.	
?_			Statements of the form "**on ?**X_0 **,** $\ldots$ **,?**X_n**->** I_1" (if any) such that $X \in \{X_0, \ldots, X_n\}$ are never executed.
?Y			X must be assigned a value in the body I_0 of the callee in every statement of the form "**on ?**X_0 **,** $\ldots$ **,?**X_n**->** I_1" (if any) such that $X \in \{X_0, \ldots, X_n\}$.
any T			

(ECI3) **Medium invocation**

actual parameter	class of formal parameter X		
	const	**receive**	**send**
_	X must have a default value at definition time. Then, X is assigned its default value.	Statements of the form "**on** X_0 **,** $\ldots$ **,**X_n**->** I_1" (if any) such that $X \in \{X_0, \ldots, X_n\}$ are never executed.	
E	X is assigned the value of E.	X is assigned the value of E.	
?_			Statements of the form "**on ?**X_0 **,** $\ldots$ **,?**X_n**->** I_1" (if any) such that $X \in \{X_0, \ldots, X_n\}$ are never executed.
?Y			X must be assigned a value in the body I_0 of the callee in every statement of the form "**on ?**X_0 **,** $\ldots$ **,?**X_n**->** I_1" (if any) such that $X \in \{X_0, \ldots, X_n\}$.
any T			

5.7 Constant

A constant is defined as follows: "**constant** X **:** *type* **is** E **end constant**"

5.7.1 Binding rules

(CB1) E must be a constant expression.

(CB2) *type* must be an existing type.

5.7.2 Typing rules

(CT1) *type* must be a valid type of E.

5.7.3 Initialization rules

(CI1) Every variable $X \in vars(E)$ must be a constant defined in the current scope.

5.8 Block

A block is defined as follows:

block_definition ::= **block** B [`[`*const_parameters*`]`]
[`(`*inout_parameters*$_0$ `;` ... `;` *inout_parameters*$_m$`)`]
[`{`*com_parameters*$_0$ `;` ... `;` *com_parameters*$_n$`}`] **is**
[**allocate** B_0`[`[*arg*$_{(0,0)}$ `,` ... `,` *arg*$_{(0,p_0)}$`]`] **as** Bi_0 `,`
...,
B_q`[`[*arg*$_{(q,0)}$ `,` ... `,` *arg*$_{(q,p_q)}$`]`] **as** Bi_q]
[*local_variables*$_0$ `,` ... `,` *local_variables*$_l$]
I_0
end block

| **block** B [`[`*const_parameters*`]`]
[`(`*inout_parameters*$_0$ `;` ... `;` *inout_parameters*$_m$`)`] **is**
!c *string* | **!lnt** *string*
end block

5.8.1 Binding rules

(BB1) Block identifiers B_0, ..., B_q must be defined in the current scope. Note that block recursion and circular dependencies to be avoided.

(BB2) Instance identifiers Bi_0, ..., Bi_q must be pairwise distinct.

(BB3) All variables declared in *const_parameters*, *inout_parameters*$_i$ ($i \in 0..m$), *com_parameters*$_i$ ($i \in 0..n$) , and *local_variables*$_i$ ($i \in 0..l$) must be pairwise distinct.

(BB4) The scope of formal parameters declared in *const_parameters*, *inout_parameters*$_i$ ($i \in 0..m$), and *com_parameters*$_i$ ($i \in 0..n$) is the text delimited by the keyword "**is**" and the keywords "**end block**".

(BB5) The scope of variables declared in *local_variables*$_i$ ($i \in 0..l$) is the text delimited by *local_variables*$_i$ and the keywords "**end block**".

(BB6) A variable declared in *const_parameters* must not be assigned a value in the body I_0.

(BB7) In the body I_0, the following statements are not allowed:

- nondeterministic assignment "X `:=` **any** *type* [**where** E]".
- nondeterministic choice "**select** I_1 `[]` ... `[]` I_n **end select**".
- signal statements "**on** X_0 `,` ... `,` X_n `->` I_1" and "**on** `?`X_0 `,` ... `,` `?`X_n `->` I_1".

(BB8) When using **!c** pragma, *string* must be a valid C identifier.

(BB9) When using **!lnt** pragma, *string* must be a valid LNT identifier.

5.8.2 Typing rules

(BT1) When using **!c** pragma, the type of each parameter declared in *const_parameters*, and *inout_parameters*$_i$ $i \in (0..m)$ must be a valid C type.

(BT2) When using **!lnt** pragma, the type of each parameter declared in *const_parameters*, and *inout_parameters*$_i$ $i \in (0..m)$ must be a valid LNT type.

5.8.3 Initialization rules

(BI1) See Section 5.6 for initialization rules concerning formal parameters.

(BI2) Permanent variables must be initialized at declaration time.

(BI3) A triple $S \triangleright I_0 \triangleright S'$ must hold, where S includes existing global constants and the set of all constant, input, and local variables that have been initialized at declaration time. S' must contain all output and send variables.

5.9 Environment

An environment specification has the following form:

environment_definition ::= **environment** *N* [[*const_parameters*]]
[(*inout_parameters_non_init*$_0$ | ... | *inout_parameters_non_init*$_m$)] **is**
[**allocate** B_0[[*arg*$_{(0,0)}$, ... , *arg*$_{(0,p_0)}$]] **as** Bi_0 ,
... ,
B_q[[*arg*$_{(q,0)}$, ... , *arg*$_{(q,p_q)}$]] **as** Bi_q]
[*local_variables*$_0$, ... , *local_variables*$_l$]
I_0
end environment

5.9.1 Binding rules

(NB1) Blocks B_0, ..., B_q must be defined in the current scope.

(NB2) Instance identifiers Bi_0, ..., Bi_q must be pairwise distinct.

(NB3) Variables declared in *const_parameters*, *inout_parameters_non_init*$_i$ ($i \in 0..m$), and *local_variables*$_i$ ($i \in 0..l$) must be pairwise distinct.

(NB4) The scope of formal parameters declared in *const_parameters* and *inout_parameters_non_init*$_i$ ($i \in 0..m$) is the text delimited by the keyword "**is**" and the keywords "**end environment**".

(NB5) The scope of variables declared in *local_variables*$_i$ ($i \in 0..l$) is the text delimited by *local_variables*$_i$ and the keywords "**end environment**".

(NB6) A variable declared in *const_parameters* must not be assigned a value in the body I_0.

(NB7) For each list *inout_parameters_non_init*$_i$ ($i \in 0..m$) of the form "**in** *declaration_list*" where X_0, ..., X_n is the set of variables of *declaration_list*, the body I_0 must contain at most one statement of the form "**on** X_0 , ... , X_n -> I_1".

(NB8) Inside the body I_0, for each signal statement of the form "**on** X_0 , ... , X_n -> I_1", X_0, ..., X_n must be the set of variables (in the same order) of *declaration_list* in some *inout_parameters_non_init*$_i$ ($i \in 0..m$) having the form "**in** *declaration_list*".

(NB9) For each list *inout_parameters_non_init*$_i$ ($i \in 0..m$) of the form "**out** *declaration_list_non_init*" where $X_0,\ldots,X_n$ is the set of variables in *declaration_list_non_init*, the body I_0 must contain at most one statement of the form "**on** `?`X_0 **,** $\ldots$ **,** `?`X_n`->` I_1".

(NB10) Inside the body I_0, for each signal statement of the form "**on** `?`X_0 **,** $\ldots$ **,** `?`X_n`->` I_1", X_0, $\ldots$, X_n must be the set of variables (in the same order) of *declaration_list* in some *inout_parameters_non_init*$_i$ ($i \in 0..m$) having the form "**out** *declaration_list_non_init*".

5.9.2 Initialization rules

(NI1) See Section 5.6 for initialization rules concerning formal parameters.

(NI2) Permanent variables must be initialized at declaration time.

(NI3) A triple $S \triangleright I_0 \triangleright S'$ must hold, where S includes existing global constants and the set of all constant and local variables that have been initialized at declaration time. S' must contain all output variables.

5.10 Medium

A medium specification has the following grammar:

medium_definition ::= **medium** M [`[`*const_parameters*`]`]
[`{`*com_parameters*$_0$ `|` $\ldots$ `|` *com_parameters*$_m$`}`] **is**
[**allocate** B_0`[`*arg*$_{(0,0)}$ **,** $\ldots$ **,** *arg*$_{(0,p_0)}$`]` **as** Bi_0 **,**
$\ldots$ **,**
B_q`[`*arg*$_{(q,0)}$ **,** $\ldots$ **,** *arg*$_{(q,p_q)}$`]` **as** Bi_q]
[*local_variables*$_0$ **,** $\ldots$ **,** *local_variables*$_l$]
I_0
end medium

5.10.1 Binding rules

(MB1) Blocks B_0, $\ldots$, B_q must be defined in the current scope.

(MB2) Instance identifiers Bi_0, $\ldots$, Bi_q must be pairwise distinct.

(MB3) Variables declared in *const_parameters*, *com_parameters*$_i$ ($i \in 0..m$), and *local_variables*$_i$ ($i \in 0..l$) must be pairwise distinct.

(MB4) The scope of formal parameters declared in *const_parameters* and *com_parameters*$_i$ ($i \in 0..m$) is the text delimited by the keyword "**is**" and the keywords "**end medium**".

(MB5) The scope of variables declared in *local_variables*$_i$ ($i \in 0..l$) is the text delimited by *local_variables*$_i$ and the keywords "**end medium**".

(MB6) A variable declared in *const_parameters* must not be assigned a value in the body I_0.

(MB7) For each list *com_parameters*$_i$ ($i \in 0..m$) of the form "**receive** *declaration_list_non_init*" where $X_0,\ldots,X_n$ is the set of variables in *declaration_list_non_init*, the body I_0 must contain at most one statement of the form "**on** X_0 **,** $\ldots$ **,** X_n`->` I_1".

