Ansys

Primer

Scade Language Primer

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.

Contact one of our Sales representatives at scade-sales@ansys.com.

Direct general questions about SCADE products to scade-info@ansys.com.

Discover latest news on our products at www.ansys.com/products/embedded-software.

LEGAL INFORMATION

TERMS OF USE

DOCUMENTATION DISCLAIMER

THIRD-PARTY LEGAL INFO

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

Shipping date: 13/04/23 **Revision**: SCS-PM-23 - DOC/rev/34608-04

Scade Language Primer Overview

This manual presents informally the evolutions introduced in Scade 6.0 language and explores step-by-step both the syntax and the dynamic behavior of the language. All chapters illustrate the main underlying concepts and mechanisms of the language with examples. It is intended as the starting point for learning about Scade 6 language.

RELATED DOCUMENTS

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

TYPOGRAPHICAL CONVENTIONS

`Courier New Bold`	Keywords of SCADE language (*e.g.,* **pre**, **node**)
`Courrier New`	Code examples or file extensions (*e.g.,* `.xscade`)
Italics	Names of models, variable elements (*e.g., newdescription*), object attributes (*e.g., B1*), or concepts

Table of Contents

Scade Language Primer Overview

Table of Contents

Table of Contents

Table of Contents

1 / Introduction

This manual provides an informal presentation of most of Scade language features. The aim is to give important insights on the concepts through examples; this is the right place to start learning the language.

- Chapter 2 / introduces the dataflow kernel which basically corresponds to Lustre[1]. This historical kernel is extended with new constructs.
- Chapter 3 / presents the *state machine* extension which is a common way to express control in real-time systems.
- Chapter 4 / presents the constructions introduced when the control can be designed in a simpler way than a state machine.
- Chapter 5 / details the new array data type and primitive operations to use it. The most innovative part here are the various iterator schemes.
- Chapter 6 / describes other language constructions that are not able to affect the behavior of the operators. These special features help solving integration issues or supporting the context of verification tools (provers or simulators).
- Chapter 7 / introduces means to organize Scade 6 programs in packages and shortly presents scopes and namespaces.
- Chapter 8 / introduces the polymorphism of Scade 6.

1. P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. "Lustre: a declarative language for programming synchronous systems" 14th ACM Symposium on Principles of Programming Languages -1987

- Chapter 9 / illustrates the main language features with longer examples.

Notation

In the rest of this manual, the Scade 6 language is referred to as Scade.

---

2 / Scade and the Dataflow Model

This chapter presents the Scade core dataflow.

2.1 Getting Started

This very first section deals with basic elements whose understanding is mandatory before proceeding to any further exploration of this document.

2.1.1 Flows

A flow (or a stream) is an infinite sequence of values of a given type; for example: `1, 2, ..., n, ...` the flow of natural numbers. To represent the value of a flow at a certain rank one writes v_k, which represents in Scade the flow `v` at the instant `k`. In the rest of this document, the dynamic behavior of primitives is described with the following notation:

v	v_1	v_2	v_3	...

The first column contains expressions defining flows (here an identifier) and the second one the sequence of values they represent (at least enough values to be explicit).

2.1.2 Temporal Operations on Flows

Scade provides two fundamental primitives to define flows:

- the previous operator **`pre`**, a one step delay,
- the flow initialization operator "–>", which builds a flow whose first value is the one of its first argument and whose following values are the ones of its second argument.

a	a_1	a_2	a_3	a_4	...
b	b_1	b_2	b_3	b_4	...
pre b	*nil*	b_1	b_2	b_3	...
a –> b	a_1	b_2	b_3	b_4	...
a -> **pre** b	a_1	b_1	b_2	b_3	...

The *nil* value appearing at the very first instant of a delayed flow represents an undefined value. The reason is that one cannot calculate the value preceding the first value of a flow. These undefined holes in a flow can be filled by flow initializations. In general, the combination of delays and

initializations allows to define delayed flows with a proper value at the first instant (see table line a –> **pre** b). This pattern is quite usual and can be iterated several times to specify the first *n* instants of a flow:

```
a -> pre (a -> pre (... pre (a -> pre b) ... ))
```

Scade includes the dataflow operator **fby** (which stands for followed by) to capture this pattern. The pattern becomes: **fby**(b;n;a) where n is a static integer.

a	a_1	a_2	...	a_n	a_{n+1}	a_{n+2}	...
b	b_1	b_2	...	b_n	b_{n+1}	b_{n+2}	...
fby(b; n; a)	a_1	a_1	...	a_1	b_1	b_2	...

2.1.3 Point-Wise Extension of Standard Operators

Operations on values are extended to a point-wise application on flows. For instance the arithmetics operator **+** applied on flows computes the following sequence:

a	a_1	a_2	a_3	a_4	...
b	b_1	b_2	b_3	b_4	...
a + b	$a_1 + b_1$	$a_2 + b_2$	$a_3 + b_3$	$a_4 + b_4$	...

2.1.4 Clocks

The point-wise application and the temporal primitives presented above preserve the *length* of flows in the sense that finite sequences of size *n* combine to finite sequences of the same size. Scade also provides operators that may alter the length of a flow (sampling operator) and conversely that extend sampled flows.

- "Sampling a flow"
- "Combining sampled flows"
- "The clock discipline"
- "Merging complementary flows"

SAMPLING A FLOW

The **when** operator allows to sample a flow from a faster one by extracting sub-streams according to a condition:

c	true	false	true	false	false	true	...
a	a_1	a_2	a_3	a_4	a_5	a_6	...
a **when** c	a_1		a_3			a_6	...

Clock c sets the instants when a **when** c is defined. The point-wise application applies to flows on the same clock: in the above sequences, a + (a **when** c) is incorrect because at cycle 2, a is defined while a **when** c is not. This forbids point-wise application. In Scade, such an operation is statically defined as incorrect, the *clock calculus* defines what is a correct model regarding the clock discipline. The *base clock* is the clock that is always true; a stream on the base clock is present at each cycle.

COMBINING SAMPLED FLOWS

Point-wise application holds for flows on the same clock. The **pre** operator behaves according to clocks.

c	true	false	true	false	false	true	...
a	a_1	a_2	a_3	a_4	a_5	a_6	...
	b_1	b_2	b_3	b_4	b_5	b_6	...
a **when** c	a_1		a_3			a_6	...
b **when** c	b_1		b_3			b_6	...
(a **when** c) + (b **when** c)	$a_1 + b_1$		$a_3 + b_3$			$a_6 + b_6$	...
pre (b **when** c)	*nil*		b_1			b_3	...

THE CLOCK DISCIPLINE

The clock notion comes with a strong discipline which forces any combination of flows to syntactically share the same clock. This discipline ensures the fact that no flow composition can implicitly ignore some values and prevents any value buffering. Filtering (ignoring values) a flow in Scade is an explicit operation that involves the **when** operator. This constraint is essential to run Scade programs with bounded memory resources.

MERGING COMPLEMENTARY FLOWS

Scade proposes a primitive to go the other way around: given several flows based on complementary clocks (such as the Boolean c and **not** c), this operator combines them to produce a faster flow. This way Scade can offer another way to build fast flows from slower ones. Instead of extending a sampled flow with its latest value, one can merge two flows with complementary clocks (c and **not** c). This way, at each cycle of the clock c, one and only one of the slower flows has a value. By taking the available values of each flow, one can define a unique flow based on a faster clock.

a	a_1	a_2	a_3	a_4	a_5	a_6	...
b	b_1	b_2	b_3	b_4	b_5	b_6	...
c	true	false	true	false	false	true	...
a **when** c	a_1		a_3			a_6	...
b **when not** c		b_2		b_4	b_5		...
merge (c; a **when** c; b **when not** c)	a_1	b_2	a_3	b_4	b_5	a_6	...

Notice that this operator only accepts clock identifiers as its first argument.

2.1.5 Nodes and Functions

A Scade program describes the dataflow relations between the outputs and inputs of a system. These relations are expressed using operators, auxiliary variables, and constants. Operators can either be temporal primitives and pointwise extended primitives, or user-defined operators called *nodes* or *functions*. A node is the basic structuring unit of Scade programs. A node is defined by its name, its interface (list of inputs and list of outputs, along with their types), together with a definition of each of these output flows. This implements the dataflow flavor of Scade programs since these definitions are the first-class citizens.

An output flow is either defined by means of an equation or by a state machine (see Chapter 3 about "State Machines"). Local flows could be defined to simplify the definition of output flows. According to the synchronous dataflow model, each operator in the program responds to its inputs within the global time duration allowed, and for every arrival of these inputs. For instance, a node computing the flow of natural numbers can be defined as:

```
node nat() returns (n:uint64)
let
   n = 1 -> (1 + pre n);
tel
```

This node computes the following sequence:

`1`	1	1	1	1	…
`pre n`	*nil*	1	2	3	…
`1 + pre n`	*nil*	2	3	4	…
`1 -> (1 + pre n)`	1	2	3	4	…

This node is called a counter as its output increases at each cycle (supposing that it never overflows). If none of the definitions inside a node refers to past values (i.e., neither temporal operators nor state machines), the operator is

combinatorial (by opposition to sequential) and this property can be made explicit by declaring it as a "function" instead of a "node". The operator definition has to be consistent with this declaration.

2.1.6 Operator Instantiation

The user-defined operators can be used as any other primitive operator.

- “Simple Operator instantiation”
- “Resettable Node Instantiation”
- “Hierarchy of instances”

SIMPLE OPERATOR INSTANTIATION

Let N be the identifier of a user-defined operator (a node or a function) and e any expression defining a flow or a list of flows. Then N(e) is an operator instantiation.

The following example shows two user-defined operators: N1 function (see absence of memorization in N1 equations) called by N2 node (see **pre** primitive use):

```
function N1 (x,y: float64) returns (sum , prod : float64)
let
  sum = x + y ;
  prod = x * y ;
tel

node N2 (a: float64) returns (b,c: float64)
let
  b, c = N1(a, a -> pre a) ; -- instantiation of function N1
tel
```

An operator instantiation can be seen as an expansion. Changing the instantiated operator’s identifier by its definition and the operator formal arguments by the expressions passed as arguments is called an expansion.

Such a manipulation often helps to better understand what an operator instantiation stands for. The following example shows the result of the expansion of the N1 operator within the N2 operator:

```
node N2_expanded (a: float64) returns (b,c: float64)
let
    b, c = ( a + (a -> pre a) , a * (a -> pre a) ) ;
tel
```

RESETTABLE NODE INSTANTIATION

An unbounded counter can be written using a node:

```
node count () returns (c: uint64)
let
    c = 0 -> (1 + pre c);
tel
```

An instance of this node produces the flow 0, 1, 2, ... on the local clock. Adding a reset input to this component, that resets the node instance, restarts the sequence 0, 1, 2, ... on every cycle for which this input is true:

```
node resettable_count (r: bool) returns (c: uint64)
let
    c = if r then 0 else (0 -> (1 + pre c));
tel
```

Each instant where r is true the counter restarts from 0.

Scade offers a way to restart a node instance; the syntax of a reset instantiation is:

```
(restart N every c)(e)
```

With this instantiation scheme, the previous example can be rewritten:

```
node resettable_count (r: bool) returns (c: uint64)
let
    c = (restart count every r)();
tel
```

This primitive affects flow initializations in the instantiated node by making them returning their first argument as if it were the first cycle.

```
const max : uint64 = 10;
node sigma (e:uint64) returns (sum:uint64)
let
    sum = 0 -> e + pre sum;
tel
node sample () returns (s:uint64)
let
    s = (restart sigma every (pre s >= max))(count ());
tel
```

count ( )	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
s	0	1	3	6	10	0	6	13	0	9	19	0	12	0	14	0	16	...
pre s	*nil*	0	1	3	6	10	0	6	13	0	9	19	0	12	0	14	0	...
pre s ≥ max	*nil*	f	f	f	f	t	f	f	t	f	f	t	f	t	f	t	f	...

Note that at the first instant a node always behaves as if a reset occurred, no matter which value is passed as a reset condition. In other words, in their first cycle, node equations always take the first parameter of the operator –>.

Note

This primitive is only useful for a node, since it deals with memorization and initialization. The **`restart`** primitive has no impact on a Scade function.

HIERARCHY OF INSTANCES

The resetting of node instantiations follows a certain hierarchy. When a node instantiation is reset, all the node instances in its definition are automatically reset as well. The following example describes a node sample that calls two

nodes `count` and `double`. The node `double` also calls the node `count`, therefore there are two instances of `count` in that design. Resetting `double` also resets its sub-node `count`.

```
node count () returns (c: int32) c = 0 -> (1 + pre c);
node double () returns (s: int32)
    s = 2 * count();
node sample (b: bool) returns (s1,s2: int32)
let
    s1 = (restart count every b)();
    s2 = (restart double every b)();
tel
```

`sample` produces the following outputs:

b	false	false	false	true	false	false	...
s1	0	1	2	0	1	2	...
s2	0	2	4	0	2	4	...

2.2 Clocks and Nodes

This section presents an important mechanism allowing to control the clock discipline of a node instantiation.

2.2.1 Clock of an Instance

In the dataflow paradigm, a node of the dataflow network is executed when all its inputs are available. When no clock is declared in the node interface, the clock discipline requires that all the inputs and outputs share the same clock. This can be illustrated with the discrete integrator:

```
node integr (e:int32) returns (s:int32)
    s = e + (0 -> pre s);

node two_instances (e : int32; clock h : bool) returns (s, t : int32)
let
    s = integr (e);
    t = merge (h; integr (e when h); (0 -> pre t) when not h);
tel
```

Since `t` should be on the same clock as `s`, its definition uses a **`merge`** because the output of `integr (e` **`when`** `h)` is on clock `h`. This node computes the following output sequence:

e	1	2	3	4	5	6	7	8	9	...
h	false	true	false	false	true	true	false	true	false	...
e **when** h		2			5	6		8		...
integr (e **when** h)		2			7	13		21		...
s	1	3	6	10	15	21	28	36	45	...
t	0	2	2	2	7	13	13	21	21	...

When a node has no input, it is possible to clock any of its instances with the notation `( ( )` **`when`** `h)`, representing a flow carrying no value on clock `h`. Let us illustrate this with two instances of the node `count` defined in 2.1.6 "Operator Instantiation":

count ( )	0	1	2	3	4	5	6	7	8	9	...
h	false	true	false	false	true	true	false	true	false	false	...
count (( ) **when** h)		0			1	2		3			

2.2.2 Clocking Node Inputs/Outputs and Local Declaration

A flow variable declaration may specify a clock:

```
clock h : bool;
x : int32 when h;
```

Specifying a clock for x implies that both its definition and usage conform to the declared clock (h). For example, x=1 **when** h is a legal definition for x. In order to declare a clock in a node signature, one has to respect some constraints:

- the clock of an input flow can neither be an output nor a local flow,
- the clock of an output flow cannot be a local flow.

In presence of such declarations, the rule that determines the execution rate extends what is written in 2.2.1 "Clock of an Instance": *The clock of a node instance is the clock of its fastest input*.

