HAL
open science

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

Fatma Jebali, Frédéric Lang, Radu Mateescu

▶ To cite this version:

Fatma Jebali, Frédéric Lang, Radu Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems (Syntax and Formal Semantics). [Research Report] RR-8527, INRIA. 2014. hal-00983711v3

HAL Id: hal-00983711
https://inria.hal.science/hal-00983711v3

Submitted on 16 Sep 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

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 (Syntax and Formal Semantics)

Fatma Jebali, Frédéric Lang, Radu Mateescu

Project-Team Convecs

Research Report n° 8527 — Avril 2014 — 78 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 the syntax and formal semantics of 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 degré de 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 la syntaxe et la sémantique formelle de 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

Contents

1 Version **6**

2 Introduction **6**

3 Mathematical Notations **8**
3.1 General Notations . . . 8
3.2 BNF Notations . . . 8

4 Lexical Elements **9**
4.1 Boolean literals . . . 9
4.2 Natural number literals . . . 9
4.3 Integer number literals . . . 9
4.4 Character and string literals . . . 9
4.5 Operators . . . 9
4.6 Comments . . . 10
4.7 Identifiers . . . 10
4.8 Reserved words . . . 10

5 Abstract Syntax **11**
5.1 Notational conventions . . . 11
5.2 Program definition . . . 11
5.3 Type definition . . . 12
5.4 Literal constants . . . 12
5.5 Expressions . . . 13
5.6 Predefined functions . . . 13
5.6.1 Type conversion . . . 13
5.6.2 Functions on arrays . . . 13
5.6.3 Functions on records . . . 13
5.7 Statements . . . 13
5.8 Constant definition . . . 14
5.9 Variable declaration . . . 14
5.10 Formal parameters . . . 15
5.11 Blocks . . . 15
5.12 Environments . . . 16
5.13 Mediums . . . 16
5.14 Actors invocation . . . 17
5.15 System definition . . . 17

6 Static Semantics **19**
6.1 Conventions . . . 19
6.2 Identifiers . . . 19
6.3 Types . . . 19
6.3.1 Binding rules . . . 20
6.4 Expressions . . . 20
6.4.1 Binding rules . . . 20
6.4.2 Typing rules . . . 20
6.5 Statements . . . 23
6.5.1 Binding rules . . . 23

6.5.2 Typing rules . . . 24
6.5.3 Initialization rules . . . 24
6.6 Actor allocation and invocation . . . 26
6.6.1 Binding rules . . . 27
6.6.2 Typing rules . . . 28
6.6.3 Initialization rules . . . 28
6.7 Constant . . . 30
6.7.1 Binding rules . . . 30
6.7.2 Typing rules . . . 30
6.8 Block . . . 30
6.8.1 Binding rules . . . 31
6.8.2 Typing rules . . . 32
6.8.3 Initialization rules . . . 32
6.9 Environment . . . 32
6.9.1 Binding rules . . . 32
6.9.2 Initialization rules . . . 33
6.10 Medium . . . 33
6.10.1 Binding rules . . . 33
6.10.2 Initialization rules . . . 34
6.11 System . . . 34
6.11.1 Binding rules . . . 34
6.12 Program . . . 35
6.12.1 Binding rules . . . 35

7 Dynamic Semantics 36
7.1 Notational conventions . . . 36
7.1.1 Stores, stacks, and memories . . . 36
7.1.2 Labelled transition system . . . 37
7.2 Dynamic semantics of expressions . . . 37
7.2.1 Constant . . . 37
7.2.2 Variable . . . 37
7.2.3 Predefined function call . . . 38
7.3 Dynamic semantics of statements . . . 38
7.3.1 Null . . . 38
7.3.2 Sequential composition . . . 38
7.3.3 Assignment . . . 39
7.3.4 Array element assignment . . . 39
7.3.5 While loop . . . 39
7.3.6 For loop . . . 39
7.3.7 Conditional . . . 39
7.3.8 Nondeterministic assignment . . . 40
7.3.9 Nondeterministic choice . . . 40
7.3.10 Case selection . . . 40
7.3.11 Signal . . . 40
7.3.12 Block invocation . . . 41
7.4 Dynamic semantics of systems . . . 44
7.4.1 Block invocation . . . 44
7.4.2 Environment invocation . . . 45
7.4.3 Medium invocation . . . 47

7.4.4 Dynamic semantics of system 49
7.5 Dynamic semantics of programs 52

8 Basic Examples **53**
8.1 Independent blocks with independent environments 53
8.2 Independent blocks with shared environments 53
8.3 Network of blocks communicating via a medium 54

9 Conclusion **56**

A Lexical Structure **59**

B Concrete Grammar **60**

C Operational semantics through examples **66**
C.1 Nested blocks 66
C.1.1 Initial state 66
C.1.2 Construction of the transitions 66
C.1.3 Generated LTS 69
C.2 Strict alternation of blocks 69
C.2.1 Initial state 70
C.2.2 Construction of the first transition 70
C.2.3 Execution of the second transition. 75
C.2.4 Construction of the whole LTS. 77

1 Version

This document is version 3.0 of the definition of the GRL language.

2 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 non-determinism 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 presents the syntax and formal semantics of a new modelling language, named *GRL* (*GALS Representation Language*), whose aim is 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 LNT [8], 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.

3 Mathematical Notations

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

3.1 General Notations

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 of length n
$a_0, \ldots, a_n$	non-empty finite sequence of elements of length n+1
$\{a_1, \ldots, a_n\}$	possibly empty set of elements $a_1, \ldots, a_n$ of size n
$\{a_0, \ldots, a_n\}$	non-empty set of elements $a_0, \ldots, a_n$ of size n+1
$a \in A$	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 set of all ordered pairs (a, b) where $a \in A$ and $b \in B$ (Cartesian product)

3.2 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

4 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.

4.1 Boolean literals

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

$$\textit{bool} \ ::= \ \textbf{true} \mid \textbf{false}$$

4.2 Natural number literals

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

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

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

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

4.3 Integer number literals

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

$$\textit{int} \ ::= \ [\texttt{-}|\texttt{+}]\ \textit{nat}$$

4.4 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].

4.5 Operators

Operators can be classified as follows:

$$\begin{array}{rcl} \textit{unary_operator} & ::= & \texttt{+} \mid \texttt{-} \mid \textbf{not} \mid \textbf{abs} \mid \textbf{nat} \mid \textbf{nat16} \mid \textbf{nat32} \mid \textbf{int} \mid \textbf{int16} \mid \textbf{int32} \\ \textit{binary_operator} & ::= & \textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \textbf{implies} \mid \textbf{equ} \mid \texttt{+} \mid \texttt{-} \mid \texttt{\%} \mid \texttt{\^{}} \mid \texttt{*} \mid \texttt{/} \\ & & \mid \texttt{!=} \mid \texttt{==} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \end{array}$$

4.6 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 between the characters "`(*`" and "`*)`" is ignored.

4.7 Identifiers

An identifier is defined by a letter followed by a possibly empty series of letters, digits and underscores. GRL prohibits that an identifier starts or ends with an underscore.

$$\begin{array}{rcl} \textit{letter} & ::= & \mathbf{a} \mid ... \mid \mathbf{z} \mid \mathbf{A} \mid ... \mid \mathbf{Z} \\ \textit{identifier} & ::= & \textit{letter}\big(_* \ (\textit{digit} \mid \textit{letter})\big)* \end{array}$$

GRL identifiers are case sensitive, so that all occurrences of the same identifier must use exactly the same case, i.e., lower-case and upper-case characters have to be respected. For instance, if a variable has identifier "XyZ", then all its occurrences must have the same identifier "XyZ", but neither "xyz" nor "Xyz". However, to avoid confusion, GRL forbids declaring in the same scope identifiers of the same nature (e.g., variables, constructors, functions, etc.) differring only by their case. Therefore, in the sequel, identifiers are not considered as "*distinct*" if they differ only by their case, e.g., "XyZ" and "XYZ" are not distinct identifiers.

4.8 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**			

5 Abstract Syntax

This section covers the grammatical structure of GRL programs using BNF notation (See Section 3.2).

5.1 Notational conventions

The following table summarizes the generic terminal symbols and the most frequently used non-terminal symbols used to specify the grammar of GRL.

Symbols	
Identifiers (generic 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*

5.2 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 has the same name as 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 (directly or transitively).

$$
\begin{array}{lcl}
\textit{program_definition} & ::= & \textbf{program}\ P\ [\ (P_0\,,\ldots,P_n)\ ]\ \textbf{is} \\
& & \quad (\textit{type_definition} \\
& & \quad \mid \textit{constant_definition} \\
& & \quad \mid \textit{block_definition} \\
& & \quad \mid \textit{environment_definition} \\
& & \quad \mid \textit{medium_definition} \\
& & \quad \mid \textit{system_definition})* \\
& & \textbf{end program}
\end{array}
$$

5.3 Type definition

The following rule defines the types recognized in a GRL specification. Typing rules are discussed in Section 6.3. T denotes a user-defined type identifier.

$$
\begin{array}{rcl}
\mathit{type} & ::= & \textbf{bool} \\
 & | & \textbf{nat} \\
 & | & \textbf{nat16} \\
 & | & \textbf{nat32} \\
 & | & \textbf{int} \\
 & | & \textbf{int16} \\
 & | & \textbf{int32} \\
 & | & \textbf{char} \\
 & | & \textbf{string} \\
 & | & T \\
 & & \\
\mathit{type_definition} & ::= & \textbf{type}\ T\ \textbf{is} \\
 & & \quad \mathit{type_expression} \\
 & & \textbf{end type} \\
 & & \\
\mathit{type_expression} & ::= & \textbf{array}\ [m\,..\,n]\ \textbf{of}\ \mathit{type} \\
 & | & \textbf{range}\ m\ ..\ n\ \textbf{of}\ \mathit{type} \\
 & | & \textbf{record}\ f_0 : \mathit{type}_0\ ,\ \ldots\ ,\ f_n : \mathit{type}_n \\
 & | & \textbf{enum}\ C_0\ ,\ \ldots\ ,\ C_n
\end{array}
$$

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 ranging from m to n, which must be literal naturals.

- 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 numbers ranging from m to n, which must be literal integers of type *type*, which itself must be one of **nat**, **nat16**, **nat32**, **int**, **int16**, or **int32**.

- The enumerated type, defined using the keyword **enum**, denotes a finite and ordered set of symbolic values (identifiers) $C_0, \ldots, C_n$.

5.4 Literal constants

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

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

5.5 Expressions

The syntax of expressions is given by the following grammar.

E	::=	X	variable
	|	(E_0)	record field access
	|	$E_0 . f$	parenthesised expression
	|	$E_1 [E_0]$	record field access
	|	*unary_operator* E_0	unary operation
	|	E_1 *binary_operator* E_2	binary operation
	|	K [**of** *type*]	literal constant
	|	$F\ (E_0 , \ldots , E_n)$	predefined function

5.6 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 4.5.

5.6.1 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. An exception can be raised if the value does not belong to a type of E.

5.6.2 Functions on arrays

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

5.6.3 Functions on records

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

5.7 Statements

Below is the list of statements that can be used in GRL programs.

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**	case 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

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

5.9 Variable declaration

Variables in GRL are declared as follows:

decl_list ::= *var_decl*$_0$, ... , *var_decl*$_n$
var_decl ::= X_0 , ... , X_m : *type* [:= E_0]
decl_list_non_init ::= *var_decl_non_init*$_0$, ... , *var_decl_non_init*$_n$
var_decl_non_init ::= X_0 , ... , X_m : *type*

Local variables are formally defined below. Permanent variables, defined after the keyword

perm, have a lifetime extending across the entire execution of the program, whereas variables defined after the keyword **temp** are temporary.

$$\begin{array}{rcl} \textit{local_variables} & ::= & \textbf{perm}\ \textit{decl_list} \\ & | & \textbf{temp}\ \textit{decl_list} \end{array}$$

5.10 Formal parameters

Formal parameters are classified as follows:

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

$$\textit{const_param} \quad ::= \quad \textbf{const}\ \textit{decl_list}$$

- Input/Output parameters are defined as a set of variable declarations preceded by the keywords **in** or **out**.

$$\begin{array}{rcl} \textit{inout_param} & ::= & \textbf{in}\ \textit{decl_list} \\ & | & \textbf{out}\ \textit{decl_list_non_init} \\ \textit{inout_param_non_init} & ::= & \textbf{in}\ \textit{decl_list_non_init} \\ & | & \textbf{out}\ \textit{decl_list_non_init} \end{array}$$

- Communication parameters are defined as a set of variable declarations preceded by the keywords **receive** or **send**.

$$\begin{array}{rcl} \textit{com_param} & ::= & \textbf{receive}\ \textit{decl_list_non_init} \\ & | & \textbf{send}\ \textit{decl_list_non_init} \end{array}$$

5.11 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.

$$\begin{array}{rcl} \textit{block_definition} & ::= & \textbf{block}\ B\ [\ [\textit{const_param}]\ ] \\ & & \quad [\ (\textit{inout_param}_0\ ;\ \ldots\ ;\ \textit{inout_param}_m)\ ] \\ & & \quad [\ \{\textit{com_param}_0\ ;\ \ldots\ ;\ \textit{com_param}_n\}\ ]\ \textbf{is} \\ & & [\ \textbf{allocate}\ B_0[[\textit{arg}_{(0,0)}\ ,\ \ldots\ ,\ \textit{arg}_{(0,p_0)}]]\ \textbf{as}\ Bi_0\ , \\ & & \quad \ldots\ , \\ & & \quad B_q[[\textit{arg}_{(q,0)}\ ,\ \ldots\ ,\ \textit{arg}_{(q,p_q)}]]\ \textbf{as}\ Bi_q\ ] \\ & & [\ \textit{local_variables}_0\ ,\ \ldots\ ,\textit{local_variables}_l\ ] \\ & & \quad I_0 \\ & & \textbf{end block} \\ & | & \textbf{block}\ B\ [\ [\textit{const_param}]\ ] \\ & & \quad [\ (\textit{inout_param}_0\ ;\ \ldots\ ;\ \textit{inout_param}_m)\ ]\ \textbf{is} \\ & & \quad \textbf{!c}\ \textit{string}\ \ |\ \ \textbf{!lnt}\ \textit{string} \\ & & \textbf{end block} \end{array}$$

A block definition can be either user-defined or 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.

Blocks are allocated using clauses of the form "**allocate** B_i [$arg_{(i,1)}$, . . . , $arg_{(i,p_i)}$] **as** Bi_i", each of which creates an instance Bi_i of block B_i that has as const parameters the list $arg_{(i,1)}$, . . . , $arg_{(i,p_i)}$.

5.12 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. 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_param*]]
[(*inout_param_non_init*$_0$ | . . . | *inout_param_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*$_l$]
$\overline{I_0}$
end environment

Environment N interacts with blocks by connecting its input (resp., output) parameters to output (resp., input) parameters of blocks. Its body should define (or not) the behaviour of the environment when an interaction via a formal parameter list occurs. A signal of the form "**on** X_0 , . . . , X_n -> I" defines the behaviour of N that corresponds to the parameter list "**in** $X_0 : T_0$, . . . , $X_n : T_n$" whereas a signal of the form "**on** ?Y_0 , . . . , ?Y_m -> I" defines the behaviour of N that corresponds to the parameter list "**out** $Y_0 : T_0$, . . . , $Y_m : T_m$".

