Lecture 2

The SCADE Language – Data Flow Kernel

Daniel Kästner
AbsInt GmbH

2012

Synchronous Programming

- Two simple ways of implementing reactive systems:
  - Event-driven

```
<Initialize Memory>
Foreach input_event do
        <Compute Outputs>
        <Update Memory>
End
```

  - Sampling

```
<Initialize Memory>
Foreach period do
        <Read Inputs>
        <Compute Outputs>
        <Update Memory>
End
```

Synchronous Programming

- Program typically implements an automaton:
  - state: contents of memory cells
  - transition: reaction, possibly involving many computations
- Synchronous hypothesis: reactions are considered atomic, i.e. they take no time. Computational steps execute like combinatorial circuits.
- Synchronous broadcast: instantaneous communication, i.e. each automaton in the system considers the outputs of others as being part of its own inputs.
- Three-phase cyclic execution model (cyclic evolution):
  1. Read input signals (input event)
  2. Compute the reaction
  3. Update outputs (output event)

Synchronous Programming

- Time is divided into discrete ticks (also called cycles, steps, instants).
- Implicit assumption: presence of a global clock. This makes application in distributed environments difficult.
- Requirements:
  - Causality: The output at any instant *t* may only depend upon input received before *t* or at *t*.
  - Behavior must be deterministic.
  - Execution time predictability: no unbounded loops, no recursion, safe upper bounds on worst-case execution time (WCET) available.
- In order to validate the timing behavior it is sufficient to prove that the worst-case execution time (WCET) of any reaction is smaller than the time interval between two execution cycles.

Synchronous Data Flow Programming

- Functional programming model: no side effects
- Inputs and outputs are flows (or streams) of values, that are infinite sequences of values of a given type.
- Let v be the expression of a flow. Then its sequence of values is denoted v_1, v_2, v_3, …
- Output flows are defined by equations in the mathematical sense; there is no notion of assignment in the imperative sense.
- Only one (active) definition per flow is allowed.
- Equation `v=expr` introduces the name `v` for the flow defined by `expr`. Any use of `v` can be safely replaced by `expr`, according to the substitution principle.

Language Overview

- StateCharts:
  - First popular formal language for the design of reactive systems.
  - Focus on specification and design, not designed as a programming language.
  - Determinism is not ensured.
  - No standardized semantics.
- Synchronous programming languages
  - ESTEREL [Berry]: textual imperative language.
  - SyncCharts: Graphical formalism for ESTEREL.
  - LUSTRE [Caspi, Halbwachs]: textual declarative language. Tailored to data-flow oriented systems (e.g. regulation systems).
  - SCADE [Esterel Inc.]. Textual/graphical programming language incorporating concepts of ESTEREL and LUSTRE.
  - SSM (Safe State Machines): Successor of SyncCharts, used in SCADE.

References

- *Scade Language Reference Manual*, Esterel Technologies SA, 2011.

- *Scade Language Primer*, Esterel Technologies SA, 2011.

- Available from SCADE Suite GUI (Help.PDF Documentation).

The SCADE Language

- Based on synchronous data-flow paradigm.
  - Data flow kernel (extended LUSTRE dialect)
  - State Machines (SSM)
- Supports both textual and graphical programming.
- SCADE compiler performs five static semantic checks:
  - namespace analysis: any object must have exactly one definition in the current context
  - type analysis: every flow and operator is used consistently with its data type
  - clock analysis: clocks of flows and operators must be well-defined
  - causality analysis: no flow instantaneously depends on itself
  - initialization analysis: There must be no uninitialized expressions (yielding nil). This ensures determinism: a model must produce the same outputs given the same inputs

Flows and Clocks

- Synchronous systems induce a discrete and logical notion of time. They can be described as the parallel composition of several processes.
- Composing processes computed on different rates is based on clocks.
- A clock defines the instants where a flow is available:
  - If the clock is *true*, the flow has the value resulting from its definition.
  - If the clock is *true* but the value of the flow is unspecified, its value is denoted *nil*.
  - If the clock is *false* the flow has no value at all.
- The logical time defines the fastest possible rate in the system, the base clock. The base clock is the clock that is always true; a stream on the base clock is present at each cycle.
- All other rates, or clocks, are derived from the base clock.