(MB8) Inside the body I_0, for each signal statement of the form "**on** X_0 **,** ... **,** X_n **->** I_1", X_0, ..., X_n must be the set of variables (in the same order) of *declaration_list* in some *com_parameters* ($i \in 0..m$) having the form "**receive** *declaration_list_non_init*".

(MB9) For each list $com\_parameters_i$ ($i \in 0..m$) of the form "**send** *declaration_list_non_init*" where X_0, ..., X_n is the set of variables in *declaration_list_non_init*, the body I_0 must contain at most one statement of the form "**on** ?X_0 **,** ... **,** ?X_n **->** I_1".

(MB10) Inside the body I_0, for each signal statement of the form "**on** ?X_0 **,** ... **,** ?X_n **->** I_1", X_0, ..., X_n must be the set of variables (in the same order) of *declaration_list* in some $com\_parameters_i$ ($i \in 0..m$) having the form "**send** *declaration_list_non_init*".

5.10.2 Initialization rules

(MI1) See Section 5.6 for initialization rules concerning formal parameters.

(MI2) Permanent variables must be initialized at declaration time.

(MI3) A triple $S \triangleright I_0 \triangleright S'$ must hold, where S includes existing global constants and the set of all constant and local variables that have been initialized at declaration time. S' must contain all send variables.

5.11 System

A system specification is given by the following grammar.

system_definition ::= **system** *S* [(*declaration_list_non_init*$_0$)] **is**
allocate *allocation*$_0$ **,** ... **,** *allocation*$_n$
[**temp** *declaration_list_non_init*$_1$]
network
block_instance$_0$ **,**
... **,**
block_instance$_p$
[**constrainedby**
environment_instance$_0$ **,**
... **,**
environment_instance$_q$]
[**connectedby**
medium_instance$_0$ **,**
... **,**
medium_instance$_r$]
end system

5.11.1 Binding rules

(SB1) The scope of formal parameters declared in "*declaration_list_non_init*$_0$" is the text delimited by the keyword "**is**" and the keywords "**end system**".

(SB2) Actors allocated in the list "*allocation*$_0$ **,** ... **,** *allocation*$_n$" must be defined in the current scope.

(SB3) Identifiers of actor instances in the list "*allocation*$_0$ **,** ... **,** *allocation*$_n$" must be pairwise distinct.

(SB4) The scope of variables declared in "*declaration_list_non_init*$_1$" is the text delimited by the keyword "**network**" and the keywords "**end system**".

(SB5) An actual parameter (then, an actual channel) must be used in exactly one block invocation *block_instance*$_i$ among the block invocation list. This prohibits direct (synchronous) communication between blocks.

(SB6) At most one input (resp., output) actual channels of a block can be connected to an output (resp., input) actual channels of an environment.

This means that a connection between a block and an environment can be carried out via exactly one input actual channel of the block (then, exactly one output actual channel of the environment) and/or via exactly one output actual channel of the block (then, exactly one input actual channel of the environment).

(SB7) At most one receive (resp., send) actual channels of a block can be connected to a send (resp., receive) actual channels of a medium.

This means that a connection between a block and a medium can be carried out via exactly one receive actual channel of the block (then, exactly one send actual channel of the medium) and/or via exactly one send actual channel of the block (then, exactly one receive actual channel of the medium).

(SB8) An input (resp., receive) actual channel of the form "X_0 **,** ... **,** X_n" passed to a block instance can be connected to an output (resp., send) actual channel of the form "**?**X_0 **,** ... **,?**X_n" passed to an environment (resp., medium) instance.

(SB9) An output (resp., send) actual channel of the form "**?**X_0 **,** ... **,?**X_n" passed to a block instance can be connected to an input (resp., receive) actual channel of the form "X_0 **,** ... **,** X_n" passed to an environment (resp., medium) instance.

(SB10) If an actual channel of the form "X_0 **,** ... **,** X_n" does not have a respective actual channel of the form "**?**X_0 **,** ... **,?**X_n" in all other actor invocations, then "X_0 **,** ... **,** X_n" is semantically equivalent to "**any** T_0 **,** ... **,any** T_n" where T_0,...,T_n are respectively the types of X_0,...,X_n.

(SB11) Actual parameters at output position (resp., at send position) must be pairwise distinct in an actor invocation, except the case of wildcard "**?_**" which may have several occurrences.

See Section 5.6 for more static semantic rules concerning actors invocation.

5.12 Program

A program is defined as follows:

program_definition ::= **program** P [**(**P_0 **,** ... **,** P_n**)**] **is**
(*type_definition*
| *constant_definition*
| *block_definition*
| *environment_definition*
| *medium_definition*
| *system_definition*)∗
end program

5.12.1 Binding rules

(PB1) A GRL file must contain exactly one program definition.

(PB2) A program must have the name of the file enclosing it (*i.e.*, a program P must be defined in a file "*P.grl*").

(PB3) Imported program identifiers P_0, ..., P_n must be pairwise distinct and must be different from P.

(PB4) If a program P imports a program P', P' must not import P or any program importing P, transitively. This allows circular dependencies between programs to be avoided.

(PB5) All system, environment, block, medium, type, and constant identifiers defined in P, P_0, ..., P_n must be pairwise distinct.

(PB6) The scope of systems, environments, blocks, mediums, types, and constants is the program enclosing them and all programs importing the program enclosing them transitively.

6 Dynamic Semantics

Dynamic semantics concern the observable behaviour of programs at run time and are described formally in this section using Structural Operational Semantic rules by means of LTSs (*Labelled Transition Systems*).

6.1 Notational conventions

This section introduces a set of concepts and conventions that are used to define the formal semantics of the GRL language.

6.1.1 Functional symbols

Store

A *store*, denoted by ρ, is a partial function from variables to values. Square brackets are used to represent a store configuration. For instance, given a set of variables X_1, ..., X_n and a set of values e_1, ..., e_n, the store ρ that maps X_i to e_i ($\forall i \in 1..n$) is $[X_1 \leftarrow e_1, \ldots, X_n \leftarrow e_n]$. In particular, "[]" is the empty store.

Given a store $\rho = [X_1 \leftarrow e_1, \ldots, X_n \leftarrow e_n]$, we write $dom(\rho)$ for its domain defined by $dom(\rho) = \{X_1, \ldots, X_n\}$.

Store update

The direct sum of two stores ρ_1 and ρ_2, denoted "$\rho_1 \oplus \rho_2$", represents the update of ρ_1 as regards ρ_2. Formally, it is a partial function defined as below:

$$(\rho_1 \oplus \rho_2)(X) \quad = \quad \begin{cases} \rho_2(X) & \text{if } X \in dom(\rho_2) \\ \rho_1(X) & \text{if } X \notin dom(\rho_2) \text{ and } X \in dom(\rho_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The notation $\bigoplus_{i \in 1..n} \rho_i$ stands for the sum $\rho_1 \oplus \ldots \oplus \rho_n$.

Stack

The symbol σ is used to denote a sequence of actor instance identifiers. Formally, σ is either the empty sequence ϵ or a non empty sequence of the form "$\sigma'.Ai$" where σ' is a sequence (recursively) and Ai is an actor instance identifier. σ will represent the call stack of the current actor instance. Note that such a stack has a bounded size since recursion is not allowed in block, environment, and medium invocations.

For instance, if an environment N invokes a block instance Bi and Ni is an instance of N, the stack σ_{Bi} of Bi is $\sigma_{Bi} = Ni.Bi$. If Bi invokes itself a block instance Bi', the stack of Bi' is $\sigma_{Bi'} = \sigma_{Bi}.Bi' = Ni.Bi.Bi'$.

Memory

A *memory* is a function from stacks to stores. Given an actor Ai executed regarding a stack σ, $\mu(\sigma)$ returns the store assigning a value to each permanent variable in Ai. For instance, given an actor instance Ai and its permanent variables $X_0, \ldots, X_n$, the memory $\mu(\sigma.Ai)$ has the form $[X_0 \leftarrow e_0, \ldots, X_n \leftarrow e_n]$.

Given a memory $\mu = [\sigma_1 \leftarrow \rho_1, \ldots, \sigma_n \leftarrow \rho_n]$, we write $dom(\mu)$ for its domain defined by $\{\sigma_1, \ldots, \sigma_n\}$.

Memory update

The direct sum of two memories μ_1 and μ_2 denoted "$\mu_1 \oplus \mu_2$" represents the update of μ_1 as regards μ_2. Formally, it is a partial function defined as below:

$$(\mu_1 \oplus \mu_2)(\sigma) \quad = \quad \begin{cases} \mu_2(\sigma) & \text{if } \sigma \in dom(\mu_2) \\ \mu_1(\sigma) & \text{if } \sigma \notin dom(\mu_2) \text{ and } \sigma \in dom(\mu_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The notation $\bigoplus_{i \in 1..n} \mu_i$ stands for the sum $\mu_1 \oplus \ldots \oplus \mu_n$.

6.1.2 Labelled transition system

An LTS is a quadruple $(\mathbf{S}, \mathbf{L}, \rightarrow, s_0)$ where:

- $\mathbf{S}$ is a set of states.
- $\mathbf{L}$ is a set of labels.
- $\rightarrow \subseteq \mathbf{S} \times \mathbf{L} \times \mathbf{S}$ is the labelled transition relation.
- $s_0 \in S$ is the initial state.

A labelled transition system is finite if its sets of states and transitions are both finite. We write $s \xrightarrow{\ell} s'$ as a shorthand for $(s, a, s') \in \rightarrow$.

6.2 Dynamic semantics of expressions

The evaluation of expressions is defined by triples of the form "$\{E\}\ \rho\ \rightarrow_{\mathtt{e}}\ e$" where E is an expression, ρ is a store, and e is a value. This relation means that the expression E returns the value e in the store ρ.