5.13 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.).

medium_definition ::= **medium** M [[*const_param*]]
[{*com_param*$_0$ | . . . | *com_param*$_m$}] **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*$_l$]
$\overline{I_0}$
end medium

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.

5.14 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_param*, *inout_param_non_init*, *com_param*, or *const_param*.
- An *actual channel* denotes a set of actual parameters of the form "arg_0 , ... , 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 6.6.1). 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*".

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

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

$$\begin{array}{rcl} allocation & ::= & B[[arg_0\,,\ldots,arg_n]]\ \textbf{as}\ Bi \\ & \mid & N[[arg_0\,,\ldots,arg_n]]\ \textbf{as}\ Ni \\ & \mid & M[[arg_0\,,\ldots,arg_n]]\ \textbf{as}\ Mi \end{array}$$

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

$$\begin{array}{rcl} block_instance & ::= & Bi\ (arg_{(0,0)}\,,\ldots,arg_{(0,m_0)}\,,\ldots,arg_{(n,0)}\,,\ldots,arg_{(n,m_n)}) \\ block_instance & ::= & Bi\ [(arg_{(0,0)}\,,\ldots,arg_{(0,m_0)}\,;\ldots;arg_{(n,0)}\,,\ldots,arg_{(n,m_n)})] \\ & & [\{arg'_{(0,0)}\,,\ldots,arg'_{(0,p_0)}\,;\ldots;arg'_{(q,0)}\,,\ldots,arg'_{(q,p_q)}\}] \\ environment_instance & ::= & Ni\ (arg_{(0,0)}\,,\ldots,arg_{(0,m_0)} \mid \ldots \mid arg_{(n,0)}\,,\ldots,arg_{(n,m_n)}) \\ medium_instance & ::= & Mi\ \{arg_{(0,0)}\,,\ldots,arg_{(0,m_0)} \mid \ldots \mid arg_{(n,0)}\,,\ldots,arg_{(n,m_n)}\} \end{array}$$

5.15 System definition

Actors can be composed to form a higher level construct, called system. A system, formally defined below, 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.

$$
\begin{array}{lll}
\textit{system_definition} & ::= & \textbf{system}\ S\ [\ (\textit{decl_list_non_init}_0)\ ]\ \textbf{is} \\
& & \quad \textbf{allocate}\ \textit{allocation}_0\ ,\ \ldots\ ,\ \textit{allocation}_n \\
& & \quad [\ \textbf{temp}\ \textit{decl_list_non_init}_1\ ] \\
& & \quad\ \textbf{network} \\
& & \quad\quad \textit{block_instance}_0\ , \\
& & \quad\quad \ldots\ , \\
& & \quad\quad \textit{block_instance}_p \\
& & \quad [\ \textbf{constrainedby} \\
& & \quad\quad \textit{environment_instance}_0\ , \\
& & \quad\quad \ldots\ , \\
& & \quad\quad \textit{environment_instance}_q\ ] \\
& & \quad [\ \textbf{connectedby} \\
& & \quad\quad \textit{medium_instance}_0\ , \\
& & \quad\quad \ldots\ , \\
& & \quad\quad \textit{medium_instance}_r\ ] \\
& & \textbf{end system}
\end{array}
$$

The list $\textit{decl_list_non_init}_0$ defines the parameters that are visible from the external world whereas the list $\textit{decl_list_non_init}_1$ defines the invisible parameters that are invisible from the external worlds.

6 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 semantic rules: binding rules, typing rules, and initialization rules.

6.1 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:

$$\begin{array}{rcl} vars(X) & = & \{X\} \\ vars(E\,.\,f) & = & vars(E) \\ vars(\texttt{(}E\texttt{)}) & = & vars(E) \\ vars(F\texttt{(}E_0\texttt{,}\ldots\texttt{,}E_n\texttt{)}) & = & vars(E_0) \cup \ldots \cup vars(E_n) \\ vars(E_0\texttt{[}E_1\texttt{]}) & = & vars(E_0) \cup vars(E_1) \\ vars(unary_operator\ E_0) & = & vars(E_0) \\ vars(E_0\ binary_operator\ E_1) & = & vars(E_0) \cup vars(E_1) \\ vars(K) & = & \emptyset \end{array}$$

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.

6.2 Identifiers

(ID1) GRL reserved words must not be used as identifiers (see Section 4.8).

6.3 Types

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

We define recursively the relation "T depends on T'" where T and T' are existing types as follows. T depends on T' if both of the following conditions are satisfied:

- T has one of the following forms: "**range** m .. n **of** T''", "**array** [m .. n] **of** T''", "**record** ... ,f : T'' , ..."
- $T' = T''$ or T'' depends on T'

6.3.1 Binding rules

(TB1) Cicular dependencies in type definitions are forbidden. If the definition of a type T depends on type T', then:

- T' must be different from T
- the definition of T' must not depend on T, transitively

- **Enumerated type** "**type** T **is enum** C_0 **,** ... **,** C_n **end type**"

  (TB2) Identifiers C_0, ..., C_n must be pairwise distinct.

- **Record type** "**type** T **is record** f_0 **:** $type_0$ **,** ... **,** f_n **:** $type_n$ **end type**"

  (TB3) Fields f_0, ..., f_n must be pairwise distinct.

  (TB4) Types $type_0, \ldots, type_n$ must be existing 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 $m \leq n$.

6.4 Expressions

6.4.1 Binding rules

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

6.4.2 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 **,** ... **,** 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 $F(E_0, \ldots, E_n)$ is the same type as the return type in the definition of F.

- **Array element access** "$E_0[E_1]$"

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

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

  (ET12) The type of $E_0[E_1]$ is *type*.

- **Record field access** "$E.f$"

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

  (ET14) The type of $E.f$ 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. Note that E_0 and E_1 must have the same type (otherwise, explicit type conversion using the type conversion operations defined above 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
`+`, `-`, `*`, `/`, `^`, `%`	**nat**	**nat**
	nat16	**nat16**
	nat32	**nat32**
`==`, `!=`, `<`, `>`, `<=`, `>=`	**nat**	**bool**
	nat16	**bool**
	nat32	**bool**
Operators on integers		

Binary operator	Operands type	Return type
+, -, *, /, ^	**int** **int16** **int32**	**int** **int16** **int32**
==, !=, <, >, <=, >=	**int** **int16** **int32**	**bool** **bool** **bool**
Operators on characters and strings		
Binary operator	Operands type	Return type
==, !=, <, >, <=, >=	**char**	**bool**
==, !=	**string**	**bool**

6.5 Statements

6.5.1 Binding rules

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

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

  (IB2) X must not be a constant variable.

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

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

- **Case 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, $X_0, \ldots, X_n$ must have been defined as input parameters.

    (IB8.2) Inside a medium definition, $X_0, \ldots, X_n$ must have been defined as receive parameters.

  – "**on** ?X_0 **,** ... **,** ?X_n -> I_0". In this case:

(IB8.3) Inside an environment definition, $X_0, \ldots, X_n$ must have been defined as output parameters.

(IB8.4) Inside a medium definition, $X_0, \ldots, X_n$ must have been defined as send parameters.

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

6.5.2 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) 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) 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 $E_0, \ldots, E_n$ must have the type *bool*.

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

  (IT11) Expression E must have the type *bool*.

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

  (IT12) Expression E must have the type *bool*.

6.5.3 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. In general, statement I is correctly initialized if under the assumption that all variables in S are initialized, it is possible to build S' such that $S \triangleright I \triangleright S'$.

- **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**

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

- **Case selection**

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

- **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**

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

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

- **Block invocation**
  See Section 6.6.

6.6 Actor allocation and invocation

Actor allocations can have the following forms:

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

Actor invocations can have the following forms:

- Block invocations inside actors have the form

$$Bi\texttt{(}arg_{(0,0)}\,,\ldots,arg_{(0,n_0)}\,,\ldots,arg_{(m,0)}\,,\ldots,arg_{(m,n_m)}\texttt{)},$$

  i.e., argument lists are separated by commas.

- Block invocations inside systems have either form

$$Bi\texttt{(}arg_{(0,0)}\,,\ldots,arg_{(0,n_0)}\texttt{;}\ldots\texttt{;}arg_{(m,0)}\,,\ldots,arg_{(m,n_m)}\texttt{)},$$

$$Bi\texttt{\{}arg_{(0,0)}\,,\ldots,arg_{(0,n_0)}\texttt{;}\ldots\texttt{;}arg_{(m,0)}\,,\ldots,arg_{(m,n_m)}\texttt{\}},$$

  or

$$\begin{array}{l} Bi\texttt{(}arg_{(0,0)}\,,\ldots,arg_{(0,n_0)}\texttt{;}\ldots\texttt{;}arg_{(m,0)}\,,\ldots,arg_{(m,n_m)}\texttt{)} \\ \quad\texttt{\{}arg'_{(0,0)}\,,\ldots,arg'_{(0,p_0)}\texttt{;}\ldots\texttt{;}arg'_{(q,0)}\,,\ldots,arg'_{(q,p_q)}\texttt{\}}, \end{array}$$

  i.e., argument lists are separated by semicolons.

- Environment invocations (necessarily inside systems) have the form

$$Ni\texttt{(}arg_{(0,0)}\,,\ldots,arg_{(0,n_0)}\texttt{|}\ldots\texttt{|}arg_{(m,0)}\,,\ldots,arg_{(m,n_m)}\texttt{)},$$

  i.e., argument lists are separated by pipes.

- Medium invocations (necessarily inside systems) have the form

$$Mi\texttt{\{}arg_{(0,0)}\,,\ldots,arg_{(0,n_0)}\texttt{|}\ldots\texttt{|}arg_{(m,0)}\,,\ldots,arg_{(m,n_m)}\texttt{\}},$$

  i.e., argument lists are separated by pipes.

6.6.1 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 **,** ... **,** X_n"
- "**any** T_0 **,** ... **,any** T_n"
- "_ **,** ... **,** _"

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 **,** ... **,** 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 **,** ... **,** ?X_n"
- "?_ **,** ... **,** ?_"

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 an environment (resp., medium) instance can have one of the following forms:

- "X_0 **,** ... **,** X_n"
- "_ **,** ... **,** _"

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

- "?X_0 , . . . , ?X_n"
- "?_ , . . . , ?_"

(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.

6.6.2 Typing rules

(ACT1) 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.

(ACT2) 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.

(ACT3) 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.

(ACT4) 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.

(ACT5) 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.

6.6.3 Initialization rules

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

(ACI1) **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.		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 (in this case, E is a variable).	
	?_			X must be assigned a value in the body I_0 of the callee.		X must be assigned a value in the body I_0 of the callee.
	? Y			X must be assigned a value in the body I_0 of the callee. Y is assigned the value of X.		X must be assigned a value in the body I_0 of the callee. Y is assigned the value of X.
	any T		X is assigned an arbitrary value of type T.			

(ACI2) **Environment invocation**

		class of formal parameter X		
		const	**in**	**out**
actual parameter	_	X must have a default value at definition time. Then, X is assigned its default value.	Statements of the form "**on** X_0**,** ...**,**X_n-> I_0" (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**,** ...**,?**X_n-> I_0" (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**,** ...**,?**X_n-> I_0" (if any) such that $X \in \{X_0, \ldots, X_n\}$.
	any T			

(ACI3) **Medium invocation**

		class of formal parameter X		
		const	**receive**	**send**
actual parameter	_	X must have a default value at definition time. Then, X is assigned its default value.	Statements of the form "**on** X_0**,** ...**,**X_n-> I_0" (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**,** ...**,?**X_n-> I_0" (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**,** ...**,?**X_n-> I_0" (if any) such that $X \in \{X_0, \ldots, X_n\}$.
	any T			

6.7 Constant

A constant is defined as follows: "**constant** X **:** *type* **is** E **end constant**". We define recursively the relation "X depends on Y" where Y is a constant as follows. X depends on Y if both of the following conditions are satisfied:

- $Y \in vars(E)$
- $Z \in vars(E)$ and Z depends on Y

6.7.1 Binding rules

(CB1) Circular dependencies in constant definitions are forbidden. If the definition of a constant X depends on a constant Y, then:

- Y must be different from X
- the definition of Y must not depend on X, transitively.

(CB1) E must be a constant expression.

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

6.7.2 Typing rules

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

6.8 Block

A block is defined as follows:

$$
\begin{array}{lll}
\textit{block_definition} & ::= & \textbf{block}\ B\ [\ [\textit{const_param}]\ ] \\
& & \quad [\ (\textit{inout_param}_0\ ;\ \ldots\ ;\ \textit{inout_param}_m)\ ] \\
& & \quad [\ \{\textit{com_param}_0\ ;\ \ldots\ ;\ \textit{com_param}_n\}\ ]\ \textbf{is} \\
& & [\ \textbf{allocate}\ B_0[[\textit{arg}_{(0,0)}\ ,\ \ldots\ ,\textit{arg}_{(0,p_0)}]]\ \textbf{as}\ Bi_0\ , \\
& & \quad \ldots\ , \\
& & \quad B_q[[\textit{arg}_{(q,0)}\ ,\ \ldots\ ,\textit{arg}_{(q,p_q)}]]\ \textbf{as}\ Bi_q\ ] \\
& & [\ \textit{local_variables}_0\ ,\ \ldots\ ,\textit{local_variables}_l\ ] \\
& & \quad I_0 \\
& & \textbf{end block} \\
& | & \textbf{block}\ B\ [\ [\textit{const_param}]\ ] \\
& & \quad [\ (\textit{inout_param}_0\ ;\ \ldots\ ;\textit{inout_param}_m)\ ]\ \textbf{is} \\
& & \quad \texttt{!c}\ \textit{string}\ \mid\ \texttt{!}\textbf{lnt}\ \textit{string} \\
& & \textbf{end block}
\end{array}
$$

6.8.1 Binding rules

(BB1) Block identifiers $B_0, \ldots, B_q$ must be defined in the current scope.

(BB2) Instance identifiers $Bi_0, \ldots, Bi_q$ must be pairwise distinct and different from B.

(BB3) Blocks $B_0, \ldots, B_q$ must be different from B and must not invoke B either directly or transitively. This avoids circular dependencies between blocks.

(BB4) All variables declared in *const_param*, $\textit{inout_param}_i$ ($i \in 0..m$), $\textit{com_param}_i$ ($i \in 0..n$) , and $\textit{local_variables}_i$ ($i \in 0..l$) must be pairwise distinct.

(BB5) The scope of formal parameters declared in *const_param* is $\textit{inout_param}_i$ ($i \in 0..m$), $\textit{com_param}_i$ ($i \in 0..n$) and the text delimited by the keyword "**is**" and the keywords "**end block**".

(BB6) The scope of formal parameters declared in $\textit{inout_param}_i$ ($i \in 0..m$), is $\textit{com_param}_i$ ($i \in 0..n$) and the text delimited by the keyword "**is**" and the keywords "**end block**".

(BB7) The scope of formal parameters declared in $\textit{com_param}_i$ ($i \in 0..n$) is the text delimited by the keyword "**is**" and the keywords "**end block**".

(BB8) The scope of variables declared in $\textit{local_variables}_i$ ($i \in 0..l$) is the body I_0.

(BB9) A variable declared in *const_param* must not be assigned a value in the body I_0.