Flows and Clocks

- Any variable and expression denotes a flow, ie a pair (x, b_x) made of
  - a possibly infinite sequence x of values of a given type
  - a clock b_x, representing a sequence of times.
- x is defined at instant i iff $b_x(i) = true$.
- A flow takes its n-th value in the n-th time of its clock.
- Example: Let x run on the base clock C, y on a slower clock. This gives the following time scales:

Basic time-scale	1	2	3	4	5	6	7	8
b_x=C	t	t	t	t	t	t	t	t
x time-scale	1	2	3	4	5	6	7	8
b_y	t	f	t	f	t	f	t	f
y time-scale	1		2		3		4	

Nodes and Functions

- A SCADE program describes the data flow relation between the inputs and the outputs of a system.
- Relations are expressed using operators, auxiliary variables, and constants.
- Operators can either be temporal primitives, data primitives, or user-defined operators.
- A user-defined operator may be declared as a function or a node.
  - A function defines an operator without internal state, i.e. no expression within the operator recursively refers to past instants. This information can be used by the compiler for code optimization.
  - A node defines an operator with an internal state.

Nodes and Functions

- Syntax:

```
node|function name params returns params opt_body
params ::= ([<var_decls>(; <var_decls>)*])
opt_body ::= ; | equation |
             <signal_block> <local_block> let equation; tel
```

- A node is the basic structuring unit of SCADE programs.
- A node/function is defined by its name, its interface (list of inputs, list of outputs, along with their types), together with a definition of these outputs.
- An output flow is either defined by means of an equation, or by a State Machine.
- Local flows can be defined in order to simplify the definition of the output flows.
- Each operator responds to its inputs for every arrival of these inputs.

Nodes and Functions

- Syntax:

```
node|function name params returns params opt_body
params ::= ([<var_decls>(; <var_decls>)*])
opt_body ::= ; | equation |
             <signal_block> <local_block> let equation; tel
```

- The body can be empty, a single equation, or a set of equations, possibly preceded by local variable/signal declarations.
- Local variables are declared in `<local_block>`, introduced by the keyword var.
- Signals are local variables introduced by the keyword *sig*. They are declared in `<signal_block>`.

Nodes and Functions

- Syntax:

```
node|function name params returns params opt_body
params ::= ([<var_decls>(; <var_decls>)*])
opt_body ::= ; | equation |
             <signal_block> <local_block> let equation; tel
```

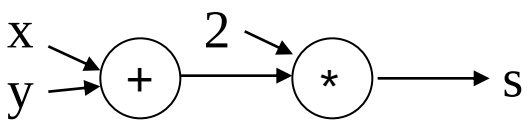
- Each output and local variable must appear exactly once in the left part of an equation.
- Every input variable and local variable must be used at least once in these equations.
- Signals must be emitted and caught at least once.

Nodes and Functions

- Example:

```
node sum (x,y: int) returns (s: int)
let
  s = 2*(x+y);
tel
```

```
function sum (x,y: int) returns (s: int)
let
  s = 2*(x+y);
tel
```



At any cycle n:
$s_n = 2*(x_n+y_n)$

Variable Declarations

- Syntax:

```
[clock] <var_id> (,<var_id>)* : <type_expr>
[<when_decl>][<default_decl>][<last_decl>]
var_id ::= Identifier
when_decl ::= when clock_expr
default_decl ::= default = expr
last_decl ::= last = expr
```

- A variable declaration binds a new name to a flow.
- Several identifiers can be declared at the same time.
- Variables can be declared
  - in a node declaration (input and output flows)
  - in scope declarations (local flows)
- The clock attribute declares that this identifier can be used to sample other flows. The identifier must be of finite enumerated type including booleans.

Variable Declarations

- Syntax:

```
[clock] <var_id> (,<var_id>)* : <type_expr>
[<when_decl>][<default_decl>][<last_decl>]
var_id ::= Identifier
when_decl ::= when clock_expr
default_decl ::= default = expr
last_decl ::= last = expr
```

- When a flow is unspecified at a given instant, the default behavior is to maintain the value it had at the previous cycle. At the first instant of activation this previous value does not exist (*nil*).
- If a last expression is given, it is only evaluated at the first instant of activation to provide a specified starting value to the flow.
- If a default expression is given, the last expression is not evaluated. At any cycle where the flow is undefined, the default expression is evaluated instead.

