

The image shows the cover of the 'Scade Language Reference Manual'. It features a large, solid brown rectangle in the lower half of the page. To the left and right of this rectangle are two smaller, lighter brown rectangles, one on each side, creating a cross-like shape. The title 'Scade Language Reference Manual' is written in white, bold, sans-serif font, centered within the large brown rectangle.

# **Scade Language Reference Manual**



## Contacts

### Legal Contact

Ansys France  
15 place Georges Pompidou  
78180 Montigny-le-Bretonneux FRANCE  
**Phone:** +33 1 30 60 15 00  
**Fax:** +33 1 30 64 19 42

### Technical Support

Ansys France  
Parc Avenue, 9 rue Michel Labrousse  
31100 Toulouse FRANCE  
**Phone:** +33 5 34 60 90 50  
**Fax:** +33 5 34 60 90 41

Submit questions to **Ansys SCADE Products Technical Support** at [scade-support@ansys.com](mailto:scade-support@ansys.com).

Contact one of our **Sales representatives** at [scade-sales@ansys.com](mailto:scade-sales@ansys.com).

Direct general questions about SCADE products to [scade-info@ansys.com](mailto:scade-info@ansys.com).

Discover latest news on our products at [www.ansys.com/products/embedded-software](http://www.ansys.com/products/embedded-software).

## Legal Information

Copyrights © 2020 ANSYS, Inc. All rights reserved. Ansys, SCADE, SCADE Suite, SCADE Display, SCADE Architect, SCADE LifeCycle, SCADE Test, and Twin Builder are trademarks or registered trademarks of ANSYS, Inc. or its subsidiaries in the U.S. or other countries. Scade Language Reference Manual – Published May 2020.

All other trademarks and tradenames contained herein are the property of their respective owners.

## Terms of Use

**Important!** Read carefully before starting this software and referring to its user documentation. This publication, as well as the software it describes, is distributed as part of the SCADE user documentation under license and may be used or copied only in accordance with the terms and conditions of the Software License Agreement (SLA) accepted during SCADE product installation. Except as permitted by the SLA, the content of this publication cannot be reproduced, stored, or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior permission of Ansys. Any existing artwork or images that you may want to reuse in your projects may be protected under copyright law. The unauthorized use of such material into your own work may constitute a violation of Ansys copyrights. If needed, make sure you obtain the written permission from Ansys. By starting the software you expressly agree to the following conditions:

- **Property:** This software is and remains the exclusive property of Ansys. It cannot be copied or distributed without written authorization from Ansys. Ansys retains title and ownership of the software.
- **Warranty:** The software is provided, as is, without warranty of any kind. Ansys does not guarantee the use, or the results from the use, of this software. All risk associated with usage results and performance is assumed by you the user.

## Documentation Disclaimer

The content of SCADE products user documentation is distributed for informational use only, is subject to change without notice, and should not be construed as a commitment by Ansys. Although every precaution has been taken to prepare this manual, Ansys assumes no responsibility or liability for any errors that may be contained in this book or any damages resulting from the use of the information contained herein.

## Third-Party Legal Info

The Legal Notice (PDF) about third-party software can be found under the **help** folder of the SCADE installation.



# Language Reference Manual Overview

This manual describes all Scade language elements in details. It presents systematically the syntax and semantics of each element. Its content is mainly aimed at SCADE Suite users familiar with the language and looking for details about specific constructs. It is the ultimate reference for users who need to get advanced insights into Scade language.

- Chapter 1: [“Introduction”](#)
- Chapter 2: [“Lexical Elements”](#)
- Chapter 3: [“Program and Packages”](#)
- Chapter 4: [“Types and Groups”](#)
- Chapter 5: [“Global Flows”](#)
- Chapter 6: [“User-Defined Operators”](#)
- Chapter 7: [“Operator Bodies”](#)
- Chapter 8: [“Expressions”](#)

## Appendixes and Index:

- Appendix A: [“Formalization”](#)
- Appendix B: [“Backus-Naur-Form”](#)
- Appendix C: [“Mapping between Textual and Graphical Representations”](#)
- Appendix D: [“Bibliography”](#)

[“Index”](#)

## RELATED DOCUMENTS

- *SCADE Suite User Manual*
- *SCADE Suite Technical Manual*
- *Scade Language Primer*

## TYPOGRAPHICAL CONVENTIONS

<b>Courier New Bold</b>	Keywords of Scade language ( <i>e.g.</i> , <b>pre</b> , <b>node</b> )
Courrier New	Code examples or file extensions ( <i>e.g.</i> , <code>.xscade</code> )
<i>italics</i>	Names of models, variable elements ( <i>e.g.</i> , <i>newdescription</i> ), or object attributes ( <i>e.g.</i> , <i>B1</i> ),



# Table of Contents

---

## Language Reference Manual Overview

1. Introduction	1 - 1
2. Lexical Elements	2 - 3
2.1 Notations	2 - 3
2.2 Lexemes	2 - 4
2.3 Symbol List	2 - 4
2.4 Keyword List	2 - 5
2.5 Comments	2 - 5
2.6 Pragmas	2 - 5
3. Program and Packages	3 - 7
3.1 Program	3 - 7
3.2 Packages	3 - 8
3.3 Declarations	3 - 9
3.4 Attributes	3 - 12
4. Types and Groups	4 - 15
4.1 Type Declarations	4 - 16
4.2 Type Expressions	4 - 19
4.3 Group Declarations	4 - 22
4.4 Group Expressions	4 - 23
5. Global Flows	5 - 25
5.1 Constants	5 - 25
5.2 Sensors	5 - 27
6. User-Defined Operators	6 - 29
6.1 User-Defined Operators	6 - 30

# Table of Contents

---

6.2	Variable Declarations	6 - 36
6.3	Clock Expressions	6 - 39
6.4	Scope Declarations	6 - 41
7.	Operator Bodies	7 - 43
7.1	Equations	7 - 44
7.2	Conditional Blocks	7 - 48
7.3	State Machines	7 - 53
7.3.1	State Machines	7 - 53
7.3.2	Transitions	7 - 58
7.3.3	Actions	7 - 60
7.3.4	Examples	7 - 61
8.	Expressions	8 - 71
8.1	Basic Expressions	8 - 72
8.1.1	Identifiers	8 - 72
8.1.2	Atoms	8 - 75
8.1.3	Lists	8 - 76
8.2	Sequential Operators	8 - 78
8.3	Combinatorial Operators	8 - 83
8.3.1	Boolean Operators	8 - 83
8.3.2	Arithmetic Operators	8 - 84
8.3.3	Flows Switches	8 - 88
8.4	Operations on Arrays and Structures	8 - 90
8.4.1	Array Expressions	8 - 90
8.4.2	Structure Expressions	8 - 95
8.4.3	Mixed Constructor	8 - 96
8.4.4	Labels and Indexes	8 - 97



# Table of Contents

---

8.5	Operator Application and Higher-Order Patterns	8 - 98
8.6	Primitive Operator Associativity and Relative Priority	8 - 119
<b>Appendixes</b>		<b>121</b>
<b>A.</b>	<b>Formalization</b>	<b>A - 123</b>
A.1	Namespace Analysis	A - 123
A.1.1	Purpose	A - 123
A.1.2	Principles	A - 123
A.2	Type Analysis	A - 126
A.2.1	Purpose	A - 126
A.2.2	Precondition	A - 126
A.2.3	Principles	A - 126
A.3	Clock Analysis	A - 128
A.3.1	Purpose	A - 128
A.3.2	Precondition	A - 129
A.3.3	Principles	A - 129
A.4	Causality Analysis	A - 131
A.4.1	Purpose	A - 131
A.4.2	Precondition	A - 131
A.4.3	Principles	A - 131
A.5	Initialization Analysis	A - 133
A.5.1	Purpose	A - 133
A.5.2	Precondition	A - 133
A.5.3	Principles	A - 134
<b>B.</b>	<b>Backus-Naur-Form</b>	<b>B - 137</b>
B.1	Declarations	B - 137
B.2	User-Defined Operators	B - 139
B.3	Expressions	B - 141

# Table of Contents

---

<b>C. Mapping between Textual and Graphical Representations</b>	<b>C - 145</b>
<b>C.1 General Syntax</b>	<b>C - 146</b>
<b>C.2 Arithmetic Operators</b>	<b>C - 147</b>
C.2.1 Operator +	C - 147
C.2.2 Operator - (binary minus)	C - 147
C.2.3 Operator - (unary minus)	C - 147
C.2.4 Operator *	C - 148
C.2.5 Operator / (polymorphic division)	C - 148
C.2.6 Operator mod	C - 148
<b>C.3 Bitwise Arithmetic Operators</b>	<b>C - 149</b>
C.3.1 Operator land	C - 149
C.3.2 Operator lnot	C - 149
C.3.3 Operator lor	C - 149
C.3.4 Operator lsl (logical left shift)	C - 150
C.3.5 Operator lsr (logical right shift)	C - 150
C.3.6 Operator lxor	C - 150
<b>C.4 Conversion Operators</b>	<b>C - 150</b>
C.4.1 Operator numeric cast	C - 150
<b>C.5 Comparison Operators</b>	<b>C - 151</b>
C.5.1 Operator <	C - 151
C.5.2 Operator <=	C - 151
C.5.3 Operator >	C - 151
C.5.4 Operator >=	C - 151
C.5.5 Operator <>	C - 151
C.5.6 Operator =	C - 152
<b>C.6 Logical Operators</b>	<b>C - 152</b>
C.6.1 Operator and	C - 152
C.6.2 Operator or	C - 152
C.6.3 Operator xor	C - 153
C.6.4 Operator #	C - 153

# Table of Contents

---

C.6.5	Operator not	C - 153
<b>C.7</b>	<b>Temporal Operators</b>	<b>C - 154</b>
C.7.1	Operator ->	C - 154
C.7.2	Operator when	C - 154
C.7.3	Operator pre	C - 155
C.7.4	Operator fby	C - 155
C.7.5	Operator merge	C - 155
C.7.6	Operator times	C - 155
<b>C.8</b>	<b>Choice Operators</b>	<b>C - 156</b>
C.8.1	Operator if ... then ... else ...	C - 156
C.8.2	Operator case ... of ...	C - 157
<b>C.9</b>	<b>Structure and Array Operators</b>	<b>C - 158</b>
C.9.1	Data Structure Constructor: Value Enumeration	C - 158
C.9.2	Array Constructor: Value Enumeration	C - 158
C.9.3	Array Constructor: Value Repetition	C - 158
C.9.4	Array Access: Static Slice	C - 159
C.9.5	Array Concatenation	C - 159
C.9.6	Array Reverse	C - 159
C.9.7	Array Transpose	C - 159
C.9.8	Structure and Array Access: Static Indexation	C - 160
C.9.9	Array Access: Dynamic Indexation	C - 160
C.9.10	Structure and Array Constructor: Array Copy with Modification	C - 161
C.9.11	Make and Flatten	C - 161
<b>C.10</b>	<b>Higher-Order Operators</b>	<b>C - 162</b>
C.10.1	Iterators: map, fold, and mapfold	C - 162
C.10.2	Iterator with Access to Index: mapi, foldi, mapfoldi	C - 163
C.10.3	Partial Iterators: mapw, mapwi, foldw, foldwi, mapfoldw, mapfoldwi	C - 164
C.10.4	Resettable Node Instantiation	C - 165
C.10.5	Conditional Activation of Node Instantiation with Default Values	C - 166
C.10.6	Conditional Activation of Node Instantiation on Clock	C - 166
<b>C.11</b>	<b>Other Design Elements</b>	<b>C - 167</b>

# Table of Contents

---

C.11.1	Operator Calls	C - 167
C.11.2	Operator Call from Library	C - 167
C.11.3	Imported Operator Calls	C - 168
C.11.4	Parameterized Operator Calls	C - 168
C.11.5	Input Reference	C - 168
C.11.6	Output Reference	C - 168
C.11.7	Consumed Local Variable Reference	C - 168
C.11.8	Produced Local Variable Reference	C - 168
C.11.9	Probe Reference	C - 169
C.11.10	Constant Reference	C - 169
C.11.11	Textual Expression Reference	C - 169
C.11.12	Terminator Reference	C - 169
C.11.13	Assume	C - 169
C.11.14	Guarantee	C - 169
<b>C.12</b>	<b>State Machines</b>	<b>C - 170</b>
C.12.1	State Machine	C - 170
C.12.2	States	C - 170
C.12.3	Transition	C - 171
C.12.4	Last Value, Signal Emission and Test	C - 173
<b>C.13</b>	<b>Conditionnal Blocks</b>	<b>C - 174</b>
C.13.1	Operator If Block	C - 174
C.13.2	Operator When Block	C - 174
<b>D.</b>	<b>Bibliography</b>	<b>D - 175</b>
<b>INDEX</b>		<b>177</b>

# Introduction

The *Scade Language Reference Manual* is intended for experienced readers who need detailed information on a particular construct of the Scade language, after having browsed the informal presentation of *Scade Language Primer*.

The manual is organized as follows:

- [Chapter 2](#) presents the lexical elements, also called lexemes, that are used in the syntax description of the language constructs. The syntax description is given in a Bachus-Naur style using the notations described in section 2.1 [“Notations”](#). In addition to numerical and string based lexemes, keywords are listed in order to prevent their use by other identifiers, as some special characters and combination of characters. Comments and pragmas, which are uninterpreted parts of a program’s text, are not concerned by these restrictions.
- [Chapter 3](#) introduces the top-level constructs of the language. It focuses on the package structure of a model, which allows to manage the namespace analysis only on local part of the model. The constructs detailed here are the declaration and use of packages, including the attribute declarations that are also used by types and constants.
- [Chapter 4](#) details the type system used in Scade. It also presents the constructs allowed to extend this system by user declared type identifiers. These identifiers can refer to plain or complex types (arrays and structures). Groups are a syntactic convenience that allows to collect and refer in an unique way the multiple outputs of a user defined operator. They can be thought of as lists of types, even though a group is not fully considered as a type itself. The constructs used to declare and use groups are described in this section.
- [Chapter 5](#) describes another class of top-level declarations: constants and sensors. These are special flows that can be manipulated in any user operator. Constants are flows that keep their value throughout program executions. Sensors are input flows that implicitly extends (when needed) the user operator signatures.

- [Chapter 6](#) presents the basic structuring construct of a Scade model: user defined operators. This section emphasizes on the signature declaration of user operators (variables and clocks declarations), and on the modularity of the static analyzes. Modularity means that every analysis is performed at the user operator level, producing a corresponding type that can be reused when this operator is called. The extension of the signature part by local identifiers within user operators naturally ends this section.
- [Chapter 7](#) details the three possible means to define the equation of the output and local variables in a user operator: standard equations, conditional blocks, and State Machines. Signals are a special kind of local variables that also require a definition. This definition does not use any of these three constructs, but a special adapted case called signal emission. Finally, assert is used to implement invariants of the model. They are not given any dynamic semantics, but can be used by proof tools.
- [Chapter 8](#) gathers the operators provided by the language to combine flows: sequential, combinatorial, iterators applied to complex flows, and higher level constructs that apply on user operator calls. The relative priority of these operators is given at the end of this section.

# 2

## Lexical Elements

This chapter presents the lexical elements, also called lexemes, that are used in the syntax description of the language constructs, the list of symbols and keywords reserved to prevent their use by other identifiers, and the list of comments and pragmas which are uninterpreted parts of a program's text.

- 2.1 [“Notations”](#)
- 2.2 [“Lexemes”](#)
- 2.3 [“Symbol List”](#)
- 2.4 [“Keyword List”](#)
- 2.5 [“Comments”](#)
- 2.6 [“Pragmas”](#)

### 2.1 Notations

Regular Expressions Notations		Extended Backus-Naur-Form Notations	
Notation	Meaning	Notation	Meaning
<code>ID = re</code>	ID is defined by regular expression re	<code>::=</code>	is defined as
<code> </code>	alternative	<code> </code>	alternative
<code>re?</code>	0 or 1 occurrence of re	<code>[[x]]</code>	0 or 1 occurrence of X
<code>re*</code>	0 occurrence of re or more	<code>{{x}}</code>	0 occurrence of X or more
<code>re+</code>	1 occurrence of X or more	<code>{{x}}+</code>	1 occurrence of X or more
<code>(re)</code>	grouping	<code>((X   Y))</code>	grouping: either X, or Y
<code>[chars]</code>	matches any of chars	<code>abc</code>	keyword terminal symbol abc
<code>[x-y]</code>	set of chars, ranging from x to y	<code>INTEGER</code>	terminal symbol other than a keyword
		<code>xyz</code>	non-terminal symbol

## 2.2 Lexemes

Scade recognizes several notations to describe values. Integers can be written as binary, octal, decimal, or hexadecimal values and mixed within the same expression; they can be either typed by a suffix or untyped. Non decimal integer literals are considered unsigned (*i.e.*, their value is always positive). Float values can be described using a decimal or a scientific notation; they can be either typed by a suffix or untyped. Character values are described using a standard ASCII alphabet.

```

DIGIT2 = [0-1]
DIGIT8 = [0-7]
DIGIT10 = [0-9]
DIGIT16 = [a-f] | [A-F] | [0-9]

INTEGER2 = 0b DIGIT2+
INTEGER8 = 0 DIGIT8+
INTEGER10 = 0 | ([1-9] DIGIT10*)
INTEGER16 = 0x DIGIT16+
INTEGER = INTEGER2
        | INTEGER8
        | INTEGER10
        | INTEGER16

TYPED_INTEGER = INTEGER (_i | _ui) (8 | 16 | 32 | 64)

EXPONENT = [eE] [+~]? DIGIT10+
FLOAT = DIGIT10+ . DIGIT10* [EXPONENT]?
        | DIGIT10* . DIGIT10+ [EXPONENT]?

TYPED_FLOAT = FLOAT (_f32 | _f64)

LETTER = _ | [a-z] | [A-Z]
ALPHANUMERIC = DIGIT10 | LETTER
WORD = LETTER ALPHANUMERIC*
CHARACTER = ALPHANUMERIC | space
           | [!"#$%&'()*+,-./:;<=>?@[\^`{|}~]

CHAR = 'CHARACTER' | '\x digit16 digit16'
ID = WORD
NAME = 'WORD'

```

## 2.3 Symbol List

The following symbols are recognized as lexemes:

```

<>   <=   >=   <<   >>   =   <   >
( )   [ ]   { }
->   ::   ..   ;   ,   .   :   ^   @   #   _   |   +   -   *   /

::= $+$ | $-$ | $*$ | $/$ | $mod$
    | $=$ | $<>$ | $<$ | $>$ | $<=$ | $>=$
PREFIXOP | $@$ | $times$ | $and$ | $or$ | $xor$
          | $land$ | $lor$ | $lxor$ | $lsl$ | $lsr$
          | +$ | -$ | not$ | lnot$ | reverse$

```

The following symbols are ignored:

```

space(' '), tabulation('\t'), carriage return('\r'), line feed('\n'), form feed('\f')

```



## 2.4 Keyword List

Scade language keywords for SCADE Suite KCG:

```
abstract, activate, and, assume, automaton
bool
case, char, clock, const
default, do
else, elsif, emit, end, enum, every
false, fby, final, flatten, float, float32, float64, fold, foldi, foldw, foldwi, function
guarantee, group
if, imported, initial, int8, int16, int32, int64, integer, is
land, last, let, lnot, lor, lsl, lsr, lxor
make, map, mapfold, mapfoldi, mapfoldw, mapfoldwi, mapi, mapw, mapwi, match, merge, mod
node, not, numeric
of, onreset, open, or
package, parameter, pre, private, probe, public
restart, resume, returns, reverse
sensor, sig, signed, specialize, state, synchro
tel, then, times, transpose, true, type
uint8, uint16, uint32, uint64, unless, unsigned, until
var
when, where, with
xor
```

Notice that the keywords **onreset**, **abstract**, **parameter** are reserved but unused.

## 2.5 Comments

There are two different syntaxes to write a comment in a plain text design:

- single line comment: starting from `--` and ending at end of the line
- multi-line comment: starting from `/*` and ending with `*/`

## 2.6 Pragmas

Pragmas allow to pass information to tools without using the language semantics. The concept of pragma belongs to the language, but not the pragmas themselves. They have the following forms:

```
pragma ::= #pragma character* #end
        | #alphanumeric+
```

The first form contains any kind of character and can be multi-line. Inside such pragmas, a doubled '#' character ('##') is interpreted as a single '#' character. This allows to write "##**end**", which does not close the pragma and results in the sequence "**end**" in the pragma. Note a single '#' character is interpreted as itself. The syntax accepted and/or recognized for pragmas is detailed by the tools that define them.



# 3

## Program and Packages

This chapter presents the top-level constructs of the language by focusing on the program, the packages, and their declarations or attribute declarations.

- 3.1 [“Program”](#)
- 3.2 [“Packages”](#)
- 3.3 [“Declarations”](#)
- 3.4 [“Attributes”](#)

### 3.1 Program

#### SYNTAX

---

```
program ::= {{ decls }}
```

---

#### STATIC SEMANTICS

A program or a package body consists in a list of declaration blocks. The order of these declarations is not relevant, according to the declarative flavor of Scade programs.

#### DYNAMIC SEMANTICS

The semantics of the language is defined at the level of user operators. The user defined operator which defines the semantics of a program is called the *root* operator.

## 3.2 Packages

The *package* (or *module* or *namespace*) mechanism is a software engineering feature provided by any programming language. It allows to design a software as a bundle of disjoint blocks. Furthermore, it makes the design and the usage of libraries easier.

### SYNTAX

```

path ::= ID {{ :: ID }}
path_id ::= [[ path :: ]] ID
package_decl ::= package [[ visibility ]] ID
                {{ decls }} end ;

```

### STATIC SEMANTICS

- A `path` is denoted by a list of identifiers separated by two colons:  $Id_1 :: Id_2 :: \dots :: Id_n$ . This path is valid if every identifier refers to a package name, and if the package  $Id_n$  is declared into the package  $Id_{n-1}$ , and so on. Resolving a path consists in searching for an occurrence of  $Id_1$  in the subpackages of the current context. If  $Id_1$  does not belong to this package list, then it is searched in the subpackages of the father context or in the father of the father context, until it is found. Once this package name is found, the algorithm searches  $Id_2$  in  $Id_1$  subpackages, then  $Id_3$  in  $Id_2$  subpackages, and so on.
- A `path_id` is a (possibly empty) path followed by an identifier referring to a declaration different from a package. The path resolution above is valid if this declaration indeed appears in the last package of the path.
- A package must have an identifier different from the packages declared at the same level. Within a package, all the declarations must have different names. A package can be given a **private** visibility status. A private package is only visible to its direct super package. It can be opened, or its declarations being accessed through a path, in this package. This package is undefined higher in the package hierarchy.

### RELATED TOPICS

- Section A-1 [“Namespace Analysis”](#)
- Section 3.4 [“Attributes”](#)

## 3.3 Declarations

### SYNTAX

Top-level declarations can be of seven kinds:

```
decls ::= open path ;  
      | package_decl  
      | group_block  
      | type_block  
      | const_block  
      | sensor_block  
      | user_op_decl
```

### STATIC SEMANTICS

- The **open** directive requires a valid *path* referring to a package name. Paths are described in section 3.2 [“Packages”](#). Opening a package leads to extend the current declaration environment with all the public top-level declarations of the opened package. Opening a package is allowed if all the public declarations contained in this package do not share the same name with declarations of the current context. This is true except for subpackages declarations: A subpackage of an opened package can have the same name as another package of the current context. The package declaration environment is not extended by the **open** directive. Moreover, the declarations within a subpackage of an opened package are not accessible by a call to this package, even though this subpackage was opened in its father package. A package cannot be opened more than once, even using different paths. A package cannot open itself.
- Package declarations are defined in section 3.2 [“Packages”](#). Groups and types are described in Chapter 4 about [“Types and Groups”](#), while constants and sensors are in Chapter 5 about [“Global Flows”](#). User defined operators are defined in [“User-Defined Operators”](#) on page 30.

A declaration can depend on another declaration block, but cyclic dependencies must be avoided.

### Example 1: Declaration using a fully qualified path:

---

```
package P1
  const foo: int16 = 3;
end;

package P2
  const bar: int16 = 4;
end;

const foobar: int16 = P1::foo + P2::bar;
```

---

### Example 2: Resolution algorithm. The `open` directive will include the declarations of the package $Q : R$ not that of $P : R$ , since this is the first occurrence of $R$ in the package structure starting from $S$ .

---

```
package P
  package R
  ...
end;

package Q
  package R
  ...
end;
package S
  open R;
end;

end;
```

---

### Example 3: The order in package declaration is not relevant:

---

```
package P1
  open P1;
  const
    iC1 : T1 = 2;

package P1
  type
    T1 = int32;
end;

end;
```

---

---

**Example 4:** Errors due to multiple declaration via `open`. Identifier `foo` used as a constant clashes with the type name `foo` introduced by the `open` directive.

---

```
package P1
  type foo = int16;
end;
open P1;
const foo: bool = true;
```

---

**Example 5:** Subpackages belonging to an opened package are not considered by the `open` directive. The second declaration of package `P2` is valid in this program, because `P2` of `P1` is not visible.

---

```
package P1
  package P2
    ...
  end;
end;
open P1;
package P2
  ...
end;
```

---

**Example 6:** Package opening is not transitive. Subpackage `P2` opened in the declaration of package `P1` below is not considered while opening this latter package. Constant `foo` is then unknown in the definition of `foobar`, raising an error:

---

```
package P1
  package P2
    const foo: int32 = 0;
  end;
  open P2;
  const bar: int32 = foo + 1;
end;
open P1;
const foobar: int32 = foo * 2;
```

---

## RELATED TOPICS

- Section A-1 [“Namespace Analysis”](#)
- Section 3.2 [“Packages”](#) about paths

## 3.4 Attributes

Some declarations may be given attributes for visibility and usability purposes.

### SYNTAX

---

```
visibility ::= private
           | public

external ::= imported

interface_status ::= [[ visibility ]] [[ external ]]
```

---

### STATIC SEMANTICS

- Within a package (or the global program), types, groups, constants, user operators, can be hidden to restrict their direct usage outside the package. Visibility rules are:
  - **public**: is the default case; the declaration can be used anywhere.
  - **private**: is the most restrictive case; the declaration can be used inside the package and its subpackages. Private declarations occurring in an opened package do not interact with declarations in the current package: no name clash can happen.
- Scade is not a general purpose language. A Scade program is intended to be embedded into a more general program, which is in charge, in particular, of sampling inputs and displaying outputs of the Scade program. The embedding program is generally written in another language, called the *host* language. Some objects must be shared between the Scade program and the embedding program. Moreover, some objects, like complex data types or functions, are easier to describe in a general purpose host language. This is why Scade allows types, constants, and user operators to be imported from the host language.

A declaration can be declared as imported by prefixing it by the keyword **imported**. In this case, the declaration has no definition: a type is merely given a name, a constant merely its type, and a user operator merely its interface.

By default, imported declarations are given a public visibility status. They can be however be given a private visibility status as any other local declarations.



---

**Example 1:** Private visibility status: type `t` is unknown outside `P1`. This program raises an error.

---

```
package P1
  type private t= int64;
  const c:t=0;
end;
open P1;
const d:t= c + 1;
```

---

**Example 2:** Imported declarations:

---

```
package ForeignArithmetics
  type imported ForeignCplx ;
  function imported plusCplx(x,y: ForeignCplx) returns (z: ForeignCplx);
  function twiceCplx (x: ForeignCplx) returns (y: ForeignInt) y = plusCplx (x, x);
end;
```

---

#### RELATED TOPICS

- Section A-2 [“Type Analysis”](#)
- Section 3.2 [“Packages”](#)
- Section 4.1 [“Type Declarations”](#)
- Section 5.1 [“Constants”](#)
- Chapter 4 about [“Types and Groups”](#)
- Chapter 6 about [“User-Defined Operators”](#)



# 4

## Types and Groups

Scade is a strongly typed language, meaning that every object is associated with a determined data type. Constant and variable declarations in the scope of a user operators must be given a type. The native type system is built out of atomic data type, arrays, structures, and groups.

Genericity is provided by means of imported data types (instantiated by a data type of the host language), and type variables (at the operator level). This system can be extended by user defined type identifiers. These declarations can be declared anywhere at the top-level of a package or a program.

- 4.1 [“Type Declarations”](#)
- 4.2 [“Type Expressions”](#)
- 4.3 [“Group Declarations”](#)
- 4.4 [“Group Expressions”](#)

---

### Note

Groups are not supported as graphical design objects in SCADE Suite IDE.

---

## 4.1 Type Declarations

Type declarations are used to build new user's data types.

### SYNTAX

```

type_block ::= type {{ type_decl; }}
type_decl ::= interface_status ID [[ = type_def ]] [[ is numeric_kind ]]
type_def  ::= type_expr
            | enum { ID {{ , ID }} }
numeric_kind ::= numeric | float | integer | signed | unsigned

```

### STATIC SEMANTICS

A type block starts with the keyword **type**. Several types can be declared using this keyword only once. A type declaration is composed of an identifier different from any other declaration in the same scope, possibly followed by a definition part, made of the = symbol and a type expression, and a numeric kind specifier, made of the keyword **is** and a numeric kind name. This identifier can be used within the current package and all its sub-packages where a type name is required. Keywords of the language cannot be used as a type identifier. Every type declaration must end with a semicolon.

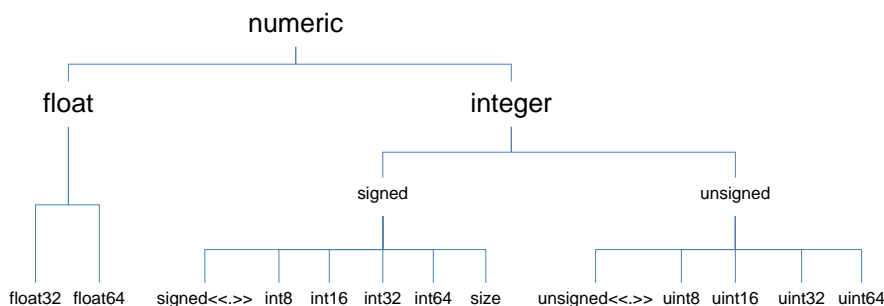
Attributes can be given to this declaration. An imported type declaration cannot have a definition part. On the contrary, a non-imported type declaration must have a definition part but cannot have a numeric kind. Non-numeric imported types can be used to declare constants and variables. They can only be manipulated through user-defined operators, imported operators, and polymorphic operators (see table below).

Table 4.1: Polymorphic operators

Temporal	<code>last, pre, -&gt;, fby, when, merge</code>
Control	<code>if, case</code>
Structure	<code>^, [], {}, make, flatten</code>
Higher Order	<code>map, fold, mapfold, mapi, foldi, mapfoldi, mapw, foldw, mapfoldw, mapwi, foldwi, mapfoldwi activate.every, activate.every.default., activate.every.initial default.</code>

Numeric kinds are used to group numeric types. The **numeric** kind distinguishes **float** types from **integer** types, which are split between **signed** and **unsigned** types as detailed in [Figure 4.1](#). The kinds are such that  $\text{float} \subseteq \text{numeric}$  and  $\text{signed}, \text{unsigned} \subseteq \text{integer} \subseteq \text{numeric}$ . It means that a type of kind **signed** can be used anywhere an **integer** or **numeric** type is expected.

An imported type declared with a numeric kind can be used just like a predefined type of the same kind. Literals and arithmetic operators can also be used with such type. For instance, in the case of an imported type declared as **unsigned**, then integer literals, arithmetic, shift and bitwise operators can be used. In the case of a **float** type, decimal integer literals, float literals and arithmetic operators can be used. If the type is declared as **numeric**, then only decimal integer literals and arithmetic operators can be used for this type (see [8.1.2 “Atoms”](#)).



**Figure 4.1:** Hierarchy of predefined numeric types

The type expression used in a type declaration must not contain any type variable (identifier preceded with a quote). Valid type expressions are detailed in [4.2](#). All the identifiers occurring in an enumeration type belong to the package environment as if they were constants. They must therefore comply to the namespace policy.

**Example 1:** Correct declarations:

---

```

type T = int32;
type V = T^2;

```

---

### Example 2: Imported numeric type:

---

```
package ForeignArithmetics
type imported ForeignInt is integer ;
function plus_two (x: ForeignInt ) returns (y: ForeignInt ) y = x + 2;
end;
```

---

### Example 3: Error due to attributes:

---

```
type imported LongInt = bool^53;
type Speed;
```

---

### Example 4: Error due to type variables:

---

```
type Generic = 'T;
```

---

### Example 5: Error due to enumeration identifiers: the identifier Blue cannot be used as a constant and an object of an enumeration.

---

```
type Color = enum {Blue, Red, Yellow};
const Blue: int32 = 0;
```

---

#### RELATED TOPICS

- Section A-2 [“Type Analysis”](#)
- Section 3.4 [“Attributes”](#).

## 4.2 Type Expressions

Type expressions may appear in types, constants, and variables declarations.

### SYNTAX

---

```

type_expr ::= bool
           | signed << expr >> | int8 | int16 | int32 | int64
           | unsigned << expr >> | uint8 | uint16 | uint32 | uint64
           | float32|float64
           | char
           | path_id
           | typevar
           | { field_decl {{ , field_decl }} }
           | type_expr ^ expr
field_decl ::= ID : type_expr
typevar    ::= NAME

```

---

### STATIC SEMANTICS

The expression appearing after the **signed** or **unsigned** keywords and after the **^** symbol is an integer constant expression that must be known at compile time. This expression, also called *size expression*, can use:

- Integer values (except typed literals)
- Non-imported constants
- Size parameters of user operators
- Arithmetic operators (unary + and -, binary + and -, \*, /, mod, land, lor, lxor, lsl, lsr)
- Static array and structure projections

Integers in size expressions are given the **size** type. More generally, any value of an integer type is considered as a value of **size** type in a size expression. This implicit cast ensures that static evaluation is done in the **size** type. The **size** type is an internal type that cannot appear in type expressions. It is also used in the generated code, in particular for loop indices.