6.2.1 Constant

The expression evaluation of a literal constant K returns the value denoted by K.

$$\frac{}{\{K\}\ \rho \rightarrow_{\mathtt{e}} K}$$

6.2.2 Variable

The result of the expression evaluation of a variable X is its value in the current store.

$$\frac{}{\{X\}\ \rho \rightarrow_{\mathtt{e}} \rho(X)}$$

6.2.3 Predefined function call

The evaluation result of the expression "$F\,\mathtt{(}E_0\,\mathtt{,}\,\ldots\,\mathtt{,}\,E_n\mathtt{)}$" is the returned value of calling the predefined function with the values of $E_0, \ldots, E_n$, unary operations, and binary operations. Predefined functions are standard, we do not provide here their formal semantics.

$$\frac{(\forall\ i\ \in 0..n)\ \{E_i\}\ \rho \rightarrow_{\mathtt{e}} e_i}{\{F\ \mathtt{(}E_0\,\mathtt{,}\,\ldots\,\mathtt{,}\,E_n\mathtt{)}\}\ \rho \rightarrow_{\mathtt{e}} F\mathtt{(}e_0\,\mathtt{,}\,\ldots\,\mathtt{,}\,e_n\mathtt{)}}$$

Record field access and array element access are replaced by predefined functions.

6.3 Dynamic semantics of statements

The execution of statements is defined by septuples of the form "$\{I\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}}\ \rho', \mu'$" where I is a statement, σ is a block instance stack, ρ and ρ' are stores, μ and μ' are memories, and ℓ is a label that has one of the following forms:

- ϵ means that the execution of I has terminated normally.
- $X_0, \ldots, X_n$ means that the execution of I has terminated on a signal emission statement of the form "**on** X_0 **,** $\ldots$ **,** X_n`->` I_0".
- $?X_0, \ldots, ?X_n$ means that the execution of I has terminated on a signal emission statement of the form "**on** `?`X_0 **,** $\ldots$ **,** `?`X_n`->` I_0".

ρ represents the current store, σ represents the stack of the actor instance enclosing the statement, μ represents the store assigning to permanent variables of the actor instance their values in the last execution cycle of the actor (and their initialization values in first cycle). This relation means that the execution of the statement I in the store ρ and the memory μ is terminated normally producing the updated store ρ' and the updated memory μ'.

6.3.1 Null

The null statement terminates normally without updating the store.

$$\frac{}{\{\mathbf{null}\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho, \mu}$$

6.3.2 Sequential composition

The execution of the statement "I_1 `;` I_2" starts by executing the statement I_1 and updating the store, then I_2 is executed in the store updated by I_1.

$$\frac{\{I_1\}\ \sigma, \rho, \mu \xrightarrow{\ell_1}_{\mathsf{i}} \rho', \mu' \qquad \{I_2\}\ \sigma, \rho', \mu' \xrightarrow{\ell_2}_{\mathsf{i}} \rho'', \mu''}{\{I_1\ \mathtt{;}\ I_2\}\ \sigma, \rho, \mu \xrightarrow{\ell_1+\ell_2}_{\mathsf{i}} \rho'', \mu''}$$

where $\epsilon + \ell = \ell + \epsilon = \ell$ for every label ℓ. Note that at least one of the labels ℓ_1 and ℓ_2 must be equal to ϵ (See rule IB3 in Section 5.5.1).

6.3.3 Assignment

The execution of an assignment statement "X`:=`E" updates the store by mapping the value of E to the assigned variable X.

$$\frac{\{E\}\ \rho \rightarrow_{\mathsf{e}} e}{\{X\ \mathtt{:=}\ E\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho \oplus [X \leftarrow e], \mu}$$

6.3.4 Array element assignment

The evaluation of the expression "$X\,\texttt{[}E_0\texttt{]}\,\texttt{:=}E_1$" where X has the type T such as "T **:array** `[`$m\,..\,n$`]` **of** T'" is given by the following rule:

$$\frac{\{E_0\}\ \rho \to_{\mathsf{e}} e_0 \qquad e_0 \in [m..n] \qquad \{E_1\}\ \rho \to_{\mathsf{e}} e_1}{\{X\,\texttt{[}E_0\texttt{]}\,\texttt{:=}E_1\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho \oplus [X \leftarrow \mathbf{update}(\rho(X), e_0, e_1)], \mu}$$

where $\mathbf{update}(e_0, e_1, e_2)$ denotes a mathematical function which assigns the value e_2 to the element of the array e_0 placed at position e_1.

6.3.5 While loop

The semantics of the statement "**while** E **loop** I **end loop**" are:

$$\frac{\{E\}\ \rho \to_{\mathsf{e}} \mathbf{false}}{\{\mathbf{while}\ E\ \mathbf{loop}\ I\ \mathbf{end\ loop}\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho, \mu}$$

$$\frac{\begin{array}{c}\{E\}\ \rho \to_{\mathsf{e}} \mathbf{true}\\ \{I\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho', \mu'\end{array}}{\{\mathbf{while}\ E\ \mathbf{loop}\ I\ \mathbf{end\ loop}\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho', \mu'}$$

6.3.6 For loop

The semantics of "**for** I_0 **while** E **by** I_1 **loop** I_2 **end loop**" are equivalent to the semantics of the following statement:

I_0 **;**
while E **loop**
  I_2 **;** I_1
end loop

6.3.7 Conditional

The semantics of the statement "**if** E_0 **then** $I_0 \ldots$ **elsif** E_n **then** I_n **else** I_{n+1} **end if**" are:

$$\frac{\begin{array}{c}(\exists i \in 0..n)\ \ \forall j \in 0..i-1\ \ \{E_i\}\ \rho \to_{\mathsf{e}} \mathbf{false}\\ \{E_i\}\ \rho \to_{\mathsf{e}} \mathbf{true}\\ \{I_i\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \rho', \mu'\end{array}}{\{\mathbf{if}\ E_0\ \mathbf{then}\ I_0 \ldots \mathbf{elsif}\ E_n\ \mathbf{then}\ I_n\ \mathbf{else}\ I_{n+1}\ \mathbf{end\ if}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \rho', \mu'}$$

$$\frac{(\forall k \in 0..n)\ \ \{E_k\}\ \rho \to_{\mathsf{e}} \mathbf{false} \qquad \{I_{n+1}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \rho', \mu'}{\{\mathbf{if}\ E_0\ \mathbf{then}\ I_0\ \ldots\ \mathbf{elsif}\ E_n\ \mathbf{then}\ I_n\ \mathbf{else}\ I_{n+1}\ \mathbf{end\ if}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \rho', \mu'}$$

Note that if the clause "**else** I_{n+1}" is absent, the **if** statement is semantically equivalent to: "**if** E_0 **then** $I_0\ \ldots$ **elsif** E_n **then** I_n **else null end if**".

6.3.8 Nondeterministic assignment

The semantics of the statement "X **:= any** T **where** E" are:

$$\frac{\begin{array}{c}e \in T\\ \{E\}\ \rho \oplus [X \leftarrow e] \to_{\mathsf{e}} \mathbf{true}\end{array}}{\{X\ \mathbf{:=\ any}\ T\ \mathbf{where}\ E\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho \oplus [X \leftarrow e], \mu}$$

6.3.9 Nondeterministic choice

The semantics of the statement "**select** I_0 [] ... [] I_n **end select**" are:

$$\frac{(\exists k \in 0..n) \quad \{I_k\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \ \rho', \mu'}{\{\textbf{select}\ I_0\ \texttt{[]}\ \ldots\ \texttt{[]}\ I_n\ \textbf{end select}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \rho', \mu'}$$

6.3.10 Selection

The semantics of the statement "**case** E **is** K_0 -> I_0 | ... | K_n -> I_n | **any** -> I_{n+1} **end case**" are:

$$\frac{\{E\}\rho \rightarrow_{\mathsf{e}} e \quad (\exists i \in 0..n)\ (\forall j \in 0..i-1)\ K_j \neq e,\ K_i = e \quad \{I_k\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \ \rho', \mu'}{\{\ \textbf{case}\ E\ \textbf{is}\ K_0\ \texttt{->}\ I_0\ \texttt{|}\ \ldots\ \texttt{|}\ K_n\ \texttt{->}\ I_n\ \texttt{|}\ \textbf{any}\ \texttt{->}\ I_{n+1}\ \textbf{end case}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \ \rho', \mu'}$$

$$\frac{\{E\}\rho \rightarrow_{\mathsf{e}} e \quad \forall i \in 0..n\ K_i \neq e \quad \{I_{n+1}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \ \rho', \mu'}{\{\ \textbf{case}\ E\ \textbf{is}\ K_0\ \texttt{->}\ I_0\ \texttt{|}\ \ldots\ \texttt{|}\ K_n\ \texttt{->}\ I_n\ \texttt{|}\ \textbf{any}\ \texttt{->}\ I_{n+1}\ \textbf{end case}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \ \rho', \mu'}$$

Note that if the clause "**any** -> I_{n+1}" is absent, static semantics ensure that $K_0, \ldots, K_n$ cover all possible values of the type of expression E (see rule IB4 in Section 5.5.1). In this case, the "**case**" statement is semantically equivalent to:

case E **is** K_0 -> I_0 | ... | K_n -> I_n | **any** -> **null end case**

6.3.11 Signal

A signal statement has one of the following forms:

on X_0 **,** ... **,** X_n-> I
on ?X_0 **,** ... **,**?X_n-> I

Signal statement terminates normally by executing the statement I in the current store and producing a label. Note that static semantics ensure that nested signal statements are forbidden (*i.e.*, I must not contain a signal statement, see rule IB5 in Section 5.5.1).

$$\frac{\{X_0, \ldots, X_n\} \subseteq dom(\rho) \quad \{I\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \ \rho', \mu'}{\{\textbf{on}\ X_0\ \textbf{,}\ \ldots\ \textbf{,}\ X_n\texttt{->}\ I\}\ \sigma, \rho, \mu \xrightarrow{X_0,\ldots,X_n}_{\mathsf{i}} \ \rho', \mu'}$$

$$\frac{\{I\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \ \rho', \mu'}{\{\textbf{on}\ ?X_0\ \textbf{,}\ \ldots\ \textbf{,}\ ?X_n\texttt{->}\ I\}\ \sigma, \rho, \mu \xrightarrow{?X_0,\ldots,?X_n}_{\mathsf{i}} \ \rho', \mu'}$$

6.3.12 Block invocation

Conventions. In the remainder of the section, we adopt the following notational conventions:

- *a* stands for *arg*.
- *vd* stands for either *variable_declaration* or *variable_declaration_non_init*.
- *dl* stands for either *declaration_list* or *declaration_list_non_init*.
- *ch* stands for "**in** *dl*", "**out** *dl*", "**receive** *dl*", or "**send** *dl*".

