Semantic Checks

- **Causality check fails.**
  - "no flow instantaneously depends on itself".

- **Initialization analysis would pass.**
  - However, initialization analysis is preceded by causality analysis.
  - Hence: precondition of initialization analysis is that program is causal (i.e. non-cyclic).

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

- The simple node activation does not produce a value at each cycle but only when the instance is activated.
  \[ y_1, \ldots, y_p = (\text{activate } N \text{ every } c)(e_1, \ldots, e_n) \]

- This is equivalent to
  \[ y_1, \ldots, y_p = N((e_1, \ldots, e_n) \text{ when } c); \]

- What happens here?

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

- 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.
Simple Node Activation

- Question: does this example work when using the base clock for h?

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

- Consider the definition of the activate statement:

```
t = (activate integr every h)(e);
\approx t = integr(e) when h;
```

- Then we get:

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

Simple Node Activation

- When \( \text{integr}(e) \) runs on the base clock, \( \text{integr}(e) \approx \text{integr}(e) \) when \( h \).
- But:
- The clock of a node can vary in different instantiations, so in \text{two_instances4} in general \( \text{integr}(e) \) does not run on the base clock.
- The base clock cannot syntactically be specified in SCADE.
- The problem whether two Boolean expressions denote the same flow is undecidable => clocks are equal if they can be syntactically unified.
- The clock of a node activation is determined by the static clock semantics.
Simple Node Activation

- **Thus:**
- The clocks of `integr(e)` and `integr(e) when h` cannot be syntactically unified, hence they run on different clocks.

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

Clock Expressions

- The **logical time** defines the fastest possible rate of the system, the **base clock**. It corresponds of the rate of a flow clocked on the constant `true` (which is not allowed by Scade syntax).
- All other rates are derived from the base clock; they are defined by **clock expressions**.

**clock_expr ::= id**

<table>
<thead>
<tr>
<th></th>
<th>not id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( id match pattern )</td>
</tr>
</tbody>
</table>

- A clock expression is either an **identifier** denoting a Boolean flow, the **negation** of such an identifier, or a **pattern matching**. In the latter case the identifier must refer to an object of an **enum data type**, and the pattern must refer to one of the items of this data type.
Clock Expressions

- Clock expressions must be defined at the first flow: their initial value cannot depend on an initialization performed elsewhere in the expression.
- Clock expressions are only required by the sampling operator when and (in a dedicated notation) by the activate operator.
- Clock expressions cannot be produced as a result by any operator.
- Example program:

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

- Conditional blocks express **control structures** when the control flow only depends on a condition computable **in the current cycle**. If it is a function of the previous state and Boolean condition, a **State Machine** has to be used.

- \( \text{clocked\_block} ::= \text{activate} \ [\text{if\_block} \mid \text{match\_block}] \)

- \( \text{if\_block} ::= \text{if} \ \text{expr} \ \text{then} \ [\text{data\_def} \mid \text{if\_block}] \)
  
  \[ \text{else} \ [\text{data\_def} \mid \text{if\_block}] \]

- \( \text{match\_block} ::= \text{when} \ \text{expr} \ \text{match} \ (\mid \text{pattern: data\_def})^* \)

- \( \text{data\_def} ::= \text{equation}; \mid \text{scope} \)

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

- **At each cycle** where the condition is present, a conditional block binds a value to each variable it defines.
- The **value** corresponds either to the evaluation of the expression in the activated branch, or to a default value when this expression is missing.

```plaintext
type t2 = enum {inc, stdby, dec};
function Sample2 (clock h: bool; clock k: t2; x: int) returns (y,z: int)
let
    activate when k match
    | inc: y = x+1;
    | stdby: y = x;
    | dec: y = x-1;
    returns y;
    z = merge (h; x when h; x when not h);
tel
```
Conditional Blocks

- **Local definitions** are only evaluated in their corresponding branch when it is activated.
- The **default value** is either the previous value of this variable (the last one computed by any branch), or the evaluation of the default expression given in the declaration of the variable.

```plaintext
type t2 = enum {inc, stdby, dec};
function Sample2 (clock h: bool; clock k: t2; x: int) returns (y,z: int)
let
  activate when k match
   | inc: y = x+1;
   | stdby: y = x;
   | dec: y = x-1;
  returns y;
  z = merge (h; x when h; x when not h);
tel
```
Conditional Blocks