Variable Declarations

- Syntax:

```
[clock] <var_id> (,<var_id>)* : <type_expr>
[<when_decl>][<default_decl>][<last_decl>]
var_id ::= Identifier
when_decl ::= when clock_expr
default_decl ::= default = expr
last_decl ::= last = expr
```

- The clocking expression `when_decl` states that the identifiers in the list are based on a different clock than the base clock of the operator.
- A valid clock expression must be used, in particular an identifier cannot be sampled on itself.
- Examples:

```
(x: bool; y: int when x)  -- as parameter declaration
var x:bool; y:int;        -- as local flow declaration
```

Line comments

Types

- Basic types: bool, int, real, char.
- Type declarations:

```
type <type_decl>;
type_decl ::= type_expr | enum { ID (,ID)* }
type_expr ::= bool | int | real | char |
                        | { field_decl (, field_decl)* }
                        | type_expr ^ expr
field_decl ::= id : type_expr
```

- Identifiers in an enumeration type must be constants.
- The type expression must not contain any type variable (identifier preceded by a quote).
- A structured type is given by a non-empty list of field declarations. Each field is given a label, which must be different from other labels of the current type, and a type.
- Structures can be nested, but not recursively refer to themselves.

Types

- Basic types: bool, int, real, char.
- Type declarations:

```
type <type_decl>;
type_decl ::= type_expr | enum { ID (,ID)* }
type_expr ::= bool | int | real | char |
                        | { field_decl (, field_decl)* }
                        | type_expr ^ expr
field_decl ::= id : type_expr
```

- Array data types are declared using any type expression followed by ^ and a constant integer size expression.
- The size expression can contain integer values, declared internal constants, arithmetic operators +, -, *, div, mod.
- type T = int;
  type Tenum = enum {red, blue, pink, purple};
  type Vect = int^3;

Equations

- Simple equations:
  - Variables are defined via equations, e.g. X=E with variable X and expression E.
  - Substitution principle: X can be substituted to E anywhere in the program and vice versa.
  - Definition principle: The behavior of X must be completely specified by this equation.
- Asserts
- Signal emissions
- Control blocks: either conditional blocks or automata

Variables and Expressions

- Operators only operate on operands sharing the same clock.
- As variables and expressions are streams, operators also produce streams. Example: With x = (0,1,2,3,4,...) and y = (2,4,6,8,10,...): x+y=(2,5,8,11,14,...)
- Expressions are built from constants, variables, and operators.
- Types of operators:
  - sequential (temporal) operators
  - combinatorial operators:
    - arithmetic, relational, and boolean expressions
    - conditional expressions
  - operations on arrays and structures
  - instantiation of user-defined operators (nodes, functions)
  - higher-order operators
- Temporal operators are polymorphic and can be applied to tuples (lists).

Temporal Operators

- `temp_expr ::= pre expr | ...`
- 'previous' operator `pre` defines a one-step delay:
  - $(pre(E))_1 = nil$ (undefined)
  - $n > 1$: $(pre(E))_n = E_{n-1}$
- Using this operator in an expression introduces a memory, i.e. it cannot be contained in a `function`.

Basic time-scale	1	2	3	4	5	6	7	8
b_X	true	true	true	true	true	true	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
pre (X)								

Temporal Operators

- `temp_expr ::= pre expr | ...`
- 'previous' operator `pre` defines a one-step delay:
  - $(pre(E))_1 = nil$ (undefined)
  - $n > 1$: $(pre(E))_n = E_{n-1}$
- Using this operator in an expression introduces a memory, i.e. it cannot be contained in a `function`.