Predefined types are Boolean, characters, integer, and float types:

- **signed<<n>>** denotes signed integers stored in  $n$  bits, that belong to the  $[-2^{n-1}..2^{n-1} - 1]$  range. We have  $\text{signed}\langle\langle n \rangle\rangle \subseteq \text{signed} \subseteq \text{integer} \subseteq \text{numeric}$ . The value denoted by the size expression  $n$  must be equal to 8, 16, 32, or 64. **intN** is an alias for **signed<<N>>**.
- **unsigned<<n>>** denotes unsigned integers stored in  $n$  bits, that belong to the  $[0..2^n - 1]$  range. We have  $\text{unsigned}\langle\langle n \rangle\rangle \subseteq \text{unsigned} \subseteq \text{integer} \subseteq \text{numeric}$ . The value denoted by the size expression  $n$  must be equal to 8, 16, 32, or 64. **uintN** is an alias for **unsigned<<N>>**.
- **floatN** denotes floating-points numbers stored in  $N$  bits. **float32** corresponds to single-precision floating-point numbers, and **float64** corresponds to double-precision floating-point numbers where  $\text{floatN} \subseteq \text{float} \subseteq \text{numeric}$ .

A type identifier, possibly qualified by a valid path, may be used as a type expression. This type identifier, denoted by `path_id`, must be declared in the corresponding path as a valid type name.

A type variable `typevar` is an identifier preceded by a quote (such as 'T'). It allows to describe generic operators that do not depend on the data type of their input variables. It may be used as a type expression in the profile of a user defined operator, either local or external. It cannot be used in a type declaration. An object belonging to a variable type can only be manipulated by user defined operators acting on a profile containing type variables or by polymorphic operators as for external data types (see [Table 4.1](#)). Restrictions on type variables can be stated through **where** declaration (see section [6.1](#)), enforcing objects of such type to be processed only by arithmetical expressions.

Structured types allow the type hierarchy to be decomposed. A structure is given by a non-empty list of field identifiers separated by commas and surrounded by curly braces. The order of the fields is relevant when considering type equivalence: two structure types are equivalent if they contain the same labels in the same order. Each field is given a label, which must be different from other labels of the current type, and a type. Fields identifiers are not in the current naming environment: they can share the name of any other object, except keywords of the language. The type of a field can be any valid type expression, including other structured types. Structures can thus be nested, but they cannot be recursively defined.

Array types are a special kind of structure in which the fields all share the same data type (which can be any type expression). Fields in an array are named by incremented integers, starting from 0. Specific operators are provided to deal with array values. Array data types are declared using any type expression followed by the  $\wedge$  symbol and an integer constant expression that must be known at compile time. The value denoted by this size expression must be strictly positive.



## DYNAMIC SEMANTICS

Operations on arrays or structures do not modify in place the arrays nor the structures. Instead, such operation produce new values. It is the responsibility of the implementation tool to allow in place operations while respecting the data-flow semantics.

**Example 1:** Invalid type expression for the declaration of type `t`: A constant identifier cannot be reused as a type expression.

---

```
const c: int32 = 2;
type t = c;
```

---

**Example 2:** When type variables used in too restricted a context, an error is raised. The `mod` operator requires an integer, but variable `a` has type `'T`, which is only a numeric.

---

```
function incorrect (a:'T) returns (b:'T) where 'T numeric
  b = a mod 3;
```

---

It is then necessary to strengthen the constraint on type `'T`:

---

```
function correct (a:'T) returns (b:'T) where 'T integer
  b = a mod 3;
```

---

**Example 3:** Structured type:

---

```
type t = {lbl1: int32, lbl2: bool, lbl3:{lbl4: int32 , lbl5: float32}};
const c: t = {lbl1: 1, lbl2: true, lbl3:{lbl4 :2, lbl5 :0.5}};
const d: int32 = c.lbl1;
```

---

**Example 4:** Array type:

---

```
type t = int32;
const wordsize : int16 = 8;
type byte = t ^ wordsize;
```

---

## RELATED TOPICS

- Section A-1 [“Namespace Analysis”](#)
- Section A-2 [“Type Analysis”](#)
- Section 5.1 [“Constants”](#)
- Section 8.4 [“Operations on Arrays and Structures”](#)
- Section 8.1.2 [“Atoms”](#) about literals
- Chapter 6 about [“User-Defined Operators”](#)

## 4.3 Group Declarations

Groups are special kind of data type. They allow to manage user operators with several outputs without gathering these outputs in a structured type (array or structure). The expressions in which such operators may occur are then lighter. There are two means to build group objects, either by a call to a user operator or by using the list operator.

### SYNTAX

---

```
group_block ::= group {{ group_decl ; }}
group_decl ::= [[ visibility ]] ID = group_expr
```

---

### STATIC SEMANTICS

A group block starts with the keyword **group**. Several groups can be declared using this keyword only once. A group declaration is composed of an identifier different from any other declaration in the same scope, followed by the = symbol and a group expression. This identifier can be used within the current package and all its sub-packages where a type name is required. Keywords of the language cannot be used as a group identifier. Every group declaration must be ended by a semicolon.

Groups are always considered as a flat list of data types: nested groups are equivalent to their flattened version. Conversely, a list of flows can be replaced by an object belonging to the corresponding group type. The only constraint required on this list is that all the flows it contains are based on the same clock. Groups can be given a visibility status, but they cannot be imported nor taken as parameters. A group can be used as a type in group or variable declarations but cannot be used as a type in a constant, sensor, or type declaration. Groups can only be manipulated through user defined operators and a limited range of polymorphic operators, as in table below.

Table 4.2: Operators available on groups

Temporal	<b>pre</b> , <b>-&gt;</b> , <b>fby</b> , <b>when</b> , <b>merge</b>
Control	<b>if</b> (not for the condition argument), <b>case</b> (not for the condition argument), <b>#</b>

### RELATED TOPICS

- Section 3.4 [“Attributes”](#)
- Section 8.1.3 [“Lists”](#)
- Chapter 6 about [“User-Defined Operators”](#).

## 4.4 Group Expressions

### SYNTAX

---

```
group_expr ::= ( type_expr {{ , type_expr }} )
```

---

### STATIC SEMANTICS

A group expression consists in a non-empty list of type expressions. Any type expression, except type variables, can be used in a group expression. A valid group identifier may be used as a group expression. An object belonging to a group, or a corresponding list of flows, only adds a clock constraint on these flows:

- Its type is the product of the types of the flows.
- Its clock is the product of the same clock type for all the flows.
- No dependence is introduced between the flows by their grouping.
- Its initialization is the product of its components initialization types.

### DYNAMIC SEMANTICS

Collecting flows in a group does not have any dynamic consequence: it does not define a new flow, but is simply a syntactic feature.

**Example 1:** Group declarations:

---

```
group
  G1 = (int32, bool);
  G2 = (bool, G1);
```

---

**Example 2:** A group cannot be used to define a type nor a sensor. The definitions below are incorrect:

---

```
group
  G = ...
type
  T1 = G^n;
sensor
  S: G;
```

---

### Example 3: A variable identifier can be declared with type G:

---

```
group G = (int32, int16, bool);  
node ex(e: G; ...) returns (s: int32; ...)  
...  
tel;
```

---

### Example 4: Groups are always flattened:

---

```
group  
  G0 = (int32 , bool);  
  G1 = (int32 , G0);  
function fl(x: int32; y: int32; z: bool) returns (r: G1)  
  r = (x, y, z);  
function f2(v: G1) returns (a: int32; b: int32; c: bool)  
  a, b, c = fl(v);
```

---

### Example 5: Groups are used to manage multiple outputs operators:

---

```
function f(a, b: int32; c: bool) returns (x,y: int32)  
  
function g(a,b: int32) returns (x: int32)  
  
node ex(c: bool; a, b: int32; ...) returns (s, t: int32; ...)  
let  
  s, t = if c then f(a, b, c) else (0, g(f(b, a, not c)));  
...  
tel;
```

---

### Example 6: A list expression can contain flows based on different clocks:

---

```
function N(x: int32; clock h: bool) returns (y: int32 when h; z: bool)  
let  
  y,z = (x when h, true);  
tel;
```

---

While an object of a group type cannot: an error is raised in this latter example.

---

```
group G = (int32, bool);  
function N(x: int32; clock h: bool) returns (y: G)  
let  
  y = (x when h, true);  
tel;
```

---

# 5

## Global Flows

A global flow is a flow that is available in any user operator of the current context. This flow can be a constant (keyword **const**) or variable (keyword **sensor**).

- 5.1 [“Constants”](#)
- 5.2 [“Sensors”](#)

### 5.1 Constants

#### SYNTAX

---

```
const_block ::= const {{ const_decl ; }}  
const_decl ::= interface_status ID : type_expr [[ = expr ]]
```

---

#### STATIC SEMANTICS

A constant block starts with the keyword **const**. Several constants can be declared using this keyword only once. A constant declaration is composed of an identifier different from any other identifier in the same scope, followed by a colon and the type of this identifier and possibly by a definition part: = symbol and an expression. This identifier can be used within the current package and all its sub-packages where an expression is required. Keywords of the language cannot be used as a constant identifier. Every constant declaration must end by a semicolon.

Attributes can be given to this declaration. An imported constant declaration must be given a type and cannot have a definition part. On the contrary, a non-imported type declaration must have a type and a definition part. Expressions allowed in the definition part of a constant declaration must be statically evaluable at compile time. Therefore, they can only be composed of:

- Values and enumeration values
- Non-imported constant identifiers
- Relational operators applied to constant expressions having a predefined type. Array and structure comparisons are not allowed.
- Boolean expressions, but #
- Arithmetic expressions

- Array constructors (extensional and exponentiation)
- Structure constructor (extensional)
- Constant array or structure projection

Temporal operators, dynamic projections, references to sensor identifiers or imported constants, calls to user-defined operators are not allowed. A constant declaration is well-typed if its declared type matches the type of its definition expression and the type information present in the typing environment. An imported constant can be based on any kind of type (except type variables). A non-imported constant can only be based on a non- imported type or an imported numeric type. In any case, a constant cannot be declared with a group type. A non-imported constant of integer type can be used as a size expression when manipulating parameterized arrays.

An identifier declared as constant must be available at each instant in every node. Therefore, the clock of a constant is the clock of the context it is used in. A constant must always be well-defined at the initial cycle.

### DYNAMIC SEMANTICS

A constant is always available and has the same value throughout program execution.

**Example 1:** A constant of an imported type cannot have a definition part

---

```
type imported t;  
const c: t = 2.0;
```

---

**Example 2:** An imported constant cannot be used as a static expression, in a type declaration or in an array index for instance

---

```
const imported C: int32;  
type T = int32 C;
```

---

**Example 3:** Invalid static expressions

---

```
const c: int32 = if true then 0 else 1;  
const d: int32 = 0 -> 1;  
const e: bool = true;  
const f: int32 = 1 when e;
```

---

### RELATED TOPICS

- Section 4.2 [“Type Expressions”](#) about size expressions
- Section 3.4 [“Attributes”](#)

## 5.2 Sensors

### SYNTAX

---

```

sensor_block ::= sensor {{ sensor_decl ; }}
sensor_decl ::= ID {{ , ID }} : type_expr

```

---

### STATIC SEMANTICS

A sensor block starts with the keyword **sensor**. Several sensors can be declared using this keyword only once. A sensor declaration is made of a comma-separated list of identifiers different from any other identifiers in the same scope, followed by a colon and the type of identifiers. These identifiers can be used within the current package and its sub-packages where an expression is required. Keywords cannot be used as sensor identifier. Every sensor declaration must end with a semicolon. A sensor can be based on any kind of type (except type variables), but not on a group type. A sensor cannot be used as a clock identifier in a user operator. A sensor must be available at each instant in every node. Therefore, the clock of a sensor is the clock of the context it is used in. Notice that in order to preserve Scade semantics, the value of a sensor flow must not change during a cycle on the global clock. This property should be ensured by the user at integration time.

### DYNAMIC SEMANTICS

A sensor defines a global flow that can be read anywhere, and which is a global input of the model.

#### Example 1: Sensor declaration

---

```

sensor temp: float32;
node gradient() returns (diff: float32)
let
  diff = 0.0 -> temp - pre temp;
tel;

```

---

#### Example 2: Sensor overloading: this program is not well typed

---

```

sensor s: int16;
function Main(s: bool) returns (b: int16) b = s;

```

---

### RELATED TOPICS

- Section 6.3 [“Clock Expressions”](#)
- Chapter 7 about [“Operator Bodies”](#)





# 6

## User-Defined Operators

User-defined operators is the basic syntactic construction provided by the language to structure a model. It allows the reuse of some parts of the model by a call to the defined operators anywhere in the model (possibly quantified by a valid path when necessary). The various analyzes performed on a model result in the definition of appropriate types at the user operator level (see Chapter 1 about [“Introduction”](#)).

- 6.1 [“User-Defined Operators”](#)
- 6.2 [“Variable Declarations”](#)
- 6.3 [“Clock Expressions”](#)
- 6.4 [“Scope Declarations”](#)

## 6.1 User-Defined Operators

### SYNTAX

---

```

user_op_decl ::= op_kind interface_status ID[[ size_decl ]]
               params returns params {{ where_decl }} [[ spec_decl ]]
               opt_body
op_kind ::= function
           | node
size_decl ::= << [[ ID{{ , ID}} ]] >>
params ::= ( [[ var_decls {{ ; var_decls }} ]] )
where_decl ::= where TYPEVAR {{ , TYPEVAR }} numeric_kind
spec_decl ::= specialize path_id
opt_body ::= ;
            | equation ;
            | [[ signal_block ]]
              [[ local_block ]]
            | let {{ equation ; }} tel [[ ; ]]

```

---

### STATIC SEMANTICS

- A user operator declaration binds a new name *id* as an object of kind *op*. This name can be used within the current package and all of its subpackages where an operator name is required. Keywords of the language cannot be used as a user operator identifier.
- A user operator may be declared as a **function** or a **node**. A function defines an operator without internal state: its dynamic semantics does not need to memorize past values. A node defines an operator with an internal state. This information is generated by the type checking of the operator and may be used by tools. The root node can have any of these two status.

An attribute can be given to a user operator declaration. An imported operator declaration cannot have a body part. On the contrary, a non-imported type declaration must have a body part.

- Size variables (input parameters that can be used within static numerical expressions), are declared just after the identifier of the user operator between `<<` and `>>`. These variables are collected with the input names: they must then be exclusive from other input/output variables. The types of input and output parameters can use size parameters as integer identifiers.

- A user operator is a network of operators for which formal input and output parameters have been defined. Empty list of inputs and outputs are allowed, but empty brackets must occur in place of parameters declarations. Inputs must all be different from one another, so must outputs, and both must be exclusive from each other. Order is not relevant inside these lists: a variable can be used by another variable of the same list before its declaration. However, outputs cannot be used in the input lists. Since a user operator opens a new scope according to the namespace analysis, identifiers used in the signature part may overload global flows or type identifiers.

Type variables occurring in the type declarations of the input variables, are also added to the input names: they must then also be exclusive from other input/output size variables. The types of input and output parameters can use type variables. Type variables occurring in output parameters must also occur in input parameters: an output variable cannot introduce a new type variable.

- The *spec* part of the signature declares a specialization of an imported operator. The path identifier must refer to an imported operator. This operator can have itself a subtyping relationship, and can be declared either as a **function** or a **node**. The current operator is not necessarily an imported operator. When the current operator is declared as a **function**, the referenced operator can have any kind. On the contrary when declared as a **node**, the referenced operator can only be a node itself. The current operator cannot have any type variable in its profile, and its type declaration must satisfy the subtyping relationships of the referenced operator if any. The clock signature of the current operator must be compatible with that of the former operator. Its init signature must be at least as strong as that of the former operator.

An imported polymorphic operator must have at least one specialization. There cannot be two specializations of the same operator having the same profile.

- The *where* part of the signature declares subtyping relationships for some of the type variables involved in the profile. The type variables must be declared in a comma separated list. The quoted identifiers in this list must appear at least once in the parameters of the operator. Only integer or float values can be used for the corresponding parameters in the instantiation of an operator having such a declaration. Similarly, only integer or float types of the corresponding kind can be used for those parameters to specialize the operator.

- The body part of an internal user operator can be empty (and thus denoted by a semicolon) in case of an imported operator, or a single equation (including a conditional block or a State Machine), or a set of equations surrounded by **let-~~tel~~** keywords and possibly preceded by local variable declarations. These keywords are mandatory in case of an empty set of equations (*i.e.*, when the outputs list is empty) for an internal operator.

Signals are local variables introduced by a special keyword **sig**. Signal identifiers cannot be used directly within the equation part. They can only appear in their quoted form. No type information is required for a signal declaration, a signal being manipulated as a Boolean flow. Their name should be different from other local and signature variables. Local variables are introduced by the keyword **var**. The identifiers declared in this block must be exclusive from the environment defined by the signature and signal parts.

The equation part defines equations, conditional blocks, and State Machines. Each output and local variable must appear exactly once in the left part of an equation. Every input, size, and local variables must be used at least once in these equations. Signals must be emitted and caught at least once. The current operator's identifier must not occur in the equation part, leading otherwise to a recursive definition which is considered as an error. Similarly, two or more nodes cannot be mutually defined. A user operator can be ended by an optional semicolon.

- A user operator is well typed if its signature and body parts are well typed in the current typing environment. The type analysis provides this operator with an operator type made of the Cartesian product of its inputs type and the Cartesian product of its outputs type, plus a *k* bit stating its need for referring a past value. A **function** cannot have to refer to a past value through an sequential operator. On the contrary, a **node** can do so, but it is allowed to refer only to current values. Every Scade primitive operator is given such a *k* bit in order to allow them or not in a **function**.

If some of the signature's types are type variables (this operator being polymorphic), some extra constraints may be associated with this type in order to restrict the range of values allowed during instantiation. These constraints are either stated by the user in the *where* part of the signature, or generated during type analysis. They should be satisfied by every instantiation of this operator.

If size variables were declared in the signature part, then they are clearly mentioned in the operator type. An instantiation of this operator is then performed in two parts: first its size arguments, then the remaining ones. Size constraints generated during the type analysis of the body part must be satisfied by the size arguments. Default constraints state that each size variable must be strictly positive.

- A user operator is well clocked if its signature and body part are. The clock analysis provides this operator with an operator clock type made of the Cartesian product of its inputs clocks and the Cartesian product of its outputs clocks. This type is generalized so that this operator can be instantiated and checked in any clock context. This generalization consists in quantify universally the fastest clock of the signature's variables. See operator instantiation in section 8.5 [“Operator Application and Higher-Order Patterns”](#) for further details.

A special case occurs when the operator has no input. In this case, the clock type of the operator takes as a default input the generalized base clock.

- A user operator is causal if its body and signature part are, and if the set of causality constraints generated during the analysis is satisfiable. The causality analysis provides this operator with a type made of the Cartesian product of universally quantified inputs causal variable, and of the corresponding product of outputs causal types. Each output type is the union of the input causal types on which this output depends on. The quantification of the input causal types allow to reuse this operator type in the various instantiations of the operator. In the case of an imported operator, the default choice is to consider that each output depends on every input.
- A user operator is well initialized if its signature and body part are. The initialization type analysis provides this operator with an operator delay type. Given the delay types of the inputs, the operator type produces the delay types of the operator outputs. Inputs delay type is either atomic, or represented by a universally quantified variable if it is not constrained. Outputs delay types are either atomic or the result of the maximum delay of some input delay variables. In the case of an imported operator, the default choice is to force every input and every output to be well initialized. The operator delay type has thus no universally quantified variable.

## DYNAMIC SEMANTICS

The dynamic semantics of user-defined operators is to produce its output flows from its input flows, the value of these output flows being given by the definitions of these flows in the operator body.

**Example 1:** Several occurrences of the same variable name:

---

```
node ex1(x: int16; x: bool) returns (y: float32) ...;
```

---

**Example 2:** Error due to the overloading of an input variable by a size variable:

---

```
node ex2 <<n>> (n: int32) returns (p: int32) ...;
```

---

**Example 3:** Error due to the overloading of a type variable by an output variable:

---

```
function ex3(x: 'T) returns (T: int32) ...;
```

---

**Example 4:** Given the function:

---

```
function ex4(clock h: bool; y, z: int16) returns (o1: int16; o2: bool)
let
  o1 = merge (h; y when h; z when not h);
  o2 = (y > z);
tel;
```

---

Then this operator is given the following types during the modular analyses:

- This operator has type:  $\text{bool} \times \text{int16} \times \text{int16} \xrightarrow{0} \text{int16} \times \text{bool}$

The 0 flag means it is a stateless function, because of the use of the **function** keyword. One could have used the **node** keyword with no error.

- This operator requires on instantiation an identifier tagged as a **clock** and does not add a clock constraint on its inputs nor on its outputs:  $\forall \alpha, (X:\alpha). (X:\alpha) \times \alpha \times \alpha \rightarrow \alpha \times \alpha$
- The first output depends on all the inputs, while the second depends only on the last two ones:  $\forall \gamma_1, \gamma_2, \gamma_3. \gamma_1 \times \gamma_2 \times \gamma_3 \rightarrow \gamma_1 \cup \gamma_2 \cup \gamma_3 \times \gamma_2 \cup \gamma_3$
- The first input is required to be well initialized on its first cycle of activation. No other constraint is added:  $\forall \delta_1, \delta_2. 0 \times \delta_1 \times \delta_2 \rightarrow \delta_1 \sqcup \delta_2 \times \delta_1 \sqcup \delta_2$

**Example 5:** Given the node:

---

```
node ex5(clock h: bool; y: int32 when h) returns (o1: int32 when h last = 0 when h)
let
  activate if y > 0 when h
  then o1 = y;
  else
  returns ..;
tel
```

---

Then this operator is given the following types during the modular analyses:

- This operator has type:  $\text{bool} \times \text{int32} \xrightarrow{1} \text{int32}$

The **1** bit is due to the use of the **node** keyword. The use of the **function** keyword would have raised an error.

- Its clock profile is:  $\forall \alpha, (X:\alpha). (X:\alpha) \times \alpha \text{ on } X \rightarrow \alpha \text{ on } X$ . It requires on instantiation an identifier tagged as a **clock**, and another argument sampled on this clock. Due to the generalization on  $\alpha$ , the first parameter can be based on any rate.
- Its output depends on both its inputs:  $\forall \gamma_1, \gamma_2. \gamma_1 \times \gamma_2 \rightarrow \gamma_1 \cup \gamma_2$ .
- Its initialization type is:  $0 \times 0 \rightarrow 0$ . It requires on input two well-defined flows and produces on output a well-defined one.

#### RELATED TOPICS

- Section 4.2 [“Type Expressions”](#) about size expressions
- Section 8.5 [“Operator Application and Higher-Order Patterns”](#) about operator instantiation
- Section 7.1 [“Equations”](#)

## 6.2 Variable Declarations

Variables can be declared at several places: for input and output parameters in a node declaration, for local variables in scope declarations.

### SYNTAX

---

```

var_decls ::= var_id {{ , var_id }} : type_expr [[ when_decl ]]
                                                [[ default_decl ]]
                                                [[ last_decl ]]

var_id ::= [[ clock ]] [[ probe ]] ID
when_decl ::= when clock_expr
default_decl ::= default = expr
last_decl ::= last = expr

```

---

### STATIC SEMANTICS

- A variable declaration binds a new name *id* to a flow. This name can be used in the current context where a variable name can occur. Keywords of the language cannot be used as a variable identifier. Several distinct identifiers can be declared at the same time in a comma separated list. Each identifier may be given specific attributes:

**clock**: to declare that this identifier can be used to sample other flows. This status is lost when entering a conditional block or a State Machine.

**probe**: a tool-dependent directive ensuring that this variable name will be preserved during compilation. It ensures the observability of the corresponding variable.

The order of these attributes is relevant: when present, the **clock** attribute must be located before the **probe** one. During instantiation, identifiers declared as clocks can only be given clock identifiers as parameter.

- The type of these identifiers must be given after a colon. Any type expression can be used. Declared group identifiers can be used as a type expression, except for identifiers that are given the **clock** tag. Other group expressions are not allowed in the type declaration part.



The type declaration can be followed by a clocking expression *when\_decl*. This expression states that the identifiers in the list are based on a different clock than the base clock of the operator. A valid clock expression must be used, in particular an identifier cannot be sampled on itself. A list of identifiers cannot be samples on one of its elements.

- Optional declarations can then follow in any order the type expression: a *default* value declaration, and a *last* declaration. Applied to a list of variable names, these declarations will be mapped onto each variable.

The **default** declaration is introduced by the '=' symbol. The default expression is any Scade expression, and may involve any other valid variable names in the scope of this declaration. This declaration can only occur for defined variables (output or local) of internal operators. The expression used in this declaration must be defined at the first cycle.

The **last** declaration is introduced by the '=' symbol. The last expression can be any expression, possibly involving other variable names present in the scope of this declaration. This declaration can occur for any kind of variables of internal operators. This expression may not be defined at first cycle.

Both default and last expressions must share the same type and the same clock. The type must match the declared type of the variables. The clock must match the declared clock in the *when\_decl* if any. When analysing causality, the default expression contributes to the causality of each variable for which it is declared, while the last expression does not necessary. Indeed, only if no default expression is mentioned in the declaration of a defined variable, and if one of the cases in a conditional block (or state in a State Machine) does not mention an equation for this variable, then a causality constraint will be added between the last expression and this variable.

No default or last expression can be attached to an object belonging to a group type. More generally, no default or last expressions can be attached to an imported operator.

- A variable declaration introduces a new causality type in the environment. A non-membership constraint is systematically posted with this new type: the variable cannot depend on itself.

## DYNAMIC SEMANTICS

When a flow is unspecified in a state of a State Machine or in a branch of a conditional block, the default behavior is to maintain the value it had at the previous cycle (whatever state or branch was activated then). At the first instant of activation, this previous value does not exist. Therefore, a **last** expression must have been declared. This expression is only evaluated at this first instant.

If a **default** expression exists for this flow, this **last** expression is not evaluated at all. At any cycle where the flow is undefined, the **default** expression is evaluated instead.

**Example 1:** Invalid use of a variable as a clock

---

```
(x: bool; y: int32 when x)
```

---

**Example 2:** Invalid default expression: it must be defined at first cycle

---

```
(x: int32; y: int32 default = pre(x))
```

---

**Example 3:** Error due to clock analysis

---

```
(clock h: bool; y: int32 default = 1 when h last = 0 when not h)
```

---

## RELATED TOPICS

- Section 8.1 [“Basic Expressions”](#) about Last operator
- Section 8.2 [“Sequential Operators”](#) about Initialization operators
- Section 7.2 [“Conditional Blocks”](#)
- Section 7.3 [“State Machines”](#)

## 6.3 Clock Expressions

In a model, flows can have different rates in the sense that they are not required to produce a value at each cycle. This may occur when putting together subsystems that are not based on the same time scale for their inputs, or to prevent the computation of an unnecessary costly expression.

The logical time defines the fastest possible rate of the system (called the *base clock*). All other rates, or *clocks*, are derived from the base clock. The specification of these different clocks is made according to sampling operators or conditional blocks. In any case, the rate is defined by a clock expression.

### SYNTAX

---

```
clock_expr ::= ID
            | not ID
            | ( ID match pattern )
```

---

### STATIC SEMANTICS

A clock expression is either a fully qualified identifier referring to a variable or constant name, or the negation of such an identifier, or a pattern matching. In the first two cases, the identifier must refer to a Boolean name. In the latter case, the identifier must refer to an object belonging to an **enum** data type, and the pattern part must refer to one of the item of this data type.

A clock expression is well clocked if the identifier `ID` on which it is based on have been declared as usable as a clock (**clock** declaration). Its clock is the clock of this identifier. A clock expression defines a slower clock that can be used by sampling operators or conditional blocks.

Clock expressions must be defined at first flow: their initial value cannot depend on an initialization performed elsewhere in the expression.

Notice that clock expressions are only required by the sampling operator **when** and by the first **activate** higher order operator. They cannot be produced as a result by any operator.

## DYNAMIC SEMANTICS

A flow that is not constrained by any clock expression has the same rate as a flow clocked on the constant `true`.

### Example 1: Valid clock expressions:

---

```
type t= enum {incr, stdby, decr};
function Sample (clock h: bool; clock k: t; x: int32) returns (y, z: int32)
let
  activate when k match
    | incr : y = x + 1;
    | stdby : y = x ;
    | decr : y = x - 1;
  returns y;
z = merge (h; x when h; x when not h);
tel;
```

---

### Example 2: Invalid clock expressions:

---

```
clock h: bool;
var k: int32;
var x: bool when k;
var y: bool when h match Blue;
var z: bool when pre h;
```

---

## RELATED TOPICS

- Section 7.2 [“Conditional Blocks”](#)
- Section 8.2 [“Sequential Operators”](#) about When operator
- Section 8.5 [“Operator Application and Higher-Order Patterns”](#) about Activate operator
- Chapter 7 about [“Operator Bodies”](#)

## 6.4 Scope Declarations

Scope declarations occur in the body part of user-defined operators, actions in conditional blocks, states and actions in State Machines.

### SYNTAX

---

```

data_def ::= equation ;
          | scope
scope ::= [[ signal_block ]] [[ local_block ]] [[ eqs ]]
signal_block ::= sig ID {{ , ID }} ;
local_block ::= var {{ var_decls ; }}
eqs ::= let {{ equation ; }} tel

```

---

### STATIC SEMANTICS

A scope is a set of equations surrounded by **let-*tel*** keywords and possibly preceded by local variable declarations.

Signals are local variables introduced by a special keyword **sig**. Signal identifiers cannot be used directly within the equation part. They can only appear in their quoted form. No type information is required for a signal declaration, a signal being manipulated as a Boolean flow. A signal is always well-defined at its first instant of activation.

Local variables are introduced by the keyword **var**. The identifiers declared in these blocks must be exclusive from those in the current environment.

The equation part defines equations, conditional blocks, State Machines, and signal emissions. Each local variable declared in this scope must appear exactly once in the left part of an equation or in a **return** statement of a conditional block or a State Machine. Every local variable must be used at least once in these equations. Signals must be emitted and caught at least once.

The left hand side of a scope (the set of variables occurring in the left hand side of an equation of this scope) can contain variables coming from an upper scope.

### Example 1: Invalid overloading of signals identifiers:

```
node ex1(x: int16) returns (y: float64)
sig x
let
  ...
tel;
```

---

### Example 2: Valid overloading of signals identifiers: inside the body of operator ex2, the identifier c refers to the local signal which masks the global constant

```
const c: int32 = 0;
function ex2(x: bool) returns (y: int32)
sig c
let
  ...
tel;
```

---

### Example 3: Unused local variable:

```
node ex3(x: int8) returns (y: int8)
var z: int8 last = 0;
let
  z = 2*pre(x) + 1;
  y = x - pre(x);
tel;
```

---

#### RELATED TOPICS

- Section 7.1 [“Equations”](#)
- Section 7.3.1 [“State Machines”](#)
- Section 7.2 [“Conditional Blocks”](#)
- Chapter 7 about [“Operator Bodies”](#)

# 7

## Operator Bodies

Output and local variables must have exactly one definition in a user operator body. This definition can be expressed either by means of a unique equation, or by stating several behaviors depending on control conditions in control blocks or State Machines. Signals declared in local scopes must be emitted at least once, but may be emitted several times.

- 7.1 [“Equations”](#)
- 7.2 [“Conditional Blocks”](#)
- 7.3 [“State Machines”](#)

## 7.1 Equations

Equations allow to define the data flow expression associated with an output or local identifier. This expression is evaluated at each cycle, according to input/output and current/previous values, then assigned to this identifier. Equations can either be declared in a data-flow or a control-flow flavor, according to the problem at hand.

### SYNTAX

```

equation ::= simple_equation
           | assert
           | emission
           | control_block return
simple_equation ::= lhs = expr
lhs ::= ( )
       | lhs_id {{ , lhs_id }}
lhs_id ::= ID
         | _
assert ::= assume ID : expr
         | guarantee ID : expr
control_block ::= state_machine
               | clocked_block
emission ::= emit emission_body
emission_body ::= NAME[[ if expr ]]
               | ( NAME{{ , NAME }} ) [[ if expr ]]
return ::= returns returns_var
returns_var ::= {{ ID , }} (( ID | .. ))

```

### STATIC SEMANTICS

- Equation blocks occur in the body part of user operators. Equation blocks are composed of different kinds of objects:
  - **Simple equations** assign an expression to a list of identifiers, putting in front the data flow aspect of this expression.
  - **Asserts** are the syntactic means to declare Scade contracts.
  - **Signal emissions** allow to specify the presence of a signal.
  - **Control blocks** are an orthogonal way to assign expressions to identifiers, putting in front the control aspect of these expressions.

The above objects can appear in any order in the body part of the user operator, until every output and local variable has been given a definition, and each signal been emitted.