(BB10) 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".

(BB11) When using `!c` pragma, *string* must be a valid C identifier.

(BB12) When using `!lnt` pragma, *string* must be a valid LNT identifier.

6.8.2 Typing rules

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

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

6.8.3 Initialization rules

(BI1) See Section 6.6 for initialization rules concerning formal parameters.

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

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

6.9 Environment

An environment specification has the following form:

environment_definition ::= **environment** N [[*const_param*]]
[(*inout_param_non_init*$_0$ | ... | *inout_param_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

6.9.1 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 lists *const_param*, *inout_param_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 lists *const_param* is *inout_param_non_init*$_i$ ($i \in 0..m$) and the text delimited by the keyword "**is**" and the keywords "**end environment**".

(NB5) The scope of formal parameters declared in *inout_param_non_init*$_i$ ($i \in 0..m$) is the text delimited by the keyword "**is**" and the keywords "**end environment**".

(NB6) The scope of variables declared in *local_variables*$_i$ ($i \in 0..l$) is the body I_0.

(NB7) A variable declared in *const_param* must not be assigned a value in the body I_0.

(NB8) For each signal statement of the form "**on** X_0 , ... , X_n-> I_1" inside the body I_0, a list of the form "**in** $X_0 : T_0$, ... , $X_n : T_n$" should have already been declared among *inout_param_non_init*$_i$ ($i \in 0..m$).

(NB9) For each signal statement of the form "**on** ?X_0 **,** ... **,**?X_n-> I_1" inside the body I_0, a list of the form "**out** X_0 **:** T_0 **,** ... **,** X_n **:** T_n" should have already been declared among $inout_param_non_init_i$ ($i \in 0..m$).

6.9.2 Initialization rules

(NI1) See Section 6.6 for initialization rules concerning formal parameters.

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

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

6.10 Medium

A medium specification has the following grammar:

```
medium_definition ::= medium M [ [const_param] ]
                       [ {com_param_0 | ... | com_param_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 medium
```

6.10.1 Binding rules

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

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

(MB3) Variables declared in $const_param$, com_param_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 $const_param$ is com_param_i ($i \in 0..m$) and the text delimited by the keyword "**is**" and the keywords "**end medium**".

(MB5) The scope of formal parameters declared com_param_i ($i \in 0..m$) is the text delimited by the keyword "**is**" and the keywords "**end medium**".

(MB6) The scope of variables declared in $local_variables_i$ ($i \in 0..l$) is I_0.

(MB7) A variable declared in $const_param$ must not be assigned a value in the body I_0.

(MB8) For each signal statement of the form "**on** X_0 **,** ... **,** X_n-> I_1" inside the body I_0, a list of the form "**receive** X_0 **:** T_0 **,** ... **,** X_n **:** T_n" should have already been declared among com_param_i ($i \in 0..m$).

(MB9) For each signal statement of the form "**on** ?X_0 **,** ... **,**?X_n-> I_1" inside the body I_0, a list of the form "**send** X_0 **:** T_0 **,** ... **,** X_n **:** T_n" should have already been declared among com_param_i ($i \in 0..m$).

6.10.2 Initialization rules

(MI1) See Section 6.6 for initialization rules concerning formal parameters.

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

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

6.11 System

A system specification is given by the following grammar.

```
system_definition ::= system S [ (decl_list_non_init0) ] is
                        allocate allocation0 , ... , allocationn
                        [ temp decl_list_non_init1 ]
                          network
                            block_instance0 ,
                            ... ,
                            block_instancep
                        [   constrainedby
                            environment_instance0 ,
                            ... ,
                            environment_instanceq ]
                        [   connectedby
                            medium_instance0 ,
                            ... ,
                            medium_instancer ]
                      end system
```

6.11.1 Binding rules

(SB1) The scope of formal parameters declared in "$decl_list_non_init_0$" 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 "$decl_list_non_init_1$" is the text delimited by the keyword "**network**" and the keywords "**end system**".

(SB5) An actual parameter 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 channel of a block can be connected to an output (resp., input) actual channel of an environment.

(SB7) At most one receive (resp., send) actual channel of a block can be connected to a send (resp., receive) actual channel of a 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 6.6 for more static semantic rules concerning actors invocation.

6.12 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

6.12.1 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*"). Letter case is not significant here.

(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.

7 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. A GRL program is formally defined in terms of an LTS (*Labelled Transition System*).

7.1 Notational conventions

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

7.1.1 Stores, stacks, and memories

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 $X_1, \ldots, X_n$ and a set of values $e_1, \ldots, e_n$, the store ρ that maps X_i to e_i ($\forall i \in 1..n$) is written as $[X_1 \leftarrow e_1, \ldots, X_n \leftarrow e_n]$. 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 sum of two stores ρ_1 and ρ_2, denoted "$\rho_1 \oplus \rho_2$", represents the update of ρ_1 with respect to 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 defined recursively by either the empty sequence ϵ or a non empty sequence of the form "$\sigma'.Ai$" where σ' is a sequence and Ai is an actor instance identifier. The symbol σ will represent the stack of actor instances called up to 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. 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 sum of two memories μ_1 and μ_2, denoted "$\mu_1 \oplus \mu_2$", represents the update of μ_1 with respect to 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 $\mu_1 \oplus \ldots \oplus \mu_n$.

7.1.2 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 $(s, a, s') \in \rightarrow$.

7.2 Dynamic semantics of expressions

The evaluation of expressions is defined by triples of the form "$\{E\}\ \rho\ \rightarrow_{\mathsf{e}}\ e$" 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.

7.2.1 Constant

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

$$\overline{\{K\}\ \rho \rightarrow_{\mathsf{e}} K}$$

7.2.2 Variable

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

$$\overline{\{X\}\ \rho \rightarrow_{\mathsf{e}} \rho(X)}$$

7.2.3 Predefined function call

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

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

Record field access and array element access are considered as predefined functions.

7.3 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 and executed 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 and executed 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'.

We assume that after binding analysis, each variable has been assigned a distinct name, thus preventing variable shadowing (i.e., a variable declared within a certain scope has the same name as a variable declared in an outer scope) to occur.

7.3.1 Null

The null statement terminates normally without updating the store.

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

7.3.2 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.

$$\frac{\begin{array}{c}\{I_1\}\ \sigma, \rho, \mu \xrightarrow{\ell_1}_{\mathsf{i}} \rho', \mu' \\ \{I_2\}\ \sigma, \rho', \mu' \xrightarrow{\ell_2}_{\mathsf{i}} \rho'', \mu''\end{array}}{\{I_1\,; I_2\}\ \sigma, \rho, \mu \xrightarrow{\ell_1+\ell_2}_{\mathsf{i}} \rho'', \mu''}$$

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 6.5.1).

7.3.3 Assignment

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

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

7.3.4 Array element assignment

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

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

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.

7.3.5 While loop

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

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

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

7.3.6 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:

$$\begin{array}{l} I_0\,; \\ \mathbf{while}\ E\ \mathbf{loop} \\ \quad I_2\,; I_1 \\ \mathbf{end\ loop} \end{array}$$

7.3.7 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:

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

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

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

7.3.8 Nondeterministic assignment

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

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

7.3.9 Nondeterministic choice

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

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

7.3.10 Case selection

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

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

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

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 6.5.1). 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**

7.3.11 Signal

A signal statement has one of the following forms:

on X_0 **,** ... **,** X_n-> I_0
on ?X_0 **,** ... **,**?X_n-> I_0

A signal statement terminates normally by executing the statement I_0 in the current store and producing a label. Note that static semantics ensure that nested signal statements are forbidden (*i.e.*, I_0 must not contain a signal statement, see rule IB5 in Section 6.5.1).

$$\frac{\{X_0,\ldots,X_n\} \subseteq dom(\rho) \quad \{I_0\}\ \sigma,\rho,\mu \xrightarrow{\epsilon}_{\mathsf{i}}\ \rho',\mu'}{\{\textbf{on}\ X_0\,,\ldots,X_n\text{->}\ I_0\}\ \sigma,\rho,\mu \xrightarrow{X_0,\ldots,X_n}_{\mathsf{i}}\ \rho',\mu'}$$

$$\frac{\{I_0\}\ \sigma,\rho,\mu \xrightarrow{\epsilon}_{\mathsf{i}}\ \rho',\mu'}{\{\textbf{on}\ ?X_0\,,\ldots,?X_n\text{->}\ I_0\}\ \sigma,\rho,\mu \xrightarrow{?X_0,\ldots,?X_n}_{\mathsf{i}}\ \rho',\mu'}$$

7.3.12 Block invocation

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

- *a* stands for *arg*.
- *vd* stands for either *var_decl* or *var_decl_non_init*.
- *dl* stands for either *decl_list* or *decl_list_non_init*.
- *al* stands for either "$a_0 \ldots a_n$".
- *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: the function *init* maps each variable in a declaration list to its respective initialization value, if any.

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

- Variable list: the function *vars* returns the ordered list of variable identifiers in either a variable declaration list, a formal parameter declaration, or an actual parameter list. The symbol ϵ denotes the empty list. Operator ++ denotes list concatenation.

$$\begin{array}{rcl}
\text{vars}(vd_0,\ldots,vd_n) & = & \text{vars}(vd_0)\texttt{++}\ldots\texttt{++}\text{vars}(vd_n)\\
\text{vars}(X_0\,,\ldots,X_m\!:\!type) & = & \langle X_0,\ldots,X_m\rangle\\
\text{vars}(X_0\,,\ldots,X_m\!:\!type\ \texttt{:=}\ E) & = & \langle X_0,\ldots,X_m\rangle\\
\text{vars}(X_0\,,\ldots,X_m) & = & \langle X_0,\ldots,X_m\rangle\\
\text{vars}(?X_0\,,\ldots,?X_m) & = & \langle X_0,\ldots,X_m\rangle\\
\text{vars}(_\,,\ldots,_) & = & \epsilon\\
\text{vars}(?_\,,\ldots,?_) & = & \epsilon
\end{array}$$

- Type list: the function *types* returns the ordered list of types in a variable declaration list.

$$\begin{array}{rcl}
\text{types}(vd_0,\ldots,vd_n) & = & \text{types}(vd_0)\texttt{++}\ldots\texttt{++}\text{types}(vd_n)\\
\text{types}(X_0\,,\ldots,X_m\!:\!type) & = & type^{m+1}\\
\text{types}(X_0\,,\ldots,X_m\!:\!type\ \texttt{:=}\ E) & = & type^{m+1}
\end{array}$$

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

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

– Arbitrary assignment: the function *any* assigns arbitrary values to parameters.

$$\text{any}(\langle X_0,\ldots,X_n\rangle,\langle T_0,\ldots,T_n\rangle,\rho) \quad = \quad \{[X_0\leftarrow e_0,...,X_n\leftarrow e_n]\mid(\forall i\in 0..n)\ e_i\in T_i\}$$

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

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

– Label printing: the function *label* defines the label that will be visible on the generated LTS and that corresponds to the execution of one block together with its connected actors. Parameters defined in the system profile will be visible on the label whereas other variables will be hidden (i.e., replaced by underscores). Labels are written in *teletypefont* in the sequel.

$$\begin{array}{rcl}
\text{label}(Bi(al_0,\ldots,al_m)\{al'_0,\ldots,al'_n\},\ \rho) & = & \mathtt{Bi}\ \ (\text{label}(\langle al_0,\ldots,al_m\rangle,\rho))\\
 & & \quad\{\text{label}(\langle al'_0,\ldots,al'_n\rangle,\rho)\}\\
\text{label}(\langle al_0,\ldots,al_m\rangle,\rho) & = & \text{label}(al_0,\rho)\mathtt{;}\ldots\mathtt{;}\text{label}(al_m,\rho)\\
\text{label}(\langle a_0,\ldots,a_m\rangle,\rho) & = & \text{label}(a_0,\rho)\mathtt{,}\ldots\mathtt{,}\text{label}(a_m,\rho)\\
\text{label}(X,\rho) & = & \begin{cases}\rho(X) & \text{if } X\in\text{vars}(dl_{sys})\\ _ & \text{otherwise}\end{cases}\\
\text{label}(\textbf{any}\ T,\rho) & = & _\\
\text{label}(_,\rho) & = & _\\
\text{label}(?X,\rho) & = & \begin{cases}?\rho(X) & \text{if } X\in\text{vars}(dl_{sys})\\ ?_ & \text{otherwise}\end{cases}\\
\text{label}(?_,\rho) & = & ?_
\end{array}$$

– Store restriction: the operator $\centerdot$ allows to return subsets of stores.

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

Construction of the store ρ_{global}. After binding analysis, global constants are assumed to be ordered as follows. If $X_1,\ldots,X_n$ is the set of global constants defined respectively with expressions $E_1,\ldots,E_n$, then for each i in $1..(n-1)$ and for each j in $(i+1)..n$ we must have $vars(E_j)\subseteq\{X_1,\ldots,X_i\}$ (i.e., X_j must not belong to $vars(E_i)$).

The store ρ_{global} is constructed by assigning each global constant X_i the evaluation of its expression E_i in the store ρ_{i-1}, ρ_0 being the empty store.

$$\begin{array}{rcl} \rho_0 & = & [] \\ \rho_i & \in & \text{assign}(E_i, X_i, \rho_{i-1}) \quad \forall i \in 1..n \\ \rho_{global} & = & \bigoplus_{i \in 1..n} \rho_i \end{array}$$

Block semantics. The behaviour of a block is the following. The block consumes its input parameters, computes in zero time, then produces its output parameters. These 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 values 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. More specifically, we assume that blocks invoked inside an actor are defined as follows.

block B [**const** dl_{const}] (**in** dl_{in_0} ; ... ;**in** dl_{in_m} ;**out** dl_{out_0} ; ... ;**out** dl_{out_n}) **is**
  allocate ...
  perm dl_{perm} ,
  temp dl_{temp}
    I_0
end block

Block B is allocated as follows: B[al_{const}] **as** Bi (where al_{const} is the actual parameter list that corresponds to the declaration list dl_{const}), and Bi can then be invoked as follows:

$$Bi(al_{in_0}\,;\ldots;al_{in_m}\,;al_{out_0}\,;\ldots;al_{out_n})$$

. The semantic rule of block invocation is the following:

$$\frac{\{I_0\}\ \sigma.Bi, \rho_{var}, \mu \xrightarrow{\epsilon}_{\mathrm{i}}\ \rho_{body}, \mu_{body}}{\{Bi(al_{in_0}\,;\ldots;al_{in_m}\,;al_{out_0}\,;\ldots;al_{out_n}\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathrm{i}}\ \rho',\ \mu'}$$

where stores ρ_{var} and ρ' and memory μ' are constructed as follows.

In a first step, the block invocation starts by constructing the store ρ_{var}, in which Bi should compute its body I_0. The store ρ_{var} is built by composing intermediate stores ρ_{const}, ρ_{input}, ρ_{perm}, and ρ_{temp} as follows:

1. Store ρ_{const} starts by assigning default values to initialized constant formal parameters using function *init*. Constant parameters depend only on global constants, whose values are available in the store ρ_{global}. Then, the store ρ_{const} updates the formal parameters with respective values of actual parameters, using function *assign* in the store ρ of the caller.

$$\rho_{const} \quad \in \quad \text{init}\ (dl_{const},\ \rho_{global}) \oplus \text{assign}\ (al_{const},\ \text{vars}(dl_{const}), \rho)$$

2. Store ρ_{input} starts by assigning default values to initialized input formal parameters using function *init*. Input parameters depend on global constants and constant parameters of the current block. The values of those parameters are available in the store $\rho_{global} \oplus \rho_{const}$.