– *ac* stands for "$a_0, \ldots, a_n$".

– *block* (respectively, *environment*, *medium*) stands for the non-terminal *block_instance* (respectively, *environment_instance*, *medium_instance*).

– ρ_{global} stands for the store assigning values to global constants.

Definitions. We define the following auxiliary functions.

– Initial store: function *init* maps each variable in a declaration list to its respective initialization value. if any.

$$\begin{array}{rcl}
\text{init}(\langle vd_0, \ldots, vd_n \rangle, \rho) & = & \text{init}(vd_0, \rho) \oplus \ldots \oplus \text{init}(vd_n, \rho) \\
\text{init}(X_0, \ldots, X_m : type, \rho) & = & [] \\
\text{init}(X_0, \ldots, X_m : type \texttt{ := } E, \rho) & = & [X_0 \leftarrow e, ..., X_m \leftarrow e] \text{ where } \{E\}\ \rho \rightarrow_{\texttt{e}} e
\end{array}$$

– Variable list: function *vars* returns the list of variable identifiers in either variable declaration list, formal parameter declaration, actual parameter list, or a label. Symbol ϵ denotes the empty list and operator ++ denotes list concatenation.

$$\begin{array}{rcl}
\text{vars}(vd_0, \ldots, vd_n) & = & \text{vars}(vd_0)\texttt{++}\ldots\texttt{++}\text{vars}(vd_n) \\
\text{vars}(X_0 \texttt{ , } \ldots \texttt{ , } X_m : type) & = & \langle X_0, \ldots, X_m \rangle \\
\text{vars}(X_0 \texttt{ , } \ldots \texttt{ , } X_m : type \texttt{ := } E) & = & \langle X_0, \ldots, X_m \rangle \\
\text{vars}(ch_0, \ldots, ch_n) & = & \text{vars}(ch_0)\texttt{++}\ldots\texttt{++}\text{vars}(ch_n) \\
\text{vars}(\textbf{in } dl) & = & \text{vars}(dl) \\
\text{vars}(\textbf{out } dl) & = & \text{vars}(dl) \\
\text{vars}(\textbf{send } dl) & = & \text{vars}(dl) \\
\text{vars}(\textbf{receive } dl) & = & \text{vars}(dl) \\
\text{vars}(X_0 \texttt{ , } \ldots \texttt{ , } X_m) & = & \langle X_0, \ldots, X_m \rangle \\
\text{vars}(\texttt{?}X_0 \texttt{ , } \ldots \texttt{ , } \texttt{?}X_m) & = & \langle X_0, \ldots, X_m \rangle \\
\text{vars}(\texttt{_}, \ldots, \texttt{_}) & = & \epsilon \\
\text{vars}(\texttt{?_}, \ldots, \texttt{?_}) & = & \epsilon
\end{array}$$

– Input declaration: function *input* returns formal parameters declared as input parameters.

$$\begin{array}{rcl}
\text{input}(ch_0, \ldots, ch_n) & = & \text{input}(ch_0)\texttt{++}\ldots\texttt{++}\text{input}(ch_n) \\
\text{input}(\textbf{in } dl) & = & dl \\
\text{input}(\textbf{out } dl) & = & \emptyset \\
\text{input}(\textbf{send } dl) & = & \emptyset \\
\text{input}(\textbf{receive } dl) & = & \emptyset
\end{array}$$

– Output declaration: function *output* returns formal parameters declared as output parameters.

$$\begin{array}{rcl}
\text{output}(ch_0, \ldots, ch_n) & = & \text{output}(ch_0)\texttt{++}\ldots\texttt{++}\text{output}(ch_n) \\
\text{output}(\textbf{in } dl) & = & \emptyset \\
\text{output}(\textbf{out } dl) & = & dl \\
\text{output}(\textbf{send } dl) & = & \emptyset \\
\text{output}(\textbf{receive } dl) & = & \emptyset
\end{array}$$

– Receive declaration: function *receive* returns formal parameters declared as receive parameters.

$$\begin{array}{rcl}
\text{receive}(ch_0, \ldots, ch_n) & = & \text{receive}(ch_0)\texttt{++}\ldots\texttt{++}\text{receive}(ch_n) \\
\text{receive}(\textbf{in } dl) & = & \emptyset \\
\text{receive}(\textbf{out } dl) & = & \emptyset \\
\text{receive}(\textbf{send } dl) & = & \emptyset \\
\text{receive}(\textbf{receive } dl) & = & dl
\end{array}$$

– Send declaration: function *send* returns formal parameters declared as send parameters.

$$\begin{array}{rcl} \text{send}(ch_0, \ldots, ch_n) & = & \text{send}(ch_0)\texttt{++}\ldots\texttt{++}\text{send}(ch_n) \\ \text{send}(\textbf{in}\ dl) & = & \emptyset \\ \text{send}(\textbf{out}\ dl) & = & \emptyset \\ \text{send}(\textbf{send}\ dl) & = & dl \\ \text{send}(\textbf{receive}\ dl) & = & \emptyset \end{array}$$

– Parameter assignment: function *assign* maps input and receive formal parameters to their respective actual values.

$$\begin{array}{rcl} \text{assign}(\langle a_0, \ldots, a_n\rangle, \langle X_0, \ldots, X_n\rangle, \rho) & = & \text{assign}(a_0, X_0, \rho) \oplus \ldots \oplus \text{assign}(a_n, X_n, \rho) \\ \text{assign}(\_, X, \rho) & = & \{[]\} \\ \text{assign}(\texttt{?}\_, X, \rho) & = & \{[]\} \\ \text{assign}(\texttt{?}\,Y, X, \rho) & = & \{[]\} \\ \text{assign}(E, X, \rho) & = & \{[X \leftarrow e] \mid vars(E) \subseteq dom(\rho)\ \wedge\ \{E\}\ \rho\ \rightarrow_{\texttt{e}}\ e\} \\ \text{assign}(\textbf{any}\ T, X, \rho) & = & \{[X \leftarrow e] \mid e \in T\} \end{array}$$

– Parameter update: function *update* maps output and send actual parameters to the values of their respective formal parameters.

$$\begin{array}{rcl} \text{update}(\langle a_0, \ldots, a_n\rangle, \langle X_1, \ldots, X_n\rangle, \rho) & = & \text{update}(a_0, X_0, \rho) \oplus\ \ldots \oplus \text{update}(a_n, X_n, \rho) \\ \text{update}(\_, X, \rho) & = & [] \\ \text{update}(\texttt{?\_}, X, \rho) & = & [] \\ \text{update}(E, X, \rho) & = & [] \\ \text{update}(?Y, X, \rho) & = & \begin{cases} [Y \leftarrow \rho(X)] & \text{if } X \in dom(\rho) \\ [] & \text{otherwise} \end{cases} \end{array}$$

– Store restriction: operator **.** allows to return subsets of stores.

$$\rho\,\textbf{.}\,\langle X_0, ..., X_n\rangle = [X_0 \leftarrow \rho(X_0), ..., X_n \leftarrow \rho(X_n)] \text{ where } \{X_0, \ldots, X_n\} \subseteq dom(\rho)$$

Block semantics. We assume that blocks invoked inside an actor are defined as follows:

block B **[const** dl_{const}**]** (*ch*$_1$ **;** $\ldots$ **;** ch_n**) is**
  allocate $\ldots$
  perm dl_{perm} **,**
  temp dl_{temp}
    I_0
end block

Block B is allocated as follows: B**[**a_0 **,** $\ldots$ **,** a_m**]** **as** Bi (where a_0 **,** $\ldots$ **,** a_m are tha actual parameters correponding to the declaration list dl_{const}), and Bi can then be invoked as follows: Bi**(**ac_0 **;** $\ldots$ **;** ac_n**)**.

1. A block invocation starts by (1) assigning the default value (if any) to each constant, input, temporary, and permanent formal parameter, producing the store ρ_{init} (2) assigning the value of each input and constant actual parameter to its respective formal parameter, producing the store ρ_{arg} (3) assigning the value of each permanent parameter stored in the block instance's memory $\mu(\sigma.Bi)$ to its respective permanent variable, producing the store ρ_{perm}.
2. Then, it evaluates the body I_0 in the store $\rho_{global} \oplus \rho_{init} \oplus \rho_{arg} \oplus \rho_{perm}$ and the memory μ, producing the updated store ρ' and the updated memory μ'.

3. The execution terminates normally by updating the caller store and memory so that the value of each output formal parameter in ρ' is associated to its respective actual parameter and the value of each permanent parameter in ρ' is updated by its respective current value.

The semantic rule of block invocation is the following:

$$\frac{\{I_0\}\ \sigma.Bi, \rho_{body}, \mu \xrightarrow{\epsilon}_{\mathsf{i}}\ \rho', \mu'}{\{Bi(\ ac_0\,;\ldots;\ ac_n)\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}}\ \rho_{final},\ \mu_{final}}$$

where:

$$\begin{array}{rcl}
\rho_{init} & = & \text{init}\ (dl_{const}\ \texttt{++}\ input(ch_1, \ldots, ch_n)\texttt{++}dl_{perm}\texttt{++}dl_{temp},\ \rho_{global}) \\
\rho_{arg} & \in & \text{assign}(\langle a_0, \ldots, a_m\rangle,\ \text{vars}(dl_{const}),\ \rho)\ \oplus\ \text{assign}(\langle ac_0, \ldots, ac_n\rangle,\ \text{vars}(ch_1, \ldots, ch_n),\ \rho) \\
\rho_{perm} & = & \begin{cases} \mu(\sigma.Bi) & \text{if } \sigma.Bi \in dom(\mu) \\ [] & \text{otherwise} \end{cases} \\
\rho_{body} & = & \rho_{global} \oplus \rho_{init} \oplus \rho_{arg} \oplus \rho_{perm} \\
\rho_{update} & = & \text{update}(\langle ac_0, \ldots, ac_n\rangle,\ \text{vars}(ch_1, \ldots, ch_n),\ \rho') \\
\rho_{final} & = & \rho \oplus \rho_{update} \\
\mu_{final} & = & \mu' \oplus [\sigma.Bi \leftarrow \rho'\,.\,\text{vars}(dl_{perm})]
\end{array}$$

6.4 Dynamic semantics of systems

The executing of actor invocations is defined by septuple of the form "$\{instance\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{c}}\ \rho', \mu'$" that have the same features as the statements execution relation, except that *instance* denotes an actor instance and σ is its corresponding stack.

