Realize Your Product Promise®

Model-Based Design
with
SCADE Suite®

Training Resources

Day 3

ANSYS

Brainstorming on Scade

Exercise

Objective:

Come back to the main principles of Scade

Requirements:

Preparation Time: 8 min

On a piece of paper, note down the main principles of Scade that you have studied during the training

AGENDA

Arrays

Iterators

Code Generation

Imported Code

ARRAYS

Sequence of values sharing the same type

Array Type

```
<type>^<size>
```

Any Scade type

Integer constant expression > 0

Do

```
int16^12
bool^4
char^M
float32^6^6
uint8^2^(N+M)
{en:bool,val:char^3}^N
```

(N and M are defined as constant)

Don't

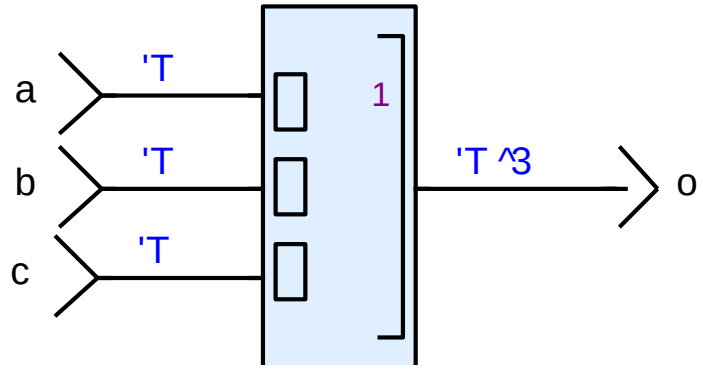
```
int16^0
5^true
bool^x
```

(x is not a constant)

Constructors

Graphical | Scade textual

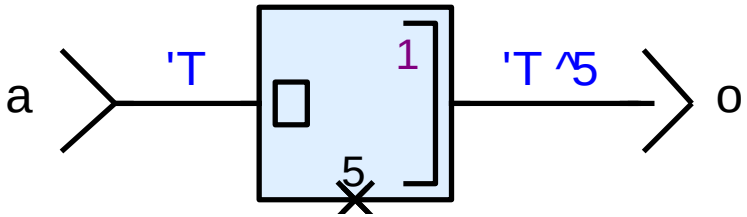
Data Array




```
[a,b,c]
```

'T × 'T × … × 'T → 'T^N

Scalar to Vector

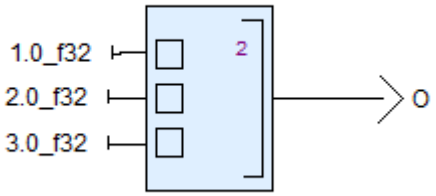



```
a^5
```

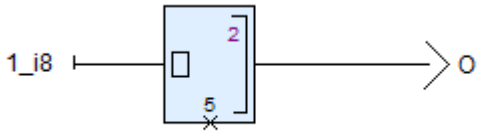
'T × 'int → 'T^N

Constructors

Example



$$o = [1.00000, 2.00000, 3.00000]$$



$$o = [1, 1, 1, 1, 1]$$

Constructors

Do

```
[true,false,true]

(1.2 : float32)^8

[[a,b,c],[d,e,f],[g,h,i]]

'b'^(M+N)

[[1.2_f32,2.6_f32],
 [0.0_f32,2.14_f32]]
```

(N and M are defined as constant)

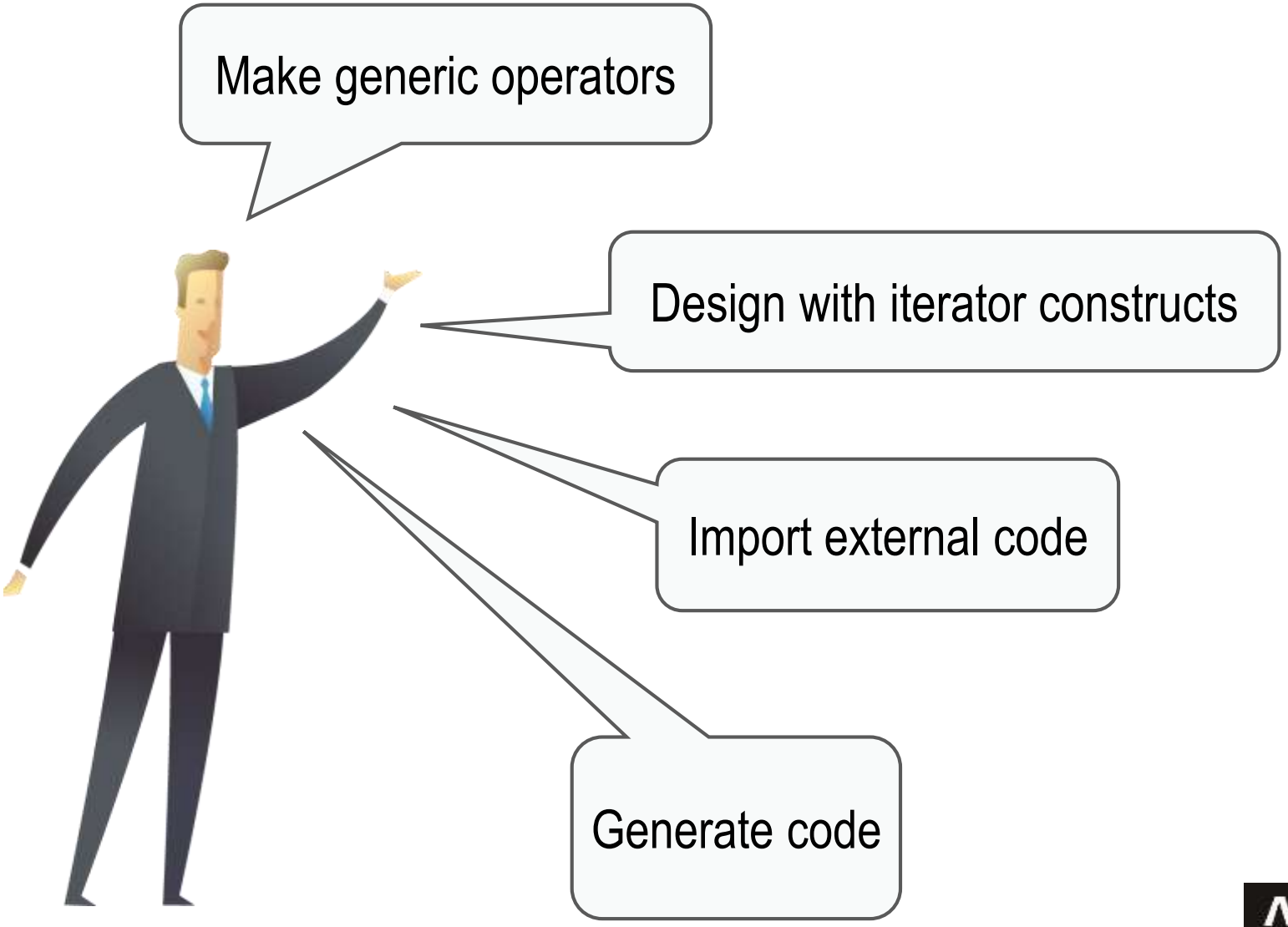
Don't

```
[(1.4_f64),(1.2_i8)]

'c'^(3*a)
```

(a is not a constant)

Lab Objectives

Lab



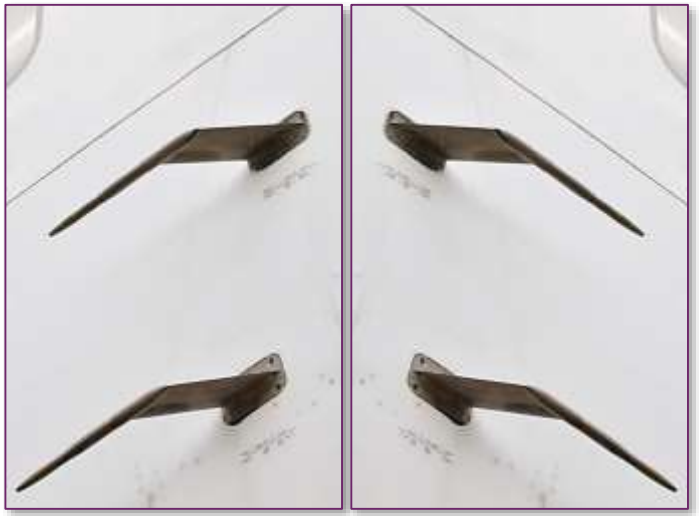
Prerequisites

Lab Prerequisite

Safety Critical Systems interact with their physical environment.

They rely on sensors to observe their physical environment, and on actuators to act on it.

A failure of a sensor or an actuator is a Single Point of Failure: so they need to be redundant.



Redundant Pitot sensors on a B737

Prerequisites

Wikipedia

A pitot tube is a pressure measurement instrument used to measure fluid flow velocity.

It is widely used to determine the airspeed of an aircraft, water speed of a boat, and to measure liquid, air and gas velocities in industrial applications.

The pitot tube measures the local velocity at a given point in the flow stream and not the average velocity in the pipe or conduit.

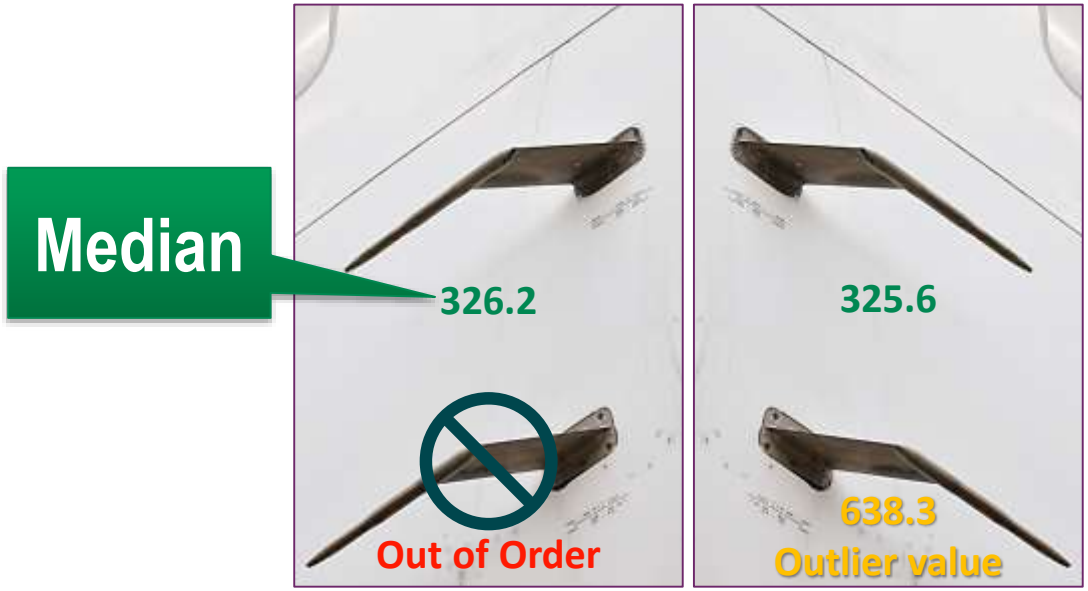
Lab Prerequisite

Prerequisites

Having redundant sensors means that from a range of values one final value has to be determined.

A naive way to determine this value is to use the mean of valid values.

A smarter way to do this is to use the median.



Lab Prerequisite

Prerequisites

Lab Prerequisite

In statistics and probability theory, the median is the numerical value separating the higher half of a data sample, a population, or a probability distribution, from the lower half.

The median of a finite list of numbers can be found by arranging all the observations from the lowest value to the highest value and picking the middle one:

The median of {3, 3, 5, 9, 11} is 5

If there is an even number of observations, then there is no single middle value; the median is then usually defined to be the mean of the two middle values, which corresponds to interpreting the median as the fully trimmed mid-range:

The median of {3, 5, 7, 9} is (5 + 7) / 2 = 6

ANSYS

Lab Objectives

Lab

The aim of this one day lab is to develop a median library operator.

It needs to be able to work on a set of values (any size).

To be able to tell if a value is above or below another one, these values need to have a sorting relation :

In Scade, this means that these values are numeric.

Lab 1: Requirement

Lab Support p.10-11

Objective:

Prepare the design environment and construct the simulation array.

Requirements:

Time: 5 min

Create a new SCADE Suite project:

Project Name: `median`

Package Name: `MEDIAN`

No library

Create two environment constants, under the `MEDIAN` package, to use as:

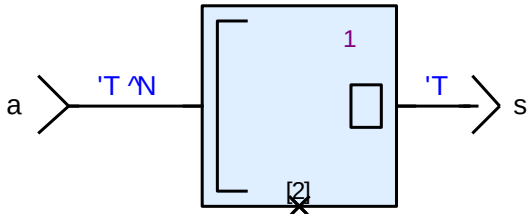
```
AIRSPEED_SENSORS (type: float64^4) =[326.2, 325.6, 638.3, 323.9]

DISTANCE_SENSORS (type: uint8^6): =[45,49,35,33,44,43]
```

Lab

Array Access

Graphical | Scade textual

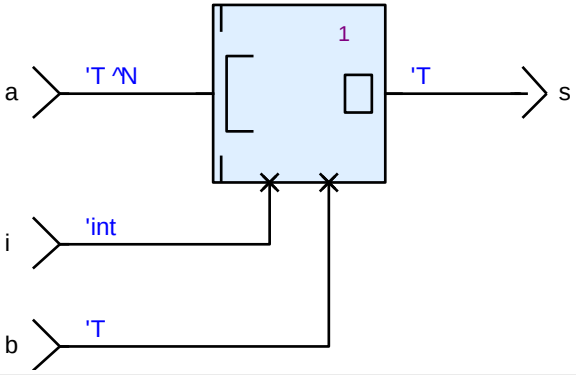
Static Projection




```
a[2]
```

'T ^N × 'int → 'T

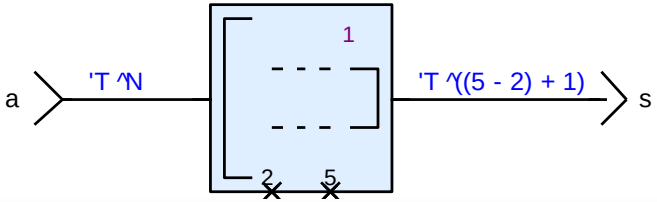
Dynamic Projection




```
(a.[i] default b)
```

'T ^N × 'int × 'T → 'T

Slice

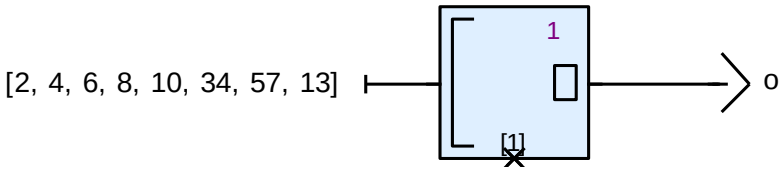



```
a[2..5]
```

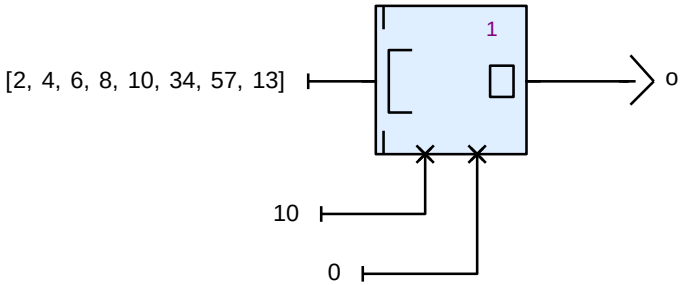
'T ^N × I × J → 'T^(J – I + 1)

Array Access

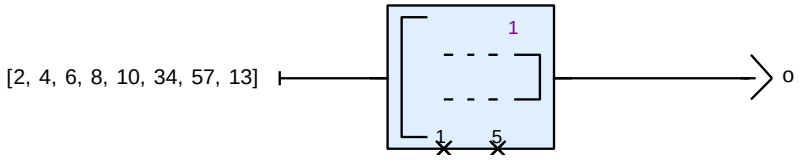
Example



○ = 4



○ = 0



○ = [4,6,8,10,34]

Array Access

```
a: float32^10    i: int8
b: int8^30       c: int8
```

Do

```
a[8]

b[25]

(a.i default 8.6)

(b.32 default c)

a[4..7]
```

Don't

```
a[15]

b[not_constant]

(a.i default 5)

a[3.5..8]

b[20..35]
```

Lab 2: Use Array Access Operators

Lab

Lab Support p.13-15

Objective:

Create the main operator and use array access operators.

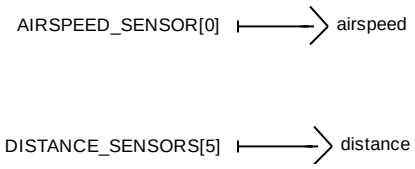
Time: 10 min

Requirements:

Create a `median` operator, under MEDIAN package returning:

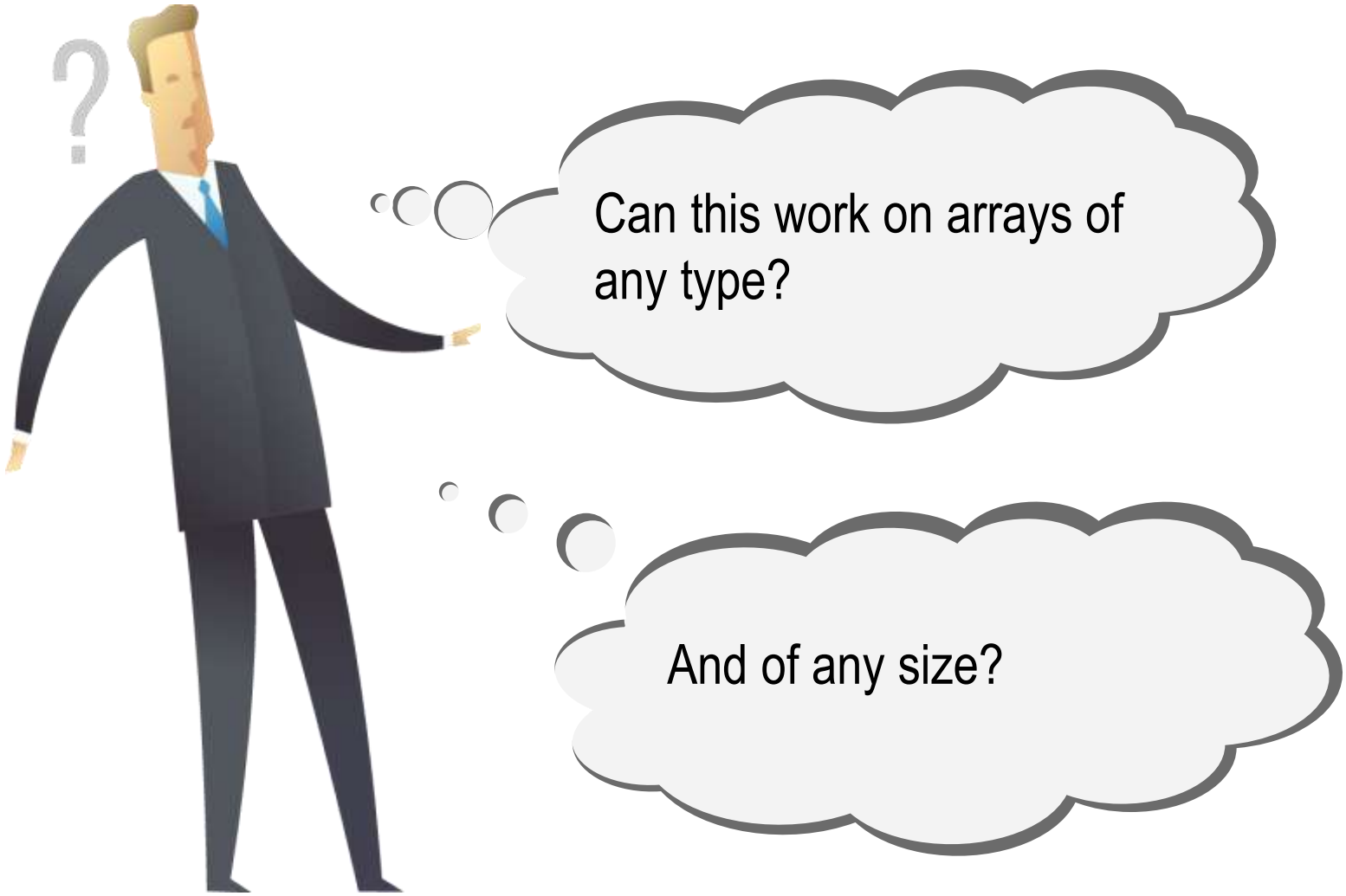
```
airspeed = airspeed_sensors[0]
distance = distance_sensors[5]
```

Name	Kind	Type
airspeed	output	float64
distance	output	uint8



Generate the code, with KCG configuration (KCGAda for Ada target)

Observe the generated code and dimension matching.


Can this work on arrays of any type?
And of any size?

Question

I want to write operators once, not each time for a specific array element type and array size.

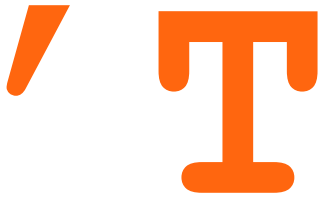
But at the same time I want to ensure type safety !

Genericity

Abstract types and sizes at design time.

For each instance the tool will generate a different, type-safe implementation : this process is named monomorphisation.

Generic Types



Any type identifier starting by ’ is a generic type,
e.g. : ’T, ’Type, ’t, ’int, ’XxX, ’num, ’N, ’AnY …

The implementation type is unknown at design time.
This lets the designer free of the final usage and its concrete types.
At code generation time, each different calling context will lead to a different concrete implementation (monomorphisation).