- A **simple equation** assigns a Scade expression to a possibly empty list of identifiers. This list is specified either by an empty list '()' when no output is produced by the expression, or by a comma separated list of identifiers without braces:
  - In the first case, the only way to produce such an empty list is through the application of a user operator without outputs. Operators allowed to manipulate such an application are those allowed for groups (see section 4.3 [“Group Declarations”](#)). Note that the empty list is not allowed in the right part.
  - In the second case, identifiers are either identifiers of output flows, identifiers of local flows (excluding signals), or an *undefined* symbol specified by an underscore character '\_'. This symbol can occur at any place and any number of times. Identifiers must be valid identifiers according to the namespace analysis. Note that these identifiers may not having been declared in the latest scope.

A simple equation is well-typed if the type of the expression is equivalent to the type of the identifiers. The type of an identifier is the type to which it is associated in the typing environment. The empty list is given a special type, noted '()'. The undefined identifier is given the type required by the expression at the corresponding place. This type can be an atomic or structured type, but not a group type. If several identifiers are declared in this left hand side, the corresponding type is the Cartesian product of all corresponding types of the identifiers. A simple equation is well-clocked if the clock of the expression is the same as the clock of the identifiers. The clock of an identifier is the clock with which it is associated in the clocking environment. The clock of the empty list of identifiers is the clock of the context it is used in. The clock of the undefined identifier is the one that is required by the expression at the corresponding place. The clock of a list of identifiers is the Cartesian product of the list of clocks.

The definition of a bunch of flow identifiers by an expression in a simple equation introduces a dependency between the defined identifiers and those reachable by the expression. Thus, a corresponding causality constraint is introduced. The causality type of an identifier is the type with which it is associated in the causality environment. This identifier also depends on the activation context it is defined in if any. The empty list of identifiers depends on nothing. Similarly for the undefined identifier. The dependencies of a list of identifiers is the Cartesian product of the causality types of identifiers.

A simple equation is well initialized if the initialization type of the expression is the same as the one of the identifiers. The initialization type of an identifier is the one to which it is associated in the initializing environment. An empty list of identifiers is

always correctly initialized (*i.e.*, it is associated with the type '0'). The undefined identifier is given the initialization type required by the expression at the corresponding place. The type of a list of identifiers is the Cartesian product of the initialization types of the identifiers.

- **Asserts** are of two forms:
  - **assume** corresponds to expectations on the Scade program and may involve current inputs and past outputs.
  - **guarantee** ensures the properties of the program. Current outputs may be used in this case.

Each of these forms can be given an identifier to allow the traceability of these contracts. This identifier must be exclusive from all other identifiers of the local scope, and from other contract identifiers. It can however overload an identifier from an older scope, preventing the usage of the former declarations. The identifier of a contract cannot be referred to. Expressions used in **assume** can only mention input identifiers or past variables. No such restriction is imposed for **guarantee** expression. An assert is well typed if the associated expression is of Boolean type. The clock of the expression must be the same as the clock of the context it is used in. An assert does not introduce constraints during the causality analysis. Finally, the expression must be well defined at its first instant of activation so as to ensure the contract even for the first cycle of the execution.

- **Signal emission** is declared by **emit** keyword. This emission concerns one or several quoted identifiers previously declared as signals. In case of emission of several signals at the same time, parenthesis are mandatory around the quoted identifiers. Signal emission may be conditioned by an expression declared after keyword **if**. A signal identifier may occur in several emissions. Contrary to other flows, unicity of the definition is not required. A simple signal emission is always well-typed. When conditioned, its associated expression must be of Boolean type. Similarly, no condition is required for the clocking analysis of a simple emission, while a conditioned one requires the expression to be well-clocked. No context information is used here, contrary to asserts. A simple emission adds a dependency between the causality type of the signal identifiers and the causality context. When conditioned, an extra causality constraint is added to state a dependency between the signal identifiers and the conditional expression. Finally, a simple signal emission is always well initialized, while a conditioned one requires its expression to be well defined at first cycle.

- **Control blocks** are either conditional blocks or automata. They are detailed in sections 7.2 [“Conditional Blocks”](#) and 7.3 [“State Machines”](#). A control block must always be followed by a **returns** statement that lists the variables defined in the control block. This possible empty list must contain only local and output variables that are not defined elsewhere. The special symbol ‘..’ used in this list is an ellipsis construct implicitly representing all variables defined in the block. Ellipsis variables are those that appeared at least once in a left hand side equation in the control block or in **returns** statement in a sub-control block. The ellipsis symbol must be used for empty list of identifiers and can be used at the last place of other lists.

### DYNAMIC SEMANTICS

A simple equation declares the flow of its left hand side to be equal to those defined by its expression at every cycle of the execution. Asserts do not have a dynamic semantics: even though the intention behind them is to state properties on the program, they do not need to be taken into account in the execution. A signal emission of signal identifiers sets their presence to **true** at the current cycle. When conditioned, their presence is set to **true** only if the expression is itself present and evaluates to true at this cycle. In case of a list of signal identifiers, every signal follows the same behavior.

**Example 1:** Overloading of an assume identifier, preventing the use of the former declaration.

---

```
const c: int32 = 0;
node ex(x: int32; b: bool) returns (y: int32)
let
    assume c: true;
    y = if b then x else c -> pre y;
tel;
```

---

**Example 2:** Error in causality analysis:  $x$  and  $y$  are mutually defined.

---

```
emit 's if x;
x = if y > 0 then false else true -> pre(x);
activate if x
then y = 1;
else y = if 's then 0 else 0 -> pre(y);
returns y;
```

---

### RELATED TOPICS

- Section A-1 [“Namespace Analysis”](#)
- Section 6.2 [“Variable Declarations”](#)
- Section 6.4 [“Scope Declarations”](#)

## 7.2 Conditional Blocks

Conditional blocks are convenient to express control structures when the control flow only depends on a condition computable in the current cycle. Depending on its type (Boolean or enumerated), this condition may lead to two or more switch cases. Each case proposes a definition of a subset of the whole set of variables defined by this conditional block. Undefined variables are either maintained to their previous value (the *last* one) or follow a default behavior stated in their declaration (see section 6.2 [“Variable Declarations”](#)).

### SYNTAX

---

```
clocked_block ::= activate [[ id ]] (( if_block | match_block ))
```

```
if_block ::= if expr then (( data_def | if_block ))
           else (( data_def | if_block ))
```

```
match_block ::= when expr match {{ | pattern : data_ef }}+
```

---

### STATIC SEMANTICS

- A conditional block can be given an identifier for traceability purposes. This identifier must be different from all control block identifiers present at the same scope level. Since a conditional block opens a new scope, the identifier can be reused within this scope. Contrary to asserts, a conditional block identifier cannot overload another kind of identifier.

A conditional block is either an *if\_block* or a *match\_block*, depending on the type of the conditional control expression. The branches of an *if\_block* can contain either another *if\_block* or a scope declaration. Both branches must be present. The branches of a *match\_block* can only contain a scope declaration. Each branch of a *match\_block* is identified by a valid pattern. Any branch of a decision block can be empty.

The set of variables defined by a conditional block is the union of all the variables that occur in the left hand side of the equations defined in the various branches. The set cannot be empty. Each variable should be given at least one definition in one of the branches of the decision block. This set is possibly specified by the return statement ending the block.

- A conditional block is well-typed if the decision block it contains is. It may introduce implicit memories when variables are not defined in every branch of the decision block. An *if\_block* is well-typed if its two branches are and if the condition is of type `bool`. A *match\_block* is well-typed if:
  - Its condition is of an enumerated type (keyword `enum`).
  - Its patterns exclusively belong to this type (no `'_'` is allowed), are all different from each other, and cover all the possible values of the type.
  - Each scope declaration in the branches are well-typed.

The order of patterns does not need to follow the order of the declared enumeration.

- A conditional block is well-clocked if the decision block it contains is. An *if\_block* is well-clocked if its condition is and if its branches are in a filtered clocking environment. The filtering operation removes the identifiers whose clock type is not a sub-clock of the condition's clock type. Using the identifiers in an expression of a branch leads to clock error. The remaining identifiers are sampled according to the clock type of the condition (*then* branch being sampled on the positive values of the clock expression, *else* branch on the negative ones). This enforces the locality of treatments in a branch: a reference to a previous value with operator `pre` refers to the last instant the branch was activated.

The filtering operation discards the clock status: identifiers declared as clocks outside the automaton cannot be used to sample data inside the automaton. A special case occurs when the condition is itself an identifier declared as a clock (see section 6.2 [“Variable Declarations”](#)). In this case, the filtering operation does not reject identifiers already sampled on this clock identifier: they can thus be used in the corresponding branch.

A *match\_block* is well clocked if its condition and its branches are in the filtered environment. The filtering operation also follows the algorithm above. In each branch, the remaining identifiers are then sampled on the clock of the corresponding pattern. The equivalent special case occurs: an identifier already sampled on an enumerated clock identifier used as a condition in a *match\_block* can be used in the corresponding branch.

- A conditional block is causal if the decision block it contains is. An *if\_block* is causal if each branch is, owing to the fact that each equation defined in the branches depends on the causality type of the condition. In other words, an identifier reachable from the condition belongs to the causality type of every variable defined

by this decision block. A *match\_block* is causal if each branch is. Similarly, every variable defined by this decision block also depends on the reachable variable of the condition.

- Every variable defined by a conditional block must be defined at the first instant. Similarly for the condition of an *if\_block* or a *match\_block*. A decision block is well initialized if each branch is.

### DYNAMIC SEMANTICS

When the condition of a conditional block is present (*i.e.*, when its associated clock expression evaluates to **true**), its evaluation leads to the activation of a single branch. When the decision block is a *if\_block*, the **then** branch is activated when the Boolean condition evaluates to **true**, while the **else** branch is when the condition evaluates to **false**. In case of a *match\_block*, the activated branch is the one which patterns equals the evaluation of the condition.

At each cycle where the condition is present, a conditional block binds a value to each variable it defines. This value corresponds either to the evaluation of the expression associated in the activated branch of the conditional block, or to a default value when this expression is missing. Local definitions are only evaluated in their corresponding branch when it is activated. The default value is either the previous value of this variable (the last one that has been computed by any other branch, not necessarily that of the previous cycle since it may not exist due to the clock of the condition), or the evaluation of the **default** expression stated in the declaration of the variable. Inactivated branches of a conditional block keep their internal state: the **pre** operator occurring in the expressions of these branches will refer to the last cycle when their branch was activated.

A conditional block can be translated to a bunch of equations, one for every variable it defines. Each equation uses a **merge** operator based on the condition of the conditional block. Each branch of this operator contains either the expression belonging to the corresponding branch sampled on the appropriated clock expression based on the condition, or a reference to the default behavior also sampled when no expression is associated with this variable in the corresponding branch. This translation illustrates the similarity between the conditions of conditional blocks and clocks.

**Example 1:** The condition must be either Boolean or must belong to an enumerated type. This node is erroneous.

---

```
node ex1(x: int16)
returns (y: int16)
let
  activate when x match
  | 0: y = 0;
  | : y = 1;
  returns ..;
tel
```

---

**Example 2:** Defined variables must have at least a definition in one of the branches. In this example, o2 has no definition.

---

```
node ex2(i: bool)
returns (o1,o2: int64)
let
  activate Act if i
  then o1 = 1;
  else o1 = 2;
  returns o1,o2;
tel
```

---

**Example 3:** Conditional blocks can be nested. Notice also that the last branch is empty.

---

```
type Tenum = enum {red, blue, pink, purple};
node ex3(eI1: Tenum; iI2: int32)
returns (iO1: int32 last = 0)
let
  activate when eI1 match
  | red: var iV1: int32;
  let
    iV1 = 10 + last 'iO1;
    iO1 = iV1 + iI2;
  tel
  | blue:
  let
    activate if iI2 > 0
    then iO1 = iI2 * iI2;
    else iO1 = -iI2 + last 'iO1;
    returns iO1;
  tel
  | pink: iO1 = 100 -> pre iO1 - 1;
  | purple:
  returns iO1;
tel
```

---

**Example 4:** The following example illustrates a causality error for variable `y`.

---

```

type T = enum {a, b, c};
node ex5(x: T) returns (y: T last = b)
let
    activate when y match
    | a: y = a -> pre(y);
    | b: y = x;
    | c:
    returns ..;
tel

```

---

**Example 5:** A branch of a decision block can be empty. The defined flow must have a **last** or a well-initialized **default** declaration.

---

```

function ex1(i: int32)
returns (o: int32 last = 0)
let
    activate if i > 0
    then o = i;
    else
returns o;
tel

```

---

```

function ex1(i: int32)
returns (o: int32 default = 4)
let
    activate if i > 0
    then o = i;
    else
returns o;
tel

```

---

## RELATED TOPICS

- Section A-3 [“Clock Analysis”](#)
- Section 6.4 [“Scope Declarations”](#)
- Section 8.3.4 [“Flows Switches”](#) about pattern



## 7.3 State Machines

State Machines offer the most elaborate combination of control and data flow information in a model. Intuitively, State Machines extends the conditional block construct when the condition cannot be computed without memorizing extra expressions. In this case, the control structure can be best expressed by mean of a State Machine whose states contain the needed information. State machines in Scade provide two ways to specify transitions between states, weak or strong, that influences the execution algorithm.

- 7.3.1 [“State Machines”](#)
- 7.3.2 [“Transitions”](#)
- 7.3.3 [“Actions”](#)
- 7.3.4 [“Examples”](#)

### 7.3.1 State Machines

#### SYNTAX

---

```
state_machine ::= automaton [[ ID ]] {{ state_decl }}
```

```
state_decl ::= [[ initial ]] [[ final ]] state ID
              [[ unless {{ transition ; }}+ ]]
              data_def
              [[ until {{ transition ; }}
              [[ synchro [[ actions ]] fork ; ]] ]]
```

---

#### STATIC SEMANTICS

- A State Machine can be given an identifier for traceability purposes. This identifier must be different from all control block identifiers present at the same scope level. Since a State Machine opens a new scope, this identifier can be reused within this scope. A State Machine identifier cannot overload another kind of identifier: this identifier cannot be referred to. A State Machine is composed of a non-empty set of identified states. Each state identifier must be unique at a given level. However, a state identifier can be reused at a different level: sub or super state.

Any state can be tagged with the label **initial**, not necessarily the first one. But only one and exactly one state must be tagged with this label. Any state can be tagged with the label **final**, even the initial one. However, this label is useless when no **synchro** transition exists immediately outside the current automaton. In particular, a **final** state is useless in the top level automaton.

The body of a state is made of the following ordered elements:

- A possibly empty list of strong transitions introduced by the keyword **unless**
- A possibly empty scope declaration
- A possibly empty list of weak transitions introduced by the keyword **until**

Strong and weak transitions can lead to the same state in which they occur. A state can remain completely empty. A **synchro** transition can occur in a (possibly empty) list of weak transitions.

Weak transitions can mention local variables and signals declared in the scope of the body, whereas strong transitions are not allowed to. When a **synchro** transition is present in a state, all the State Machines that appear immediately inside this state must have at least one state labelled with **final** attribute. If one of these State Machines does not have such a final state, the synchro transition will never fire. On the contrary, if no State Machine appear within this state, this **synchro** transition will always be fired. All the states of a given State Machine should be reachable from the initial state. This analysis does not rely on the satisfaction of the conditions used on the transitions, but only on the existence of paths of transitions starting at the initial state and covering every state of the automaton.

The set of variables defined by a State Machine is the union of all the variables that occur in the left hand side of the equations defined in the various states and transitions. This set cannot be empty and is defined in the return statement ending this block. Each variable should be given at least one definition in either one of the state or transition of the State Machine. A variable occurring in a left hand side of a transition can occur in the left hand side of any other transition but not in inside any state. A variable cannot be given two definitions in the same state, nor in the same transition.

- A State Machine is well typed if all its states are. A state is well typed if all its outgoing transitions, both strong and weak, are well typed, and if its scope declaration is. This rule holds whenever this state is labelled initial and/or final.

- A State Machine is well clocked if all its state are. As for the branches in conditional blocks, each state is analyzed in a filtered environment. The filtering algorithm follows the same principles as for conditional blocks: removing identifiers that are not on a subclock of the clock of the automaton and sampling the remaining ones. In conditional blocks, this sampling operation uses a clock derived from the condition. In State Machines, an abstract clock is built out of the enumeration of all the states at a given level (one enumerator per state). The sampling can be seen as what is done in a *match\_block*, the patterns corresponding to the states.

A state is well clocked if its strong transitions are in the filtered environment of the automaton. The scope of a state is analysed on another version of the filtered environment, the weak transitions being analysed on the same version extended by the identifiers introduced by the scope. This distinction in filtering between strong and weak transitions is used in the definition of the behaviour of State Machines (see below): the strong transitions being defined when the state is selected, the body and weak transitions when the state is activated.

- The causal dependencies of a State Machine is analyzed state by state. A state is causal if its strong transitions are, and if its scope and weak transitions are, owing to the fact that each equation in the latter depends on the causality types of the strong transitions. Every identifier reachable from a strong transitions is also reachable from the scope and weak transitions of the same state. A strong transition cannot depend on a computation made in the target state. On the other hand, a weak transition can depend on computations made in the active state.
- A State Machine is well initialized if all its states are. Each defined variable must be well defined at its first cycle of activation. A state is well initialized if its strong transitions are in the context of the automaton, if its scope is, and if its weak transitions are in the context extended by variables declared in the scope. Two cases are distinguished for the analysis of the scope: whether the state is initial or directly reachable by a strong transition from the initial state, and other states. In the first case, the scope is analyzed in the same context as strong transitions, whereas in the second case, the variables defined by the automaton are all considered well initialized when calling them through a **last** operator (similarly when their definition is missing). Indeed, the second case concerns states which cannot be directly reachable at the first cycle of the execution, and thus implies that an initial value as already been bound to these variables.

## DYNAMIC SEMANTICS

The behavior of State Machines can be simulated by conditional blocks. The translation between State Machines and conditional blocks relies on the definition of two global flows: the *selected* state and the *active* state. The selected state identifies the state whose strong transitions are examined. The active state is the state whose body and weak transitions are examined. The active state is the same as the selected state unless a strong transition has occurred toward another state. And next cycle's selected state is the same as the current active state unless a weak transition has occurred toward a different state. The type of these flows is an enumerated type that represents the states of the automaton.

Given these two flows, the behavior of State Machines follows three main rules:

- 1 At each cycle, the State Machine evaluation starts at the selected state.
- 2 At each cycle, there is only one active state.
- 3 At each cycle, at most one transition can be fired.

At each cycle, the selected state is state *S* if it is declared as **initial** and it is the first cycle of execution, or a **restart** operator (see below or in section 8.5 [“Operator Application and Higher-Order Patterns”](#)) has been applied to this automaton. Otherwise, *S* is selected if it has been designated such at the previous cycle.

Once the selected state is known, the behavior of the State Machine follows this sequence of steps:

- Evaluate, if any, all the strong transitions conditions of the selected state. The first one being true in the textual order of appearance is then fired, leading to a state identified as the active one. In case no strong transitions exists or none of them being true, the selected state becomes the active one.
- The body of the active state is evaluated.
- If no strong transition has been fired at the current cycle, the conditions of the weak transitions are evaluated. The first one being true in the textual order of appearance is fired, leading to a state identified as the next selected state. In case no weak transition exists, or none of them being fireable, if all the State Machines defined within the current state are all in their final state, the synchro actions (if any) are evaluated. Finally, the next cycle's selected state is the current active state.

Due to the third rule, the combination of weak and strong transitions must not lead to the evaluation of two states bodies at the same cycle. Consider the case of a weak transition followed by a strong one:

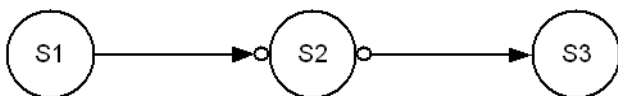


Figure 7.1: Weak Strong mix

If the *weak* transition is fired from active state *S1*, target state *S2* is selected at next cycle. Then at this next cycle, if the strong condition holds, state *S2* is not executed, and *S3* becomes active. On the other hand, when a strong transition is followed by a weak one as in the following example:



Figure 7.2: Strong Weak mix

If a *strong* transition is fired from selected state *S1*, target state *S2* is active. But, since there is a fired strong transition, weak transitions of *S2* are not evaluated. *S2* will be the selected state at next cycle.

Note that during the parallel evaluation of strong or weak transitions, possible internal states of conditions may change.

The local flows defined in the body of a state are only defined when this state is activated. The flows defined by a State Machine must have a definition at every cycle this automaton is present. When unspecified in a state, the default definition of a variable is given in the `default` declaration of this variable, or consists in maintaining its last value (possibly computed in a different state) when this information is not specified. A flow defined on transitions is evaluated only when this transition is fired. In other case, the same default behavior is applied.

## 7.3.2 Transitions

### SYNTAX

---

```

transition ::= if expr arrow

        arrow ::= [[ actions ]] fork

        fork ::= target
                | if expr arrow {{ elsif_fork }} [[ else_fork ]] end

elsif_fork ::= elsif expr arrow

else_fork ::= else arrow

        target ::= restart ID
                | resume ID

```

---

### STATIC SEMANTICS

- A transition is composed of a conditional expression introduced by the keyword **if** possibly followed by an action and ended by a fork. Actions are detailed in the section below. A fork is either a simple target or a more complex transition that may depend on several conditions. This complex transition follows an **if** structure, possibly extended by one or more subcases declared by **elsif** keyword. The initial **if** and each of the subsequent **elsif** are followed by a condition and another possibly complex target. The final branching may be either a **else** or a **elsif** one. This nesting of complex transition must be closed by keyword **end**.

The various conditional expressions used in a transition must only refer to valid identifiers in the same environment as the state they appear into. A target is either a **restart** or a **resume** toward a state belonging to the current automaton. This state cannot belong to a sub or super automaton.

The union set of variables that occur at a left hand side in an action is the set of variables defined on this transition. These variables must be distinct from those defined in a state of the current automaton.

- A transition is well typed if all its conditions are Boolean expressions and if its actions are well typed in the same typing environment. The targets of a transition are always well typed. A **synchro** transition is well typed if its associated actions are, and if its fork part is.
- A transition is well clocked if its initial condition is in the filtered environment when the state is active. The clocking analysis is then performed in the clocking environment again filtered on the instant when this condition holds. The fork structure is then explored: each condition has to be well clocked in the filtered environment and leads to two filtered environments: one when this condition holds to analyze the positive cases (including actions), and the other one when this condition does not hold to analyze the **elseif** and **else** cases. A **synchro** transition is well clocked if its associated actions and fork are in the environment filtered on instants when this transition is fired.
- A transition is causal if its guard is and if its fork part is. This transition defines a causal context representing the dependencies on all the guards that appear in it (forked or not). This context is used to analyze any actions in this transition, meaning that a variable defined in this transition depends on all the guards of the forked transition. A **synchro** transition is causal if its fork part is, defining thus a causal context in which the associated actions should be causal. In other words, any variable defined in such a transition depends on all the guards of the forked transition.
- The conditions of a transition must all be defined at the initial cycle where they are activated. Its associated actions must be well initialized.

## DYNAMIC SEMANTICS

The evaluation of a transition starts with the evaluation of its initial Boolean condition. If it evaluates to **false**, nothing else is undertaken. If it evaluates to **true**, the following conditions of its fork part, when it exists, are all evaluated in parallel following a classical data-flow semantics. This implies that if these conditions contain an internal state (*i.e.*, when using the **times** operator), this state is modified accordingly. Since the **else** part is not mandatory, a combination of fork transition may be fireable after evaluation of its conditions. If one of these conditions evaluates to **true**, this transition becomes fireable. If it is fired (*i.e.*, if it the first one being fireable in the textual order), its corresponding action is evaluated. The evaluation of a fork transition is thus equivalent to the evaluation of a list of transitions with conditions derived from the conditions involved in the fork.

Firing a transition contributes to defining the next selected state of the automaton.

- When a fired transition specifies a target state with **restart**, then this state (body, conditions and actions on strong and weak transitions) are reset.
  - If this transition is strong, then the body and weak part of the target state are reset during the same cycle, while its strong part will be reset at the next cycle.
  - If this transition is weak, all parts of the target state are reset at the current cycle.

Resetting a state means that this state is considered being again in its initial state. This means that a **fb**y or a **->** operator evaluates its left argument. However, a reference to a previous value with **pre** (resp. a last value with **last**) evaluates to the last local value (resp. the last global value). In particular, the **last** declaration, if any, is not taken into account any more.

- On the contrary, when the target state is specified with **resume**, the internal state, if any, is kept.

#### RELATED TOPICS

- Section 8.2 [“Sequential Operators”](#)

### 7.3.3 Actions

#### SYNTAX

---

```
actions ::= do { [[ emit ]] emission_body
               {{ ; [[ emit ]] emission_body }} }
          | do data_def
```

---

#### STATIC SEMANTICS

An action in a transition is either an non-empty list of signal emission surrounded by braces and separated by a semicolon, or a complete scope declaration. Signals emitted in an action must be valid identifiers in the current environment. The static analysis of actions is similar to that of scope declarations (see section 6.4 [“Scope Declarations”](#)).

#### DYNAMIC SEMANTICS

An action is evaluated only if it correspond to a fired transition: the first one in the textual order for which a condition can evaluate to **true**.

#### RELATED TOPICS

- Section 7.1 [“Equations”](#) about signal emission
- Section 6.4 [“Scope Declarations”](#)



## 7.3.4 Examples

**Example 1:** This first example browses the full syntax available within automaton: strong and weak transitions, restart and resume, flows defined in state or in transitions, local scopes.

---

```

node StateMachine_1(bI1: bool)
returns (bO1: bool default = true; iO2: int16 default = 0; bO3: bool default = false)
let
  automaton SM1
  initial state ST1
  unless if bI1 resume ST2;
  sig
    sig1;
  var
    iV1: int16;
  let
    iV1 = 10;
    emit 'sig1;
    bO1 = 'sig1;
    iO2 = iV1 -> pre iO2 + 2;
  tel

  state ST2
  sig
    sig1;
  var
    bV1: bool;
  until if true do let emit 'sig1;
    bV1 = 'sig1;
    bO3 = bV1;
    tel
    restart ST1;

  returns ..;
tel

```

---

**Example 2:** This example illustrates some of the namespace rules that allow to overload an identifier.

---

```

type
  SM1 = uint16;
node StateMachine_012(iI1: int32)
returns (iO1: int32)
let
  automaton SM1
  initial final state ST1
  sig
    SM1;
  var
    ST1: int32;
  let
    ST1 = iI1;
    emit 'SM1;
    iO1 = ST1 + 1;
  tel
  returns ..;
tel

```

---

**Example 3:** Variable `bv2` declared in the scope of state `ST1` can be used in the condition of the weak transition, but could not be used in state `ST2` where it is unknown. On the contrary, the strong transition of state `ST2` mentions `bv1`. `bv1` is a shared variable between the states. As it is defined in state `ST1`, it must be also defined in state `ST2`. As there is no definition in state `ST2`, a default one is provided which is: `bv1 = last 'bv1`. Here this leads to a causality error between the definition of variable `bv1` in state `ST2` and its use by the strong transition. A possible correction of this error is given by the node on the right example. It consists in referring only the past value of `bv1` instead of its current value, either through a `last` operator (which does not need an initialization since `ST2` is not accessible by a strong transition from initial state `ST1`), or through a `pre` correctly initialized (since condition must be locally well initialized). Of course the dynamic semantics of these two possible corrections differs: the first one refers to the previous global value of `bv1`, while the second to the previous local one.

First example

```
node StateMachine_062 (iI1: int32)
returns (iO1: int32)
var bv1: bool;
let
  automaton SM1
  initial state ST1
  var bv2: bool;
  let
    bv2 = false -> not pre bv2;
    bv1 = iI1 <> 0;
    iO1 = iI1 * 2;
  tel
  until if bv1 and bv2 restart ST2;

  state ST2
  unless if bv1 resume ST1;
  returns ..;
tel
```

Second example

```
node StateMachine_062 (iI1: int32)
returns (iO1: int32)
var bv1: bool;
let
  automaton SM1
  initial state ST1
  var bv2: bool;
  let
    bv2 = false -> not pre bv2;
    bv1 = iI1 <> 0;
    iO1 = iI1 * 2;
  tel
  until if bv1 and bv2 restart ST2;

  state ST2
  unless if last 'bv1 resume ST1;
  returns ..;
tel
```

**Example 4:** This example illustrates the use of the **default** declaration. Instead of maintaining the last value of the variables when the control is in a state where there is no definition for this variable, the **default** expression is evaluated.

```

node StateMachine_073(iI1: int16; bI2: bool)
returns (iO1: int16 default = 10; iO2: int16 default = 5 * iO1)
let
  automaton SM1
  initial state ST1
  unless if bI2 resume ST2;
  let
    iO1 = iI1;
  tel
  until if true restart ST2;
  state ST2
  let
    iO2 = 0 -> pre iO1 + iI1;
  tel
  until if true restart ST1;
  returns ..;
tel

```

iI1	1	1	1	2	0	...
bI2	false	false	true	false	true	...
Active	S1	S2	S2	S1	S2	...
iO1	1	10	10	2	10	...
iO2	5	0	11	10	10	...

**Example 5:** Variable **iO1** cannot be defined both in the transitions of **ST1** and in the body of **ST2**. Variables defined in states and in transitions must be exclusive.

```

node StateMachine_078(iI1: int32; bI2: bool)
returns (iO1, iO2: int32)
let
  automaton
  initial state ST1
  unless if bI2 do iO1 = iI1 + 1; restart ST2;
  until if true do iO1 = 10; restart ST2;
  state ST2
  unless if bI2 do iO2 = 0; restart ST1;
  iO1 = 0;
  until if true do iO2 = -10; restart ST1;
  returns ..;
tel

```

**Example 6:** The filtering performed when entering the state of a State Machine prevents using variables already sampled on an outside clock, here `iI1` and `iO1`. Moreover, the clock status of `bI2` being lost, local variable `iV1` is then clocked on an invalid clock identifier.

---

```
node StateMachine_112(iI1: int32 when bI2; clock bI2: bool)
returns (iO1: int32 when bI2)
let
  automaton
    initial state ST1
    var iV1: int32 when bI2;
    let
      iV1 = (0 when bI2) -> pre iV1 + 1 when bI2;
      iO1 = iI1 -> pre iV1;
    tel
    until if bI2 restart ST1;
    returns iO1;
  tel
```

---

**Example 7:** A variable declared within the scope of a state or a transition cannot be used outside this scope. Here, variable `bV1` being declared in the weak transition of state `ST1`, it cannot be used as the condition of this transition. Notice also that state `ST2` would have been a trash state since it is not possible to get out of it.

---

```
node StateMachine_063(iI1: int16)
returns (iO1: int16)
let
  automaton SM1
    initial state ST1
    let
      iO1 = iI1 * 2;
    tel
    until if bV1 do
      var bV1: bool;
      let
        bV1 = iI1 > 10;
      tel
      restart ST2;
      state ST2
      returns ..;
    tel
  tel
```

---

**Example 8:** Due to a lack of definition and **default** declarations, variable `iO2` is maintained when state `ST2` is active. Notice that this state is entered through a **weak** restart transition, but is well initialized even though no **last** declaration is made for this variable. Indeed, a **restart** transition does not require a **last** declaration for the variables called through a **last** operator. It evaluates a call to the last value as usual. restart transition only evaluates ->

and **fb**y operators to their initial value. The initialization would have been wrong in case of a strong transition from ST1 to ST2. Indeed, global initialization is required on the initial state and all the states reachable by a strong transition from it.

---

```
node StateMachine_122 (iI1: int32; bI2: bool)
returns (iO1: int32; iO2: int32)
let
  automaton
  initial state ST1
  let
    iO1 = iI1;
    iO2 = iI1 * 2;
  tel
  until if bI2 restart ST2;

  state ST2
  let
    iO1 = 1 -> - iI1;
  tel
  until if true restart ST1;
  returns ..;
tel
```

---

**Example 9:** The restart transitions are asynchronous. When entering state ST2, every flow in this state is restarted. This implies that the value of variable *z* in state ST2\_2 always evaluates to 2 while entering this state. Contrary to the example above, this variable

being local to state `ST2`, it is restarted after the `restart` transition. A lack of `last` declaration for this variable would lead to an initialization error. Moreover, since state `ST2_2` must also be restarted while entering automaton `A2`, `y` evaluates to 0.

---

```

node N() returns (x, y: int32 last = 0)
let
  automaton A
    initial state ST1
    let tel
      until if true restart ST2 ;

  state ST2
    var z: int32 last = 2;
    let
      automaton A2
        initial state ST2_1
        let
          z = 3 + last 'z;
          x = last 'x + 1;
        tel
          until if x mod 2 = 0 resume ST2_2;

    state ST2_2
      let
        x = last 'x - 1;
        y = 0 -> pre z - 1;
      tel
        until if true resume ST2_2 ;
    returns .. ;
  tel
    until if true resume ST1 ;
  returns .. ;
tel

```

---

Active	ST1	ST2_2	ST1	ST2_1	ST1	ST2_2	ST1	ST2_1	...
x	0	1	1	2	2	1	1	2	...
y	0	0	0	0	0	4	4	4	...
z		2		5		5		8	...

**Example 10:** This example illustrates the semantical equivalence between automaton and conditional blocks. The following automaton.

---

```

node N(...) returns (...)
let
  automaton
    initial state ST0
    unless if c0 resume ST1;
    <eq0>;
    state ST1
    <eq1>;
    until if c1 resume ST0;
    returns ..;
tel

```

---

It can be translated into a conditional block as follows. The flow `state_sel` carries the notion of selected state, `state_act` that of activated state, and `state_next` that of the next cycle's selected state. The equation defining `state_act` uses `state_sel` and the strong transition part of the automaton's states. The equation defining `state_next` uses `state_act` and the body and weak part of the automaton. These variable belong to an enumerated type that represents the states of the automaton (one enumerator per

state). Since only one transition must be fired during a cycle, the knowledge of already fired strong transition in the selected state must be used before firing a weak transition in the active one. Therefore, a type enumerating all possible transitions is used.