Basic time-scale	1	2	3	4	5	6	7	8
b_X	true	true	true	true	true	true	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
pre (X)	*nil*	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Temporal Operators

- `temp_expr ::= ... | expr -> expr | ...`
- 'init' operator `->`
  - $(E \rightarrow F)_1 = E_1$
  - $n > 1$: $(E \rightarrow F)_n = F_n$
- Builds flow whose first value is the one of its first argument, the remaining values being given by the second argument.
- Both arguments must run on the same clock.

b_{XY}	true	true	true	true	true	true	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X -> Y								

Temporal Operators

- `temp_expr ::= ... | expr -> expr | ...`
- 'init' operator `->`
  - $(E \rightarrow F)_1 = E_1$
  - $n > 1$: $(E \rightarrow F)_n = F_n$
- Builds flow whose first value is the one of its first argument, the remaining values being given by the second argument.
- Both arguments must run on the same clock.

b_{XY}	true	true	true	true	true	true	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X -> Y	x_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8

Temporal Operators

- Combining the `pre` with the `->` operator permits to define delayed flows with an well-defined value at the first instant.

b_{XY}	true	true	true	true	true	true	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
pre (Y)	*nil*	y_1	y_2	y_3	y_4	y_5	y_6	y_7
X -> pre(Y)								

Temporal Operators

- Combining the `pre` with the `->` operator permits to define delayed flows with an well-defined value at the first instant.

b_{XY}	true	true	true	true	true	true	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
pre (Y)	*nil*	y_1	y_2	y_3	y_4	y_5	y_6	y_7
X -> pre(Y)	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7

Temporal Operators

- `temp_expr ::= ... | fby (list; expr; list) | ...`
- 'followed by' operator `fby`
  - $m \in \{1, \ldots, n\}$: $fby\,(F; n; E)_m = E_1$
  - $m > n$: $fby\,(F; n; E)_m = F_{m-n}$
- $n > 1$: $fby(F; n; E) \cong pre\big(fby(F; n-1; E)\big)$;
- should rather be called 'preceded by'

X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X -> pre(Y)	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7
fby (Y; 1; X)								
fby (Y; 2; X)								

Temporal Operators

- `temp_expr ::= ... | fby (list; expr; list) | ...`
- 'followed by' operator `fby`
  - $m \in \{1, \ldots, n\}$: $fby\,(F; n; E)_m = E_1$
  - $m > n$: $fby\,(F; n; E)_m = F_{m-n}$
- $n > 1$: $fby(F; n; E) \cong pre\big(fby(F; n-1; E)\big)$;
- should rather be called 'preceded by'

X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X -> pre(Y)	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7
fby (Y; 1; X)	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7
fby (Y; 2; X)								

Temporal Operators

- `temp_expr ::= ... | fby (list; expr; list) | ...`
- 'followed by' operator `fby`
  - $m \in \{1, \ldots, n\}$: $fby\,(F; n; E)_m = E_1$
  - $m > n$: $fby\,(F; n; E)_m = F_{m-n}$
- $n > 1$: $fby(F; n; E) \cong pre\big(fby(F; n-1; E)\big)$;
- should rather be called 'preceded by'

X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X -> pre(Y)	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7
fby (Y; 1; X)	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7
fby (Y; 2; X)	x_1	x_1	y_1	y_2	y_3	y_4	y_5	y_6

Temporal Operators

- `temp_expr ::= ... | expr when clock_expr | ...`
- (Down-)Sampling: when
  - Let E be an expression and B a boolean expression with the same clock: (E when B) is the sequence of values of E when B is true.
- The clock expression E gives the instants where (E when B) is defined.

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y = X when B		x_2		x_4			x_7	

Note: `gaps´ are not filled.

Temporal Operators

- `temp_expr ::= … | merge (clock_expr; expr (;expr)* )`
- Given flows based on complementary clocks, the merge operator combines them to produce a faster clock.
- Note that the first operand has to be a clock identifier.

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X when B								
Y when not B								
merge (B; X when B; Y when not B)								

Temporal Operators

- `temp_expr ::= … | merge (clock_expr; expr (;expr)* )`
- Given flows based on complementary clocks, the merge operator combines them to produce a faster clock.
- Note that the first operand has to be a clock identifier.

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X when B		x_2		x_4			x_7	x_8
Y when not B								
merge (B; X when B; Y when not B)								

Temporal Operators

- `temp_expr ::= … | merge (clock_expr; expr (;expr)* )`
- Given flows based on complementary clocks, the merge operator combines them to produce a faster clock.
- Note that the first operand has to be a clock identifier.