2.2.3 Node Activation with Output Memorization

As seen before, using clocks is a way to control the instants when a node instance is evaluated. The drawback of clocks is that they may be heavy to use. In some cases, it is more intuitive to explicit the control of an instance execution. For that purpose Scade includes a new construct (in place of condacts):

```
y1, ..., yp = (activate N every c initial default (i1, ..., ip)) (e1, ..., en)
```

Its parameters are the name of the instantiated node N, the Boolean c that defines the activation (or the instance clock), the inputs of the instance (e1, ..., en) and **initial default** indicates the initial value to use whenever c is false at the first cycle.

The following equation defines the same flows:

```
y1, ..., yp = merge (c;
    N ((e1, ..., en) when c);
    ((i1, ..., ip) -> pre (y1, ..., yp)) when not c);
```

Note that this translation needs to exhibit an equation to introduce the memory loop. The **pre** used above allows the memorization of the outputs to hold until next activation. The example in 2.2.1 "Clock of an Instance" can be equivalently rewritten:

```
node two_instances (e:int32; h:bool) returns (s, t:int32)
let
    s = integr (e);
    t = (activate integr every h initial default 0) (e);
tel
```

This node now computes the following output sequence:

e	1	2	3	4	5	6	7	8	9	...
h	false	true	false	false	true	true	false	true	false	...
s	1	3	6	10	15	21	28	36	45	...
t	0	2	2	2	7	13	13	21	21	...

2.2.4 Node Activation with Default Flows

The pattern introduced in 2.2.3 "Node Activation with Output Memorization" memorizes the outputs to maintain their value when the node is not activated. Scade proposes an alternative way to deal with these cases, where the outputs of the pattern are connected to a default flow to provide a value every time the condition is false, including the initial instant.

```
(activate N every c default (d1, ..., dp)) (e1, ..., en);
```

The above is semantically equivalent to:

```
merge (c; N ((e1, ..., en) when c); (d1, ..., dp) when not c);
```

Let us consider the previous example with this new construct:

```
node two_instances (e: int32; h: bool) returns (s, t: int32)
let
    s = integr (e);
    t = (activate integr every h default 0) (e);
tel
```

This node computes the following output sequence:

e	1	2	3	4	5	6	7	8	9	...
h	false	true	false	false	true	true	false	true	false	...
s	1	3	6	10	15	21	28	36	45	...
t	0	2	0	0	7	13	0	21	0	...

The output `t` is `0` every time `h` is false.

2.2.5 Simple Node Activation

The activations introduced in 2.2.3 "Node Activation with Output Memorization" and 2.2.4 "Node Activation with Default Flows" are rather easy to use in the sense that they do not introduce any constraint between the Boolean that drives the activation and the availability of the instance output. Scade proposes a notation for the case where an availability constraint (a flow clock) is preferred to a possibly useless memory.

As this kind of activation does not produce a value at each cycle but only when the instance is activated, the clock of the outputs is the Boolean that conditions the activation. The syntactic pattern for this kind of instantiation is:

```
(activate N every c) (e1, ..., en);
```

This is equivalent to:

```
N ((e1, ..., en) when c);
```

Let us consider the example above with this new construct:

```
node two_instances (e: int32; clock h: bool) returns (s, t: int32)
let
    s = integr (e);
    t = (activate integr every h) (e);
tel
```

A semantic error is raised because the flow `t` is supposed to be defined at each local cycle while the expression (**`activate`** `integr` **`every`** h) (e) is on clock h. In other words `t` should always have a value but its definition does not provide one when h is false. To have a model that is correct for the clock analysis, the definition of the case where h is false must be provided; to do that one can use **`merge`**:

```
...
    t = merge (h; (activate integr every h) (e); 42 when not h);
...
```

This expression computes the following output sequence:

e	1	2	3	4	5	6	7	8	9	...
h	false	true	false	false	true	true	false	true	false	...
(**activate** integr **every** h)(e)		2			7	13		21		...
s	1	3	6	10	15	21	28	36	45	...
t	42	2	42	42	7	13	42	21	42	...

One can also merge the outputs of two different node instances for a full definition of `t`:

```
node id (e:int32) returns (s:int32)
let
    s = e;
tel

node merging_two_instances (e:int32; clock h:bool) returns (s:int32)
let
    s = merge (h; (activate integr every h) (e); (activate id every not h) (e));
tel
```

This expression computes the following output sequence:

e	1	2	3	4	5	6	7	8	9	...
h	false	true	false	false	true	true	false	true	false	...
(**activate** integr **every** h)(e)		2			7	13		21		...
(**activate** id **every not** h)(e)	1		3	4			7		9	...
s	1	2	3	4	7	13	7	21	9	...

This pattern consists in instantiating two nodes that are activated alternatively and merged into a single flow. This is a pretty standard way to mimic the imperative *if-then-else* construct in dataflow languages.

2.2.6 Combining Reset and Clocks

Resets do not introduce any constraint between the clocks of reset conditions and the clocks of resettable node instances. It is, to some extent, an *asynchronous reset*. As already said, reset applies to all the instance hierarchy, thus to be modular it must apply to a node with a clock that does not match the reset condition clock. The following example illustrates this asynchronism:

```
node nat () returns (s:int32)
let
    s = 1 -> (1 + pre s);
tel
node rst_clk(rst:bool; clock h:bool) returns (s:int32; t:int32 when h)
let
    s = (restart nat every rst) ();
    t = (restart nat every rst) (() when h);
tel
```

This node computes the following output sequence:

rst	false	false	false	false	false	true	false	false	false	false	true	...
h	false	false	true	true	false	false	false	true	true	true	true	...
s	1	2	3	4	5	1	2	3	4	5	1	...
t			1	2				1	2	3	1	...

In this sequence, the first occurrence of a **true** in the `rst` flow appears while the clock of `t` is **false**; the reset has been taken into account as the next value of `t` that follows is 1. The second occurrence of reset is synchronous with the cycle where `t` is defined; one can observe its immediate effect.

2.3 Data Flow Add-Ons

This section extends the kernel with extra primitives to facilitate the writing of Scade programs.

- 2.3.1 "Groups"
- 2.3.2 "External Code"
- 2.3.3 "Global Flows"

Note

Groups are not supported as graphical objects in SCADE Suite user interface.

2.3.1 Groups

Groups are special types introduced to enable identifiers to represent several flows. A *group* is defined by a list of types:

```
group
   G = (int32, int32, bool);
```

A group can also be defined by a list containing both types and groups.

```
group
G1 = (int32, bool);
G2 = (float64, bool);
G3 = (bool, G1, G2); -- <=> bool, int32, bool, float64, bool
G4 = (G1^5, G2); -- <=> int32^5, bool^5, float64, bool
```

A group is defined regardless of the parenthesis that may appear in its definition.

```
group
G5 = (G1, float64, G2, float64); - <=> int32, bool, float64, float64, bool, float64
```

Two groups are equivalent if after flattening and replacing sub-groups by their definitions they represent the same list of types (remember that type equivalence is structural).

A group cannot be used to define a type, for instance the definitions below are incorrect:

```
group
G = ...
type
T1 = G  n; -- not ALLOWED
T2 = a: G, b: int32; -- not ALLOWED
```

An identifier can be declared with type G where G is a group:

```
group
    G = (int32, int32, bool);
function f(a,b: int32; c: bool) returns (x: int32)
node ex(e: G; ...) returns (s: int32; ...)
let
    ...
    s = f(e);
tel
```

SEMANTICAL ASPECTS

A group of flows is semantically different from a flow of structured type:

- to construct a structure all flows must be on the same clock and any projection of this structure is causally related to its content.
- in an implicit grouping, each flow may have its own clock and no dependence is introduced between the flows.
- in a named group, all flows must be on the same clock.

DISCUSSION

The groups are perfectly coherent with what is sometimes called *list* in Lustre. For instance, in Lustre the following program is correct:

```
function f(a,b: int32; c:bool) returns (x,y: int32; z: bool)
function g(a,b: int32; c: bool) returns (x: int32)
node ex(c: bool; a,b: int32; ...) returns (s,t: int32; u: bool...)
var
    ...
let
    ...
    s,t,u = if c then f(a,b,c) else (0, g(f(b,a,c)),c);
    ...
tel
```

g takes three flows, but appears here with a single parameter `f(a, b, c)`.

Groups do not introduce a new complexity in the expression language. This example can be equivalently rewritten to explicit the groups that are implicitly used as shown below:

```
group
    G = (int32, int32, bool);
function f(a,b: int32; c: bool) returns (x,y: int32; z: bool)
function g(a,b: int32; c: bool) returns (x: int32)
node ex(c: bool; a,b: int32; ...) returns (res:G ...)
    var
    x,y,z : G;
    ...
let
    ...
    x = f(a,b,c);
    z = f(b,a,c);
    z = (0, g(y),c);
    res = if c then x else z;
tel
```

However, explicit groups must only contain flows based on the same clock. One can write:

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

But not:

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

2.3.2 External Code

A Scade model can import code written in another language. In order to preserve Scade paradigm and determinism, this code has to respect certain properties.

Important

It is important to note that as far as these requirements are satisfied the semantics of Scade is preserved. Otherwise, additional work that depends on the implementation is necessary to prove that the system is deterministic and respects the synchronous semantics.

There are different cases of imported code: *pure combinatorial* and *sequential*. The integration and the interface of this code is tool dependent.

THE COMBINATORIAL CASE

An imported operator is combinatorial if the values it returns only depend on current inputs (*i.e.,* each time it is called with a given input vector it returns the same output vector). In the case of a C function, if its implementation uses global variables or static locals, the risk is high not to meet this requirement.

On the model side only the interface provided is:

```
function imported f (i1: int32; i2: bool) returns (o1: bool; o2: int32);
```

THE SEQUENTIAL CASE

An imported operator is sequential if it implements a function on flows ie. If, for a given initial state of its memory and for a given input sequence of values, its output sequence is always the same. This property must hold for all the instances of this operator. As a consequence this external code must be:

1. instantiatable
2. resettable

The point 1 requires to have a way to separate the state of the instance from the code itself, thus no globals or static variables are allowed to implement the memory of the sequential machine. The point 2 requires to have a piece of code that, given a memory instance is able to reset it. The syntax to declare an imported node is:

```
node imported N (i1: int32; i2: bool) returns (o1: bool; o2: int32);
```

2.3.3 Global Flows

Some flows of Scade models are defined outside the scope of a node and can be used in the node definitions: those are called global flows. There are two kinds of global flows: *constants* and *sensors*.

CONSTANTS

Constants can be imported or defined by an expression involving other constants and immediate values. A constant identifier represents a flow whose value is the same at each tick. They are based on the local clock of the node they are used in.

```
const
   C: int32 = 42;
   D: int32 = 2 * C;
   imported E: int32;
```

These declarations compute the following sequence:

C	42	42	42	...
D	84	84	84	...
E	v	v	v	...

where v is the value of the imported constant given at integration time.

SENSORS

The other kind of global flows allows to easily connect an application to its environment by allowing any node to access some global input. Its name reflects its main usage: accessing the value provided by a physical sensor (temperature sensor for instance). Like constants, sensors can be used in any node definition where they are visible.

Important

In order to preserve the Scade semantics (particularly the determinism), a sensor flow is on the fastest clock of the application (often called base clock) and its value cannot change during a cycle on this clock. Code generation tools must explain how this construct is handled at integration time.

A flow defined by a sensor is on the fastest clock and used on the local clock of a node instance. The context it is used in can therefore implicitly filter some values.

```
sensor
speed: float64;
temperature: float64;

function Temp_plus_one () returns (s: float64)
let
    s = temperature + 1.0;
tel

node application (c: bool) returns (s, t: float64)
let
    s = speed;
    t = (activate Temp_plus_one every c initial default 0.0) ();
tel
```

This node compute the following sequence:

`temperature`	19.9	19.7	19.7	19.6	19.5	...
`speed`	41.2	41.2	41.5	41.6	42.0	...
`s`	41.2	41.2	41.5	41.6	42.0	...
`c`	false	true	true	false	false	...
`t`	0.0	20.7	20.7	20.7	20.7	...

3 / State Machines

Like equations, state machines exist primitively in Scade language. A program containing state machines can always be translated into a pure dataflow program. However, since these machines provide a way to shed a light on the control flow of a Scade program, they sometimes allow a more compact, readable writing style that can be greatly profitable.

3.1 Introducing State Machines

A state machine definition can take place wherever an equation can stand. A state machine has an optional name, which can be any valid identifier. Furthermore, a state machine provides a definition for one or several flows.

The flows defined by a state machine may be explicited in its **`returns`** declaration, or implicitly by using the '..' notation.

```
let
    -- equations
    x = f(y);
    -- a state machine
    automaton My_State_Machine
    -- this state machine defines X1,X2 and X3 flows
    --
    -- here stands the state definitions composing the state machine
    --
    returns X1 , X2 , X3;
    -- other equations or other state machines (in parallel)
tel
```

A state machine is a collection of states, each state defines its own equation set where (at least) the output flows of the state machine are defined.

```
automaton My_State_Machine
    state First_State
    let
    X1 = expr_1_1;
    X2 = expr_2_1;
    X3 = expr_3_1;
    tel

    state Second_State
    let
    X1 = expr_1_2;
    X2 = expr_2_2;
    X3 = expr_3_2;
    tel
    -- other states defining X1, X2, X3
returns X1 , X2 , X3;
```

Such a state machine can be called a *modal machine*, a state defining a *mode* where any flow defined by the state machine has its particular definition. The rules to determine which mode is applicable are defined by transitions between states.

3.1.1 Example Built Step by Step

Consider a simple example: a node taking a Boolean flow and returning a Boolean flow that is true when the input was true an even number of cycles and false otherwise.

```
node even_times (c: bool) returns (o: bool)
```

Let us add a state machine which computes the flow o :

```
node even_times (c: bool) returns (o: bool)
let
    automaton
    ...
    returns o;
tel
```

This state machine has two states: one that corresponds to the instants where c was true an odd number of cycles and the other when it was true an even number of cycles.

```
node even_times (c: bool) returns (o: bool)
let
  automaton
  state EVEN
  let
  o = true;
  tel

  state ODD
  let
  o = false;
  tel
 returns o;
tel
```

Now, let us give rules to switch between both modes (*i.e.,* to switch between both states). This is expressed as transitions from one state to another when the input *c* is present. The EVEN state definition can be read as follows: *if the flow c is true, then take into account the equations of the state ODD; otherwise equations of state EVEN are relevant*.

```
node even_times (c: bool) returns (o: bool)
let
  automaton
  state EVEN
  unless if c resume ODD;
  let
  o = true;
  tel
  state ODD
  unless if c resume EVEN;
  let
  o = false;
  tel
  returns o;
tel
```

Finally, one has to be able to specify in which mode a State machine is starting. This is done by flagging the state EVEN with the keyword **`initial`**, according to the convention that 0 is considered as being even.

```
node even_times (c: bool) returns (o: bool)
let
  automaton
  initial state EVEN
  unless if c resume ODD;
  let
  o = true;
  tel
  state ODD
  unless if c resume EVEN;
  let
  o = false;
  tel
  returns o;
tel
```

This last instance of the example returns the following values:

c	true	false	true	true	true	false	false	...
o	false	false	true	false	true	true	true	...