6.4.1 Block invocation

We assume that a blocks invoked inside a system are defined as follows:

block B **[const** dl_{const}**]** $(ch_1\,;\ldots;ch_m)\{ch'_1\,;\ldots;ch'_n\}$ **is**
  allocate ...
  perm dl_{perm}**,**
  temp dl_{temp}
    I_0
end block

Block B is allocated as follows: B**[**a_0**,**$\ldots$**,**a_p**]** **as** Bi (where a_0**,**$\ldots$**,**a_m are the actual parameters correponding to the declaration list dl_{const}), and Bi can then be invoked as follows: Bi **(**$ac_0\,;\ldots;ac_m$**)** **{**$ac'_0\,;\ldots;ac'_n$**}**.

The semantics of blocks invoked inside a system are slightly different from those of blocks invoked inside an actor because of the occurrence of send and receive parameters. The semantic rule of block invocation is the following:

$$\frac{\{I_0\}\ \sigma.Bi,\ \rho_{body},\ \mu\ \xrightarrow{\epsilon}_{\mathsf{i}}\ \rho',\ \mu'}{\{Bi(ac_0\,;\ldots;ac_m)\{ac'_0\,;\ldots;ac'_n\}\}\ \sigma, \rho, \mu\ \xrightarrow{\epsilon}_{\mathsf{c}}\ \rho_{final},\ \mu_{final}}$$

where:

$$
\begin{array}{rcl}
\rho_{init} & = & \text{init } (dl_{const} \texttt{ ++ } input(ch_1, \ldots, ch_m) \texttt{++} receive(ch'_1, \ldots, ch'_n) \texttt{++} dl_{perm} \texttt{++} dl_{temp},\ \rho_{global}) \\
\rho_{arg} & \in & \text{assign}(\langle a_0, \ldots, a_m \rangle,\ \text{vars}(dl_{const}),\ \rho) \\
 & & \oplus\text{assign}(\langle ac_0, \ldots, ac_n \rangle,\ \text{vars}(ch_1, \ldots, ch_m),\ \rho) \\
 & & \oplus\text{assign}(\langle ac'_0, \ldots, ac'_p \rangle,\ \text{vars}(ch'_1, \ldots, ch'_n),\ \rho) \\
\rho_{perm} & = & \begin{cases} \mu(\sigma.Bi) & \text{if } \sigma.Bi \in dom(\mu) \\ [] & \text{otherwise} \end{cases} \\
\rho_{body} & = & \rho_{global} \oplus \rho_{init} \oplus \rho_{arg} \oplus \rho_{perm} \\
\rho_{update} & = & \text{update}(\langle ac_0, \ldots, ac_n \rangle,\ \text{vars}(ch_1, \ldots, ch_m),\ \rho') \\
 & & \oplus\text{update}(\langle ac'_0, \ldots, ac'_p \rangle,\ \text{vars}(ch'_1, \ldots, ch'_n),\ \rho') \\
\rho_{final} & = & \rho \oplus \rho_{update} \\
\mu_{final} & = & \mu' \oplus [\sigma.Bi \leftarrow \rho' \centerdot \text{vars}(dl_{perm})]
\end{array}
$$

6.4.2 Environment invocation

We assume that an environment is defined as follows:

environment N **[const** dl_{const}**]** $(ch_1 \mid \ldots \mid ch_n)$ **is**
  allocate ...
  perm dl_{perm} **,**
  temp dl_{temp}
    I_0
end environment

Environment N is allocated as follows: N **[**a_0 **,** ... **,** a_m**]** **as** Ni, and Ni can then be invoked as follows: Ni $(ac_0 \mid \ldots \mid ac_n)$.

The execution of environment invocations starts by the selection of one actual channel among those passed to the environment. If the actual channel is connected (i.e., the set of its variables is not empty), then the execution continues by checking whether the body of the environment defines a signal statement corresponding to the channel under consideration, in which case the body I_0 executes in the current store and memory and terminates normally producing an updated store and memory, and holding a label. If the channel is not connected or the body I_0 does not contain a signal statement corresponding to the actual channel, then I_0 is not executed.
The semantic rule of environment invocation is the following:

$$
\frac{\begin{array}{l} i \in [0..n] \qquad vars(ac_i) \neq \epsilon \\ \{I_0\}\ \sigma.Ni, \rho_{body}, \mu \xrightarrow{\ell}_{\mathsf{i}}\ \rho', \mu' \qquad vars(\ell) = vars(ch_i) \end{array}}{\{Ni\ (ac_0 \mid \ldots \mid ac_n)\}\ \sigma, \rho, \mu \xrightarrow{ac_i}_{\mathsf{c}}\ \rho_{final},\ \mu_{final}}
$$

where:

$$
\begin{array}{rcl}
\rho_{init} & = & \text{init } (dl_{const} \texttt{ ++ } input(ch_1, \ldots, ch_n) \texttt{++} dl_{perm} \texttt{++} dl_{temp},\ \rho_{global}) \\
\rho_{arg} & \in & \text{assign}(\langle a_0, \ldots, a_m \rangle,\ \text{vars}(dl_{const}),\ \rho) \\
 & & \oplus\text{assign}(\langle ac_0, \ldots, ac_n \rangle, \text{vars}(ch_1, \ldots, ch_n),\ \rho) \\
\rho_{perm} & = & \begin{cases} \mu(\sigma.Ni) & \text{if } \sigma.Ni \in dom(\mu) \\ [] & \text{otherwise} \end{cases} \\
\rho_{body} & = & \rho_{global} \oplus \rho_{init} \oplus \rho_{arg} \oplus \rho_{perm} \\
\rho_{update} & = & \text{update}(\langle ac_0, \ldots, ac_n \rangle, \text{vars}(ch_1, \ldots, ch_n), \rho') \\
\rho_{final} & = & \rho \oplus \rho_{update} \\
\mu_{final} & = & \mu' \oplus [\sigma.Ni \leftarrow \rho' \centerdot \text{vars}(dl_{perm})]
\end{array}
$$

6.4.3 Medium invocation

Medium invocation has exactly the same semantics as environment invocation, except that input parameters (resp., output parameters) are replaced by receive parameters (resp., send parameters).

6.4.4 Dynamic semantics of system

The execution of a system is governed by the parallel execution of the block instances invoked by the system. It is defined by triples of the form "$\mu \xrightarrow{\ell}_{\mathtt{s}} \mu'$" where μ and μ' are memories, and ℓ is a transition of the system. The LTS of the system is constructed as follows. States are represented by the memories of all actors invoked inside the system, and the initial state maps variables of actor memories to their respective initialization values. Transitions are represented by block invocations and have the form "$Bi\,\mathtt{(}ac_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac_n\mathtt{)\{}ac'_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac'_m\mathtt{\}}$". Transition relation of the form "$\mu \xrightarrow{Bi\,\mathtt{(}ac_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac_n\mathtt{)\{}ac'_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac'_m\mathtt{\}}}_{\mathtt{s}} \mu'$" means that the combined execution of block Bi together with its connected environments and mediums in the memory μ produces the updated memory μ'. More specifically, the semantics of the system defined below are the following.

system S $\mathtt{(}dl_{sys}\mathtt{)}$ **is**
  allocate ...
  temp dl_{temp}
  network
    $block_0\,\mathtt{,}\,\ldots\,\mathtt{,}\,block_p$
  constrainedby
    $environment_0\,\mathtt{,}\,\ldots\,\mathtt{,}\,environment_q$
  connectedby
    $medium_0\,\mathtt{,}\,\ldots\,\mathtt{,}\,medium_r$
end system

where:

$$\begin{array}{rcl} block & ::= & Bi\ \mathtt{(}ac_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac_n\mathtt{)\{}ac'_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac'_m\mathtt{\}} \\ environment & ::= & Ni\ \mathtt{(}ac_0\,\mathtt{|}\,\ldots\,\mathtt{|}\,ac_n\mathtt{)} \\ medium & ::= & Mi\ \mathtt{\{}ac_0\,\mathtt{|}\,\ldots\,\mathtt{|}\,ac_n\mathtt{\}} \end{array}$$

Given a block instance of the form "$Bi_i\,\mathtt{(}ac_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac_m\mathtt{)\{}ac'_0\,\mathtt{;}\,\ldots\,\mathtt{;}\,ac'_n\mathtt{\}}$", we define the following symbols:

- $\mathcal{R}_i$ denotes the set of indexes of medium instances that have send actual channels connected to receive actual channels of $block_i$.
- $\mathcal{S}_i$ denotes the set of indexes of medium instances that have receive actual channels connected to send actual channels of $block_i$.
- $\mathcal{RS}_{ij}$ denotes the set of actual channels that are both receive actual channels of $block_i$ and send actual channels of $medium_j$.
- $\mathcal{SR}_{ij}$ denotes the set of actual channels that are both send actual channels of $block_i$ and receive actual channels of $medium_j$.
- $\mathcal{I}_i$ denotes the set of indexes of environment instances that have output actual channels connected to input actual channels of $block_i$.
- $\mathcal{O}_i$ denotes the set of indexes of environment instances that have input actual channels connected to output actual channels of $block_i$.
- $\mathcal{IO}_{ij}$ denotes the set of actual channels that are both input actual channels of $block_i$ and output actual channels of $environment_j$.