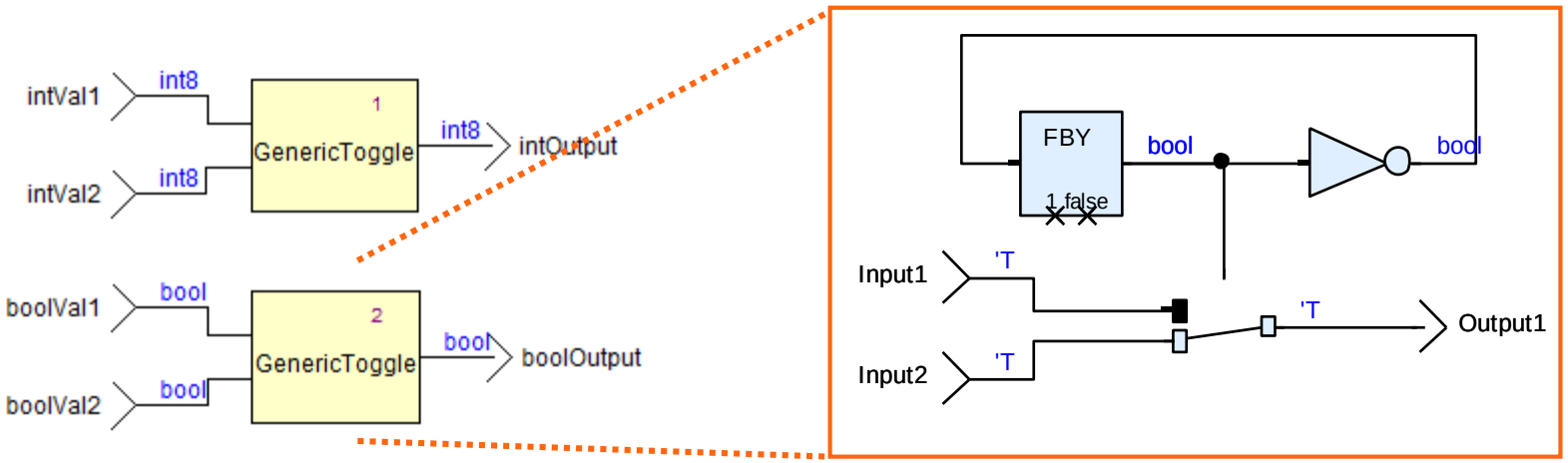
Generic Operator

Example

Operators with generic types

Some predefined operators are polymorphic:

- if-then-else, =, +, etc.
- Has inputs or outputs whose types are generic, i.e defined as `'GenericName`




```
node GenericToggle(Input1 : 'T; Input2 : 'T)
returns (Output1 : 'T)
```

Lab 3: Use a Generic Function (1/2)

Lab Support p.17-22

Objective:

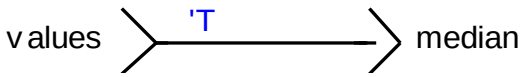
Create a generic function

Requirements:

Time: 10 min

Update the `median` function to be a generic function

Name	Kind	Type
`value`	input	'T
`median`	output	'T

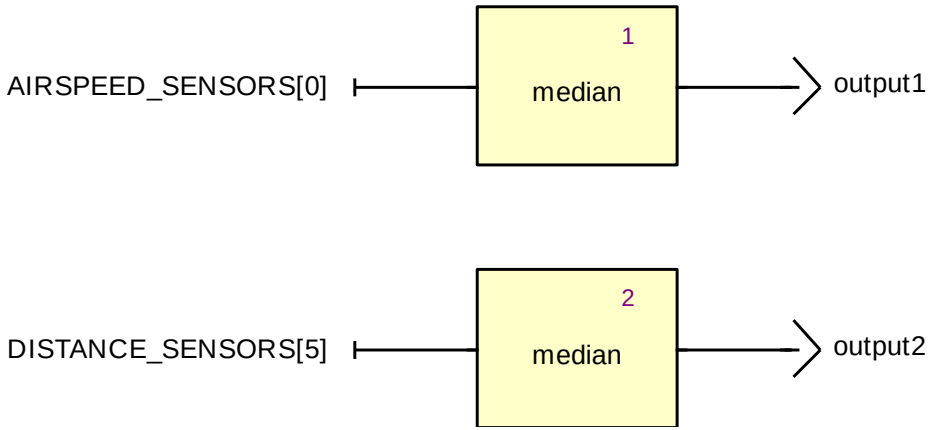


Lab

Lab 3: Use a Generic Function (2/2)

Create a `test` operator to simulate several median instances

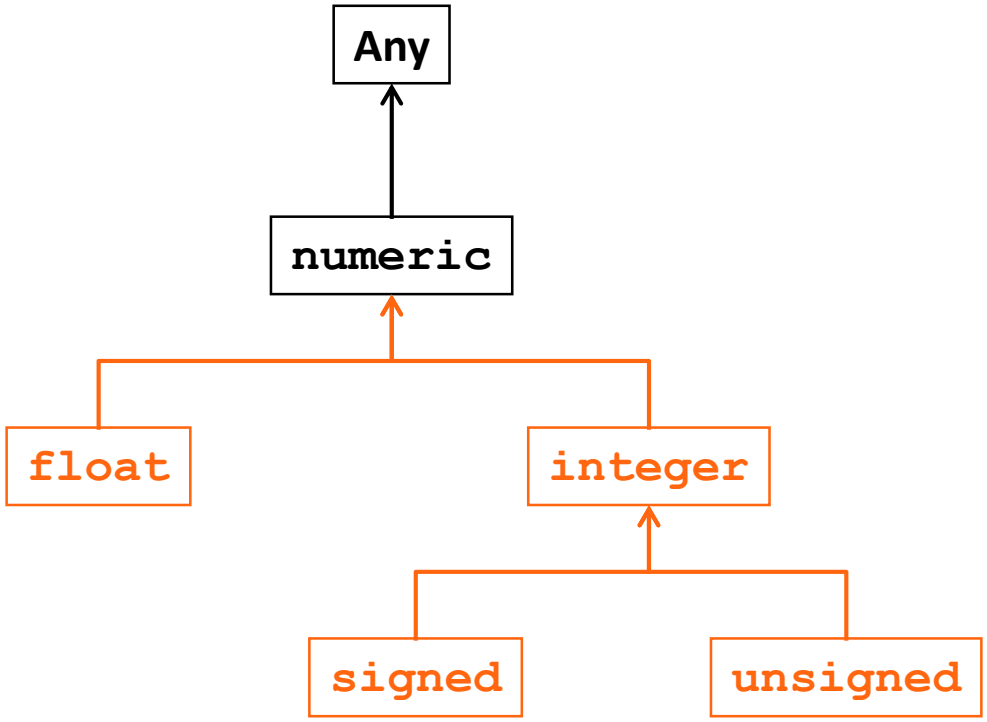
Name	Kind	Type
`output1`	ouput	float64
`output2`	output	uint8



Lab

Constrained Generic Types

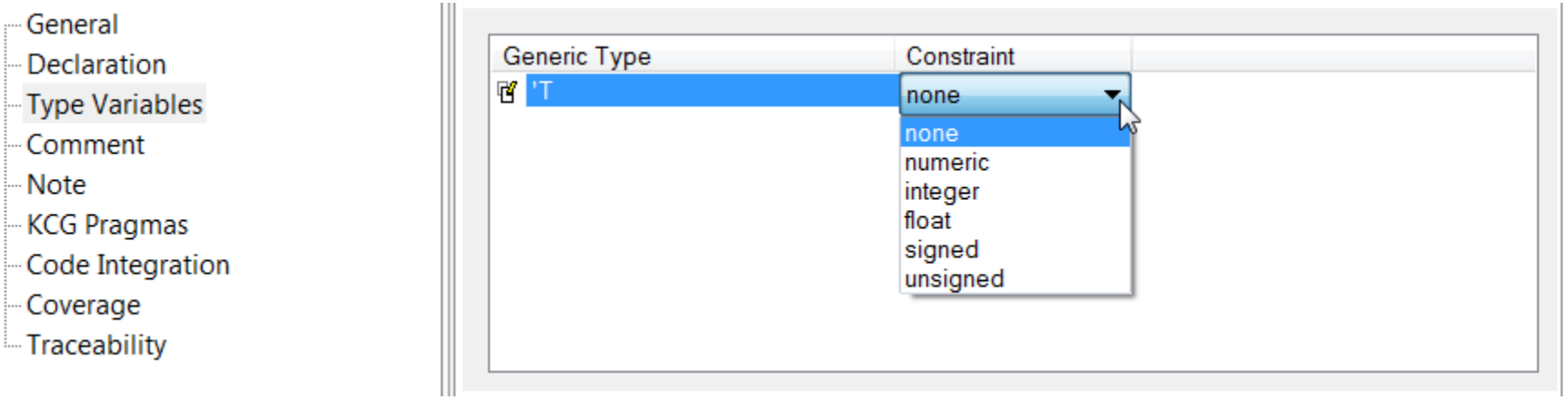
It is possible to constrain a generic type to a narrower definition:



Constrained Generic Types

These constraints can be applied in the scope of **an operator** and on imported types:

Select the required operator and from “Type Variables”:



Lab 4: Constraint to Numeric Type

Lab

Objective:

Constraint to numeric type

Requirements:

Constraint the `median` input type to be numeric

Time: 5 min

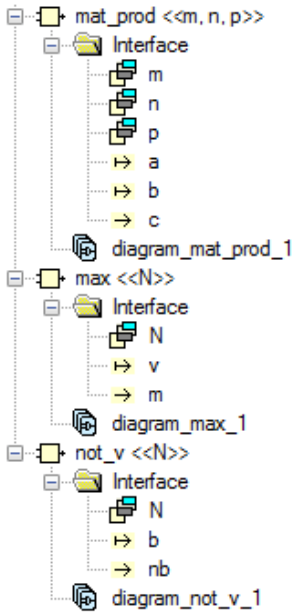
Generic Size

Design operators treating arrays without knowing their size.

Operators can bear size parameters (abstraction of the size).

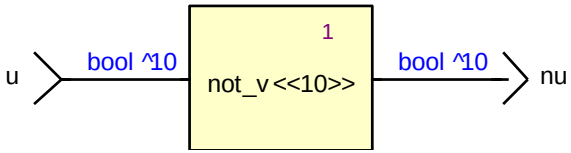
f<<N>>

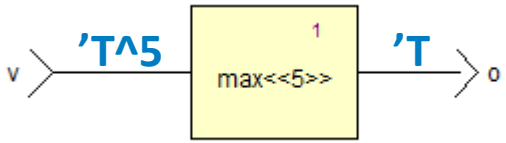
```
mat_prod<<m,n,p>>(a:'T^n^m,b:'T^p^n) returns (c:'T^p^m)

max<<N>>(v:'T^N) returns (m:'T) where 'T is numeric

not_v<<N>>(b:bool^N) returns (nb:bool^N)
```

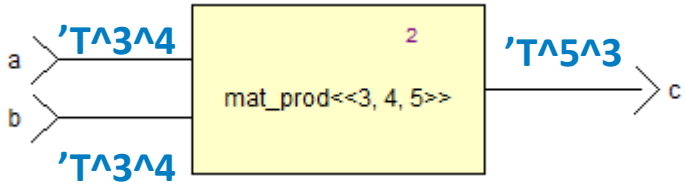


Parameterized Operators

```
not_v<<N>>(u:bool^N) returns (nu:bool^N)

max<<N>>(v:'T^N) returns (m:'T) where 'T is numeric

mat_prod<<m,n,p>>(a:'T^n^m,b:'T^p^n) returns (c:'T^p^m)
```







Parameterized Operators: Data Types

signed<<N>> : signed integer with N bits (N must be equal to 8, 16, 32 or 64)

unsigned<<N>> : unsigned integer with N bits

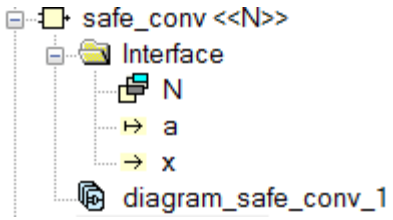
Two predefined types with a size parameter:

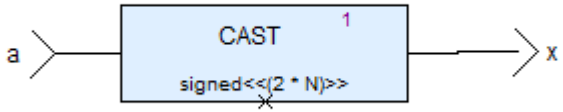
- To allow to define generic Scade nodes where the number of bits of the parameter types can be manipulated as a size parameter
- Useful for implementing Scade functions to encode bit vectors

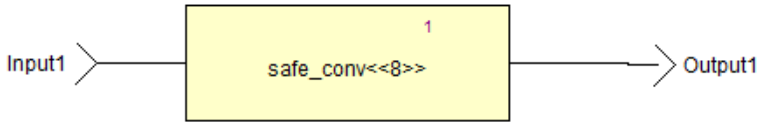
Parameterized Operators: Data Types

Convert an unsigned to a bigger signed:

function safe_conv **<<**N**>>** (a: **unsigned<<**N**>>**) **returns** (x: **signed<<**2*N**>>**)
let
    x = (a : **signed<<**2*N**>>**);
tel







Lab 5: Requirement (2/2)

Lab Support p.26-32

Objective:

Prepare the median design environment

Time: 10 min

Requirements:

Modify the `median` implementation to work with arrays of any type and of any size:

- Instantiate it with 3 different array sizes/types

Update `test` operator

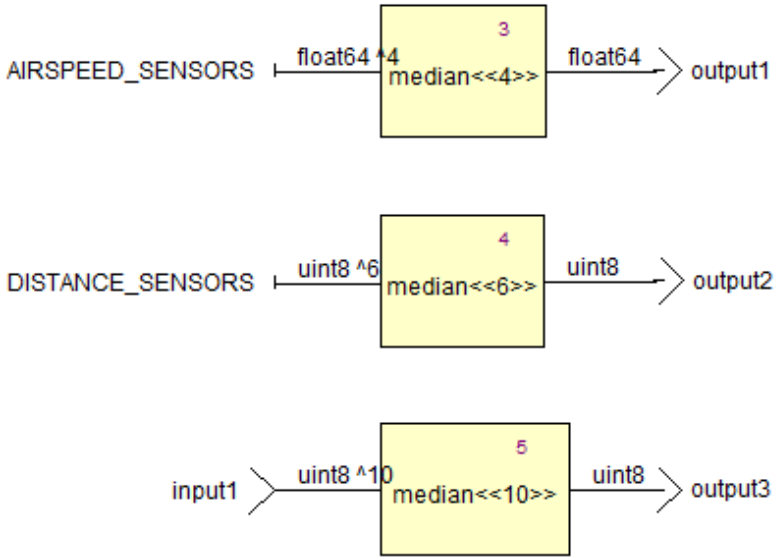
- Observe the effects of monomorphisation in the generated code

Lab

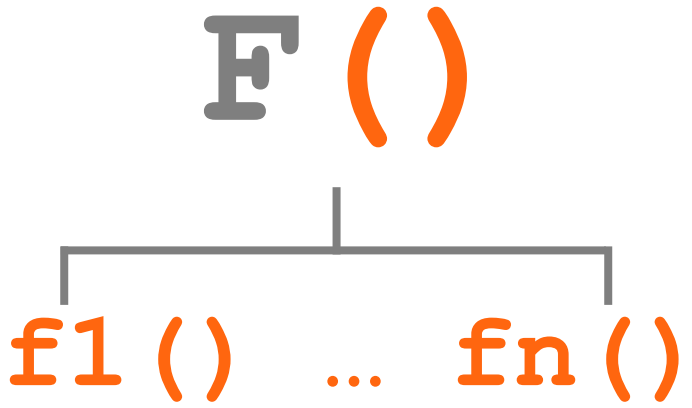
Lab 5: Requirement (2/2)

Update `test` operator

Name	Kind	Type
input1	input	uint8^10
output1	output	float64
output2	output	uint8
output3	output	uint8



Specialization



Define different behaviors for a polymorphic operator depending on the interface type of its instances.

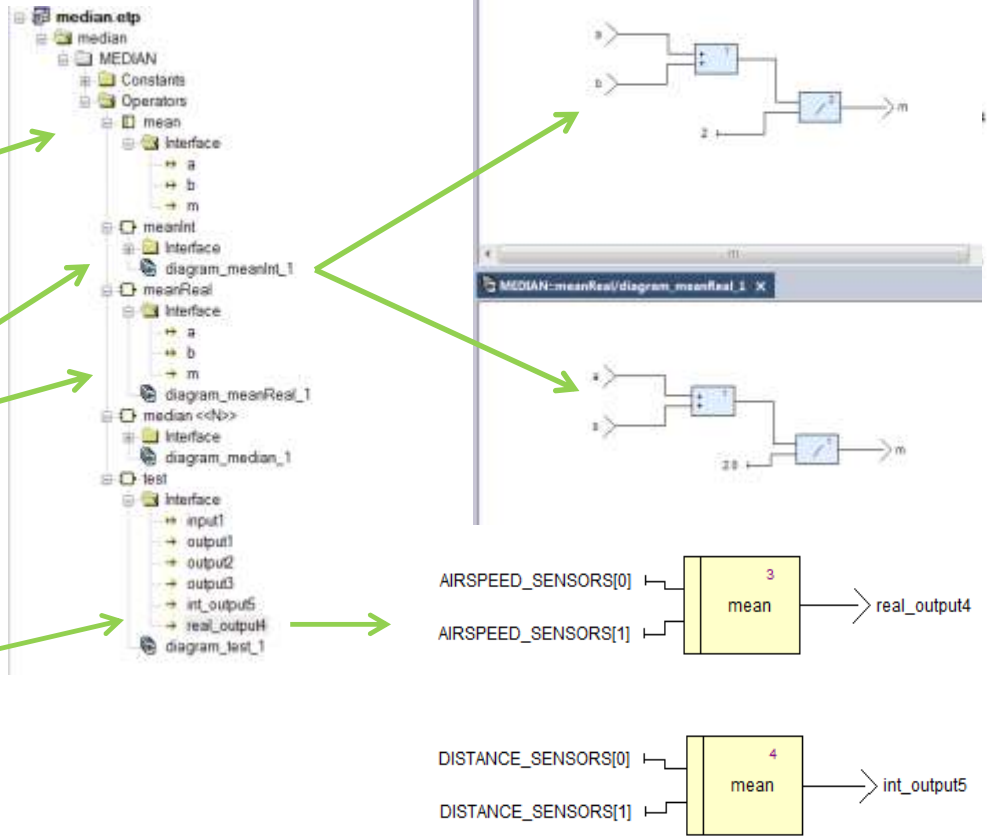
It must be an imported operator.

Polymorphism: Specialization

Specialized operator declared as imported (has at least one input of generic type ‘T)

Specialization operators

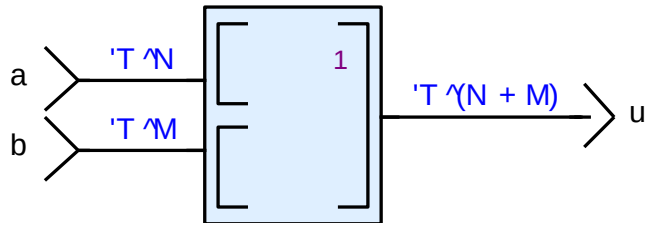
Instantiation operators



Array Operations

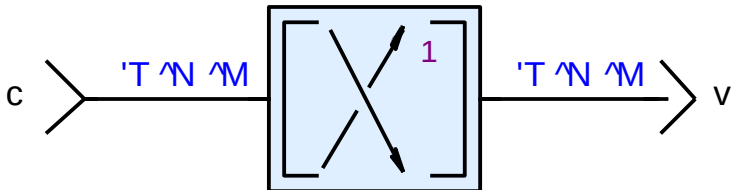
Graphical

Scade textual



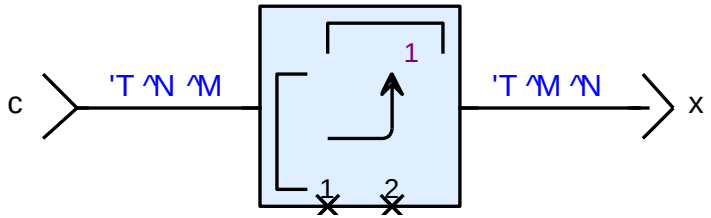

```
a @ b
```

T^N'T^M*...*'T^X -> 'T^(N+M+...+X)*




```
reverse c
```

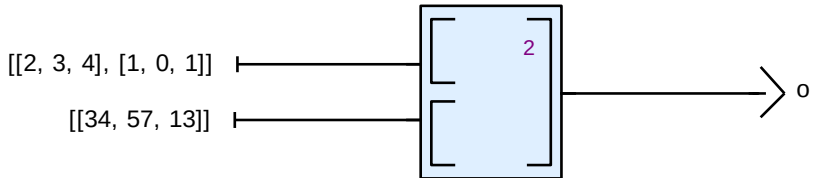
T^N-> T^N



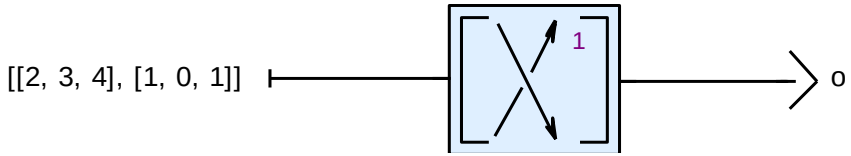

```
transpose(a;1;2)
```

*T^N^M^...^X*int*int -> T^X^M^...^N*

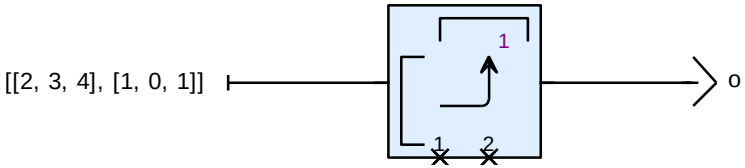
Array Operations




```
o = [[2,3,4],[1,0,1],[34,57,13]]
```




```
o = [[1,0,1],[2,3,4]]
```




```
o = [[2,1],[3,0],[4,1]]
```

Lab 6: Requirement (1/3)

Lab Support p.34-38

Objective:

Create a `mean` operator in the median design environment

Time: 10 min

Requirements:

Create a specialized `mean` operator with specialization for integer and floating values

Name	Kind	Type
a	input	'T
b	input	'T
m	output	'T

'T is numeric

Lab

Lab 6: Requirement (2/3)

Create 2 specialized operators:

- `meanInt`

Name	Kind	Type
a	input	uint8
b	input	uint8
m	output	uint8

- `meanReal`

Name	Kind	Type
a	input	float64
b	input	float64
m	output	float64

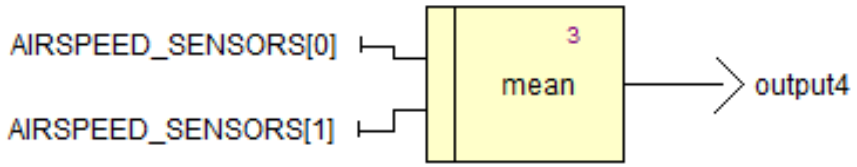
Lab

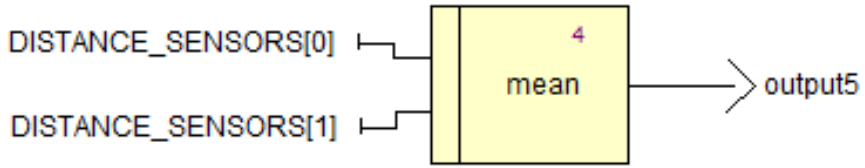
Lab 6: Requirement (3/3)

Lab

Update `test` operator

Name	Kind	Type
input1	input	uint8^10
output1	output	float64
output2	output	uint8
output3	output	uint8
output4	output	float64
output5	output	uint8





AGENDA

Arrays

Iterators

Code Generation

Imported Code

How to apply treatments to array elements?

How to implement the sum of a and b, arrays of 4 integers?

And of a,b: 'T^N?

Can we do that in a safe and efficient way?

What do we Need?