Then, the store ρ_{input} updates the formal parameters with respective values of actual parameters, using function *assign* in the store ρ of the caller.

$$\rho_{input} \in \bigoplus_{i \in 0..m} \text{init } (dl_{in_i},\ \rho_{global} \oplus \rho_{const}) \oplus \bigoplus_{i \in 0..m} \text{assign } (al_{in_i},\ \text{vars}(dl_{in_i}),\ \rho)$$

3. Store ρ_{perm} assigns to permanent variables their respective values stored in the memory μ, except in the first execution cycle of Bi where permanent variables are assigned their default values using function *init*. Permanent variables depend on both global constants and constant parameters of the current block. The values of those parameters are available in the store $\rho_{global} \oplus \rho_{const}$.

$$\rho_{perm} = \begin{cases} \mu(\sigma.Bi) & \text{if } \sigma.Bi \in dom(\mu) \\ \text{init } (dl_{perm},\ \rho_{global} \oplus \rho_{const}) & \text{otherwise} \end{cases}$$

4. Store ρ_{temp} assigns default values to temporary variables. Temporary variables depend on global constants, constant parameters, and input parameters of the current block. The values of those parameters are available in the store $\rho_{global} \oplus \rho_{const} \oplus \rho_{input}$.

$$\rho_{temp} = \text{init } (dl_{temp},\ \rho_{global} \oplus \rho_{const} \oplus \rho_{input})$$

5. Store ρ_{var} is obtained by the sum of stores ρ_{global}, ρ_{const}, ρ_{input}, ρ_{perm}, and ρ_{temp}.

$$\rho_{var} = \rho_{global} \oplus \rho_{const} \oplus \rho_{input} \oplus \rho_{perm} \oplus \rho_{temp}$$

In a second step, the body I_0 is evaluated in the store ρ_{var} and the current memory μ, producing an updated store ρ_{body} and an updated memory ρ_{body}. Note that the values of permanent variables of subblocks invoked inside I_0 has been updated in the memory μ_{body}.

Finally, the execution terminates normally by updating the current store ρ with the values of output actual parameters using function *update* in the store ρ_{body}, and by updating the current memory with the values of permanent variables of Ni available in the store ρ_{body}, the values of permanent variables of subblocks invoked inside I_0 being updated in memory μ_{body}.

$$\begin{aligned} \rho' &= \rho \oplus \bigoplus_{i \in 0..n} \text{update } (al_{out_i},\ \text{vars}(dl_{out_i}),\ \rho_{body}) \\ \mu' &= \mu_{body} \oplus [\sigma.Bi \leftarrow \rho_{body} \cdot \text{vars}(dl_{perm})] \end{aligned}$$

7.4 Dynamic semantics of systems

The execution of an actor invocation is defined by a septuple of the form "$\{instance\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{c}} \rho', \mu'$" where *instance* denote an actor invocation, σ is the actor instance stack, ρ and ρ' are stores, μ and μ' are memories, and ℓ is a label.

7.4.1 Block invocation

We assume that a block invoked inside a system is defined as follows.

block B [**const** dl_{const}] (**in** dl_{in_0} ; ... ;**in** dl_{in_m} ;**out** dl_{out_0} ; ... ;**out** dl_{out_n})
{**receive** dl_{rec_0} ; ... ;**receive** dl_{rec_p} ;**send** dl_{send_0} ; ... ;**send** dl_{send_q}} **is**
allocate ...
perm dl_{perm} ,
temp dl_{temp}
I_0
end block

Block B is allocated as follows: B[al_{const}] **as** Bi (where al_{const} is the actual parameter list that corresponds to the declaration list dl_{const}), and Bi can then be invoked as follows:

$$Bi(al_{in_0}\,;\ldots;al_{in_m}\,;al_{out_0}\,;\ldots;al_{out_n})\{al_{rec_0}\,;\ldots;al_{rec_p}\,;al_{send_0}\,;\ldots;al_{send_q}\}$$

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. Receive (resp. send) parameters are computed similarly to input (resp. output) parameters, except that receive parameters can not have default values. The semantic rule of block invocation is the following.

$$\frac{\{I_0\}\ \sigma.Bi,\ \rho_{var},\ \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{body},\ \mu_{body}}{\left\{\begin{array}{l} Bi(al_{in_0}\,;\ldots;al_{in_m}\,;al_{out_0}\,;\ldots;al_{out_n}) \\ \quad\{al_{rec_0}\,;\ldots;al_{rec_p}\,;al_{send_0}\,;\ldots;al_{send_q}\} \end{array}\right\}\sigma,\rho,\mu \xrightarrow{\epsilon}_{\mathsf{c}} \rho',\ \mu'}$$

where stores ρ_{var} and ρ' and memory μ' are constructed as follows.

$$\begin{array}{rcl}
\rho_{const} & \in & \text{init}\ (dl_{const},\ \rho_{global}) \oplus \text{assign}\ (al_{const},\ \text{vars}(dl_{const}),\rho) \\
\rho_{inrec} & \in & \bigoplus_{i\in 0..m} \text{init}\ (dl_{in_i},\ \rho_{global}\oplus\rho_{const}) \oplus \bigoplus_{i\in 0..m} \text{assign}\ (al_{in_i},\ \text{vars}(dl_{in_i}),\rho) \\
 & & \oplus \bigoplus_{i\in 0..p} \text{assign}\ (al_{rec_i},\ \text{vars}(dl_{rec_i}),\rho) \\
\rho_{perm} & = & \left\{\begin{array}{ll} \mu(\sigma.Bi) & \text{if } \sigma.Bi \in dom(\mu) \\ \text{init}\ (dl_{perm},\ \rho_{global}\oplus\rho_{const}) & \text{otherwise} \end{array}\right. \\
\rho_{temp} & = & \text{init}\ (dl_{temp},\ \rho_{global}\oplus\rho_{const}\oplus\rho_{inrec}) \\
\rho_{var} & = & \rho_{global}\oplus\rho_{const}\oplus\rho_{inrec}\oplus\rho_{perm}\oplus\rho_{temp} \\
\rho' & = & \rho \oplus \bigoplus_{i\in 0..n} \text{update}\ (al_{out_i},\ \text{vars}(dl_{out_i}),\ \rho_{body}) \\
 & & \oplus \bigoplus_{i\in 0..q} \text{update}\ (al_{send_i},\ \text{vars}(dl_{send_i}),\ \rho_{body}) \\
\mu' & = & \mu_{body} \oplus [\sigma.Bi \leftarrow \rho_{body} \cdot \text{vars}(dl_{perm})]
\end{array}$$

7.4.2 Environment invocation

We assume that an environment is defined as follows.

environment N [**const** dl_{const}] (**in** dl_{in_0} | ... |**in** dl_{in_m} |**out** dl_{out_0} | ... |**out** dl_{out_n}) **is**
allocate ...
perm dl_{perm} ,
temp dl_{temp}
I_0
end environment

Environment N is allocated as follows: N [**const** al_{const}] **as** Ni, and Ni can then be invoked as follows: $Ni(al_{in_0} \mid \ldots \mid al_{in_m} \mid al_{out_0} \mid \ldots \mid al_{out_n})$.

The execution of environment invocation is guided by signals. It starts by the selection of one actual channel among those passed to the environment. The selection is forced by the call context, i.e., reception or sending of messages from or to a block. 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 is executed in the current store and the current memory and terminates normally by producing an updated store, and updating memory, and passing a label to the context. In such a case, both the actual channel and the environment are called *activated*. 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.

For the sake of accuracy, we distinguish the semantic rules of environment invocation when an input channel is activated and when an output channel is activated since computations on stores are not the same.

Activation of input channels. The semantic rule of environment invocation over the activation of an input actual channel is the following.

$$\frac{\begin{array}{c} i \in 0..m \qquad vars(al_{in_i}) \neq \epsilon \\ \{I_0\}\ \sigma.Ni, \rho_{var}, \mu \xrightarrow{vars(dl_{in_i})}_{\mathsf{i}}\ \rho_{body}, \mu_{body} \end{array}}{\{Ni\ (al_{in_0} \mid \ldots \mid al_{in_m} \mid al_{out_0} \mid \ldots \mid al_{out_n})\}\ \sigma, \rho, \mu \xrightarrow{vars(al_{in_i})}_{\mathsf{c}}\ \rho,\ \mu'}$$

where store ρ_{var} and memory μ' are constructed as follows.

1. Store ρ_{const} is constructed in the same way as in block invocation.

$$\rho_{const} \quad \in \quad \text{init}\ (dl_{const},\ \rho_{global}) \oplus \text{assign}\ (al_{const},\ \text{vars}(dl_{const}), \rho)$$

2. Input parameters of environments have no default values (See rule ACI2 in section 6.6.3). Store ρ_{input} assigns the values of actual parameters of the activated channels to their respective formal parameters, using function *assign* in the store ρ of the caller.

$$\rho_{input} \quad \in \quad \text{assign}\ (al_{in_i},\ \text{vars}(dl_{in_i}), \rho)$$

3. Stores ρ_{perm}, ρ_{temp}, and ρ_{var} are constructed in the same way as in block invocation.

$$\begin{array}{rcl} \rho_{perm} & = & \begin{cases} \mu(\sigma.Ni) & \text{if } \sigma.Ni \in dom(\mu) \\ \text{init}\ (dl_{perm},\ \rho_{global} \oplus \rho_{const}) & \text{otherwise} \end{cases} \\ \rho_{temp} & = & \text{init}\ (dl_{temp},\ \rho_{global} \oplus \rho_{const} \oplus \rho_{input}) \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{input} \oplus \rho_{perm} \oplus \rho_{temp} \end{array}$$

4. No output parameters are produced when an input channel is activated then the store ρ is not updated. Memory μ' updates the current memory with values of permanent variables of all Ni subblocks available in the memory μ_{body} and values of permanent variables of Ni available in the store ρ_{body}.

$$\mu' \quad = \quad \mu_{body} \oplus [\sigma.Ni \leftarrow \rho_{body} \cdot \text{vars}(dl_{perm})]$$

The execution of the body I_0 terminates by passing a label to its context (Ni invocation) containing the list of variables of the executed signal. The execution of Ni invocation terminates by passing a label to its context (the caller of Ni) containing the list of actual parameters of the activated channel.

Activation of output channels. The semantic rule of environment invocation over the activation of an output actual channel is the following.

$$\frac{\begin{array}{c} i \in 0..n \qquad vars(al_{out_i}) \neq \epsilon \\ \{I_0\}\ \sigma.Ni, \rho_{var}, \mu \xrightarrow{?\{vars(dl_{out_i})\}}_{\mathsf{i}} \rho_{body}, \mu_{body} \end{array}}{\{Ni\ (al_{in_0} \mid \ldots \mid al_{in_m} \mid al_{out_0} \mid \ldots \mid al_{out_n})\}\ \sigma, \rho, \mu \xrightarrow{?\{vars(al_{out_i})\}}_{\mathsf{c}} \rho',\ \mu'}$$

where stores ρ_{var} and ρ', and memory μ' are constructed as follows.

1. Store ρ_{const} is constructed as follows.

$$\rho_{const} \in \text{init}\ (dl_{const},\ \rho_{global}) \oplus \text{assign}\ (al_{const},\ \text{vars}(dl_{const}), \rho)$$

2. There is no input actual parameters available when an output channel is activated.

3. Stores ρ_{perm}, ρ_{temp}, and ρ_{var} are constructed as follows.

$$\begin{array}{rcl} \rho_{perm} & = & \begin{cases} \mu(\sigma.Ni) & \text{if } \sigma.Ni \in dom(\mu) \\ \text{init}\ (dl_{perm},\ \rho_{global} \oplus \rho_{const}) & \text{otherwise} \end{cases} \\ \rho_{temp} & = & \text{init}\ (dl_{temp},\ \rho_{global} \oplus \rho_{const}) \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{perm} \oplus \rho_{temp} \end{array}$$

4. Store ρ' updates the current store ρ with the values of parameters of the activated channel available in the store ρ_{body}. Memory μ' updates the current memory with the values of permanent variables of Ni available in the store ρ_{body}, the values of permanent variables of subblocks invoked inside I_0 being updated in memory μ_{body}.

$$\begin{array}{rcl} \rho' & = & \rho \oplus \text{update}\ (al_{out_i},\ \text{vars}(dl_{out_i}),\ \rho_{body}) \\ \mu' & = & \mu_{body} \oplus [\sigma.Ni \leftarrow \rho_{body} \cdot \text{vars}(dl_{perm})] \end{array}$$

The execution of the body I_0 terminates by passing a label to its context (Ni invocation) containing the list of variables of the executed signal preceded by a question mark. The execution of Ni invocation terminates by passing a label to its context (the caller of Ni) containing the list of actual parameters of the activated channel preceded by a question mark.

7.4.3 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). We assume that a medium is defined as follows.

medium M [**const** dl_{const}] {**receive** dl_{rec_0} | ... | **receive** dl_{rec_p} | **send** dl_{send_0} | ... | **send** dl_{send_q}} **is**
  allocate ...
  perm dl_{perm} ,
  temp dl_{temp}
    I_0
end medium

Medium M is allocated as follows: M [**const** al_{const}] **as** Mi, and Mi can then be invoked as follows: Mi (al_{rec_0} | ... | al_{rec_p} | al_{send_0} | ... | al_{send_q}).