- **Inactivated branches** of a conditional block keep their internal state: the `pre` operator occurring in the expressions of these branches will refer to the last cycle when their branch was activated.

```plaintext
type t2 = enum {inc, stdby, dec};
function Sample2 (clock h: bool; clock k: t2; x: int) returns (y,z: int)
let
    activate when k match
    | inc: y = x+1;
    | stdby: y = x;
    | dec: y = x-1;
returns y;
    z = merge (h; x when h; x when not h);
 tel
```
Conditional Blocks

- Note that with conditional blocks syntactically several different definitions of the same flow can be specified in the same node. But only one of them may be active at the same time.

```plaintext
type t2 = enum {inc, stdby, dec};
function Sample2 (clock h: bool; clock k: t2; x: int) returns (y, z: int)
let
  activate when k match
  | inc: y = x+1;
  | stdby: y = x;
  | dec: y = x-1;
  returns y;
  z = merge (h; x when h; x when not h);
tel
```
Scopes

- A **scope** is either a single equation (including a conditional block), or a set of equations surrounded by `let-tel` and possibly preceded by local variable or signal declarations.

**Syntax:**

```plaintext
data_def ::= equation; | scope  
scope ::= [signal_block][local_block][eqs]  
signal_block ::= sig ID (,ID)*  
local_block ::= var (var_decls;)*  
eqs ::= let (equation;)* tel
```

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

- The argument of the `last` operator is the name of a **signal**, or of an input, local or output variable of the local scope, prefixed by a quote.
Example

type Tenum = enum {red, blue, pink, purple};

node ex3(eI1: Tenum; iI2: int)
  returns (iO1: int last = 0)
let
  activate when eI1 match
  | red: var iV1: int;
  let
    iV1 = 10 + last 'iO1;
    iO1 = iV1 + iI2;
  tel
  | blue:
  let
    activate if iI2 > 0
    then iO1 = iI2 * iI2;
    else iO1 = -iI2 + last 'iO1;
    returns iO1;
  tel
  | pink: iO1 = 100 -> pre iO1 - 1;
  | purple:
  returns iO1;
  tel
Global Flows

- Global flows are defined outside the scope of a node and can be used in node definitions.

- Two types:
  - **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.
  - **sensors:**
    - represent global inputs from the environment, e.g. for accessing the value provided by a physical sensor.
    - runs on the base clock
    - values **cannot change during a cycle** of the base clock.
    - must be **used at least once** in the program
Global Flows

Example:

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

sensor
    speed: real;
    temperature: real;