– $\mathcal{OI}_{ij}$ denotes the set of actual channels that are both output actual channels of $block_i$ and input actual channels of $environment_j$.

Since static semantics forces two actors to be connected via exactly one actual channel, sets $\mathcal{RS}_{ij}$, $\mathcal{SR}_{ij}$, $\mathcal{IO}_{ij}$, and $\mathcal{OI}_{ij}$ contain at most one element.
Function *any* assigns arbitrary values to the parameters of the system:

$$\begin{array}{rcl} \text{any}(vd_0, \ldots, vd_n, \rho) & = & \text{any}(vd_0, \rho) \oplus \ldots \oplus \text{any}(vd_n, \rho) \\ \text{any}(X_0, \ldots, X_n : T, \rho) & = & \{[X_0 \leftarrow e_0, ..., X_n \leftarrow e_n] \mid e_0, \ldots, e_n \in T\} \end{array}$$

The execution of the system S starts by assigning arbitrary values to the inputs and temporary variables of S. Then, for each block instance Bi_i $(i \in 0..p)$, it executes:

1. all medium instances whose index is in $\mathcal{R}_i$, producing the store $\rho_{\mathcal{R}}$ and the memory $\mu_{\mathcal{R}}$
2. all environment instances whose index is in $\mathcal{I}_i$, producing the store $\rho_{\mathcal{I}}$ and the memory $\mu_{\mathcal{I}}$
3. the block call
4. all environment instances whose index is in $\mathcal{O}_i$, producing the store $\rho_{\mathcal{O}}$ and the memory $\mu_{\mathcal{O}}$
5. all medium instances whose index is in $\mathcal{S}_i$, producing the store ρ' and the memory μ'.

The execution of the system terminates by updating the current memory μ and passing a label $Bi_i(ac_0; \ldots; ac_m)\{ac'_0; \ldots; ac'_n\}$ to its context. The semantic rule of the system is the following:

$$\frac{\begin{array}{l} i \in 0..p \qquad \rho \in any(dl_{sys} \texttt{++} dl_{temp},\ []) \\ (\forall j \in \mathcal{R}_i) \qquad \ell_j = \mathcal{RS}_{ij} \qquad \{medium_j\}\ \epsilon,\ \rho,\ \mu \xrightarrow{\ell_j}_{\mathsf{c}} \rho_{\mathcal{R}_j},\ \mu_{\mathcal{R}_j} \\ \rho_{\mathcal{R}} = \rho \oplus \bigoplus_{j \in \mathcal{R}_i} \rho_{\mathcal{R}_j} \qquad \mu_{\mathcal{R}} = \mu \oplus \bigoplus_{j \in \mathcal{R}_i} \mu_{\mathcal{R}_j} \\ (\forall j \in \mathcal{I}_i) \qquad \ell_j = \mathcal{IO}_{ij} \qquad \{environment_j\}\ \epsilon,\ \rho_{\mathcal{R}},\ \mu_{\mathcal{R}} \xrightarrow{\ell_j}_{\mathsf{c}} \rho_{\mathcal{I}_j},\ \mu_{\mathcal{I}_j} \\ \rho_{\mathcal{I}} = \rho_{\mathcal{R}} \oplus \bigoplus_{j \in \mathcal{I}_i} \rho_{\mathcal{I}_j} \qquad \mu_{\mathcal{I}} = \mu_{\mathcal{R}} \oplus \bigoplus_{j \in \mathcal{I}_i} \mu_{\mathcal{I}_j} \\ \{block_i\}\ \epsilon,\ \rho_{\mathcal{I}},\ \mu_{\mathcal{I}} \xrightarrow{\epsilon}_{\mathsf{c}} \rho_i,\ \mu_i \\ (\forall j \in \mathcal{O}_i) \qquad \ell_j = ?\mathcal{OI}_{ij} \qquad \{environment_j\}\ \epsilon,\ \rho_i,\ \mu_i \xrightarrow{\ell_j}_{\mathsf{c}} \rho_{\mathcal{O}_j},\ \mu_{\mathcal{O}_j} \\ \rho_{\mathcal{O}} = \rho_i \oplus \bigoplus_{j \in \mathcal{O}_i} \rho_{\mathcal{O}_j} \qquad \mu_{\mathcal{O}} = \mu_i \oplus \bigoplus_{j \in \mathcal{O}_i} \mu_{\mathcal{O}_j} \\ (\forall j \in \mathcal{S}_i) \qquad \ell_j = ?\mathcal{SR}_{ij} \qquad \{medium_j\}\ \epsilon,\ \rho_{\mathcal{O}},\ \mu_{\mathcal{O}} \xrightarrow{\ell_j}_{\mathsf{c}} \rho_{\mathcal{S}_j},\ \mu_{\mathcal{S}_j} \\ \rho_{\mathcal{S}} = \rho_{\mathcal{O}} \oplus \bigoplus_{j \in \mathcal{S}_i} \rho_{\mathcal{S}_j} \qquad \mu_{\mathcal{S}} = \rho_{\mathcal{O}} \oplus \bigoplus_{j \in \mathcal{S}_i} \mu_{\mathcal{S}_j} \qquad \mu' = \mu_{\mathcal{S}} \end{array}}{\mu \xrightarrow{Bi_i(label(\langle ac_0,\ldots,ac_m\rangle,\rho_{\mathcal{S}})\{label(\langle ac'_0,\ldots,ac'_n\rangle,\rho_{\mathcal{S}})\}}_{\mathsf{s}} \mu'}$$

Where *label* is a function defined as follows:

$$\begin{array}{rcl} \text{label}(\langle ac_0, \ldots, ac_m \rangle, \rho) & = & \text{label}(ac_0, \rho) + \ldots + \text{label}(ac_m, \rho) \\ \text{label}(\langle a_0, \ldots, a_m \rangle, \rho) & = & \text{label}(a_0, \rho) + \ldots + \text{label}(a_m, \rho) \\ \text{label}(X, \rho) & = & \begin{cases} \rho(X) & \text{if } X \in \text{vars}(dl_{sys}) \\ \_ & \text{otherwise} \end{cases} \\ \text{label}(\textbf{any}\ T, \rho) & = & \_ \\ \text{label}(\_, \rho) & = & \_ \\ \text{label}(?X, \rho) & = & \begin{cases} ?\rho(X) & \text{if } X \in \text{vars}(dl_{sys}) \\ ?\_ & \text{otherwise} \end{cases} \\ \text{label}(?\_, \rho) & = & ?\_ \end{array}$$

The semantics of the whole network are obtained by interleaving all possible execution of $Bi_0, \ldots Bi_p$.

6.5 Dynamic semantics of programs

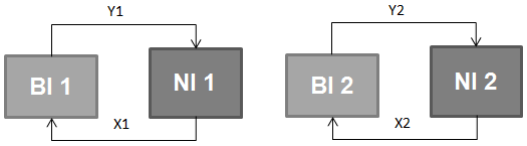
A GRL program can import other programs. The semantics of a program are defined as the semantics of a particular system in the program or in an imported program. Such a system is called the *main* system of the program. By default, the main system of the program is a system named "Main". Alternatively, tools will provide the user with the possibility to specify another system as the main system using command-line options.

7 Basic Examples

This section presents some examples of systems modeled in GRL.

7.1 Independent blocks with independent environments

In this example, environment Ni_1 (respectively Ni_2) ensures that the input X_1 of block Bi_1 (respectively the input X_2 of block Bi_2) is always larger than the output Y_1 (respectively Y_2) at the previous cycle.

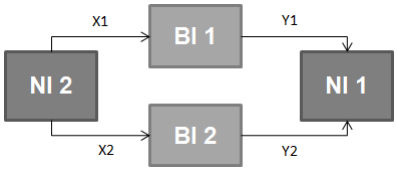



```
system S (X1,X2:nat, Y1,Y2:nat) is
    allocate B as Bi1, B as Bi2,
             N as Ni1, N as Ni2
    network
      Bi1(X1;?Y1),
      Bi2(X2;?Y2)
    constrainedby
      Ni1(Y1|?X1),
      Ni2(Y2|?X2)
end system

block B (in X:nat; out Y:nat) is
    Y := X
end block

environment N (in Y:nat | out X:nat) is
    perm last_Y : nat := 0
      select
        on Y -> last_Y := Y
      []
        on ?X -> X := any nat where X ≥ last_Y
      end select
end environment
```

7.2 Independent blocks with shared environments

In this example:

- Environment Ni_1 ensures that Bi_1 and Bi_2 are executed alternately.
- Environment Ni_2 ensures that the inputs X_1 and X_2 of blocks Bi_1 and Bi_2 (respectively) can vary (increase or decrease) by at most one unit at each cycle.



```
system S (X1,X2:nat, Token1,Token2:bool, Y1,Y2:nat) is
    allocate B as Bi1, B as Bi2,
             N1 as Ni1, N2 as Ni2
    network
      Bi1(X1;Token1;?Y1),
      Bi2(X2;Token2;?Y2)
    constrainedby
      Ni1(Y1|Y2),
      Ni2(?X1|?X2)
end system

block B (in X:nat; in Token:bool; out Y:nat) is
    perm lastY:nat := 0
    if (Token)
        Y := X; lastY := Y
    else
        Y := lastY
    end if
end block


environment N1 (in Y1:nat | in Y2:nat | out Token1:bool | out Token2:bool) is
    perm last1 : bool := false
      if not(last1) then
        on Y1 -> Token1 := true; last1 := true
      else
        on Y2 -> Token2 := true; last1 := false
      end if
end environment

environment N2 (out X1:nat | out X2:nat) is
    perm last1,last2:nat := 0
      select
        on ?X2 -> X2 := any nat where X2+1 >= last2 and X2 <= last2+1;
                  last2 := X2
      []
        on ?X1 -> X1 := any nat where X1+1 >= last1 and X1 <= last1+1;
                  last1 := X1
      end select
end environment
```