A way to write $\vec{c} = \vec{a} + \vec{b}$

$\forall i \in [0..N\text{-}1],\ c_i = a_i + b_i$

Pointwise application of **+** to $\vec{a}$ and $\vec{b}$ vectors

Exercise

Exercise 1

Objective:

Design the calculation to iterate

Time: 10 min

Requirements:

In a new Exercises project, implement the following function:

```
function elem_plus ( a_i, b_i : 'T)
  c_i = a_i + b_i ;
returns (c_i : 'T)
```

'T is numeric

Exercise 1: Solution

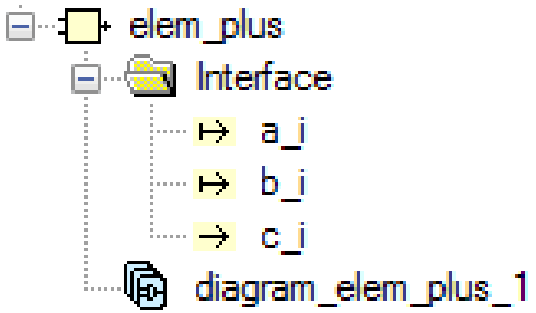
Exercise

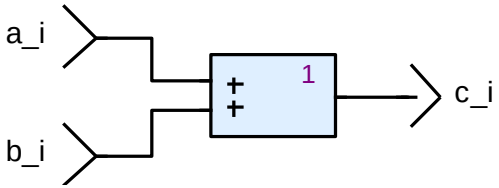
Create a new project named Exercises:

- Project Name: Exercises
- No library

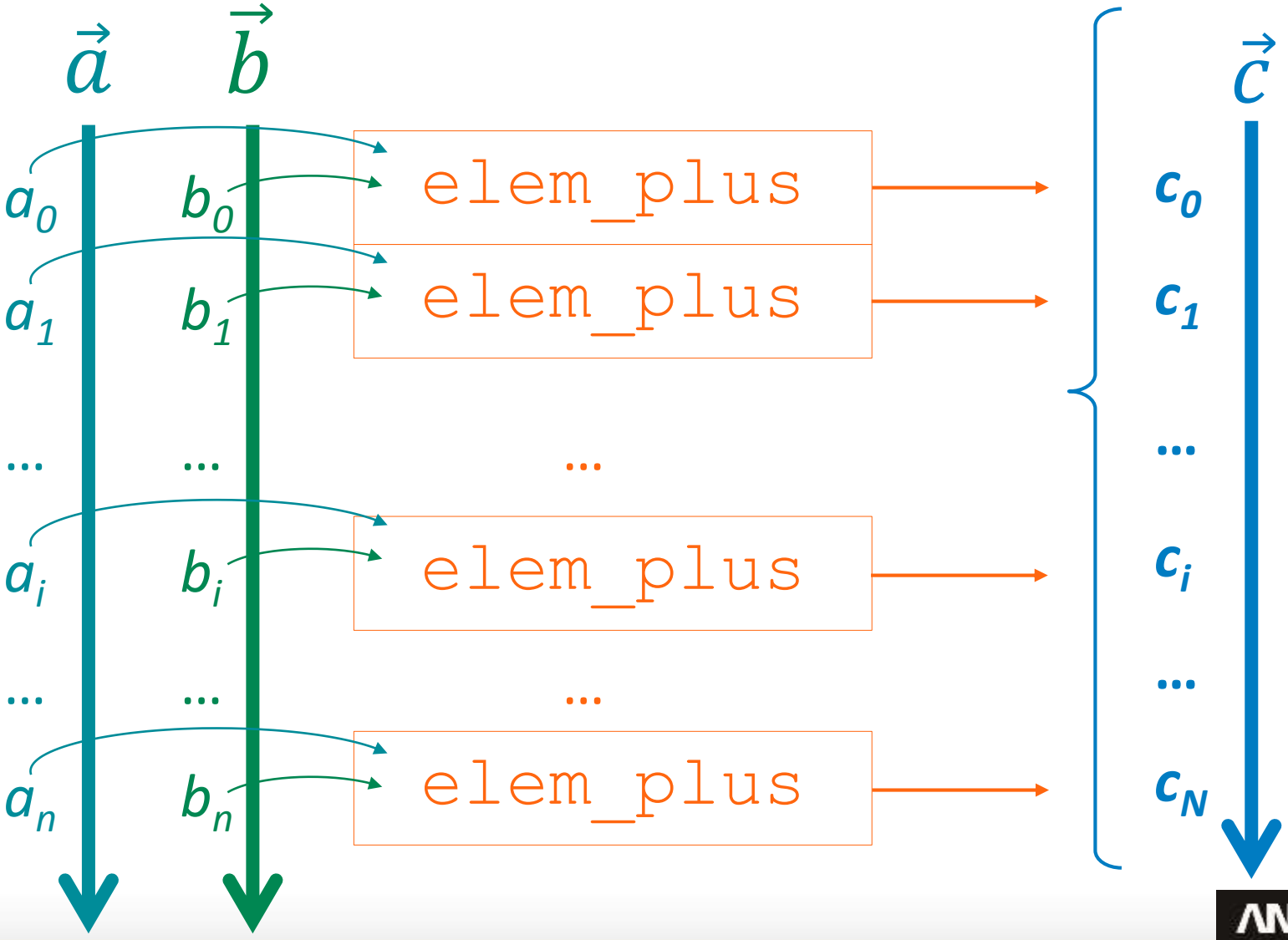
Create a new operator: elem_plus

- Inputs: a_i, b_i:'T
- Output: c_i:'T
- 'T is numeric

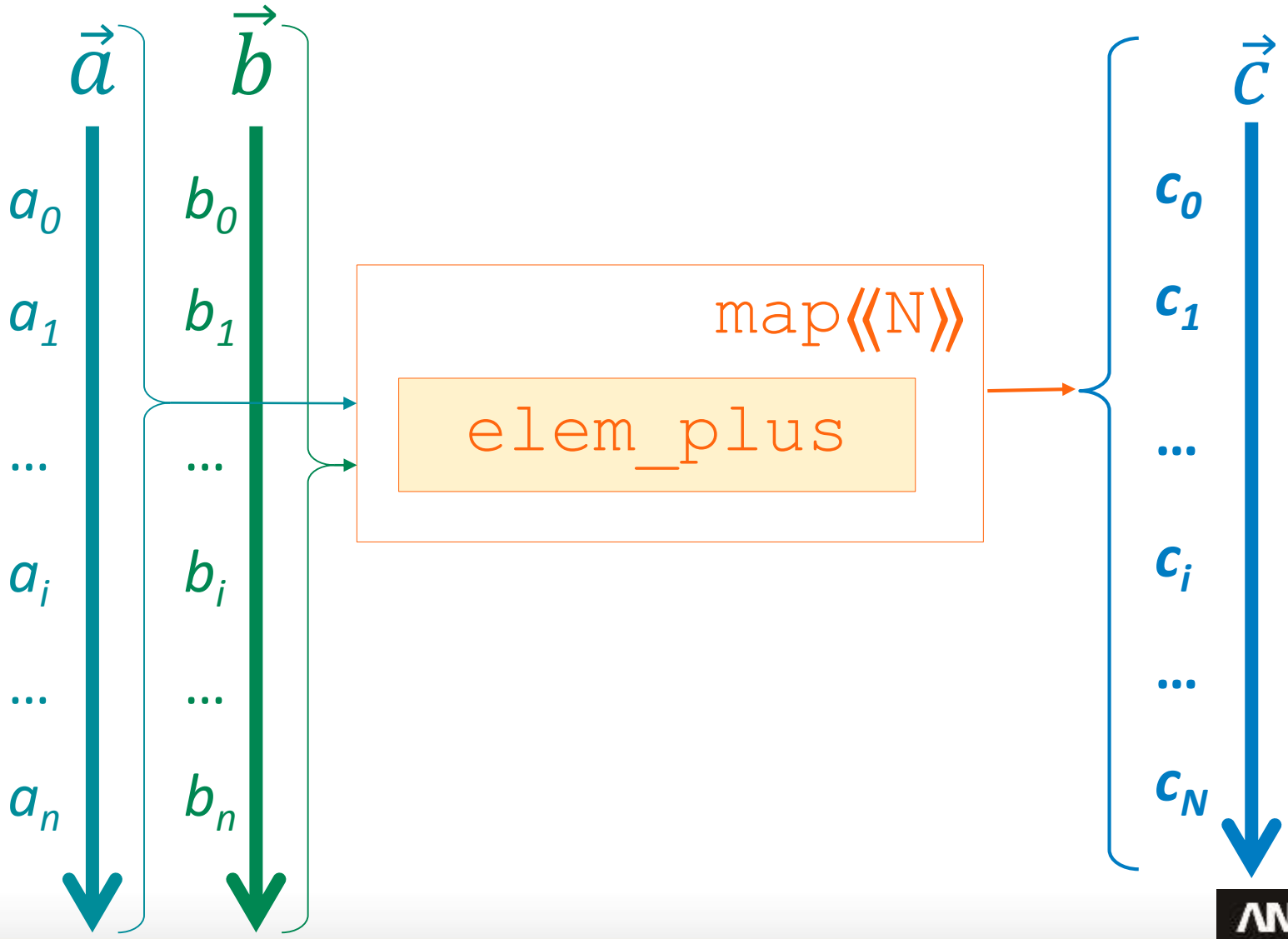




Pointwise Application of `elem_plus`



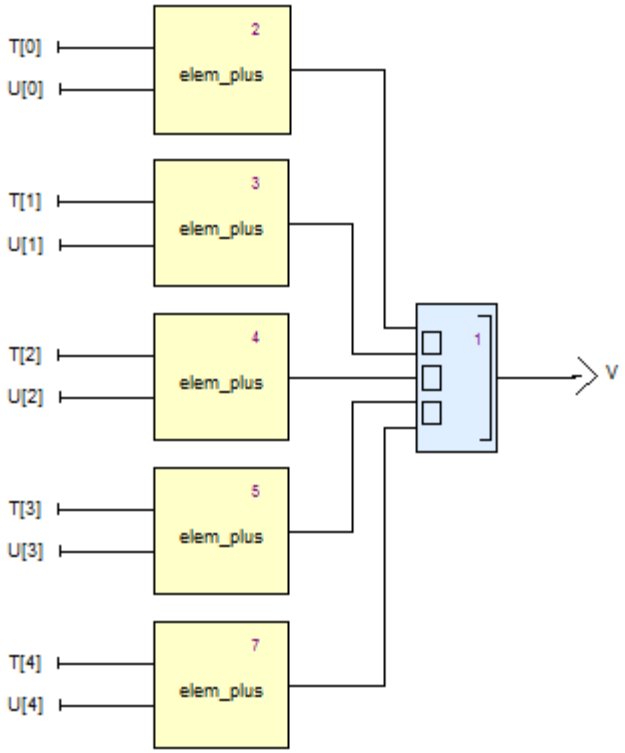
Pointwise Application of `elem_plus`



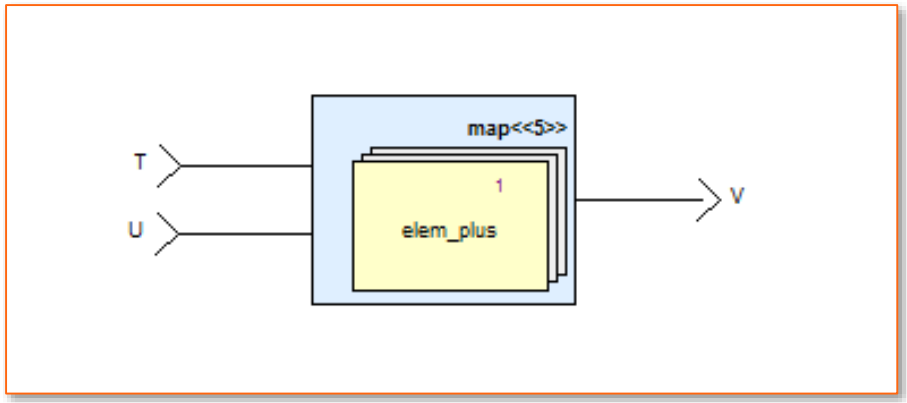
Pointwise Application of `elem_plus`

Example

Equivalent expanded model



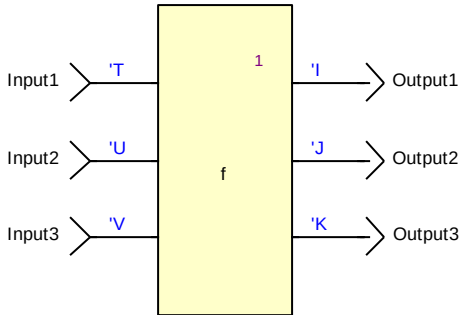
T, U and V : array of 5 integers



Map Signature

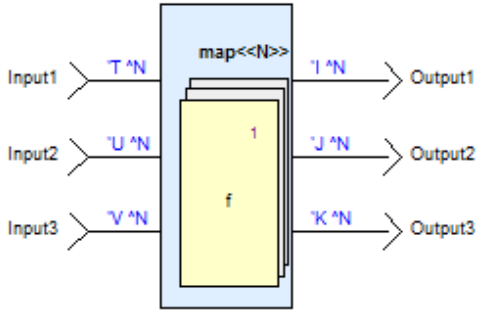
f is a function of arbitrary signature:

$$
\begin{array}{c} \text{'T} \\ \text{'U} \\ \dots \\ \text{'V} \end{array} \longrightarrow \begin{array}{c} \text{'I} \\ \text{'J} \\ \dots \\ \text{'K} \end{array}
$$



Then the pointwise application of f over arrays of size N, *map⟪N⟫(f)* has for signature:

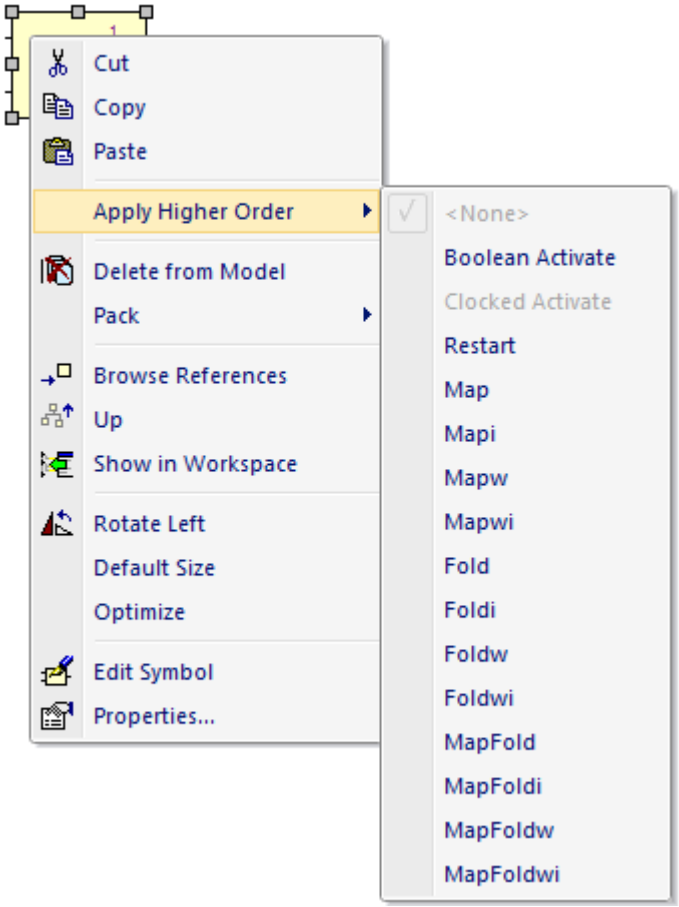
$$
\begin{array}{c} \text{'T\^{}N} \\ \text{'U\^{}N} \\ \dots \\ \text{'V\^{}N} \end{array} \longrightarrow \begin{array}{c} \text{'I\^{}N} \\ \text{'J\^{}N} \\ \dots \\ \text{'K\^{}N} \end{array}
$$

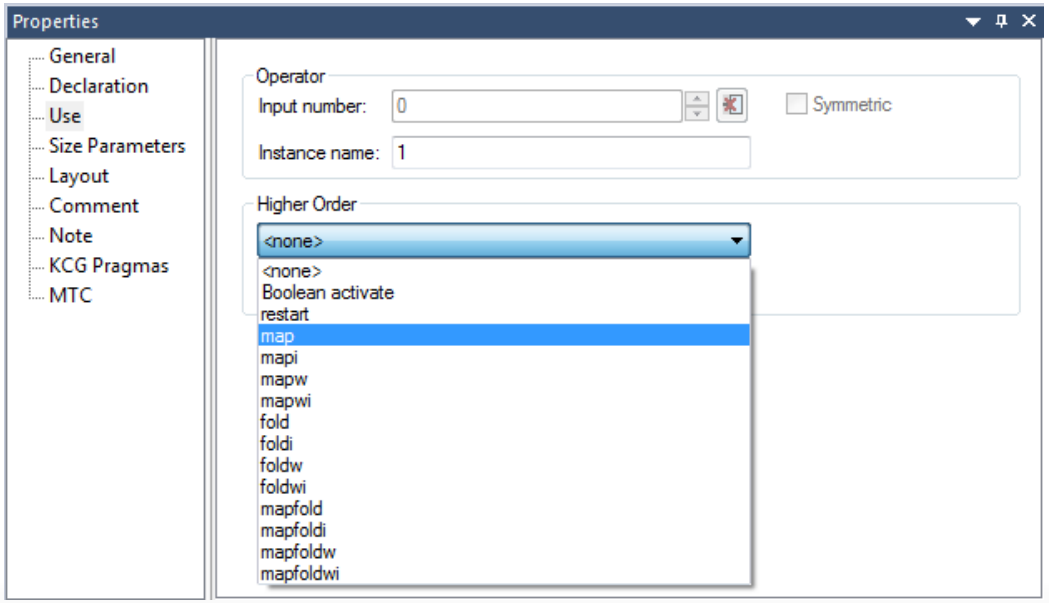


Applying Iterators

Iterators can be applied using

- A contextual menu
- The properties dialog of an operator





Exercise 2

Exercise

Objective:

Use the map iterator

Requirements:

Time: 10 min

Implement the following operators:

```
vec_add (a,b:'T^N) returns (c:'T^N)
   ∀ i ∈ [0..N-1], ci = ai + bi
vec_incr (a:'T^N,incr:'T) returns (b:'T^N)
   ∀ i ∈ [0..N-1], bi = ai + incr
```

In all these cases, 'T is numeric

Exercise 2: Solution

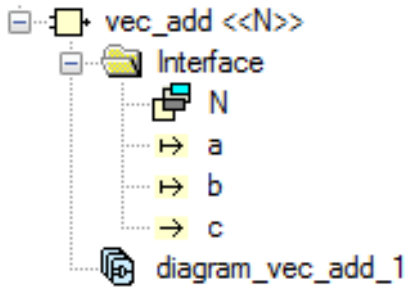
Exercise

Open the Exercises project

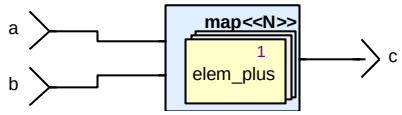
Create a new operator: vec_add

- Parameter: N
- Inputs: a, b:'T^N
- Output: c:'T^N

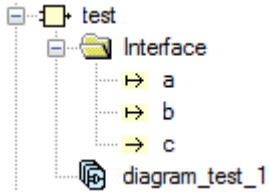
'T is numeric

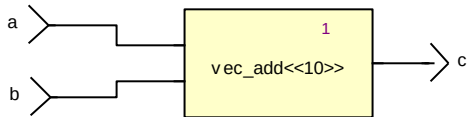


Instantiate elem_plus operator and apply a map order N (double click on operator under map, and replace 1 by N)



Create a "test" operator, where you instantiate vec_add for arrays of 10 floating values



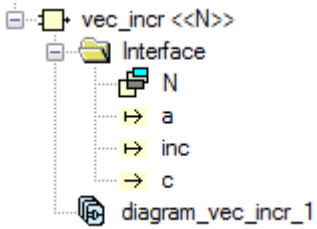


Exercise 2: Solution

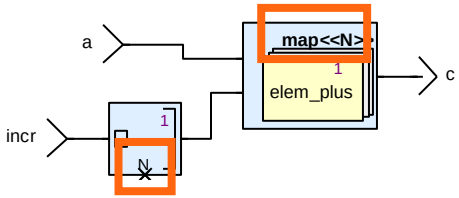
Exercise

Create a new operator: vec_incr

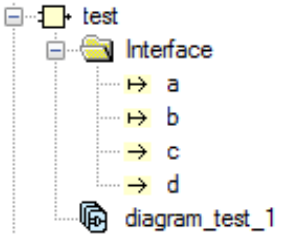
- Parameter: N
- Inputs: a:'T^N, inc:'T
- Output: c:'T^N
- 'T is numeric

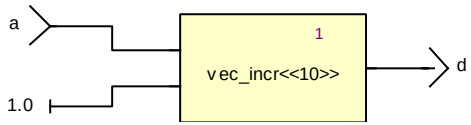


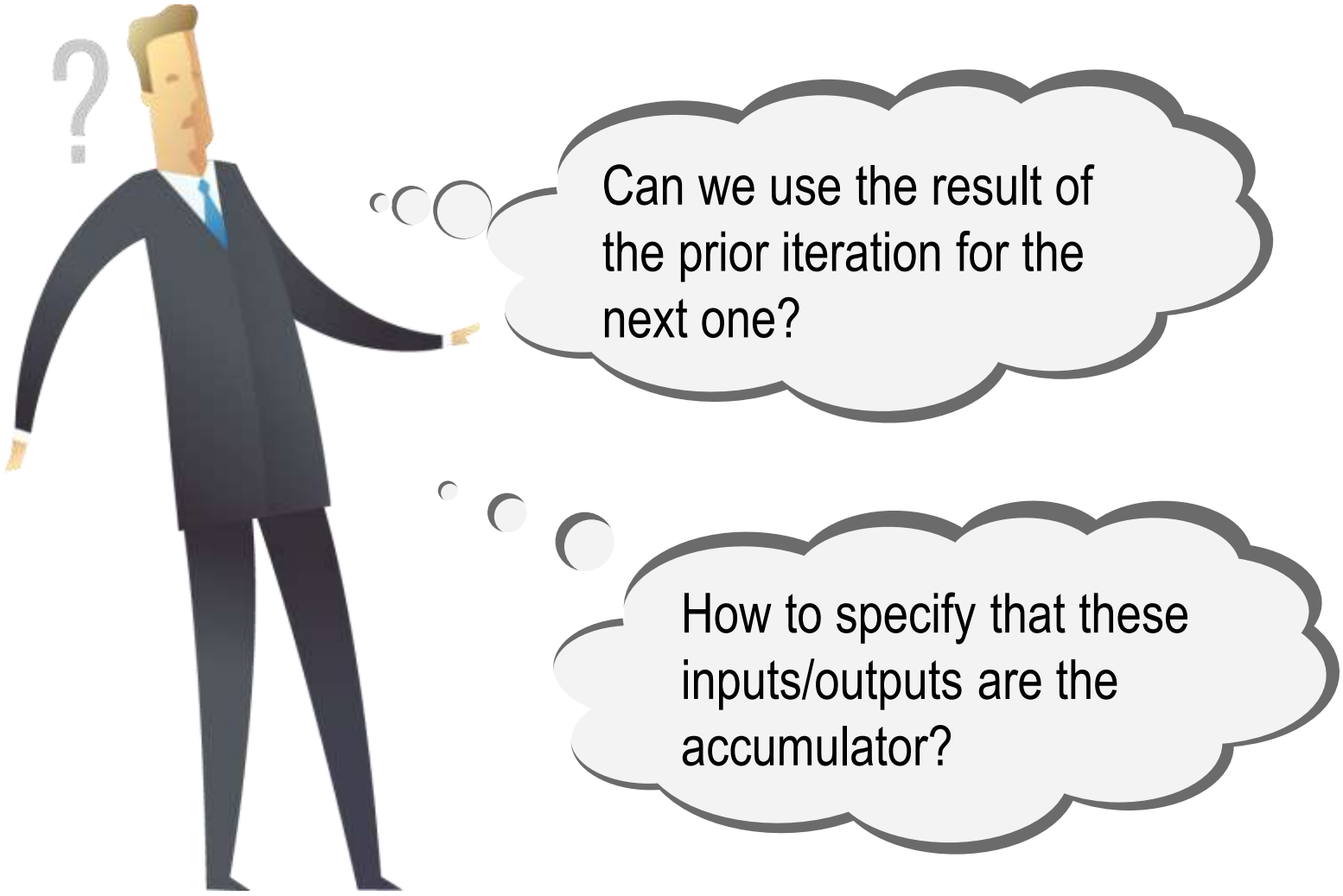
Instantiate elem_plus operator and apply a map order N



Add an output to test operator, and instantiate elem_plus for array of 10 floating values, run and play with the simulator

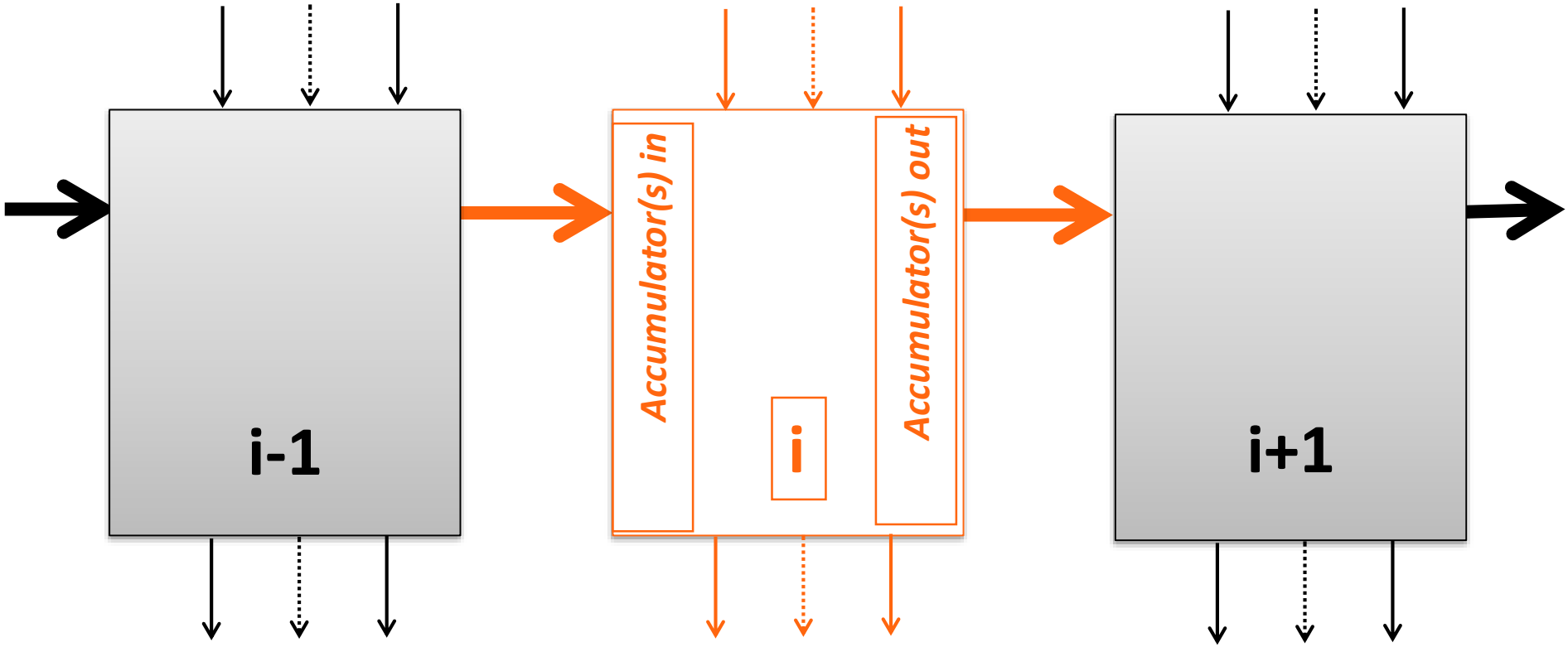





Can we use the result of the prior iteration for the next one?
How to specify that these inputs/outputs are the accumulator?