Activation of receive channels. The semantic rule of medium invocation over the activation of a receive actual channel is the following.

$$\frac{\begin{array}{c} i \in 0..p \qquad vars(al_{rec_i}) \neq \epsilon \\ \{I_0\}\ \sigma.Mi, \rho_{var}, \mu \xrightarrow{vars(dl_{rec_i})}_{\mathsf{i}} \rho_{body}, \mu_{body} \end{array}}{\{Mi\ (al_{rec_0} \mid \ldots \mid al_{rec_p} \mid al_{send_0} \mid \ldots \mid al_{send_q})\}\ \sigma, \rho, \mu \xrightarrow{vars(al_{rec_i})}_{\mathsf{c}} \rho,\ \mu'}$$

where.

$$\begin{array}{rcl} \rho_{const} & \in & \text{init}\ (dl_{const},\ \rho_{global}) \oplus \text{assign}\ (al_{const},\ \text{vars}(dl_{const}), \rho) \\ \rho_{rec} & \in & \text{assign}\ (al_{rec_i},\ \text{vars}(dl_{rec_i}), \rho) \\ \rho_{perm} & = & \begin{cases} \mu(\sigma.Mi) & \text{if } \sigma.Mi \in dom(\mu) \\ \text{init}\ (dl_{perm},\ \rho_{global} \oplus \rho_{const}) & \text{otherwise} \end{cases} \\ \rho_{temp} & = & \text{init}\ (dl_{temp},\ \rho_{global} \oplus \rho_{const} \oplus \rho_{rec}) \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{rec} \oplus \rho_{perm} \oplus \rho_{temp} \\ \mu' & = & \mu_{body} \oplus [\sigma.Mi \leftarrow \rho_{body} \cdot \text{vars}(dl_{perm})] \end{array}$$

Activation of send channels. The semantic rule of medium invocation over the activation of a send actual channel is the following.

$$\frac{\begin{array}{c} i \in 0..q \qquad vars(al_{send_i}) \neq \epsilon \\ \{I_0\}\ \sigma.Mi, \rho_{var}, \mu \xrightarrow{?\{vars(dl_{send_i})\}}_{\mathsf{i}} \rho_{body}, \mu_{body} \end{array}}{\{Mi\ (al_{rec_0} \mid \ldots \mid al_{rec_p} \mid al_{send_0} \mid \ldots \mid al_{send_q})\}\ \sigma, \rho, \mu \xrightarrow{?\{vars(al_{send_i})\}}_{\mathsf{c}} \rho',\ \mu'}$$

where.

$$\begin{array}{rcl} \rho_{const} & \in & \text{init}\ (dl_{const},\ \rho_{global}) \oplus \text{assign}\ (al_{const},\ \text{vars}(dl_{const}), \rho) \\ \rho_{perm} & = & \begin{cases} \mu(\sigma.Mi) & \text{if } \sigma.Mi \in dom(\mu) \\ \text{init}\ (dl_{perm},\ \rho_{global} \oplus \rho_{const}) & \text{otherwise} \end{cases} \\ \rho_{temp} & = & \text{init}\ (dl_{temp},\ \rho_{global} \oplus \rho_{const}) \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{perm} \oplus \rho_{temp} \\ \rho' & = & \rho \oplus \text{update}\ (al_{send_i},\ \text{vars}(dl_{send_i}),\ \rho_{body}) \\ \mu' & = & \mu_{body} \oplus [\sigma.Mi \leftarrow \rho_{body} \cdot \text{vars}(dl_{perm})] \end{array}$$

7.4.4 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 "$\mu \xrightarrow{\ell}_{\mathsf{s}} \mu'$" where μ and μ' are memories, and ℓ is a label 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, the initial state being the empty memory. Labels are represented by block invocations and have the form "$Bi(al_{in_0}; \ldots; al_{in_m}; al_{out_0}; \ldots; al_{out_n})\{al_{rec_0}; \ldots; al_{rec_p}; al_{send_0}; \ldots; al_{send_q}\}$". Transition "$\mu \xrightarrow{Bi(al_{in_0};\ldots;al_{in_m};al_{out_0};\ldots;al_{out_n})\{al_{rec_0};\ldots;al_{rec_p};al_{send_0};\ldots;al_{send_q}\}}_{\mathsf{s}} \mu'$" means that the combined execution of block Bi together with its connected environments and mediums in the memory μ produces the memory μ' updating the memory of Bi and the memories of its connected actors. More specifically, we assume that a system is defined as follows.

system S (dl_{sys}) **is**
  allocate ...
  temp dl_{temp}
  network
    $block_0, \ldots, block_r$
  constrainedby
    $environment_0, \ldots, environment_s$
  connectedby
    $medium_0, \ldots, medium_t$
end system

where:

$$\begin{array}{rcl} block & ::= & Bi\ (al_{in_0}; \ldots; al_{in_m}; al_{out_0}; \ldots; al_{out_n})\{al_{rec_0}; \ldots; al_{rec_p}; al_{send_0}; \ldots; al_{send_q}\} \\ environment & ::= & Ni\ (al_{in_0} \mid \ldots \mid al_{in_m} \mid al_{out_0} \mid \ldots \mid al_{out_n}) \\ medium & ::= & Mi\ \{al_{rec_0} \mid \ldots \mid al_{rec_p} \mid al_{send_0} \mid \ldots \mid al_{send_q}\} \end{array}$$

System execution is focused around block executions. A block has an *active* behaviour: it executes cyclically (indefinitely) and at each cycle activates its connected environments and mediums, which have *passive* behaviour. Given a block invocation of the form

$$Bi_i\ (al_{in_0}; \ldots; al_{in_m}; al_{out_0}; \ldots; al_{out_n})\{al_{rec_0}; \ldots; al_{rec_p}; al_{send_0}; \ldots; al_{send_q}\}$$

, we define the following symbols.

Connections identification.

- $\mathcal{I}_i$ (for input) (resp., $\mathcal{O}_i$ (for output)) denotes the set of indexes of environments that have output (resp., input) actual channels connected to input (resp., output) actual channels of block Bi_i. This set contains at most one element.
- $\mathcal{R}_i$ (for receive) (resp., $\mathcal{S}_i$ (for send)) denotes the set of indexes of mediums that have send (resp., receive) actual channels connected to receive (resp., send) actual channels of block Bi_i. This set contains at most one element.
- $\mathcal{A}_i$ denotes the set of indexes of input and receive actual channels of block Bi_i of the form "$X_0, \ldots, X_n$" that are not connected to any output actual channel of environments of S and send actual channel of mediums of S, respectively.

$$\begin{array}{rcl} \mathcal{A}_i & = & \{k \in \{in_0, \ldots, in_m\} \mid \mathrm{vars}(al_k) \neq \epsilon \wedge \forall j \in 0..s \quad j \notin \mathcal{I}_i\} \\ & & \cup\{k \in \{rec_0, \ldots, rec_p\} \mid \mathrm{vars}(al_k) \neq \epsilon \wedge \forall j \in 0..t \quad j \notin \mathcal{R}_i\} \end{array}$$

Labels construction.

- $link(j, \mathcal{I}_i)$ where $j \in \mathcal{I}_i$ (resp., $link(j, \mathcal{O}_i)$ where $j \in \mathcal{O}_i$) denotes the set of variables used to connect an input (resp., output) actual channel of block Bi_i to an output (resp., input) actual channel of environment Ni_j.
- $link(j, \mathcal{R}_i)$ where $j \in \mathcal{R}_i$ (resp., $link(j, \mathcal{S}_i)$ where $j \in \mathcal{S}_i$) denotes the set of variables used to connect a receive (resp., send) actual channel of block Bi_i to a send (resp., receive) actual channel of medium Mi_j.

The semantic rule of the system is the following.

$$\frac{\begin{array}{ll} i \in \ 0..r & \\ (\forall j \in \mathcal{R}_i) & \{medium_j\}\ \epsilon,\ [],\ \mu_{\mathcal{R}_j} \xrightarrow{?\{link(j,\mathcal{R}_i)\}}_{\mathsf{c}} \rho_{\mathcal{R}_j},\ \mu'_{\mathcal{R}_j} \\ (\forall j \in \mathcal{I}_i) & \{environment_j\}\ \epsilon,\ [],\ \mu_{\mathcal{I}_j} \xrightarrow{?\{link(j,\mathcal{I}_i)\}}_{\mathsf{c}} \rho_{\mathcal{I}_j},\ \mu'_{\mathcal{I}_j} \\ \{block_i\}\ \epsilon,\ \rho_i,\ \mu_i \xrightarrow{\epsilon}_{\mathsf{c}} \rho'_i,\ \mu'_i & \\ (\forall j \in \mathcal{O}_i) & \{environment_j\}\ \epsilon,\ \rho_{\mathcal{O}_j},\ \mu_{\mathcal{O}_j} \xrightarrow{link(j,\mathcal{O}_i)}_{\mathsf{c}} \rho_{\mathcal{O}_j},\ \mu'_{\mathcal{O}_j} \\ (\forall j \in \mathcal{S}_i) & \{medium_j\}\ \epsilon,\ \rho_{\mathcal{S}_j},\ \mu_{\mathcal{S}_j} \xrightarrow{link(j,\mathcal{S}_i)}_{\mathsf{c}} \rho_{\mathcal{S}_j},\ \mu'_{\mathcal{S}_j} \end{array}}{\mu \xrightarrow{\text{label}(block_i,\ \rho'_i)}_{\mathsf{s}} \mu_s}$$

Intermediate stores and memories are constructed as follows.

1. Store ρ_{any} assigns arbitrary values to parameters of input and receive actual channels of block Bi_i ($i \in 0..r$) whose indexes are in $\mathcal{A}_i$.

$$\rho_{any} \quad \in \quad \bigoplus_{j \in \mathcal{A}_i} \text{any}\ (al_j, \text{types}(dl_j), [])$$

2. Each medium, whose index is in $\mathcal{R}_i$, is invoked in the empty store [] and the memory $\mu_{\mathcal{R}_j}$ corresponding to the current memory of medium Mi_j if any and to the empty memory otherwise. The execution terminates by producing an intermediate store $\rho_{\mathcal{R}_j}$, an intermediate memory $\mu'_{\mathcal{R}_j}$, and passing a label containing the values of the activated channel to the context.

$$\mu_{\mathcal{R}_j} \quad = \quad \begin{cases} [Mi_j \leftarrow \mu(Mi_j)] & \text{if } Mi_j \in \text{dom}(\mu) \\ [] & \text{otherwise} \end{cases}$$

   Medium invocation over the activation of a send channel requires only the values of constant actual parameters to execute. Those parameters are necessarily constant expressions then depend only on global constants at system level (this is not the case at actor level, e.g., a block having a constant parameter can depend on constant parameters of the caller actor). Hence, the empty store is sufficient as input store in medium execution.

3. Each environment, whose index is in $\mathcal{I}_i$, is invoked in the empty store [] and the memory $\mu_{\mathcal{I}_j}$ corresponding to the current memory of environment Ni_j if any and to the empty memory otherwise. The execution terminates by producing an intermediate store $\rho_{\mathcal{I}_j}$, an intermediate memory $\mu'_{\mathcal{I}_j}$, and passing a label containing the values of the activated channel to the context.

$$\mu_{\mathcal{I}_j} \quad = \quad \begin{cases} [Ni_j \leftarrow \mu(Ni_j)] & \text{if } Ni_j \in \text{dom}(\mu) \\ [] & \text{otherwise} \end{cases}$$

4. The block Bi_i is then invoked in the store ρ_i assigning values to input and receive actual channels of Bi_i and in the memory μ_i corresponding to the current memory of Bi_i if any and to the empty memory otherwise. The execution terminates by producing the store ρ'_i and the memory μ'_i.

$$\begin{array}{rcl} \rho_i & = & \rho_{any} \oplus \bigoplus_{j \in \mathcal{R}_i} \rho_{\mathcal{R}_j} \oplus \bigoplus_{j \in \mathcal{I}_i} \rho_{\mathcal{I}_j} \\ \mu_i & = & \begin{cases} [Bi_i \leftarrow \mu(Bi_i)] & \text{if } Bi_i \in \text{dom}(\mu) \\ [] & \text{otherwise} \end{cases} \end{array}$$

5. Each environment, whose index is in $\mathcal{O}_i$, is invoked in the store $\rho_{\mathcal{O}_j}$ and the memory $\mu_{\mathcal{O}_j}$ keeping unchanged the store $\rho_{\mathcal{O}_j}$, producing an intermediate memory $\mu'_{\mathcal{O}_j}$, and passing a label containing the values of the activated channel to the context.

   Environment invocation over the activation of an input channel requires the values of constant actual parameters and of the parameters of the activated channel. The latter parameters are produced by the execution of the block Bi_i and are available in the store ρ'_i.

   If the environment has already been invoked over the activation of an output channel connected to block Bi_i (i.e., $j \in \mathcal{I}_i$), then the updated values of permanent variables are available in the memory $\mu'_{\mathcal{I}_j}$ produced by the first execution of the environment triggered by the current execution cycle of Bi_i. If the environment has been invoked in a previous execution of the block Bi_i or of another block, this means that the updated values of permanent variables are available in the current memory μ. Otherwise, the environment is invoked in the empty memory $[]$.

$$\begin{array}{rcl} \rho_{\mathcal{O}_j} & = & \rho'_i \cdot link(j, \mathcal{O}_i) \\ \mu_{\mathcal{O}_j} & = & \begin{cases} \mu'_{\mathcal{I}_j} & \text{if } j \in \mathcal{I}_i \\ [Ni_j \leftarrow \mu(Ni_j)] & \text{if } j \notin \mathcal{I}_i \text{ and } Ni_j \in \text{dom}(\mu) \\ [] & \text{otherwise} \end{cases} \end{array}$$

6. Each medium, whose index is in $\mathcal{S}_i$, is invoked in the store $\rho_{\mathcal{S}_j}$ and the memory $\mu_{\mathcal{S}_j}$ keeping unchanged the store $\rho_{\mathcal{S}_j}$, producing an intermediate memory $\mu'_{\mathcal{S}_j}$, and passing a label containing the values of the activated channel to the context.

$$\begin{array}{rcl} \rho_{\mathcal{S}_j} & = & \rho'_i \cdot link(j, \mathcal{S}_i) \\ \mu_{\mathcal{S}_j} & = & \begin{cases} \mu'_{\mathcal{R}_j} & \text{if } j \in \mathcal{R}_i \\ [Mi_j \leftarrow \mu(Mi_j)] & \text{if } j \notin \mathcal{R}_i \text{ and } Mi_j \in \text{dom}(\mu) \\ [] & \text{otherwise} \end{cases} \end{array}$$

7. The execution of the system terminates by producing the store ρ'_i (stores $\rho_{\mathcal{O}_j}$ and $\rho_{\mathcal{S}_j}$ being included in ρ'_i), updating the current memory μ with the separate (and independent) memories of all invoked actors, and passing a label to the context.

$$\mu_s = \mu \oplus \bigoplus_{j \in \mathcal{R}_i} \mu'_{\mathcal{R}_j} \oplus \bigoplus_{j \in \mathcal{I}_i} \mu'_{\mathcal{I}_j} \oplus \mu'_i \oplus \bigoplus_{j \in \mathcal{O}_i} \mu'_{\mathcal{O}_j} \oplus \bigoplus_{j \in \mathcal{S}_i} \mu'_{\mathcal{S}_j}$$

The semantics of the whole network are obtained by interleaving all possible executions of Bi_0, ..., Bi_p.

7.5 Dynamic semantics of programs

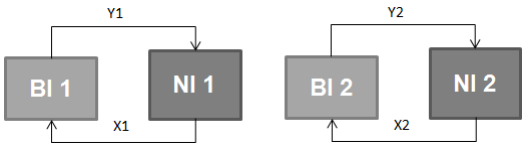
A GRL program can import other programs. The semantics of a program are defined as the semantics of a particular system in the program or in an imported program. Such a system is called the *main* system of the program. By default, the main system of the program is a system named "Main". Alternatively,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

8 Basic Examples

This section presents some examples of systems modeled in GRL.

8.1 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.

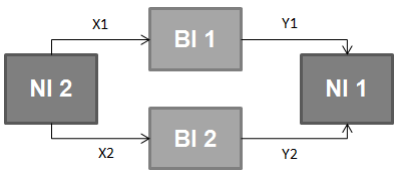



```
system S (X1,X2:nat, Y1,Y2:nat) is
    allocate B as Bi1, B as Bi2,
             N as Ni1, N as Ni2
    network
      Bi1(X1;?Y1),
      Bi2(X2;?Y2)
    constrainedby
      Ni1(Y1|?X1),
      Ni2(Y2|?X2)
end system

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


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

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

block B (in X:nat; out Y:nat) is
    perm lastY:nat := 0
         Y := X+1; lastY := Y
end block

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

8.3 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 :nat:= 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
```

9 Conclusion

In this report, 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.