node N() returns ()
let
    ...
    tel
```
Signals

- Signals in SCADE are used for passing status information, e.g. for communication in state machines.
- Signals correspond to Boolean flows constantly set to false, except at the instant they are emitted.
  - The presence of a signal is false, iff it is not emitted in the current instant.
  - The presence of a signal is true, iff it is emitted in the current instant.
- The difference to normal flows is that a signal can be emitted several times simultaneously.
- Signal declaration:
  ```
  sig ID (, ID)*
  ```
- Signal usage: the name of a signal is prefixed by a quote: `ID
Signals

- A signal is always local to a node (or a state machine) and can neither be input nor output of the node.
- The status (or presence) of a signal $S$ is directly accessible from its name ('$S$) and gives a Boolean flow that can be used in any expression.
- A signal $S$ is emitted by the statement: `emit 'S`.
- The operator: `last 'S`
  - returns the status of the signal $S$ at the previous cycle
  - is undefined at the first cycle.
Array Data Types

- Scade arrays are flows. The language does not provide any way to perform in-place modifications, nor dynamic memory allocation of arrays. The array size has to be known at compile time.
- Array types are a special kind of structure, in which all fields share the same data type. Fields in an array are named by incremented integers, starting from 0.
- 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, and arithmetic operators +, -, *, div, mod.
Basic Array Operators

- An array can be **defined** by enumerating its values:
  ```
  type Vect = int^3; const v123: Vect = [1,2,3];
  ```

- An array can be **defined** by value repetition using the operator ^:
  ```
  type Vect = int^3; const v3: Vect = 3^3;
  ```

- Let v be an array and i a static expression. Then v[i] is the (i+1)th array element.
  ```
  function ScalarProduct3(x,y: Vect) returns (z:int)
  let
      z = x[0]*y[0] + x[1]*y[1] + x[2]*y[2];
  tel
  ```

- For **dynamic indexation** a default value must be defined when accessing an array. The default is used when the index is outside of the array bounds. Note that the index is preceded by a dot:
  ```
  (v.[i] default d)
  ```

- The dot operator . is also used to access structure elements:
  ```
  struct_expr ::= expr.ID
  ```
Basic Array Operators

- **A static slice** can be extracted from an array by providing two static indices \(i_1\) and \(i_2\). Let \(i_1 \leq i_2\) and both belong to the correct range. Then \(v[i_1 .. i_2]\) is the sliced array \([v[i_1], v[i_1 + 1], ..., v[i_2]]\) of size \(i_2 - i_1 + 1\).

- **Array concatenation**: the binary operator \(@\) builds an array of size \(n_1 + n_2\) from an array of size \(n_1\) and an array of size \(n_2\). Let \(T_1\) be an array of size \(n_1\) and \(T_2\) an array of size \(n_2\) and let \(v = T_1@T_2\). Then:
  \[
  \forall i \in \{0, ..., n_1 - 1\}: v[i] = T_1[i] \text{ and } \\
  \forall i \in \{n_1, ..., n_2 - 1\}: v[i] = T_2[i - n_1]
  \]

- The operator \texttt{reverse} permutes the values in an array. Let \(v\) and \(w\) be arrays of length \(n\) and let \(v = \texttt{reverse} w\). Then:
  \[
  \forall i \in \{0, ..., n - 1\}: v[i] = w[n - 1 - i].
  \]

- The operator \texttt{transpose(c;i;j)} swaps the \(i\)th dimension of \(c\) with the \(j\)th dimension of \(c\).
Mixed Constructor

- Let \( w \) be an array of size \( n \), \( k \) a static integer expression, and \( e \) an expression of the basic type of the array. The expression
  \[
  v = (w \text{ with } [k]=e)
  \]
  is an array such that
  \[
  v[i] = \begin{cases} 
  w[i] & \forall i \in \{0, \ldots, n-1\} \setminus k \\
  e & \text{if } i = k \text{ and } k \in \{0, \ldots, n-1\}
  \end{cases}
  \]

- Let \( \text{exp} \) be a structure expression, let \( L \) be one of its labels, and \( e \) be an expression of the corresponding type of this label. Let us denote \( L_{AB} \) the set of all label names. Then the expression
  \[
  (v = \text{exp with .k=e})
  \]
  is defined as:
  \[
  v.lbl = \begin{cases} 
  \text{exp.lbl} & \forall \text{lbl} \in L_{AB} \setminus k \\
  e, & \text{if } \text{lbl} = k
  \end{cases}
  \]
Mixed Constructor

- Example:

```haskell
type Tstr = {l1: int, l2: real};
  Tarr = int^3;

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

end
```
Multidimensional Operators

- **Array types** are uni-dimensional. A matrix is encoded as an array of arrays.
- The type definition `Mat64 = real^4^6` defines a matrix of six arrays of four elements each.
- Consider the variable `M` of type `Mat64`. Let
  \[ M = [m_{00}, m_{01}, m_{02}, m_{03}, m_{10}, m_{11}, m_{12}, m_{13}, ...] \]. Then `M[i][j]` first checks whether `i` belongs to the interval [0..5], then that `j` belongs to [0..3] and returns element `m_{ij}`. The **static projection** is left-associative, i.e. `M[i][j] = (M[i])[j]`.
- Note that the order of index declaration is the reverse from the order used for selection.
- Example:
  ```
  const C: int^3^2 = [[2,4,6],[1,3,5]];
  => C[0][2] = 6;
  ```
Iterators

- Scade provides predefined ways to apply a node to arguments of an array data type.
- These operators successively apply this node on every element of the array and produce either a new array (the \texttt{map} family) or an element of the basic type of the array (the \texttt{fold} family).

General syntax:
```plaintext
apply_{expr} ::= operator \ (list)
operator ::= iterator \ operator \ <\langle \text{dimension} \rangle >
iterator ::= \texttt{map} \ | \ \texttt{fold} \ | \ \texttt{mapfold} \ | \ \texttt{mapi} \ | \ \texttt{foldi}
```

Example:
```
X = (\texttt{map} \ N \ <\langle \text{d} \rangle >)(x, \ldots, z);
```
The Map Iterator

- Let \( N \) be an operator which requires \( m \) arguments of type \( t^{in} \) and produces \( n \) results of type \( t^{out} \), and let \( x^1, \ldots, x^m \) be arrays of size \( d \) with elementary type \( t^{in} \).
- Then the \textbf{map} iterator \( v = (\text{map } N \ll <d>>)(x^1, \ldots, x^m) \): produces a new operator whose result are \( n \) arrays \( v^i \) of size \( d \) with elementary type \( t^{out} \).

The entries of the new arrays are build by applying operator \( N \) iteratively to all successive elements of \( x^1, \ldots, x^m \):

\[
\forall j \in \{0, \ldots, d - 1\} : v^1[j], \ldots, v^n[j] = N(x^1[j], \ldots, x^m[j]);
\]