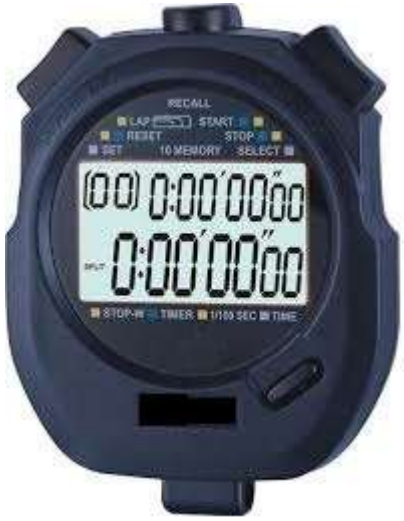
What do we Need?

Accumulator(s) to carry data from one iteration to the next one



What do we Need?

A way to model a stopwatch:



We need to use a carry to increment seconds, minutes, and hours

Exercise 3

Exercise

Objective:

Define the modCount operator

Requirements:

Time: 10 min

Implement the following modCount operator:

- Increment (+1) *count* on each *incr* until *modulo* is reached
- When *modulo* is reached, restarts from zero and outputs *rippleClock*

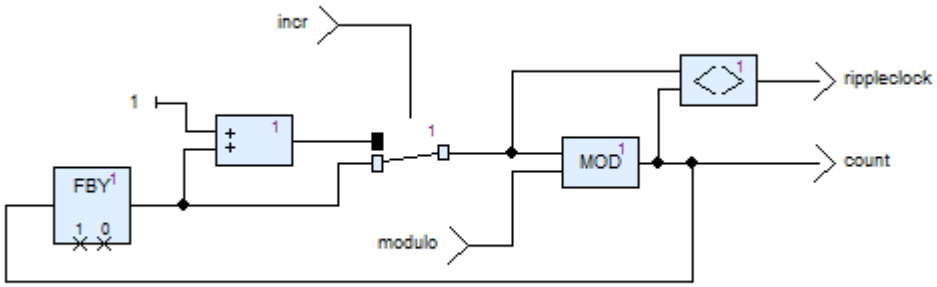
Exercise 3: Solution

Exercise

Open the Exercises project

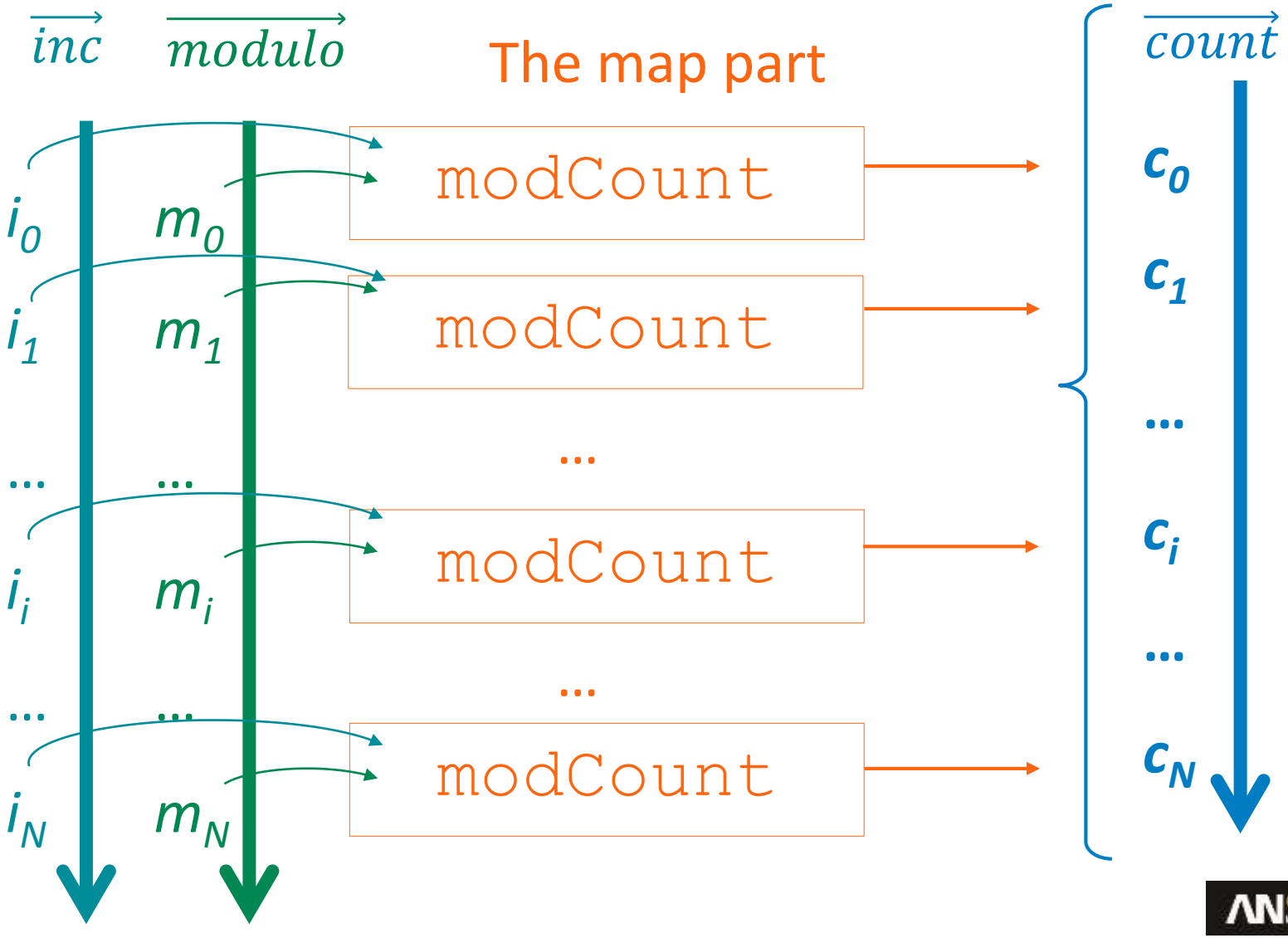
Create a new operator: modCount

- Inputs: incr: bool
  modulo: integer
- Outputs: rippleclock: bool
  count: integer