---

```

type A_states = enum {ST0, ST1};
    A_trans = enum {no_trans, ST0_strong1, ST1_weak1};
node N(...) returns (...)
var
    state_sel, state_act, state_next: A_states;
    fired_strong: A_trans;
let
    state_sel = ST0 -> pre state_next;
    activate when state_sel match
    |ST0 : let
        state_act = if c0 then ST1 else ST0;
        fired_strong = if c0 then ST0_strong1 else no_trans;
        tel
    |ST1 : let
        state_act = ST1;
        fired_strong = no_trans;
        tel
    returns state_act, fired_strong;
    activate when state_sel match
    |ST0 : let
        <eq0>;
        state_next = ST0;
        tel
    |ST1 : let
        <eq1>;
        activate if fired_strong <> no_trans
        then state_next = ST1;
        else state_next = if c1 then ST0 else ST1;
        returns ..;
        tel
    returns state_next;
tel

```

---

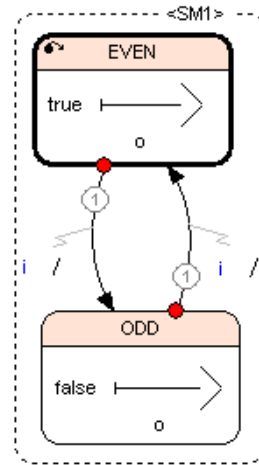


**Example 11:** A single state machine with two states and data flow definitions in states.

```

automaton SM1
  initial state EVEN
  unless if i restart ODD;
  let
    o = true;
  tel
  state ODD
  unless if i restart EVEN;
  let
    o = false;
  tel
  returns o ;

```

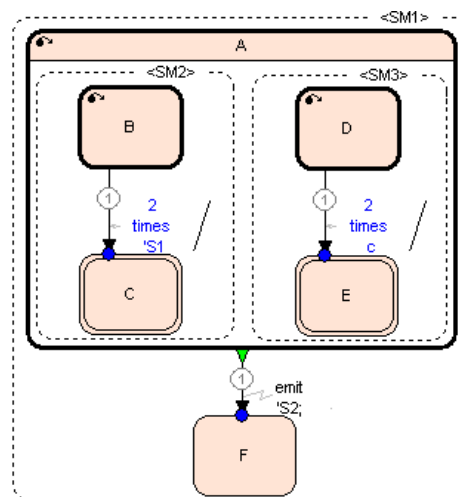


**Example 12:** SM1 has a macro state A which includes two state machines in parallel. Notice the synchronization transition noted `synchro` in source code.

```

automaton SM1
  initial state A
  let
    automaton SM2
      initial state B
      until if 2 times 'S1 restart C;
      final state C
      returns .. ;
    automaton SM3
      initial state D
      until if 2 times c restart E;
      final state E
      returns .. ;
  tel
  until
    synchro do let emit 'S2; tel restart F;
  state B
  returns .. ;

```

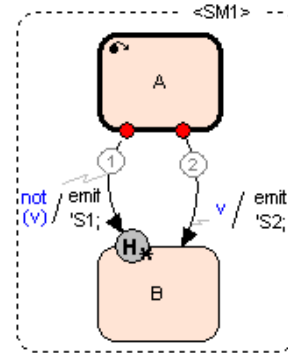


**Example 13:** State B can be started on condition `v`, or resumed by history on condition `not v`.

```

automaton SM1
  initial state A
  unless
    if not v do let emit 'S1; tel resume B;
    if v do let emit 'S2; tel restart B;
  state B
returns .. ;

```

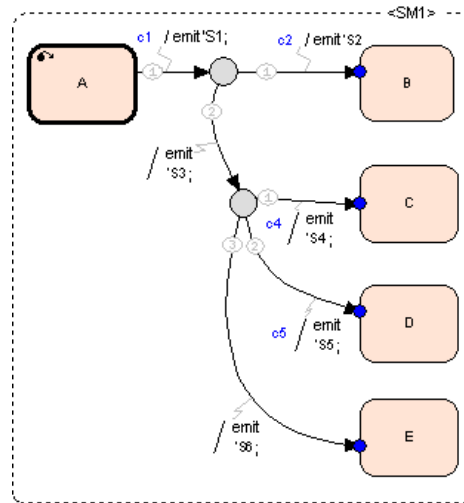


**Example 14:** The following state machine illustrates the use of fork transitions.

```

automaton SM1
  initial state A
  until
    if c1 do let emit 'S1; tel
    if c2 do let emit 'S2; tel restart B
  else do let emit 'S3; tel
    if c4 do let emit 'S4; tel restart C
  elsif c5 do let emit 'S5; tel restart D
  else do let emit 'S6; tel restart E
  end
end;
state B
state C
state D
state E
returns .. ;

```



# 8

## Expressions

This chapter describes the operators and constructs available in the language to combine flows.

- 8.1 [“Basic Expressions”](#)
- 8.2 [“Sequential Operators”](#)
- 8.3 [“Combinatorial Operators”](#)
- 8.4 [“Operations on Arrays and Structures”](#)
- 8.5 [“Operator Application and Higher-Order Patterns”](#)
- 8.6 [“Primitive Operator Associativity and Relative Priority”](#)

Expressions are the basic blocks of the Scade language. They allow to combine flows simply by declaring what should be done during each cycle.

### SYNTAX

---

```
expr ::= id_expr
      | atom
      | list_expr
      | tempo_expr
      | arith_expr
      | relation_expr
      | bool_expr
      | array_expr
      | struct_expr
      | mixed_constructor
      | switch_expr
      | apply_expr
      | depend_expr
```

---

### STATIC SEMANTICS

Expressions are built from valid identifiers, that may represent either constant flows or variable ones, atoms belonging to the basic data types available in Scade and a bunch of predefined operators along with user defined ones. The static semantics of each kind of expressions is given in their respective section of this manual.

## DYNAMIC SEMANTICS

Expressions in the Scade 6 language are defined on flows. Two kinds of operators can be distinguished:

- 1 A *sequential* operator produces values that also depend on previous (in the cycle sequence sense) inputs.
- 2 An operator is *combinatorial* if the values that it produces at any cycle only depends on the inputs of the operator at this cycle.

The semantics of a combinatorial operator can be exhaustively described by its meaning at a given cycle, then point-wisely extending it to flows. On the contrary, sequential operators may need to refer to an arbitrary ancient value in order to precisely define their semantics.

## 8.1 Basic Expressions

- 8.1.1 [“Identifiers”](#)
- 8.1.2 [“Atoms”](#)
- 8.1.3 [“Lists”](#)

### 8.1.1 Identifiers

#### SYNTAX

---

```
id_expr ::= path_id
         | NAME
         | last NAME
```

---

#### STATIC SEMANTICS

- A path identifier must refer a valid identifier in the package hierarchy. When this identifier is not qualified by a path, it can belong either to the local scope (input, output or local variable, excluding signals identifiers), or to the global scope (constant or sensor). A name is an identifier preceded by a quote. It must refer to a valid signal identifier belonging to the local scope. On the contrary, when used by a **last** primitive, the identifier must refer to any input, local, or output variable in the local scope.

- Whenever the namespace analysis confirms the validity of an identifier, this ensures that this identifier has been correctly declared (see section 6.4 [“Scope Declarations”](#) or 6.2 [“Variable Declarations”](#)). This declaration gives the identifier a type, a clock, an empty dependency type, and an undefined initialization type.
- A name used as a signal presence is considered as a Boolean. It has the clock of the context it is used in, such as global flows. A signal does not have a dependency type. However, it contributes to the causal types of the variables that make use of its presence. Finally, a signal presence is always well-initialized.
- A quoted identifier used in a **last** primitive must be a named flow: it cannot be a group identifier nor an expression as for **pre** operator. Applying this operator keeps the same flow type and clock type as the identifier.

By definition of the causality analysis, the dependencies of a defined variables is the set of variables whose value at the current cycle is needed to compute this former variable. Since the **last** primitive is used to recall a previous value (see the Dynamic Semantics below), the quoted identifier is not added to the causal type of the variable currently defined. However, since the initial value of a **last** application can be given (when it exists) by the **last** part of the declaration of this identifier, the variables involved in this declaration are added to the current causal type.

A **last** application is well-initialized when a **last** declaration has been made for the identifier at use and when the identifier is well-defined at its first instant of activation. It results in the same type as that of the corresponding expression. A **last** application is also well-initialized when it occurs in a non-initial state (and its available transitions) of a State Machine non reachable by a strong transition from the initial one.

## DYNAMIC SEMANTICS

When a flow identifier refers to a constant, it evaluates to the corresponding constant value. In case of a sensor or an input variable, its value is the value of this flow at the current cycle. A local or output identifier has the same value as the expression that defines it at the current cycle.

A signal presence evaluates to **true** if this signal is emitted at least one in the current cycle. It evaluates to **false** otherwise.

The **last** primitive evaluates to the latest value of the flow at use. This primitive coincides with the **pre** operator outside control blocks. Within a control block though, the recalled value is the last one computed, whatever the control state was concerned.

**Example:** The following example illustrates the semantic differences between **last** and **pre**

<pre>node UpDownPre() returns(x:int16) let   automaton SSM     initial state Up     let       x = 0 -&gt; pre(x) + 1;     tel     until if (x&gt;=3) resume Down;   state Down   let     x = 2 -&gt; pre(x) - 1;   tel   until if (x&lt;=-3) resume Up;   returns x; tel</pre>	<pre>node UpDownLast() returns(x:int16) let   automaton SSM     initial state Up     let       x = 0 -&gt; last 'x + 1;     tel     until if (x&gt;=3) resume Down;   state Down   let     x = last 'x - 1;   tel   until if (x&lt;=-3) resume Up;   returns x; tel</pre>
--	---

UpDownPre()	0	1	2	3	2	-1	-2	-3	4	-4	5	-5	6	-6	7	...
UpDownLast()	0	1	2	3	2	-1	-2	-3	-2	-1	0	1	2	3	2	...

**RELATED TOPICS**

- Section 3.2 [“Packages”](#)
- Section 6.2 [“Variable Declarations”](#)
- Section 7.3 [“State Machines”](#)

## 8.1.2 Atoms

### SYNTAX

---

```

atom ::= bool_atom
      | CHAR
      | INTEGER
      | FLOAT
      | TYPED_INTEGER
      | TYPED_FLOAT
bool_atom ::= true
          | false

```

---

### STATIC SEMANTICS

Atoms are uniquely defined and cannot be overloaded by any variable identifier. Characters are single symbols (digits, letters, or usual keys) surrounded by quotes. Numerical atoms are sequences of digits following the usual syntactical rules. Boolean atoms are referred to using **true** and **false**.

Character and Boolean atoms are typed directly with the data type they represent. There are two kinds of numeric literals: typed literals (*eg.*, `1_i32`), whose type is given by their suffix, and untyped literals (*eg.*, `1`), whose type is inferred from the context.

For instance, in the expression `x+1`, the literal `1` has the same type as `x`, given that it is a predefined numeric type, an imported numeric type, or a type variable constrained by a **where** declaration. An expression is ill-typed if the type of a literal cannot be inferred from the context, except in size expressions where it always has the internal **size** type.

Decimal integer literals can have any numeric type (integer or float), while other integer literals (binary, octal, hexadecimal) can have any integer type. Similarly, a float literal can have any float type. An integer literal can have a predefined integer type if it is within the range of the type. No such restriction applies to imported integer types or other numeric types. The clock of atoms fits the context they are used in: whatever sampling operations have been made, an atom is always well-clocked. Atoms do not introduce any dependency constraint. Finally, atoms are always well-defined at the first cycle.

## DYNAMIC SEMANTICS

Atoms are constant flows: their value do not change during the sequence of execution.

### Example 1: Decimal literal with float type

---

```
function correct_incr (i: float64) returns (o: float64)
  o = i + 1;
```

---

### Example 2: Incorrect hexadecimal literal with float type

---

```
function incorrect_incr (i: float64) returns (o: float64)
  o = i + 0x1;
```

---

### Example 3: Incorrect unconstrained literal

---

```
function incorrect (i: float64) returns (o: float64)
  o = if 6 > 7 then i else 0.0;
```

---

### Example 4: Incorrect integer literal outside of range

---

```
function incorrect_range (i: int8) returns (o: int8)
  o = i + 128;
```

---

## RELATED TOPICS

- Section 4.2 [“Type Expressions”](#)
- Section 2.2 [“Lexemes”](#)
- Chapter 6 about [“User-Defined Operators”](#)

## 8.1.3 Lists

### SYNTAX

---

```
list_expr ::= ( list )
  list ::= [[ expr {{ , expr }} ]]
```

---

### STATIC SEMANTICS

A list of expressions gathers a possible empty list of expressions surrounded by round braces and separated by commas. The static semantics of a list of expressions is the product type of the semantics of the expressions. Nested lists are always considered only under their flattened form. In case of an empty list, its type is the empty type '()', its clock type is the clock of the context it is defined in, it does not add any dependency, and is well-initialized. A list of expressions can be associated with a group



type (see section 4.3 [“Group Declarations”](#)). In this case, it is required that the clock of all components must be identical. Only a few primitive operators can be applied on lists of expressions, as shown in [Table 4.2](#). On the contrary, any user defined operator can be applied to a list of expression.

### DYNAMIC SEMANTICS

The expression list is not an operation on flows in the sense that it does not produce a flow, it is just a syntactic way to group flows.

**Example 1:** Structural equality does not apply to lists of expressions. The following example is erroneous

---

```
node ex1(x, y: int64) returns (z: int64)
let
  z = 0 -> if (x, y) = (pre x, pre y)
    then pre x + last 'y
    else 0;
tel
```

---

**Example 2:** User operators can be easily fed with list of expressions

---

```
group G = (int64, int64, bool);
function f(a, b: int64; c: bool) returns (x, y: int64) ...;
function g(a, b: int64) returns (x: int64) ...;
node ex(c: bool; a, b: int64; ...) returns (s, t: int64; ...)
var l: G;
let
  l = (a+b, a-b, c);
  s, t = if c then f(l) else (0, g(f(b, a, not c)));
tel;
```

---

### RELATED TOPICS

- Section 4.3 [“Group Declarations”](#)

## 8.2 Sequential Operators

Temporal primitive operators are sequential operators: their semantics depends on past values of the flows they apply to. Consequently, these operators cannot be used in the expression associated with an internal constant declaration.

### SYNTAX

---

```
tempo_expr ::= pre expr
            | expr -> expr
            | fby ( list ; expr ; list )
            | expr times expr
            | expr when clock_expr
            | merge ( expr ; list {{ ; list }} )
```

---

### STATIC SEMANTICS

Except the **times** primitive, all these operators can be manipulated by external objects and lists of expressions.

- The **pre** operator is the basic operator allowing to refer a past value of an expression. It corresponds to the usual concept of memory in block diagram descriptions.

This operator applies on a list of expressions, or a single expression, of any type and builds an object of the same type. Using this operator in an expression introduces a memory (requiring thus to be used in a **node** instead of a **function**). This operator does not affect the clock of its input. According to the definition of the causality analysis, identifiers used in the input of a **pre** operator are not considered as adding dependencies. Given that the input expressions are well-initialized, the resulting application is not defined at its first cycle of activation.

- Combined with the previous operator, the **->** allows to build deterministic applications, since it allows to fix the initial value of flows. This combination can be nested in order to define these values on several beginning instants.

It applies either on lists of expressions or single ones, given that the two arguments have the same number of expressions. The resulting expression builds an object of the same type, and introduces a memory. The **->** operator does not affect the clock of its input arguments. These arguments must be based on the same clock expression.

Both arguments contribute to the causality type of the current variable under definition. The resulting expression is well-initialized if its arguments are, and keeps the delay type of its first argument.

- The combination of `->` and `pre` can be equivalently written using the `fbby` operator. This translation is given in the dynamic semantics part.

The `fbby` operator applies to a list of expressions, possibly omitting the round braces, an integer size argument, and another list with the same number of items. This operator thus has a variable number of arguments, depending on the length of the lists involved. Its size argument must evaluate to a strictly positive value. The resulting expression introduces a memory. This operator does not affect the clock of its arguments. It introduces dependencies only on the variables involved in its initial values (*i.e.*, its third argument). Indeed, its first argument being wrapped in a `pre` operator, all the variables involved in it are ignored. It requires both its arguments to be well-defined at the first instant and produces in return a well-defined expression. Note that it is different from the `->` operator which allows its left argument to introduce a delay.

- The `times` operator implements an often used construction based on the occurrences of the same event several times until a decision is taken.

This operator applies only on a single signed integer expression and a single Boolean one, and results in a Boolean expression. The first argument is either a size expression or can have any signed integer type. This operator can be applied neither on imported objects nor on lists of expressions. This operator introduces a memory. It does not affect the clock of its inputs. Both its arguments are adding dependencies during the causality analysis. It requires well-defined arguments at the initial instant where they are activated and produces itself a well-defined expression.

- The `when` operator is the basic filtering operator, allowing to consider only representative instants of a flow, or to combine subsystems sampled on different rates, synchronizing them on the slowest clock.

It applies on any expression or list of expressions as its first argument, and on a valid clock expression as its second one. It builds an object having the same data type as its first argument. This operator does not introduce a memory. Given the clock on which its second argument is based on, its first argument must be based on the same one (in case of a list of expressions, they are all required to be based on the same one). This expression has the same clock than (res. product of the same clock) the

clock its second argument defines. Both arguments are taken into account during the causality analysis. The clock expression has to be well-defined at its first instant of activation. The delay type of the result expression is that of its first argument.

- The **merge** operator allows to build a fast flow given complementary slower ones. It uses the clock calculus to rigorously implement an imperative **if\_then\_else** construct into the data-flow paradigm. Its first argument must be a statically finite enumerated type (including Booleans, but not characters). This type must have at least two distinct enumerators. The first argument must be a valid clock identifier declared as such using the **clock** keyword. It then takes as further input as many expressions as the number of enumerators in this type. These arguments must all have the same data type, which can be any available data type, including imported types or groups. The resulting expression has the same type as these arguments. These following arguments must constitute an exhaustive partition of the instants when the first argument is defined. Every arguments are taken into account for causality analysis: the resulting expression depends on all its inputs. Finally, it requires all its arguments to be well-defined at their first instant and produces a well-defined expression.

#### DYNAMIC SEMANTICS

- Operator **pre** is a sequential primitive that shifts flows on the last instant backward when this flow was defined within the same scope. It thus produces an undefined value at its first instant of activation called *nil*.

a	a1	a2	a3	a4	...
<b>pre</b> a	<i>nil</i>	a1	a2	a3	...

- The **->** primitive evaluates its left argument at its first instant of evaluation or after a **restart**, and its right argument otherwise.

a	a1	a2	a3	a4	...
b	b1	b2	b3	b4	...
a -> b	a1	b2	b3	b4	...

- The flow **fby(b;n;a)** combines the first two primitives in order to access previous values and produce only well-initialized flows. It can be equivalently defined by:

$$\text{fby}(b; n; a) = a \rightarrow \text{pre } \text{fby}(b; n-1; a) = \underbrace{a \rightarrow \text{pre } (a \rightarrow \text{pre } (\dots (a \rightarrow \text{pre } b) \dots))}_{n \text{ times}}$$

<b>fby</b> (b; 1; a)	a1	b1	b2	b3	...
<b>fby</b> (b; 2; a)	a1	a1	b1	b2	...
<b>fby</b> (b; 3; a)	a1	a1	a1	b1	...

It then evaluates its third argument on the first  $n$  cycles (or the first  $n$  cycles after a **restart**), and its first one on the remaining cycles.

- times** is not primitive in the sense that it can be defined with the data-flow kernel of the language. The node below defines the behavior of this operator:

```
node times_behavior (n : 'a; c : bool) returns (o : bool) where 'a signed
var
  v3, v4 : 'a;
let
  v4 = n -> pre (v3);
  v3 = if (v4 < 0)
    then v4
    else (if c then v4 - 1 else v4);
  o = c and (v3 = 0);
tel
```

This operator takes a value  $n$  at the first instant (or after a **restart**) and then counts down each time  $c$  is true. When the internal counter reaches zero, its output is true for one cycle and then becomes false.

- when** takes two arguments, a flow and a clock expression. These two arguments are evaluated at every cycle. If the second evaluates to **true**, then the result is equal to the result of the evaluation of its first argument:

$$(x \text{ when } h)_n \begin{cases} \text{no value} & \text{if } h = \text{false} \\ x_n & \text{if } h = \text{true} \end{cases}$$

x	x1	x2	x3	x4	x5	...
h	false	true	true	false	true	...
x when h		x2	x3	b4	x5	...

The clock expressions corresponds to different syntactical identifiable ways to specify the filtering flow.

- **merge** takes, as first argument, a clock identifier  $h$  that is used to select one of its other inputs. Suppose that  $h$  belongs to a finitely enumerated type whose enumerators are  $\{e^1, \dots, e^p\}$ , the following equation holds:

$$(\text{merge}(h; x^1; \dots; x^p))_n \mid \begin{array}{l} x_n^1 \text{ if } h \text{ match } e^1 \\ x_n^p \text{ if } h \text{ match } e^p \end{array}$$

In case  $h$  is Boolean,  $e^1 = \text{true}$  and  $e^2 = \text{false}$ . This operator evaluates the value of its clock expression. Depending on this value, the corresponding expression is defined at the current cycle; its value results from its evaluation.

$h$	true	true	false	true	false	false	...
$a$	a1	a2	a3	a4	a5	a6	...
$b$	b1	b2	b3	b4	b5	b6	...
$a \text{ when } h$	a1	a2		a4			...
$b \text{ when not } h$			b3		b5	b6	...
$\text{merge } (h; a \text{ when } h; b \text{ when not } h)$	a1	a2	b3	a4	b5	b6	...

#### RELATED TOPICS

- Section 4.2 [“Type Expressions”](#) about size expressions
- Section 6.3 [“Clock Expressions”](#)
- Chapter 6 about [“User-Defined Operators”](#)

## 8.3 Combinatorial Operators

- 8.3.1 [“Boolean Operators”](#)
- 8.3.2 [“Arithmetic Operators”](#)
- 8.3.3 [“Relational Operators”](#)
- 8.3.4 [“Flows Switches”](#)

### 8.3.1 Boolean Operators

#### SYNTAX

---

```

bool_expr ::= not expr
           | expr bin_bool_op expr
           | # ( list )
bin_bool_op ::= and | or | xor

```

---

#### STATIC SEMANTICS

Boolean operators do not introduce an implicit memory in the user operator they are defined in. They all only apply on single Boolean flows. Boolean operators do not affect the clock of their arguments. They all add dependencies from their inputs. These operators are well-initialized given that their arguments are. They produce an expression which has the same initialization type as the maximum delay of their arguments.

#### DYNAMIC SEMANTICS

- The **not** operator is defined by the following truth table:

a	false	true
<b>not a</b>	true	false

- The **or** operator is defined by the following truth table:

a	false	false	true	true
b	false	true	false	true
<b>or</b>	false	true	true	true

- The **and** operator is defined by the following truth table:

a	false	false	true	true
b	false	true	false	true
<b>and</b>	false	false	false	true

- The **xor** operator is defined by the following truth table:

a	false	false	true	true
b	false	true	false	true
<b>xor</b>	false	true	true	false

- The output of the **# (exclusive)** operator is false when at least two of its inputs are true. It is true otherwise. More formally, let *int\_of\_bool* be the function that maps true with 1 and false with 0, then:

$$\#(e_1, \dots, e_n) = \left( \sum_{i=1}^n \text{int\_of\_bool}(e_i) \right) \leq 1$$

## 8.3.2 Arithmetic Operators

### SYNTAX

```
arith_expr ::= unary_arith_op expr
            | expr bin_arith_op expr
            | ( expr : type_expr )
unary_arith_op ::= - | + | lnot
bin_arith_op ::= + | - | * | / | mod | land | lor | lxor | lsl | lsr
```

### STATIC SEMANTICS

The numeric cast operation ( $e : t$ ) requires that the type of  $e$  and the type  $t$  be numeric. The expression  $e$  is given type  $t$  by default if its type is unconstrained (e.g., for a literal).

Arithmetic operators do not introduce any memory. They all apply on flows, not on list of flows. Arithmetic operators take inputs and return outputs of the same type, except for shift operators. Input flows must be integers, float, or any of them in case of polymorphic operators. Bitwise operators **lnot**, **land**, **lor**, and **lxor** require



unsigned integers. Shift operators `lsl` and `lsr` require a first input of an unsigned type and return a value of the same type. Their second argument can have any integer type. If this type is unconstrained, then it is given the type of the first argument.

Arithmetic operators do not affect the clock of their arguments. They all add dependencies from their inputs. These operators are well-initialized given that their arguments are. They produce an expression which has the same initialization type as the maximum delay of their arguments.

### DYNAMIC SEMANTICS

The construction  $(e : \tau)$  casts the numeric expression  $e$  to the numeric type  $\tau$ . These arithmetic operators are pointwise extensions of the usual arithmetic ones.

<code>-</code>	unary minus
<code>+</code>	unary plus
<code>+.+</code>	sum
<code>.-.</code>	difference
<code>.*.</code>	multiplication
<code>.mod.</code>	rest of the division between two integer expressions
<code>./.</code>	quotient of the division between two numerical expressions
<code>lnot</code>	bitwise not of an unsigned integer
<code>.land.</code>	bitwise and of unsigned integers
<code>.lor.</code>	bitwise or of unsigned integers
<code>.lxor.</code>	bitwise xor of unsigned integers
<code>.lsl.</code>	logical left shift on an unsigned integer
<code>.lsr.</code>	logical right shift on an unsigned integer

The semantics of arithmetic operators is considered abstract. It is made concrete by the tool implementing the language. It is however constrained as follows:

- Arithmetic operators on predefined integer types shall be defined as the usual arithmetic operators in the range of the type. The range of `signed<<N>>` = `intN` is  $[-2^{N-1} \dots 2^{N-1} - 1]$ . The range of `unsigned<<N>>` = `uintN` is  $[0 \dots 2^N - 1]$ . This applies if the arguments and result of the operator are within the range of the type, with the following exceptions:

- The `/` and `mod` operators shall be defined respectively as the quotient and remainder of the integer division if their arguments are integers of the same sign in the range of the type and the result is also in the range of the type.
- The shift operators on **unsigned** `<<N>>= uintN` shall be defined as the usual operators if the first argument is in the range of the type and the second argument is in the inclusive range 0 to  $N-1$ .
- Arithmetic operators are never constrained for float and imported numeric types. The semantics is then completely defined by the tool.
- Numeric casts from a predefined integer type to another predefined integer type shall be defined as the identity if the input is in the range of the target type. The semantics of all other numeric casts is abstract.

No properties are assumed for arithmetic operators (like associativity, commutativity, distributivity, neutral elements, etc). No symbolic computation or rewriting is done by the tool.

**Example:**      Cast operator used to constrain a literal type

---

```
function correct (i: float64 ) returns (o: float64 )  
  o = if (6: int8 ) > 7 then i else 0.0;
```

---

### 8.3.3 Relational Operators

#### SYNTAX

---

```
relation_expr ::= expr bin_relation_op expr
```

```
bin_relation_op ::= = | <> | < | > | <= | >=
```

---

#### STATIC SEMANTICS

Relational operators do not introduce an implicit memory. They all apply on flows, but not on list of flows, nor on objects of an external data type. Input flows for = and <> operators must be of any predefined type. Other relational operators require their arguments to be numeric ones only (integer or float).

Relational operators do not affect the clock of their arguments. They all add dependencies from their inputs. These operators are well-initialized given that their arguments are. They produce an expression which has the same initialization type as the maximum delay of their arguments.

#### DYNAMIC SEMANTICS

- .=. structural equality between any type of values
- .<>. structural inequality between any type of values
- .<=. lower or equal on numerics
- .<. lower on numerics
- .>=. greater or equal on numerics
- .>. greater on numerics

When applied to structured values, equality and inequality are not extended to produce a structured result, the result is a single Boolean value representing the conjunction of the equalities performed on each component of the structured arguments. The semantics of numeric comparisons is abstract. It shall coincide with the usual comparison for predefined integer types if the inputs are within the range of the type.

### 8.3.4 Flows Switches

#### SYNTAX

---

```

switch_expr ::= if expr then expr else expr
              | ( case expr of {{ case_expr }}+ )
case_expr  ::= | pattern : expr
pattern    ::= path_id
              | CHAR
              | [[ - ]] INTEGER
              | [[ - ]] TYPED_INTEGER
              | bool_atom
              | _

```

---

#### STATIC SEMANTICS

- The **if\_then\_else** operator takes three arguments on input. The first one has to be a single Boolean expression (no structured expressions, no lists), the two other can be any expression of any type, including imported types and lists of expressions. These last two expressions must share the same type, and the resulting expression builds an object belonging to this type. If these expressions are lists of expressions, the first argument is taken into account for any component of the list, so as to form a list of **if\_then\_else** constructs. Given the clock of the first argument, the two others have to be completely based on the same clock. An **if\_then\_else** expression depends on its three arguments; it is well-initialized if its arguments are. Its resulting initialization type is the maximum delay of its three types.
- The **case** operator takes a variable number of arguments, depending on the number of patterns explored. Its first argument is either an expression belonging to a declared finite enumeration, or a (relative) integer, or a character. Other arguments are couples composed of a pattern and an expression. All the expressions involved have to belong to the same type, which can be any type, even structured or imported. Patterns have to be of the same type as the first argument, and must all be different from each other. If the first argument belongs to a finite enumeration, patterns can exhaustively cover the enumerators by mentioning their names. Depending on the path where the corresponding types have been declared, these names may have to be preceded by a sequence of package identifiers. If the first argument is an integer expression, patterns have to be integer values. Integer constants that are given a name are not allowed in a pattern: the substitution

principle does not work here. Similarly, if the first argument is a character expression, patterns have to be characters. Notice that a **case** expression can also be used with a Boolean expression, simulating then an **if\_then\_else** construct. Patterns must be exhaustive (*i.e.*, all cases must be covered). A default pattern, noted **'\_'**, allows to trap patterns that are not explicitly written. This default pattern is mandatory in case of an integer or character condition. It can only be omitted for a finitely enumerated condition. Given the clock of the first argument, the others expressions have to be based on the same clock. A **case** expression depends on all its arguments; it is well-initialized if its arguments are. Its resulting initialization type is the maximum delay of its arguments type.

### DYNAMIC SEMANTICS

The **if\_then\_else** operator evaluates all its arguments. If the first one evaluates to **true**, then the result is the result of the **then** branch evaluation; if it evaluates to **false**, the result is the result of the **else** branch evaluation.

x	true	false	true	...
a	a1	a2	a3	...
b	b1	b2	b3	...
<b>if x then a else b</b>	a1	b2	a3	...

Similarly, the **case** operator evaluates all its input arguments. If the first one evaluates exactly to one of the patterns mentioned, then the result is the result of the corresponding expression. If no pattern matches the result of the condition, then the result is the result of the evaluation of the default expression. Notice that since patterns themselves are not evaluated, they must only refer to definite values.

## 8.4 Operations on Arrays and Structures

- 8.4.1 [“Array Expressions”](#)
- 8.4.2 [“Structure Expressions”](#)
- 8.4.3 [“Mixed Constructor”](#)
- 8.4.4 [“Labels and Indexes”](#)

### 8.4.1 Array Expressions

#### SYNTAX

---

```
array_expr ::= reverse expr
            | expr @ expr
            | expr index
            | expr [ expr .. expr ]
            | ( expr . {{ index }}+ default expr )
            | transpose ( expr ; INTEGER; INTEGER )
            | expr ^ expr
            | [ list ]
```

---

#### STATIC SEMANTICS

- **reverse** takes as input a single flow belonging to an array type. It produces as result an array of the same length. It does not affect the clock of its argument. The **reverse** operator depends on its input. It has the same initialization type as its input.
- Given two arrays, based on the same type of elements, but with possible different sizes  $n_1$  and  $n_2$ , the append operator @ builds an array on the same basic type whose size is the sum  $n_1 + n_2$ . Supposing that these two arrays are based on the same clock, the appended array is also based on this clock. Naturally, it depends on both its inputs. Its initialization type is the maximum delay of its inputs.
- Given an array of size  $n$  and a static integer expression  $e$ , the static projection  $[e]$  returns an object whose type is the basic type of the array. An extra constraint requires that the static expression must evaluate to a positive integer lesser than the size  $n$  of the array:  $0 \leq e < n$ . Notice that the index in arrays start from 0. The clock of the result is the same one as that of the array. Since sampling operators are not allowed in static expressions, this ensures that this expression is always available. Due to these restrictions on the allowed constructions for static expressions, the

static projection only depends on the array at hand, not on this index expression. Notice that all variables implied in the array at any location will be considered (see dynamic semantics below). The initialization type of the result is the same as that of the array argument.