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 ;
```

C Operational semantics through examples

In this section, we show how LTSs are constructed from GRL programs by applying the SOS rules defined in chapter 7 on two simple examples.

C.1 Nested blocks

The first GRL program we consider is given by the following listing:

```
system S (a,b,c:nat) is
    allocate B as B
    temp d : nat
    network
      B(a,b;?c;?d)
end system

constant C:nat is 2 end constant

block B (in x:nat := 0, y:nat := 0; out z:nat; out w:nat) is
    allocate Sum [0] as Sum
    perm p : nat := 0
    Sum (x, y; ?z);
    p := p+1;
    w := p
end block

block Sum [const thres:nat := C](in i1,i2:nat; out o:nat) is
    perm p : nat := thres
        o := i1+i2;
        p := p+1
end block
```

Note that in this example, the values of variable p (in block B and block Sum) and of variable o (in block Sum) are incremented infinitely. To keep the example as simple as possible, we do not treat the case when an overflow occurs.

C.1.1 Initial state

The initial memory of S is empty: $\mu_0 = []$. The execution of S and the construction of the respective LTS are guided by the execution of block B.

C.1.2 Construction of the transitions

Block B is not connected to any actor inside the system S. Then, $\mathcal{R}_0 = \epsilon$, $\mathcal{S}_0 = \epsilon$, $\mathcal{I}_0 = \epsilon$, $\mathcal{O}_0 = \epsilon$ (0 being the index of B within the list of blocks of S).

Input actual channels "a, b", whose index is 0 in the list of actual channels of block B, are not connected to any environment then $\mathcal{A}_0 = \{0\}$ and parameters "a" and "b" are assigned arbitrary values of type nat. The store ρ_{any} is then:

$$\begin{array}{lcl} \rho_{any} & \in & \text{any}(\langle a, b\rangle, nat \times nat, []) \\ \rho_{any} & \in & \text{any}(a, nat, []) \oplus \text{any}(b, nat, []) \\ \rho_{any} & = & [a \leftarrow ea,\ b \leftarrow eb] \text{ where } ea \in nat \text{ and } eb \in nat \end{array}$$

The construction of the LTS is governed by the following rule:

$$\frac{\{B(a,b;?c;?d)\}\epsilon,\ \rho_{any},\ \mu_0 \xrightarrow{\epsilon}_{\mathsf{c}} \rho_1,\ \mu_1}{\mu_0 \xrightarrow{\texttt{B(ea, eb; ?ea+eb; ?_)}}_{\mathsf{s}} \mu_1} \quad (R1)$$

Block B is executed in the store ρ_{any} and the memory μ_0 and terminates by producing the store ρ_1 and the memory μ_1 (See rule R2 below for ρ_1 and μ_1 computation).

$$\begin{array}{lcl} \rho_1 & = & [a \leftarrow ea,\ b \leftarrow eb,\ c \leftarrow ea+eb,\ d \leftarrow 1] \\ \mu_1 & = & \left[\begin{array}{l} B.Sum \leftarrow [p \leftarrow 1] \\ B \leftarrow [p \leftarrow 1] \end{array}\right] \end{array}$$

Execution of block B. Intermediate stores that B needs to compute its body are the following.

$$\begin{array}{lcl}
\rho_{global} & \in & \text{assign}(2,\ c,\ [\,]) \\
 & = & [c \leftarrow 2] \\
\rho_{const} & = & [\,] \\
\rho_{inrec} & \in & \text{init}(x\texttt{:}\mathbf{nat}\ \texttt{:=}\ 0\texttt{,}\ y\texttt{:}\mathbf{nat}\ \texttt{:=}\ 0,\ \rho_{global} \oplus \rho_{const}) \oplus \text{assign}(\langle a,\ b\rangle,\ \langle x,\ y\rangle,\ \rho_{any}) \\
 & \in & \text{init}(x,\ 0,\ \rho_{global} \oplus \rho_{const}) \oplus \text{init}(y,\ 0,\ \rho_{global} \oplus \rho_{const}) \\
 & & \oplus\text{assign}(a,\ x,\ \rho_{any}) \oplus \text{assign}(b,\ y,\ \rho_{any}) \\
 & = & [x \leftarrow 0] \oplus [y \leftarrow 0] \oplus [x \leftarrow \rho_{any}(a)] \oplus [y \leftarrow \rho_{any}(b)] \\
 & = & [x \leftarrow ea,\ y \leftarrow eb] \\
\rho_{perm} & = & \text{init}(p\texttt{:}\mathbf{nat}\ \texttt{:=}\ 0, \rho_{global} \oplus \rho_{const}) \qquad \text{since } B \notin \text{dom}(\mu_0)\ (\text{dom}(\mu_0)\ =\ \epsilon) \\
 & = & [p \leftarrow 0] \\
\rho_{temp} & = & [\,] \\
\rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{inrec} \oplus \rho_{perm} \oplus \rho_{temp} \\
 & = & [c \leftarrow 2] \oplus [\,] \oplus [x \leftarrow ea,\ y \leftarrow eb] \oplus [p \leftarrow 0] \oplus [\,] \\
 & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 0]
\end{array}$$

The body of B is then executed in the store ρ_{var} and the memory μ_0 and terminates by producing the store ρ_{body} and the memory μ_{body}.

$$\frac{\begin{array}{l}\{Sum(x;y;?z)\}B,\ \rho_{var},\ \mu_0 \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{11},\ \mu_{11} \\ \{p\ \texttt{:=}\ p\texttt{+}1\}B,\ \rho_{11},\ \mu_{11} \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{12},\ \mu_{12} \\ \{w\ \texttt{:=}\ p\}B,\ \rho_{12},\ \mu_{12} \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{body},\ \mu_{body}\end{array}}{\{B(a,b;?c;?d)\}\epsilon,\ \rho_{any},\ \mu_0 \xrightarrow{\epsilon}_{\mathsf{c}} \rho_1,\ \mu_1} \quad (R2)$$

Store ρ_{11} and memory μ_{11} are obtained by the execution of block Sum invocation in the store ρ_{var} and the memory μ_0 (See rule R3 for ρ_{11} and μ_{11} computation).

$$\begin{array}{lcl} \rho_{11} & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 0,\ z \leftarrow ea+eb] \\ \mu_{11} & = & [\ B.Sum \leftarrow [p \leftarrow 1]\ ] \end{array}$$

Stores ρ_{12}, ρ_{body}, and ρ_1 together with memories μ_{12}, μ_{body}, and μ_1 are defined as follows.

$$
\begin{array}{lcl}
\rho_{12} & = & \rho_{11} \oplus [p \leftarrow \rho_{11}(p) + 1] \\
 & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 0,\ z \leftarrow ea + eb] \oplus [p \leftarrow 1] \\
 & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 1,\ z \leftarrow ea + eb] \\
\rho_{body} & = & \rho_{12} \oplus [w \leftarrow \rho_{12}(p)] \\
 & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 1,\ z \leftarrow ea + eb] \oplus [w \leftarrow 1] \\
 & & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 1,\ z \leftarrow ea + eb,\ w \leftarrow 1] \\
\rho_1 & = & \rho_{any} \oplus \mathrm{update}(\texttt{?}c, z, \rho_{body}) \oplus \mathrm{update}(\texttt{?}d, w, \rho_{body}) \\
 & = & [a \leftarrow ea,\ b \leftarrow eb] \oplus [c \leftarrow \rho_{body}(z)] \oplus [d \leftarrow \rho_{body}(w)] \\
 & = & [a \leftarrow ea,\ b \leftarrow eb,\ c \leftarrow ea + eb,\ d \leftarrow 1]
\end{array}
$$

$$
\begin{array}{lcl}
\mu_{12} & = & \left[\ B.Sum \leftarrow [p \leftarrow 1]\ \right] \\
\mu_{body} & = & \left[\ B.Sum \leftarrow [p \leftarrow 1]\ \right] \\
\mu_1 & = & \mu_{body} \oplus [B \leftarrow \rho_{body} \centerdot \mathrm{vars}(p\,\mathbf{:nat}\ \texttt{:=}\ 0)] \\
 & = & \left[\begin{array}{l} B.Sum \leftarrow [p \leftarrow 1] \\ B \leftarrow [p \leftarrow 1] \end{array}\right]
\end{array}
$$

Execution of block Sum. Intermediate stores that block *Sum* needs to compute its body are the following.

$$
\begin{array}{lcll}
\rho_{const1} & \in & \mathrm{init}(thres\,\mathbf{:nat}\ \texttt{:=}\ C, \rho_{global}) \oplus \mathrm{assign}(0, thres, \rho_{var}) & \\
 & = & [thres \leftarrow \rho_{global}(C)] \oplus [thres \leftarrow 0] & \\
 & = & [thres \leftarrow 2] \oplus [thres \leftarrow 0] & \\
 & = & [thres \leftarrow 0] & \\
\rho_{input1} & = & \mathrm{assign}(x, i1, \rho_{var}) \oplus \mathrm{assign}(y, i2, \rho_{var}) & \\
 & = & [i1 \leftarrow \rho_{var}(x)] \oplus [i2 \leftarrow \rho_{var}(y)] & \\
 & = & [i1 \leftarrow ea,\ i2 \leftarrow eb] & \\
\rho_{perm1} & = & \mathrm{init}(p\,\mathbf{:nat}\ \texttt{:=}\ thres,\ \rho_{global} \oplus \rho_{const1}) & \text{since } B.Sum \notin \mathrm{dom}(\mu_0) \\
 & = & [p \leftarrow (\rho_{global} \oplus \rho_{const1})(thres)] & \\
 & = & [p \leftarrow 0] & \\
\rho_{temp1} & = & [] & \\
\rho_{var1} & = & \rho_{global} \oplus \rho_{const1} \oplus \rho_{input1} \oplus \rho_{perm1} \oplus \rho_{temp1} & \\
 & = & [c \leftarrow 2] \oplus [thres \leftarrow 0] \oplus [i1 \leftarrow ea,\ i2 \leftarrow eb] \oplus [p \leftarrow 0] \oplus [] & \\
 & = & [c \leftarrow 2,\ thres \leftarrow 0,\ i1 \leftarrow ea,\ i2 \leftarrow eb,\ p \leftarrow 0] &
\end{array}
$$

The body of block *Sum* is then executed in the store ρ_{var1} and the memory μ_0 and terminates by producing the store ρ_{11} and the memory μ_{11}.

$$
\frac{\begin{array}{l} \{o\ \texttt{:=}\ i1\texttt{+}i2\}B.Sum,\ \rho_{var1},\ \mu_0 \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{111},\ \mu_{111} \\ \{p\ \texttt{:=}\ p\texttt{+}1\}B.Sum,\ \rho_{111},\ \mu_{111} \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{body1},\ \mu_{body1} \end{array}}{\{Sum(x\,\texttt{;}\,y\,\texttt{;?}z)\}B,\ \rho_{var},\ \mu_0 \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{11},\ \mu_{11}} \quad (R3)
$$

Stores ρ_{111}, ρ_{body1}, and ρ_{11} together with memories μ_{111}, μ_{body1}, and μ_{11} are defined as follows.

$$
\begin{array}{lcl}
\rho_{111} & = & \rho_{var1} \oplus [o \leftarrow \rho_{var1}(i1) + \rho_{var1}(i2)] \\
 & = & [c \leftarrow 2,\ thres \leftarrow 0,\ i1 \leftarrow ea,\ i2 \leftarrow eb,\ p \leftarrow 0] \oplus [o \leftarrow ea + eb] \\
 & = & [c \leftarrow 2,\ thres \leftarrow 0,\ i1 \leftarrow ea,\ i2 \leftarrow eb,\ p \leftarrow 0,\ o \leftarrow ea + eb] \\
\rho_{body1} & = & \rho_{111} \oplus [p \leftarrow \rho_{111}(p) + 1] \\
 & = & [c \leftarrow 2,\ thres \leftarrow 0,\ i1 \leftarrow ea,\ i2 \leftarrow eb,\ p \leftarrow 0,\ o \leftarrow ea + eb] \oplus [p \leftarrow 1] \\
 & = & [c \leftarrow 2,\ thres \leftarrow 0,\ i1 \leftarrow ea,\ i2 \leftarrow eb,\ p \leftarrow 1,\ o \leftarrow ea + eb] \\
\rho_{11} & = & \rho_{var} \oplus \mathrm{update}(?z, o, \rho_{body1}) \\
 & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 0] \oplus [z \leftarrow \rho_{body1}(o)] \\
 & = & [c \leftarrow 2,\ x \leftarrow ea,\ y \leftarrow eb,\ p \leftarrow 0,\ z \leftarrow ea + eb]
\end{array}
$$

$$
\begin{array}{lcl}
\mu_{111} & = & \mu_0 \\
 & = & [] \\
\mu_{body1} & = & \mu_{111} \\
 & = & [] \\
\mu_{11} & = & \mu_{body1} \oplus [B.Sum \leftarrow \rho_{body1} \centerdot \mathrm{vars}(p\!:\!\mathbf{nat}\ \texttt{:=}\ thres)] \\
 & = & [B.Sum \leftarrow [p \leftarrow 1]]
\end{array}
$$

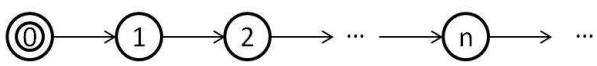
C.1.3 Generated LTS

The label corresponding to the execution of block B inside system S is obtained using function *label* as follows.

$$
\begin{array}{lcl}
\mathrm{label}(B\ \texttt{(}a\texttt{,}\ b\texttt{;}\ ?c\texttt{;}\ ?d\texttt{)},\ \rho_1) & = & \texttt{B}\ \texttt{(}\mathrm{label}(\langle\langle a,\ b\rangle,\ ?c,\ ?d\rangle,\ \rho_1)\texttt{)} \\
 & = & \texttt{B}\ \texttt{(}\mathrm{label}(a, \rho_1)\texttt{,}\ \mathrm{label}(b, \rho_1)\texttt{;} \\
 & & \quad \mathrm{label}(?c, \rho_1)\texttt{;}\ \mathrm{label}(?d, \rho_1)\texttt{)} \\
 & = & \texttt{B}\ \texttt{(}\rho_1(a)\texttt{,}\ \rho_1(b)\texttt{;}\ ?\rho_1(c)\texttt{;}\ ?_\texttt{)} \\
 & = & \texttt{B}\ \texttt{(ea; eb; ?ea+eb; ?_)}
\end{array}
$$

The value of variable d is hidden in the produced label since d has been declared as temporary variable of the system.

The generated LTS consists of an infinite sequence of states, because of the occurrence of permanent variables in blocks B and Sum whose values increment indefinitely. Each state corresponds to the memory of block B, the initial state being empty. All transitions have the form "`B (ea; eb; ?ea + eb; ?_)`".



C.2 Strict alternation of blocks

The second GRL program we consider is given by the following listing.