Pointwise Application of `modCount`

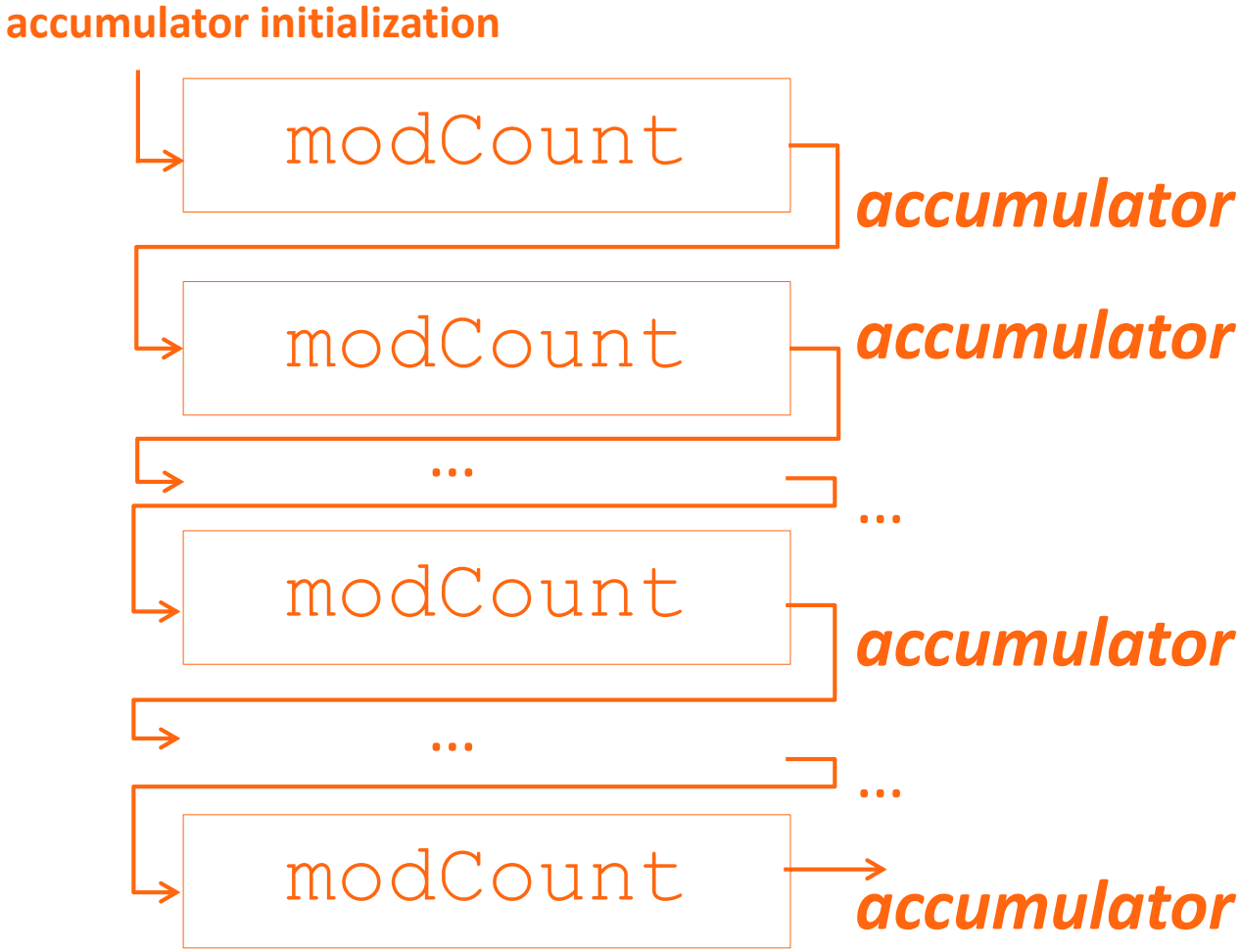
Example



Pointwise Application of `modCount`

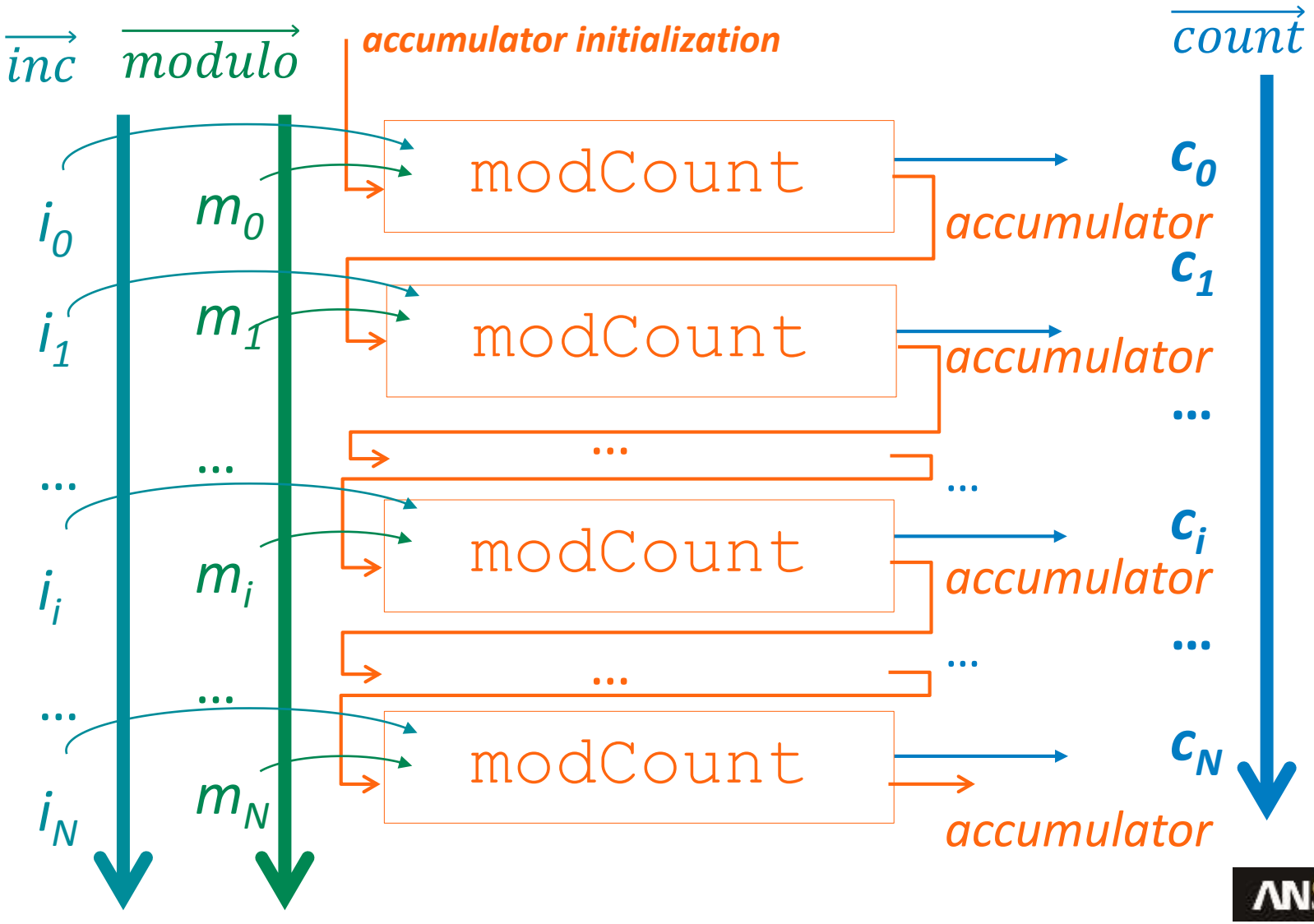
Example

The accumulator part



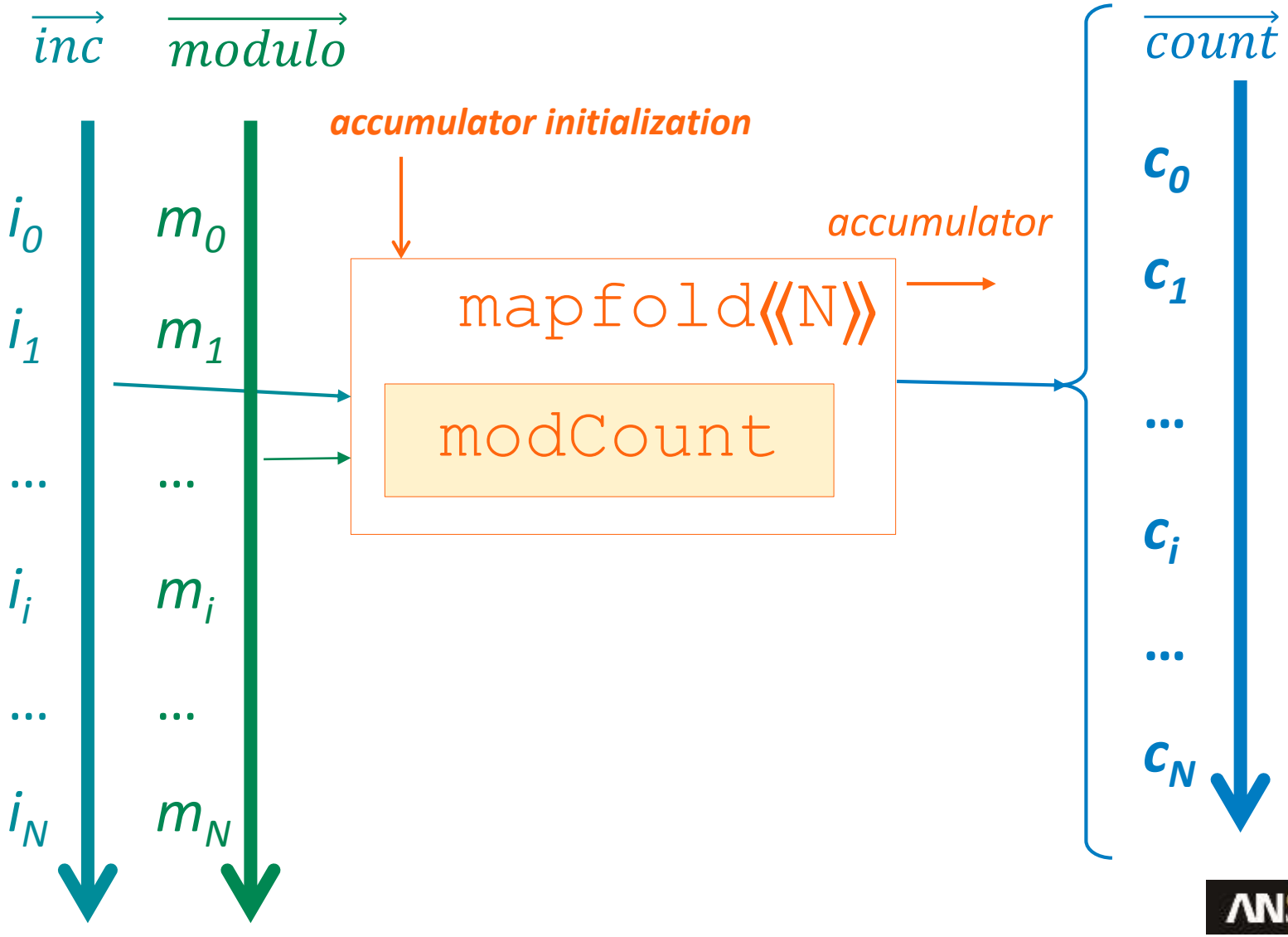
Pointwise application of `modCount`

Example

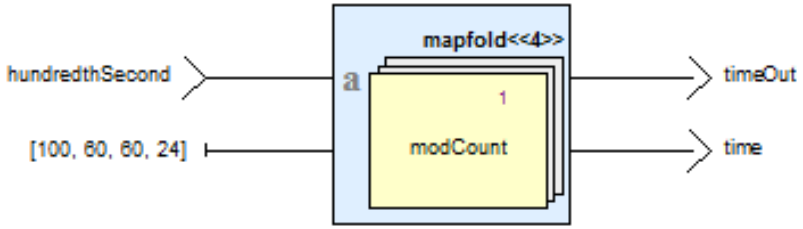


Pointwise application of `modCount`

Example



Pointwise application of `modCount`



`time[0]`: hundredth of second

`time[1]`: seconds

`time[2]`: minutes

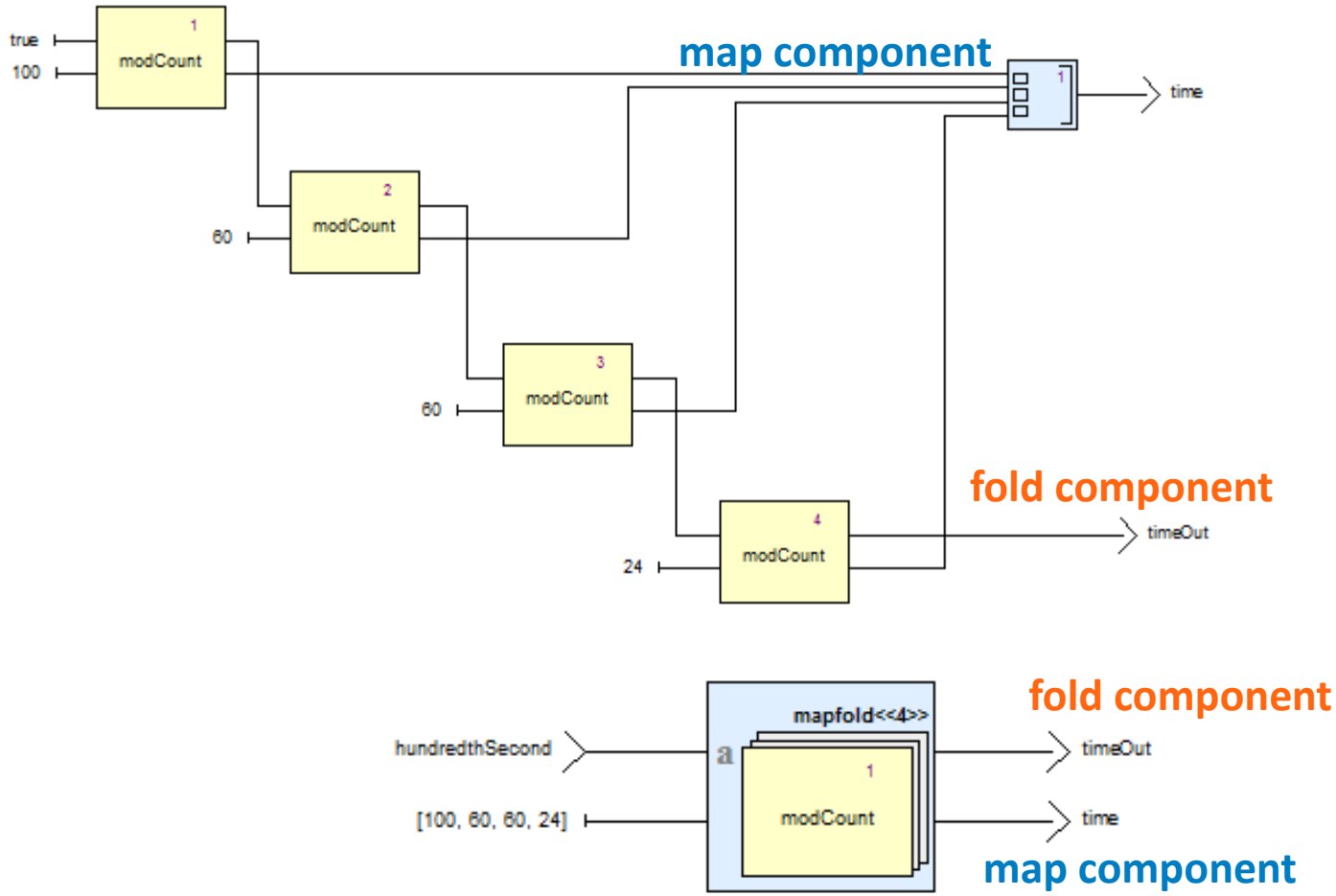
`time[3]`: hours

Example

Pointwise Application of ModCount

Example

Equivalent expanded model

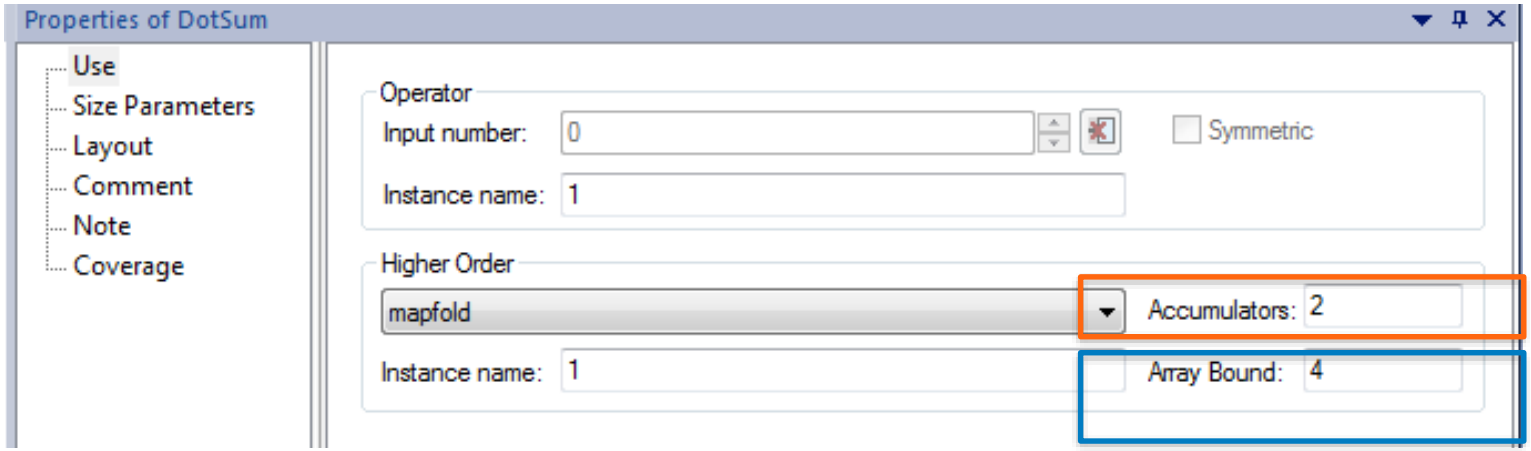


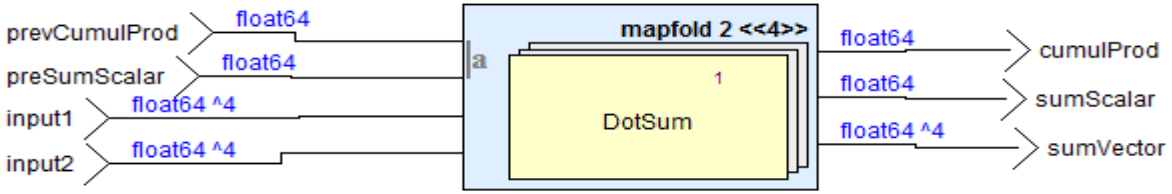
Mapfold

Any number of accumulators (>=0)

- Behaves as a map for accumulator = 0

Array bound

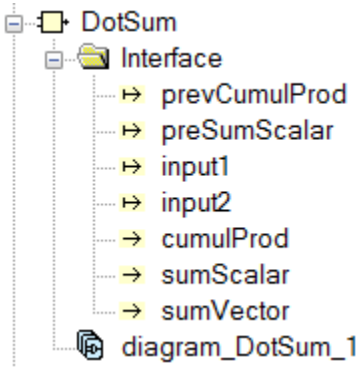


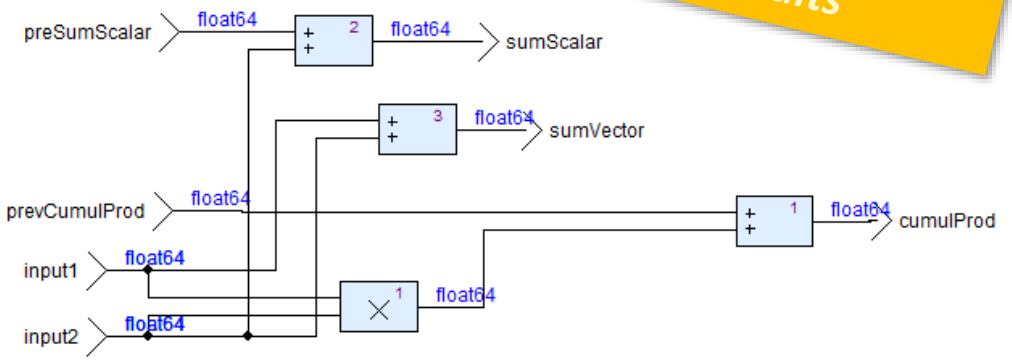


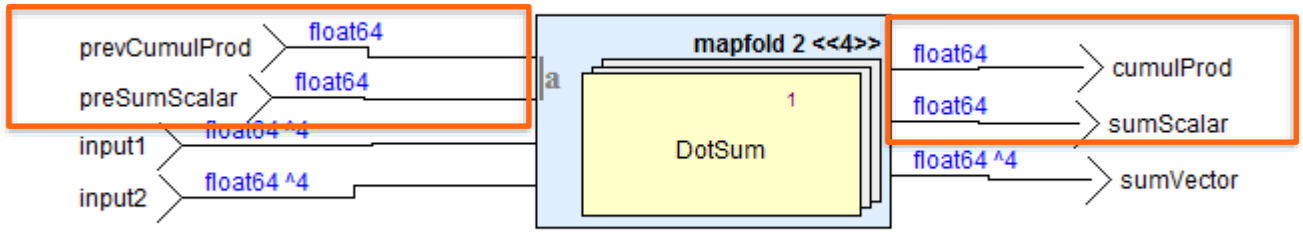
Mapfold: 2 Accumulators

Example

The first inputs are the initialization values of accumulators
The first outputs are the accumulated results







Exercise 4

Exercise

Objective:

Apply the mapfold iterator

Time: 10 min

Requirements:

Apply mapfold on modCount operator to implement the stopwatch

Exercise 4: Solution

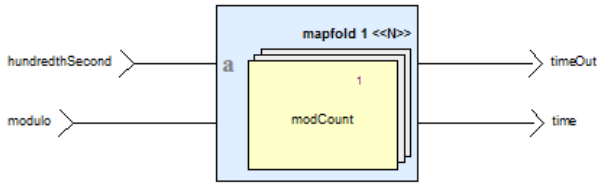
Exercise

Open the Exercises project

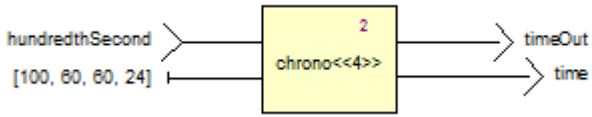
Create a new operator: Chrono

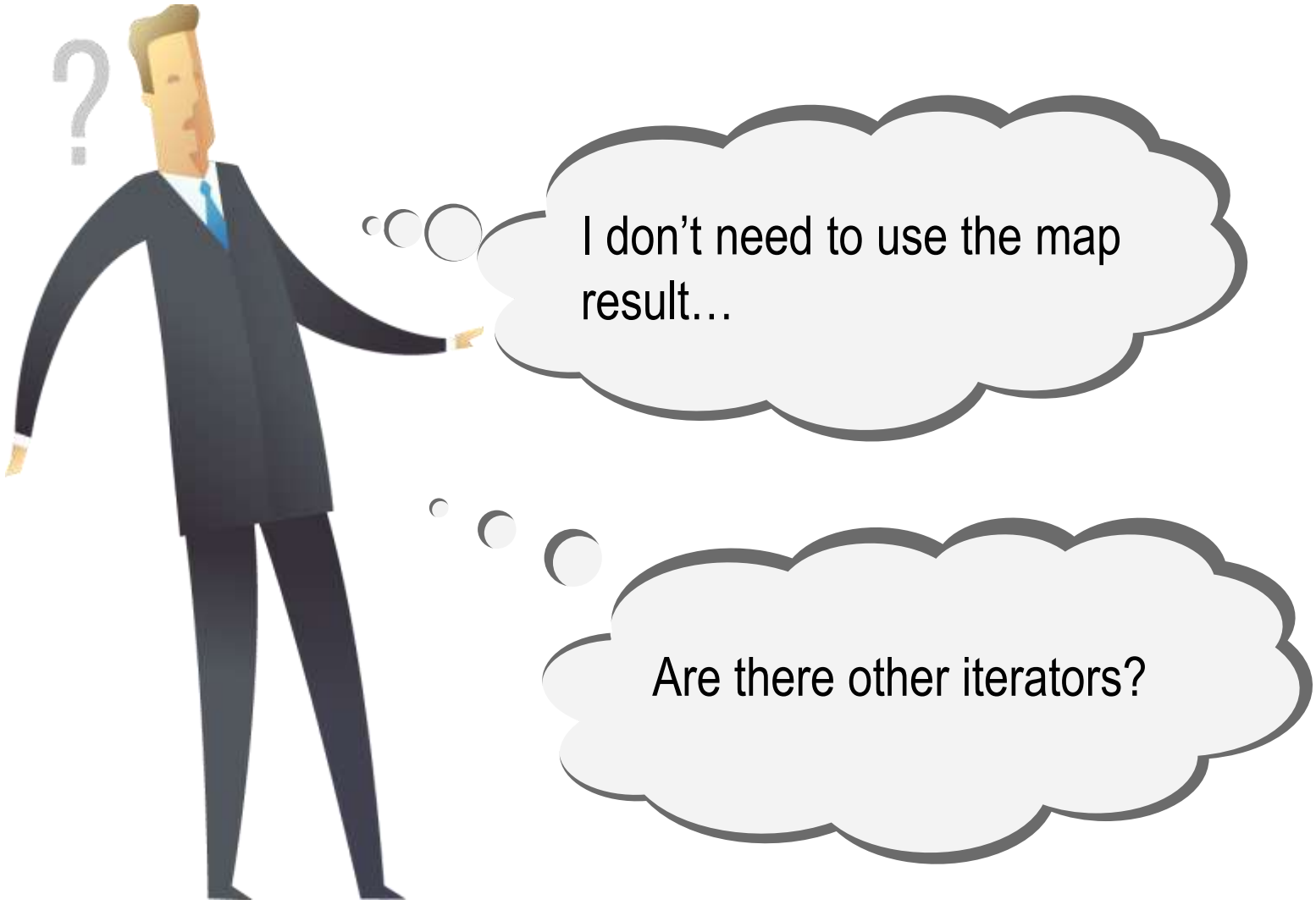
- Parameter: N
- Inputs: hundredthSecond: bool, modulo: int32^N
- Outputs: timeOut: bool, time: int32^N

Instantiate modCount operator and apply a mapfold order N



Add inputs and outputs to the “test” operator and instantiate Chrono




I don’t need to use the map result…
Are there other iterators?

What do we Need?

A way to write

$$a.b = \sum_{i=1}^{N} a_i.b_i$$

Pointwise application of . to $\vec{a}$ and $\vec{b}$ vectors

Fold

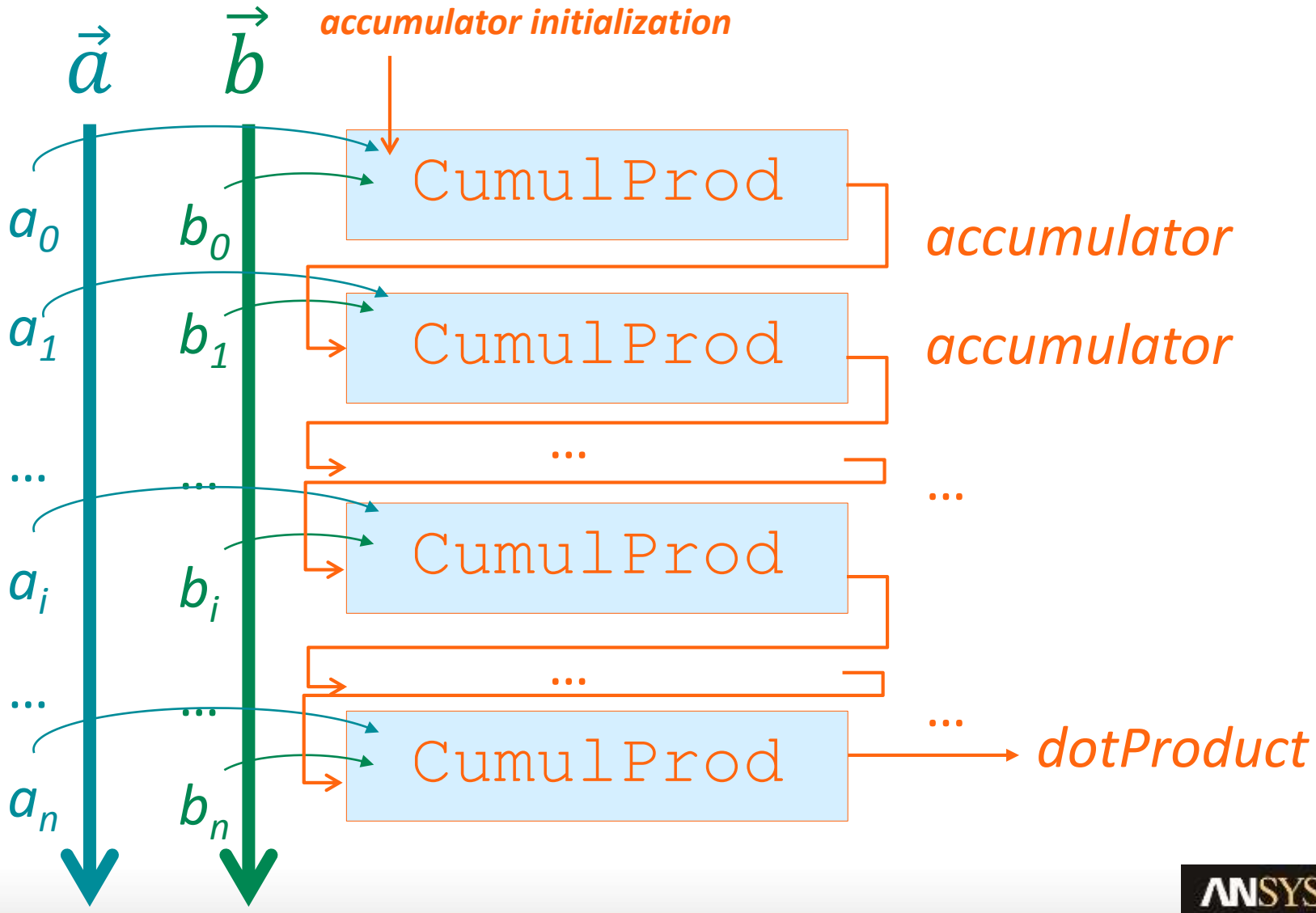
Let f be a function of arbitrary signature :

$$'Accumulator \times 'T \times 'U \times \ldots \times 'V \rightarrow 'Accumulator$$

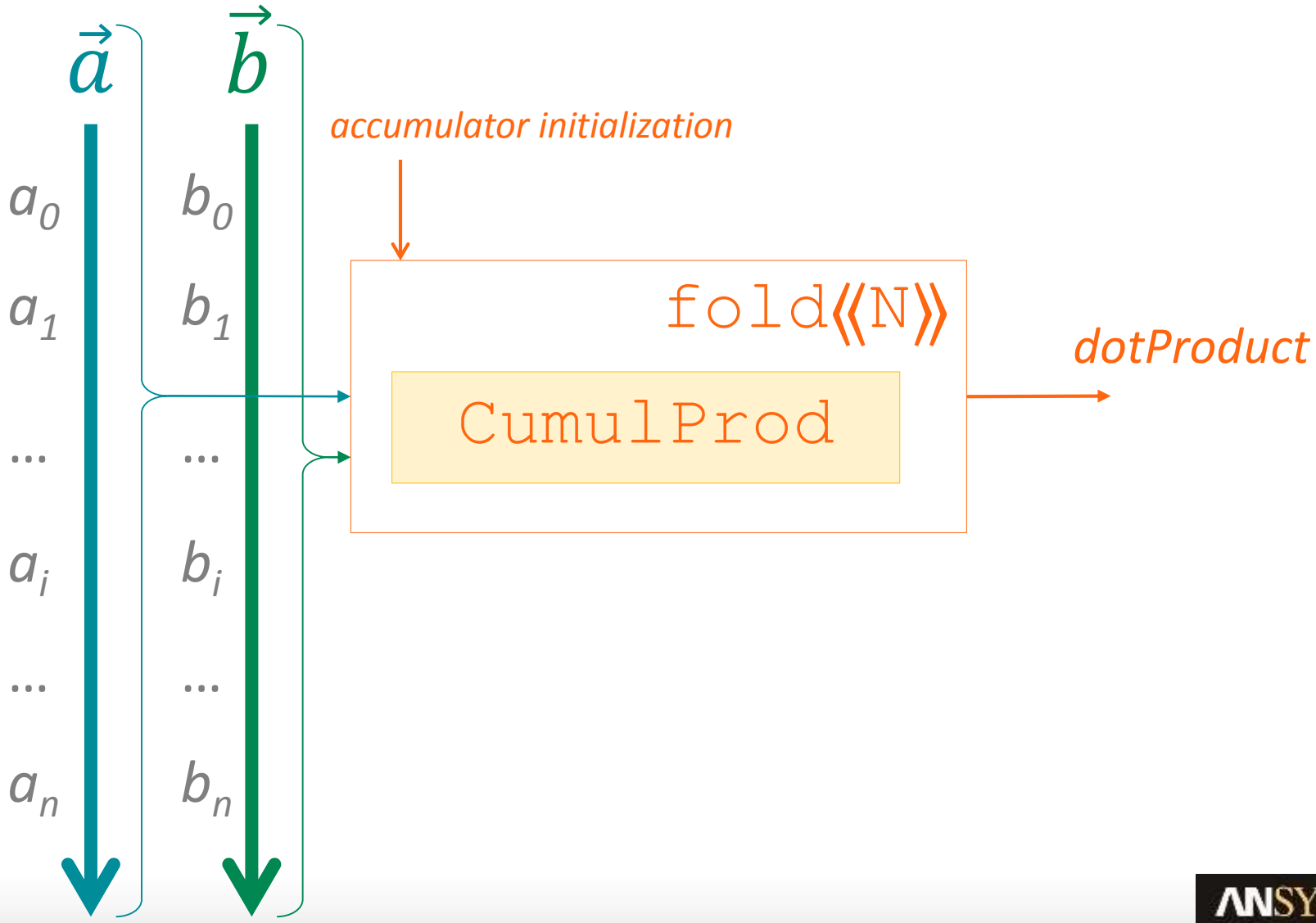
Then the pointwise application of f over arrays of size N, fold⟪N⟫(f) has for signature:

$$'Accumulator \times 'T\hat{}N \times 'U\hat{}N \times \ldots \times 'V\hat{}N \rightarrow 'Accumulator$$

Pointwise Application of `CumulProd`



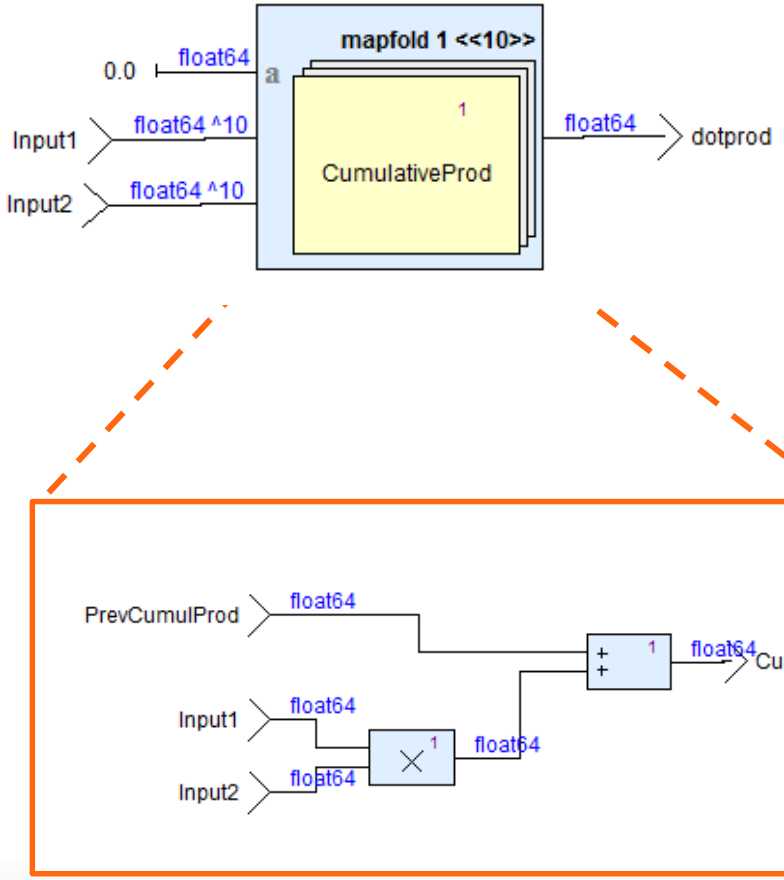
Pointwise Application of `CumulProd`



The Dot Product: fold => mapfold

Only the accumulator result is useful:

Use directly mapfold



Lab 7: Accumulator (1/3)

Lab Support p.40-43

Objective:

Create a `min` operator with a mapfold, in the median design environment

Time: 10 min

Requirements:

Create an operator that provides the minimal numeric value between two inputs.