- Given an array of size  $n$  and two static integer expressions  $e$  and  $f$ , the static slice operator  $[e..f]$  returns an array of size  $f-e+1$  with the same basic type. An extra constraint requires that the second expression is greater than the first one, and both being positive and lesser than the initial size:  $0 \leq e \leq f < n$ . Note that both expressions can be equal. The clock of the result is the same as that of the array. As for the previous operator, no explicit clock constraint is required on the static expressions since they are implicitly always available. The resulting array has the same causal type as its initial array. Variables involved in parts of the array that do not belong to the final one are still considered, due to dynamic semantics. The initialization type of the result is the same as that of the initial array argument.
- Given an array, an integer Scade expression and an expression of the same type as the basic type of the array, the dynamic projection with default builds an object of this basic type. The integer expression is either a size expression or can have any integer type. This operator can be extended to any structured expression. In this case, the path expression can be any combination of identified labels and integer Scade expressions. The default expression has to be of the same type as the remaining expression after the various projections. Contrary to the static projections above, the path expression used as index have to be on the same clock as the array and the default expression. The result is then based on the same clock. The output of a dynamic projection depends on the input structured expression, on its path expression, and on the default expression. Concerning the initialization analysis, given the types of the initial array and of the default expression, the result has the maximum delay of these two types. No initialization constraint is required on the path expression. In case of a badly initialized one, the default expression will protect the result.
- Given two strictly positive integers and an array of arrays with at least as many dimensions as the max of these two integers, **transpose** builds an array of arrays with shifted dimensions corresponding to the arguments. The two integer values can be given in any order. This operator does not affect the clock of its array argument. It has the same dependencies as the initial array, and the same initialization type.

- The first mean to build an array object is through the exponential operator. Given an expression  $e$  of any type, including imported or list, and a static integer size expression  $n$ , then  $e^n$  builds an array of size  $n$  on the basic type of  $e$ .  $n$  is required to evaluate to a strictly positive integer. The resulting expression is based on the same clock as its expression argument  $e$ . No constraint is added for  $n$  since it will always be available: A static expression cannot contain any temporal operator. The dependencies of the resulting array are the same as those of the initial expression  $e$ . Similarly for the initialization type.
- The other mean to build an array is through the extensional operator. Given a non empty list of elements on the same basic type, the list of these elements surrounded by square braces builds an array on this type. The size of this array is at least equal at the number of elements but can be greater since elements are allowed to be lists of elements themselves of the same type. All the elements collected into the array have to be based on the same clock, which is thus the clock of the resulting array. This array depends on all the variables involved in every element of the list. The resulting initialization type is the maximum delay of all the initialization types of the arguments involved.

### DYNAMIC SEMANTICS

Let us first introduce some notations that are used throughout the dynamic semantics of these array operators: let  $a = [a_0, \dots, a_{n-1}]$  be an array of size  $n$ , and  $b = [b_0, \dots, b_{m-1}]$  be an array of size  $m$ . Static integer expressions are noted  $e, f$ . These expressions evaluate to positive integer values noted  $i, j$ .

- If  $r = \text{reverse } a$ , then  $r$  is an array of the same size with reversed elements:  $r = [a_{n-1}, \dots, a_0]$ . More formally:  

$$\forall k \in [0 \dots (n-1)], r[k] = a[n-k-1]$$
- If  $r = a @ b$ , then  $r$  is an array of size  $n+m$  whose elements are those of  $a$  followed by those of  $b$ . Formally:  

$$\forall k \in [0 \dots (n-1)], r[k] = a[k] \text{ and } \forall k \in [n \dots (n+m-1)], r[k] = b[k-n]$$
- Recall that the static expression  $e$  evaluates to integer value  $i$ . Assuming that  $i$  lies between 0 and  $n-1$ , then  $r = a[e]$  is the  $i^{\text{th}}$  element of the array (*i.e.*,  $a_i$ ). Remind that the static projection is left associative, therefore  $r[e][f]$  is  $(r[e])[f]$ .



- Suppose that evaluations of  $e$  and  $f$ , noted  $i$  and  $j$ , lie between 0 and  $n-1$  and follow an increasing order. Then  $r = a[e \dots f]$  is the sliced array  $[a_i, a_{i+1}, \dots, a_j]$  of size  $j-i+1$ .
- Let  $\text{Exp}$  be any integer Scade expression that evaluates to the integer value  $v$ , and  $d$  be an expression of the same type as the elements of  $a$ . If  $r = a \cdot [\text{Exp}] \text{ default } d$ , then  $r$  is the  $v^{\text{th}}$  element of array  $a$  if  $v$  lies between 0 and  $n-1$ , otherwise  $r$  is equal to  $d$ :

$$a \cdot [\text{Exp}] \text{ default } d = \begin{cases} a_v & \text{if } 0 \leq v < n-1 \\ d & \text{otherwise} \end{cases}$$

- The **transpose** ( $c; i; j$ ) expression represents the array containing the same elements as  $c$  such that the  $i^{\text{th}}$  dimension are swapped with the  $j^{\text{th}}$  dimension. Let  $c$  and  $d$  be two arrays of type  $t^{d_n \dots d_1}$ , and  $c = \text{transpose } (d; i; j)$  then the following holds:

$$\forall k \in [0, dn] \dots, k1 \in [0, d1[, \forall i, j \in [1, n]$$

$$c[k1] \dots [k_j] \dots [k_i] \dots [kn] = d[k1] \dots [k_i] \dots [k_j] \dots [kn]$$

- Let  $\text{Exp}$  be any Scade expression and  $e$  a static integer expression that evaluates to  $i$ , then  $r = \text{Exp} \wedge e$  is an array of size  $i$  with every element equal to  $\text{Exp}$ :

$$\forall k \in [0 \dots i-1], r[k] = \text{Exp}$$

- Let  $\text{exp}_0, \dots, \text{exp}_n$  be any Scade expressions of the same type, then  $r = [\text{exp}_0, \dots, \text{exp}_n]$  is the array containing the evaluations of these expressions noted  $i_0, \dots, i_n$ :

$$\forall k \in [0 \dots n], r[k] = i_k$$

## Example 1: Valid array expressions

```

const
  aC1: int322 = [1,1];
  aC2: int321 = [0];

function ex(aI1, aI2: int322; i: uint16)
returns (aO1, aO2, aO3, aO4: int324; aO5: int3223)
let
  aO1 = reverse aI2 @ reverse aI1;
  aO2 = aC2 @ [[1,2,3],[2,3,4]][2*3 - 5];
  aO3 = aO2 [1 .. 2] @ aI2;
  aO4 = aI1 @ ([aI1,aI2].[i] default aC1);
  aO5 = [[0,1],[2,3],[4,5]];
tel

```

## Example 2: Arrays can be used by polymorphic operators. They can be based on imported types and be composed by lists of expressions

```

type imported T;
const imported C:T;
group G=(int32, int32);

node ex(aI1, aI2: int322; clock clk1: bool; aI3: T)
returns (aO1, aO2: int322; aO3: T2)
var v1: G;
let
  v1 = (aI1[0],aI2[1]);
  aO1 = fby(aI1; 1; [v1]);
  aO2 = merge (clk1; aI1 when clk1; aI2 when not clk1);
  aO3 = reverse (C -> aI3)2;
tel

```

## Example 3: An array is not a list of expressions. An error is raised in this example

```

node ex(x: uint64) returns (z, t: uint64)
let
  z,t = [x,x];
tel

```

## 8.4.2 Structure Expressions

### SYNTAX

---

```
struct_expr ::= expr . ID
              | { label_expr { { , label_expr } } }
```

---

### STATIC SEMANTICS

- Given a structured expression `expr` and an identifier `ID` belonging to the list of identifiers provided by the corresponding structure type, then the structure projection `expr . ID` is an element whose type is the one of the corresponding label in the structure. The resulting element is based on the same clock as the initial expression. It depends on the same variables as the whole structured expression. Its initialization type is similar to the one of the initial expression.
- The first mean to build a structure element is to assign a list of expressions to their corresponding labels in the structure. This list must have the same length as the number of labels in the structure type. Each element is composed of a valid label identifier and an expression of the corresponding type. Labels must occur in the same order as in the type declaration. This list is surrounded by curly braces. Contrary to a raw list of expressions, each expression in the structure must be based on the same clock. The resulting structure depends on the union of the dependencies of these expressions. These expressions must not have to be well-initialized, nor to have the same initialization type. The resulting structure has the maximum delay of all the initialization types.

The second mean to build a structure element is given in section 8.5 [“Operator Application and Higher-Order Patterns”](#) by operator `make`.

### DYNAMIC SEMANTICS

Suppose `e` is structure with labels `l_1, ..., l_n`. Then let `l_i` be one of these labels. `e.l_i` represents the value of the field name `l_i`. The reverse operation requires  $n$  expressions `e_1, ..., e_n`. Then `{l_1:e_1, ..., l_n:e_n}` represents a value of the corresponding structure type.

**Example:** Structures and arrays can be fully mixed

```

type
  Tstr1 = {l1: int32};
  Tstr2 = {l1: int32, l2: int32^2};
function ex(sI1: Tstr2; sI2: Tstr1^2)
  returns (iO1: int32)
var
  iV1, iV2: int32;
let
  iV1 = sI1.l2[0];
  iV2 = sI2[0].l1;
  iO1 = iV1 + iV2;
tel

```

### 8.4.3 Mixed Constructor

#### SYNTAX

```
mixed_constructor ::= ( expr with {{ label_or_index }}+ = expr )
```

#### STATIC SEMANTICS

This constructor allows to build a copy of a structured object or an array except for one of its element which is then given a new value set in the declaration.

- Applied to a structure, this constructor requires that the given label belongs to those provided by the type declaration and that the new value has the corresponding type.
- Applied to an array, the index can be dynamic and extended to a list of indexes as for dynamic projection. The index is either a size expression or can have any integer type. The new value has to belong to the basic type of the array. Only single expressions (no list of expressions) are allowed for the new value.

In both cases, the original expression and the new element must be based on the same clock. The output expression depends on all its inputs. Given the initialization types of its inputs, the resulting expression has the maximum delay of both types.

#### DYNAMIC SEMANTICS

- Let  $w$  be an array of size  $n$ ,  $k$  a static integer expression, and  $e$  an expression of the basic type of the array. The expression  $(w \text{ with } [k] = e)$  is an array such that:

$$v[i] = \begin{cases} w[i] & \forall i \in [0 \dots n-1] \setminus k \\ e & \text{if } i = k \text{ and } k \in [0 \dots n-1] \end{cases}$$

Note that if  $k$  does not belong to  $[0 \dots n-1]$ , then  $v=w$ .

- Let  $\text{exp}$  be a structure expression, let  $l$  be one of its labels, and  $e$  be an expression of the corresponding type of this label. Let us note  $L$  the set of all the labels names. Then the expression  $v = \text{exp with } l = e$  is:

$$v \cdot \text{lbl} = \begin{cases} \text{exp} \cdot \text{lbl} & \forall \text{lbl} \in L \setminus k \\ e & \text{if } \text{lbl} = l \end{cases}$$

Contrary to the array case, an error is raised if the modified label does not belong to the allowed set of labels.

**Example:** Mixed constructor can be used with indexes or labels. Indexes can be outside the array range, while labels must belong to the structure type (contrary to dynamic projection). Mixed constructor can be nested: the outermost occurrence is the effective one.

---

```

type
  Tstr = {l1: int32, l2: float64};
  Tarr = int32^3;
function ex(sI1: Tstr; sI2:{l1:int32})
returns (sO1: Tstr; aO2: Tarr; sO3:{l1: int32})
let
  sO1 = (sI1 with .l2 = 3.0);
  aO2 = (sI2.l1^3 with [0] = 0);
  sO3 = ((sI2 with .l1 = 1) with .l1 = 3);
tel

```

---

## 8.4.4 Labels and Indexes

Labels and indexes are used by the previous projection operators and the mixed constructor.

### SYNTAX

---

```

label_expr ::= ID : expr
            index ::= [ expr ]
label_or_index ::= . ID
                | index

```

---

## 8.5 Operator Application and Higher-Order Patterns

A user-defined operator can be used in its scope as any primitive operator to build expressions. Scade also provides higher-order primitives that modify the behavior of operators: They take as input an operator and return another operator that can thus be applied as usual operators.

### SYNTAX

---

```

apply_expr  ::= operator ( list )
operator    ::= prefix
              | ( prefix << [[ expr {{ , expr }} ]] >> )
              | ( make path_id )
              | ( flatten path_id )
              | ( iterator operator << expr >> )
              | ( activate operator every clock_expr )
              | ( activate operator every expr default expr )
              | ( activate operator every expr initial default expr )
              | ( restart operator every expr )
              | ( iterator_mw operator << expr >>
                  if expr default expr )
              | ( foldw operator << expr >> if expr )
              | ( foldwi operator << expr >> if expr )
prefix      ::= path_id
              | PREFIXOP
iterator     ::= map | fold | mapi | foldi
              | mapfold [[ INTEGER ]]
              | mapfoldi [[ INTEGER ]]
iterator_mw  ::= mapw | mapwi
              | mapfoldw [[ INTEGER ]]
              | mapfoldwi [[ INTEGER ]]

```

---

### STATIC SEMANTICS

An operator application requires a functional operator and a possibly empty list of parameters. There can be less parameters than input variables of the operator, due to the possible grouping of some parameters into lists, which are expanded into their elements. These individual elements and the remaining parameters are the actual parameters which must fit the input variables of the operator in terms of number, types, clocks, and initializations. Moreover, when an input variable of the operator is declared as a clock identifier, only a valid clock identifier can be used as its actual parameter.

The following list enumerates the different kind of functional operators available. Notice that operators *op* mentioned in the following are either primitive Scade operators in their prefix form or user defined ones.

- A non parameterized user-defined operator is identified by its name, possibly preceded by a package path. The identifier must refer to a valid user operator, different from the current operator name. In case of an operator that introduces a memory (*i.e.*, a **node**) the application is an object which also introduces a memory. On the contrary, the application of a **function** can be used in any context. The application of a user operator creates as many flows as there are outputs of this operator. These flows fit the corresponding outputs in terms of type, clock, causality, and delay type.

Universally quantified operator types generated during the analyses of this user operator are specialized during the instantiation:

- The fastest abstract clock is bound to the fastest clock of the parameters. The parameters clocks being based on this actual fastest clock, it is then possible to check that the parameters fit the clock constraints enforced by the clock type of the operator.
- The causal type of every input is bound to the causal type of its corresponding parameter. Outputs of the instantiation can then be given their causality in terms of the actual causal types.
- Given the delay types of the input variables, the delay type of each output is either atomic or evaluated according to the specification of the maximum delay operation. Three cases occur for the inputs delay type:
  - 1 Non-constrained inputs are given a universally quantified delay type. These inputs can be instantiated by an expression having any type.
  - 2 Inputs constrained to be well-defined must be given a well-defined parameter.
  - 3 Inputs constrained to be undefined at the first instant can be given any kind of expression.
- A parameterized operator is also identified by its name, possibly preceded by a package path. Each of its size parameters requires a size integer expression as argument. Constraints on these size parameters must be satisfied by the actual size arguments. A size parameter used in an expression is treated as an integer literal. It means that its type can be inferred from the context and that a constraint is added to make sure that its value is within the range of predefined integer types. The type of this instance of the parametrized operator is then a non-parameterized operator type

on which the rules above apply. The resulting non-parameterized operator has the same characteristics as the initial one: the clock type, causality type, and delay type are not affected by sizes instantiation.

- Operator **make** is the second mean to build an object of a structure data type. It takes as input an identifier (possibly preceded by a package path) referring to a structure type name. It produces an operator that can then be applied to a list of parameters in order to build an object belonging to this structure type. These parameters can be lists of parameters. A preliminary expansion of these lists leads to a flat list which must fit the number of labels of the structure and their type. Since all the fields of a structure must be based on the same clock, the application of operator **make** to a structure type builds an operator whose application to field values based on a given clock builds an element based on the same one. The resulting structure naturally depends on all its inputs. It has the maximum delay of all its inputs.
- Operator **flatten** destructures a complex type into a list of elements. It applies to both structure and array types. It requires as input an identifier referring to either a structure type name or an array type name. It builds an operator than can then be applied to a flow belonging to the complex type in order to produce as many output flows as the size of the structure or array. The size must be exactly known: **flatten** cannot be applied on parameterized arrays. Each output flow has the data type corresponding to its place in case of a structure type, and the base type in case of an array type.  
The application of **flatten** to a type name builds an operator whose application to a complex object based on a given clock builds flows based on the same clock. Every resulting flow has the same causality as the complex object, and the same delay type.  
Note that there is no such operator for groups, since this flattening operation is automatically performed during type equivalence.
- Operator **map** takes as input an operator *op* which can have any operator type, and a static integer expression. It produces a new operator which then requires on input as many arrays as the number of inputs of *op*. These arrays must be composed of elements whose type matches the type of the corresponding input, and have the same size as the static integer expression. This equality between sizes is a constraint that does not need to be syntactically proved but must be statically provable. The result is a list of arrays of the same size and whose basic types correspond to the output types of *op*. The input operator *op* must take all its inputs



based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs and cannot affect the base clock on output. This new operator has the most general causality possible: all its outputs depends on all its inputs. It does not affect the initialization signature of the initial operator.

- Operator **fold** takes as input an operator *op* and a static integer expression. This argument operator must have a non-empty list of inputs and a single output whose type  $\tau$  must be the same as the first input. The **fold** application produces a new operator which requires on input an element of type  $\tau$  (called accumulator) and as many arrays as the remaining inputs of *op*. These arrays must be composed of elements whose type matches the type of the corresponding input, and have the same size as the static integer expression. The result is an element of type  $\tau$ . The input operator *op* must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs and cannot affect the base clock on output. This new operator has the most general causality possible: all its outputs depends on all its inputs.

The input operator must not increase the delay type of its first argument. A well-defined accumulator must result in a new well-defined one, while an undefined one can result in any delay type. No other constraints apply on the delay type of *op*. The application of **fold** to this operator results in the same operator delay type.

- Operator **mapfold** represents a family of iterators parameterized by an immediate integer value  $a$  such that  $a \geq 0$ . Given an immediate integer value  $a$  and a static integer expression, **mapfold**  $a$  can be applied to an operator *op* that takes at least  $a$  inputs and returns at least  $a$  outputs. Its  $a$  first inputs are called *accumulators*; their types are pointwise equivalent to those of its  $a$  first outputs. This application leads to a new operator with at least  $a$  inputs for the initial values of the accumulators. The other inputs are arrays which must be composed of elements whose type matches the type of the corresponding input, and have the same size as the static integer expression. The outputs are  $a$  output accumulators followed by a list of arrays composed of elements whose type matches the type of the corresponding output, and have the same size as the static integer expression.

**mapfold** can be applied to an operator with all its inputs and outputs on the same clock; the operator resulting from this application also has all its inputs and outputs on the same clock. An operator obtained by the application of a **mapfold** has the

most restrictive causality constraint: all its outputs depend on all its inputs. *op* must return only well-initialized flows for its a first inputs. **mapfold** applied to this operator results in the same operator delay type.

- Operator **mapfoldi** represents a family of iterators parameterized by an immediate integer value  $a \geq 0$ . **mapfoldi**  $a$  extends the profile of *op* as it is required by **mapfold**  $a$  by an integer value in the first position. Other rules are kept as such.

**mapfoldi** can be applied to an operator with all its inputs and outputs on the same clock; the operator resulting from this application also has all its inputs and outputs on the same clock. An operator obtained by the application of a **mapfoldi** has the most restrictive causality constraint: all its outputs depends on all its inputs. *op* must return only well-initialized flows for its a first inputs. **mapfoldi** applied to this operator results in the same operator delay type.

- Operator **mapi** extends the profile of the input operator required by operator **map** by an integer value in the first position. The first argument of the input operator of **mapi** can be any integer type, except a type variable. Other rules are kept as such.

The input operator of a **mapi** operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs and cannot affect the base clock on output. This new operator has the most general causality possible: all its outputs depends on all its inputs. It does not affect the initialization type of the initial operator.

- Operator **foldi** extends the profile of the input operator required by operator **fold** by an integer value in the first position. The first argument of the input operator of **foldi** can be any integer type, except a type variable. Other rules are kept as such.

The input operator of a **foldi** operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs and cannot affect the base clock on output. This new operator has the most general causality possible: all its outputs depends on all its inputs.

The input operator must not increase the delay type of its first argument. A well-defined accumulator must result in a new well-defined one, while an undefined one can result in any delay type. No other constraints apply on the delay type of *op*. The application of **foldi** to this operator results in the same operator delay type.

- The first **activate** operator takes as input an operator with any operator type, and a clock expression (see section 6.3 [“Clock Expressions”](#)). It produces an operator with the same profile as the original operator.

The input operator must not add clock constraints between its inputs, they all must be based on the same clock, and has to return all its outputs on the same one. Given the new clock represented by the input clock expression, the application of **activate** to this operator samples its outputs on this new clock, and produces outputs which are all synchronized on the clock represented by the clock expression. Given the causality type of the input operator, the application of **activate** to this operator extends the causality types of the outputs by a dependency on the clock expression. The first **activate** does not affect the initialization signature of its input operator.

- The second **activate** takes as input an operator with any profile, a Boolean expression and an expression with the same type as the output of the original operator (it can be a list of output types). It produces an operator with the same profile as the original operator and does not introduce a memory.

The input operator of an **activate** operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs and cannot affect the base clock on output. The conditional and default expressions must also be based on the same clock.

Given the causality type of the input operator, the application of **activate** to this operator extends the causality types of the outputs by a dependency on the conditional and default expressions. An **activate** with default initialization is well-initialized if it is applied to an operator that produces outputs of type 0. The conditional and initial default expression must also be well-defined.

- The third **activate** operator takes the same inputs as the previous one. It produces also an operator with the same profile as the original operator, but introduces a memory.

The input operator of an **activate** operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs and cannot affect the base clock on output. The conditional and default expressions must also be based on the same clock. Given the causality type of the input operator, the application of **activate** to this operator extends the causality

types of the outputs by a dependency on the conditional and default expressions. An **activate** with default is well-initialized if it is applied to an operator that produces outputs of type 0. The conditional and default expression must also be well-defined.

- The **restart** operator takes as input an operator with any profile and a Boolean expression. It produces an operator with the same profile as the original operator. It does not affect the clock type of its input operator. Contrary to other higher order operators, it does not require that this input operator has a flat clock type: it can enforce clock constraints between its inputs and affect the clock of its outputs. The resulting operator has the same causality and delay type as the initial one.
- Operator **mapw** takes as input an operator with a non empty list of outputs (with first item being a Boolean), a Boolean expression, an expression with the same type as the list of outputs without its first element, and a static integer expression. It produces a new operator which then requires on input as many arrays as the number of inputs of the original operator less one (the first one). These arrays must be composed of elements whose type matches the type of the corresponding inputs, and have the same size as the static integer expression. The result is an integer value and a list of arrays of the same size and whose types correspond to the output types of the original operator.

The input operator of a **mapw** operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs, and cannot affect the base clock on output. The conditional expression and the default one must also be based on the same clock. This new operator has the most general causality possible: all its outputs depends on all its inputs, and on the conditional and default expressions.

The input operator must have an initialization type such that its first output must be well-initialized. This constraint is required since this output is used as the control condition of the loop. Given that the condition and the default expression are also well-initialized, the resulting application has the same type as the initial operator.

- Operator **mapwi** extends the previous rules by requiring that the original operator takes as input at least an integer value and a list of expressions. The first argument of the input operator of **mapwi** can be any integer type, except a type variable. Array parameters passed to the resulting application then need to match the remaining list of input parameters.

The input operator of a `mapwi` operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs, and cannot affect the base clock on output. The conditional expression and the default one must also be based on the same clock. This new operator has the most general causality possible: all its outputs depends on all its inputs, and on the conditional and default expressions.

The input operator must have an initialization type such that its first output must be well-initialized. This constraint is required since this output is used as the control condition of the loop. Given that the condition and the default expressions are also well-initialized, the resulting application has the same type as the initial operator.

- Operator `foldw` takes as input an operator and a static integer expression. This argument operator must have a non empty list of inputs and two outputs: a Boolean one, and one whose type  $\tau$  must be the same as the first input. The `foldw` application produces a new operator which requires on input an element of type  $\tau$  and as many arrays as the remaining inputs of the original operator. These arrays must be composed of elements whose type matches the type of the corresponding input, and have the same size as the static integer expression. The result is an integer value and an element of type  $\tau$ .

The input operator of a `foldw` operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs, and cannot affect the base clock on output. The conditional expression must also be based on the same clock. This new operator has the most general causality possible: all its outputs depends on all its inputs and on the conditional expression.

The input operator must not increase the delay type of its first argument. A well-defined accumulator must result in a new well-defined one, while an undefined one can result in any delay type. Moreover, its first output must be well-initialized. This latter constraint is required since this output is used as the control condition of the loop. Given that the condition is also well-initialized, the resulting application has the same type as the initial operator.

- Operator `foldwi` extends the previous rules by requiring that the original operator takes as input at least an integer expression, an object of type  $\tau$ , and a list of expressions.

The input operator of a **foldw** operator must take all its inputs based on a unique given clock, and has to return outputs based on the same one. The new operator follows the same rules: it cannot require clock constraints between its inputs, and cannot affect the base clock on output. The conditional expression must also be based on the same clock. This new operator has the most general causality possible: all its outputs depends on all its inputs and on the conditional expression.

The input operator must not increase the delay type of its first argument. A well-defined accumulator must result in a new well-defined one, while an undefined one can result in any delay type. Moreover, its first output must be well-initialized. This latter constraint is required since this output is used as the control condition of the loop. Given that the condition is also well-initialized, the resulting application has the same type as the initial operator.

- Operator **mapfoldw** represents a family of iterators parameterized by an immediate integer value  $a$  such that  $a \geq 0$ . Given an immediate integer value  $a$  and a static integer expression, **mapfoldw**  $a$  can be applied to an operator  $op$  that takes at least  $a+1$  inputs and returns at least  $a+1$  outputs. Its  $a$  first inputs are called *accumulators*; their types are pointwise equivalent to those of its  $a$  first outputs. This application leads to a new operator with at least  $a + 1$  inputs:

a Boolean expression followed by;

a list of default expression which types point-wisely match those of  $op$  outputs (from the  $a + 1$ -th to the latest) followed by;

the  $a$  the initial values followed by;

the other inputs are arrays which must be composed of elements whose type matches the type of the corresponding input of  $op$ , and have the same size as the static integer expression.

The outputs are:

an integer value followed by;

a Boolean followed by;

an output accumulators followed by;

a list of arrays composed of elements whose type matches the type of the corresponding output of  $op$ , and have the same size as the static integer expression

**mapfoldw** can be applied to an operator with all its inputs and outputs on the same clock; the operator resulting from this application also has all its inputs and outputs on the same clock. An operator obtained by the application of a **mapfoldw** has the most restrictive causality constraint: all its outputs depends on all its inputs including input condition and default expressions. *op* must return only well-initialized flows for its a first inputs. **mapfoldw** applied to this operator results in the same operator delay type.

- Operator **mapfoldwi** represents a family of iterators parameterized by an immediate integer value *a* such that  $a \geq 0$ . Given an immediate integer value *a* and a static integer expression, **mapfoldwi** *a* can be applied to an operator *op* that takes at least *a* + 1 inputs and returns at least *a* + 1 outputs; the type of *op* is the same as for **mapfoldw**, extended with an additional integer input in first position.

**mapfoldwi** can be applied to an operator with all its inputs and outputs on the same clock; the operator resulting from this application also has all its inputs and outputs on the same clock. An operator obtained by the application of a **mapfoldwi** has the most restrictive causality constraint: all its outputs depends on all its inputs including input condition and default expressions. *op* must return only well-initialized flows for its *a* first inputs. **mapfoldwi** applied to this operator results in the same operator delay type.

## DYNAMIC SEMANTICS

Operators **make** and **flatten** do not have a dynamic semantics, since they are just structuring or decomposing operators. They behave as their static semantics implies.

Other operators are described below:

- The instantiation of an operator *op* by a list of input parameters  $exp_1, \dots, exp_n$  consists in the evaluation of every parameter toward some values that are then applied to this operator. Values application is the substitution of formal input parameters by their corresponding values. The resulting expression for each output is then itself evaluated. The global result is thus a list of values representing the evaluation of each output.
- The instantiation of the parameterized operator sizes ( $op \ll expr_0, \dots, expr_n \gg$ ) consists in replacing its formal size parameters by the actual expressions  $expr_1, \dots, expr_n$ . *Size parameters* differ from standard parameters by the fact that they can be used in size expressions (typically array sizes) and therefore require  $expr_1, \dots, expr_n$  to be static expressions. The operator obtained can then be instantiated by its other parameters.

- Let  $\text{op}$  be an operator taking  $n$  parameters as input and producing  $k$  output values. Let  $A_1, \dots, A_n$  be arrays of size  $\text{size}$  having the corresponding basic types as the inputs of operator  $\text{op}$ . Then  $v_1, \dots, v_m$  such that:

---


$$v_1, \dots, v_m = (\text{map } \text{op } \ll \text{size} \gg)(A_1, \dots, A_n)$$


---

are arrays of size  $\text{size}$  defined by:

$$\forall i \in [0 \dots \text{size}] [v_1[i], \dots, v_m[i]] = \text{op}(A_1[i], \dots, A_n[i])$$

- mapi** behaves as **map**, but operator  $\text{op}$  is required to take an extra integer argument as its first input. The current iteration index is passed as this first argument:

$$\forall i \in [0 \dots \text{size}] [v_1[i], \dots, v_m[i]] = \text{op}(i, A_1[i], \dots, A_n[i])$$

- Let  $\text{op}$  be an operator taking  $n+1$  parameters as input and producing one output value of the same type as its first input. Let  $A_1, \dots, A_n$  be arrays of size  $\text{size}$  having the corresponding basic types as the inputs of operator  $\text{op}$ , and  $\text{exp}$  be an expression of the first input type. Then  $\text{acc}$  such that:

---


$$\text{acc} = (\text{fold } \text{op } \ll \text{size} \gg)(\text{exp}, A_1, \dots, A_n)$$


---

is an expression of the same type as  $\text{exp}$  defined by:

$$\begin{cases} \text{acc} = \text{acc}_{\text{size}} \\ \forall i \in [0 \dots \text{size}] [\text{acc}_{i+1} = \text{op}(\text{acc}_i, A_1[i], \dots, A_n[i])] \\ \text{acc}_0 = \text{exp} \end{cases}$$

- foldi** behaves the same as **fold**, but operator  $\text{op}$  is required to take an extra integer argument as its first input. The current iteration index is passed as this first argument:

$$\begin{cases} \text{acc} = \text{acc}_{\text{size}} \\ \forall i \in [0 \dots \text{size}] [\text{acc}_{i+1} = \text{op}(i, \text{acc}_i, A_1[i], \dots, A_n[i])] \\ \text{acc}_0 = \text{exp} \end{cases}$$

- Let  $\text{op}$  be an operator taking  $a+n$  parameters as input and producing  $a+m$  output values, such that their  $a$  first item have the same type. Let  $A_1, \dots, A_n$  be arrays of size  $\text{size}$  having the corresponding basic types as the inputs of operator  $\text{op}$ , and  $\text{exp}^1, \dots, \text{exp}^a$  be expressions of these first  $a$  items types. The equation:

---


$$\text{acc}^1, \dots, \text{acc}^a, v_1, \dots, v_m = (\text{mapfold } a \text{ op } \ll \text{size} \gg)(\text{exp}^1, \dots, \text{exp}^a, A_1, \dots, A_n)$$


---



is equivalent to:

$$\left\{ \begin{array}{l} \forall j \in [1 \dots a], \text{acc}^j = \text{acc}_{\text{size}}^j \\ \forall i \in [0 \dots \text{size}], \text{acc}_{i+1}^1, \dots, \text{acc}_{i+1}^a, v_I[i], \dots, v_m[i] = \text{op}(\text{acc}_i^1, \dots, \text{acc}_i^a, A_I[i], \dots, A_n[i]) \\ \forall j \in [1 \dots a], \text{acc}_0^j = \text{exp}^j \end{array} \right.$$

When the number of accumulators is not specified, the default value for  $a$  is 1, *i.e.*, operator (**mapfold**  $\text{op} \ll \text{size} \gg$ ) is equivalent to (**mapfold** 1  $\text{op} \ll \text{size} \gg$ ).

- Let  $\text{op}$  be an operator taking  $a+n+1$  parameters as input and producing  $a+m$  output values. Let  $A_1, \dots, A_n$  be arrays of size  $\text{size}$  having the corresponding basic types as the inputs of operator  $\text{op}$ , and  $\text{exp}^1, \dots, \text{exp}^a$  be expression of these first  $a$  items type. The equation:

---


$$\text{acc}^1, \dots, \text{acc}^a, v_1, \dots, v_m = (\text{mapfoldi } a \text{ op } \ll \text{size} \gg)(\text{exp}^1, \dots, \text{exp}^a, A_1, \dots, A_n)$$


---

is equivalent to:

$$\left\{ \begin{array}{l} \forall j \in [1 \dots a], \text{acc}^j = \text{acc}_{\text{size}}^j \\ \forall i \in [0 \dots \text{size}], \text{acc}_{i+1}^1, \dots, \text{acc}_{i+1}^a, v_I[i], \dots, v_m[i] = \text{op}(i, \text{acc}_i^1, \dots, \text{acc}_i^a, A_I[i], \dots, A_n[i]) \\ \forall j \in [1 \dots a], \text{acc}_0^j = \text{exp}^j \end{array} \right.$$

When the number of accumulators is not set, the default value for  $a$  is 1, *i.e.*, operator (**mapfoldi**  $\text{op} \ll \text{size} \gg$ ) is equivalent to (**mapfoldi** 1  $\text{op} \ll \text{size} \gg$ ).