3.1.2 Same Example in Pure DataFlow Style

The example from 3.1.1 can be rewritten using a pure dataflow style. Though trivial, the rewriting pattern presented below is general enough to be used in most cases.

As its name implies, a state machine has a “state”. At any instant, it can access a memory whose content depends on the past instants. In our case, this memory registers the fact that an even or odd number of c has passed. This memory can be implemented by a local Boolean flow `mystate` which is true when the number of cycles input c hwas true since the beginning of the execution is even.

Every time the Boolean c is true, the state of the machine changes; otherwise it remains the same. This is encoded by the expression:

```
(if c then not (pre mystate) else pre mystate)
```

Finally, at the first instant the state is EVEN if c is false, otherwise it is ODD. So the memory is defined by the expression:

```
(not c -> if c then not pre mystate else pre mystate)
```

As in the state machine model, the output is inferred from the state. The program becomes:

```
node even_times (c: bool) returns (o: bool)
var
    mystate: bool;
let
    mystate = not c -> if c then not pre mystate else pre mystate;
    o = if mystate then i + 1 else -2 * i;
tel
```

In this example, the pure dataflow program is simpler and lighter than its state machine version which is not the case in general. Note that Scade offers both styles, which allows the designer to choose the most adequate one.

3.2 State Transitions

Scade offers two different kinds of state transitions: *strong* and *weak*. They both correspond to common ways to fire a transition in control models. In Scade, a state machine has one and only one active state per cycle. This property preserves the unicity of the definition of a flow during a cycle. After presenting these definition, the scheduling of state machines is briefly sketched.

Terminology

The description of scheduling requires the introduction of two terms:

- selected state: the state whose outgoing strong transitions are inspected.
- active state: the state whose content holds for the cycle.

3.2.1 Strong Preemption

Strong preemptions in a state specify where the computation in the current cycle takes place.

PRINCIPLES

Syntactically, strong preemptions appear before the specification of the body of their source state and are introduced by the keyword **`unless`**. This notation expresses the fact that *unless* one of the guards of a strong preemption is true, this state is activated. Let us precise these notions of selected/activated state on the previous example, which uses strong preemptions:

`c`	true	false	true	true	true	false	false	...
`selected`	EVEN	ODD	ODD	EVEN	ODD	EVEN	EVEN	...
`activated`	ODD	ODD	EVEN	ODD	EVEN	EVEN	EVEN	...

State machines specified this way are close to Mealy machines in the sense that the set of equations defining the flows in the current cycle can depend on the current operator inputs.

RESETTING/RESUMING A STATE

In previous example, the strong preemption specifies the target state using the keyword **`resume`**, meaning that the target state recovers its internal state (if it has one). Another way to specify preemption is by using the keyword **`restart`**. In this case, all content of the target state is reset: dataflow expressions (see "Resettable Node Instantiation" on page 10 for dataflow definition of reset) and state machines contained in the target state.

The previous example does not contain anything (expression or sub-state machine) to show the difference between **resume** and **restart**. Let us modify it to illustrate this difference:

```
node example_1 (i: int32; c: bool) returns (o: int32)
let
  automaton
     initial state EVEN
     unless if c resume ODD;
     let
       o = 100 -> i + 1 ;
     tel
     state ODD
     unless if c restart EVEN;
     let
       o = 203 -> -2 * i;
     tel
  returns o;
tel
```

This node now computes the following output sequence:

c	true	false	true	true	true	false	false	...
i	1	2	1	-1	3	0	-1	...
o	203	-4	100	2	100	1	0	...

The dataflow operator "**->**" is equivalent to a simple two-state state machine where the first argument is the value of the flow at the first cycle (the very first one of its clock or the first that follows a reset).

Considering the first cycle in state *EVEN*, the output is 100 and for the first cycle in state *ODD* the output is 203. After the first execution of each states, one can see that entering *EVEN* with **restart** returns 100, while entering *ODD* with **resume** does not use the left-hand side of "**->**" anymore.

This last example can be written using state machine hierarchy by replacing the -> operator by the equivalent two-state state machine. The behavior is the same:

```
node example_2 (i: int32; c: bool) returns (o: int32)
let
  automaton
    initial state EVEN
    unless if c resume ODD;
    let
      automaton
        initial state First_cycle
        let
          o = 100;
        tel
        until if true resume Other_cycles;

        state Other_cycles
        let
          o = i + 1;
        tel
      returns o;
    tel

    state ODD
    unless if c restart EVEN;
    let
      automaton
      initial state First_cycle
      let
        o = 203;
      tel
      until if true resume Other_cycles;
      state Other_cycles
      let
        o = -2 * i;
      tel
      returns o;
  tel
  returns o;
tel
```

CAUSALITY ISSUES

Since strong preemptions may change the current active state, it is not possible to use guards that instantaneously depend on flows defined in the selected state.

```
node example_3 (i: int32; c: bool) returns (o: int32)
let
  automaton
    initial state EVEN
    unless if (o mod 2 = 0) resume ODD;
    let
      o = i + 1;
    tel
    state ODD
    unless if c restart EVEN;
    let
      o = -2 * i;
    tel
  returns o;
tel
```

This causality error can be informally expressed as: o is defined by state EVEN unless it is even, in which case it is defined by state ODD. Thus the definition of o depends on its current value, which is a non-causal situation.

STRONG TRANSITION PRIORITIES

In the previous example, a state only had one outgoing transition. Of course in general it is possible to specify more transitions with different targets and guards. In the cycles where two or more of these transitions have a valid guard, the transition with higher priority is fired. This priority is a total order and is given by the order in the source text:

```
node example_4 (a, b: bool) returns (o: int32)
let
  automaton
    initial state A
    unless
      if a resume B; -- highest priority
      if b resume C;
    let
      o = 1;
    tel
    state B
    unless if true resume A;
    let
      o = 2;
    tel
    state C
      unless if true resume A;
    let
      o = 3;
    tel
  returns o;
tel
```

When the guards a and b are both true in the same cycle, this node computes the following output sequence:

a	false	true	false	false	false	false	true	false	false	false	...
b	false	false	false	true	false	false	true	false	false	false	...
o	1	2	1	3	1	1	2	1	1	1	...

3.2.2 Weak Preemption

Contrary to strong preemptions, the active state can be left only at the end of the cycle if the condition of an outgoing weak preemption (or transition) is true in the current cycle.

PRINCIPLES

The weak preemption is also called *weak delayed* because the target state of the transition is activated in the next cycle only. Let us change the transitions on the `even_times` example by weak preemptions:

```
node example_1 (i: int32; c: bool) returns (o: int32)
let
  automaton
     initial state EVEN
     let
       o = 100 -> i + 1 ;
     tel
     until if c resume ODD;
     state ODD
     let
       o = 203 -> -2 * i;
     tel
     until if c restart EVEN;
  returns o;
tel
```

Weak transitions are specified after the body of the state to emphasize the idea that they specify the state for the next cycle. The EVEN state definition now can be read as follows: *Take into account equations of state EVEN; if c is true switch to state ODD for the next cycle; otherwise stay in the same state*.

c	true	false	true	true	true	false	false	...
i	1	2	1	-1	3	0	-1	...
o	2	-4	-2	0	-6	1	0	...
selected	EVEN	ODD	ODD	EVEN	ODD	EVEN	EVEN	...
activated	EVEN	ODD	ODD	EVEN	ODD	EVEN	EVEN	...

In this new version, output o computes an even number of occurrences of input flow c, but delayed by one instant. Its corresponding dataflow translation is:

```
node example_2 (i: int32; c: bool) returns (o: int32)
let
  automaton
    initial state EVEN
    let
      automaton
        initial state First_cycle
        let
          o = 100;
        tel
        until if true resume Other_cycles;

        state Other_cycles
        let
          o = i + 1;
        tel
      returns o;
    tel
    until if c resume ODD;

    state ODD
    let
      automaton
      initial state First_cycle
      let
        o = 203;
      tel
      until if true resume Other_cycles;
      state Other_cycles
      let
        o = -2 * i;
      tel
      returns o;;
    tel
    until if c restart EVEN;
  returns o;
tel
```

The differences with the translation above are the forced initialization of `evenState` whatever value c takes, and the fact that state transitions are taken on the previous value of c. Causality issues

Because the weak preemption specifies the next state, it is possible to use guards that depend on flows defined in the source state of the transition. For instance the following example is correct and does not suffer of any causality problem.

```
node example_3 (i: int32; c: bool) returns (o: int32)
let
  automaton
    initial state EVEN
    let
      o = i + 1;
    tel
    until if (o mod 2 = 0) resume ODD;

    state ODD
    let
      o = -2 * i;
    tel
    until if c restart EVEN;
  returns o;
tel
```

This node computes the following output sequence:

i	1	2	3	4	5	6	7	8	9	10	...
c	false	false	true	false	true	false	true	true	true	false	...
o	2	-4	-6	5	6	-12	-14	9	10	-20	...
o **mod** 2 = 0	true	true	true	false	true	true	true	false	true	true	...

WEAK TRANSITION PRIORITIES

As for strong transitions, a priority is sometimes needed to decide which weak transition is fired; this is also a total order given by the order in the source text:

```
node example_4 (a, b: bool) returns (o: int32)
let
  automaton
    initial state A
    let
      o = 1;
    tel
    until
      if a resume B; -- highest priority
      if b resume C;

    state B
    let
      o = 2;
    tel
    until if true resume A;

    state C
    let
      o = 3;
    tel
    until if true resume A;
  returns o;
tel
```

When the guards a and b are both true in the same cycle, this node computes the following output sequence:

a	false	true	false	false	false	false	true	false	false	false	...
b	false	false	false	true	false	false	true	false	false	false	...
o	1	1	2	1	3	1	1	2	1	1	...

3.2.3 Mixing Weak and Strong Preemption in Single State Machine

A state machine can have more than one kind of transition, weak and strong can be used together in the same state machine and from the same source state. To allow such a mix, one must add some rules to define what occurs in presence of both kinds:

- *the outgoing strong transitions of a state have priority on the outgoing weak transitions of this state*.
- *only one transition (weak or strong) can be fired per cycle*.

Note that a state entered through a weak transition can be exited through a strong transition without being activated. This scenario is clear when looking precisely at the scheduling (see 3.2.5 "Specification of the Evaluation Instants" for details). This particular configuration (a weak followed by a strong) is highlighted here because *this is the only case in Scade where a state is passed through without being activated*.

This would be the case for the following three-state state machine:

```
automaton
    initial state A
    ...
    until if true resume B

    state B
    unless if true resume C
    ...

    state C
    ...
returns
```

The evaluation of selected and activated states is as follows:

selected	A	B	C	...
activated	A	C	...	...

3.2.4 Scope/Namespace of a State

A state defines its own namespace; it is therefore possible to declare variables which are local to a given state. These declarations have the same syntax as in the one adopted for nodes. They take place after the unless transition block but before the equation block. As a consequence, local state variables cannot occur in the unless transitions but can be defined and used in the equations of the state and in the until transitions.

The following example illustrates the usage of local variables:

```
node even_times_notify (c: bool) returns (o: bool; number: int32)
let
  automaton
    initial state EVEN
    unless if c resume ODD;
    var nb_even: int32;
    let
      o = true;
      nb_even = 1 -> pre nb_even + 1;
      number = nb_even;
    tel
    state ODD
      unless if c resume EVEN;
    var nb_odd : int32;
    let
      o = false;
      nb_odd = 1 -> pre nb_odd + 1;
      number = nb_odd;
    tel
  returns o, number;
tel
```

This node computes the following output sequence:

i	false	false	false	false	true	false	true	true	true	false	...
o	true	true	true	true			true		true	true	...
number	1	2	3	4	1	2	5	3	6	7	...

3.2.5 Specification of the Evaluation Instants

For a state machine, each synchronous cycle consists in deciding which set of equations holds and then evaluating it, and in deciding if there is a transition to fire. This process must satisfy the following principles:

- *at most one transition fired per cycle.*
- *exactly one active state per cycle.*

PRINCIPLES OF STATE MACHINE EXECUTION

The execution of a state machine can be decomposed into the following steps:

1. Determining the *selected state*.
2. If this state has outgoing strong transitions, evaluating all the guard of the strong transitions and inspect them (possibly fire a strong transition).
3. Determining the *active state*.
4. Computing actions in the *active state*.
5. If no strong transition has been fired at step 2, evaluating all the guards of the weak transitions and inspect them (possibly fire a weak transition).

Step 4 computes the selected state that it used at step 1. For the first cycle, this selected state is initialized with the initial state of the state machine. Resetting a state machine consists in forcing its *selected state* to be its specified initial state.

THE CLOCK POINT OF VIEW

The semantics of scheduling can also be presented informally from a clock point of view. This is a way to define precisely the state machine extension by translating it into the dataflow kernel of Scade. The inspection of strong transitions and the computation of flows may occur in different states. As a

result, each cycle is decomposed into two clocks: *selected state clock* (Sclk) and *active state clock* (Aclk). Each syntactic location in a state machine is related to a clock specifying the conditions of its evaluation:

```
node N(...) returns (...)
let
    ...
    -- local clock of the node
    ...
    automata A
    ...
    state S1
    unless
    ...
    -- selected state clock of S1: Sclk(S1)
    ...
    var
    ...
    let
    ...
    -- local clock of state S1: Aclk(S1)
    ...
    tel
    until
    ...
    -- Aclk(S1) too
    ...
    returns ..;
    ...
tel
```

The hierarchy of states is directly mapped to the hierarchy of clocks. Thus a state machine is activated if the clock of its context is true (the local clock of the node for the state machine A; clock Aclk(S1) for a state machine defined in state S1).

3.3 Sharing Memories Between States

This section deals with shared memories and states from the following aspects:

3.3.1 Need for Shared Memories

In the previous examples the states of the state machines never communicate values, but in general there is some kind of continuity between each set of equations. A simple example is the specification of a counter that increases until a maximum value and then decreases until a minimum one and repeats this process. This can be specified with a two-state state machine

which increments or decrements the last value, depending on its state, to compute the next one. A shared flow computes the last value of the counter which is then shared between both states:

```
node UpDown () returns (x: int8)
var
   last_x: int8;
let
   last_x = 0 -> pre x;   -- the flow last_x contains the value of x at the previous
                          -- cycle, independently of the state.
   automaton UD
     initial state Up
     let
     x = 0 -> last_x + 1;
     tel
   until if x >= 5 resume Down;

   state Down
    let
    x = last_x - 1;
    tel
   until if x <= -5 resume Up;
   returns x ;
tel
```

The above node computes the following output sequence:

x	0	1	2	3	4	5	4	3	2	1	0	-1	-2	-3	-4	-5	-4	-3	-2	-1	...