Instantiate the `min` operator in the median operator to provide the minimal numeric value of an input vector.

```
min<<N>>(v:'T^N) returns (m:'T)
```

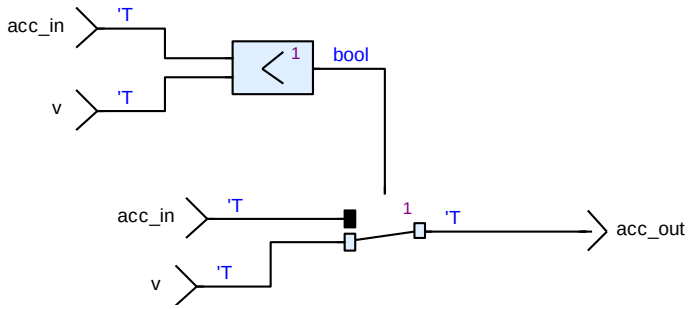
'T is numeric

Lab

Lab 7: Accumulator (2/3)

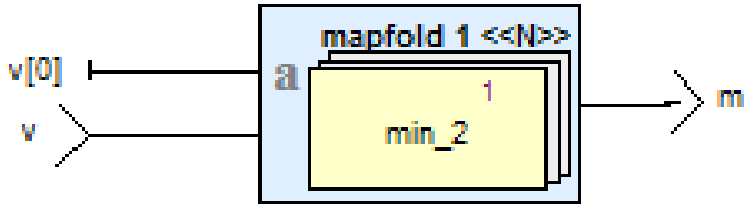
Design a `min_2` operator

Name	Kind	Type
`Acc_in`	input	‘T
`v`	input	‘T
`m`	output	‘T



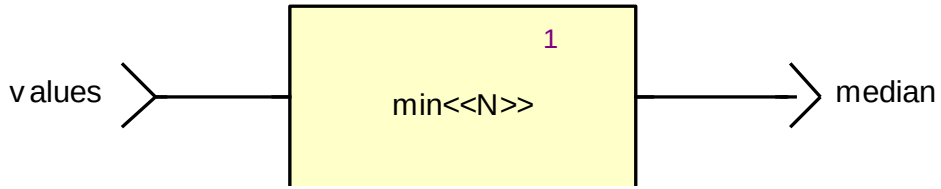
Design the `min` operator

Name	Kind	Type
`v`	input	‘T^N
`m`	output	‘T



Lab 7: Accumulator (3/3)

Udpate the `median` operator



Lab

Lab 8: Accumulator

Lab Support p.45-47

Objective:

Complete the useful operators for the median computation.

Requirements:

Time: 20 min

Create the following operators (use **mapfold**):

Find the maximal numeric value between two values

- `max(acc_in:'T,a:'T) returns (acc:'T)`

Count the number of values greater than the reference value in an input vector

- `Count_gt(acc_in: uint8, ref:'T, a:'T^N) returns (acc:uint8)`

Count the number of values lower than the reference value in an input vector

- `Count_lt(acc_in:uint8, ref:'T, a:'T^N) returns (acc:uint8)`

'T is numeric

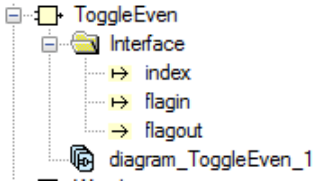
Lab

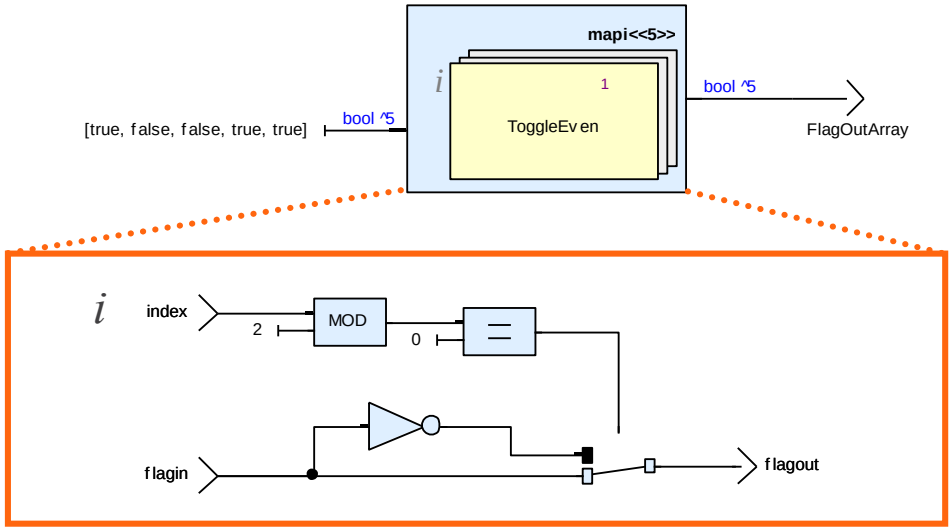
The mapi Iterator

It is the same iterator as map but with an index of iteration passed as the first input of the mapped operator (in the order of the interface).

The first input of the mapped operator is the index:

Example: toggle Booleans in even cells.





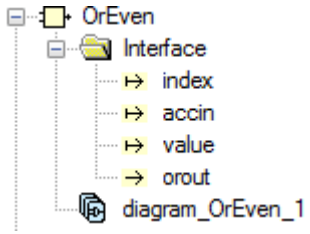
`[true,false,false,true,true]` → `[false,false,true,true,false]`

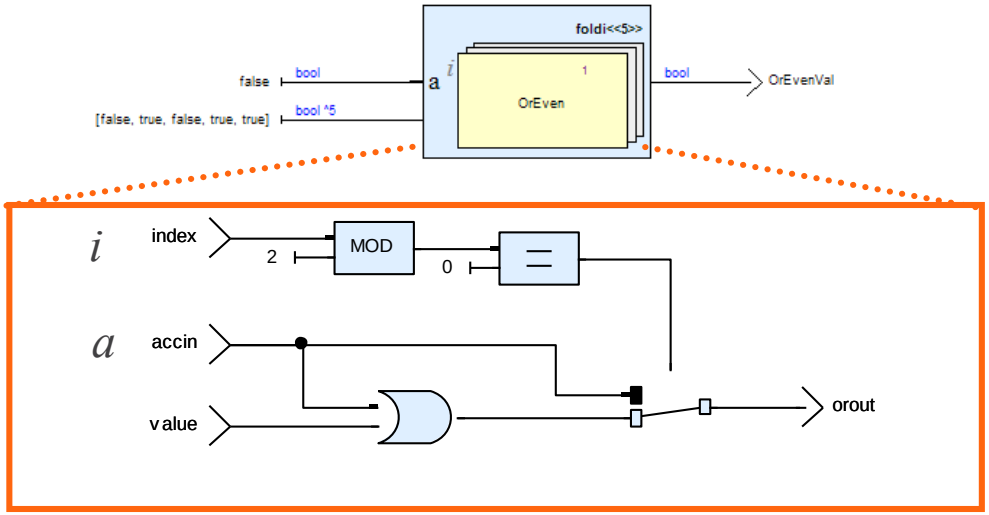
The foldi Iterator

Same as the fold iterator, with the iteration index passed as the first input of the mapped operator (in the order of the interface).

The accumulator is passed as the second input:

Example: logical OR between even cells.





`[false,true,false,true,true]` → `true`

The mapw/mapwi Iterator

Apply the map over an operator under a condition.

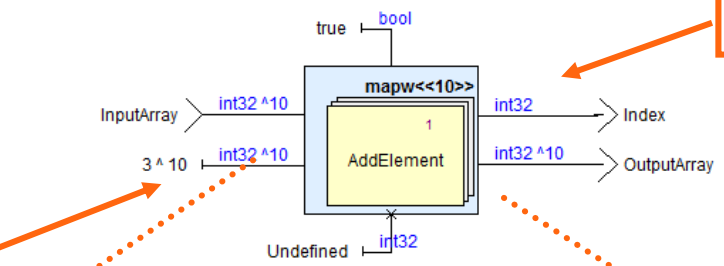
The condition may depend on the index, accessible on the first input with the mapwi.

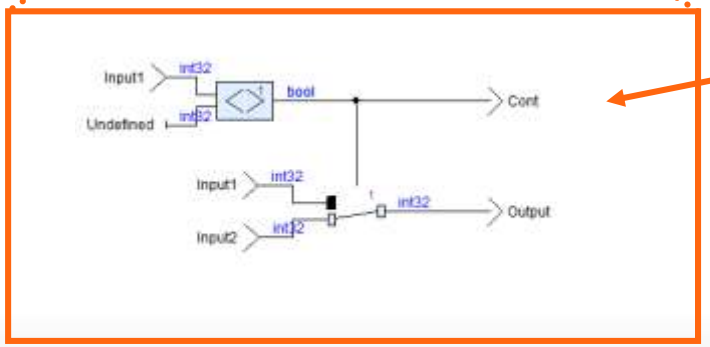
Example: add 3 at the end of a list.

Exit index for the first unprocessed iteration

Default value used to fill the end of the array when the iteration is interrupted

Condition for the continuation of the iteration



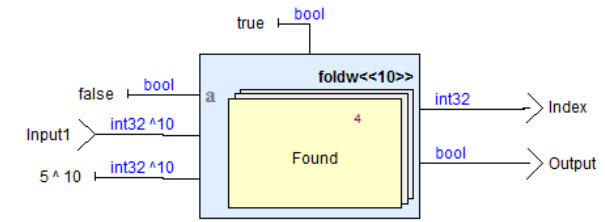


The foldw/foldwi Iterator

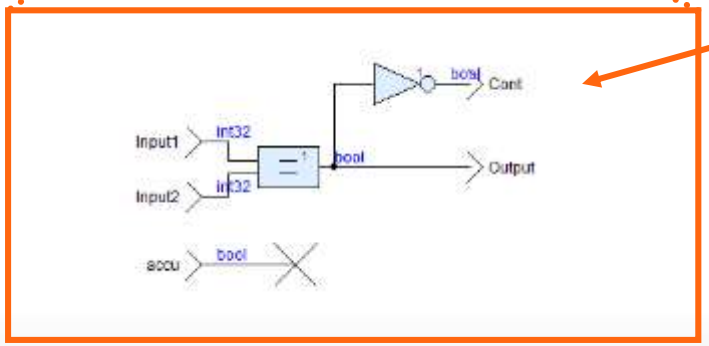
Apply the fold over an operator under a condition.

The condition may depend on the index, accessible on the first input with foldwi.

Example: search a value (5) in an array.



Exit index for first unprocessed iteration



condition for the continuation of the iteration

To Summarize

The different iterators are…

```
map
fold
mapfold
mapi/w/wi
foldi/w/wi
mapfoldi/w/wi
```

`map`

Apply to an operator on a vector of inputs producing a vector of outputs

`fold`

Apply to an operator on a vector of inputs producing a single output as accumulated results of the operator calls

`Partial iterators (w)`

- Generated code can be not optimized
- Exit from a bounded loop does not provide some benefit for the Worst Case Execution Time

Lab 9: Torben Algorithm

Lab Support p.49-61

Objective:

Complete the median design

Requirements:

Time: 25 min

Use the Torben algorithm to complete the design of the median

Lab

Lab 9: Prerequisite

Lab
Prerequisite

http://ndevilla.free.fr/median/median/index.html

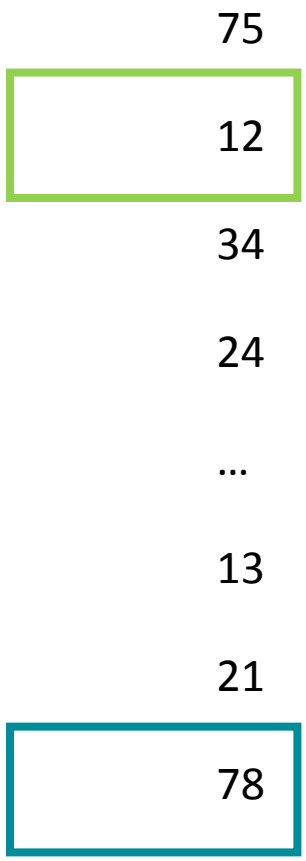
Median filtering is a commonly used technique in signal processing. Typically used on signals that may contain outliers skewing the usual statistical estimators, it is usually considered too expensive to be implemented in real-time or CPU-intensive applications.

In a safety environment, the worst case execution time (WCET) is more important than the best execution time. Also, we prefer to avoid the large memory copy even if the algorithm is slower.

ANSYS

Torben's Algorithm

1. Start by finding max and min values
2. And make a guess : the median should be the mean between this min and max value : $\frac{min+max}{2}$
3. min = 12, max = 78, guess = 45



Lab Prerequisite

Torben's Algorithm

1. min and max represent the minimal and maximal value of the algorithm.
2. We now count the number of values that are lower than our guess ant the number that are above our guess :
   5 are smaller than 45, 2 are above
3. This means that our median is between our guess and the previous min value: we can use our guess as the new max value. We can even improve this by using the value immediately below our guess as the new max (as our max is possibly not present in our lis)
4. The value immediately below 45 is 34, this becomes our new max

Lab Prerequisite

Torben's Algorithm

1. Our new guess is then (12+34)/2 = **23**
2. below = 3, above = 4
3. The definition of the median is to have as many above than below. So we know in this case that our median is the value immediately above our guess, that is **24**
4. This is an n logn algorithm

Lab Prerequisite

Lab 9: Prerequisite (C Code)

```c
/*
 * The following code is public domain.
 * Algorithm by Torben Mogensen, implementation by N. Devillard.
 * This code in public domain.
 */

#include "median_c.h"

elem_type torben(elem_type *m, size_array n)
{
    int         i, less, greater, equal;
    elem_type  min, max, guess, maxltguess, mingtguess;

    min = max = m[0] ;
    for (i=1 ; i<n ; i++) {
        if (m[i]<min) min=m[i];
        if (m[i]>max) max=m[i];
    }

    while (1) {
        guess = (min+max)/2;
        less = 0; greater = 0; equal = 0;
        maxltguess = min ;
        mingtguess = max ;
        for (i=0; i<n; i++) {
            if (m[i]<guess) {
                less++;
                if (m[i]>maxltguess) maxltguess = m[i] ;
            } else if (m[i]>guess) {
                greater++;
                if (m[i]<mingtguess) mingtguess = m[i] ;
            } else equal++;
        }
        if (less <= (n+1)/2 && greater <= (n+1)/2) break ;
        else if (less>greater) max = maxltguess ;
        else min = mingtguess;
    }
    if (less >= (n+1)/2) return maxltguess;
    else if (less+equal >= (n+1)/2) return guess;
    else return mingtguess;
}
```

min and max initialization

guess computation

counting the distribution around guess value, select new min and max

stop condition

final median value

Lab 9: Prerequisite (C Code)

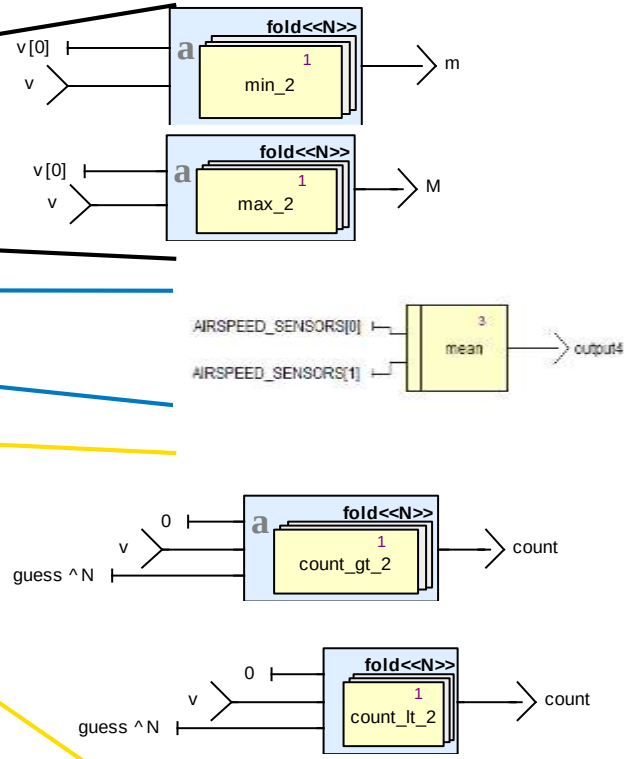
Lab Prerequisite

C code algorithm

```
 * The following code is public domain.
 * Algorithm by Torben Mogensen, implementation by N. Devillard.
 * This code in public domain.
 */

#include "median_c.h"

elem_type torben(elem_type *m, size_array n)
{
    int         i, less, greater, equal;
    elem_type  min, max, guess, maxltguess, mingtguess;

    min = max = m[0] ;
    for (i=1 ; i<n ; i++) {
        if (m[i]<min) min=m[i];
        if (m[i]>max) max=m[i];
    }

    while (1) {
        guess = (min+max)/2;
        less = 0; greater = 0; equal = 0;
        maxltguess = min ;
        mingtguess = max ;
        for (i=0; i<n; i++) {
            if (m[i]<guess) {
                less++;
                if (m[i]>maxltguess) maxltguess = m[i] ;
            } else if (m[i]>guess) {
                greater++;
                if (m[i]<mingtguess) mingtguess = m[i] ;
            } else equal++;
        }
        if (less <= (n+1)/2 && greater <= (n+1)/2) break ;
        else if (less>greater) max = maxltguess ;
        else min = mingtguess;
    }
    if (less >= (n+1)/2) return maxltguess;
    else if (less+equal >= (n+1)/2) return guess;
    else return mingtguess;
}
```

SCADE Suite model already designed

Use mapfold



Lab 9: Prerequisite (Ada)

Ada

```
function torben(m:Kcg_Types.Array_Float64; n:Integer) return Kcg_Config.Kcg_Float64
is
less, greater, equal: Integer;
min, max : Kcg_Config.Kcg_Float64:=m(0);
guess, maxltguess, mingtguess : Kcg_Config.Kcg_Float64;
flag : Boolean :=true;

begin
  for i in 1..n loop
        if m(i) < min then
                min := m(i);
        end if;
        if m(i) > max then
                max := m(i);
        end if;
  end loop;
        while (flag) loop
                guess := (min + max)/2.0;
                less:= 0; greater:= 0; equal:= 0;
                maxltguess := min;
                mingtguess := max;
                for i in 0..n loop
                        if m(i) < guess then
                                less := less + 1;
                                if m(i) > maxltguess then
                                        maxltguess := m(i);
                                end if;
                        end if;
                        if m(i) > guess then
                                greater := greater + 1;
                                if m(i)< mingtguess then
                                        mingtguess := m(i);
                                end if;
                        end if;
                        if m(i) = guess then
                                equal:= equal + 1;
                        end if;
                end loop;
                if (less <= (n+1)/2 and greater <= (n+1)/2) then flag :=false;
                end if;
                if (less>greater) then
                        max := maxltguess;
                else
                        min := mingtguess;
                end if;
        end loop;
        if (less >= (n+1)/2) then return maxltguess;
        end if;
        if (less+equal >= (n+1)/2) then return guess;
        else return mingtguess;
        end if;
end torben;
```

min and max initialization

guess computation

counting the distribution around guess value, select new min and max

stop condition

final median value

Lab Prerequisite

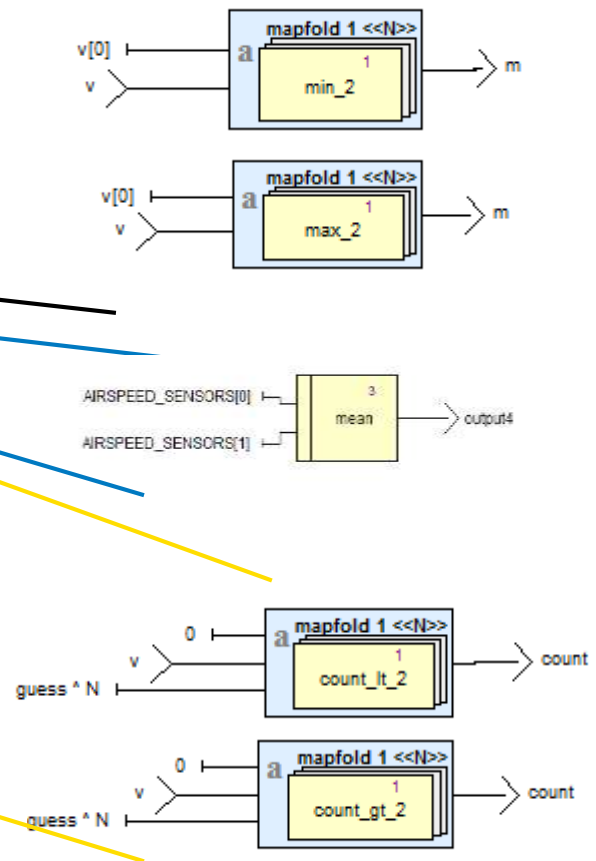
ANSYS

Lab 9: Prerequisite (Ada)

Ada

```
function torben(m:Kcg_Types.Array_Float64; n:Integer) return Kcg_Config.Kcg_Float64
is
less, greater, equal: Integer;
min, max : Kcg_Config.Kcg_Float64:=m(0);
guess, maxltguess, mingtguess : Kcg_Config.Kcg_Float64;
flag : Boolean :=true;

begin
 for i in 1..n loop
       if m(i) < min then
              min := m(i);
       end if;
       if m(i) > max then
              max := m(i);
       end if;
  end loop;
       while (flag) loop
              guess := (min + max)/2.0;
              less := 0; greater := 0; equal := 0;
              maxltguess := min;
              mingtguess := max;
              for i in 0..n loop
                     if m(i) < guess then
                            less := less + 1;
                            if m(i) > maxltguess then
                                   maxltguess := m(i);
                            end if;
                     end if;
                     if m(i) > guess then
                            greater := greater + 1;
                            if m(i)< mingtguess then
                                   mingtguess := m(i);
                            end if;
                     end if;
                     if m(i) = guess then
                            equal:= equal + 1;
                     end if;
              end loop;
              if (less <= (n+1)/2 and greater <= (n+1)/2) then flag :=false;
              end if;
              if (less>greater) then
                     max := maxltguess;
              else
                     min := mingtguess;
              end if;
       end loop;
       if (less >= (n+1)/2) then return maxltguess;
       end if;
       if (less+equal >= (n+1)/2) then return guess;
       else return mingtguess;
       end if;
end torben;
```

SCADE Suite model already designed



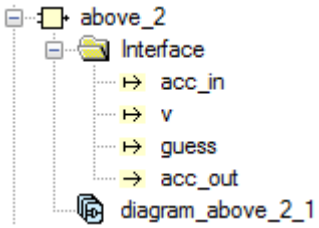
Lab 9: Solution

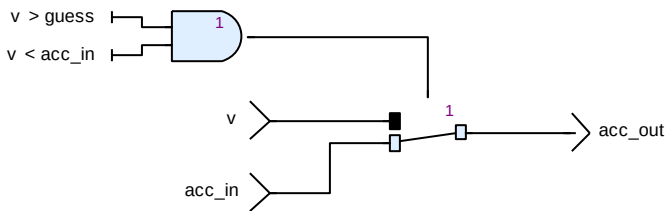
Lab Solution

You need to implement a new min/max selection and the median value finalization

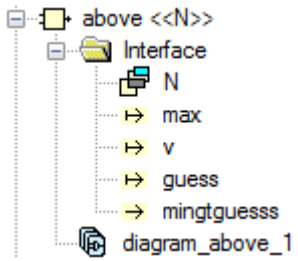
1- Implement “above” operator to define new min/max value for next iteration:

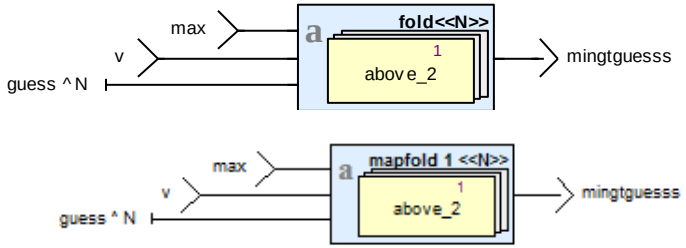
above_2(accIn,v,guess:’T) returns (accOut:’T)





above<<N>>(max:’T,v:’T^N,guess:’T) returns (mingtguess:’T)



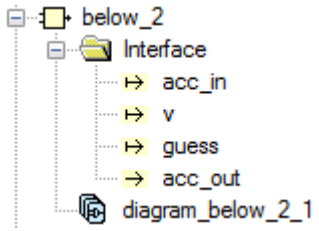


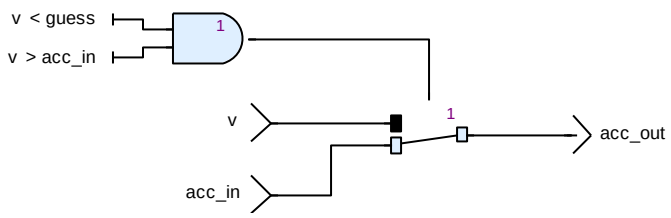
Lab 9: Solution

Lab Solution

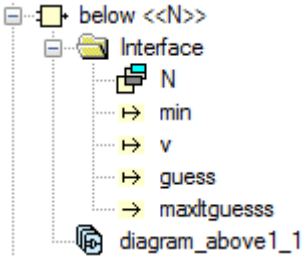
2- Implement the “below” operator to define new min/max value for next iteration:

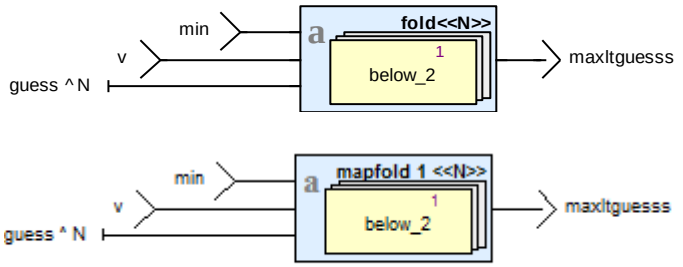
below_2(accIn,v,guess:’T) returns (accOut:’T)





below<<N>>(min:’T,v:’T^N,guess:’T) returns (maxltguess:’T)



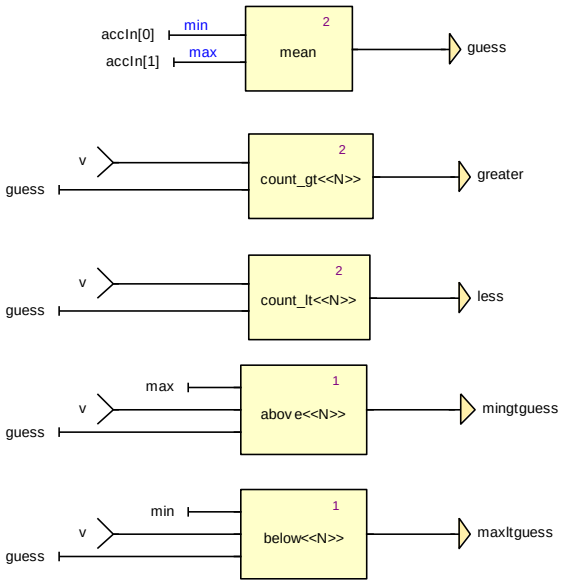


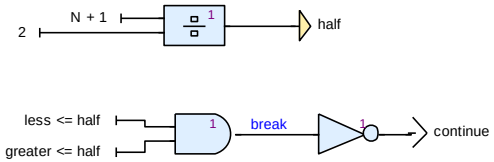
Lab 9: Solution

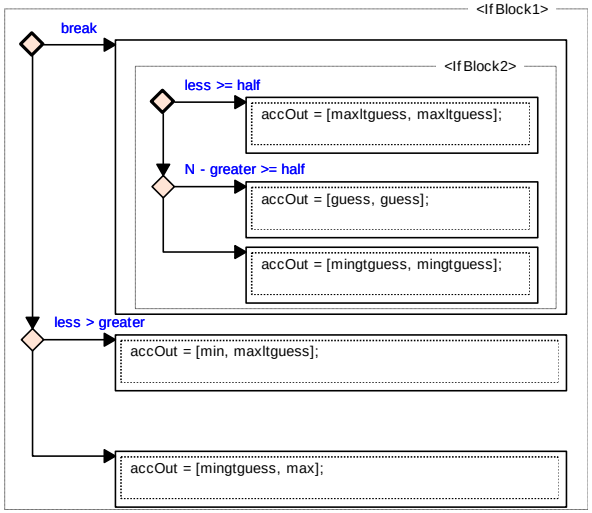
3- Create “guess<<N>>” operator to compute the median.

- This operator is mapfoldw to find the median value
- The accumulator is min and max value, stored in array

guess<<N>>(accIn:’T^2,v :’T^N) returns (continue:bool,accOut:’T^2)





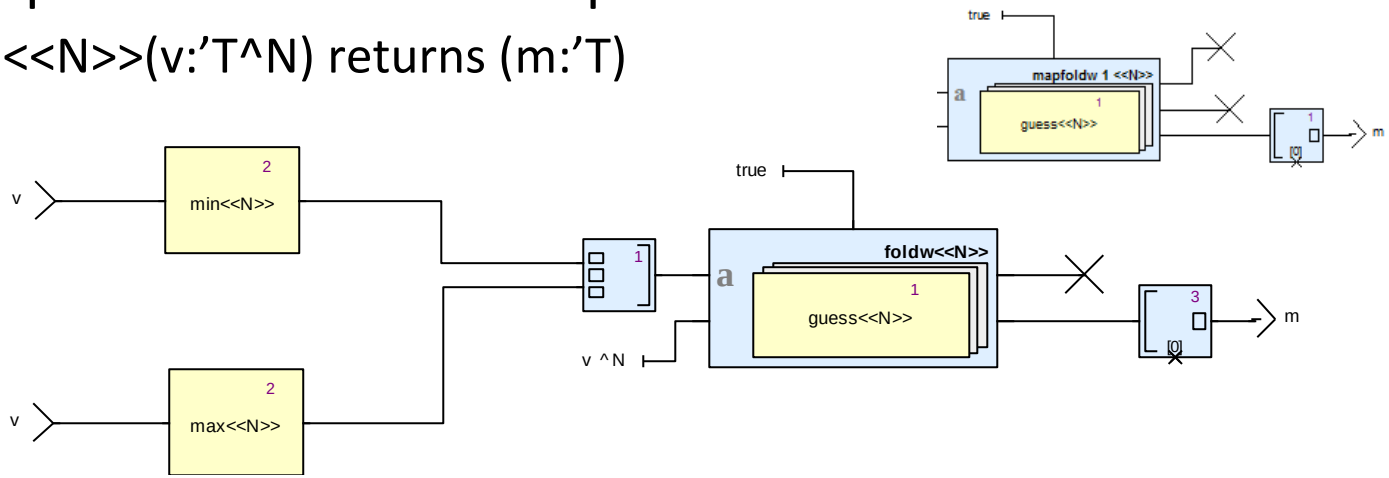


Lab 9: Solution

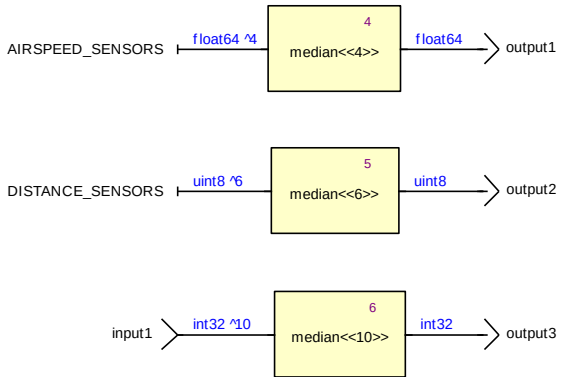
Lab Solution

4- Complete median<<N>> operator:

median<<N>>(v:'T^N) returns (m:'T)



5- Test median<<N>> operator (you can use the current created environment):



AGENDA

Arrays

Iterators

Code Generation

Imported Code

How is the SCADE Suite generated code optimized?

How to integrate a SCADE Suite design in a system?

SCADE Suite KCG

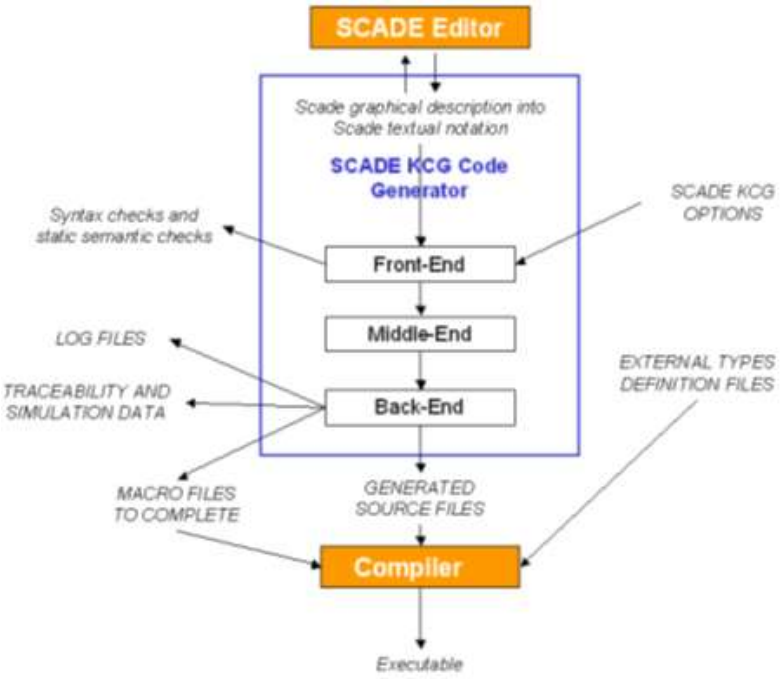
Qualifiable/certified code generator

Generated code properties:

- Readable, traceable (names propagation and annotations)
- Portable (independent from the target and the system scheduler)
- Modular
- Static memory allocation
- Finite execution duration
- Size optimizations

SCADE Suite KCG Architecture

SCADE Suite KCG take as input a SCADE Suite model



SCADE Suite KCG front-end

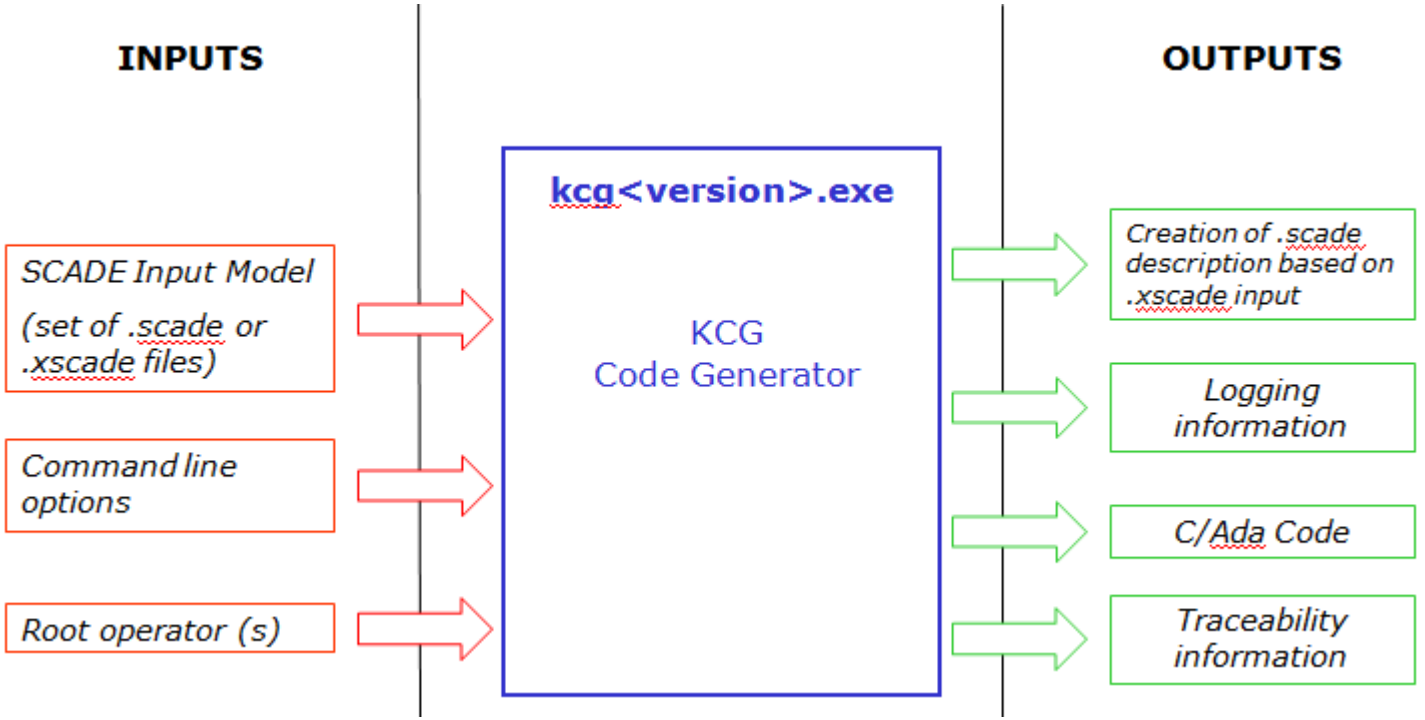
- Transforms the multi-files model into one SCADE Suite Program
- Prunes out the graphical information
- Propagates the selected annotations
- Performs syntax and semantics checking

SCADE Suite KCG middle-end: core compilation, optimizations

SCADE Suite KCG back-end

- Generates C/Ada code
- Generates the traceability files

SCADE Suite KCG Code Generator



Lab 10: KCG

Lab Support p. 63-67

Objective:

Understand the generated files

Time: 10 min

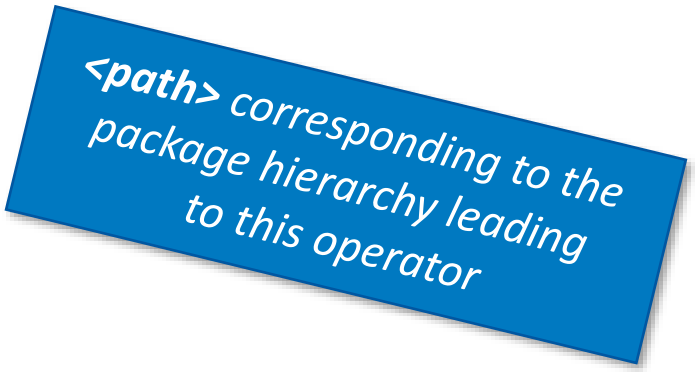
Requirements:

Generate KCG code of your median design, with default options

Observe the generated files (in KCG or KCGAda directory)

Lab

C Code Generation Files



<Target Directory> folder contains:

- kcg_types.h: declarations of basic Scade types, user-defined types, Structures for the state machines;
- kcg_types.c: macros definition of array copy;
- kcg_const.h: declaration of constants;
- kcg_const.c: implementation of constants;
- kcg_sensor.h: declaration of sensor inputs;
- <root>_<path>.h: declaration of:
  - I/Os root operator structures and sub-operators ones
  - <root >_init_<path>, <operator>_reset_<path> and <root>_<path> functions
- <root>_<path>.c: Implementation of:
  - <root>_init_<path>
  - <root>_reset_<path>
  - <root>_<path>

Ada Code Generation Files

Ada

The default definitions (predefined types, …) are not generated but are contained in

- kcg_config.ads located under <installation>\SCADE\include\Ada
- Kcg_config.adb located under <installation>\SCADE\lib\Ada

```
-- Predefined types definitions
subtype Kcg_Char is Character;
subtype Kcg_Int8 is Interfaces.Integer_8;
subtype Kcg_Int16 is Interfaces.Integer_16;
subtype Kcg_Int32  is Interfaces.Integer_32;
subtype Kcg_Int64  is Interfaces.Integer_64;
subtype Kcg_Uint8  is Interfaces.Unsigned_8;
subtype Kcg_Uint16 is Interfaces.Unsigned_16;
subtype Kcg_Uint32 is Interfaces.Unsigned_32;
subtype Kcg_Uint64 is Interfaces.Unsigned_64;
subtype Kcg_Float32 is Interfaces.IEEE_Float_32;
subtype Kcg_Float64 is Interfaces.IEEE_Float_64;
subtype Kcg_Size is Integer;
```

ANSYS

Ada Code Generation Files

path *corresponding to the package hierarchy leading to the package or operator*

Ada

<Target Directory> folder contains:

- kcg_types.ads: manifest user-defined types as *subtype*, structures for the state machines as *type*;
- <package-path>.ads: Ada specification for non-imported elements (non-expanded operators)
  - Non-manifest Scade types as *subtypes*
  - Constants
  - I/Os root operator and sub-operators structures (output contexts) as *type*
  - init, reset and cyclic procedures for root and sub-nodes
  - Function if the non expanded Scade function is not the root operator and returns a single output otherwise a procedure

Ada Code Generation Files

Ada

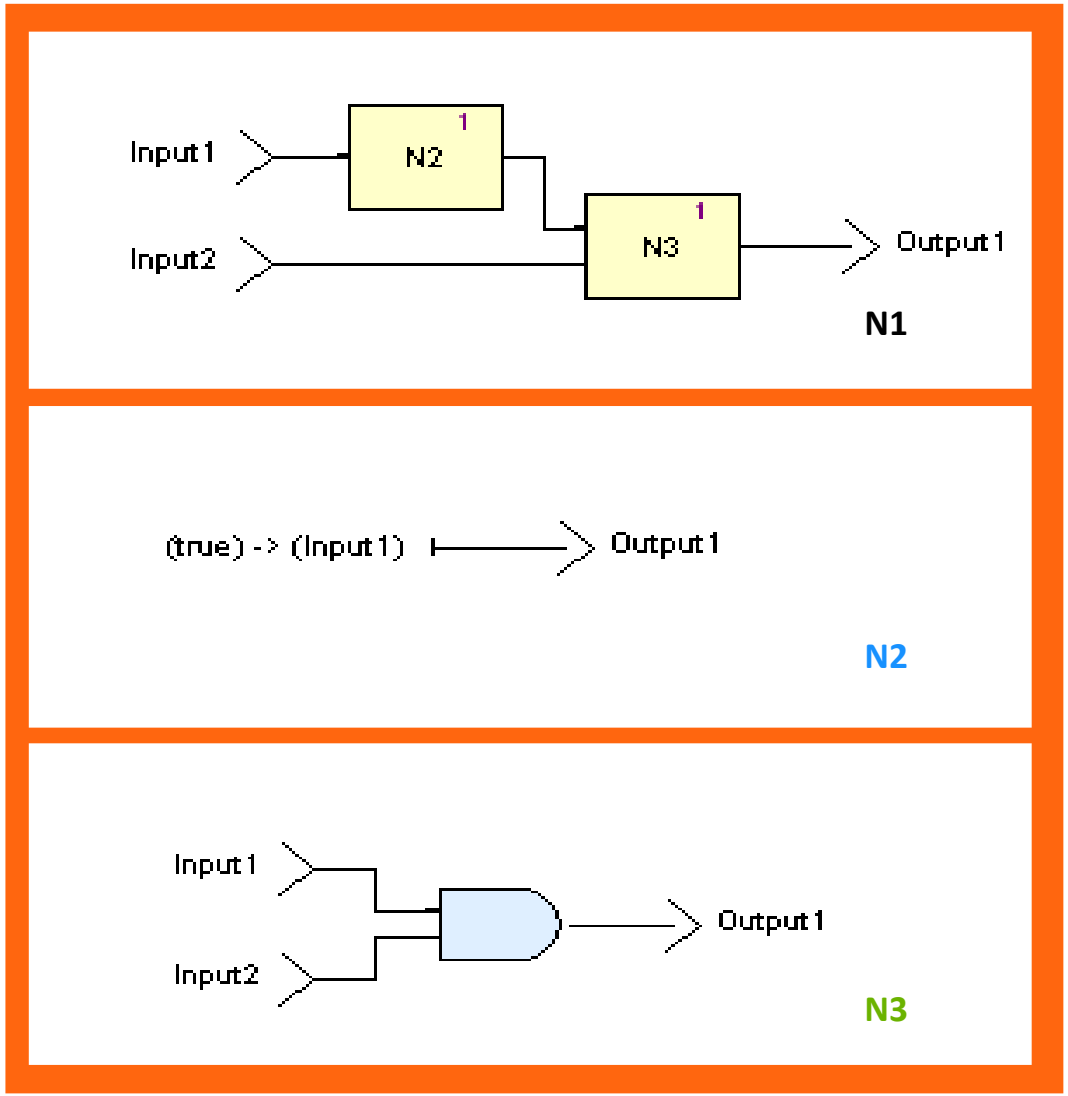
- <package-path>.adb: Ada implementation (body) for non-imported and non-expanded operators:
  - init, reset and cyclic Ada procedures for root and sub-nodes
  - Ada function if the Scade function is not the root operator and returns a single output otherwise a procedure
  - The cyclic Ada procedures and functions are generated as separate compilation units into <package-path-operator>.adb
  - The reset / init Ada procedures are generated as separate compilation units respectively into <package-path-operator>_reset/init.adb

path *corresponding to the package hierarchy leading to the package or operator*

ANSYS

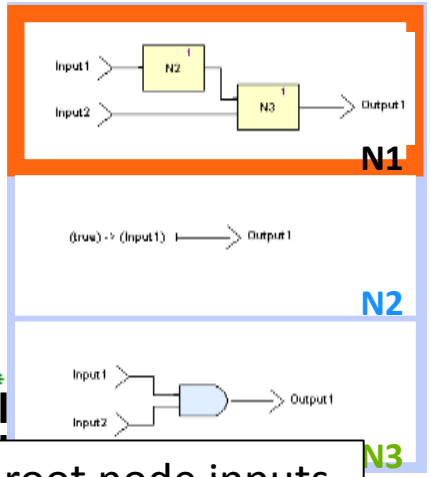
SCADE Design

Example



Root Operator: N1.h

Example




```
#include "kcg_types.h"
#include "N3.h"
#include "N2.h"

/* ------------------- input structure ------------------- */
typedef struct {
  kcg_bool /* N1::Input1 */ Input1;
  kcg_bool /* N1::Input2 */ Input2;
} inC_N1;

/* ------------------  no output structure  ------------------ */

/* ------------------------  context type  ------------------------ */
typedef struct {
  /* --------------------------  outputs  ----------
  kcg_bool /* N1::Output1 */ Output1;
  /* ----------------------  no local probes  -
  /* -----------------  no initialization variab
  /* ----------------------  no local memory  -
  /* --------------------  sub nodes' contexts
  outC_N2 /* 1 */ Context_1;
  /* ----------------- no clocks of observable d
} outC_N1;

/* ===========  node initialization and cycle functions  =========== */
/* N1 */
extern void N1(inC_N1 *inC, outC_N1 *outC);

#ifndef KCG_NO_EXTERN_CALL_TO_RESET
extern void N1_reset(outC_N1 *outC);
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */

#ifndef KCG_USER_DEFINED_INIT
extern void N1_init(outC_N1 *outC);
#endif /* KCG_USER_DEFINED_INIT */
```