- The equation  $\text{lhs} = (\text{activate } \text{op } \text{every clock\_expr})(\text{exp}_1, \dots, \text{exp}_n)$  defines a bunch of flows  $\text{lhs}$  defined equivalently by:

---


$$\text{lhs} = \text{op}(\text{exp}_1, \dots, \text{exp}_n \text{ when clock\_expr});$$


---

The evaluation of this expression requires first the evaluation of each parameter expression  $\text{expr}_i$  to corresponding values  $v_i$ . If the clock expression evaluates to **true**, the instantiation of  $\text{op}$  by the values  $v_i$  is performed. Otherwise, no value is produced and the internal memory of operator  $\text{op}$ , if any, is unmodified.

- The equation `lhs=(activate op every cond_expr default deflt_expr)(exp1,...,expn)` defines a bunch of flows `lhs` whose equivalent definition is:

---

```
lhs = merge(h;
  op (exp1 ,..., expn when h);
  deflt_expr when not h);
```

---

`h` is the Boolean clock defined by the equation `h=cond_expr`.

The evaluation of this expression requires first the evaluation of each parameter expression `expri` to corresponding values `vi` and of each default expression `deflt_expri` to corresponding values `di`. If the clock expression evaluates to **true**, the instantiation of `op` by the values `vi` is performed. Otherwise, the default values `di` are returned and the memory of operator `op`, if any, is unmodified.

- The equation `lhs=(activate op every cond_expr initial default deflt_expr)(exp1,...,expn)` defines a bunch of flows `lhs` whose equivalent definition is:

---

```
lhs = merge(h;
  op (exp1 ,..., expn when h);
  init_expr -> pre(lhs ) when not h);
```

---

`h` is the Boolean clock defined by the equation `h=cond_expr`.

The evaluation of this expression requires first the evaluation of each parameter expression `expri` to corresponding values `vi` and of each default expression `deflt_expri` to corresponding values `di`. If the clock expression evaluates to **true**, the instantiation of `op` by the values `vi` is performed. Otherwise, the memory of operator `op`, if any, is unmodified; and if it is the first instant of activation then the default values `di` are returned; and for all other instants the previous result is returned.

- The expression `(restart op every cond_expr)(exp1,...,expn)` is equivalent to calling at each cycle the instantiation `op(exp1,...,expn)`. Moreover, when `cond_expr` is true:
  - `->` operators return their first argument, like at first cycle;
  - flows initialized with a **last** declaration are reset to the corresponding value;
  - State Machines are reset to their initial state;
  - **time** operators are reset;

- implementation ensures that imported nodes have their reset function called. This applies to the whole model subtree having `op` as top-level.
- **mapw** is a conditional iterator: it performs the same treatment as **map**, but stops as soon as a condition falls down to **false**. The iteration number where this happens is memorized to an integer value. Since the produced arrays must have the same size as the input ones, these arrays are filled with default values for the remaining slots after this iteration number.

Let `op` be an operator taking `n` parameters as input and producing `k+1` output values, its first output being a Boolean expression. Let  $A_1, \dots, A_n$  be arrays of size `size` having the corresponding basic types as the inputs of operator `op`, `initcond` a Boolean expression, and  $d_1, \dots, d_m$  some default values of the same type as the outputs of `op`. The equation:

---

```
idx, v1, ..., vm = (mapw op <<size >>
                    if initcond
                    default (d1, ..., dm)(A1, ..., An);
```

---

is equivalent to:

$$\left\{ \begin{array}{l} \text{cond}_0 = \text{initcond} \\ \forall i \in [0 \dots \text{idx}], (\text{cond}_{i+1}, v_1[i], \dots, v_m[i]) = \text{op}(A_1[i], \dots, A_n[i]) \\ \forall i \in [0 \dots \text{idx} - 1], \text{cond}_i = \text{true} \\ \text{idx} = \text{size} \vee \text{cond}_{\text{idx}-1} = \text{false} \\ \forall i \in [\text{idx} \dots \text{size}], \text{cond}_i = \text{cond}_{i-1} \\ \forall i \in [\text{idx} \dots \text{size}], \forall j \in [1, m], v_j[i] = d_j \end{array} \right.$$

From an operational point of view there are `size` instances of `op` activated in order from 0 to `size-1` while the corresponding condition is true. As soon as one instance returns a false condition, the instances that follow are not activated in the current cycle and the values needed for the output arrays are provided by the default expressions. In particular, if `initcond` is false, none of the `op` instances are activated and the returned value for `idx` is 0. Note that **mapw** corresponds to a particular **mapfoldw**; the equation above is equivalent to:

---

```
idx, _, v1, ..., vm = (mapfoldw 0 op <<size >>
                    if initcond
                    default (d1, ..., dm)(A1, ..., An);
```

---

- **mapwi** behaves the same as **mapw**, but operator  $\text{op}$  is required to take an extra integer argument as its first input. The current iteration index is passed as this first argument:

---

```
idx, v1, ..., vm = (mapwi op <<size>>
                    if initcond
                    default (d1, ..., dm)(A1, ..., An);
```

---

is equivalent to:

$$\left\{ \begin{array}{l} \text{cond}_0 = \text{initcond} \\ \forall i \in [0 \dots \text{idx}], (\text{cond}_{i+1}, v_1[i], \dots, v_m[i]) = \text{op}(i, A_1[i], \dots, A_n[i]) \\ \forall i \in [0 \dots \text{idx} - 1], \text{cond}_i = \text{true} \\ \text{idx} = \text{size} \vee \text{cond}_{\text{idx}-1} = \text{false} \\ \forall i \in [\text{idx} \dots \text{size}], \text{cond}_i = \text{cond}_{i-1} \\ \forall i \in [\text{idx} \dots \text{size}], \forall j \in [1, m], v_j[i] = d_j \end{array} \right.$$

From an operational point of view there are  $\text{size}$  instances of  $\text{op}$  activated in order from 0 to  $\text{size}-1$  while the corresponding condition is true. As soon as one instance returns a false condition, the instances that follow are not activated in the current cycle and the values needed for the output arrays are provided by the default expressions. In particular, if  $\text{initcond}$  is false, none of the  $\text{op}$  instances are activated and the returned value for  $\text{idx}$  is 0. Note that **mapwi** corresponds to a particular **mapfoldwi**; the equation above is equivalent to:

---

```
idx, _, v1, ..., vm = (mapfoldwi 0 op <<size>>
                    if initcond
                    default (d1, ..., dm)(A1, ..., An);
```

---

- **foldw** is also a conditional iterator: it performs the same treatment as **fold**, but stops as soon as a condition falls down to **false**. The iteration number where this happens is memorized to an integer value. Contrary to **mapw** iterator, no default value is necessary, since the produced output is a single value.

Let  $\text{op}$  be an operator taking  $n+1$  parameters as input and producing two output values, a Boolean and an output having the same type as the first input. Let  $A_1, \dots, A_n$  be arrays of size  $\text{size}$  having the corresponding basic types as the inputs of operator  $\text{op}$ , and  $\text{acc0}$  be an expression of the first input type. Then  $\text{idx}$  and  $\text{acc}$  such that:

---

```
idx, acc = (foldw op <<size>> if cond0)(acc0, A1, ..., An)
```

---

are defined by  $\text{idx}=\text{idx}_{\text{size}}$  and  $\text{acc}=\text{acc}_{\text{size}}$  such that:

$$\forall i \in [0 \dots \text{size}[, \text{idx}_{i+1}, \text{cond}_{i+1}, \text{acc}_{i+1}] = \begin{cases} \text{idx}_i + 1, \text{op}(\text{acc}_i, A_1[i], \dots, A_n[i]) & \text{if } \text{cond}_i = \text{true} \\ \text{idx}_i, \text{acc}_i, \text{cond}_i & \text{otherwise} \end{cases}$$

Note that **foldw** corresponds to a particular **mapfoldw**; the equation above is equivalent to:

---

```
idx,_, acc = (mapfoldw 1 op <<size>> if cond_0)(acc_0, A_1, ..., A_n)
```

---

- **foldwi** behaves the same as **foldw**, but operator **op** is required to take an extra integer argument as its first input. The current iteration index is passed as this first argument:

$$\forall i \in [0 \dots \text{size}[, \text{idx}_{i+1}, \text{cond}_{i+1}, \text{acc}_{i+1}] = \begin{cases} \text{idx}_i + 1, \text{op}(\text{acc}_i, A_1[i], \dots, A_n[i]) & \text{if } \text{cond}_i = \text{true} \\ \text{idx}_i, \text{acc}_i, \text{cond}_i & \text{otherwise} \end{cases}$$

Note that **foldwi** corresponds to a particular **mapfoldwi**; the equation above is equivalent to:

---

```
idx,_, acc = (mapfoldwi 1 op <<size>> if cond_0)(acc_0, A_1, ..., A_n)
```

---

- Let **op** be an operator taking  $a+n$  parameters as input and producing  $a+m+1$  output values, such that their  $a$  first items have the same type. Let  $A_1, \dots, A_n$  be arrays of size **size** having the corresponding basic types as the inputs of operator **op**, and  $\text{exp}^1, \dots, \text{exp}^a$  be expression of these first  $a$  items type. The equation:

---

```
idx, cond, acc1, ..., acca, v1, ..., vm =  
(mapfoldw a op << size>> if initcond default (d1, ..., dm))(exp1, ..., expa, A1, ..., An)
```

---

is equivalent to:

$$\left\{ \begin{array}{l} \text{cond}_0 = \text{initcond} \\ \forall j \in [1 \dots a], \text{acc}_0^j = \text{exp}^j \\ \forall i \in [0 \dots \text{idx}], \text{cond}_{i+1}, \text{acc}_{i+1}^1, \dots, \text{acc}_{i+1}^a, v_i^1, \dots, v_i^m = \text{op}(\text{acc}_i^1, \dots, \text{acc}_i^a, A_1[i], \dots, A_n[i]) \\ \forall i \in [0 \dots \text{idx}-1], \text{cond}_i = \mathbf{true} \\ \text{idx} = \text{size} \vee \text{cond}_{\text{idx}-1} = \mathbf{false} \\ \forall i \in [\text{idx} \dots \text{size}], \text{cond}_i = \text{cond}_{i-1} \\ \forall i \in [\text{idx} \dots \text{size}], \forall j \in [1, m], v_j[i] = d_j \\ \forall i \in [\text{idx} \dots \text{size}], \forall j \in [1 \dots a], \text{acc}_i^j = \text{acc}_{i-1}^j \\ \text{cond} = \text{cond}_{\text{size}} \\ \forall j \in [1 \dots a], \text{acc}^j = \text{acc}_{\text{size}}^j \end{array} \right.$$

When the number of accumulators is not set, the default value for  $a$  is 1, *i.e.*, operator (**mapfoldw**  $\text{op} \ll \text{size} \gg$ ) is equivalent to (**mapfoldw** 1  $\text{op} \ll \text{size} \gg$ ).

From an operational point of view, there are  $\text{size}$  instances of  $\text{op}$  activated in order from 0 to  $\text{size}-1$  while the corresponding condition is true. As soon as one instance returns a false condition, the instances that follow are not activated in the current cycle and the values needed for the output arrays are provided by the default expressions. In particular, if  $\text{initcond}$  is false, none of the  $\text{op}$  instances are activated and the returned value for  $\text{idx}$  is 0.

- Let  $\text{op}$  be an operator taking  $a+n+1$  parameters as input and producing  $a+m+1$  output values, such that their first items have the same type. Let  $A_1, \dots, A_n$  be arrays of size  $\text{size}$  having the corresponding basic types as the inputs of operator  $\text{op}$ , and  $\text{exp}^1, \dots, \text{exp}^a$  be expression of these first  $a$  items type. The equation:

---


$$\text{idx}, \text{cond}, \text{acc}^1, \dots, \text{acc}^a, v_1, \dots, v_m = (\mathbf{mapfoldwi} \ a \ \text{op} \ \ll \text{size} \gg \ \mathbf{if} \ \text{initcond} \ \mathbf{default} \ (d_1, \dots, d_m))(\text{exp}^1, \dots, \text{exp}^a, A_1, \dots, A_n)$$


---

is equivalent to:

$$\left\{ \begin{array}{l}
 \text{cond}_0 = \text{initcond} \\
 \forall j \in [1 \dots a], \text{acc}_0^j = \text{exp}^j \\
 \forall i \in [0 \dots \text{idx}], \text{cond}_{i+1}, \text{acc}_{i+1}^1, \dots, \text{acc}_{i+1}^a, v_I[i], \dots, v_m[i] = \text{op}(i, \text{acc}_i^1, \dots, \text{acc}_i^a, A_1[i], \dots, A_n[i]) \\
 \forall i \in [0 \dots \text{idx}-1], \text{cond}_i = \mathbf{true} \\
 \text{idx} = \text{size} \vee \text{cond}_{\text{idx}-1} = \mathbf{false} \\
 \forall i \in [\text{idx} \dots \text{size}], \text{cond}_i = \text{cond}_{i-1} \\
 \forall i \in [\text{idx} \dots \text{size}], \forall j \in [1, m], v_j[i] = d_j \\
 \forall i \in [\text{idx} \dots \text{size}], \forall j \in [1 \dots a], \text{acc}_i^j = \text{acc}_{i-1}^j \\
 \text{cond} = \text{cond}_{\text{size}} \\
 \forall j \in [1 \dots a], \text{acc}^j = \text{acc}_{\text{size}}^j
 \end{array} \right.$$

When the number of accumulators is not set, the default value for  $a$  is 1, *i.e.*, operator (**mapfoldwi**  $\text{op} \ll \text{size} \gg$ ) is equivalent to (**mapfoldwi** 1  $\text{op} \ll \text{size} \gg$ ).

From an operational point of view, there are  $\text{size}$  instances of  $\text{op}$  activated in order from 0 to  $\text{size}-1$  while the corresponding condition is true. As soon as one instance returns a false condition, the instances that follow are not activated in the current cycle and the values needed for the output arrays are provided by the default expressions. In particular, if  $\text{initcond}$  is false, none of the  $\text{op}$  instances are activated and the returned value for  $\text{idx}$  is 0.

**Example 1:** Given operator  $F$  with the following types:

- $\text{bool} \times \text{int32} \times 'T \rightarrow \text{int32} \times \text{bool}$  where  $'T$  is numeric
- $\forall \alpha, (X:\alpha). (X:\alpha) \times \alpha \times \alpha \rightarrow \alpha \times \alpha$
- $\forall \gamma_1, \gamma_2, \gamma_3. \gamma_1 \times \gamma_2 \times \gamma_3 \rightarrow \gamma_1 \cup \gamma_2 \cup \gamma_3 \times \gamma_2 \cup \gamma_3$
- $\forall \delta_1, \delta_2. 0 \times \delta_1 \times \delta_2 \rightarrow \delta_1 \delta_2 \times \delta_2$

Then its application to the following arguments is valid.

```

clock h1:bool;
clock h2,o2: bool when h1;
x,y,o1: int32 when h1;
o1,o2= F(h2,pre x + y, 2 * y);

```

The subtyping constraint on 'T is satisfied by its third parameter. This application does not introduce a memory. The fastest rate of its application is h1, and the expressions are all based on the same rate as specified in the clock type. o1 depends on h,x,y, while o2 only depends on x,y. This application implies that h2 must be well-defined on its first instant of activation, that o1 is not well-defined, and that o2 is well-defined.

**Example 2:** Incorrect operator application since the type of the size parameter instance cannot be inferred from the context

---

```
function f<<n >>(a: int32^n) returns (o: int32)
  o = if n < 3 then -1 else a[n -3];
function N_incorrect (a1: int32^10) returns (o1: int32)
  o1 = (f <<10>>)(a1);
```

---

It can be fixed by adding a cast operator to specify the type of the size argument:

---

```
function N (a1: int32^10) returns (o1: int32)
  o1 = (f < <(10: int8)>>)(a1);
```

---

**Example 3:** Incorrect operator application since the size argument is not within the range of the expected type

---

```
function f<<n >>(a: int32^n) returns (o: int32)
  o = if (n: int8) < 3 then -1 else a[n -3];
function N_incorrect (a1: int32^10) returns (o1: int32)
  o1 = (f < <129 > >)(a1);
```

---

**Example 4:** Incorrect operator application since the second parameter of a call to N1 must be a clock identifier

---

```
function ex1(iI1: int16; clock clk1: bool)
  returns(iO1: int16 when clk1)
  let
    iO1 = N1(iI1 when clk1, not clk1);
  tel
function N1(iI1: int16 when clk1; clock clk1: bool)
  returns(iO1: int16 when clk1)
  let
    iO1 = iI1;
  tel
```

---



---

**Example 5:** The first `activate` can be used to implement a control-flow switch

---

```
function imported N(i: int32) returns (b: bool);
node ex2(iI1: int32; clock clk1: bool)
returns (iO1: bool)
let
  iO1 = merge (clk1; (activate N every clk1)(iI1 + 1);
              (activate N every not clk1)(fby(iI1; 1; 0)));
tel
```

---

**Example 6:** Contrary to conditional blocks, the second and third `activate` operators only require a Boolean condition. Enumerated types are not allowed and the following example is false.

---

```
type T = enum {a, b, c};

node ex3(bI1: T)
returns (iO1: int32)
let
  iO1 = (activate N every (bI1 match a) default 3)(bI1);
tel
```

---

**Example 7:** The accumulator of a `fold` iterator can be undefined at the first instant. However, when it is well-defined, the input operator cannot increase its delay type. In the example below, an initialization error is raised because of the delay profile of N.

---

```
node ex4(aI1: int32)
returns (rO1: float32)
let
  rO1 = (fold N <<3>>)(0.0, aI1^3);
tel

node N(rI1: float32; iI2: int32)
returns (rO1: float32)
let
  rO1 = pre rI1 - pre float32 iI2;
tel
```

---

**Example 8:** Iterators can be applied to predefined operators. Operator `fold` can be applied to comparison operators in case of Boolean arrays, but not in other cases. It is required that the first argument of input operators be of the same type as its resulting one. This is not the case for comparison, where the result is always a Boolean value.

---

```
function Fold_020 (aI1, aI2:int32^3; aB3:bool^4)
returns (iO1:int32; iO2:int32^3; bO3:bool)
let
  iO1 = (fold $+$ <<3>>) (0, aI1);
  iO2 = (map $mod$ <<3>>)(aI1, aI2);
  bO3 = (fold $=$ <<4>>) (true, aB3);
tel
```

---

**Example 9:** Incorrect iterator since the type of the first argument of the input operator of `mapi` cannot be an unconstrained type variable

---

```
function f(idx:'t; i: float32) returns (o: float32) where 't integer
  o = i + (idx: float32);
function N(aI1: float32^3) returns (aO1: float32^3)
let
  aO1 = (mapi f <<3>>)(aI1);
tel
```

---

## RELATED TOPICS

- Section 4.3 [“Group Declarations”](#)
- Section 8.4 [“Operations on Arrays and Structures”](#)
- Section 6.1 [“User-Defined Operators”](#)
- Section 6.3 [“Clock Expressions”](#)
- Section 4.2 [“Type Expressions”](#)

## 8.6 Primitive Operator Associativity and Relative Priority

The table below gives the priorities of expression operators from highest to lowest. Let  $N1$ ,  $N2$ , and  $N3$  be variables and  $op1$  and  $op2$  two primitive operators. The expression  $N1\ op1\ N2\ op2\ N3$  is equivalent to  $(N1\ op1\ N2)\ op2\ N3$  if the priority of  $op1$  is greater than or equal to that of  $op2$ ; otherwise it is equivalent to  $N1\ op1\ (N2\ op2\ N3)$ .

.
[]
@
^
not lnot
when
reverse
pre
+ − (unary)
* / mod
+ − (binary)
lsl lsr
= < > <= >= < >
and
or xor
land
lor lxor
- >
if
times (right associative)

Except **times**, all the infix operators are left associative.

**Example:**

```
a = x + y + z ; -- equivalent to a = (x + y) + z ;
a = x + y * z ; -- equivalent to a = x + (y * z) ;
a = x times y times z ; -- equivalent to a = x times (y times z) ;
```



# Appendixes

- Appendix A: ["Formalization"](#)
- Appendix B: ["Backus-Naur-Form"](#)
- Appendix C: ["Mapping between Textual and Graphical Representations"](#)
- Appendix D: ["Bibliography"](#)
- ["Index"](#)



# A

## Formalization

This appendix presents the analyses performed on Scade models. Analysis of the Scade 6 language is done in several phases:

- A-1 [“Namespace Analysis”](#)
- A-2 [“Type Analysis”](#)
- A-3 [“Clock Analysis”](#)
- A-4 [“Causality Analysis”](#)
- A-5 [“Initialization Analysis”](#)

### A-1 Namespace Analysis

- A.1.1 [“Purpose”](#)
- A.1.2 [“Principles”](#)

#### A.1.1 Purpose

The namespace analysis checks the proper definition and use of identifiers. The namespace analysis ensures two properties: any used object is declared in the current context, and there is no name clash.

#### A.1.2 Principles

To perform this analysis, identifiers are collected into two naming environments: one for package identifiers and one for declaration identifiers. When adding a new identifier to one of these environments (depending on the syntactical construct used to declare this identifier), a membership procedure has to check that this name has not already been used in the corresponding environment: two packages of the same level cannot have the same name, a type and a sensor cannot have the same name, etc.

However, objects in different namespaces or at different levels can share the same name: a package and a constant, a subpackage and its father package. Conversely, when using an identifier, the membership procedure must guarantee that this name belongs to the corresponding naming environment.

The initial declaration environment contains the predefined operator names defined in section 2.3 [“Symbol List”](#) and the reserved keywords of the language defined in section 2.4 [“Keyword List”](#), preventing them from being reused anywhere else in the program. The initial package naming environment is empty.

Identifiers declared at the top-level of a program are appended to this initial environment. See 3.3 [“Declarations”](#) to have an exhaustive presentation of these declarations. A special case occurs for labels defined in a structure type (see section 4.2 [“Type Expressions”](#)): they can overload an identifier defined in the declaration environment.

In order to allow the reuse of identifiers, the namespace analysis takes place within *scopes*. A scope is a limited context within which the two properties above have to be proved. Several scopes are successively entered and left during this analysis. Basically, a new scope is entered in every package declaration (see Chapter 3 about [“Program and Packages”](#)), and every user-defined operator (see Chapter 6 about [“User-Defined Operators”](#)).

When entering a scope, the non-membership constraint for new identifiers is relaxed: an identifier is allowed to overload an already existing one in its corresponding naming environment. Any reference to this identifier will be bound to the latest declaration.

However, the package and the user operator constructs define different policies concerning the ability to use identifiers that belong to the father scope and that have not been overloaded:

- Any package scope (even subpackages) starts with both initial environments, thus no identifier can be directly accessed. Nevertheless, a naming mechanism using *paths* allows to clearly refer to different package identifiers.
- An operator scope extends the current declaration environment. Identifiers declared at the package level belonging to this environment can be directly accessed. The same path mechanism can allow to refer to different package identifiers. Identifiers declared in this scope must be distinct from each other, but can overload identifiers from a previous scope.

When leaving a scope, added identifiers are removed. Of course, an exception is made for the identifier of the declaration that created this scope. For instance, an operator identifier can be referred to anywhere but in the scope it defines.



**Example 1:** Errors due to multiple use of the same identifier in an environment at the same level: Two packages at the same level cannot have the same name. Similarly, a type and a constant at the same level cannot have the same name.

---

```
package P1 type speed = int64;
... const speed: speed;
end;
package P1
...
end;
```

---

**Example 2:** The package namespace and the declaration namespace being separated, identifiers can be shared between them. This program is correct.

---

```
package P1
...
end;
const P1: int32=0;
```

---

**Example 3:** Overloading of a constant identifier belonging to the previous scope, preventing its use in the current scope. The second use of `b` will raise an error.

---

```
const b: int32 =0;
node ex(x: int32; b: bool) returns (y: int32)
let
  y = if b then x else b -> pre y;
tel;
```

---

## A-2 Type Analysis

- A.2.1 [“Purpose”](#)
- A.2.2 [“Precondition”](#)
- A.2.3 [“Principles”](#)

### A.2.1 Purpose

The aim of this analysis is to statically verify that every flow and every operator on flows are used consistently with their data types.

For instance, in the following example:

---

```
node N(e: bool) returns (s: bool)
let
  s = e and 42;
tel;
```

---

The operator **and** of type  $\text{bool} \times \text{bool} \xrightarrow{0} \text{bool}$ , which supposes to produce a Boolean value when given two Boolean expressions on input, is used with an integer as its second argument which is not consistent with its type.

### A.2.2 Precondition

This system applies to a program that has been proved correct with respect to its usage of names, as defined by the namespace analysis defined in [“Namespace Analysis”](#) on page 123.

### A.2.3 Principles

Type checking performs an analysis of all the expressions contained in a Scade program according to a typing environment. The typing environment follows the traversed scopes as defined in [“Namespace Analysis”](#) on page 123: anytime a declaration identifier is added, a corresponding typing information is added to the typing environment. This information is removed along a removal of this identifier while leaving a scope.

The initial typing environment contains all the typing information concerning some of the recognized lexemes defined in section 2.2 [“Lexemes”](#) (those defining numerical or character values), and the predefined Scade operators.

The type system underlying Scade is a simplified parametric polymorphism. Predefined operators can be associated either with fully instantiated types (such as the one for operator **and** above) or universally quantified types when several instance types may be used with this operator. For instance, operator **if** has the following type:

$$\text{if.then.else.} : \forall \tau. \text{bool} \times \tau \times \tau \xrightarrow{0} \tau$$

It means that given a Boolean expression for the condition and two objects of same type  $\tau$ , this operator builds an object of type  $\tau$ . Polymorphism means that the last two arguments can be based on any type. The 0 index above the arrow expresses the fact that this operator does not need to refer to a past value in order to compute its output.

Type analysis relies on two operations on types: type equivalence and sub-typing.

- 1 Decision of *type equivalence* is needed in order to check that the application of arguments to an operator is valid, either because this operator requires a specific type as input or because some arguments must share the same type. This is the case in the example above where second and third arguments of operator **if** must share the same type  $\tau$ , whatever this type is. Type equivalence between types  $\tau_1$  and  $\tau_2$  consists in checking that type constructors (array and structure constructors) occur in the same places with the same number of arguments, and that atomic data type (**intN**, **uintN**, **floatN**, **bool**, **char**, plus external data types) are syntactically equal. During type equivalence, user defined type names are aliased to their corresponding type definition when it exists. External data types are taken as atomic types. This type equivalence is postponed in presence of parametrized arrays.

In this case, size constraints are generated: given two array types  $\tau_1^n$  and  $\tau_2^{n'}$ , they are equivalent if and only if  $\tau_1$  and  $\tau_2$  are equivalent and  $n=n'$  also. This latter size constraint is replicated when the size parameters  $n$  and  $n'$  of the user operator are instantiated with allowed size expressions (can also refer to other size variables). A size constraint is satisfied if the evaluation of its left and right parts are equal.

- 2 Decision for *subtyping* is needed to check that a given type correctly instantiates a type variable of an operator profile that cannot range over every type. In the above example for operator **if**, any type fits the  $\tau$  variable. Consider the example of an arithmetic operator, which is given type:

$$.+ : \forall \tau. \tau \times \tau \rightarrow \tau \text{ where } \tau \text{ numeric}$$

To be valid, an instantiation of this operator requires two unstructured numerical values of type: integer, float, or any equivalent type. Such a sub typing constraint can be stated by the user when defining an operator (see 6.1 [“User-Defined Operators”](#)).

The type analysis also provides each user operator with a type. In the same scope as this operator's declaration, the type analysis can then check the correction of an instantiation of this operator. This part of the analysis is detailed in section 6.1 [“User-Defined Operators”](#).

Another role of the type system is to check the correct use of literals. An integer (resp. float) literal can have any integer (resp. float) type which is inferred from its use. If the inference fails to determine this type, then the literal is ill-typed. The type system also checks that literals of a predefined type are within the range of this type.

## A-3 Clock Analysis

- A.3.1 [“Purpose”](#)
- A.3.2 [“Precondition”](#)
- A.3.3 [“Principles”](#)

### A.3.1 Purpose

Flows can have different rates in the sense that they are not required to produce a value at each cycle, they may also be absent. It is possible to filter the values of a flow on a Boolean condition to obtain a *slower* one. Thus, not all the flows have the same *length* and combining them leads to some questions, for instance let us consider the streams *a* and *b*:

a	1	3	5	7	11	13	17	21	...
b		5		10		6		12	...

*a* is twice as fast as *b*, can we build the flow *a+b*? To do this, one needs some synchronization mechanism such as a buffer to memorize the items of *a* when *b* is not present. In this example, it is clear that the buffer would have one more item every two cycles (based on the rate of *a*) and thus would not be bounded.

The aim of the clock calculus is to avoid such a combination (*a+b*) of different rates that may require an unbounded amount of memory to be implemented.

The point here is to give an answer to this very general problem; the clock calculus is a simple solution expressed here as a type system. It has the advantage of providing a clear and simple characterization of the rejected programs, and the inconvenience of rejecting more programs than other techniques.

### A.3.2 Precondition

This system applies on a program that is correctly typed in the sense of the type system presented in [“Type Analysis”](#) on page 126.

### A.3.3 Principles

Clock checking works in a similar way as type analysis, but based on a specific clock typing system. Every expression of a Scade program is analyzed according to a clocking environment that follows the declaration naming environment. The initial clocking environment contains the clocking information of the predefined Scade operators.

The available clock expressions described in section 6.3 [“Clock Expressions”](#) allow to construct only three kinds of clocking information:  $\_ \text{ on } X$ ,  $\_ \text{ on not } X$ , and  $\_ \text{ on } X \text{ match } P$ , where  $\_$  represents the clock of the local context this expression appears into,  $X$  is the name of a declared clock (see 6.2 [“Variable Declarations”](#)), and  $P$  is a pattern from an enumerated type. This type system is natively polymorphic on the rate of execution. It allows every operator to be used on different rates in different clocking contexts. Hence, contrary to typing information, no fully instantiated clocking type is associated with predefined operators. Consider the simple operator  $\rightarrow$  with this clock type:

$$\_ \rightarrow \_ : \forall \alpha. \alpha \times \alpha \rightarrow \alpha$$

It means that given two objects on clock  $\alpha$ , this operator builds an object on the same clock. This is the most simple kind of clocking type for an operator.

Consider now the clocking type of **merge** operator:

$$\text{merge} \dots \forall \alpha. (X:\alpha)(X:\alpha) \times \alpha \text{ on } X \times \alpha \text{ on not } X \rightarrow \alpha$$

Given a clock identifier  $X$  based on any clock  $\alpha$ , an object sampled on  $X$  (for instance using the **when** construct) and an object sampled on the complementary clock, this operator builds an output based on clock  $\alpha$ .

The clock analysis relies on the ability to check the clock equivalence of two clock expressions. Decision of clock equivalence is needed in order to check that the application of arguments to an operator has a valid clock type because some arguments are based on the same clock, as for operator  $\rightarrow$  above.

Clock equivalence relies on a syntactical analysis of the clock expressions involved. Contrary to type analysis, clock analysis forces clock expressions to be syntactically equal: two aliased clocks or semantically equal are not considered as being equivalent. During this analysis, global flows and literals are given the clock of the local contexts they are used in. The clock of the local context is given by an abstract clock when

entering a user operator. This clock is modified only when entering a clocked block or a State Machine. Within such a control structure, the environment is filtered in order to remove the declarations that are not on a rate compatible with this control block. Removed identifiers can thus no more be used within this block. When leaving the block though, these identifiers are recovered. The removed identifiers are either those that are on a slower clock than the clock used to control the block ( $\alpha$  on  $h$  is slower than  $\alpha$ ), or those that are on an non-comparable one ( $\alpha$  on  $h1$  is not comparable with  $\alpha$  on  $h2$ ). This filtering operation comes from the fact that clocks implement control.

Note that since literals, global variables, and sensors shall always be available (thus have the clock of the context they are used in), there is no need to sample them on the same clock as the condition if needed. The clock analysis also provides each user operator with a clock type, universally quantified on an abstract clock variable. In the same scope as this operator's declaration, the clock analysis can then check the correction of the various instantiations of this operator. This part of the analysis is detailed in 6.1 [“User-Defined Operators”](#).

**Example 1:** Clock analysis is performed on a syntactical basis. Even though the clock variable  $k$  is semantically equivalent to clock variable  $h$ ,  $y$  is wrong clocked.

---

```
clock h: bool;
clock k: bool;
k = if true then h else false;
y = x when h + x when k;
```

---

**Example 2:** Given the declarations in the first three lines, the remaining expressions are wrong clocked. Operator `or` requires its arguments to share the same clock, which is not the case for `e pre x` and `y`. Similarly, literal `false` is given the clock of the local context, here the general abstract clock  $a$ , while `pre x` is given the same clock as  $x$ , that is  $a$  on  $h$ , while operator `->` requires arguments based on the same one. Finally, when entering the conditional block, the clocking environment is filtered in order to remove all the identifiers based on a clock that is not as fast or comparable to that of  $y$ . Thus  $x$  is removed and cannot be used within a branch of the definition of  $t$ .

---

```
clock h: bool;
var x: bool when h;
var y: bool;

z = x or y;
u = false -> pre x;
activate if y
  then t = if x then 1 else 0;
  else t = -1;
returns t;
```

---

## A-4 Causality Analysis

- A.4.1 [“Purpose”](#)
- A.4.2 [“Precondition”](#)
- A.4.3 [“Principles”](#)

### A.4.1 Purpose

The causality analysis aims at verifying that no flow instantaneously depends on itself. In other words, a flow cannot appear without `pre` or `last` (defined in 8.1.1 [“Identifiers”](#) and 8.2 [“Sequential Operators”](#)) in its own definition expression. For instance, a program like:

---

```
...
nat = 1 + nat;
...
```

---

is not causal and is rejected by the analysis. This is fair as there is no integer flow `nat` whose value at cycle `n` is equal to itself plus one! A causal definition of `nat` would be:

---

```
...
nat = 1 -> (1 + pre nat);
...
```

---

This kind of instantaneous cyclic definition may appear directly (as in the example above) or indirectly through other named flows or node instances.