```
system S (a1,a2:nat, b1,b2:nat, c1,c2:nat) is
    allocate B as B1, B as B2,
             N as N1, N as N2
    temp d1,d2:nat
    network
      B1(a1,b1;?c1;?_),
      B2(a2,b2;?c2;?_)
    constrainedby
      N1(d1|d2),
      N2(c1|c2)
end system

block B (in x,y:nat := 0; out z:nat; out w:nat) is
    perm p:nat := 0
    z := x+y;
    p := p+1;
    w := p
end block

environment N (in z1:nat | in z2:nat) is
    perm last1:bool := false
      if not(last1) then
        on z1 -> last1 := true
      else
        on z2 -> last1 := false
      end if
end environment
```

C.2.1 Initial state

The initial memory of S is empty: $\rho_0 = []$. The execution of S and the construction of the LTS are guided by the parallel execution of blocks $B1$ and $B2$.

C.2.2 Construction of the first transition

Blocks $B1$ and $B2$ evolve in asynchronous parallelism, then two transitions can be performed from the initial state, each transition corresponding to the invocation of one block together with its connected actors.

Execution of block $B1$ and its connected actors. Connections of $B1$ (whose index is 0 within the list of blocks inside S) with other actors are defined as follows.

- No medium is connected to $B1$, then $\mathcal{R}_0 = \epsilon$ and $\mathcal{S}_0 = \epsilon$.
- No environment is connected to the inputs of $B1$, then $\mathcal{I}_0 = \epsilon$.
- $B1$ has the output actual channel "?*c1*" connected to the input actual channel "*c1*" of environment $N2$ (whose index is 1 within the list of environments inside S), then $\mathcal{O}_0 = \{1\}$ and $link(1, \mathcal{O}_0) = \{c1\}$.

– The input actual channel "*a1* , *b1*" (whose index is 0 in the list of actual channels of block $B1$) is not connected to any environment. Then, $\mathcal{A}_0 = \{0\}$ and parameters "$a1$" and "$b1$" are assigned arbitrary values of type *nat*.

The store ρ_0 in which $B1$ should execute is then:

$$\begin{array}{lcl} \rho_0 & \in & \text{any}(\langle a1, b1\rangle, nat \times nat, []) \\ \rho_0 & = & [a1 \leftarrow ea1,\ b1 \leftarrow eb1] \text{ where } ea1 \in nat \text{ and } eb1 \in nat \end{array}$$

The execution of block $B1$ inside system S is governed by the following rule:

$$\frac{\begin{array}{l}\{B1\texttt{(}a1\texttt{,}b1\texttt{;?}c1\texttt{;?_)}\}\epsilon,\ \rho_0,\ \mu_{01} \xrightarrow{\epsilon}_{\mathsf{c}} \rho_{01},\ \mu_{02} \\ \{N2\texttt{(}c1 \mid c2\texttt{)}\}\epsilon,\ \rho_{02},\ \mu_{03} \xrightarrow{c1}_{\mathsf{c}} \rho_{02},\ \mu_{04}\end{array}}{\mu_0 \xrightarrow{\texttt{B1(ea1, eb1; ?ea1+eb1; ?_)}}_{\mathsf{s}} \mu_1} \quad (R4)$$

Block $B1$ is executed in the initial store ρ_0 and the memory μ_{01} and terminates by producing the store ρ_{01} and the memory μ_{02} (See rule R5 below for ρ_{01} and μ_{02} computation).

$$\begin{array}{lcll} \mu_{01} & = & [] & \text{since } B1 \notin \text{ dom } (\mu_0) \\ \\ \rho_{01} & = & [a1 \leftarrow ea1,\ b1 \leftarrow eb1,\ c1 \leftarrow ea1 + eb1] & \\ \mu_{02} & = & [B1 \leftarrow [p \leftarrow 1]] & \end{array}$$

Environment $N2$ is executed in the store ρ_{02} and the memory μ_{02} defined below and terminates by keeping unchanged the store ρ_{02}, producing the memory μ_{04} (See rule R6 below for ρ_{02} and μ_{04} computation), and passing the label $c1$ to its context.

$$\begin{array}{lcll} \rho_{02} & = & \rho_{01} \bullet link(1, \mathcal{O}_0) & \\ & = & [c1 \leftarrow ea1 + eb1] & \\ \mu_{03} & = & [] & \text{since } 1 \notin \mathcal{I}_0 \text{ and } N2 \notin \text{dom}(\mu_0) \\ \\ \mu_{04} & = & [N2 \leftarrow [last1 \leftarrow true]] & \end{array}$$

Execution of block $B1$**.** Intermediate stores that B1 needs to compute its body are the following.

$$\begin{array}{lcll} \rho_{global} & = & [] & \\ \rho_{const} & = & [] & \\ \rho_{inrec} & \in & \text{init}(x\texttt{,}y\texttt{:}\mathbf{nat}\ \texttt{:=}\ 0,\ \rho_{global} \oplus \rho_{const}) \oplus \text{assign}(\langle a1,\ b1\rangle,\ \langle x,\ y\rangle,\ \rho_0) & \\ & = & [x \leftarrow 0,\ y \leftarrow 0] \oplus [x \leftarrow \rho_0(a1)] \oplus [y \leftarrow \rho_0(b1)] & \\ & = & [x \leftarrow ea,\ y \leftarrow eb] & \\ \rho_{perm} & = & \text{init}(p\texttt{:}\mathbf{nat}\ \texttt{:=}\ 0,\ \rho_{global} \oplus \rho_{const}) & \text{since dom}(\mu_0) \ = \ \epsilon \\ & = & [p \leftarrow 0] & \\ \rho_{temp} & = & [] & \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{inrec} \oplus \rho_{perm} \oplus \rho_{temp} & \\ & = & [] \oplus [] \oplus [x \leftarrow ea1,\ y \leftarrow eb1] \oplus [p \leftarrow 0] \oplus [] & \\ & = & [x \leftarrow ea1,\ y \leftarrow eb1,\ p \leftarrow 0] & \end{array}$$

The body of $B1$ is then executed in the store ρ_{var} and the memory μ_{01} and terminates by producing the store ρ_{body} and the memory μ_{body}.

$$\frac{\begin{array}{l}\{z\ \texttt{:=}\ x\texttt{+}y\}B1,\ \rho_{var1},\ \mu_{01} \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{011},\ \mu_{011}\\ \{p\ \texttt{:=}\ p\texttt{+}1\}B1,\ \rho_{011},\ \mu_{011} \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{012},\ \mu_{012}\\ \{w\ \texttt{:=}\ p\}B1,\ \rho_{012},\ \mu_{012} \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{body},\ \mu_{body}\end{array}}{\{B1\texttt{(}a1\texttt{,}b1\texttt{;?}c1\texttt{;?_)}\}\epsilon,\ \rho_0,\ \mu_0 \xrightarrow{\epsilon}_{\mathrm{i}} \rho_{01},\ \mu_{02}} \quad (R5)$$

Stores ρ_{011}, ρ_{012}, and ρ_{body} together with memories μ_{011}, μ_{012}, and μ_{body} are constructed as follows.

$$\begin{array}{lcl}
\rho_{011} & = & [x \leftarrow ea1,\ y \leftarrow eb1,\ p \leftarrow 0,\ z \leftarrow ea1 + eb1]\\
\rho_{012} & = & [x \leftarrow ea1,\ y \leftarrow eb1,\ p \leftarrow 1,\ z \leftarrow ea1 + eb1]\\
\rho_{body} & = & [x \leftarrow ea1,\ y \leftarrow eb1,\ p \leftarrow 1,\ z \leftarrow ea1 + eb1,\ w \leftarrow 1]\\
\rho_{01} & = & \rho_0 \oplus \mathrm{update}(\texttt{?}c, z, \rho_{body}) \oplus \mathrm{update}(\texttt{?_}, w, \rho_{body})\\
 & = & [a1 \leftarrow ea1,\ b1 \leftarrow eb1,\ c1 \leftarrow ea1 + eb1]\\
\\
\mu_{011} & = & []\\
\mu_{012} & = & []\\
\mu_{body} & = & []\\
\mu_{02} & = & \mu_0 \oplus [B1 \leftarrow \rho_{body} \cdot \mathrm{vars}(p\texttt{:}\mathbf{nat}\ \texttt{:=}\ 0)]\\
 & = & [B1 \leftarrow [p \leftarrow 1]]
\end{array}$$

Execution of environment $N2$. The invocation of $N2$ is triggered by the reception of the value "$ea1 + eb1$" from block $B1$ over the actual channel "$c1$". Intermediate stores that $N2$ needs to compute are the following.

$$\begin{array}{lcll}
\rho_{global} & = & [] & \\
\rho_{const} & = & [] & \\
\rho_{input} & \in & \mathrm{assign}(c1,\ z1,\ \rho_{02}) & \\
 & = & [z1 \leftarrow ea1 + eb1] & \\
\rho_{perm} & = & \mathrm{init}(last1\texttt{:}\mathbf{bool}\ \texttt{:=}\ \mathbf{false},\ \rho_{global} \oplus \rho_{const}) & \text{since } N2 \notin \mathrm{dom}(\mu_{03})\\
 & = & [last1 \leftarrow false] & \\
\rho_{temp} & = & [] & \\
\rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{input} \oplus \rho_{perm} \oplus \rho_{temp} & \\
 & = & [] \oplus [] \oplus [z1 \leftarrow ea1 + eb1] \oplus [last1 \leftarrow false] \oplus [] & \\
 & = & [z1 \leftarrow ea1 + eb1,\ last1 \leftarrow false] &
\end{array}$$

Then, the body of $N2$ is executed in the store ρ_{var} and the memory μ_{03} ignoring all signal statements that do not correspond to "$z1$" (the respective formal parameter of actual parameter "$c1$") and terminates by keeping unchanged the store ρ_{var}, producing the memory μ_{04}, and passing the label `c1` to the context.

$$\frac{\{\mathit{last1}\ \texttt{:=}\ \mathbf{true}\}N2,\ \rho_{var},\ \mu_{03} \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{body},\ \mu_{body}}{\{\mathbf{not}\texttt{(}\mathit{last1}\texttt{)}\}\rho_{var} \rightarrow_{\mathsf{e}} \mathbf{true}} \quad (R62)$$

$$\frac{\begin{array}{l}\{\mathbf{on}\ \mathit{z1}\ \texttt{->}\ \mathit{last1}\ \texttt{:=}\ \mathbf{true}\}N2,\ \rho_{var},\ \mu_{03} \xrightarrow{z1}_{\mathsf{i}} \rho_{body},\ \mu_{body} \\ z1 \in \mathrm{dom}(\rho_{var})\end{array}}{\left\{\begin{array}{l}\mathbf{if\ not}\texttt{(}\mathit{last1}\texttt{)}\ \mathbf{then} \\ \quad \mathbf{on}\ \mathit{z1}\ \texttt{->}\ \mathit{last1}\ \texttt{:=}\ \mathbf{true} \\ \mathbf{else} \\ \quad \mathbf{on}\ \mathit{z2}\ \texttt{->}\ \mathit{last1}\ \texttt{:=}\ \mathbf{false} \\ \mathbf{end\ if}\end{array}\right\}\ N2,\ \rho_{var},\ \mu_{03} \xrightarrow{z1}_{\mathsf{i}} \rho_{body},\ \mu_{body}} \quad (R61)$$

$$\frac{\{N2\texttt{(}c1 \mid c2\texttt{)}\}\epsilon,\ \rho_{02},\ \mu_{03} \xrightarrow{c1}_{\mathsf{i}} \rho_{02},\ \mu_{04}}{} \quad (R6)$$

Store ρ_{body} together with memories μ_{body} and ρ_{03} are defined as follows.

$$\begin{array}{lcl}\rho_{body} & = & \rho_{var} \oplus [\mathit{last1} \leftarrow \mathit{true}] \\ & = & [z1 \leftarrow ea1 + eb1,\ \mathit{last1} \leftarrow \mathit{false}] \oplus [\mathit{last1} \leftarrow \mathit{true}] \\ & = & [z1 \leftarrow ea1 + eb1,\ \mathit{last1} \leftarrow \mathit{true}]\end{array}$$

$$\begin{array}{lcl}\mu_{body} & = & \mu_{03} \\ & = & [] \\ \mu_{04} & = & \mu_{body} \oplus [N2 \leftarrow \rho_{body} \cdot \mathrm{vars}(\mathit{last1}\ \texttt{:}\mathbf{bool}\ \texttt{:=}\ \mathbf{false})] \\ & = & [N2 \leftarrow [\mathit{last1} \leftarrow \mathit{true}]]\end{array}$$

Final store and memory. The execution of block $B1$ and its connected environment $N2$ terminates by producing the store ρ_1 and the memory μ_1.

$$\begin{array}{lcl}\rho_1 & = & \rho_{01} \\ & = & [a1 \leftarrow ea1,\ b1 \leftarrow eb1,\ c1 \leftarrow ea1 + eb1] \\ \mu_1 & = & \mu_0 \oplus \mu_{02} \oplus \mu_{04} \\ & = & \left[\begin{array}{l} B1 \leftarrow [p \leftarrow 1] \\ N2 \leftarrow [\mathit{last1} \leftarrow \mathit{true}]\end{array}\right]\end{array}$$

Execution of block $B2$ and its connected actors. Connections of $B2$ (whose index is 0 within the list of blocks inside S) with other actors are defined as follows.

- $\mathcal{R}_1 = \epsilon$ and $\mathcal{S}_1 = \epsilon$.
- $\mathcal{I}_1 = \epsilon$.
- $\mathcal{O}_1 = \{1\}$ and $\mathit{link}(1, \mathcal{O}_1) = \{c2\}$.
- $\mathcal{A}_1 = \{0\}$.

The store ρ_0 in which $B2$ should execute is then:

$$\begin{array}{lcl}\rho_0 & \in & \mathrm{any}(\langle a2, b2\rangle, \mathit{nat} \times \mathit{nat}, []) \\ \rho_0 & = & [a2 \leftarrow ea2,\ b2 \leftarrow eb2] \text{ where } ea2 \in \mathit{nat} \text{ and } eb2 \in \mathit{nat}\end{array}$$

The execution of block $B2$ inside system S is governed by the following rule:

$$\frac{\{B2\texttt{(}a2\texttt{,}b2\texttt{;?}c2\texttt{;?_)}\}\epsilon,\ \rho_0,\ \mu_{01} \xrightarrow{\epsilon}_{\mathsf{c}} \rho_{11},\ \mu_{11} \quad \{N2\texttt{(}c1 \mid c2\texttt{)}\}\epsilon,\ \rho_{12},\ \mu_{12} \xrightarrow{c2}_{\mathsf{c}} \rho_{12},\ \mu_{13}}{\mu_0 \xrightarrow{\texttt{B2(ea2, eb2; ?ea2+eb2; ?_)}}_{\mathsf{s}} \mu_1} \quad (R7)$$

Block $B2$ is executed in the initial store ρ_0 and the memory μ_{01} and terminates by producing the store ρ_{11} and the memory μ_{11}.

$$\begin{array}{lcl} \mu_{01} & = & [] \qquad \text{since } B2 \notin \text{ dom } (\mu_0) \\ \\ \rho_{11} & = & [a2 \leftarrow ea2,\ b2 \leftarrow eb2,\ c2 \leftarrow ea2 + eb2] \\ \mu_{11} & = & [B2 \leftarrow [p \leftarrow 1]] \end{array}$$