It is important to note that the identifier `last_x` cannot be replaced by its definition inside the state, because **`pre`** x in state Up gives the previous value of x the last time state Up was active. To understand this difference, let us have a look at this program and at its execution:

```
node UpDown_wrong () returns (x: int8)
let
   automaton UD
     initial state A
     let
     x = 0 -> pre x + 1;
     tel
   until if x >= 5 resume B;

     state B
     let
     x = 0 -> pre x - 1;
     tel
  until if x <= -5 resume A;
  returns x ;
tel
```

The well-defined version of this node requires an initialization of the flow x in state B. When not provided, the program is undeterministic and rejected by the static analysis. This leads to the following execution:

x	0	1	2	3	4	5	0	-1	-2	-3	-4	-5	6	-6	7	-7	8	-8	9	-9	…

3.3.2 Language Primitive to Share Memories

Access to the latest value of a flow in its scope is often required when mixing dataflow style with state machines. The *mode-automata* point of view relies on the ability to access values computed in other states. Scade introduces a way to access this value: the primitive **`last`** `'x`. This primitive does not apply to expressions but to named flows (that is the reason of the `'x` notation which represents the name `x` and not the expression `x`).

```
node UpDown_2 () returns (x: int8)
let
  automaton UD
    initial state A
    let
    x = 0 -> last 'x + 1;
    tel
  until if x >= 5 resume B;

    state B
    let
    x = last 'x - 1;
    tel
  until if x <= -5 resume A;
  returns x ;
tel
```

This new operator behaves like UpDown:

x	0	1	2	3	4	5	4	3	2	1	0	-1	-2	-3	-4	-5	-4	-3	-2	-1	...

3.3.3 Initializing Shared Memory

Like the **`pre`** of a flow, access to the **`last`** of a named flow is undefined in the first cycle. In our example, the first cycle of the node starts in state A where `x` is defined thanks to the dataflow initialization "**`->`**". The `state` B does not suffer from any initialization problem because it cannot be activated in the first cycle. If B is active, `state` A has already provided a first value to `x` and consequently **`last`** `'x` is defined.

In some cases, this initialization may be hard to provide in the states and would be easier to provide at the declaration level. As the problem of definition appears when trying to access the **last** of a named flow, Scade proposes the notation `x:int8 last=0` that provides a value for **last** `'x` if it is needed at the first instant. For instance, a variant of the node UpDown:

```
node UpDown3 () returns (x: int8 last = 0)
let
  automaton UD
    initial state A
    let
    x = last 'x + 1;
    tel
  until if x >= 5 resume B;

    state B
    let
    x = last 'x - 1;
    tel
  until if x <= -5 resume A;
  returns x ;
tel
```

Back to an equivalent program without **last**:

```
node UpDown4 () returns (x: int8)
var
  last_x : int8;
let
  last_x = 0 -> pre x;
  automaton UD
    initial state Up
    let
    x = last_x + 1;
    tel
  until if x >= 5 resume Down;

    state Down
    let
    x = last_x - 1;
    tel
  until if x <= -5 resume Up;
  returns x ;
tel
```

UpDown3 and UpDown4 behave like UpDown.

3.3.4 Default Actions in States

The leading idea of this extension of Scade with state machines is that states contain sets of equations and the active set is determined by the active state of the state machine. The other important point is that a declared flow (local or output) must have exactly one definition for each cycle where its scope is active. When a definition is missing in a given state, several semantics are possible:

1. considering the program as incorrect,
2. deciding to have a default behavior:
   - give a default value
   - maintain the latest value

The choice in Scade is to maintain the latest value. For instance, consider the following node that specifies a two buttons adjusting of an integer value:

```
node adjust (adj, mode: bool) returns (o: int16 last = 0)
let
  automaton A

    initial state Idle
      unless if mode resume Incr;
      let
        o = last 'o;
      tel

    state Incr
      unless if mode resume Decr;
      let
        o = if adj then last 'o + 1 else last 'o;
      tel

    state Decr
      unless if mode resume Idle;
      let
        o = if adj then last 'o - 1 else last 'o;
      tel

  returns o;
tel
```

In the state `Idle`, the equation `0 = last 'o` expresses the fact that the output `S` must be maintained. The example could be rewritten without explicitly maintaining the value, without modifying the way it behaves:

```
node adjust (adj, mode: bool) returns (o: int16 last = 0)
let
  automaton A

    initial state Idle
      unless if mode resume Incr;

    state Incr
      unless if mode resume Decr;
      let
        o = if adj then last 'o + 1 else last 'o;
      tel

    state Decr
      unless if mode resume Idle;
      let
        o = if adj then last 'o - 1 else last 'o;
      tel

  returns o;
tel
```

It is important to note that omitting a definition leads to the apparition of a memory for the **`last`** of the default action. Thus, to have clear and efficient designs, much attention must be paid when leaving a flow undefined. The intent must be to maintain values and not to leave them undefined. Leaving undefined values means that the flow definition is not relevant in a given state and may reveal poor design (the hierarchy of states/scopes is not adapted, the flow is not declared at the right level).

3.3.5 Modifying Default Action

The default action for a flow definition can be specified at declaration level. This default behavior can involve any visible flow in the scope where the declaration occurs.

The syntax of a declaration with specification of a default behavior is:

```
x: int32 default = 42;
y: int32 default = 2 * x + pre y;
```

The default expression is any Scade expression. For instance if a state machine has a state `idle` and other states and one wants to count each time the state `idle` is not active, one can do it this way:

```
node count_not_idle (...) returns (...)
var
    count: int32 default = 1 + last 'count last = 0;
    let
    automaton
    initial state Idle
    unless ...
    let
    count = last 'count;
    ...
    tel

    state A
    unless ...
    let
    ...
    tel

    state B
    unless ...
    let
    ...
    tel

    state C
    unless ...
    let
    ...
    tel
    returns ..;
tel
```

The states A, B and C did not specify any definition for the flow `count`. In this case, the default definition of `count` is no more `count = last 'count`, thus in the states where this is exactly the desired definition, it must be written explicitly as in state idle.

The semantics of this kind of declaration is given by this equivalent:

```
node count_not_idle (...) returns (...)
var
    count: int32 last = 0;
    count_default: int32;
let
    count_default = 1 + last 'count;
    automaton
    initial state Idle
    unless ...
    let
    count = last 'count;
    ...
    tel

    state A
    unless ...
    let
    count = count_default;
    ...
    tel

    state B
    unless ...
    let
    count = count_default;
    ...
    tel

    state C
    unless ...
    let
    count = count_default;
    ...
    tel
    returns ..;
tel
```

The clock of the default definition is the one of the scope that declared the flow.

In the example of the cruise control (see 9.3 "Cruise Control"), the output `ThrottleCmd` is always defined by the input `Accel` except in the nested state `RegulOn`. One can use the default definition to avoid this kind of repetition; the interface of this node becomes:

```
node CruiseControl( On : bool ; Off : bool ; Resume : bool ;
    Speed : Speed ; Set : bool ; QuickAccel : bool ;
    QuickDecel : bool ; Accel : Percent ; Brake : Percent)
returns (CruiseSpeed : Speed last = 0.0; ThrottleCmd : Percent default = Accel)
...
```

3.4 Pure Signals

Scade proposes pure signals as a primitive notion; it is called "pure" because it cannot carry a value, only its presence can be tested. In this case, signals in Scade can thus be seen as Boolean flows that are false, except at the instants they are emitted. The main difference with usual flows is that a signal can be emitted several times simultaneously while a flow is defined by a single equation at each cycle.

3.4.1 Basic Principles

A pure signal is identified by a name. In its scope, a signal can be emitted and its presence in the synchronous cycle can be tested. If it is not emitted, its presence is false; if it is emitted, the presence is true. The name of a signal is an identifier prefixed by a quote: `'S`.

3.4.2 Declaration, Scope and Presence

A signal is always local to a node or a state and can neither be an input nor an output of this node. Only signal statuses (Boolean flows) can be communicated through inputs and outputs. The status (or presence) of a signal is directly accessible from its name (`'S`) and gives a Boolean flow that can be used in an expression.

3.4.3 Emitting Signal in a Set of Equations

Signal emissions can occur in states or at the top level of a node. They are specified as:

```
emit 'S
emit 'S if c -- signal emission can be guarded by a Boolean flow
```

The node `even_times` presented in 3.1.1 "Example Built Step by Step" can take advantage of signals:

```
node even_times_sig(i: bool) returns (o: bool)
sig
  sig_o;
let
  o = 'sig_o;
  automaton
    initial state EVEN
      unless if i resume ODD;
    let
      emit 'sig_o;
    tel
    state ODD
      unless if i resume EVEN;
  returns 0 ;
tel
```

As signals cannot go through the interface of a node, the Boolean output must be defined to be equal to the status of the signal: `o = 'sig_o`.

3.4.4 Emitting Signal on Transition

A transition can specify signal emissions that are done when it is fired. Syntactically it is specified after the expression of the guard between braces, the keyword is optional because the context delimited by the braces avoids ambiguities:

```
if c do {emit 'S if e}
or without the keyword
if c do {'S if e})
```

For instance a frequency divider can be written with a signal emission:

```
node fdiv (c: bool) returns (half_c: bool)
sig
  s_half_c;
let
  half_c = 's_half_c;
  automaton
    initial state S1
      unless if c resume S2;
    state S2
      unless if c do {'s_half_c} resume S1;
  returns .. ;
tel
```

The frequency divider produces true every two occurrences of true in its input. The state machine is the same as in the previous example (node `even_times_sig`) except that the signal emission is in the transition.

c	false	true	false	true	false	true	false	false	true	false	false	false	true	...
half_c	false	false	false	true	false	false	false	false	true	false	false	false	false	...

3.4.5 Capturing the Latest Status of a Signal

Section 3.3 "Sharing Memories Between States" introduced the primitive **`last`** that gives access to the latest value of a flow in its scope. This primitive also applies to signals. It returns the status of the signal at the previous cycle and is undefined at the first cycle. The following example is a two-state state machine which strongly leaves a state when the status of the signal that this state emits was true at the latest cycle where the signal scope was active. Because the signals are emitted in the states, guarding the strong transitions with the current status would lead to a causality cycle.

```
node example_9() returns (o1, o2: bool)
sig
  s1, s2;
let
  o1 = 's1;
  o2 = 's2;
  automaton
    initial state S1
    unless if (false -> last 's1) resume S2;
  let
  emit 's1;
  tel
    state S2
    unless if (false -> last 's2) resume S1;
  let
  emit 's2;
  tel
  returns ..;
tel
```

The behavior of this state machine consists in alternating its active state at each cycle:

o1	true	false	true	false	true	false	...
o2	false	true	false	true	false	true	

3.5 State Machine Composition

State machines can be composed together hierarchically, by creating a state machine within the state of another state machine, or in parallel. Two means are possible:

- Conditioning a transition on a flow (or a signal) defined in another state machine: see 3.5.1 "Parallel Composition of State Machines".
- Conditioning a transition on the selected state of other state machines: see 3.5.2 "Synchronization Transition".

3.5.1 Parallel Composition of State Machines

State machines can be composed in parallel in a simple way: just putting them to the equation set as in 3.1 "Introducing State Machines". The simple example below illustrates this kind of state machine composition. The operator in this example allows to adjust an integer value using two buttons `adj` and `mode` as in the example developed in section 3.3.4 "Default Actions in States", but now the increment value depends on the number of consecutive cycles the button `adj` was pressed: 1, then 10 after 3

consecutive cycles and then 100 after 3 other consecutive cycles. Its behavior explains why it is named `accelerating_adjust`. The increment value is managed in parallel with the adjustment of the output value:

```
node accelerating_adjust (adj, mode: bool) returns (o: int16 last = 0)
var incr: int16;
let
  automaton ManageMode
    initial state Idle
      unless if mode resume Incr;
    state Incr
      unless if mode resume Decr;
      o = if adj then last 'o + incr else last 'o;
    state Decr
      unless if mode resume Idle;
      o = if adj then last 'o - incr else last 'o;
  returns o;
  automaton ManageIncrement
    initial state One
      incr = 1;
      until
        if 3 times adj restart Ten;
        if not adj restart One;
    state Ten
      incr = 10;
      until
        if 3 times adj restart Hundred;
        if not adj restart One;
    state Hundred
      incr = 100;
      until if not adj restart One;
  returns incr;
tel
```

Here is an example of sequence produced by this operator:

adj	false	true	true	true	true	true	true	true	false	false	true	...
mode	true	false	false	false	false	false	false	false	false	false	false	...
0	0	1	2	3	13	23	33	133	133	133	134	...

The above example can be improved to have an increment such that the value is a multiple of 10 or 100 depending on the state of the `ManageIncrement` state machine. This can be done using the last of the adjusting value to define a first increment to the ten or hundred value that follows the current value before incrementing by 10 to 100.

Here both state machines are mutually recursive since `ManageMode` uses `inc` to define `o` and `ManageIncrement` uses `o` to define `incr`, though there is no causality issue because `o` is used through its last value. This leads to the following definition:

```
node accelerating_adjust (adj, mode: bool) returns (o: int16 last = 0)
var incr : int16;
let
  automaton ManageMode
    initial state Idle
      unless if mode resume Incr;
    state Incr
      unless if mode resume Decr;
      o = if adj then last 'o + incr else last 'o;
    state Decr
      unless if mode resume Idle;
      o = if adj then last 'o - incr else last 'o;
  returns o;
  automaton ManageIncrement
    initial state One
      incr = 1;
      until
        if 3 times adj restart Ten;
        if not adj restart One;
    state Ten
      incr = (10 - last 'o mod 10) -> 10;
      until
        if 3 times adj restart Hundred;
        if not adj restart One;
    state Hundred
      incr = (100 - last 'o mod 100) -> 100;
      until if not adj restart One;
  returns incr;
tel
```

Given the same sequence of commands, the behavior is now:

adj	false	true	true	true	true	true	true	true	false	false	true	...
mode	true	false	false	false	false	false	false	false	false	false	false	...
0	0	1	2	3	10	20	30	100	100	100	101	...

3.5.2 Synchronization Transition

A state can be flagged as final by using the keyword **`final`** before the keyword **`state`**. Any state can be final (even the initial one) and a state machine is not limited to only one final state. Given a state *S* containing several sub-state machines running in parallel, if at the next cycle all these sub-state machines are in a state marked as final, then a synchronization transition can be fired from state *S*.

```
node ABRO(A: bool; B: bool; R: bool) returns (O: bool)
let
  automaton
  initial state Top
  unless if R restart Top;
  sig
  O_Sig;
  let
  O = 'O_Sig;

  automaton
  initial state Wait_A_B
  let
  automaton Auto_A
  initial state Await_A
  until if A restart A_received;
  final state A_received
  returns ..;

  automaton Auto_B
  initial state Await_B
  until if B restart B_received;
  final state B_received
  returns ..;
  tel
  until
  synchro do {emit 'O_Sig} restart End;

  state End
  returns ..;
  tel
  returns ..;
tel
```

3.6 Complex Transitions

In previous examples, the conditions on transitions are fairly simple since they are limited to combinatorial (or memoryless) operations. However, any Scade expression is allowed to trigger a transition. This section presents several constructs of the language dedicated to the specification of transition triggers. Finally, it presents actions on transitions.