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X when B		x_2		x_4			x_7	x_8
Y when not B	y_1		y_3		y_5	y_6		
merge (B; X when B; Y when not B)								

Temporal Operators

- `temp_expr ::= … | merge (clock_expr; expr (;expr)* )`
- Given flows based on complementary clocks, the merge operator combines them to produce a faster clock.
- Note that the first operand has to be a clock identifier.

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X when B		x_2		x_4			x_7	x_8
Y when not B	y_1		y_3		y_5	y_6		
merge (B; X when B; Y when not B)	y_1	x_2	y_3	x_4	y_5	y_6	x_7	x_8

Temporal Operators – Overview

- temp_expr ::= **pre** expr
  | expr **->** expr
  | **fby** (list; expr; list)
  | expr **when** clock_expr
  | **merge** (expr; list (;list)*)

Combinatorial Operators

- Variants:
  - arithmetic operators
  - relational operators
  - boolean operators
  - flow switches
- Combinatorial operators operate on flows.
- They do not introduce implicit memory, and do not affect the clock of their argument.

Basic time-scale	1	2	3	4	5	6	7	8
X	1	3	5	7	9	11	13	15
Y	2	4	6	8	10	12	14	16
X+Y								

Combinatorial Operators

- Variants:
  - arithmetic operators
  - relational operators
  - boolean operators
  - flow switches
- Combinatorial operators operate on flows.
- They do not introduce implicit memory, and do not affect the clock of their argument.

Basic time-scale	1	2	3	4	5	6	7	8
X	1	3	5	7	9	11	13	15
Y	2	4	6	8	10	12	14	16
X+Y	3	7	11	15	19	23	27	31

Arithmetic Operators

```
arith_expr       ::= unary_arith_op expr
                   | expr bin_arith_op expr
unary_arith_op   ::= - | + | int | real
binary_arith_op  ::= - | + | * | / | mod | div
```

- Input flows must be integers or reals.
- Most operators accept polymorphic arguments (int or real). Only mod requires two integer expressions as inputs.
- The operators int and real are cast operators, producing integer or real values.
- The operators div and mod (integer division/modulo) produce an integer result.
- The operator / (real division) produces a real result.

Relational Operators

- `rel_expr ::= expr rel_op expr`
  `rel_op ::= = | <> | < | > | <= | >=`

- Input flows for = and <> operators must be of primitive type.
- The other relational operators require their arguments to be of numeric type (int or real).

X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
X+Y	x_1+y_1	x_2+y_2	x_3+y_3	x_4+y_4	x_5+y_5	x_6+y_6	x_7+y_7	x_8+y_8
X = Y	$x_1=y_1$	$x_2=y_2$	$x_3=y_3$	$x_4=y_4$	$x_5=y_5$	$x_6=y_6$	$x_7=y_7$	$x_8=y_8$

Boolean Operators

- ```
  bool_expr       ::= not expr
                    | expr bin_bool_op expr
                    | # ( list )
  bin_bool_op  ::= and | or | xor
  ```
- Input flows for boolean operators are Boolean flows.
- The output of the # ('exclusive') operator is true when at most one if its input its true.

A	true	false	true	true	false	true	false	true
B	false	false	true	true	true	false	true	false
A and B	false	false	true	true	false	false	false	false
A xor B	true	false	false	false	true	true	true	true


  ```

Flow Switches

- ```
  switch_expr    ::= if expr then expr else expr
                   | ( case expr of (case_expr)+ )
  case_expr      ::= | pattern : expr
  pattern        ::= path_id | CONST | _
  ```

- if-then-else operator:
  - The first argument is a single Boolean expression.
  - The other two can be any expression of any type. They must share the same type.
  - All arguments have to be based on the same clock.

- Y = if (X<0) then -X else X;


  ```

Flow Switches

```
switch_expr    ::= if expr then expr else expr
                 | ( case expr of (case_expr)+ )
case_expr      ::= | pattern : expr
pattern        ::= path_id | CONST | _
```

- case-operator:
  - The first argument of the case operator must be a declared enumeration value, an integer or Boolean constant, or a character.
  - Patterns have to be of the same type as the first argument, and must all be different from one another.
  - Expressions can have any type. All expressions must share the same type. Patterns must be exhaustive.
  - The default pattern _ matches all patterns that are not explicitly specified.
  - All arguments have to be based on the same clock.