7.3 Network of blocks communicating via a medium

In this example:

- In every execution cycle, block Bi_1 receives from the medium Mi the value of its output Y_1 in the last execution cycle. Then, after computing, Bi_1 sends the new value of Y_1 to Mi.
- In every execution cycle, block Bi_2 receives from the medium Mi the sum of the output Y_1 in the previous execution cycles. Then, after computing, Bi_2 sends a boolean signal to allow or not Mi to initlize the sum.


```
system S (X1,X2:nat, Y1,Y2:nat) is
    allocate B1 as Bi1, B2[128] as Bi2,
             M as Mi
    network
      Bi1(X1;?Y1){Z1;?W1},
      Bi2(X2;?Y2){Z2;?W2}
    connectedby
      Mi{W1|W2|?Z1|?Z2}
end system

block B1 (in X1:nat; out Y1:nat) {receive Z1:nat; send W1:nat} is
    Y1 := X1 + Z1;
    W1 := Y1
end block


block B2 [const seuil:nat](in X2:nat; out Y2:nat)
          {receive Z2:nat; send W2:bool} is
   Y2 := X2;
   if (Z2 >= seuil) then
       W2 := true
   else
       W2 := false
   end if
end block

medium M {receive W1:nat | receive W2:nat | send Z1:nat | send Z2:nat} is
  perm buf1,buf2 := 0
    select
      on W1 -> buf1 := W1;
                buf2 := buf2+W1
    []
      on ?Z1 -> Z1 := buf1;
                buf1 := 0
    []
      on W2 -> if (W2 == true) then
                  buf2 := 0
                end if
    []
      on ?Z2 -> Z2 := buf2
    end select

end medium
```

8 Conclusion

In this paper, we have defined the syntax and semantics of the GRL language designed to fulfill the need of a general-purpose modelling approach to provide industrial design process with formal verification to ensure the correctness of GALS systems construction. GRL programs draw an abstraction of those systems behaviour by means of finite state machines or labelled transition systems (LtSs, for short). The behaviour of each actor of the system (block, environment, medium) is modeled separately and then all models are composed in parallel, either in a flat or hierarchical manner.

A parser of the language has already been developed using Syntax [7] and LNT technology [11] for compiler construction. LNT is also an input language of Cadp [12], a widely spread toolbox for concurrent systems construction, which offers a large range of functionalities, including interactive simulation, formal verification, and testing. A translator from GRL to LNT has been designed and implemented but is out of the scope of this report. This way, LNT models can be generated and system properties can be verified using efficient verification methods such as model-checking, equivalence-checking, and compositional verification.

References

[1] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In Mary S. Van Deusen and Bernard Lang, editors, *POPL*, pages 85–98. ACM Press, 1993.

[4] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[5] Gérard Berry and Ellen Sentovich. Multiclock esterel. 2144:110–125, 2001.

[6] David S. Bormann. Asynchronous wrapper for heterogeneous systems. In *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, ICCD '97, pages 307–, Washington, DC, USA, 1997. IEEE Computer Society.

[7] Pierre Boullier, Philippe Deschamps, and Benoit Sagot. *Le Sysème SYNTAX Analyse Déterministe Manuel d'Utilisation et de Mise en Oeuvre Sous UNIX*. Paris, France, 2008.

[8] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Frédéric Lang, Christine McKinty, Vincent Powazny, Wendelin Serwe, and Gideon Smeding. *Reference Manual of the LOTOS NT to LOTOS Translator*. Grenoble, France, 2013.

[9] Nicolas Coste, Holger Hermanns, Etienne Lantreibecq, and Wendelin Serwe. Towards Performance Prediction of Compositional Models in Industrial GALS Designs. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, Grenoble, France, 2009. Saddek Bensalem, Springer Verlag.

[10] Frédéric Doucet, Massimiliano Menarini, Ingolf H. Kr´uger, Rajesh K. Gupta, and Jean-Pierre Talpin. A verification approach for gals integration of synchronous components. *Electr. Notes Theor. Comput. Sci.*, 146(2):105–131, 2006.

[11] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler construction using lotos nt. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer, 2002.

[12] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[13] Hubert Garavel, Gwen Salaun, and Wendelin Serwe. On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP. *Science of Computer Programming*, 2009.

[14] Hubert Garavel and Damien Thivolle. Verification of gals systems by combining synchronous languages and process calculi. In Corina S. Pasareanu, editor, *SPIN*, volume 5578 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2009.

[15] Nicolas Halbwachs and Louis Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, ACSD '06, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society.

[16] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[17] Fatma Jebali, Frédéric Lang, and Radu Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. Rapport de recherche, April 2014.

[18] Etienne Lantreibecq and Wendelin Serwe. Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit using CADP. In *Formal Methods for Industrial Critical Systems*, Trento, Italie, 2011.

[19] Avinash Malik, Zoran Salcic, Partha S. Roop, and Alain Girault. Systemj: A gals language for system level design. *Comput. Lang. Syst. Struct.*, 36(4):317–344, December 2010.

[20] Mohammad Reza Mousavi, Paul Le Guernic, Jean-Pierre Talpin, Sandeep Kumar Shukla, and Twan Basten. Modeling and validating globally asynchronous design in synchronous frameworks. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10384–, Washington, DC, USA, 2004. IEEE Computer Society.

[21] S. Ramesh. Communicating reactive state machines: Design, model and implementation. 1998.

Contents

A Lexical Structure

```
Classes

        ANY             = #000..#377; -- accept 8-bit characters
        WHITE_SPACE     = BS + HT + LF + VT + FF + CR + NL + SP ;
        STRING_CHAR     = ANY - (QUOTE + "\\" + EOL) ;
        ESCAPE_CHAR     = "a" + "b" + "f" + "n" + "r" + "t" + "v" + "\\" + "'" + "\"" + "?" ;
        OCTAL           = "0".."7" ;
        HEXA            = "a".."f" + "A".."F" + DIGIT ;
        ANY_BUT_EOL     = ANY - EOL ;

--------------------------------------------------------------------------------

Abbreviations

        SEPARATOR       = WHITE_SPACE ;
        COMMENT         = "(" "*"&True {ANY}* "*" ")" ;
        LINE_COMMENT    = "-" "-"&True {ANY_BUT_EOL}* EOL ;

--------------------------------------------------------------------------------

Tokens

        Comments        = {SEPARATOR | COMMENT | LINE_COMMENT}+ ;
                          Unbounded Context All ;
                          Priority Shift > Reduce ;

        %IDENTIFIER     = LETTER { {"_"}* (LETTER | DIGIT) }* ;
                          Priority Shift > Reduce ;

        %STRING         = -QUOTE {
                                  STRING_CHAR |
                                  "\\" ESCAPE_CHAR |
                                  "\\" OCTAL&True [OCTAL [OCTAL]]
                                 }*
                          -QUOTE ;

        %CHARACTER      = "'" (^"'\n\\" | "\\"&True ANY_BUT_EOL | "\\"&True OCTAL&True [OCTAL [OCTAL]]) "'" ;

        %INTEGER        = "0" X {HEXA}+ | "0" {OCTAL}*  |"123456789" {DIGIT}* ;
                          Priority Shift > Reduce ;
```

B Concrete Grammar

```
*============================================================
*                           Program
*============================================================

<program> = "program" <identifier> <imported-programs> "is"
<definition-list>
    "end" "program" ;

<imported-programs> = ;
<imported-programs> = "(" <identifier-list> ")" ;

<definition-list> = ;
<definition-list> = <definition>  <definition-list> ;

<definition> = <system> ;
<definition> = <block> ;
<definition> = <environment> ;
<definition> = <medium> ;
<definition> = <constant> ;
<definition> = <type-definition> ;

*============================================================
*                           System
*============================================================

<system> = "system" <identifier> <system-inout-parameters> "is"
                 <system-contents>
           "end" "system" ;

<system-inout-parameters> = ;
<system-inout-parameters> = "(" <system-inout-parameters-list> ")";

<system-inout-parameters-list> = <inout-parameters-non-init> ;
<system-inout-parameters-list> = <inout-parameters-non-init> ";" <system-inout-parameters-list> ;

<system-contents> = <instance-list> <system-temp-variables> <network-spec> <env-spec> <medium-spec>;

<system-temp-variables> = ;
<system-temp-variables> = "temp" <variable-declaration-list-non-init> ;

<network-spec> = "network" <com-block-instance-list> ;

<com-block-instance-list> = <com-block-instance> ;
<com-block-instance-list> = <com-block-instance> "," <com-block-instance-list>;


<com-block-instance> = <identifier> "(" <block-argument-list> ")";
<com-block-instance> = <identifier> "{" <block-argument-list> "}";
<com-block-instance> = <identifier> "(" <block-argument-list> ")"  "{" <block-argument-list> "}";

<env-spec> = ;
<env-spec> = "constrainedby" <env-instance-list>;

<env-instance-list> = <env-instance> ;
<env-instance-list> = <env-instance> "," <env-instance-list> ;

<env-instance> = <identifier> "(" <env-medium-argument-list> ")" ;

<medium-spec> = ;
<medium-spec> = "connectedby" <medium-instance-list>;

<medium-instance-list> = <medium-instance> ;
<medium-instance-list> = <medium-instance> "," <medium-instance-list> ;

<medium-instance> = <identifier> "{" <env-medium-argument-list> "}" ;

*============================================================
*                           Block
*============================================================

<block> = "block" <identifier> <const-parameters> <block-inout-parameters> <block-com-parameters> "is"
```