3.6.1 Factors in Guards

A factor in a guard specifies how many times (local cycles) the condition must be true before the transition is fired. The syntax for is:

```
n times c
```

The condition must be true exactly n times (independently of the values taken by the guards of the other outgoing transitions) and then becomes false forever. If the state is resumed later, this condition cannot be true again, only a restart of the source state is able to reactivate the counter. The

frequency divider presented in 3.4.4 "Emitting Signal on Transition" divides by two the frequency of its input. It could divide by three by just adding a state to count one more cycle. A more general way is to use a factor:

```
node fdiv_n (n: int32; c: bool) returns (nth_c: bool)
sig
    s_nth_c;
let
    nth_c = 's_nth_c;
    automaton
    initial state S1
    unless if (n-1) times c resume S2;
    state S2
    unless if c do {emit 's_nth_c} restart S1;
    returns ..;
tel
```

This node computes the division of the frequency of its input by n. In this extended version, the state S1 counts n-1 occurrences of true in the state S1 and then goes to S2. Note the transition that returns to S1 restarts S1: it is necessary to reset the counter implementing the factor.

This counter is part of the source state of the transition and is reset when its source state is reset. It can be simplified into one state:

```
node fdivn_2 (n: int32; c: bool) returns (nth_c: bool)
sig
    s_nth_c;
let
    nth_c = 's_nth_c;
    automaton
    initial state S1
    unless if n times c do {emit 's_nth_c} restart S1;
    returns .. ;
tel
```

Note that the factor is not restricted by the language to be a constant expression, any flow can be used. The value initializing the counter is the value of the dataflow expression at the cycle where the counter is reset.

The operator **`times`** is part of the Scade language and can be used anywhere a Boolean value can, or appear several times in an expression. For instance the guard of a transition can be `(3 times C)` or `(2 times D)`. It can also be used in a block of dataflow equations. The exact behavior is defined by the following equivalent node:

```
node times_equivalent (n: int32; c: bool) returns (o: bool)
var
    v3, v4: int32;
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
```

3.6.2 Forks

A transition has one source state and one target state. Yet, several transitions going from the same source state may share parts of their guards. Scade provides a way to refine a transition by forking its target state allowing the specification of decision paths that are explored in a single cycle. The syntax of such a transition is:

```
unless
  if c1 if c11 restart S2
        elsif c12 if c121 restart S3
                  elsif c122 resume S4
                  else resume S5 end
        elsif c13 resume S6
        else resume S7 end;
```

To illustrate this construct, consider a node that manages the position of a point piloted by five entries: four for the directions and one to unlock the command; this way, the user needs to push two buttons to start a command.

```
node point (u, d, l, r: bool; unlock: bool) returns (x, y: int8 last = 0)
let
  automaton
  initial state Stop
  unless if unlock if u resume Up
    elsif d resume Down
    elsif l resume Left
    elsif r resume Right end ;
  let
  tel

  state Up
  unless if (not u) resume Stop ;
    y = last 'y + 1;

  state Down
  unless if (not d) resume Stop ;
    y = last 'y - 1;

  state Left
  unless if (not l) resume Stop ;
    x = last 'x - 1;

  state Right
  unless if (not r) resume Stop ;
    x = last 'x + 1;
  returns x,y ;
tel
```

3.6.3 Flow Definitions on Transition

At this point, a state machine is made of several dataflow definitions embedded into states and the activation of these states is driven by transitions. Furthermore, as seen in 3.4 “Pure Signals”, the transitions can carry *actions*, which are signal emissions. These actions are activated anytime the transitions which carry them is fired.

This section presents another way to design a model with state machines where the transitions carry dataflow definitions. The previous operator (`point`) can be equivalently rewritten by computing the flows directly onto the transitions.

```
node point (u, d, l, r: bool; unlock: bool) returns (x, y: int8 last = 0)
let
  automaton
  initial state Stop
  unless
  if unlock
  if u do let y = last 'y + 1; tel resume Up
  elsif d do let y = last 'y - 1; tel resume Down
  elsif l do let x = last 'x - 1; tel resume Left
  elsif r do let x = last 'x + 1; tel resume Right end;

  state Up
  unless
    if u do let y = last 'y + 1; tel resume Up ;
    if true resume Stop;
  state Down
  unless if (not d) resume Stop ;
    if d do let y = last 'y - 1; tel resume Down ;
    if true resume Stop;
  state Left
  unless
    if l do let x = last 'x - 1; tel resume Left ;
    if true resume Stop;
  state Right
  unless
    if r do let x = last 'x + 1; tel resume Right ;
    if true resume Stop;
  returns x, y;
tel
```

The actions carried by the transitions are equation blocks enclosed between the **`let`**- **`tel`** keywords. One of this block is activated when its transition is fired.

SYNTAX

An action on a transition may be:

- signal emissions:

```
do {emit 'S1; emit 'S2 if expr}
```

- an equation block:

```
do
    sig
    -- signal declarations
    var
    -- variable declarations
    let
    -- equations, state machines, conditional blocks, signal emissions
    tel
```

It is not allowed to define a flow into a state and on a transition of the same state machine. However, a variable defined on transitions can be used into a state with respect to the causality rules.

INSTANTS OF ACTIVATION AND DEFAULT VALUES

The transitions of a state machine define a set of variables. The states of the state machine define another set of variables, both are disjoint. The union of these two sets is the set of variables defined by the state machine and it appears in the `return` statement of the state machine.

At each instant, there is at most one transition which is fired (called the active transition). The actions of this transition are activated and the variables are computed as if standing into the active state. In the same way as variables defined into states, a variable that does not occur into the active transition takes its default value.

4 / Conditional Blocks

The previous chapter introduced the state machines and the way they behave conjointly with Data Flows. A state machine is a very common way to introduce control structures in an application, it covers the case where the control itself (the active state) is a function of the previous state and some Boolean conditions.

There are simpler cases of control that correspond to situations where the current activation depends only on conditions computable in the current cycle, this kind of intention is best expressed by using a simple decision mechanism.

To cover these cases, Scade introduces the conditional blocks, which from a semantics point of view are close to the dataflow activation condition, but with a more imperative flavor. Moreover, this new construct is very close to state machines and provides an easy way to have shared memories (see 3.3 "Sharing Memories Between States").

4.1 Boolean Case: Activate if Construct

This new construct has the following syntax:

```
function roots (a, b, c: float64) returns (xr, xi, yr, yi: float64)
var
  delta, _2a: float64;
let
  delta = b*b - 4.0 * a*c ;
  _2a = 2.0 * a;

  activate
  if delta > 0.0
  then
    var d : float64;
    let
      d = sqrt (delta) ;
      xr, xi = ((-b + d) / _2a, 0.0) ;
      yr, yi = ((-b - d) / _2a, 0.0) ;
    tel
  else if delta = 0.0
  then
    let
      xr, xi = (-b / _2a, 0.0);
      yr, yi = (xr, xi );
    tel
  else -- delta < 0.0
    let
      xr, xi = (-b / _2a, sqrt (-delta) / _2a);
      yr, yi = (xr, - xi);
    tel
  returns xr, yr, xi, yi;
tel
```

The activation condition pattern with default value given in 2.2.4 “Node Activation with Default Flows” is expressed with the following code example:

```
node two_instances (e: int64; h: bool) returns (s, t: int64)
let
    s = integr (e);
    t = (activate integr every h default 0) (e);
tel
```

With the new construct, the above pattern has an alternative notation that can be expressed as follows:

```
node two_instances (e: int64; h bool) returns (s: int64; t: int64)
let
  s = integr (e);
  activate
    if h
    then t = integr (e);
    else t = 0;
  returns t;
tel
```

Considering these simple cases, both of the above patterns are semantically equivalent, yet they allow to represent two different views of the same function.

The first one, by choosing to specify the activation in the expression language, can represent graphically a conventional Data Flow diagram and admits a coherent interpretation in this representation.

The second one can represent the decision path control, then inside each decision block, the equation set can be represented by Data Flows.

The reason for choosing one or the other pattern is more a matter of convenience that may depend on the culture of the user. The activation condition in the expression allows to cover simple cases. Instead, when the control is complex and the set of equations share a lot of memories, it is certainly preferable to use conditional block patterns.

4.2 Enumerated Case: Activate when Construct

Along the same principles exposed in 4.1, Scade provides a way to activate a block depending on an enumerated value under the constraint that all cases of the enumeration are covered (exhaustiveness of cases).

This new construct has the following syntax:

```
type
Tcommand = enum {Left, Right, Up, Down, Rotate, Stop};

const
teta: float64 = 0.1;

node coordinate (command : Tcommand)
returns (x: float64 last = 0.0; y: float64 last = 0.0)
let
  activate Direction when command match
    | Left : x = last 'x - 1.0;
    | Right : x = last 'x + 1.0;
    | Up : y = last 'y + 1.0;
    | Down : y = last 'y - 1.0;
    | Stop : -- do nothing
    | Rotate :
    var s, c: float64;
    let
      s = sin (teta);
      c = cos (teta);
      x = last 'x * c;
      y = last 'y * s;
    tel
  returns x, y;
tel
```

5 / Array Data Types

Scade allows the manipulation of array data values as any other data type. Scade arrays are not the “memory blocks” that classical programming languages (such as C) define; a Scade array is a flow like any other value manipulated in a Scade design. Especially, the language does not provide any way to perform in-place modifications, nor any allocation/freeing primitive to handle memory directly. Furthermore, any array defined in a Scade design must have a size known at compile time to guarantee that the memory used by its execution is always constant.

Note

In Scade, the first element of an array has index 0.

5.1 Type Notation

An array type expression is specified by the base type and the size of the array separated by the ^ character. Such a construct can be used anywhere as an expression: in a type definition, in a node signature, in a flow declaration, etc.

```
type Vect = int32 ^ 3 ;
```

This defines the type `Vect`, a vector of three integers.

The *base type* can be any type; in particular it can be an array type to declare arrays of arrays. The *dimension* must be a statically evaluable strictly positive integer expression. Statically evaluable means that the expression defining the size contains only constants and statically evaluable operators (no memory, no function call, only Boolean or integer manipulations).

5.2 Basic Operators

This data type comes with a family of operators and predefined functions to perform simple manipulations of array values.

5.2.1 Array Constructor: Value Enumeration

The simplest way to define an array is to enumerate its values:

```
type Vect_T = T ^ 3 ;
const v : Vect_T = [ exp_1 , exp_2 , exp_3 ] ;
```

Example: Node computing the square root of a float.

```
type SignRoot = float64^2;
function SquareRoot(x: float64) returns (root: SignRoot)
let
    root = [if x>0.0 then sqrt(x) else 0.0,
    if x>0.0 then 0.0 else - sqrt(-x)];
tel
```

5.2.2 Array Constructor: Value Repetition

Another way to define an array is to repeat a value using the operator '^'.

```
type Vect_T = T ^ 3 ;
const v : Vect_T = expr_val ^ 3;
```

5.2.3 Array Access: Static Indexation

If `v` is an array, `v[i]` is its $(i+1)^{th}$ element (where `i` is a static expression)

```
function ScalarProduct2(x,y: int32^3) returns (z:int32)
let
    z = x[0]*y[0] + x[1]*y[1] + x[2]*y[2] ;
tel
```

5.2.4 Array Access: Dynamic Indexation

When implementing generic algorithms on arrays, it is often useful to be able to access a particular flow in the array according to, say an input value. Therefore, the constraint related to the static evaluability of array sizes can be relaxed in a well-defined construct, allowing the user to access an array according to a dynamic expression. Notice that any Scade expression can be used in this case.

To prevent any runtime error, the dynamic indexation is defined as a total function: a default value must be defined when accessing an array. This value is used in case the dynamic index is outside of the array bounds.

```
(v.[i] default d)
```

If `i` is within the range of the array `v`, the operator returns the element of the array at this index; otherwise, the operator returns `d`. Note the slight difference with static array access: here the index is preceded by a dot (`_.[]`). This operator supports a list of indexes:

```
(v.[in]...[i2][i1] default d)
```

In this last case, `v` is supposed to have a type like `T^N1^N2^...^Nn` and `d` must be of type `T`. The generalized behavior consists in checking that each `ik` is in the range determined by `Nk` and returns the element at this position if the condition is satisfied; it returns `d` otherwise.

5.2.5 Array Access: Static Slice

It is possible to extract an array from a wider array by providing two static indexes (`i1` and `i2`) that specify the slice:

```
v[i1 .. i2]
```

`i1` is smaller or equal to `i2`, and both belong to a correct range. The size of this slice is `i2 - i1 + 1`.

5.2.6 Array Concatenation

This binary operator builds an array of size n1+n2 from an array of size n1 and an array of size n2. such that if:

```
v = t1 @ t2;
```

`t1` (resp. `t2`) is the array of size n1 (resp. n2), the following holds:

$\forall i \in [0 \ldots n_1[, v[i] = t1[i]$ and $\forall i \in [n_1 \ldots n_2(1)[, v[i] = t2[i - n_1]$

Example: Shifting the content of an array to left or right. The node below implements the rotating case, where the new value is the one that is pushed out by the shifting operation:

```
const
n: int32 = 100;
function shift (e:int32^n) returns (s_left, s_right : int32^n)
let
    s_left = e[1 .. (n-1)] @ [e[0]];
    s_right = [e[n-1]] @ e[0 .. (n-2)];
tel
```

Shifting to the left (resp. right) is done by concatenating the slice containing the n-1 last (resp. first) elements with the array containing the first (resp. last) one.

5.2.7 Reverse

Reverse permutes the values in an array such that, if:

```
v = reverse w;
```

The following invariant holds: $\forall i \in [0 \ldots (n-1)] \backslash k, v[i] = w[n-i-1]$

5.2.8 Array Constructor: Array Copy with Modification

It is possible to build an array from an existing array while specifying an index, possibly dynamic, for which they differ:

```
v = (w with [k] = e);
```

If *w* is of type *array of size n*, this construct satisfies:

$\forall i \in [0 \ldots (n-1)] \backslash k, v[i] = w[i]$ and $k \in [0 \ldots (n-1)] \Rightarrow v[k] = e$

In the cycles where k is out of the bounds of the array $k \notin [0 \ldots (n-1)]$, note that this operator behaves as an identity (v = w). Following the extended notation of the dynamic indexation, this constructor supports a list of indexes:

```
v = (w with [k1][k2]...[kn] = e);
```

5.3 Multidimensional Operators

Array types are unidimensional: it is not possible in Scade to directly define multidimensional arrays. A matrix is encoded by an array of arrays, as in:

```
type SpaceLinearMorphism = float64 ^ 4 ^ 6 ;
mat : SpaceLinearMorphism ;
```

This defines an array of six vectors, each vector being of size 4. The projection `mat[i][j]` corresponds to the access to the i^{th} vector (where i belongs to [0..5]), then access to the j^{th} component of this vector.