Flow Switches

- ```
  switch_expr    ::= if expr then expr else expr
                   | ( case expr of (case_expr)+ )
  case_expr      ::= | pattern : expr
  pattern        ::= path_id | CONST | _
  ```

- case-operator:

- Example:

```
cval = (case (color) of
        | red: 0
        | blue: 1
        | green: 2);
```


```

Example

- Compute the minimum and maximum of an input sequence, starting from value 0.

```
node minmax (in : int)
  returns (min,max : int)
let


tel
```

base clock	1	2	3	4	5	6	7	8
in	0	1	10	-10	50	-50	3	43
min								
max								

Example

- Compute the minimum and maximum of an input sequence, starting from value 0.

```
node minmax (in : int)
  returns (min,max : int)
let
  min = 0-> if in < pre(min) then in else pre(min);

tel
```

base clock	1	2	3	4	5	6	7	8
in	0	1	10	-10	50	-50	3	43
min								
max								

Example

- Compute the minimum and maximum of an input sequence, starting from value 0.

```
node minmax (in : int)
  returns (min,max : int)
let
  min = 0-> if in < pre(min) then in else pre(min);
  max = 0-> if in > pre(max) then in else pre(max);
tel
```

base clock	1	2	3	4	5	6	7	8
in	0	1	10	-10	50	-50	3	43
min								
max								

Example

- Compute the minimum and maximum of an input sequence, starting from value 0.

```
node minmax (in : int)
  returns (min,max : int)
let
  min = 0-> if in < pre(min) then in else pre(min);
  max = 0-> if in > pre(max) then in else pre(max);
tel
```

base clock	1	2	3	4	5	6	7	8
in	0	1	10	-10	50	-50	3	43
min	0	0	0	-10	-10	-50	-50	-50
max	0	1	10	10	50	50	50	50

Operator Precedence

- Precedence of primitive operators in decreasing order:
  - not
  - when
  - int, real
  - pre
  - unary +, -
  - *, /, mod, div
  - binary +, -
  - =, <, >, <=, >=, <>
  - and
  - or, xor
  - ->
  - if
- Hint: Use ( ) to clarify operator precedence.

Question

- What happens here?

```
node minmax (in : int)
  returns (min,max : int)
let
  min = if in < 0->pre(min) then in else 0->pre(min);
  max = if in > 0->pre(max) then in else 0->pre(max);
tel
```

Operator Precedence

- What happens here?

```
node minmax (in : int)
  returns (min,max : int)
let
  min = if in < 0->pre(min) then in else 0->pre(min);
  max = if in > 0->pre(max) then in else 0->pre(max);
tel
```

- `Semantics error: This expression has type bool but should have type int.`
- Refers to: in < 0->pre(min)
  which is equivalent to: (in<0) -> pre(min)
- The first operand of the -> operator is of type bool, the second one of type int, but they have to share the same type.

Instantiation of User-Defined Operators

- Let N be a user-defined operator (ie a node or a function) of type 'a → 'a and let e be any expression of type 'a. Then N(e) is an operator instantiation.
- Semantics is given by operator expansion: An operator instantiation is equivalent to replacing the operator identifier by its definition and the formal arguments by the expressions passed as arguments.

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

node N2 (a: real) returns (b,c,d,e: real)
let
  b,c = N1(a, a->pre a);                    -- N1 instantiated
  d,e = (a + (a->pre a), a * (a->pre a));   -- N1 expanded
tel
```

Instantiation of User-Defined Operators

- **restart** op **every** expr (exp_1, … exp_n)
- The restart operator takes as input an operator op, a Boolean expression and a list of actual parameters of op. It is equivalent to calling the instantiation op(exp_1, … exp_n) at each cycle, and performing a reset of op when expr is true.