### A.4.2 Precondition

This system applies to a program that is correctly typed in the sense of the type system presented in [“Type Analysis”](#) on page 126.

### A.4.3 Principles

Causality analysis performs an analysis of all the flow definitions contained in a user operator. As for previous analysis, the causality information is given as a typing information, according to a specific type system. A causality environment following the declaration naming environment is built while analyzing user operators. The initial causality environment contains the causality types of all predefined operators.

The causality type system is based on sets of variable witnesses (*i.e.*, identifiers of variables) and a union operation. Consider the causality type of operator ‘+’:

$$+. : \forall \gamma_1, \gamma_2. \gamma_1 \times \gamma_2 \rightarrow \gamma_1 \cup \gamma_2$$

Given two objects of causality  $\gamma_1$  and  $\gamma_2$ , this operator produces an object which depends on both of its inputs (*i.e.*, that has causality type  $\gamma_1 \cup \gamma_2$ ).

This analysis relies on the ability to solve, at the operator level, a set of causality constraints generated during the analysis of its body part. These constraints are either inclusion constraints (stating that a causal type or a variable identifier is included in another causal type), or non membership constraints (stating that a defined variable, local, or output, does not belong to its own causal type). These constraints are generated while:

- declaring a variable, stating that this variable does not belong to its causality type;
- instantiating an operator: for each input of the operator, one has to check that the causal types of the actual parameters are included in the type of the formal parameters;
- defining an equation: the causal type of the right hand side has to be included in the causal type of the left hand side.
- traversing a clocked block or a State Machine: the control variables or variables appearing in transitions are added to current causality types

The causality analysis provides each user operator with a causality type, universally quantified on an abstracted input causality variables. In the same scope as this operator's declaration, the causality analysis can then check the correction of the various instantiations of this operator. This part of the analysis is detailed in 6.1 [“User-Defined Operators”](#).



## A-5 Initialization Analysis

- A.5.1 [“Purpose”](#)
- A.5.2 [“Precondition”](#)
- A.5.3 [“Principles”](#)

### A.5.1 Purpose

This analysis aims at verifying that a model produces defined values, in other words that this model is deterministic in the sense that for given input sequences, the output sequences are completely defined (without *Nil*).

The unit delay operator **pre** is able to shift a flow of values, without specifying the first value of the shifted flow. For example:

a	1	2	3	4	...
pre a	<i>nil</i>	1	2	3	...

The flow **pre a** (of `int32` type) has an undefined value (called *Nil* in the semantics) at the very first cycle. The *flow initialization*  $\rightarrow$  provides a way to specify the first value of a flow to fill this unspecified hole.

a	1	2	3	4	...
21 $\rightarrow$ pre a	21	1	2	3	...

In practice, this initialization of flow may be detached from a call to a past value with operator **pre**. For instance, the discrete derivation can be initialized without initializing the delayed flow one want to derive:  $0 \rightarrow (x - \text{pre } x)$ .

This analysis consists in checking that a flow is never delayed more than once without initializing the first value; thus the expression **pre (pre x)** is rejected while **pre (1  $\rightarrow$  pre x)** is accepted. The basic idea is to check that directly or not (through node instances, flow equalities), this situation never occurs. This idea extends to the access of the latest value of a shared flow (**last 'x**) that may not be defined at the very first cycle.

### A.5.2 Precondition

This system applies on a program that is correctly typed in the sense of the type system presented in [“Type Analysis”](#) on page 126.

### A.5.3 Principles

Initialization analysis performs an analysis of all the Scade expressions contained in a program. As for previous analysis, the initialization information is given as a typing information, according to a specific type system. Two initialization environments following the declaration naming environment are built while analyzing user operators. The first one collects initialization information of the declared variables. The second one collects initialization information of the **last** expression of these declarations when it exists (see 6.2 [“Variable Declarations”](#)). Both initial initialization environments contain the initialization types of all predefined operators.

The initialization type system is based on two literals **0** and **1** (for expressions that introduce either no delay or one delay), and a join operation to compute the maximum delay of two types. Consider, the initialization type of operator '+':

$$. + . : \forall \delta_1, \delta_2. \delta_1 \times \delta_2 \rightarrow \delta_1 \sqcup \delta_2$$

Given two objects of initialization  $\delta_1$  and  $\delta_2$ , this operator produces an object that has the maximum delay from both its arguments, namely it has initialization type  $\delta_1 \sqcup \delta_2$ . The associative and commutative operator is defined by:

$$\forall d, \quad 1 \sqcup d = 1$$

$$\forall d, \quad 0 \sqcup d = d$$

The initialization analysis provides each user operator with an initialization type, universally quantified on an abstracted input initialization variables. In the same scope as this operator's declaration, the initialization analysis can then check the correction of the various instantiations of this operator. This part of the analysis is detailed in 6.1 [“User-Defined Operators”](#).

**Example 1:** Consider the following example:

---

```
node N1(x: int16) returns (z: int16)
var y: int16;
let
  y = 0 -> pre z;
  z = pre x;
tel
```

---

The initialization environment starts with the following abstract bindings:  $x:\delta_1, y:\delta_2, z:\delta_3$ . When typing the first equation:  $y = 0 \rightarrow \text{pre } z$ , since operator **pre** requires its argument to have type '**0**',  $\delta_3$  is '**0**'. This equation is well typed of type '**0**', then leading to

unify  $\delta_2$  with '0'. The second equation assigns type '0' to  $x$  because of the use of operator **pre**. This equation is well typed with type '1', then leading to unify the type of  $z$  with '1'. An error is raised since  $\delta_3$  has already been unified with '0'.

**Example 2:** Consider now the following example:

---

```
node N2(x, y: int16) returns (z: int16)
let
  z = pre x + y;
tel
```

---

This node is well typed according to the initialization analysis. It requires its input variable  $x$  to be of type '0', and its output variable  $z$  to be of type '1'. This means that this operator produces a delay value. Its generalized initialization type is:

N2       $\forall \delta \cdot 0 \times \delta \rightarrow 1$



# B

## Backus-Naur-Form

This appendix presents the syntax used for the following elements of the language:

- B-1 [“Declarations”](#)
- B-2 [“User-Defined Operators”](#)
- B-3 [“Expressions”](#)

### B-1 Declarations

```
program ::= {{ decls }}

decls ::= open path ;
        | package_decl
        | group_block
        | type_block
        | const_block
        | sensor_block
        | user_op_decl
```

- [“Packages”](#)
- [“Types”](#)
- [“Groups”](#)
- [“Globals”](#)

#### PACKAGES

```
path ::= ID {{ :: ID }}
path_id ::= [[ path :: ]] ID
package_decl ::= package [[ visibility ]] ID
                {{ decls }} end ;
visibility ::= private
            | public
external ::= imported
interface_status ::= [[ visibility ]] [[ external ]]
```

## TYPES

```

type_block ::= type {{ type_decl ; }}
type_decl ::= interface_status ID [[ = type_def ]] [[ is numeric_kind ]]
type_def ::= type_expr
              | enum { ID {{ , ID }} }
numeric_kind ::= numeric | float | integer | signed | unsigned
type_expr ::= bool
              | signed << expr >> | int8 | int16 | int32 | int64
              | unsigned << expr >> | uint8 | uint16 | uint32 | uint64
              | float32|float64
              | char
              | path_id
              | typevar
              | { field_decl {{ , field_decl }} }
              | type_expr ^ expr
field_decl ::= ID : type_expr
typevar ::= NAME

```

## GROUPS

```

group_block ::= group {{ group_decl ; }}
group_decl ::= [[ visibility ]] ID = group_expr
group_expr ::= ( type_expr {{ , type_expr }} )

```

## GLOBALS

```

const_block ::= const {{ const_decl ; }}
const_decl ::= interface_status ID : type_expr [[ = expr ]]

sensor_block ::= sensor {{ sensor_decl ; }}
sensor_decl ::= ID {{ , ID }} : type_expr

```

## B-2 User-Defined Operators

- [“Variable Declarations”](#)
- [“Operators”](#)
- [“Scope Declarations”](#)
- [“Equations”](#)
- [“Control Blocks”](#)
- [“State Machines”](#)

### VARIABLE DECLARATIONS

```

var_decls ::= var_id {{ , var_id }} : type_expr [[ when_decl ]]
                                                [[ default_decl ]]
                                                [[ last_decl ]]

var_id ::= [[ clock ]] [[ probe ]] ID
when_decl ::= when clock_expr
default_decl ::= default = expr
last_decl ::= last = expr

```

### OPERATORS

```

user_op_decl ::= op_kind interface_status ID [[ size_decl ]]
               params returns params {{ where_decl }} [[ spec_decl ]]
               opt_body

op_kind ::= function
          | node
size_decl ::= << [[ ID {{ , ID }} ]] >>
params ::= ( [[ var_decls {{ ; var_decls }} ]] )
where_decl ::= where typevar {{ , typevar }} numeric_kind
spec_decl ::= specialize path_id
opt_body ::= ;
            | equation ;
            | [[ signal_block ]]
              [[ local_block ]]
            | let {{ equation ; }} tel [[ ; ]]

```

## SCOPE DECLARATIONS

```

data_def ::= equation ;
          | scope
scope ::= [[ signal_block ]] [[ local_block ]] [[ eqs ]]
signal_block ::= sig ID {{ , ID }} ;
local_block ::= var {{ var decls ; }}
eqs ::= let {{ equation ; }} tel

```

## EQUATIONS

```

equation ::= simple_equation
          | assert
          | emission
          | control_block return
simple_equation ::= lhs = expr
lhs ::= ( )
      | lhs_id {{ , lhs_id }}
lhs_id ::= ID
        | -
assert ::= assume ID : expr
        | guarantee ID : expr
control_block ::= state_machine
               | clocked_block
emission ::= emit emission_body
emission_body ::= NAME [[ if expr ]]
               | ( NAME {{ , NAME }} ) [[ if expr ]]
return ::= returns returns_var
returns_var ::= {{ ID , }} (( ID | .. ))

```

## CONTROL BLOCKS

```

clocked_block ::= activate [[ ID ]] (( if_block | match_block ))
if_block ::= if expr then (( data_def | if_block ))
           | else (( data_def | if_block ))
match_block ::= when expr match {{ | pattern : data_def }}+

```



## STATE MACHINES

```

state_machine ::= automaton [[ ID ]] {{ state_decl }}+
state_decl ::= [[ initial ]] [[ final ]] state ID
               [[ unless {{ transition ; }}+ ]]
               data_def
               [[ until {{ transition ; }}
               [[ synchro [[ actions ]] fork ; ]] ]]
transition ::= if expr arrow
arrow ::= [[ actions ]] fork
fork ::= target
        | if expr arrow {{ elsif_fork }} [[ else_fork ]] end
elsif_fork ::= elsif expr arrow
else_fork ::= else arrow
target ::= restart ID
         | resume ID
actions ::= do { [[ emit ]] emission_body
               {{ ; [[ emit ]] emission_body }} }
         | do data_def

```

## B-3 Expressions

```

clock_expr ::= id
            | not id
            | ( id match pattern )
expr ::= id_expr
       | atom
       | list_expr
       | tempo_expr
       | arith_expr
       | relation_expr
       | bool_expr
       | array_expr
       | struct_expr
       | mixed_constructor
       | switch_expr
       | apply_expr
       | depend_expr
       | atom ::= bool_atom
                  | CHAR
                  | INTEGER
                  | FLOAT
                  | TYPED_INTEGER
                  | TYPED_FLOAT
bool_atom ::= true
            | false
id_expr ::= path_id
          | NAME
          | last NAME
list_expr ::= ( list )
list ::= [[ expr {{ , expr }} ]]

```

- [“Sequential”](#)
- [“Combinatorial”](#)
- [“Arrays and Structures”](#)
- [“Flow Switches”](#)
- [“Operator Application and Higher-Order Patterns”](#)

## SEQUENTIAL

```
tempo_expr ::= pre expr
            | expr -> expr
            | fby ( list ; expr ; list )
            | expr times expr
            | expr when clock_expr
            | merge ( expr ; list {{ ; list }} )
```

## COMBINATORIAL

```
arith_expr ::= unary_arith_op expr
            | expr bin_arith_op expr
            | ( expr : type_expr )
unary_arith_op ::= - | + | lnot
bin_arith_op ::= + | - | * | / | mod | land | lor | lxor | lsl | lsr
relation_expr ::= expr bin_relation_op expr
bin_relation_op ::= = | <> | < | > | <= | >=
bool_expr ::= not expr
            | expr bin_bool_op expr
            | # ( list )
bin_bool_op ::= and | or | xor
```

## ARRAYS AND STRUCTURES

```
array_expr ::= reverse expr
            | expr @ expr
            | expr index
            | expr [ expr .. expr ]
            | ( expr . {{ label or index }}+ default expr )
            | transpose ( expr ; INTEGER ; INTEGER )
            | expr ^ expr
            | [ list ]
struct_expr ::= expr . ID
            | { label_expr {{ , label_expr }} }
mixed_constructor ::= ( expr with {{ label_or_index }}+ = expr )
label_expr ::= ID : expr
            | index ::= [ expr ]
label_or_index ::= . ID
                | index
```

## FLOW SWITCHES

```

switch_expr ::= if expr then expr else expr
              | ( case expr of {{ case_expr }}+ )
case_expr  ::= | pattern : expr
pattern    ::= path_id
              | CHAR
              | [[ - ]] INTEGER
              | [[ - ]] TYPED_INTEGER
              | bool atom
              | _

```

## OPERATOR APPLICATION AND HIGHER-ORDER PATTERNS

```

apply_expr ::= operator ( list )
operator   ::= prefix
              | ( prefix << [[ expr {{ , expr }} ]] >> )
              | ( make path_id )
              | ( flatten path_id )
              | ( iterator operator << expr >> )
              | ( activate operator every clock_expr )
              | ( activate operator every expr default expr )
              | ( activate operator every expr initial default expr )
              | ( restart operator every expr )
              | ( iterator_mw operator << expr >>
                  if expr default expr )
              | ( foldw operator << expr >> if expr )
              | ( foldwi operator << expr >> if expr )
prefix     ::= path_id
              | PREFIXOP
iterator    ::= map | fold | mapi | foldi
              | mapfold [[ INTEGER ]]
              | mapfoldi [[ INTEGER ]]
iterator_mw ::= mapw | mapwi
              | mapfoldw [[ INTEGER ]]
              | mapfoldwi [[ INTEGER ]]

```



# C

## Mapping between Textual and Graphical Representations

This appendix presents the graphical mapping between the Scade 6 textual language and the graphical representation used in SCADE Suite® IDE.

This mapping is partial and based on classical kinds of representations such as: *data-flow diagrams*, *hierarchical State Machines*, and *decision diagrams*. From a language point of view this appendix covers the part of Scade 6 accessible from the following non-terminal symbols of the syntax: *simple\_equation*, *spec\_equation*, *emission*, *state\_machine*, and *clocked\_block* (for a complete syntax description, see Appendix B about [“Backus-Naur-Form”](#)).

- C-1 [“General Syntax”](#)

### DATA-FLOW REPRESENTATIONS:

- C-2 [“Arithmetic Operators”](#)
- C-3 [“Bitwise Arithmetic Operators”](#)
- C-5 [“Comparison Operators”](#)
- C-6 [“Logical Operators”](#)
- C-7 [“Temporal Operators”](#)
- C-8 [“Choice Operators”](#)
- C-9 [“Structure and Array Operators”](#)
- C-10 [“Higher-Order Operators”](#)
- C-11 [“Other Design Elements”](#)

### CONTROL FLOW REPRESENTATIONS:

- C-12 [“State Machines”](#)
- C-13 [“Conditionnal Blocks”](#)

## C-1 General Syntax

The graphic syntax of Scade language supports the graphical representation of equations in SCADE Editor. Any graphical representations is associated with:

- Each reference to an element from the left side of an equation (output, local variable, terminator)
- Each reference to an element from the right side of an equation (input, hidden input, local variable, probe, constant, operators)
- Each link between references

Writing a textual equation in graphical format may require prior conversion of the equation into several textual equations with the addition of internal variables.

**Example:** Consider the textual description below where `_L1`, `_L2`, `_L3` are internal variables:

---

```
_L1 = e1 ;  
_L2 = O1 (_L1) ;  
s1 = _L2 ;
```

---

- Equation `_L1 = e1` results from the creation of a reference to input `e1`.
- Equation `_L2 = O1 (_L1)` results from a call to operator `O1` and the link of its input with a reference to `e1`.
- Equation `s1 = _L2` results from the link between the output of operator `O1` call and the reference to output `s1`.

A more compact description of this equation system can be:

---

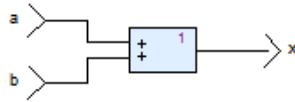
```
s1 = O1 (e1) ;
```

---

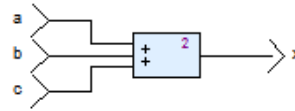
## C-2 Arithmetic Operators

### C.2.1 Operator +

$x = a + b ;$

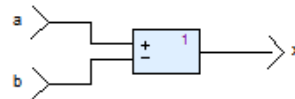


$y = a + b + c ;$



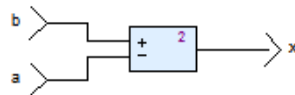
### C.2.2 Operator - (binary minus)

$x = a - b ;$



using symmetric symbol

$x = a - b ;$



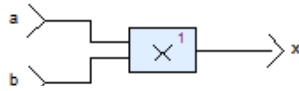
### C.2.3 Operator - (unary minus)

$c = - (a) ;$

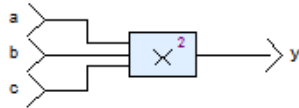


### C.2.4 Operator \*

`x = a * b ;`

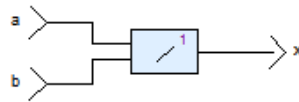


`y = a * b * c ;`



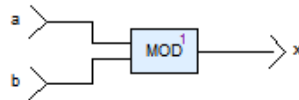
### C.2.5 Operator / (polymorphic division)

`x = a / b ;`



### C.2.6 Operator mod

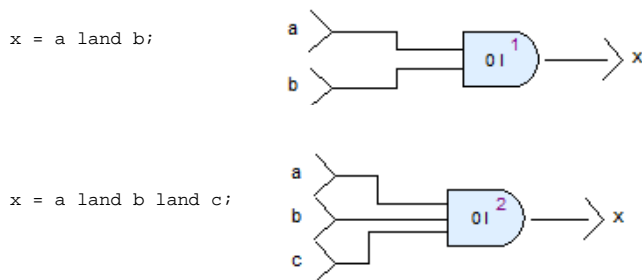
`x = a mod b ;`



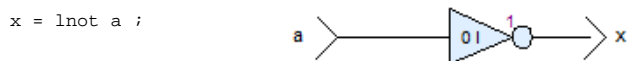


## C-3 Bitwise Arithmetic Operators

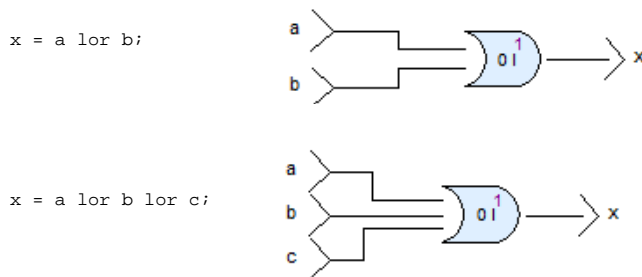
### C.3.1 Operator land



### C.3.2 Operator lnot

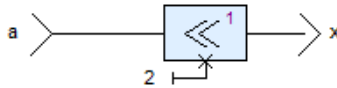


### C.3.3 Operator lor



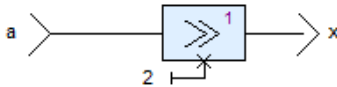
### C.3.4 Operator lsl (logical left shift)

`x = a lsl 2;`



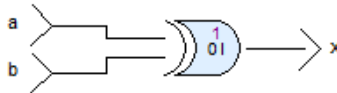
### C.3.5 Operator lsr (logical right shift)

`x = a lsr 2;`



### C.3.6 Operator lxor

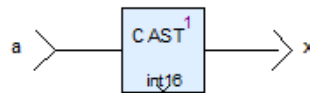
`x = a lxor b;`



## C-4 Conversion Operators

### C.4.1 Operator numeric cast

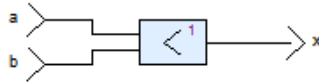
`x = (a: int16) ;`



## C-5 Comparison Operators

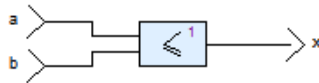
### C.5.1 Operator <

`x = a < b ;`



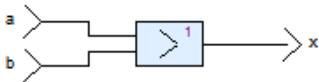
### C.5.2 Operator <=

`x = a <= b ;`



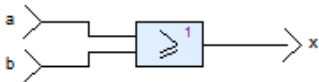
### C.5.3 Operator >

`x = a > b ;`



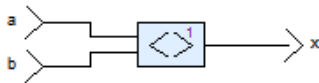
### C.5.4 Operator >=

`x = a >= b ;`



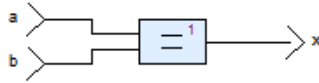
### C.5.5 Operator <>

`x = a <> b ;`



### C.5.6 Operator =

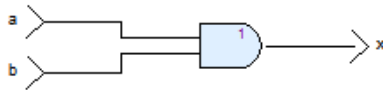
`x = a = b ;`



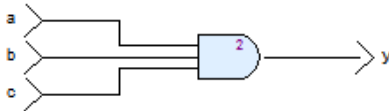
## C-6 Logical Operators

### C.6.1 Operator and

`x = a and b ;`

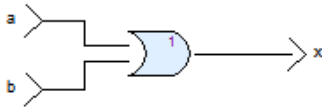


`y = a and b and c ;`

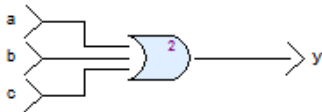


### C.6.2 Operator or

`x = a or b ;`

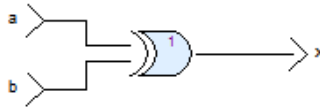


`y = a or b or c ;`



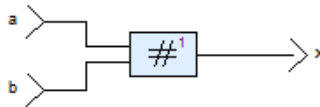
### C.6.3 Operator xor

`x = a xor b ;`

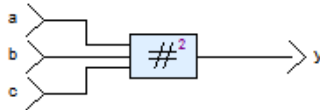


### C.6.4 Operator #

`x = #(a,b) ;`



`y = #(a,b,c) ;`



### C.6.5 Operator not

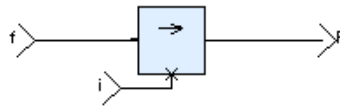
`x = not(a) ;`



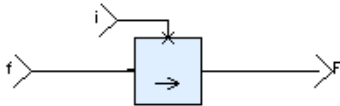
## C-7 Temporal Operators

### C.7.1 Operator $\rightarrow$

$F = i \rightarrow f ;$



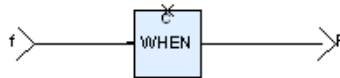
using symmetric symbol



### C.7.2 Operator when

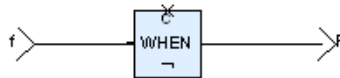
$F = (f) \text{ when } C ;$

with clock



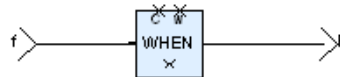
not clock

$F = (f) \text{ when } (\text{not } C) ;$



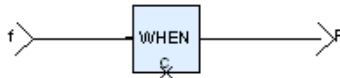
with clock match value

$F = (f) \text{ when } (C \text{ match } v1) ;$



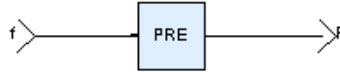
using symmetric symbol

$F = (f) \text{ when } C ;$



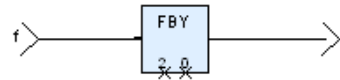
### C.7.3 Operator pre

`F = pre(f) ;`



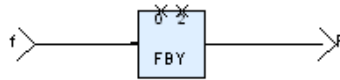
### C.7.4 Operator fby

`F = fby (f;2;0) ;`



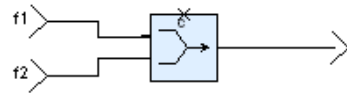
using symmetric symbol

`F = fby (f;0;2) ;`

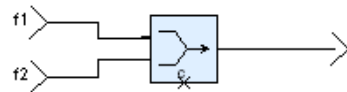


### C.7.5 Operator merge

`f = merge (c;f1;f2) ;`

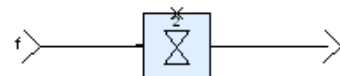


using symmetric symbol



### C.7.6 Operator times

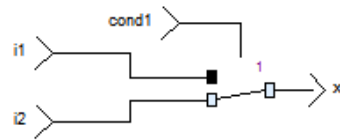
`F = 2 times f ;`



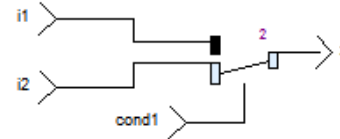
## C-8 Choice Operators

### C.8.1 Operator if ... then ... else ...

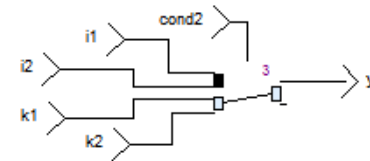
```
x = if cond1 then (i1) else (i2) ;
```



using symmetric symbol



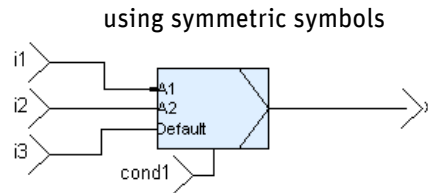
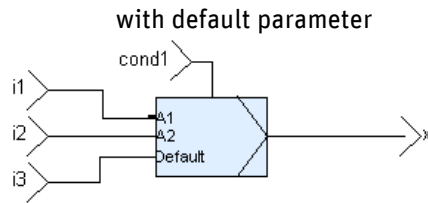
```
x,y = if cond2 then (i1,i2) else (k1,k2) ;
```



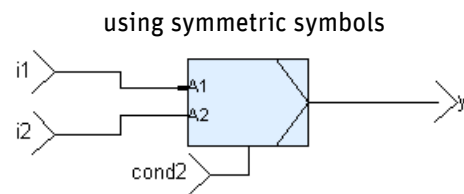
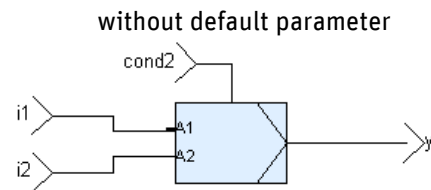


## C.8.2 Operator case ... of ...

```
x = ( case cond1 of
| A1 : i1
| A2 : i2
|   : i3 ) ;
```



```
y = ( case cond2 of
| A1 : i1
| A2 : i2 ;
```

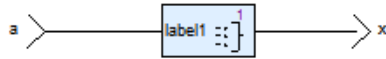


## C-9 Structure and Array Operators

### C.9.1 Data Structure Constructor: Value Enumeration

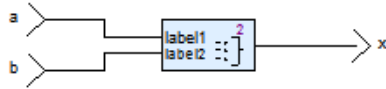
Building a structure from a list of values.

```
x = { label1 : a }
```



with multiple lists

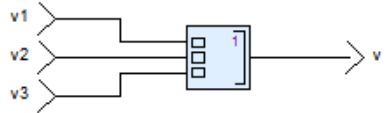
```
x = { label1 : a, label2 : b }
```



### C.9.2 Array Constructor: Value Enumeration

Building an array from a list of values.

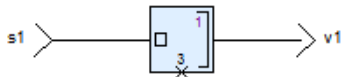
```
v = [v1, v2, v3] ;
```



### C.9.3 Array Constructor: Value Repetition

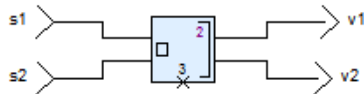
Using a factor to build an array by repeating the same value.

```
v1 = s1^3 ;
```



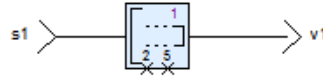
with multiple input values

```
v1 , v2 = (s1^3 , s2^3) ;
```



### C.9.4 Array Access: Static Slice

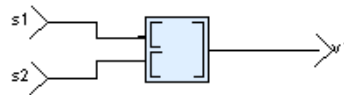
```
v1 = w1[2 .. 5] ;
```



### C.9.5 Array Concatenation

v1 is the concatenation of s1 and s2 arrays.

```
v1 = s1 @ s2 ;
```



### C.9.6 Array Reverse

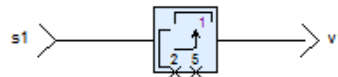
```
v1 = reverse s1 ;
```



### C.9.7 Array Transpose

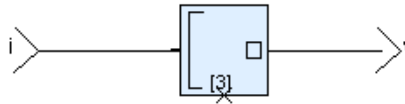
v1 is equal to s1 by exchanging the 2<sup>nd</sup> and 5<sup>th</sup> dimensions.

```
v1 = transpose (s1;2;5) ;
```

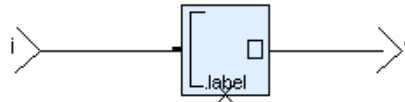


### C.9.8 Structure and Array Access: Static Indexation

```
v = i[3] ;
```

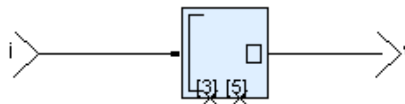


```
v = i.label ;
```



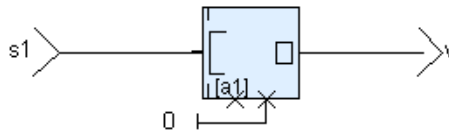
with multiple indexes

```
v = i[3][5] ;
```



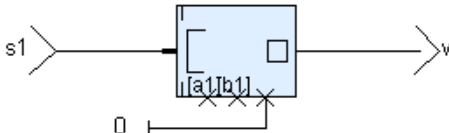
### C.9.9 Array Access: Dynamic Indexation

```
v = (s1.[a1] default 0) ;
```



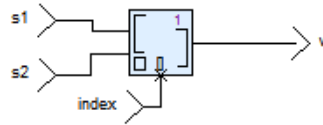
with multiple indexes

```
v = (s1.[a1][b1] default 0) ;
```

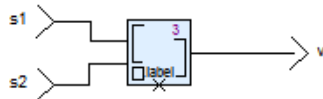


### C.9.10 Structure and Array Constructor: Array Copy with Modification

```
v = (s1 with [index] = s2) ;
```

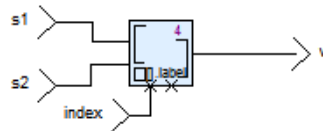


```
v = (s1 with .label = s2) ;
```



with combination of index and labels

```
v = (s1 with [index].label = s2) ;
```

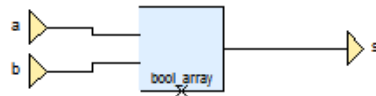


### C.9.11 Make and Flatten

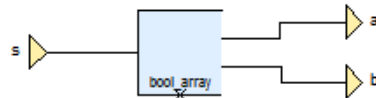
Assuming the following type and variables:

```
type bool_array = [label1 : bool , label2 : int16] ;
var a : bool ;
b : int16^2 ;
s : bool_array ;
```

```
s = (make bool_array)(a, b) ;
```



```
a, b = (flatten bool_array)(s) ;
```



## C-10 Higher-Order Operators

In the following subsections, the following definitions are used:

---

```

node N (a, b: int16) returns (c: int16)
node N1 (a, b, c: int16) returns (d: int16)
node N2 (a, b: int16) returns (c: bool, d: int16)
node N3 (a, b: int16) returns (c: int16; d: bool)
node N4 (a, b, c: int16) returns (d: bool; e: int16)
Node N5 (a, b, c: int16) returns (d, e: int16)
Node N6 (a, b, c: int16) returns (d: bool; e, f: int16)

```

---

and the following variables:

---

```

var v1, v2, w: int16 ^10;
v, w_default: int16;
acc : int16;
idx : int16;
cond : bool;
Enum: [enum type]

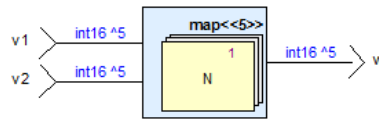
```

---

### C.10.1 Iterators: map, fold, and mapfold

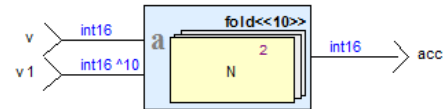
- Operator map:

```
w = (map N << 5 >>) (v1, v2);
```



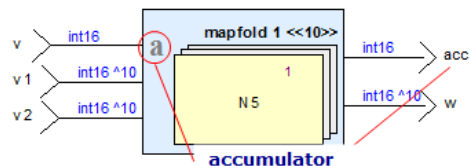
- Operator fold:

```
acc = (fold N << 10 >>) (v, v1);
```



- Operator mapfold:

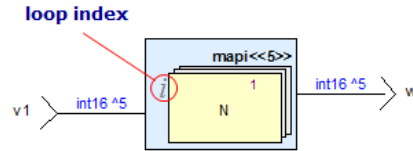
```
acc, w = (mapfold N5 << 10 >>) (v, v1, v2);
```