```plaintext
function sum_scalar (a,b: int) returns (c:int)
let
    c = a+b;
return

function vector_sum (t,u: int^3) returns (v: int^3)
let
    v = (map sum_scalar <<3>>)(t,u);
return
```
The Fold Iterator

- Fold iterators apply a scalar operator to an array and an accumulator, iterating from the first array cell to the last one. Each computation is applied to the value of the current cell and the previous accumulator value, leading to a new accumulator value. The final result is the last accumulator value.

- Let $N$ be an operator which requires $m+1$ arguments of type $t$ and produces one output value of type $t$. Let $x^1, \ldots, x^m$ be arrays of size $d$ with elementary type $t$ and let $a$ be an expression of type $t$.

- Then the fold iterator $acc = (\text{fold } N \ll d\gg)(a, x^1, \ldots, x^k)$; produces a new operator whose result $acc$ is a value of elementary type $t$. The result $acc$ is iteratively computed in $d$ steps as:

$$
\begin{align*}
acc_0 &= a \\
acc_{j+1} &= N(acc_j, x^1[j], \ldots, x^m[j]) \text{ for all } j \in \{0, \ldots, d - 1\} \\
acc &= acc_d
\end{align*}
$$
The Fold Iterator

- Example:

```plaintext
function mac(a, x, y: int) returns (z: int)
let
  z = a + x*y;
tel

function scalarprod(x, y: int^3) returns (z: int)
let
  z = (fold mac <<<3>>())(0, x, y);
tel
```
Map and Fold

- Example:

```
function mac(a, x, y: int) returns (z: int)
let
  z = a + x*y;
tel

function scalarprod(x, y: int^3) returns (z: int)
let
  z = (fold mac <<3>>)(0, x, y);
tel

function MatVectProd (A: ; u: ) returns (w: )
let

tel
```

\[
\begin{bmatrix}
2 & 4 & 6 \\
1 & 3 & 5
\end{bmatrix} \times \begin{bmatrix}
1 \\
1
\end{bmatrix} = \begin{bmatrix}
12 \\
9
\end{bmatrix}
\]
Map and Fold

- Example:

```plaintext
function mac(a, x, y: int) returns (z: int)
let
    z = a + x*y;
end

function scalarprod(x, y: int^3) returns (z: int)
let
    z = (fold mac <<3>>)(0, x, y);
end

function MatVectProd (A: int^3^2; u: int^3) returns (w: int^2)
let
end

\[
\begin{bmatrix}
2 & 4 & 6 \\
1 & 3 & 5
\end{bmatrix}
\times
\begin{bmatrix}
1 \\
1
\end{bmatrix}
= \begin{bmatrix}
12 \\
9
\end{bmatrix}
\]
```
Map and Fold

- Example:

```plaintext
function mac(a, x, y: int) returns (z: int)
let
  z = a + x*y;
tel

function scalarprod(x, y: int^3) returns (z: int)
let
  z = (fold mac <<3>>) (0, x, y);
tel

function MatVectProd (A: int^3^2; u: int^3) returns (w: int^2)
let
  w = (map scalarprod <<2>>) (A, u^2);
tel

\[
\begin{bmatrix}
  2 & 4 & 6 \\
  1 & 3 & 5
\end{bmatrix}
\times
\begin{bmatrix}
  1 \\
  1
\end{bmatrix}
= \begin{bmatrix}
  12 \\
  9
\end{bmatrix}
\]
```
The Mapi Iterator

- The `mapi` iterator is derived from the `map` iterator by implicitly using the current iteration index (dimension) as first parameter of operator \( N \) in every step of the iterative computation. In other words the constant array \([0, \ldots, d-1]\) is always used as first implicit argument of \( N \).

- Let \( N \) be an operator which requires \( m+1 \) arguments, the first one of type integer, the other ones of type \( t^{\text{in}} \) and which produces \( n \) results of type \( t^{\text{out}} \), and let \( x^1, \ldots, x^m \) be arrays of size \( d \) with elementary type \( t^{\text{in}} \).

- Then the `mapi` iterator \( v = (\text{mapi } N <<d>>)(x_1, \ldots, x_m) \) produces a new operator whose result are \( n \) arrays \( v^i \) of size \( d \) with elementary type \( t^{\text{out}} \). The entries of the new arrays are build by applying operator \( N \) iteratively to the iteration counter and all successive elements of \( x^1, \ldots, x^m \):

\[
\forall j \in \{0,\ldots,d-1\} : v^1[j],\ldots,v^n[j] = N(j, x^1[j],\ldots,x^m[j]);
\]
The Mapi Iterator

- Example:

```plaintext
def function id (i: int) returns (o:int)
    let
        o = i;
    tel

def function first_10 () returns (t: int^10)
    let
        t = (mapi id <<10>>) ();
    tel
```

Output is: \( t = [0,1,2,\ldots,9] \)
The Foldi Iterator

- The operator \texttt{foldi} behaves the same as \texttt{fold}, but operator \texttt{N} is required to take an extra integer argument as first input. The current iteration index (dimension) is passed as first argument.
- Let \texttt{N} be an operator which requires one argument of type \texttt{int} and \( m+1 \) arguments of type \( t \) and produces one output value of type \( t \). Let \( x^1, \ldots, x^m \) be arrays of size \( d \) with elementary type \( t \) and let \( a \) be an expression of type \( t \).
- Then the \texttt{foldi} iterator \( acc = (\texttt{foldi} N \ll d>>)(a, x^1, \ldots, x^k) \); produces a new operator whose result \( acc \) is a value of elementary type \( t \). The result \( acc \) is iteratively computed in \( d \) steps as:

\[
\begin{align*}
acc_0 &= a \\
acc_{j+1} &= N(j, acc_j, x^1[j],\ldots,x^m[j]) \text{ for all } j \in \{0,\ldots,d-1\} \\
acc &= acc_d
\end{align*}
\]
The Foldi Iterator

- Example:

```plaintext
function add_two(a,b: int) returns (s: int)
    let
        s = a+b;
    tel

function sum_first10 () returns (o: int)
    let
        o = (foldi add_two <<10>>)(0);
    tel