Environment $N2$ is executed in the store ρ_{12} and the memory μ_{12} defined below and is expected to terminate by keeping unchanged the store ρ_{12} and producing the memory μ_{13} (See rule R8 below for ρ_{12} and μ_{13} computation).

$$\begin{array}{lcl} \rho_{12} & = & \rho_{11} \cdot link(1, \mathcal{O}_1) \\ & = & [c2 \leftarrow ea2 + eb2] \\ \mu_{12} & = & [] \qquad \text{since } 1 \notin \mathcal{I}_1 \text{ and } N2 \notin \text{dom}(\mu_0) \end{array}$$

Execution of block $B2$**.** Blocks $B1$ and $B2$ are instances of the same block B, then the execution of $B2$ is similar to the execution of $B1$ (See rule R5 above).

Execution of environment $N2$**.** The invocation of $N2$ is triggered by the reception of the value "$ea2 + eb2$" from block $B2$ over the actual channel "$c2$". Then the body of $N2$ is executed ignoring all signal statements that do not correspond to "$z2$", the respective formal parameter of actual parameter "$c2$". Intermediate stores that $N2$ needs to compute are the following.

$$\begin{array}{lcll} \rho_{global} & = & [] & \\ \rho_{const} & = & [] & \\ \rho_{input} & \in & \text{assign}(c2,\ z2,\ \rho_{12}) & \\ & = & [z2 \leftarrow ea2 + eb2] & \\ \rho_{perm} & = & \text{init}(last1 : \textbf{bool} := \textbf{false},\ \rho_{global} \oplus \rho_{const}) & \text{since } N2 \notin \text{dom}(\mu_{12}) \\ & = & [last1 \leftarrow false] & \\ \rho_{temp} & = & [] & \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{input} \oplus \rho_{perm} \oplus \rho_{temp} & \\ & = & [] \oplus [] \oplus [z2 \leftarrow ea2 + eb2] \oplus [last1 \leftarrow false] \oplus [] & \\ & = & [z2 \leftarrow ea2 + eb2,\ last1 \leftarrow false] & \end{array}$$

The body of $N2$ can then be executed in the store ρ_{var} and the memory μ_{12} according to the following rule:

$$\frac{\{\mathit{last1}\ \texttt{:=}\ \textbf{false}\}N2,\ \rho_{var},\ \mu_{12} \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{body},\ \mu_{body}}{\{\textbf{not}(\mathit{last1})\}\rho_{var} \rightarrow_{\mathsf{e}} \textbf{false}} \quad (R82)$$

$$\frac{\begin{array}{l}\{\textbf{on}\ \mathit{z2}\ \texttt{->}\ \mathit{last1}\ \texttt{:=}\ \textbf{false}\}N2,\ \rho_{var},\ \mu_{12} \xrightarrow{z2}_{\mathsf{i}} \rho_{body},\ \mu_{body} \\ z2 \in \mathrm{dom}(\rho_{var})\end{array}}{\left\{\begin{array}{l}\textbf{if not}(\mathit{last1})\ \textbf{then} \\ \quad \textbf{on}\ \mathit{z1}\ \texttt{->}\ \mathit{last1}\ \texttt{:=}\ \textbf{true} \\ \textbf{else} \\ \quad \textbf{on}\ \mathit{z2}\ \texttt{->}\ \mathit{last1}\ \texttt{:=}\ \textbf{false} \\ \textbf{end if}\end{array}\right\}\ N2,\ \rho_{var},\ \mu_{12} \xrightarrow{z2}_{\mathsf{i}} \rho_{body},\ \mu_{body}} \quad (R81)$$

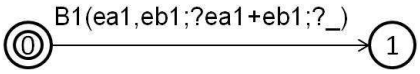
$$\frac{\{\mathit{N2}\texttt{(}\mathit{c1} \mid \mathit{c2}\texttt{)}\}\epsilon,\ \rho_{12},\ \mu_{12} \xrightarrow{c2}_{\mathsf{i}} \rho_{12},\ \mu_{13}}{} \quad (R8)$$

Variable "*last*1" is evaluated to "*false*" in the store ρ_{var}, then the conditional statement allows only the execution of the signal "**on** *z1* `->` *last1* `:=` **true**" guarded by the condition "**not**`(`*last1*`)`" (rule R81). However, variable $z1$ can not be evaluated in the store ρ_{var} ($z1 \notin rho_{var}$). Rule R81 can not be crossed blocking the execution of environment N2, and no store neither memory can be produced. Consequently, the execution of block $B2$ is cancelled and the global memory of the system S is not updated since it requires the execution of the block and all its connected actors to be performed normally and atomically.

Construction of the transition. We have shown previously that only block $B1$ can be executed from the initial state μ_0 of the system. The label corresponding to the first transition of the system is obtained using function *label* as follows.

$$\begin{array}{rcl}\mathrm{label}(\mathit{B1}\ \texttt{(}\mathit{a1}\texttt{,}\ \mathit{b1}\texttt{;}\ \texttt{?}\mathit{c1}\texttt{;}\ \texttt{?_}\texttt{)},\ \rho_1) & = & \texttt{B1}\ (\mathrm{label}(\langle\langle a1,\ b1\rangle,\ ?c1,\ ?_\rangle,\ \rho_1)) \\ & = & \texttt{B1}\ (\mathrm{label}(a1,\rho_1)\texttt{,}\ \mathrm{label}(b1,\rho_1)\texttt{;} \\ & & \quad \mathrm{label}(?c1,\rho_1)\texttt{;}\ \mathrm{label}(?_,\rho_1)) \\ & = & \texttt{B1}\ (\rho_1(a1)\texttt{,}\ \rho_1(b1)\texttt{;}\ \texttt{?}\rho_1(c1)\texttt{;}\ \texttt{?_}) \\ & = & \texttt{B1}\ (\texttt{ea1;}\ \texttt{eb1;}\ \texttt{?ea1+eb1;}\ \texttt{?_})\end{array}$$

The LTS of the system after the execution of $B1$ is the following.



C.2.3 Execution of the second transition.

Block $B1$ behaves similarly to block $B2$ in the first round (transition), and conversely.

Execution of block $B2$ and its connected actors. Connections of $B2$ with other actors are as defined in the previous section: $\mathcal{R}_1 = \epsilon$, $\mathcal{S}_1 = \epsilon$, $\mathcal{I}_1 = \epsilon$, $\mathcal{O}_1 = \{1\}$ and $link(1, \mathcal{O}_1) = \{c2\}$, $\mathcal{A}_1 = \{0\}$. The initial store is $\rho_0 = [a2 \leftarrow ea2,\ b2 \leftarrow eb2]$ where $ea2 \in nat$ and $eb2 \in nat$. The execution of system S is governed by the following rule:

$$\frac{\begin{array}{l}\{\mathit{B2}\texttt{(}\mathit{a2}\texttt{,}\mathit{b2}\texttt{;}\texttt{?}\mathit{c2}\texttt{;}\texttt{?_}\texttt{)}\}\epsilon,\ \rho_0,\ \mu_{11} \xrightarrow{\epsilon}_{\mathsf{c}} \rho_{11},\ \mu_{12} \\ \{\mathit{N2}\texttt{(}\mathit{c1} \mid \mathit{c2}\texttt{)}\}\epsilon,\ \rho_{12},\ \mu_{13} \xrightarrow{c2}_{\mathsf{c}} \rho_{12},\ \mu_{14}\end{array}}{\mu_1 \xrightarrow{\texttt{B2(ea2, eb2; ?ea2+eb2; ?_)}}_{\mathsf{s}} \mu_2} \quad (R7)$$

Block $B2$ is executed in the initial store ρ_0 and the memory μ_{11}, and terminates by producing the store ρ_{11} and the memory μ_{12} exactly in the same way as block $B1$ execution in the previous section since its the first execution of $B2$ together with its connected actors (see rule R5 above for intermediate computations).

$$\begin{array}{lcl} \mu_{11} & = & [] \qquad \text{since } B2 \notin \text{ dom } (\mu_1) \\ \\ \rho_{11} & = & [a2 \leftarrow ea2,\ b2 \leftarrow eb2,\ c2 \leftarrow ea2 + eb2] \\ \mu_{12} & = & [B2 \leftarrow [p \leftarrow 1]] \end{array}$$

Environment $N2$ is executed in the store ρ_{12} and the memory μ_{13} defined below and terminates by keeping unchanged the store ρ_{12}, producing the memory μ_{14} (See rule R10 below for ρ_{12} and μ_{14} computation), and passing the label $c2$ to its context.

$$\begin{array}{lcl} \rho_{12} & = & \rho_{11} \cdot link(1, \mathcal{O}_1) \\ & = & [c2 \leftarrow ea2 + eb2] \\ \mu_{13} & = & \mu_1(N2) \qquad \text{since } 1 \notin \mathcal{I}_1 \text{ and } N2 \in \text{ dom } (\mu_1) \\ & = & [N2 \leftarrow [last1 \leftarrow true]] \\ \mu_{14} & = & [N2 \leftarrow [last1 \leftarrow false]] \end{array}$$

Execution of environment $N2$. The invocation of $N2$ is triggered by the reception of the value "$ea2 + eb2$" from block $B2$ over the actual channel "$c2$". Intermediate stores that $N2$ needs to compute are as the following.

$$\begin{array}{lcl} \rho_{global} & = & [] \\ \rho_{const} & = & [] \\ \rho_{input} & = & [z2 \leftarrow ea2 + eb2] \\ \rho_{perm} & = & \mu_{13}(N2) \qquad \text{since } N2 \in \text{dom}(\mu_{13}) \\ & = & [last1 \leftarrow true] \\ \rho_{temp} & = & [] \\ \rho_{var} & = & \rho_{global} \oplus \rho_{const} \oplus \rho_{input} \oplus \rho_{perm} \oplus \rho_{temp} \\ & = & [] \oplus [] \oplus [z2 \leftarrow ea2 + eb2] \oplus [last1 \leftarrow true] \oplus [] \\ & = & [z2 \leftarrow ea2 + eb2,\ last1 \leftarrow true] \end{array}$$

The body of $N2$ can then be executed in the store ρ_{var} and the memory μ_{13} and terminates by producing the store ρ_{body} and the memory μ_{body} according to the following rule:

$$\dfrac{\dfrac{\{\textit{last1}\ \texttt{:=}\ \textbf{false}\}N2,\ \rho_{var},\ \mu_{13} \xrightarrow{\epsilon}_{\mathsf{i}} \rho_{body},\ \mu_{body}}{\begin{array}{l}\{\textbf{not}(\textit{last1})\}\rho_{var} \rightarrow_{\mathsf{e}} \textbf{false} \\ \{\textbf{on}\ \textit{z2}\ \texttt{->}\ \textit{last1}\ \texttt{:=}\ \textbf{false}\}N2,\ \rho_{var},\ \mu_{13} \xrightarrow{z2}_{\mathsf{i}} \rho_{body},\ \mu_{body} \\ z2 \in \text{dom}(\rho_{var})\end{array}}\ (R102)}{\left\{\begin{array}{l}\textbf{if not}(\textit{last1})\ \textbf{then} \\ \quad \textbf{on}\ \textit{z1}\ \texttt{->}\ \textit{last1}\ \texttt{:=}\ \textbf{true} \\ \textbf{else} \\ \quad \textbf{on}\ \textit{z2}\ \texttt{->}\ \textit{last1}\ \texttt{:=}\ \textbf{false} \\ \textbf{end if}\end{array}\right\} N2,\ \rho_{var},\ \mu_{13} \xrightarrow{z2}_{\mathsf{i}} \rho_{body},\ \mu_{body}}\ (R101)}{\{\textit{N2}\,(\textit{c1} \mid \textit{c2})\}\epsilon,\ \rho_{12},\ \mu_{13} \xrightarrow{c2}_{\mathsf{i}} \rho_{12},\ \mu_{14}}\ (R10)$$

Store ρ_{body} together with memories μ_{body} and ρ_{13} are defined as follows.

$$\begin{array}{lcl}
\rho_{body} & = & \rho_{var} \oplus [last1 \leftarrow false] \\
 & = & [z2 \leftarrow ea2 + eb2,\ last1 \leftarrow true] \oplus [last1 \leftarrow false] \\
 & = & [z2 \leftarrow ea2 + eb2,\ last1 \leftarrow false] \\
\mu_{body} & = & \mu_{13} \\
 & = & [N2 \leftarrow [last1 \leftarrow true]] \\
\mu_{14} & = & \mu_{body} \oplus [N2 \leftarrow \rho_{body} \cdot \mathrm{vars}(last1 : \mathbf{bool} := \mathbf{false})] \\
 & = & [N2 \leftarrow [last1 \leftarrow false]]
\end{array}$$

Final store and memory. The execution of block $B2$ and its connected environment $N2$ terminates by producing the store ρ_2 and the memory μ_2.

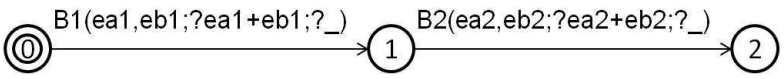
$$\begin{array}{lcl}
\rho_2 & = & \rho_{11} \\
 & = & [a2 \leftarrow ea2,\ b2 \leftarrow eb2,\ c2 \leftarrow ea2 + eb2] \\
\mu_2 & = & \mu_1 \oplus \mu_{12} \oplus \mu_{14} \\
 & = & \left[\begin{array}{l} B1 \leftarrow [p \leftarrow 1] \\ N2 \leftarrow [last1 \leftarrow true] \end{array}\right] \oplus \left[\ B2 \leftarrow [p \leftarrow 1]\ \right] \oplus \left[\ N2 \leftarrow [last1 \leftarrow false]\ \right] \\
 & = & \left[\begin{array}{l} B1 \leftarrow [p \leftarrow 1] \\ B2 \leftarrow [p \leftarrow 1] \\ N2 \leftarrow [last1 \leftarrow false] \end{array}\right]
\end{array}$$

Execution of block $B1$ and its connected actors. The execution of block $B1$ is blocked in memory μ_1 in the same way as the execution of block $B2$ in memory μ_0 (See rule R8 above for details).

Construction of the transition. Only block $B2$ can be executed from the state μ_1 of the system. The label corresponding to the second transition of the system is obtained as follows.

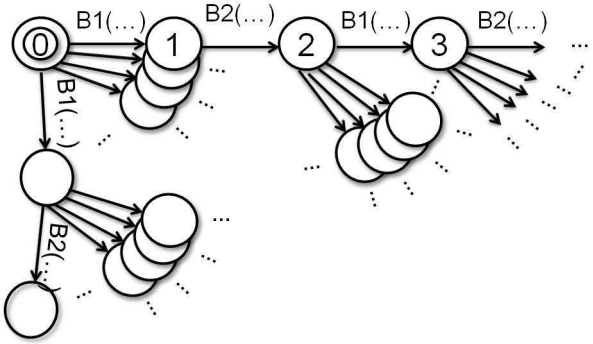
$$\mathrm{label}(B2\ (a2,\ b2;\ ?c2;\ ?_),\ \rho_1) \quad = \quad \texttt{B2 (ea2; eb2; ?ea2+eb2; ?_)}$$

The LTS of the system after the execution of $B2$ is the following.



C.2.4 Construction of the whole LTS.

The remainder of the LTS is constructed as detailed above. Each state corresponds to the actual memories of blocks $B1$ and $B2$ and environment $N2$. Transitions correspond to either the invocation of $B1$ or the invocation of $B2$, the alternation being ensured by environment $N2$. Note that environment $N1$ is never invoked since not connected to neither $B1$ nor $B2$ and variables $d1$ and $d2$ are not considered in computations.


B1(…)
B2(…)
B1(…)
B2(…)
0
1
2
3
B1(…)
B2(…)

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