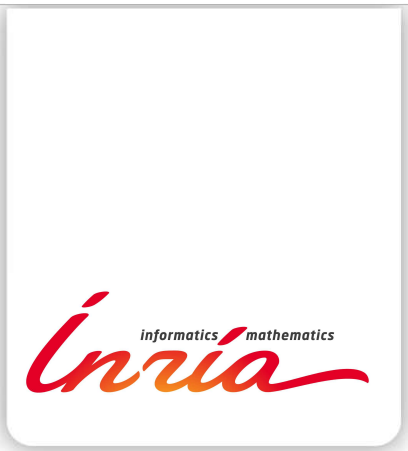
```
                <block-body>
          "end" "block" ;

<block-body> = <block-medium-env-contents>;
<block-body> = <external-pragma>;

<external-pragma> = "!c" <string> ;
<external-pragma> = "!lnt" <string> ;

<block-inout-parameters> = ;
<block-inout-parameters> = "(" <block-inout-parameters-list> ")";

<block-inout-parameters-list> = <inout-parameters>;
<block-inout-parameters-list> = <inout-parameters> ";" <block-inout-parameters-list> ;

<block-com-parameters> =  ;
<block-com-parameters> = "{" <block-com-parameters-list> "}";

<block-com-parameters-list> = <com-parameters> ;
<block-com-parameters-list> = <com-parameters> ";" <block-com-parameters-list>;

<instance-list> = ;
<instance-list> = "allocate" <instance-declaration-list>;

<instance-declaration-list> = <instance-declaration> ;
<instance-declaration-list> = <instance-declaration> "," <instance-declaration-list> ;

<instance-declaration> = <identifier> <const-arguments> "as" <instance-identifier>;

<block-medium-env-contents> = <instance-list> <local-variables-list> <body> ;

<body> = <statement> ;

*============================================================
*                            Environment
*============================================================

<environment> = "environment" <identifier> <const-parameters> <env-inout-parameters> "is"
                        <block-medium-env-contents>
                "end" "environment" ;

<env-inout-parameters> = ;
<env-inout-parameters> = "(" <env-inout-parameters-list> ")";

<env-inout-parameters-list> = <inout-parameters-non-init> ;
<env-inout-parameters-list> = <inout-parameters-non-init> "|" <env-inout-parameters-list>;

*============================================================
*                            Medium
*============================================================

<medium> = "medium" <identifier> <const-parameters> <medium-com-parameters> "is"
                  <block-medium-env-contents>
           "end" "medium" ;

<medium-com-parameters> = ;
<medium-com-parameters> = "{" <medium-com-parameters-list> "}";

<medium-com-parameters-list> = <com-parameters> ;
<medium-com-parameters-list> = <com-parameters> "|" <medium-com-parameters-list>;

*============================================================
*                            Formal parameters
*============================================================

<const-parameters> = ;
<const-parameters> = "[" "const" <variable-declaration-list> "]";

<inout-parameters> = "in" <variable-declaration-list>;
<inout-parameters> = "out" <variable-declaration-list-non-init> ;

<inout-parameters-non-init> = "in" <variable-declaration-list-non-init> ;
<inout-parameters-non-init> = "out" <variable-declaration-list-non-init> ;
```

```
<com-parameters> =  <receive-parameters>;
<com-parameters> =  <send-parameters> ;

<receive-parameters> = "receive" <variable-declaration-list-non-init> ;

<send-parameters> = "send" <variable-declaration-list-non-init> ;

*============================================================
*                           Variable Declaration
*============================================================

<local-variables-list> = ;
<local-variables-list> = <non-empty-local-variables-list> ;

<non-empty-local-variables-list> = <perm-temp-local-variables-list> ;
<non-empty-local-variables-list> = <perm-temp-local-variables-list> <non-empty-local-variables-list> ;

<perm-temp-local-variables-list> = "perm" <variable-declaration-list> ;
<perm-temp-local-variables-list> = "temp" <variable-declaration-list> ;

<variable-declaration-list> = <variable-declaration> ;
<variable-declaration-list> = <variable-declaration> "," <variable-declaration-list> ;

<variable-declaration> = <variable-list> ":" <type> <opt-initialization> ;

<opt-initialization> = ;
<opt-initialization> = ":=" <expression> ;

<variable-declaration-list-non-init> = <variable-declaration-non-init> ;
<variable-declaration-list-non-init> = <variable-declaration-non-init> "," <variable-declaration-list-non-init> ;

<variable-declaration-non-init> = <variable-list> ":" <type>;

<variable-list> = <variable> ;
<variable-list> = <variable> "," <variable-list> ;

<variable> = <identifier> ;

*============================================================
*                           Arguments
*============================================================

<const-arguments> = ;
<const-arguments> = "[" <argument-list> "]";

<block-argument-list> = <argument-list> ;
<block-argument-list> = <argument-list> ";" <block-argument-list> ;

<env-medium-argument-list> = <argument-list> ;
<env-medium-argument-list> = <argument-list> "|" <env-medium-argument-list> ;

<argument-list> = <argument> ;
<argument-list> = <argument> "," <argument-list> ;

<argument> = <expression> ;
<argument> = "?" <variable> ;
<argument> = "_" ;
<argument> = "?" "_" ;
<argument> = "any" <type> ;

*============================================================
*                           Global Constant
*============================================================

<constant> = "constant" <identifier> ":" <type> "is"
                  <expression>
             "end" "constant" ;

*============================================================
*                           Statements
*============================================================
```

```
<statement> = <basic-statement> ;
<statement> = <basic-statement> ";" <statement> ;
<statement> = "on" <com-variable-list> "->" <statement>;

<basic-statement> = "null" ;
<basic-statement> = <variable> ":=" <expression> ;
<basic-statement> = "if" <expression> "then" <statement> <elsif-statement-list> <else-statement> "end" "if" ;
<basic-statement> = "while" <expression> "loop" <statement> "end" "loop" ;
<basic-statement> = "for" <statement> "while" <expression> "by" <statement> "loop" <statement> "end" "loop" ;
<basic-statement> = <variable> "." <variable> ":=" <expression>;
<basic-statement> = <variable> "[" <expression> "]" ":=" <expression>;
<basic-statement> = "case" <expression> "is" <case-item-list> "end" "case";
<basic-statement> = <nondeterministic-assign> <opt-clause>;
<basic-statement> = <instance-identifier> "(" <block-argument-list> ")" ;

<elsif-statement-list> = ;
<elsif-statement-list> =  "elsif" <expression> "then" <statement> <elsif-statement-list> ;

<else-statement> = ;
<else-statement> = "else" <statement> ;

<case-item-list> = <case-item>;
<case-item-list> = <case-item> "|" <case-item-list> ;

<case-item> = <unary-expression> "->" <statement> ;
<case-item> = "any" "->" <statement>;

<basic-statement> = "select" <select-item-list> "end" "select";

<select-item-list> = <statement>;
<select-item-list> = <statement> "[]" <select-item-list>;

<nondeterministic-assign> = <variable> ":=" "any" <type> ;

<opt-clause> = ;
<opt-clause> = "where" <expression>;

<instance-identifier> = <identifier> ;

<com-variable-list> = <com-variable> ;
<com-variable-list> = <com-variable> "," <com-variable-list> ;
<com-variable> = <variable>;
<com-variable> = "?" <variable>;

*============================================================
*                           Expressions
*============================================================

<expression> = <binary-expression> ;

<binary-expression> = <unary-expression> ;
<binary-expression> = <binary-expression> <binary-operator> <unary-expression> ;

<unary-expression> = <basic-expression> ;
<unary-expression> = <unary-operator> <unary-expression> ;

<basic-expression> = <identifier> "[" <expression> "]" ;
<basic-expression> = <identifier> "." <variable> ;
<basic-expression> = <identifier> ;
<basic-expression> = <identifier> "(" ")" ;
<basic-expression> = <identifier> "(" <expression-list> ")";
<basic-expression> = <literal-const> ;
<basic-expression> = <literal-const> "of" <type> ;
<basic-expression> = <enum-identifier> "of" <type> ;
<basic-expression> = "(" <expression> ")" ;

<expression-list> = <expression> ;
<expression-list> = <expression> "," <expression-list> ;

*============================================================
*                           Types
*============================================================
```

```
<type> = "bool" ;
<type> = "nat" ;
<type> = "nat16" ;
<type> = "nat32" ;
<type> = "int" ;
<type> = "int16" ;
<type> = "int32" ;
<type> = "char" ;
<type> = "string" ;
<type> = <identifier> ;

<type-definition> = "type" <identifier> "is" <type-expression> "end" "type";

<type-expression> = "array" "[" <unary-expression> ".." <unary-expression> "]" "of" <type> ;
<type-expression> = "range" <unary-expression> ".." <unary-expression> ;
<type-expression> = "enum"  <enum-list> ;
<type-expression> = "record" <record-list> ;

<enum-list> = <enum-identifier> ;
<enum-list> = <enum-identifier> "," <enum-list> ;

<enum-identifier>  = <identifier> ;

<record-list> = <record-variable> ;
<record-list> = <record-variable> "," <record-list> ;

<record-variable>  = <variable> ":" <type> ;

*============================================================
*                           Operators
*============================================================

<binary-operator> = "+" ;
<binary-operator> = "-" ;
<binary-operator> = "*" ;
<binary-operator> = "/" ;
<binary-operator> = "%" ;
<binary-operator> = "^" ;
<binary-operator> = "<" ;
<binary-operator> = ">" ;
<binary-operator> = "<=" ;
<binary-operator> = ">=" ;
<binary-operator> = "==" ;
<binary-operator> = "!=" ;
<binary-operator> = "or" ;
<binary-operator> = "and" ;
<binary-operator> = "implies" ;
<binary-operator> = "equ" ;
<binary-operator> = "xor" ;

<unary-operator> = "+" ;
<unary-operator> = "-" ;
<unary-operator> = "not" ;
<unary-operator> = "abs" ;

*============================================================
*                           Identifiers
*============================================================

<identifier> = %IDENTIFIER ;

<identifier-list> = <identifier> ;
<identifier-list> = <identifier> "," <identifier-list> ;


*============================================================
*                           Literal Constants
*============================================================

<literal-const> = %INTEGER ;
<literal-const> = %CHARACTER ;
<literal-const> = %STRING ;
<literal-const> = "true" ;
```

```
<literal-const> = "false" ;

<string> = %STRING ;
```



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399