5.3.1 Transpose Two Dimensions

It is possible to swap two arbitrary dimensions of an array of array of etc...

Let type T have the following definition:

```
T = t^nk^...^nd2^...^nd1^...^n1;
```

d1, d2 are integers in range `[1..k]`. Let *w* be an array of type T. Let v be defined as:

```
v = transpose (w; d1; d2);
```

Then, *v* is an array of type : `tnk...nd1...nd2...n1`. The dimensions d1 and d2 are permuted and the content of the arrays satisfies:

$$\forall i_1,i_2,\ldots\ i_k,\ v[i_1]\cdots\ \underline{[i_{d2}]}\cdots\ \underline{[i_{d1}]}\cdots\ [i_k] = w[i_1]\cdots\ \underline{[i_{d2}]}\cdots\ \underline{[i_{d1}]}\cdots\ [i_k]$$

This operator naturally extends the usual bidimensionnal transpose operator:

```
const
    m: uint16 = 10;
    n: uint16 = 20;
type
    Mat_mn = float64^m^n;
    Mat_nm = float64^n^m;
function MatrixTransposition (A: Mat_mn) returns (B: Mat_nm)
let
    B = transpose (A; 1; 2);
tel
```

5.4 Iterators

Scade provides primitives to iterate operator instantiation over array items following different patterns: map, fold or a combination of both mapfold.

The general syntactic form of iterators is:

```
x = (iterator Op <<size>>) (arguments) ;
```


5.4.1 Map

A **map** offers a way to introduce several instances of an operator where each instance takes its argument from a list of array. This form of application produces arrays containing the production of each instance.

For instance, let us consider an operator Op that takes two inputs and produces one input. The point-wise application of this operator consists in taking the i-th value of the input arrays to produce the i-th value of the

output array. The example below illustrates this with the point-wise application of Op to two arrays of size n; where this application defines array z.

```
z = [Op(x[0],y[0]), Op([x[1], y[1]), ..., Op(x[n-1],y[n-1]);
```

This expanded form can be expressed more concisely and for any size n with the map iterator applied on operator Op:

```
z = (map Op <<n>>)(x,y);
```

Example 1: Pointwise sum of two arrays

```
function sum_scalar(a,b: int32) returns (c: int32)
let
    c = a + b ;
tel

function sum_array(t,u: int32^3) returns (v: int32^3)
let
    v = (map sum_scalar <<3>>)(t, u) ;
tel
```

Array arguments filled to a **map** iterator must all have the size specified in the iterator application. It is possible to combine a scalar with an array using an operator on arrays by promoting the scalar as an array. For instance, creating an array in which all items are those of an array w of size 3 plus a given increment i can be done with `sum_array(w,i^3)`. The second example below illustrates this principle: multiplying a scalar by a vector. This vectorization of a scalar value (expression x^3 in example 2) can be understood as a distribution of x over the iteration (`map mult_scalar <<3>>`).

Example 2: Multiplication of all the array elements by a scalar

```
function mult_scalar(a,b: int32) returns (c: int32)
let
    c = a * b ;
tel

function mult_scalar_array(x: int32; t: int32^3) returns (u: int32^3)
let
    u = (map mult_scalar <<3>>) (x^3, t) ;
tel
```

When using a map scheme, the instances introduced by the map do not communicate (none of them produces a value consumed by another); this can be seen in the initial example with its expanded form (*i.e.,* expressed without map).

5.4.2 Fold

In previous section, we have seen that `map` allows to introduce several independent instances of an operator. Another common interesting scheme for multi-instantiation is the cascade where the first instance feeds the second, etc until the last one that produces the expected result. For instance, consider the function `sum_array` defined below:

Example: Sum of the elements of an array

```
function sum_array(t: int64^3) returns (s: int64)
let
    s = sum_scalar(sum_scalar(sum_scalar(0, t[0]), t[1]), t[2]) ;
tel
```

It can be expressed in a more concise way using a **`fold`** iteration scheme.

```
function sum_array(t: int64^3) returns (s: int64)
let
    s = (fold sum_scalar <<3>>) (0, t) ;
tel
```

In this example, the iteration starts with 0 as the initial value of the accumulator. It is also possible to define the sum `t[0] + t[1] + t[2]` as a **`fold`** iteration initiated by `t[0]` and based on the slice `t[1..2]`. The Scade code to express this is:

```
function sum_array(t: int64^3) returns (s: int64)
let
    s = (fold sum_scalar <<2>>) (t[0], t[1 .. 2]) ;
tel
```

This operator can be generalized to work on an array of an arbitrary size `n`:

```
function sum_array<<n>>(t: int64^n) returns (s: int64)
let
s = (fold sum_scalar <<n-1>>) (t[0], t[1 .. (n-1)]) ;
tel
```

5.4.3 Mapfold

The **mapfold** iterator combines **map** and **fold** schemes.

Example: Build an array enumerating the first natural integers

```
function sum_dup(a,b: int64) returns (s1,s2: int64)
let
    s1 = a + b ;
    s2 = s1 ;
tel

function enum_int() returns (t: int64^10)
var
    aux: int64;
let
    aux, t = (mapfold sum_dup <<10>>) (0, 1^10) ;
tel
```

5.4.4 Iterators with Access to the Index

To facilitate the mapping between usual loop instructions and iteration schemes of Scade, the language has two iteration schemes where iterated operators take as first argument an integer flow which is fed by the index of the iteration. These schemes are not primitives and can be written using the **mapfold** scheme.

- "mapi"
- "foldi"

MAPI

Let Op be an operator requiring 3 arguments, the first being an integer, then x and y two arrays of size n. The following expression:

```
z = [Op(0, x[0],y[0]), Op(1, x[1],y[1]),… , Op(n-1, x[n – 1], y[n – 1])];
```

can be expressed in a concise and general (independent of n) way as:

```
z = (mapi Op <<n>>) (x, y);
```

Another way to get an array enumerating the first natural integers is:

```
function id(i: int64) returns (o: int64) o = i ;

function enum_int() returns (t : int64^10)
let
    t = (mapi id <<10>>) () ;
tel
```

FOLDI

Similarly, **`foldi`** iterator applies an operator whose first argument has to be of integer type, and according to the **`fold`** scheme.

For instance, defining a function that takes A an array of integers and computes the sum of `A[i]` such that `A[i] > i` can be done in the following way:

```
function add_if_above_diag(i, acc, a: int32) returns (acc_o: int32)
  acc_o = acc + (if a > i then a else 0);

function sum_above_diag <<n>>(A: int32^n) returns (sum: int32)
  sum = (foldi add_if_above_diag <<n>>)(0, A);
```

5.4.5 Partial Iterators

The iterators presented before are *total* in the sense that they apply on all the items of an array. Scade language also provides the possibility to restrict the iteration to the first items of their arguments until a certain condition is met.

- “Partial map”
- “Partial fold”

PARTIAL MAP

The main difference between **`map`** and **`mapw`** iterators is that the iterated operator `Op` also has a Boolean value that, when false, stops the iteration. Given such an operator taking `k` flows and returning `n+1` flows among which the first is Boolean, the application of the iterator has the following general form:

```
idx, t1,..., tn = (mapw Op <<size>> if cond default d1,..., dn)(a1,..., ak)
```

The iteration starts according to condition `cond`. This returns a new condition `cond'`, and values that are assigned to first elements of arrays `t1,...,tn`. If this new condition is evaluated to true, the iteration moves to the next rank of the input arrays. When the iteration stops, if the last index was not reached, then indices that were not visited by the iteration are filled using the specified default values `(d1, ..., dn)`. This behavior is strictly necessary to respect the Scade typing discipline, *i.e.,* all the values of a flow are of the same type and array sizes are part of the type, thus the produced array must be of the same size. The **`mapw`** operator must produce complete arrays of values of the given size. Hence the remaining values of the output arrays must be filled, using the default values. `idx` is given the first iteration index where application has not been allowed anymore.

This iterator also comes with its index access version: **`mapwi`**, in which case `Op` should take as first input an integer flow that is filled with the current array index considered. Consider the following example:

```
const n: int32 = 10;
function fill (i, max_i: int32) returns (condition: bool; v: int32)
let
     v = i;
     condition = i < max_i;
tel

function fill_up (max_i: int32) returns (s: int32^n)
let
     _ , s = (mapwi fill <<n>> if true default 0)(max_i^n);
tel
```

Therefore `v = fill_up(6)` leads to `v=[0, 1, 2, 3, 4, 5, 0, 0, 0, 0]`. The main interest of **`mapw`** is to avoid the computation of trailing values when iterated operator is complex or costly.

PARTIAL FOLD

Similarly, a **`fold`** iteration could sometimes be cut as soon as the desired result is obtained (looking at the presence of a given value in an array for instance).

This pattern takes an operator `Op` that, given *k* flows as inputs, produces two output flows: a Boolean flow and another flow. As for **`mapw`** iterator, the first Boolean flow is used to control the next application of operator `Op`. The following example implements an operator whose outputs are: a Boolean corresponding to the disjunction of the Boolean elements of an input array and an integer corresponding to the first index for which the condition is false. This computation could be stopped as soon as one element of the array is true.

```
const n : uint8 = 10;
function Op (c,d: bool) returns (cond_o, s: bool)
let
    s = c or d;
    cond_o = not s;
tel

function disjunction(B: bool^n) returns (exist: bool; i: bool)
let
    i, exist = (foldw Op <<n>> if true) (false, B);
tel
```

In the above example, the iterated operator `Op` is a stateless function. This iterator applies also to stateful nodes, for these cases, the semantics has to define the activation of each instance. Starting from the instance that falsifies the conditions all following instances are not activated during the cycle: their internal state is preserved. The following example gives an illustration of this behavior.

Example: Iteration of a one bit counter with a stop condition that is equal to the carry value *co*. This means that the iteration stops as soon as *co* is false. The iteration returns the position of the last modified bit (in the range [1..n]).

```
const n: uint8 = 10;
node single_bit_counter (ci: bool) returns (condition, co: bool)
var
    r, v : bool;
let
    v = false -> pre r;
    r = ci xor v;
    co = v and ci;
    condition = co;
tel

node last_changed_bit () returns (i: uint8)
let
    i, _ = (foldw single_bit_counter <<n>> if true)(true);
tel
```

Note the use of _ to discard the value of the accumulator returned by the **`foldw`** operator. As the array indexes start at 0, the result can be interpreted as the last modified bit indexed from 1. This expression computes the following sequence:

last_changed_bit()	1	2	1	3	1	2	1	4	1	2	...

Again, **`foldwi`** is the iterator extended to take into account the index of the application. Given an integer *max_i*, the operator `sum_up` sums up the elements of the array `T` whose indexes are in the range `[0..max_i]`.

```
const
    n : uint8 = 10;
    V : int_n = [9,8,7,6,5,4,3,2,1,0];

function sum_stop (i,x,y,max_i: int32) returns (condition: bool; s: int32)
let
    s = x + y;
    condition = i < max_i;
tel

function sum_up (T: int32^n; max_i: int32) returns (s: int32)
let
    _ , s = (foldwi sum_stop <<n>> if true)(0, T, max_i^n);
tel

function example_sum_up (max_i: int32) returns (sum: int32)
let
    sum = sum_up (V, max_i);
tel
```

max_i	3	2	1	0	...
example_sum_up(max_i)	30	24	17	9	...

Partial iterators and Worst Case Execution Time

Partial iteration is interesting only if the application context (real-time requirements, method used to determine Worst-Case Execution Time) allows to take advantage of it. Otherwise, it is recommended to use total ones that leads to simpler design.

The usage of partial iteration (or bounded while-loops) is not, in general, interesting for embedded applications where WCET must be provided. Computing WCET usually requires to give a bound for all while loops. Such bound information is taken into account by automatic tools for computing an execution time safe upper-bound. Thus, if WCET is computed by analyzing the code only, partial iteration is useless.

In cases where the user has some information about data (statistics about the average number of pertinent values in an array), WCET may be better approximated by taking these information into account. These partial iterators are relevant only in this last case, and their usage should always be carefully justified.

5.5 Parametrization of Operator with Sizes

Most of the algorithms on arrays can be parametrized by the sizes of the input or output arrays. The aim of the proposition is to introduce a way to describe such parameters in a safe way that can ensure that the sizes of the arrays are respected. Sizes are identifiers that appear in the formal interface.

For instance, **node** `f <<n,m>> (...)` specifies a node parametrized by two dimensions `m` and `n`. Here is the example of the matrix product that applies to any dimensions:

```
function prod_sum (a,v,w: float64) returns (aa: float64)
let aa = a + v*w; tel

function ScalProd <<n>> (V,W: float64^n) returns (sp: float64)
let
   sp = (fold prod_sum <<n>>) (0.0, V, W);
tel

function MatVectProd <<m,n>> (A: float64^m^n; u: float64^n) returns (w: float64^m)
let
   w = (map (ScalProd <<n>>) <<m>>)(transpose (A; 1; 2), u^m);
tel

function MatProd <<m,n,p >>(A: float64^m^n; B: float64^n^p) returns (C: float64^m^p)
let
   C = (map (MatVectProd <<m,n>>) <<p>>)(A^p, B);
tel
```

Example:

```
node f(M1: float64^10^5; M2: float64^5^10; M3: float64^10^10) returns (S: float64^10^10)
let
    S = (MatProd <<10,10,10>> )((MatProd <<10,5,10>> )(M1, M2), M3);
tel
```

Sizes are all required to be statically evaluable strictly positive integer constants. Moreover, sizes can be filled to other sizes in a node instantiation. This allows the propagation of integer constants while developing large projects.

6 / Modeling Features

As a modeling language, Scade provides some features that do not affect the semantics of the program in terms of input/output relations, but allows to add constraints whose interpretation and implementation is tool dependent. The kind of activities that is usually done on a model includes:

- Code generation
- Simulation
- Formal verification
- Test generation

Note

The precise impact of these features on modeling steps is defined in the corresponding specification of the tools.

6.1 Scade Contracts: Assume and Guarantee Observers

Design-by-contract is a famous software engineering principle. A contract is a specification of expectations before applying a function (or a node) and, supposing they are verified, of some properties on the results. Within the synchronous data flow model of Scade, this contract could be presented as a pair of observers:

- one corresponding to expectations: **`assume`** observes inputs and the past of outputs
- one corresponding to ensured properties: **`guarantee`** observes inputs and outputs

6.1.1 Syntax

Example:

```
function N (a,b: int32) returns (c: int32)
let
    assume hyp1 : a > b;
    guarantee c_positive : c > 0;
    c = 10 * (a - b);
tel
```

The identifier used for the observer is in the same namespace as the flows identifiers.

6.1.2 Type and Causality Checking

TYPE

The expression of an observer is of type `bool`.

RULE

The expression of an **`assume`** *does not depend* instantaneously on the outputs.