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

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

node resettable_count2 (r: bool) returns (c:int)
let
  c = (restart count every r)();
tel
```

Instantiation of User-Defined Operators

- The `restart` operator causes flow initializations (`->` operator) in the instantiated node to return their first arguments as in the first cycle.
- When a node instantiation is reset, all node instantiations in its definition are automatically reset as well.

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

node double () returns (s: int)
let
  s = 2 * count();
tel

node sample (b1,b2: bool) returns (s1,s2: int)
let
  s1 = (restart count every b1)();
  s2 = (restart double every b2)();
tel
```

Instantiation of User-Defined Operators

- Example:

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

node double () returns (s: int)
let  s = 2 * count(); tel

node sample (b1,b2: bool) returns (s1,s2: int)
let
  s1 = (restart count every b1)();
  s2 = (restart double every b2)();
tel
```

b1	false	false	true	false	false	false	true	false
b2	false	false	false	false	true	false	true	false
s1								
s2								

Instantiation of User-Defined Operators

- Example:

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

node double () returns (s: int)
let  s = 2 * count(); tel

node sample (b1,b2: bool) returns (s1,s2: int)
let
  s1 = (restart count every b1)();
  s2 = (restart double every b2)();
tel
```

b1	false	false	true	false	false	false	true	false
b2	false	false	false	false	true	false	true	false
s1	0	1	0	1	2	3	0	1
s2	0	2	4	6	0	2	0	2

Clocks and Nodes

- When no clock is declared in the node interface, all inputs and outputs must share the same clock. The node is executed when all its inputs are available.
- When a node N has no input it is possible to clock any of its instances using the notation N(() when h).
- Clocks can be declared in a node signature under the following 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
- When clocks are declared in the node interface, the clock of a node instance is the clock of its fastest input.

Clocks and Nodes

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

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

e	1	2	3	4	5	6	7	8
h	false	true	false	false	true	true	false	true
e when h								
integr (e when h)								
s								
t								

Clocks and Nodes

```
node integr (e:int) returns (x:int)
  x = e + (0 -> pre x);

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

e	1	2	3	4	5	6	7	8
h	false	true	false	false	true	true	false	true
e when h		2			5	6		8
integr (e when h)								
s								
t								

Clocks and Nodes

```
node integr (e:int) returns (x:int)
  x = e + (0 -> pre x);

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

e	1	2	3	4	5	6	7	8
h	false	true	false	false	true	true	false	true
e when h		2			5	6		8
integr (e when h)		2			7	13		21
s								
t								

Clocks and Nodes

```
node integr (e:int) returns (x:int)
  x = e + (0 -> pre x);

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

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

Clocks and Nodes

```
node integr (e:int) returns (x:int)
  x = e + (0 -> pre x);

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

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

Node Activation with Output Memorization

- Using clocks is a way to control the instants when a node instance is evaluated, but explicit control of an instance execution typically is more intuitive.
- $y_1, \ldots, y_p$ **= (activate** N **every** c **initial default(**$i_1, \ldots, i_p$**))** (e_1, …, e_n)
- The parameters are the name of an instantiated node N, a Boolean c defining the instance clock, the inputs of the instance e_k, and the values i_k returned for the cycles until c becomes true for the first time.
- After c has become true for the first time the output flow maintains its last value as long as c is false.
- Let ck be the Boolean clock defined by the equation h=c. Then:

```
y1, …, yp = (activate N every c initial default(i1, …, ip)) (e1, …, en)
≡
y1, …, yp = merge (c; N((e1, …, en) when c);
                       ((i1, …, ip) -> pre (y1, …, yp)) when not c);
```

Node Activation with Output Memorization

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

node two_instances2 (e: int; h: bool) returns (s,t: int)
let
  s = integr(e);
  t = (activate integr every h initial default 0)(e);
tel
```

e	1	2	3	4	5	6	7	8
h	false	true	false	false	true	true	false	true
s	1	3	6	10	15	21	28	36
t								

Node Activation with Output Memorization

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

node two_instances2 (e: int; h: bool) returns (s,t: int)
let
  s = integr(e);
  t = (activate integr every h initial default 0)(e);
tel
```