## C.10.2 Iterator with Access to Index: mapi, foldi, mapfoldi

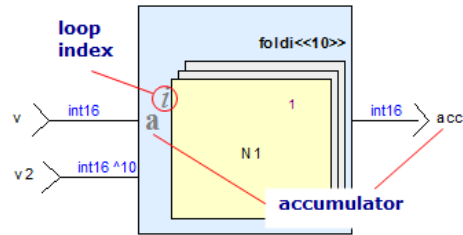
- Operator mapi:

```
w = (mapi N << 5 >>) (v1);
```



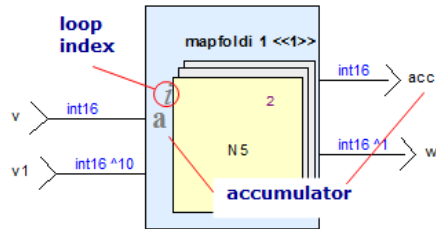
- Operator foldi:

```
acc = (foldi N1 << 10 >>) (v, v2);
```



- Operator mapfoldi:

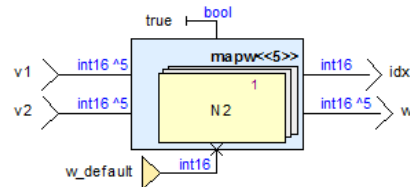
```
acc, w = (mapfoldi N5 << 10 >>) (v, v1);
```



### C.10.3 Partial Iterators: mapw, mapwi, foldw, foldwi, mapfoldw, mapfoldwi

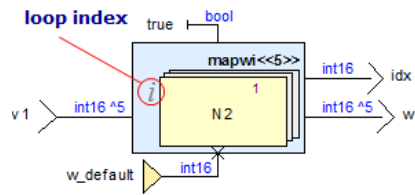
- Operator mapw:

```
idx, w = (mapw N2 << 5 >> if true default
w_default) (v1, v2);
```



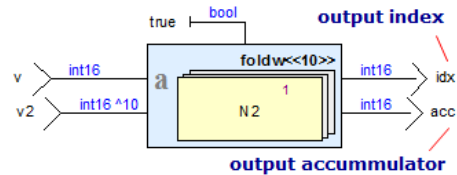
- Operator mapwi:

```
idx, w = (mapwi N2 << 10 >> if true default
w_default) (v1);
```



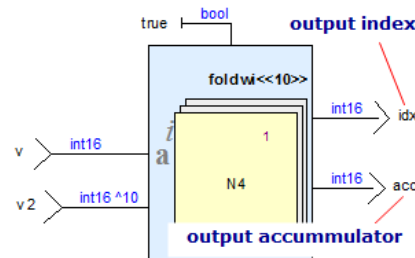
- Operator foldw:

```
idx, acc = (foldw N2 << 10 >> if true) (v, v2);
```



- Operator foldwi:

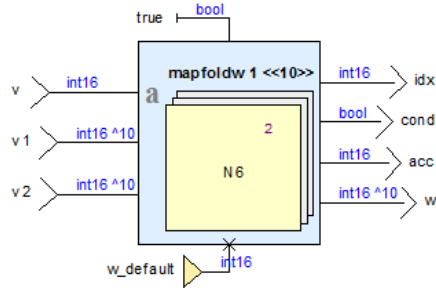
```
idx, acc = (foldwi N4 << 10 >> if true) (v,
v2);
```





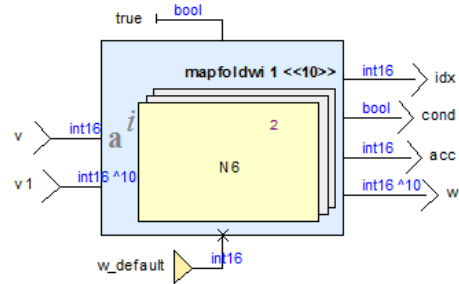
- Operator `mapfoldw`:

```
idx, cond, acc, w = (mapfoldw N6 << 10 >> if
true default w_default) (v, v1, v2);
```



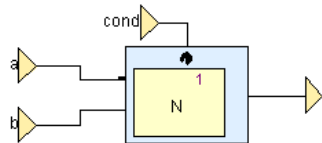
- Operator `mapfoldwi`:

```
idx, cond, acc, w = (mapfoldwi N << 10 >> if
true default w_default) (v, v1);
```



## C.10.4 Resettable Node Instantiation

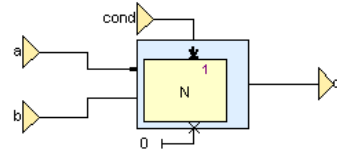
```
d = (restart N every cond) (a, b);
```



### C.10.5 Conditional Activation of Node Instantiation with Default Values

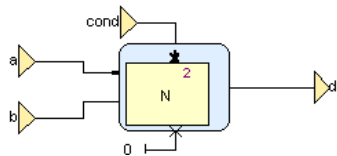
- Activation with default (no memory):

```
d = (activate N every cond default 0) (a, b) ;
```



- Activation with initial default (implies memory):

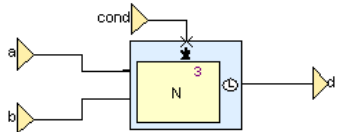
```
d = (activate N every cond initial default 0)  
(a, b) ;
```



### C.10.6 Conditional Activation of Node Instantiation on Clock

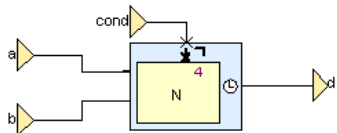
- Activation on Boolean clock:

```
d = (activate N every cond) (a, b) ;
```

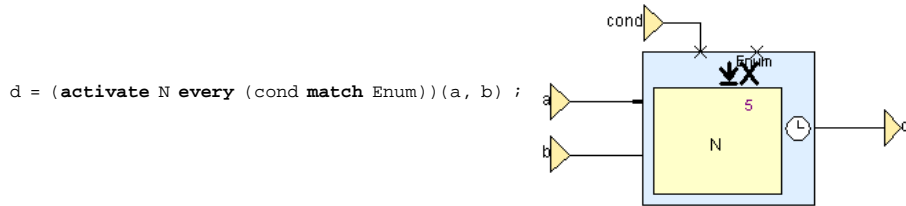


- Activation on complementary Boolean clock:

```
d = (activate N every not cond) (a, b) ;
```



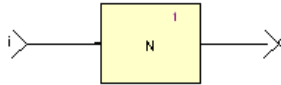
- Activation on enumerated clock:



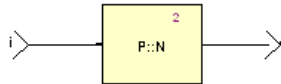
## C-11 Other Design Elements

### C.11.1 Operator Calls

`o = N ( i ) ;`



`o = P::N ( i ) ;`

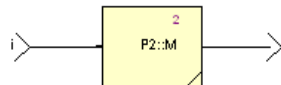


### C.11.2 Operator Call from Library

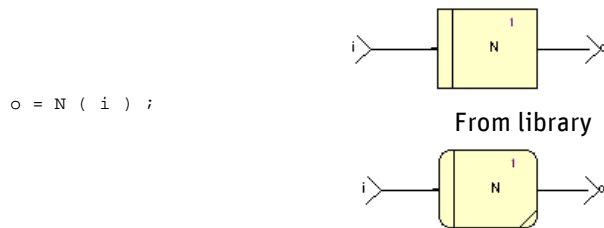
`o = M ( i ) ;`



`o = P2::M ( i ) ;`



### C.11.3 Imported Operator Calls



### C.11.4 Parameterized Operator Calls



### C.11.5 Input Reference



### C.11.6 Output Reference



### C.11.7 Consumed Local Variable Reference




### C.11.8 Produced Local Variable Reference



### C.11.9 Probe Reference

`Probe1 =` 

### C.11.10 Constant Reference

`= Package::Constant1` 

### C.11.11 Textual Expression Reference

`0.0` 


### C.11.12 Terminator Reference

`-` 

### C.11.13 Assume

`assume A1 : v<>0 ;` 

### C.11.14 Guarantee

`guarantee A1 : v<>0 ;` 

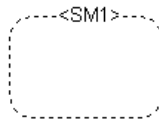
## C-12 State Machines

### C.12.1 State Machine

```

automaton SM1
returns .. ;

```



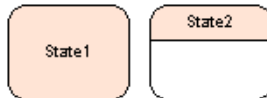
### C.12.2 States

- States:

```

state State1
state State2
  let
  tel

```

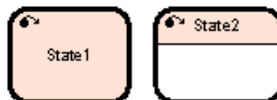


- Initial states:

```

initial state State1
initial state State2
  let
  tel

```

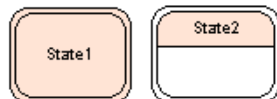


- Final states:

```

final state State1
final state State2
  let
  tel

```

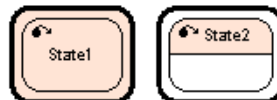


- Initial final states:

```

initial final state State1
initial final state State2
  let
  tel

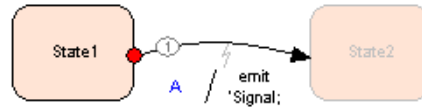
```



### C.12.3 Transition

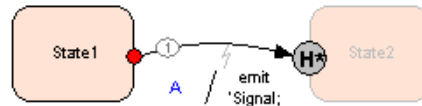
- Strong transition with target state reset:

```
state State1
  unless if A do let emit 'Signal; tel
restart State2 ;
```



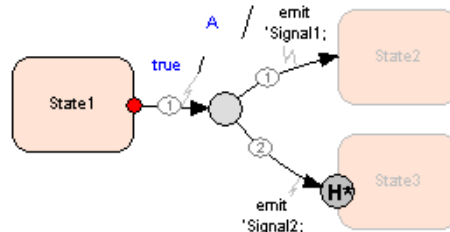
- Strong transition with target state resumed:

```
state State1
  unless if A do let emit 'Signal; tel
resume State2 ;
```



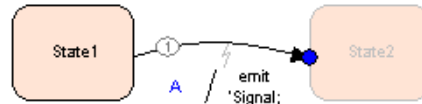
- Fork strong transition with target states reset or resumed:

```
state State1
  unless
    if true
      if A do let emit 'Signal1; tel restart
State2
    else do let emit 'Signal2; tel resume
State3
  end;
```



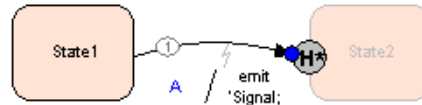
- Weak transition with target state reset:

```
state State1
  until if A do let emit 'Signal; tel
restart State2 ;
```



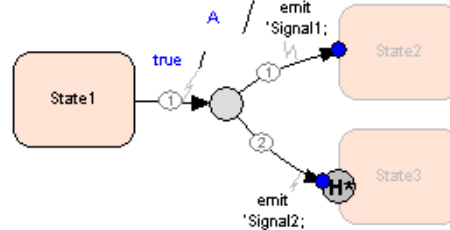
- Weak transition with target state resumed:

```
state State1
  until if A do let emit 'Signal; tel
resume State2 ;
```



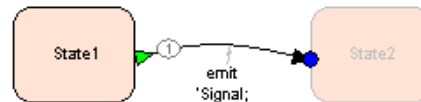
- Fork weak transition with target states reset or resumed:

```
state State1
  until
    if true
      if A do let emit 'Signal1; tel restart
State2
      else do let emit 'Signal2; tel resume
State3
      end;
end;
```



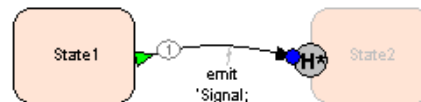
- Synchro transition with target state reset:

```
state State1
  until
    synchro do let emit 'Signal; tel
restart State2 ;
```



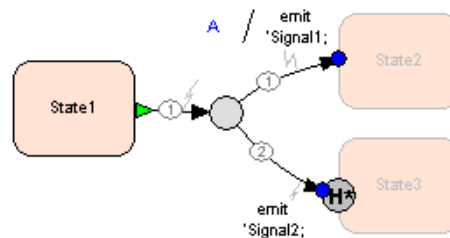
- Synchro transition with target state resumed:

```
state State1
  until
    synchro do let emit 'Signal; tel resume
State2 ;
```



- Fork synchro transition with target states reset or resumed:

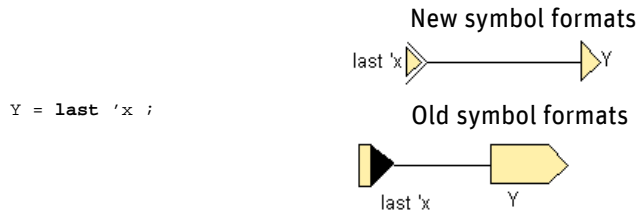
```
state State1
  until
    synchro
    if A do let emit 'Signal1; tel restart
State2
    else do let emit 'Signal2; tel resume
State3
    end;
end;
```



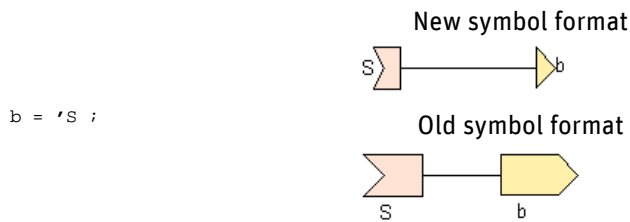


### C.12.4 Last Value, Signal Emission and Test

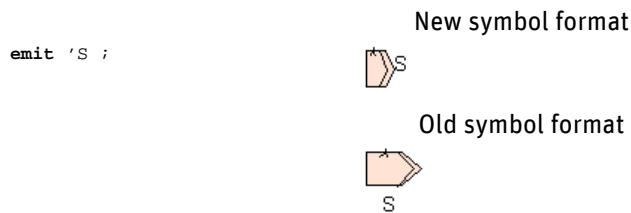
- **Last value:** Access to last of a flow



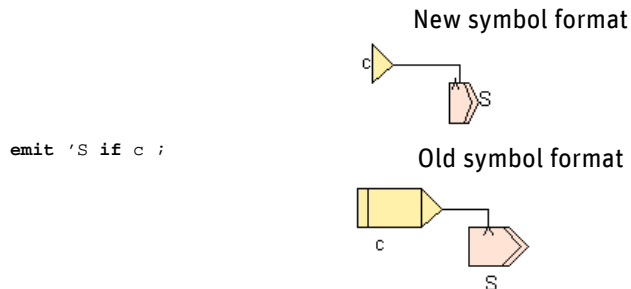
- **Status of a signal:** From control to data-flow: access to signal status



- **Signal emission:** From data-flow to control



- **Signal emission with guard:**

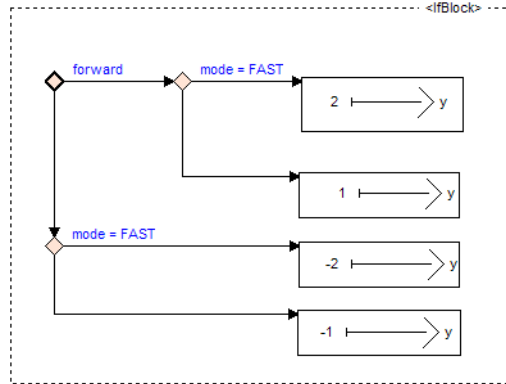


## C-13 Conditionnal Blocks

### C.13.1 Operator If Block

With forward, mode: enum SLOW, FAST.

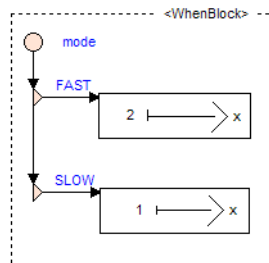
```
activate IfBlock if forward
  then if mode = FAST
  then
    let y = 2; tel
  else
    let y = 1; tel
  else if mode = FAST
  then
    let y = -2; tel
  else
    let y = -1; tel
returns ..
```



### C.13.2 Operator When Block

With mode: enum SLOW, FAST.

```
activate WhenBlock when mode match
  | FAST : let x = 2; tel
  | SLOW : let x = 1; tel
returns ..;
```



- [1] D. Harel and A. Pnueli. *On the Developments of Reactive Systems. In Logic And Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.
- [2] P. Caspi and N. Halbwachs and D. Pilaud and J. Plaice, Lustre: a declarative language for programming synchronous systems, 14th ACM Symposium on Principles of Programming Languages, 1987.
- [3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [4] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [5] Gérard Berry. The Constructive Semantics of Pure Esterel. Draft Version 3, July 1999.
- [6] Nicolas Halbwachs. Synchronous programming of reactive systems, Kluwer Academic Pub. 1993.
- [7] D. Harel. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3:231–275, 1987.
- [8] Charles André. Representation and Analysis of Reactive Behaviors: A synchronous Approach. Invited Paper at CESA’96.
- [9] Marc Pouzet. Lucid Synchronne, un langage synchrone d’ordre supérieur. Mémoire d’Habilitation à diriger des recherches, Université Paris 6, 2002.
- [10] Paul Caspi and Marc Pouzet, Synchronous Kahn Networks. ACM SIGPLAN International Conference on Functional Programming, 1996, Philadelphia, Pennsylvania.
- [11] Marc Pouzet. Lucid Synchronne, version 3. Tutorial and reference manual, Université Paris-Sud, LRI, April 2006, Distribution available at: [www.lri.fr/~sim.pouzet/lucid-synchronne](http://www.lri.fr/~sim.pouzet/lucid-synchronne)
- [12] Grégoire Hamon. Calcul d’horloges et structures de contrôle dans Lucid Synchronne, un langage de flots synchrones à la ML. PhD. Thesis, Université Paris 6.
- [13] Yann Rémond. Un support langage pour les modes de fonctionnement des systèmes temps-réel : extension de LUSTRE par des automates de modes. PhD. Thesis, Université Grenoble I, 2001.
- [14] Lionel Morel Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille PhD. Thesis, Université Grenoble I, 2005.

- [15] Jean-Louis Colaço et Marc Pouzet Prototypage, Rapport final du projet GENIE II, Verilog SA, janvier 2000
- [16] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.
- [17] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.
- [18] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):245–255, August 2004.
- [19] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State
- [20] Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.
- [21] Jean-Louis Colaço and Grégoire Hamon and Marc Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems, In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October, 2006.
- [22] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.
- [23] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.
- [24] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.
- [25] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with signal. *Another Look at Real Time Programming, Proceedings of the IEEE*, Special Issue, Sept. 1991.
- [26] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91(1):64–83, January 2003.
- [27] C. André. Synccharts: A visual representation of reactive behaviors. Technical report tr-95-52, I3S/CNRS, Sophia-Antipolis, France, 1990.

# Index

---

## Symbols

- (minus)
  - syntax and semantics [84](#)
- > (init)
  - dynamic semantics [80](#)
  - syntax and semantics [78](#)
- [ ] operator
  - and array construction [92](#)
  - and static projection [90](#)
  - and static slicing [91](#)
- @ (append)
  - syntax and semantics [90](#)
- \* (multiplication)
  - syntax and semantics [84](#)
- # (sharp)
  - dynamic semantics [84](#)
  - syntax and semantics [83](#)
- ^ operator
  - in array expressions [92](#)
  - in array type expression [20](#)
- + (plus)
  - syntax and semantics [84](#)
- < (less than)
  - syntax and semantics [87](#)
- <= (less or equal)
  - syntax and semantics [87](#)
- <> (different)
  - syntax and semantics [87](#)
- = (equal)
  - syntax and semantics [87](#)
- > (greater than)
  - syntax and semantics [87](#)
- >= (greater or equal)
  - syntax and semantics [87](#)

## A

- activate (Boolean clock)
  - dynamic semantics [109](#)
  - syntax and semantics [103](#)
- activate (initial default)
  - dynamic semantics [110](#)
  - syntax and semantics [103](#)
- activate (with default)
  - dynamic semantics [110](#)
  - syntax and semantics [103](#)
- Analysis
  - principles for causality [131](#)
  - principles for clock [129](#)
  - principles for initialization [134](#)
  - principles for namespace [123](#)
  - principles for typing [126](#)
- and
  - dynamic semantics [84](#)
  - syntax and semantics [83](#)
- Arithmetic operators
  - declaration syntax and semantics [84](#)
- Array expressions
  - declaration syntax and semantics [90](#)
  - indexes syntax [97](#)
- Array types
  - semantics [20](#)
- Asserts
  - declaration semantics [46](#)
- Atomic values
  - declaration syntax and semantics [75](#)
- Attributes
  - on top-level declarations [12](#)

## B

- Backus-Naur-forms
  - extended notation [3](#)
- Binary operators
  - declaration syntax and semantics [84](#)
- Boolean operators
  - declaration syntax and semantics [83](#)

## C

- case
  - dynamic semantics [89](#)
  - syntax and semantics [88](#)
- Causality analysis [131](#)
- clock
  - semantic attribute in variable declaration [36](#)
- Clock analysis [128](#)
- Clock checking
  - principles [129](#)
- Clock equivalence [129](#)
- Clock expressions
  - syntax and semantics [39](#)
- Clocking environment [129](#)
- Clocks
  - semantic analysis [128](#)
- Combinatorial operators
  - declaration syntax and semantics [83](#)
- Comments
  - syntax description [5](#)
- Conditional blocks

# Index

---

- declaration syntax and semantics 48
- Constant blocks
  - declaration and attributes 25
- Constants
  - declaration syntax and semantics 25
  - visibility attribute semantics 12
- Constructors
  - labels and indexes syntax 97
- Constructors (array and structure)
  - declaration syntax and semantics 96
- Constructs (language)
  - array expressions 90
  - clock expressions 39
  - combinatorial operators 83
  - conditional blocks 48
  - constants 25
  - constructors (array and structure) 96
  - equations 44
  - group expressions 23
  - groups 22
  - higher order operators 98
  - packages 8
  - Scade lexical elements 3
  - scopes 41
  - sensors 27
  - sequential operators 78
  - state machines 53
  - structure expressions 95
  - type expressions 19
  - types 16
  - user-defined operators 30
  - variables 36
- Contracts
  - assertion declaration semantics 46
- Control blocks
  - declaration semantics 47
- Cyclic definitions 131
- D**
- Data types
  - consistency analysis 126
- Data typing
  - syntax and semantics 15
- Declaration blocks 7
- Declarations
  - and namespace analysis 123
  - array expressions 90
  - atomic values 75
  - attribute syntax and semantics 12
  - clock expressions 39
  - combinatorial operators 83
  - conditional blocks 48
  - constants 25
  - constructors (array and structure) 96
  - from host language 12
  - group expressions 23
  - groups 22
  - higher order operators 98
  - identifiers 72
  - imported attribute semantics 12
  - lists of expressions 76
  - operator as function 30
  - operator as node 30
  - packages 8
  - parameterized operator 30
  - private attribute semantics 12
  - public attribute semantics 12
  - scopes 41
  - sensors 27
  - sequential operators 78
  - specialization semantics 31
  - state machines 53
  - state transitions 58
  - structure expressions 95
  - top-level syntax and semantics 9
  - transition actions 60
  - type expressions 19
  - types 16
  - user-defined operators 30
  - variables 36
- Default value
  - variable declaration semantics 37
- E**
- Equation blocks
  - concept and declaration semantics 44
- Equations
  - syntax and semantics 44
- Expressions
  - as simple equation 45
  - atomic value syntax and semantics 75
  - identifier syntax and semantics 72
  - list syntax and semantics 76
  - syntax and semantics 71
- F**
- fbv

# Index

---

- dynamic semantics [81](#)
- syntax and semantics [79](#)
- flatten
  - dynamic semantics [107](#)
  - syntax and semantics [100](#)
- Flow switches
  - declaration syntax and semantics [88](#)
- Flows
  - and clock expressions [39](#)
  - at different rates [128](#)
  - cyclic definitions and causality [131](#)
- fold
  - dynamic semantics [108](#)
  - syntax and semantics [101](#)
- foldi
  - dynamic semantics [108](#)
  - syntax and semantics [102](#)
- foldw
  - dynamic semantics [112](#)
  - syntax and semantics [105](#)
- foldwi
  - dynamic semantics [113](#)
  - syntax and semantics [105](#)

## G

- Global flows
  - syntax and semantics [25](#)
- Group blocks
  - declaration and attributes [22](#)
- Group expressions
  - syntax and semantics [23](#)
- Groups

- declaration syntax and semantics [22](#)
- visibility attribute semantics [12](#)

## H

- Higher order operators
  - declaration syntax and semantics [98](#)
- Host language
  - importing in Scade programs [12](#)

## I

- Identifiers
  - declaration syntax and semantics [72](#)
- if\_then\_else
  - dynamic semantics [89](#)
  - syntax and semantics [88](#)
- Imported
  - declaration attribute semantics [12](#)
- Imported operators
  - specialization semantics [31](#)
- Indexes
  - syntax [97](#)
- Initialization
  - semantic analysis [133](#)
- Initialization analysis [133](#)

## K

- Keywords

- list of reserved [5](#)
- activate (Boolean clock) [103](#)
- activate (initial default) [103](#)
- activate (with default) [103](#)
- assume [46](#)
- bool [19](#)
- case [88](#)
- char [19](#)
- const [25](#)
- default [37](#)
- else [58](#)
- elsif [58](#)
- emit [46](#)
- end [58](#)
- enum [49](#)
- fbv [79](#)
- final [54](#)
- flatten [100](#)
- float [17](#)
- float32 [19](#)
- float64 [19](#)
- floatN [20](#)
- fold [101](#)
- foldi [102](#)
- foldw [105](#)
- foldwi [105](#)
- group [22](#)
- guarantee [46](#)
- if [58](#)
- imported [12](#)
- initial [54](#)
- int16 [19](#)
- int32 [19](#)
- int64 [19](#)
- int8 [19](#)
- integer [17](#)
- intN [20](#)
- land [84](#)
- last [37](#)
- let [32](#), [41](#)

# Index

---

lnot 84  
lor 84  
lsl 85  
lsr 85  
lxor 84  
make 100  
map 100  
mapfold 101  
mapfoldi 102  
mapfoldw 106  
mapfoldwi 107  
mapi 102  
mapw 104  
mapwi 104  
merge 80  
numeric 17  
pre 78  
restart 58, 104  
resume 58  
reverse 90  
sensor 27  
sig 41  
signed 17, 19  
synchro 54  
tel 32, 41  
times 79  
transpose 91  
type 16  
uint16 19  
uint32 19  
uint64 19  
uint8 19  
uintN 20  
unless 54  
unsigned 17, 19  
until 54  
var 41  
when 79

## L

### Labels

syntax 97

### land

syntax and semantics 84

### Language

mapping with graphical  
symbols 145  
syntax as Backus-Naur  
form 137  
syntax description 3

### Last value

variable declaration  
semantics 37

### Lexemes

see Lexical elements

### Lexical elements

comment syntax 5  
keyword list 5  
pragma syntax 5  
symbol list 4  
syntax description 3  
value description 4

### Lists of expressions

declaration syntax and  
semantics 76

### lnot

syntax and semantics 84

### Local blocks

declaration and attributes 41

### Locals

see Scopes

### lor

syntax and semantics 84

### lsl

syntax and semantics 85

### lsr

syntax and semantics 85

### lxor

syntax and semantics 84

## M

### make

dynamic semantics 107  
syntax and semantics 100

### map

syntax and semantics 100

### mapfold

dynamic semantics 108  
syntax and semantics 101

### mapfoldi

dynamic semantics 109  
syntax and semantics 102

### mapfoldw

dynamic semantics 113  
syntax and semantics 106

### mapfoldwi

dynamic semantics 114  
syntax and semantics 107

### mapi

dynamic semantics 108  
syntax and semantics 102

### mapw

dynamic semantics 111  
syntax and semantics 104

### mapwi

dynamic semantics 112  
syntax and semantics 104

### merge

dynamic semantics 82



# Index

---

syntax and semantics 80

## Model structure

package syntax and  
semantics 7

## N

Name clash 123

## Namespace

see Packages  
semantic analysis 123

Namespace analysis 123

## not

dynamic semantics 83  
syntax and semantics 83

## Notations

as Backus-Naur forms 137  
comments 5  
describing values 4  
extended Backus-Naur-form 3  
keywords 5  
pragmas 5  
regular expressions 3  
symbols 4

## O

## Objects

use and namespace  
analysis 123

## open

directive for package  
declaration 9

## Opening packages

static semantics 9

## or

dynamic semantics 83  
syntax and semantics 83

## P

Package body 7

## Packages

declaration syntax and  
semantics 8  
extending declaration  
environment 9  
visibility attribute  
semantics 12

## Parameterized operator

dynamic semantics 107

## Path

semantics for package  
declarations 8

## Pragmas

syntax description 5

## pre

dynamic semantics 80  
syntax and semantics 78

## Predefined types

expression semantics 20

## Primitives

- (minus) 84  
-> 78, 80  
@ (append) 90  
\* (multiplication) 84  
# (sharp) 83, 84  
+ (plus) 84  
< (less than) 87  
<= (less or equal) 87  
<> (different) 87  
= (equal) 87  
> (greater than) 87

>= (greater or equal) 87  
relative priority 119  
activate (Boolean clock) 103  
activate (initial default) 103  
activate (with default) 103  
and 83, 84  
case 88  
fby 79, 81  
flatten 100  
fold 101  
foldi 102  
foldw 105  
foldwi 105  
if\_then\_else 88  
land 84  
lnot 84  
lor 84  
lsl 85  
lsr 85  
lxor 84  
make 100  
map 100  
mapfold 101  
mapfoldi 102  
mapfoldw 106  
mapfoldwi 107  
mapi 102  
mapw 104  
mapwi 104  
merge 80, 82  
not 83  
or 83  
pre 78, 80  
restart 104  
reverse 90  
times 79, 81  
transpose 91  
when 79, 81  
xor 83, 84

Primitives (arithmetic)

# Index

---

- syntax and semantics [84](#)
- Primitives (array operations)
  - syntax and semantics [90](#)
- Primitives (boolean logic)
  - syntax and semantics [83](#)
- Primitives (flow switches)
  - syntax and semantics [88](#)
- Primitives (higher order)
  - syntax and semantics [98](#)
- Primitives (mixed operations)
  - syntax and semantics [96](#)
- Primitives (relational)
  - syntax and semantics [87](#)
- Primitives (structure operations)
  - syntax and semantics [95](#)
- Primitives (temporal)
  - syntax and semantics [78](#)
- Private
  - declaration attribute
  - semantics [12](#)
- Private packages [8](#)
  - see also Visibility [12](#)
- probe
  - semantic attribute in variable declaration [36](#)
- Program
  - top-level declarations [9](#)
- Public
  - declaration attribute
  - semantics [12](#)

## R

- Rate of system

- see Clock expressions
- Rates
  - and flows [128](#)
- Regular expressions
  - notation [3](#)
- Relational operators
  - declaration syntax and semantics [87](#)
- restart
  - dynamic semantics [110](#)
  - syntax and semantics [104](#)
- reverse
  - syntax and semantics [90](#)
- Root operator
  - in program semantics [7](#)

## S

- Scade language
  - combining flows with expressions [71](#)
  - comment syntax [5](#)
  - conditional control expression [48](#)
  - contracts [46](#)
  - equation block expressions [44](#)
  - expressing data and control flow [43](#)
  - global flows [25](#)
  - introduction to lexical elements [3](#)
  - native type system [15](#)
  - pragma syntax [5](#)
  - recognized symbols [4](#)
  - reserved keywords [5](#)
  - state machine control expression [53](#)

- syntax as Backus-Naur form [137](#)
- textual and graphical mapping [145](#)
- top-level constructs and packages [7](#)
- user-defined operators [29](#)
- Scade program
  - declaration syntax and semantics [7](#)
- Scopes
  - declaration syntax and semantics [41](#)
- Semantics
  - causality analysis basics [131](#)
  - clock analysis basics [128](#)
  - initialization analysis basics [133](#)
  - namespace analysis basics [123](#)
  - type analysis basics [126](#)
- Sensor blocks
  - declaration and attributes [27](#)
- Sensors
  - declaration syntax and semantics [27](#)
- Sequential operators
  - declaration syntax and semantics [78](#)
- Signal blocks
  - declaration and attributes [41](#)
- Signal emission
  - declaration semantics [46](#)
- Signals
  - see Scopes
- Simple equation
  - declaration and semantics [45](#)
- Size parameters

# Index

---

declaration semantics [30](#)

## State machines

declaration syntax and semantics [53](#)

## State transition

declaration syntax and semantics [58](#)

## Structure expressions

declaration syntax and semantics [95](#)  
labels syntax [97](#)

## Structured types

semantics [20](#)

## Subpackages

and open directive [9](#)  
path and package context [8](#)

## Symbols

syntax description [4](#)

## Syntax

description [3](#)

## T

## Textual language

and graphical representations [145](#)

## times

dynamic semantics [81](#)  
syntax and semantics [79](#)

## Transition actions

declaration syntax and semantics [60](#)

## transpose

syntax and semantics [91](#)

## Type analysis [126](#)

## Type blocks

declaration and attributes [16](#)

## Type checking

principles [126](#)

## Type expressions

syntax and semantics [19](#)

## Type identifier

as type expression [20](#)

## Type variable

semantics for generic operators [20](#)

## Types

declaration syntax and semantics [16](#)  
semantic analysis [126](#)  
visibility attribute semantics [12](#)

## Types and groups

syntax and semantics [15](#)

## Typing environment [126](#)

## U

## Unary operators

declaration syntax and semantics [84](#)

## User operators

visibility attribute semantics [12](#)

## User-defined operators

declaration syntax and semantics [30](#)  
expressing data and control flow [43](#)  
syntax and semantics [29](#)

## V

## Variables

declaration syntax and semantics [36](#)

## Visibility

attribute semantics [12](#)

## W

## when

dynamic semantics [81](#)  
syntax and semantics [79](#)

## X

## xor

dynamic semantics [84](#)  
syntax and semantics [83](#)