Output is: o = 45
```
Required Properties

- **Causality**: The output at any instant $t$ may only depend upon input received before or at $t$.

- **Bounded memory**: There must be a finite bound such that, at each instant the number of past input values necessary to produce a new output value remains smaller than that bound.

- **Efficient** code generation.

- Execution time **predictability**: no unbounded loops, no recursion, safe upper bounds on worst-case execution time (WCET) available.
Static Semantics

- The compiler performs static semantics checks to determine whether a given program is feasible.
- The static semantics guarantees that the execution of a Scade model is always possible at every cycle of the base clock. It consists of five parts:
  - **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.
Clock Consistency

- Consider the following (illegal) example:
  \[ b = \text{true} \rightarrow \text{not pre } b; \]
  \[ y = x + (x \text{ when } b); \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>( x_0 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>x when b</td>
<td>( x_0 )</td>
<td>( x_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x+(x when b)</td>
<td>( x_0 + x_0 )</td>
<td>( x_1 + x_2 )</td>
<td>( x_2 + x_4 )</td>
<td>( x_3 + x_6 )</td>
</tr>
</tbody>
</table>

- The computation of the \( 2n \)th value of \( y \) needs the \( 2n \)th and the \( n \)th values of \( x \).
- Problem: not possible with bounded memory.
- Consequence: only streams of the same clock can be combined.
- Problem: undecidable whether two boolean expressions denote the same flow.
Clock Analysis

- Thus: In SCADE two boolean expressions define the same clock iff they can be **syntactically** unified. Any combination of flows has to syntactically share the same clock.

- Examples:
  \[ x = a \text{ when } (y > z) \]
  \[ y = b + c \]
  \[ u = d \text{ when } (b + c > z) \]
  \[ v = e \text{ when } (z < y) \]
  - \( x \) and \( u \) share the same clock.
  - \( x \) and \( v \) have different clocks.

- Conditional blocks and state machines can be expressed based on clocks. Activating a branch or a state then means that the corresponding clock is true, while the clock of all other branches and states is false.
Clock Rules

- Let a clock environment $\omega$ be a function from identifiers to clocks.
- Let $CK(E, \omega)$ be the clock of the expression $E$ in the environment $\omega$.
- For an equation $X = E$ holds $\omega(X) = CK(E, \omega)$.
- Let $\perp$ be the undefined clock and $\top$ the erroneous clock. Then $ck \sqsubseteq ck' \iff (ck = \perp \lor ck' = \top \lor ck \equiv ck')$
- Let $\sqcup$ denote the least upper bound operator.

- **Constants**: For any constant $k$, $CK(k, \omega) = true$ (the basic clock).
- **Variables**: For any identifier $X$, $CK(X, \omega) = \omega(X)$.
- Arguments and results of combinatorial operators must share the same clock: $CK(op(E_1, E_2, \ldots, E_n), \omega) = \bigsqcup_{i=1}^{n} CK(E_i, \omega)$
Clock Rules

- $CK(\text{pre}(E), \omega) = CK(E, \omega)$

- $CK(E_1 \rightarrow E_2), \omega) = CK(E_1, \omega) \sqcup CK(E_2, \omega)$

- The input expression of a map / fold / mapi / foldi operator must take all its inputs based on a unique clock $ck$ and has to return outputs based on $ck$. The new operator follows exactly the same rules.
Clock Rules

- Let $K$ be a clock expression on identifier $X$, i.e.
  $K = X \mid \text{not } X \mid X \text{ match } p$.
  Then the clock of $K$ is the clock of $X$: $CK(K, \omega) = CK(X, \omega) = ck$.

- $K$ defines a new clock
  $[K] = ck' = ck \text{ on } X \mid ck \text{ on not } X \mid ck \text{ on } X \text{ match } p$.

Then $ck'$ is defined as follows:

- $ck' = ck \text{ on } X$:
  $ck'$ is true when $ck$ is true and $X$ evaluates to true

- $ck' = ck \text{ on not } X$:
  $ck'$ is true when $ck$ is true and $X$ evaluates to false

- $ck' = ck \text{ on } X \text{ match } p$:
  $ck'$ is true when $ck$ is true and $X$ evaluates to $p$. 
Clock Rules

- Consider the expression \( X = E \ when \ K \). Then \( E \) has to be on the same clock as \( K \), i.e. \( CK(E, \omega) = CK(K, \omega) = ck \) and the clock of \( X = (E \ when \ K) \) is the clock defined by \( K \):
  \[ CK(X = E \ when \ K, \omega) = \lceil K \rceil = ck'. \]

- Consider the expression \( \text{merge}(K; E_1; E_2) \). Let \( CK(K, \omega) = \omega(K) = ck \). Then the clock of \( E_1 \) has to be \( CK(E_1, \omega) = ck \) on \( X \) and \( CK(E_2, \omega) = ck \) on not \( X \) for some \( X \), or vice versa. Then:
  \[ CK(\text{merge}(K; E_1; E_2, \omega)) = ck \ (\neq \lceil K \rceil). \]

- Consider the expression \( \text{merge}(K; E_1; \ldots; E_k) \). Let \( CK(K, \omega) = \omega(K) = ck \). Then the clock of each \( E_i \in \{E_1, \ldots, E_k\} \) has to be \( CK(E_i, \omega) = ck \) on \( X \) match \( p_i \) for some identifier \( X \) such that \( \cup_i p_i \) is the enumeration of all values of the type of \( X \). Then:
  \[ CK(\text{merge}(K; E_1; \ldots; E_k, \omega)) = ck. \]
Clock Rules

- Let $N$ be a user-defined operator (node or function) with the input clocks $ck^i_1,...,ck^i_m$ and the output clocks $ck^o_1,...,ck^o_n$. Then $N$ is associated with a clock constraint $ck^i_1,...,ck^i_m \rightarrow ck^o_1,...,ck^o_n$.

- Let $N(E_1,...,E_m)$ be an operator application, and let $ck_k$ be the fastest clock among the parameters $E_1,...,E_m$. Then the parameters have to fit the clock constraint of $N$ when based on this fastest clock $ck_k$. 
Clock Rules

- The input operator of an `activate` operator must take all its inputs based on a unique given clock `ck` and has to return outputs based on `ck`. All additional parameters of the `activate` (clock expressions, conditional and default expressions) must also be based on the same clock `ck`.

- The clock of the application of an `activate` operator is the clock defined by the clock expression passed as argument to the `activate`.

- Simple node activation: \( X = (activate \ N \ every \ K)(E_1, \ldots, E_m) \). Let \( ck' \) be the clock defined by the clock expression \( K \). Then:

\[
\begin{align*}
CK((activate \ N \ every \ K)(E_1, \ldots, E_m), \omega) \\
= CK(N((E_1, \ldots, E_m) \ when \ K), \omega) \\
= CK(K, \omega) = ck
\end{align*}
\]
Clock Rules

- Node activation with output flows:
  \[ X = (\text{activate } N \text{ every } c \text{ default } (d_1, \ldots, d_p))(E_1, \ldots, E_m) \]. Let \( c \) be a conditional expression with \( CK(c, \omega) = ck \) and let \( ck' \) be the Boolean clock defined by the equation \( h=c \). Then

\[
CK\left( (\text{activate } N \text{ every } h \text{ default } (d_1, \ldots, d_p)) (E_1, \ldots, E_m), \omega \right) \\
= CK(merge(h; N((E_1, \ldots, E_m)\text{when } h); (d_1, \ldots, d_p)\text{when not } h), \omega) \\
= CK(h, \omega) = ck
\]
Clock Rules

- Node activation with output memorization:
  \[ X = (activate \ N \ every \ c \ initial \ default \ (i_1,\ldots,i_p))(E_1,\ldots,E_m). \]
  Let \( c \) be a conditional expression and let \( ck' \) be the Boolean clock defined by the equation \( h = c. \)
  Then

  \[
  CK(activate \ N \ every \ c \ initial \ default \ (i_1,\ldots,i_p))(E_1,\ldots,E_m), \omega) \\
  = CK(merge(h; N((E_1,\ldots,E_m) \ when \ h); \\
      ((i_1,\ldots,i_p) \rightarrow \text{pre}(E_1,\ldots,E_p)) \ when \ not \ h), \omega) \\
  = CK(h, \omega) = ck
  \]
## Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ n = (0 \rightarrow \text{pre}(n) + 1) \]

\[ e = (\text{true} \rightarrow \text{not pre}(e)) \]

\text{n when e}

\text{Counter((1,1,false) when C)}

\text{Counter(1,1,false) when C}