6.1.3 Initialization Analysis

RULE

The expression of an observer (**`assume`** *or* **`guarantee`**) *must always be defined (no nil value in the Boolean flow)*.

JUSTIFICATION

An observer is defined with the idea that observing a correct implementation consists in observing an invariant, *i.e.,* the constant flow `true, true, ...` for the observed invariant. But an uninitialized flow (**`pre`** `(a` **`or`** `b)` for instance) always starts with an undefined value *nil*. This verification is important because a contract that does not satisfy it may be considered true for the very first cycle of its clock while it is actually indeterminate.

6.1.4 Compiler and Simulator Support

The information given by these observers is not used to optimize the code. The generated instrumented code used for debug purposes in the simulator gives access to the value of all the observers by their identifier and a path in the instance graph of the application (as for standard Lustre variables). The instrumentation consists in transforming the observer into an equation and declaring the name of the observer as an observable flow :

```
function N (a,b: int32) returns (c: int32)
var
    probe hyp1, probe c_positive: bool;
let
    hyp1 = a > b;
    c_positive = c > 0;
    c = 10 * (a - b);
tel
```

6.2 Probing a Flow

The keyword **probe** used in a local flow variable declaration is used to express the intention to observe this flow.

```
...
var
    probe x: int32;
...
```

Important

A probed flow must be defined at the first cycle to prevent that the observed value is not relevant at its first cycle. Moreover, a probe should not be observed at the cycles where the clock of the probed flow is false.

7 / Structuring Models

This chapter presents issues related to the management of large projects. First, a structuring mechanism that allows to encapsulate Scade objects is described. Then, the naming discipline is sketched to exhibit some allowed overloadings.

7.1 Packages

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

7.1.1 Principles

A package definition is a set of declarations introduced by the keyword **package**, followed by a name and terminated by the keyword **end** followed by a semi-colon.

```
package Integer
type T = int32 ;
const ZERO : T = 0 ;
function plus(x,y: T) returns (z: T) z = x+y;
end;
```

This example declares the package `Integer` which contains a type `T`, a constant `ZERO` and a function `plus`. These objects can directly be accessed inside the package (notice that `ZERO` and `plus` declarations use the type `T`). A package defines a namespace, that is to say a lexical scope for naming. Consequently, it is possible to declare two different objects (types, constants, nodes or any other) in two different packages with the same identifier. There is no ambiguity w.r.t. the complete name including the module identifier.

```
package Float
    type T = float64 ;
    const ZERO : T = 0.0 ;
    function plus(x,y: T) returns (z: T) z = x+y;
end;
```

To access an object `o` declared in a package `p` outside this package, one has to use a complete name notation: a unique path of packages and sub-packages, separated by two colons `::` and ended by the name of the object:

```
function double (x: Integer::T) returns (y: Integer::T)
y = Integer::plus(x, x);
```

Of course, objects declared in a father package can be accessed directly from its subpackages. In this case, the first name matching this object while backtracking into the father stack is taken:

```
package P1
    const foo:int32 = 2;
    package P2
    const bar:int32 = foo + 1;
    end;
end;
```

7.1.2 Global Package

The declarations belonging to the top level (*i.e.,* not inside any package), are declarations of the *global* package which contains all the declarations of a model. The global package is the only package which can be split into several text files.

7.1.3 Visibility Rules

By default, all the declarations of a package are visible outside this package through the complete name notation. However, for software engineering purposes, it is possible to hide a declaration and to forbid its direct usage outside the package. Any declaration can be prefixed by a visibility status:

- **`public`**: is the default case; the declaration can be used anywhere.
- **`private`**: is the most restrictive case; the declaration can only be used inside the package. For the rest of the design, the declaration simply does not exist.

7.1.4 Opening a Package

To alleviate the usage of packages, it is allowed to *open* a package into another package. This is done using the following declaration, which can stand anywhere in the package (the order of declaration is never relevant in a Scade design):

```
open PackageName;
open PackageName::SubPackageName;
```

When a package is opened, its public declarations can be accessed directly within the current package and all its sub-packages. However, the declarations of the sub-packages of the opened package cannot be accessed as illustrated in the example above: the first line is not enough to access the declarations of `SubPackageName`. The complete name with the packages path is not mandatory anymore, but can still be used. A package can be opened in another package only if there is no name collision resulting from

this operation. For instance, it is forbidden to open a package containing the declaration of an object `foo`, if the package where the opening is performed also contains a declaration of `foo`, as in:

```
package P1
    const foo: int32 = 3;
end;
package P2
    open P1;
    const foo: int32 = 4;
end;
```

Above the clash is between the declarations of the two constants `foo`.

7.2 Namespaces

This section describes the namespace and scoping rules of Scade. A namespace is an abstract container providing context for names (identifiers or paths), allowing disambiguation of items having the same name by acknowledging their distinct namespaces. As a rule, names in a given namespace cannot have more than one meaning, that is, two or more things cannot share the same name within a single namespace.

The Scade language defines two distinct namespaces: the *package namespace*, noted **PN**, and the *declaration namespace*, noted **DN**. A package and a declaration of any kind (type, constant, etc.) can have the same name and be referred to in an unambiguous way at the same point of a program:

```
package P1
   ...
end;
const P1: int32 = 3;
```


7.2.1 Packages and Namespace

A package definition P1 introduces its two local namespaces PN1 for sub-packages and DN1 for definitions. When opening another package P2, the definitions in P2 are added to the name spaces of P1, thus the condition DN1 ∩ DN2 = Ø must be satisfied.

Names introduced by a sub package hide the ones introduced by upper packages.

It is then possible to access any declaration made in a package from another package (as long as they are visible): either through a complete name, or after opening the corresponding package.

7.2.2 Declarations and Namespace

Within the declaration namespace, three different namespaces are distinguished:

1. for the *flows* (variables, sensors, constants, enumerated values), *signals*, *assumes/guarantees*, *node* (imported or not), *types* and *groups*.
2. for the *state machines*, *states*, *conditional blocks*.
3. for the *structure labels*.

For instance, the state of a state machine can have the same name as the node it is defined in, a label may have the same name as the type it belongs to, and so on. The extension policy is different from the one used for packages. When entering one of these three namespaces, the whole content of the current state of **DN** is available. However, if a local declaration in a state machine, a state transition, etc., is performed, then it automatically overloads previous declaration in **DN**. And contrary to the package policy, there is no way to access an overloaded object.

7.2.3 Constructs

Let us now detail for each construct of the Scade language that introduce new scopes, how the various parts of the declaration namespace are built.

- "User operator"
- "Structured types"
- "State machine"
- "State"
- "Transition actions"
- "Conditional blocks"
- "Package"

USER OPERATOR

```
function ID (...
...
    -- new Namespace 1
    -- new Namespace 2
... ...
tel
```

```
node ID (...
...
-- new Namespace 1
-- new Namespace 2
...
tel
```

```
function imported ID (... -- new Namespace 1 ...);

node imported ID (... -- new Namespace 1 ...);
```

STRUCTURED TYPES

```
type
    ...
    T = ...
    -- new Namespace 3
    ...
    ;
```

STATE MACHINE

```
...
    automaton ID
    ...
    -- new Namespace 2
    ...
    returns ...;
...
```

STATE

```
...
    state ID
    ...
    ...
    -- new Namespace 1
    -- new Namespace 2
    ...
...
```

TRANSITION ACTIONS

```
...
    until/unless if
    ...
    do
    ...
    -- new Namespace 1
    -- new Namespace 2
    ...
    tel
    returns ...;
...
```

CONDITIONAL BLOCKS

```
...
  activate ID
  if ...
  then
    ...
    -- new Namespace 1
    -- new Namespace 2
    ...
  else
    ...
    -- new Namespace 1
    -- new Namespace 2
    ...
  returns ...;
...
```

```
...
  activate ID when ... match
  | ...:
    ...
    -- new Namespace 1
    ...

  | ...
  returns ...;
...
```

PACKAGE

```
package ID
    ...
    -- new Namespace 1
    -- new Namespace 2
    ...
end;
```

8 / Genericity

The previous chapter presented a means to structure models when managing large models. The present chapter introduces the way to define generic Scade operators.

8.1 Polymorphism

This section deals with polymorphism from the following perspectives:

8.1.1 Principles

In some cases, a node is defined independently from the type of its arguments. The following node samples two integer flows alternatively at each instant:

```
node sample_int32 (a,b: int32) returns (c: int32)
var
flag : bool ;
let
    flag = true -> not (pre flag) ;
    c = if flag then a else b ;
tel
```

The fact that a,b and c are integers is not used in the node definition. If one defines the same node for an arbitrary type Foo, only the signature of the node is different.

```
node sample_Foo (a,b: Foo) returns (c: Foo)
var
  flag: bool ;
let
    flag = true -> not (pre flag) ;
    c = if flag then a else b ;
tel
```

Scade provides a way to define generic nodes, also called polymorphic nodes, that abstract away unintended data types. Genericity is expressed by using type variables written as quoted ident (preferably starting with an upper case to differentiate them from signals presence). The node above can then be written:

```
node sample_generic (a,b: 'T) returns (c: 'T)
var
  flag: bool ;
let
    flag = true -> not (pre flag) ;
    c = if flag then a else b ;
tel
```

The quoted ident 'T represents a type variable: a, b and c can be of any type. The only constraint they have to fulfill is that they belong to the same type. If more than one type variable is required in a node profile, one has to use a different quoted ident.

Important Note

Type variables are universally quantified. The type of the `sample_generic` node is read as: "for any type T, the node takes two flows of type T and returns a flow of type T".

8.1.2 Typing Rules

The typing rules in presence of type variables are very close to the usual Scade rules. There are two more constraints for nodes and functions:

- the type variables occurring in the right hand side of the signature must also appear in the left hand side.
- the type variables occurring in the local variable types must appear in the left hand side of the signature.

For instance, the following node signature is refused, since the type variable has not been introduced on the input side:

```
node Id (a: int32) returns (b:'T)
```

8.1.3 Restrictions

Type variables cannot be used everywhere you have types. No type variable are allowed in:

- Type declarations: `type foo = 'T` is forbidden
- Sensors or imported constants
- The profile of the root node of any package, even the global one:

```
package PolymorphicArithmetics
    function imported Pol_plus(x,y: 'T) returns (z: 'T);
    function twice(x: 'T) returns (y: 'T) y = Pol_plus(x, x);
end;
```

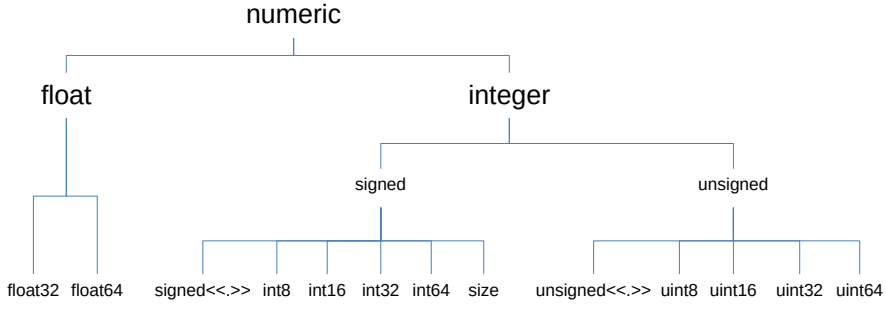
The restrictions listed here and the type verification mechanism ensure that a program that has been accepted by the compiler does not lead to a runtime error resulting from the instantiation of the type variables by contradicting types.

8.2 Types Hierarchy and Sub-Typing

Usual programming languages use *ad-hoc polymorphism* to overload arithmetic operators. This approach is not really relevant in a parametric polymorphic context. Moreover, such systems are not sophisticated enough to deal with a complex numeric type system. Scade implements a sub-typing mechanism similar to the type system suggested and described by Phillip Wadler.

8.2.1 Numeric Type Hierarchy

Numerical data types are organized as a type hierarchy:



Type kinds are represented as boxed types, and data types as simple types. The kind of a type variable is used to restrict the polymorphism to the type hierarchy. The functional node that returns the double of its input may be declared as:

```
node twice (a : 'T) returns (b: 'T) where 'T numeric
let
    b = a + a ;
tel
```

This node is polymorph, it can be applied on an integer flow or a floating point flow, but it does not have the freedom of a polymorph node in the sense of the previous section.

8.2.2 Predefined Operators

Arithmetic operators have a predefined polymorphic numeric type which follows the signature of an operation. For instance the operator + performing the sum operation has the following type:

$$\forall\ T \in \text{numeric} : \quad T \times T \rightarrow T$$

Note

The type-checking performs no sanity check, it does not check overflow or division by zero.

8.2.3 Polymorphic Literals

The literals built only with integers have a polymorphic type of kind **`numeric`**. The literals built with integers and symbol `.` or `e`, have a polymorphic type of kind **`float`**. Numeric literals are more complicated because their polymorphic nature introduces a free type variable. In the more general case, this variable is linked with the type variables of the arguments of the node where the literal occurs.

```
node HalfSucc(a:'T) returns (b:'T) where 'T numeric
var
  count : int32;
let
  count = 0 -> pre count + 1; -- 1st occurrence of the literal 1
  b = if count mod 2 = 0 then a + 1 else a; -- 2nd occurrence
tel
```

In this example, the typing rules enforces the first `1` to be an `int32` and the second one to be a `'T`. To prevent type mismatch when analyzing this example, Scade provides a syntax to force a decimal integer value to be considered as a polymorphic value. The previous example becomes:

```
node HalfSucc(a:'T) returns (b:'T) where 'T numeric
var
  count : int32;
let
  count = 0 -> pre count + 1; -- 1st occurrence of the literal 1
  b = if count mod 2 = 0 then a + (1:'T) else a; -- 2nd occurrence
tel
```

8.3 Overloading with Specialization

Scade provides a feature like overloading: the call of the same function/node identifer may invoque different function/node bodies which differ from each other by their signature.

8.3.1 specialize

The following example states two functions `AverageInt` and `AverageReal` which *specialize* the function `Average` when the arguments are integers, respectively reals:

```
function imported Average (a,b:'T) returns (c:'T) where 'T numeric ;

function AverageInt (a,b: int32) returns (c: int32) specialize Average
  c = (a + b) div 2;

function AverageReal (a,b: float64) returns (c: float64) specialize Average
  c = (a + b)/2.0;
```

The usage of `AverageInt` and `AverageReal` is the same as usual but the calls to the `Average` function invoke effectively the `AverageInt` function for any instance where the type variable `'T` is instantiated by `int32`. For other types than **`int32`** and **`float64`**, the standard mechanism applies and the `Average` function is called.

8.3.2 Restrictions

There are two conditions to specialize a node by an other:

1. The specialized nodes must be imported.
2. The specializing nodes must be monomorphic (no type variable in its signature).

8.3.3 Using specialize

This feature is dedicated to imported polymorphic nodes/functions. It allows to provide different Scade or imported implementation to an imported polymorphic node/function.

9 / Examples

This chapter provides a set of complete examples that illustrate the language.