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

Node Activation with Default Flows

- The variant

  $y_1,\ldots,y_p$ **= (activate** N **every** c **default(**$d_1,\ldots,d_p$**))**
  (e_1**,** …**,** e_n**)**

  provides a default value d_i for the output flow every time the condition c is *false*, including the initial instant.
- Let ck be the Boolean clock defined by the equation h=c. Then the semantics is given by:

```
y1, …, yp = (activate N every c default(d1, …, dp)) (e1, …, en)
≡
y1, …, yp = merge (c; N((e1, …, en) when c);
                        (d1, …, dp) when not c);
```

Node Activation with Default Flows

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

node two_instances3 (e: int; h: bool) returns (s,t: int)
let
  s = integr(e);
  t = (activate integr every h default 0)(e);
tel
```

e	1	2	3	4	5	6	7	8
h	false	true	false	false	true	true	false	true
s	1	3	6	10	15	21	28	36
t								

Node Activation with Default Flows

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

node two_instances3 (e: int; h: bool) returns (s,t: int)
let
  s = integr(e);
  t = (activate integr every h default 0)(e);
tel
```

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

Simple Node Activation

- The simple node activation does not produce a value at each cycle but only when the instance is activated.

  $y_1,\ldots,y_p$ **= (activate** N **every** c**)(**e_1**,** …**,** e_n**)**

- This is equivalent to

  $y_1,\ldots,y_p$ **=** N**((**e_1**,** …**,** e_n**) when** c**);**

- What happens here?

```
node two_instances4 (e: int; h: bool) returns (s,t: int)
let
  s = integr(e);
  t = (activate integr every h)(e);
tel
```

Simple Node Activation

- The simple node activation does not produce a value at each cycle but only when the instance is activated.

  $y_1,\ldots,y_p$ **= (activate** N **every** c**)(**e_1**, …,** e_n**)**

- This is equivalent to

  $y_1,\ldots,y_p$ **=** N**((**e_1**, …,** e_n**) when** c**);**

- What happens here?

```
node two_instances4 (e: int; h: bool) returns (s,t: int)
let
  s = integr(e);
  t = (activate integr every h)(e);
tel
```

  - A semantic error is raised since the clocks of input and output flow have to be the same (no clock on any input), but t is on clock h.

Combining Resets and Clocks

- Resets do not introduce any constraint between the clocks of reset conditions and the clocks of resettable node instances. They can be considered asynchronous resets.

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

node rst_clk (rst: bool; clock h: bool) returns (s:int; t:int when h)
let
  s = (restart nat every rst) ();
  t = (restart nat every rst) (() when h);
tel
```

rst	false	false	false	false	false	true	false	false	false	true
h	false	false	true	true	false	false	false	true	true	true
s										
t										

Combining Resets and Clocks

- Resets do not introduce any constraint between the clocks of reset conditions and the clocks of resettable node instances. They can be considered asynchronous resets.

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

node rst_clk (rst: bool; clock h: bool) returns (s:int; t:int when h)
let
  s = (restart nat every rst) ();
  t = (restart nat every rst) (() when h);
tel
```

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

Combining Resets and Clocks

- Resets do not introduce any constraint between the clocks of reset conditions and the clocks of resettable node instances. They can be considered asynchronous resets.

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

node rst_clk (rst: bool; clock h: bool) returns (s:int; t:int when h)
let
  s = (restart nat every rst) ();
  t = (restart nat every rst) (() when h);
tel
```

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

Question

- Is there a difference?

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

node callintegr (e: int; clock h: bool) returns (x,y: int when h)
let
  x = integr(e) when h;
  y = integr(e when h);
tel
```

e	1	1	1	1	1	1	1	1	1	1
h	false	false	true	true	true	false	false	true	true	true
x										
y										

Question

- Is there a difference? – yes.

```
node integr (e:int) returns (x:int)
let  x = e + (0 -> pre x); tel

node callintegr (e: int; clock h: bool) returns (x,y: int when h)
let
  x = integr(e) when h;
  y = integr(e when h);
tel
```

e	1	1	1	1	1	1	1	1	1	1
h	false	false	true	true	true	false	false	true	true	true
x			3	4	5			8	9	10
y			1	2	3			4	5	6