```c
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
    else pre(count)+incr;
tel
```
Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (true-&gt;not pre(e))</td>
<td>n when e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td>Counter(1,1,false) when C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```javascript
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
    else pre(count)+incr;
```

node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
    else pre(count)+incr;

node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
    else pre(count)+incr;
```
Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (true-&gt;not pre(e))</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
</tbody>
</table>

n when e

Counter((1,1,false) when C)

Counter(1,1,false) when C

```plaintext
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
        else pre(count)+incr;
tel
```
## Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (true-&gt;not pre(e))</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Counter((1,1,false) when C)

Counter(1,1,false) when C

```plaintext
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
      else pre(count)+incr;
tel
```
### Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (true-&gt;not pre(e))</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter(1,1,false) when C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
node Counter (init, incr: int; reset: bool)
  returns (count:int);
let
  count = init -> if reset then init
  else pre(count)+incr;
```
Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (true-&gt;not pre(e))</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter(1,1,false) when C</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
              else pre(count)+incr;
tel
```
Causality Analysis

- SCADE only allows **acyclic** equation systems. Note: acyclic equations have a unique solution.
- \( X=E \) is acyclic if \( X \) does not occur in \( E \) unless as **subterm** of the **pre** or **last** operator.
- Examples:
  - \( X = X \) and **pre**(\( X \)) is cyclic
  - \( X = Y \) and **pre**(\( X \)) is acyclic
  - \( X = Y; \)
    - \( Y = X; \) is cyclic
  - \( X = Y; \)
    - \( Y = \text{pre} \ X; \) is acyclic (though not well-initialized)
- Also structural deadlocks which are not true ones are rejected:
  - \( X = \text{if } C \text{ then } Y \text{ else } Z; \)
  - \( Y = \text{if } C \text{ then } Z \text{ else } X; \)
Initialization Analysis

- Goal: demonstrate that for given input sequences the output sequences are completely defined (without *nil*).

- A flow must never be delayed more than once without initializing the first value.

- An access to the latest value of a shared flow (*last* 'x) must not happen in the first cycle, where it may not be defined.

- Example:
  
  \[
  x = \text{pre}(\text{pre}(x)); \quad \longrightarrow \text{not well-initialized}
  \]
  
  \[
  x = 1 \rightarrow \text{pre}(\text{pre}(x)); \quad \longrightarrow \text{not well-initialized}
  \]
Initialization Analysis

- The initialization analysis is based on:
  - two literals 0 and 1 denoting expressions that either introduce one delay or no delay, respectively
  - a join operator \( \sqcup \) to compute the maximal delay of two types:
    - \( \forall d: 1 \sqcup d = 1 \)
    - \( \forall d: 0 \sqcup d = d \)
  - Intuitively, a flow \( x \) is called well-initialized iff \( \text{delay}(x) = 0 \).
- Components:
  - Delay analysis function \( \text{delay} \)
  - Constraint analysis function \( \text{dcons} \)
Initialization Analysis

- Initialization analysis of node/function declaration
  node \( N (X_1: \Phi_1; \ldots; X_m: \Phi_m) \) returns \((Y_1: \Psi_1; \ldots; Y_n: \Psi_n)\):

  - Each input variable \( X_i \) is associated with an unbound delay variable \( \rho_i \).
  - Delay formula are computed for all output variables.
  - Initialization constraints are collected for expressions in the body. From them, initialization constraints on input variables are derived.
  - The body of \( N \) is well-initialized, iff
    - No initialization constraints are violated, and
    - For well-initialized input delay variables all output variables are well-initialized: \( \forall i: \rho_i = 0 \Rightarrow \forall k: delay(o_k) = 0 \).
Initialization Analysis

- \(\text{delay}(c) = 0\ \forall\ \text{constants } c\)

- \(\text{delay}(X) = \begin{cases} 
0, & \text{if } X \text{ is a signal} \\
\rho_i, & \text{if } X \text{ is input parameter } X_i \\
\text{delay}(E), & \text{if } X \text{ is local variable or output variable defined by } X = E 
\end{cases}\)

- \(\text{delay}(X \circ Y) = \text{delay}(X) \sqcup \text{delay}(Y)\) for combinatorial operators \(\circ\)

- \(\text{delay}(\text{pre } E) = 1\)

- \(\text{delay}(X \rightarrow Y) = \text{delay}(X)\)

- \(\text{delay(\text{if } E_1 \text{ then } E_2 \text{ else } E_3)} = \text{delay}(E_1) \sqcup \text{delay}(E_1) \sqcup \text{delay}(E_3)\)

- \(\text{delay(\text{case } E_1 \text{ of } E_2 \ldots E_k)} = \text{delay}(E_1) \cup \cdots \cup \text{delay}(E_k)\)
Initialization Analysis

- $delay(f by (E_1; d; E_2)) = delay(E_1) \sqcup delay(E_2)$
  - Note that this is different from the $\rightarrow$ operator.
- $delay(E_1 \text{ when } E_2) = delay(E_1) \sqcup delay(E_2)$
- $delay(merge(h; E_1, ..., E_k)) = delay(h) \sqcup delay(E_1) ... \sqcup delay(E_k)$
- $delay(last 'X) = \begin{cases} delay(X), \text{ if a last-value has been declared for X} \\ 1, \text{ otherwise} \end{cases}$
Initialization Analysis

```plaintext
node initlast1() returns (out: int)
let
  out = (1 + last 'out);
 tel

node initlast2() returns (out: int last=3)
let
  out = (1 + last 'out);
 tel

node initlast3() returns (out: int last=3)
let
  out = (1 + pre out);
 tel
```

- Initialization Error
- Initialization analysis passes
- Initialization Error
Initialization Analysis

- $d\text{cons}^N(E_1 \rightarrow E_2, C^N) = C^N \cup d\text{cons}^N(E_1) \cup d\text{cons}^N(E_2)$

- $d\text{cons}^N(fby(E_1; d; E_2), C^N) = C^N \cup \{\text{delay}(E_1) = 0\}$
  $\cup \{\text{delay}(E_2) = 0\}$

- $d\text{cons}^N(E_1 \text{ when } E_2, C^N) = C^N \cup \{\text{delay}(E_2) = 0\}$

- $d\text{cons}^N(merge(h; E_1, \ldots, E_k), C^N) = C^N \cup \{\text{delay}(h) = 0\}$
  $\cup \{\text{delay}(E_1) = 0\} \cup \ldots$
  $\cup \{\text{delay}(E_k) = 0\}$

- $d\text{cons}^N(last \ 'X, C^N) = C^N \cup \{\text{delay}(X) = 0\}$

- $d\text{cons}^N(pre E, C^N) = C^N \cup \{\text{delay}(E) = 0\}$
Initialization Analysis

- Is this node well initialized?

```plaintext
node init1() returns (out: int)
let
  out = 1 + pre( 1 -> (pre(out)));
tel
```

- What does this node do?

```plaintext
node init2() returns (out: int)
let
  out = 1 + (1 -> pre(1 -> pre(out)));
tel
```

➢ out = (2, 2, 3, 3, 4, 4, ...)

Initialization Error