9.1 ABC

```
--
-- Button: manage a single button
--
-- in:
--   button: true, whenever button is pressed
--   lock: true, whenever lock is pressed
--   unlock: true, whenever unlock is pressed
--   deselect: deselect button
-- out:
--   preselLight: command of preselection light
--   selectLight: command of selection light
node Button(button,
                lock,
                unlock,
                deselect: bool)
  returns (preselLight,
           selectLight: bool)
sig
  sigLightPresel,
  sigLightSel;
let
  preselLight =  'sigLightPresel;
  selectLight =  'sigLightSel;

  automaton
    -- Unselect: if lock, another button is locked
    --            else waits for own preselection
    initial state Unselected
      unless
        if (lock) restart WaitUnlock;
        if (button) restart Preselected;

-- Preselect: wait for lock, or deselection by itself or other
    state Preselected
      unless
        if (lock) restart Locked;
        if (deselect) restart Unselected;
      let
        emit 'sigLightPresel;
      tel

    -- Locked state: wait for unlock only, and returns in Preselected
    state Locked
      unless if (unlock) restart Preselected;
      let
        emit 'sigLightSel;
      tel
...
```

(continued...)

```
    -- other lock: wait for unlock only, and returns in Unselected
    state WaitUnlock
      unless if (unlock) restart Unselected;

  returns .. ;

tel

node Raising(in, initVal: bool)
  returns (out: bool)
let
  out = initVal -> not(pre(in)) and in;
tel

node Falling(in, initVal: bool)
  returns (out: bool)
let
  out = initVal -> pre(in) and not(in);
tel

--
-- ABC module
--
node ABC(A, B, C, LOCK: bool)
  returns(A_PRESELECTED_ON, B_PRESELECTED_ON, C_PRESELECTED_ON,
          A_PRESELECTED_OFF, B_PRESELECTED_OFF, C_PRESELECTED_OFF,
          A_LOCKED_ON, B_LOCKED_ON, C_LOCKED_ON,
          A_LOCKED_OFF, B_LOCKED_OFF, C_LOCKED_OFF: bool)
sig
   lockSig,
   unlockSig;

var
  A_preselLight,
  B_preselLight,
  C_preselLight,
  A_selLight,
  B_selLight,
  C_selLight,
  deselect,
  localLock,
  localUnlock: bool;

let
...
```

(continued...)

```
  --
  -- Manage lock/unlock signal
  --
  localUnlock =  'unlockSig;
  localLock =  'lockSig;

  -- detect raising/falling edge
  automaton
    initial state LockLow
      unless if (LOCK) do {emit 'lockSig} restart LockHigh;

    state LockHigh
      unless if (LOCK) do {emit 'unlockSig} restart LockLow;
  returns .. ;

  --
  -- call buttons
  --
  A_preselLight, A_selLight  =
    Button(A, localLock, localUnlock, deselect);
  B_preselLight, B_selLight  =
    Button(B, localLock, localUnlock, deselect);
  C_preselLight, C_selLight =
    Button(C, localLock, localUnlock, deselect);

  --
  -- handle deselection
  --
  deselect = (A or B or C);

  ---
  ---
  A_PRESELECTED_ON = Raising(A_preselLight, false);
  A_PRESELECTED_OFF = Falling(A_preselLight, true);
  A_LOCKED_ON = Raising(A_selLight, false);
  A_LOCKED_OFF = Falling(A_selLight, true);

  B_PRESELECTED_ON = Raising(B_preselLight, false);
  B_PRESELECTED_OFF = Falling(B_preselLight, true);
  B_LOCKED_ON = Raising(B_selLight, false);
  B_LOCKED_OFF = Falling(B_selLight, true);

  C_PRESELECTED_ON = Raising(C_preselLight, false);
  C_PRESELECTED_OFF = Falling(C_preselLight, true);
  C_LOCKED_ON = Raising(C_selLight, false);
  C_LOCKED_OFF = Falling(C_selLight, true);
```

tel

9.2 Digital Stop Watch

```
------------------------- Watch Interface-----------------------
-- stst : start/stop button
-- rst : reset button
-- set : set time button
-- md : mode selection button
-- HH, MM, SS : time data display
-- L : is displaying lap time
-- S : is in setting time mode
-- Sh : is in setting hour mode
---------------------------------------------------------------------
node watch (stst, rst, set, md: bool)
  returns (HH, MM, SS : int8;
           L, S, Sh : bool default = false last = false)
var
  isStart : bool default = false; -- is chrono started?
  is_w : bool default = false;    -- is in clock mode?
  m, s, d : int8 last = 0;       -- chrono timers
  wh, wm, w, ws : int8;          -- clock timers
let
  w = 0 -> (pre w + 1) mod 100;
  ws = 0 -> (if w < pre w
             then pre ws + 1
             else pre ws       ) mod 60;
  automaton Stopwatch
    initial state Stop
    unless
      if stst and not is_w
        resume Start;
      if rst and not (false -> pre L) and not is_w
        restart Stop;
    m, s, d = (0, 0, 0) -> (last 'm, last 's, last 'd);
    state Start
    unless if stst and not  is_w resume Stop;
    let
      d = (last 'd + 1) mod 100;
      s = (if d < last 'd
           then last 's + 1
           else last 's     ) mod 60;
      m = if s < last 's then last 'm + 1 else last 'm;
      isStart = true;
    tel
  returns m, s, d, isStart;
  automaton Watch
    initial state Count
    let
      wm = 0 -> (if ws < last 'ws
                 then last 'wm + 1
                 else last 'wm     ) mod 60;
      wh = 0 -> (if wm < last 'wm
                 then last 'wh + 1
                 else last 'wh     ) mod 24;
...
```

(continued...)

```
    tel
    until if set and  is_w restart Set;
    state Set
    let
      S = true;
      automaton SetWatch
      initial state SetHours
      let
        Sh = true;
        wh = (if stst then last 'wh + 1
              else if rst then last 'wh + 23
              else last 'wh) mod 24;
      tel
      until if set and is_w restart SetMinutes;
      state SetMinutes
      wm = (if stst then last 'wm + 1
            else if rst then last 'wm + 59
            else last 'wm) mod 60;
      until if set and is_w restart SetEnd;
      final state SetEnd
      returns Sh, wh, wm;
    tel
    until synchro resume Count;
  returns S, Sh, wh, wm;
  automaton Display
    initial state DisplayWatch
    unless if md and not S resume DisplayStopwatch;
    HH, MM, SS, is_w = (wh, wm, ws, true);
    state DisplayStopwatch
    unless if md and not S resume DisplayWatch;
    var lm,ls,ld : int8 last = 0; -- chrono display
    let
      HH, MM, SS = (lm, ls, ld);
      automaton LapManagement
        initial state Stopwatch
        lm, ls, ld = (m, s, d);
        until if rst and  isStart restart Lap;
        state Lap
        L = true;
        until if rst restart Stopwatch;
      returns lm, ls, ld, L;
    tel
  returns HH, MM, SS, is_w, L;
tel
```

9.3 Cruise Control

```
type
  Percent = float64;
  Speed = float64;

const
  SpeedInc: Speed = 5.0 ;
  SpeedMax: Speed = 150.0 ;
  SpeedMin: Speed = 30.0 ;
  Kp: float64 = 10.0 ;
  Ki: float64 = 0.2 ;

node CruiseControl (On: bool; Off: bool; Resume: bool;
                    CurSpeed: Speed; Set: bool; QuickAccel: bool;
                    QuickDecel: bool; Accel: Percent; Brake: Percent)
     returns (CruiseSpeed: Speed last = 0.0; ThrottleCmd: Percent)
var
  accelerator: bool ;
  brake: bool ;
let
   brake = Brake > 0.0;
   accelerator = Accel > 0.0;
   automaton
   initial state _Off
   let
     ThrottleCmd = Accel;
   tel
   until if (On) restart _On;
   state _On
   let
     automaton
     initial state _Regulation
     var
       between: bool ;
     let
       CruiseSpeed = CruiseSpeedMgt(On, Set, QuickAccel, QuickDecel, CurSpeed);
       between = CurSpeed >= SpeedMin and CurSpeed <= SpeedMax;
       automaton
       initial state _RegulOn
       let
         ThrottleCmd = Regulator(CruiseSpeed , CurSpeed);
       tel
       until if (accelerator or not between) restart _StandBy;
       state _StandBy
       let
         ThrottleCmd = Accel ;
       tel
       until if (not accelerator and between) restart _RegulOn;
       returns .. ;
...
```

(continued...)

```
     tel
     until if (brake) restart _Interrupt;
     state _Interrupt
     let
       ThrottleCmd = Accel;
     tel
     until if (Resume) restart _Regulation;
     returns ..;
   tel
   until if (Off) restart _Off;
   returns .. ;
tel
node Regulator (CruiseSpeed, CurSpeed: Speed) returns (Throttle: Percent)
var
  delta, aux: Percent;
let
  delta = CruiseSpeed - CurSpeed;
  Throttle = delta * Kp + aux * Ki;
  aux = delta + (0.0 -> pre aux);
tel
node CruiseSpeedMgt(On: bool; Set: bool;
        QuickAccel: bool; QuickDecel: bool; CurSpeed: Speed)
        returns (CruiseSpeed: Speed)
var
  moreSpeed: float64;
  lessSpeed: float64;
  preCruiseSpeed: float64;
let
  moreSpeed = preCruiseSpeed + SpeedInc;
  lessSpeed = preCruiseSpeed - SpeedInc;
  preCruiseSpeed = 0.0 -> pre CruiseSpeed;
  CruiseSpeed = if QuickDecel and (lessSpeed >= SpeedMin)
                then lessSpeed
                else
                  if QuickAccel and (moreSpeed <= SpeedMax)
                  then moreSpeed
                  else
                    if (On or Set)
                       and (CurSpeed <= SpeedMax)
                       and (CurSpeed >= SpeedMin)
                    then CurSpeed
                    else preCruiseSpeed;
tel
```

9.4 Operations on Matrices

```
const
  n = 10;
  m = 12;
  p = 15;

-- function to iter to compute a scalar product
-- a : partial sum, from 0 to i-1 (fold accumulator)
-- v : nth element of the first vector
-- w : nth element of the second vector
function prod_sum (a, v, w: float64) returns (aa: float64)
let
  aa = a + v*w;
tel

-- scalar product of two vectors: V . W
function ScalProd (V, W: float64^n) returns (sp: float64)
let
  sp = (fold prod_sum <<n>>) (0.0, V, W);
tel

-- product of a matrix by a vector: A * u
function MatVectProd (A: float64^m^n; u: float64^n) returns (w: float64^m)
let
  w = (map ScalProd <<m>>)(transpose (A; 1; 2), u^m);
tel

-- matrix product: A * B
-- (m,n) x (n,p)
function MatProd (A: float64^m^n; B: float64^n^p) returns (C: float64^m^p)
let
  C = (map MatVectProd <<p>>)(A^p, B);
tel


function rootMatProd (A: float64^m^n; B: float64^n^p) returns (C: float64^m^p)
let
  C = MatProd (A, B);
tel
```

9.5 Extended ABC

```
type bk_color = enum {grey, yellow, green};
type fr_color = enum {black, white};
node Button (button, lock, unlock, other: bool)
  returns (background : bk_color default = grey;
           foreground : fr_color default = white)
let
  automaton
    initial state Unselected
    unless
      if lock restart LockedUnselection ;
      if button restart Preselected;
    state Preselected
    unless
      if lock restart LockedSelection;
      if button or other restart Unselected;
    background = yellow;
    state LockedSelection
    unless if unlock restart Preselected;
    background = green;
    state LockedUnselection
    unless if unlock restart Unselected;
    foreground = black;
  returns background, foreground;
tel
const n: int16 = 8; -- number of buttons
node TwoStepsSelect(Lock: bool; buttons: bool^n)
  returns (bk_buttons : bk_color^n;
           fg_buttons : fr_color^n;
           LockLight : bool default = false)
sig lockSig, unlockSig;
var buttonPressed : bool;
let
  automaton LockManagement
    initial state LockLow
      unless
        if Lock do {emit 'lockSig} restart LockHigh;
    state LockHigh
      unless
        if Lock do {emit 'unlockSig} restart LockLow;
      LockLight = true;
  returns LockLight;
  bk_buttons, fg_buttons = -- instantiate buttons
    (map Button <<n>>)
      (buttons, 'lockSig^n, 'unlockSig^n, buttonPressed^n);
  buttonPressed =          -- exists one pressed button?
    (fold $or$ <<n>>) (false, buttons);
tel
```

9.6 FIFO

```
const
   -- FIFO size: it should be a package parameter
   SIZE: int32 = 4;

type
  -- FIFO element type: it should be a package parameter
  TYPE = int32;

  TAB = TYPE^SIZE;

  STATUS = enum {OK, UNDERFLOW, OVERFLOW, EMPTY, FULL};

const
  DEFAULT_VAL: TYPE = -10 ;
  DEFAULT_MEM: TYPE^SIZE = DEFAULT_VAL^SIZE;

-------------------------------------------------------
-- FIFO:
-------------------------------------------------------

node Fifo (read, write: bool; push_val: TYPE)
         returns (val: TYPE; st: STATUS)
sig
  bypass;
var
  memory: TAB last = DEFAULT_MEM;
  addr: int32 last = 0;
  size: int32 last = 0;

let

  val = if 'bypass then push_val
        else (last 'memory.[last 'addr] default DEFAULT_VAL);

  automaton
    initial state Empty
    unless
      if (write) if (read) do {'bypass} resume Empty
                 else resume Half end;
    let
      st = if (read and not write) then UNDERFLOW else EMPTY;
    tel

    state Half
    var
      addr_write: int32;
...
```

```
(continued...)

    let
       st = if size = 0 then EMPTY else if size = SIZE then FULL else OK;

       size = if (write and not read) then last 'size + 1
              else if (read and not write) then last 'size - 1
                else last 'size;

       addr       = if read then (last 'addr + 1) mod SIZE else last 'addr;
       addr_write = (last 'addr + last 'size) mod SIZE;
       memory     = if write then (last 'memory with [addr_write] = push_val)
                    else last 'memory;

    tel
    until
      if (size = SIZE) resume Full;
      if (size = 0) resume Empty;


    state Full
    unless
      if (read and not write) resume Half;
    let
      st =  if (write and not read) then OVERFLOW else FULL;

      addr   = if write and read then (last 'addr + 1) mod SIZE else last 'addr;
      memory = if write and read then (last 'memory with [last 'addr] = push_val)
               else last 'memory;


    tel
  returns .. ;
tel

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

node testFifo (read, write: bool; q: int32) returns (val_out, status: int32)
var
 st: STATUS;
let

  val_out , st = Fifo(read, write, q);

  status = if st = OK then 0
           else if st = EMPTY then 1
           else if st = FULL then 2
           else if st = UNDERFLOW then -10
           else if st = OVERFLOW then -20
           else -100 ;
tel
```

Index

Index

Index

Index

Index

Index