Input context: Structure with root node inputs

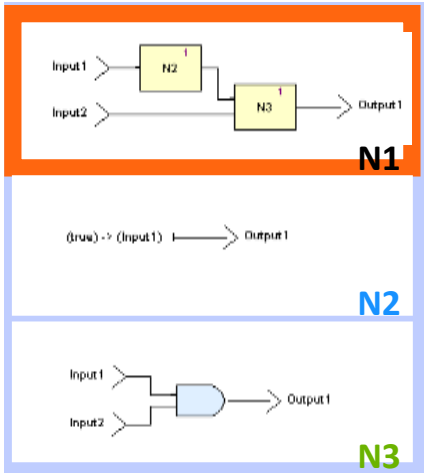
Output context

- Structure with root node outputs and memories
- Sub-nodes contexts

Cyclic, reset and **initialization** functions

Root Operator: N1.c

Example




```
#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "N1.h"

#ifndef KCG_USER_DEFINED_INIT
void N1_init(outC_N1 *outC)
{
  outC->Output1 = kcg_true;
  /* 1 */ N2_init(&outC->Context_1);
}
#endif /* KCG_USER_DEFINED_INIT */


#ifndef KCG_NO_EXTERN_CALL_TO_RESET
void N1_reset(outC_N1 *outC)
{
  /* 1 */ N2_reset(&outC->Context_1);
}
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */

/* N1 */
void N1(inC_N1 *inC, outC_N1 *outC)
{
  /* 1 */ N2(inC->Input1, &outC->Context_1);
  outC->Output1 = /* 1 */ N3(outC->Context_1.Output1, inC->Input2);
}
```

initialization and **reset functions**

Note: N3 is declared as a function (it has no memory as opposed to the node)

Cyclic function

Sub-Operators Implementation C File

Example

Scade function with one scalar output:

- Cyclic C function with inputs passed as parameters and returning the output value

```
/* N3 */
extern kcg_bool N3(kcg_bool Input1, kcg_bool Input2);
```

Scade function with several outputs:

- Cyclic C void with inputs and outputs passed as parameters

```
/* N4 */
extern void N4(
  kcg_bool Input1,
  kcg_bool Input2,
  /* N4::Output1 */kcg_bool *Output1,
  /* N4::Output2 */kcg_bool *Output2);
```

- No reset and initialization C function for SCADE Suite function

Scade node with one or several outputs:

- Cyclic C void with inputs passed as parameters, outputs and memories passed in a context

```
/* N2 */
extern void N2(kcg_bool Input1, outC_N2 *outC);
```

ANSYS

Initialization, Reset and Cyclic C Functions

Example

```
void N1_init(outC_N1 *outC)
{
  outC->Output1 = kcg_true;
  N2_init(&outC->Context_1);
}
```

```
void N2_init(outC_N2 *outC)
{
  outC->init = kcg_true;
  outC->Output1 = kcg_true;
}
```

```
void N1_reset(outC_N1 *outC)
{
 /* 1 */N2_reset(&outC->Context_1);
}
```

```
void N2_reset(outC_N2 *outC)
{
  outC->init = kcg_true;
}
```

```
void N1(inC_N1 *inC, outC_N1 *outC)
{
 /* N1::_L1 */ kcg_bool _L1;

 /* 1 */ N2(inC->Input1, &outC-
>Context_1);

_L1 = outC->Context_1.Output1;
outC->Output1 = /* 1 */ N3(_L1, inC-
>Input2);
}
```

```
void N2(/* N2::Input1 */ kcg_bool Input1,
outC_N2 *outC)
{
  if (outC->init) {
    outC->Output1 = kcg_true;
  }
  else {
    outC->Output1 = Input1;
  }
  outC->init = kcg_false;
}
```

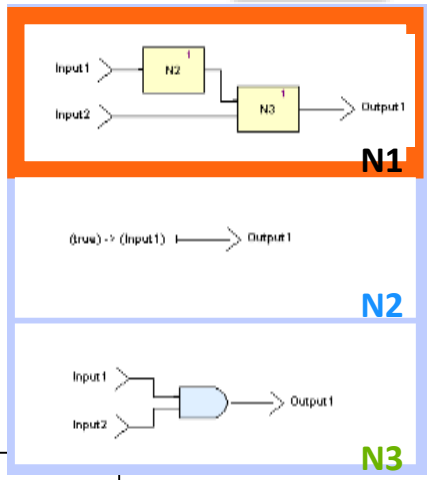
Ada Package specification: .ads

Ada

Example

```
type Context_N1 is
  record
  -- ----------------------
  -- Output1/, _L4/
  Output1 : Boolean;
  -- ----------------------
  -- ----------------- no (
  -- ----------------------
  -- --------------------
  -- ------------------  n(
  -- ----------------------
  -- _L3=(Top::N2#1)/
  SubCtx_N2_1 : Context_N2;
  end record;
```

```
type Context_N2 is
  record
  -- --------------------------  outputs  ----------
  -- Output1/, _L2/
  Output1 : Boolean;
  -- ----------------------  no local probes  -------
  -- ----------------- no clocks of observable data -
  -- ----------------------  local memories  --------
  init : Boolean;
  -- --------------------  no debug variables  -------
  -- ------------------  no assertion variables  -----
  -- --------
  end record;
```



Output contexts

- Structure with root node outputs and memories
- Sub-nodes contexts

```
procedure N1_Reset(
  Ctx : in out Context_N1);

procedure N1_Init(
  Ctx : out Context_N1);

-- Top::N1/
procedure N1(
  -- Input1/, _L1/
  Input1 : in Type1;
  -- Input2/, _L2/
  Input2 : in Boolean;
  Ctx : in out Context_N1);
```

```
procedure N2_Reset(
  Ctx : in out Context_N2);

procedure N2_Init(
  Ctx : out Context_N2);

-- Top::N2/
procedure N2(
  -- Input1/
  Input1 : in Boolean;
  Ctx : in out Context_N2);
```

```
-- Top::N3/
function N3(
  -- Input1/, _L1/
  Input1 : in Boolean;
  -- Input2/, _L2/
  Input2 : in Boolean) return Boolean;
```

Reset, init and cyclic procedures
Cyclic functions

ANSYS

Root Operator Ada Body Files

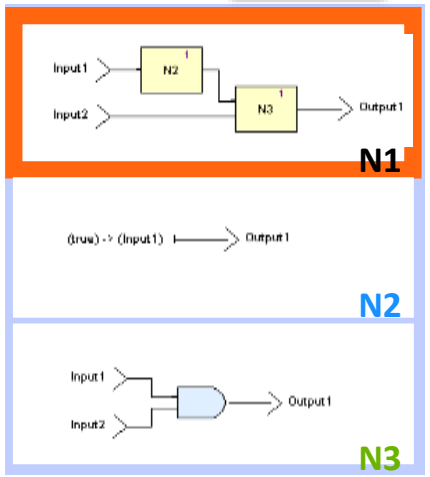
Ada

Example

```
-- Top::N1/
procedure N1(
   -- Input1/, _L1/
   Input1 : in Type1;
   -- Input2/, _L2/
   Input2 : in Boolean;
   Ctx : in out Context_N1)
is

begin
   -- _L3=(Top::N2#1
   N2(Input1  =>  Input1, Ctx  =>  Ctx.SubCtx_N2_1);
   Ctx.Output1 := -- _L4=(Top::N3#1)/
   N3(Input1  =>  Ctx.SubCtx_N2_1.Output1, Input2  =>  Input2);
end N1;
```

Cyclic function

<package>-n1.adb




```
procedure N1_Init(
   Ctx : out Context_N1)
is

begin
   Ctx.Output1 := True;
   -- _L3=(Top::N2#1)/
   N2_Init(Ctx  =>  Ctx.SubCtx_N2_1);
end N1_Init;
```

Initialization procedure

<package>-n1_init.adb

```
procedure N1_Reset(
   Ctx : in out Context_N1)
is

begin
   -- _L3=(Top::N2#1)/
   N2_Reset(Ctx  =>  Ctx.SubCtx_N2_1);
end N1_Reset;
```

Reset procedure

<package>- n1_reset.adb

ANSYS

Sub-Operator Body Ada File

Ada

Example

```
-- Top::N3/
function N3(
  -- Input1/, _L1/
  Input1 : in Boolean;
  -- Input2/, _L2/
  Input2 : in Boolean) return Boolean
is

  -- Output1/, _L3/
  Output1 : Boolean;
begin
  Output1 := Input1 and Input2;
  return Output1;
end N3;

  -- Top::N4/
  procedure N4(
    -- Input1/, _L1/
    Input1 : in Type1;
    -- Input2/, _L2/
    Input2 : in Boolean;
    -- Output1/
    Output1 : out Boolean;
    -- Output2/
    Output2 : out Boolean)
  is

  begin
    Output1 := Input1;
    Output2 := Input2;
  end N4;
```

Scade function N3 with one scalar output:

- Cyclic Ada function with inputs passed as parameters and returning the output value

Scade function N4 with several outputs:

- Cyclic Ada procedure with inputs and outputs passed as parameters

Scade function:

- **No reset and initialization Ada procedures and functions**

ANSYS

Sub-Operator Body Ada Files

Ada

Example

Scade node with one or several outputs:

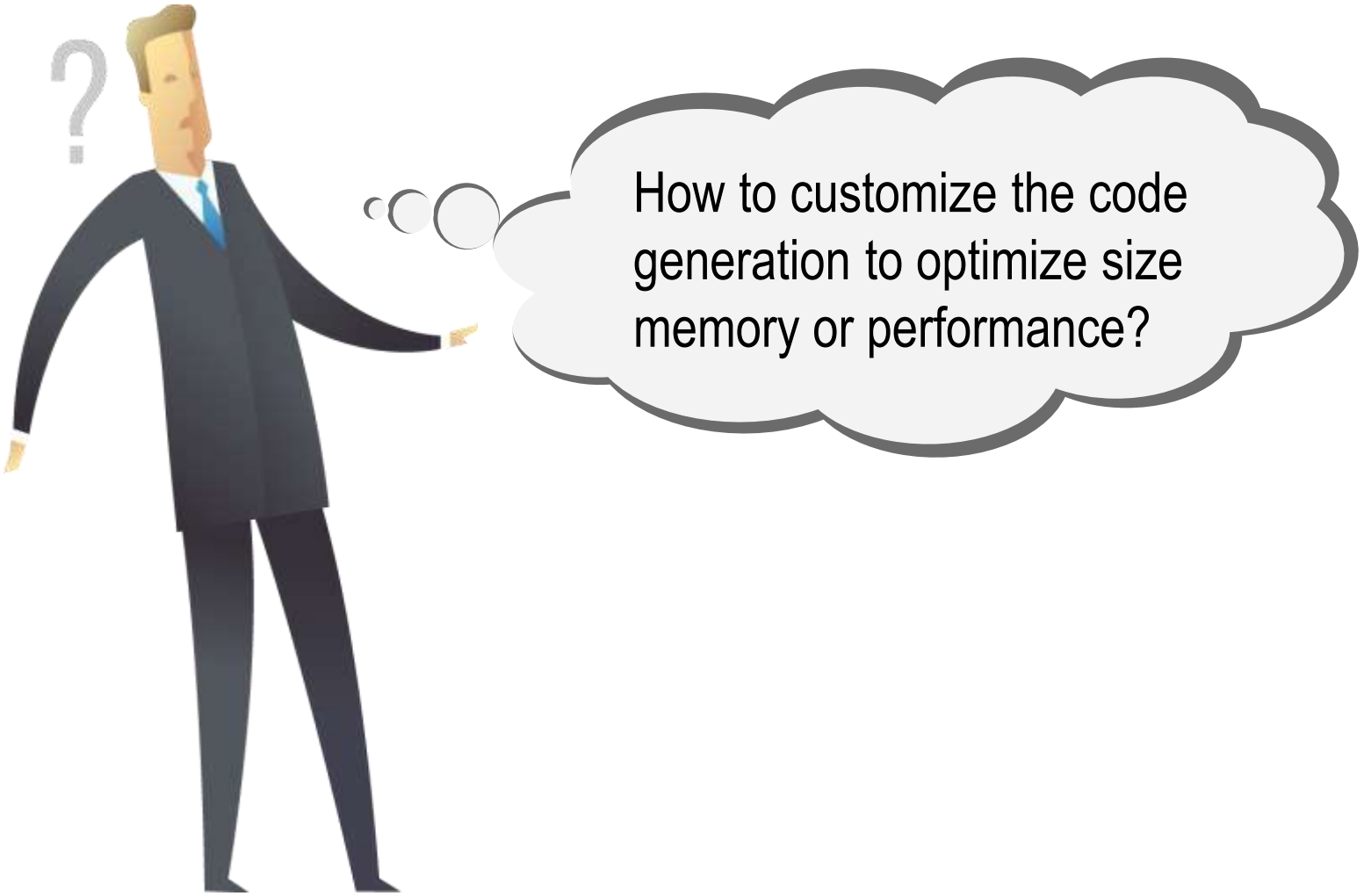
- Cyclic Ada procedure with inputs passed as parameters, outputs and memories passed in a context
- Init and reset Ada procedures into _init.adb and _reset.adb separate files

```
-- Top::N2/
procedure N2(
  -- Input1/
  Input1 : in Boolean;
  Ctx : in out Context_N2)
is

begin
  Ctx.Output1 := Ctx.init or Input1;
  Ctx.init := False;
end N2;
```

```
procedure N2_Init(
  Ctx : out Context_N2)
is

begin
  Ctx.Output1 := True;
  Ctx.init := True;
end N2_Init;
```

```
procedure N2_Reset(
  Ctx : in out Context_N2)
is

begin
  Ctx.init := True;
end N2_Reset;
```

ANSYS


How to customize the code generation to optimize size memory or performance?

General Options

Root operator (-node <nodename>)

- Select root operator(s) for code generation

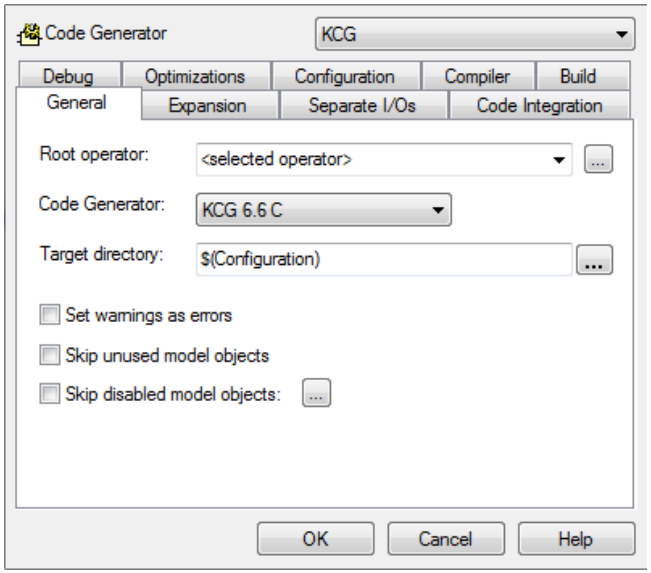
Code Generator: C or Ada

Target dir (-target_dir <dirname>)

- Specifies the generation directory

Set warnings as errors (-warn_error)

- Manage warnings as errors



Not qualified / certified SCADE Suite KCG options:

- Skip unused model objects
  - Generates code related only to the call graph of the selected root node
- Skip disabled model objects
  - Generates code related only to the packages, operators and operators diagrams that are not disabled (a Browse button allows to select the elements to disable)

Compiler Options

CPU Type:

- Select win32 or win64 bits code generation

Compiler:

- Select Mingw GNU C (by default) or a version of installed Visual C compilers

Preprocessor definitions

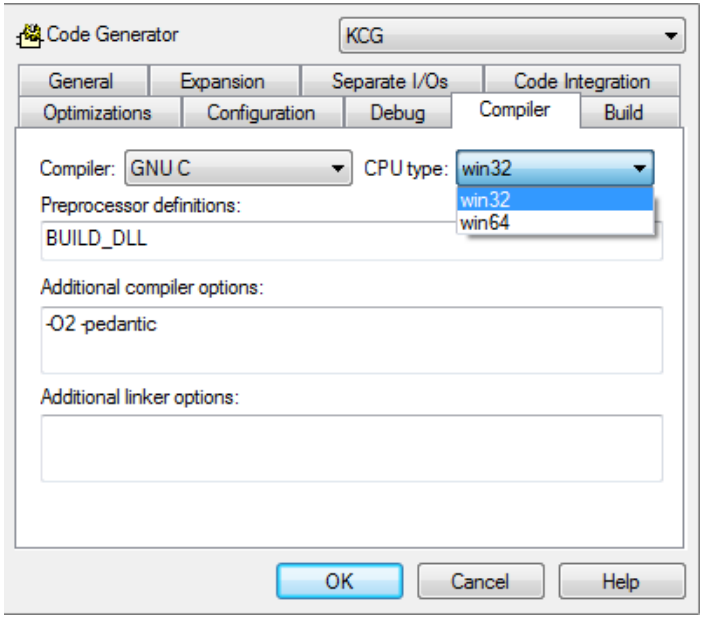
- Specifies preprocessor definitions

Additional compiler options

- Specifies more compiling options

Additional linker options

- Specifies more linker options



Note: separate list items with a space and use quotes if an item contains spaces. Items must be compliant with the Tcl language format

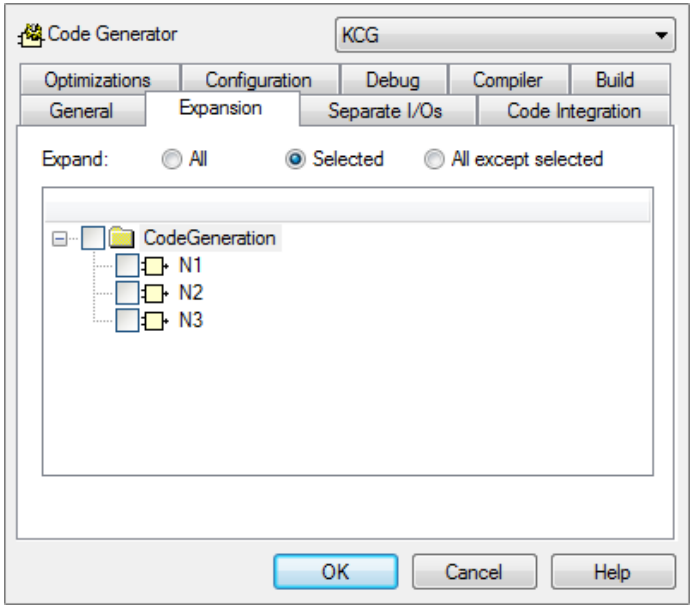
Expansion Options

All (-expall)

- This option specifies that all operators except the root operator are expanded during code generation

Selected
(-exp [identifierlist])

- This option specifies operators that you want to be expanded



All except selected
(-noexp [identifierlist])

- This option specifies operators that you want to be non-expanded
- By default, operators are not expanded
- The –exp and -noexp options cannot be applied to the root operator

Expansion: CPU vs. Memory (C Code)

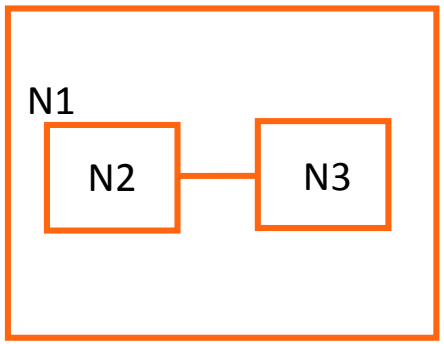
Example

Functional (non-expanded)

Each SCADE Suite operator is generated as a C function.

Expanded

Each SCADE Suite operator call is expanded.



SCADE Suite description

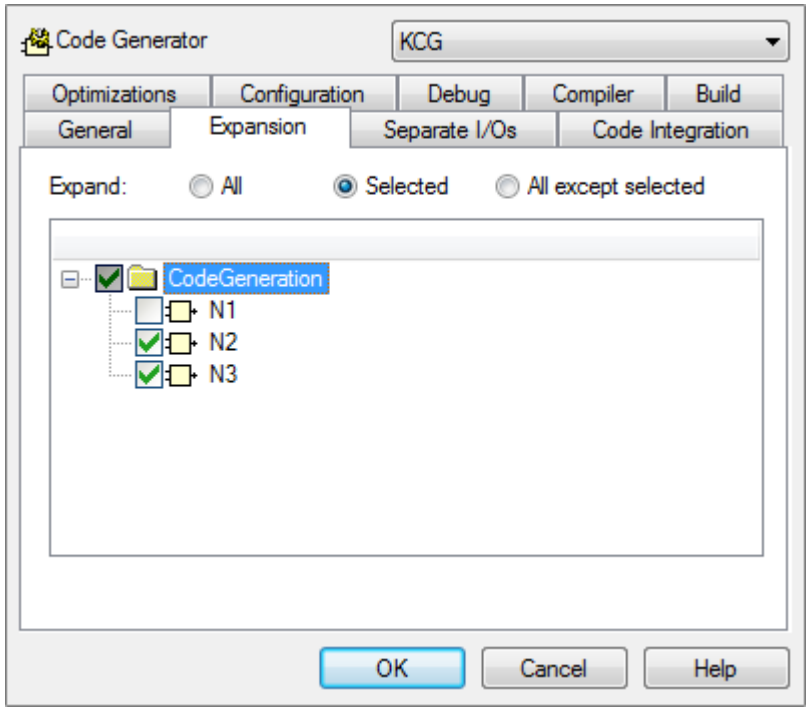
No expansion (call mode)	Expansion (inline mode)

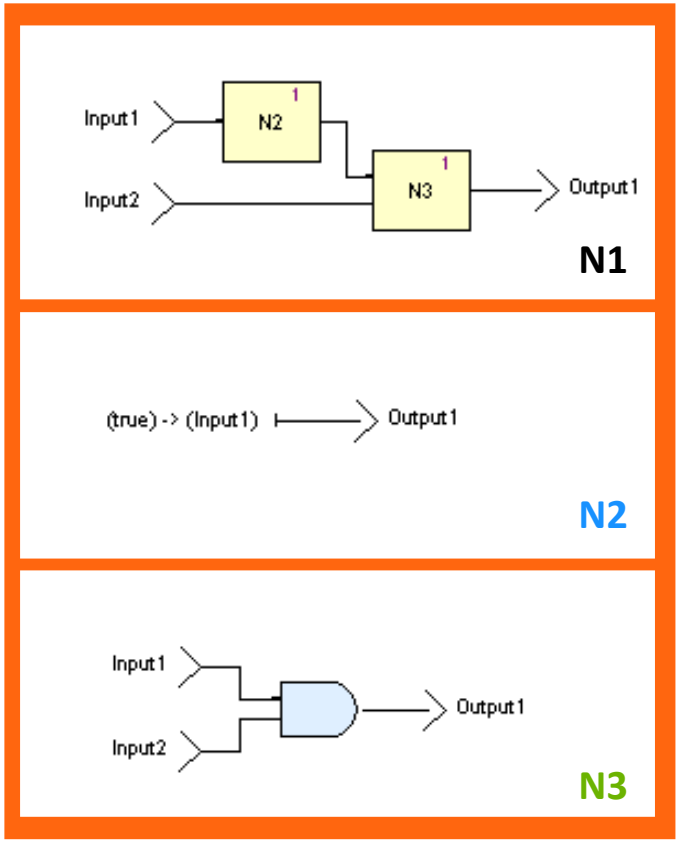
`N2 () {…}`
`N3 () {…}`
`N1 () {`
`  …`
`  N2 ();`
`  …`
`  N3 ();`
`  …`
`}`
	`N1 {`
	`  …`
	`  Code for N2 and N3,`
	`  possibly optimized`
	`  …`
	`}`

Expansion: CPU vs. Memory (C Code)

Example

Expansion per operator (except always root node)





Expansion: CPU vs. Memory (C Code)

Example

N2 and N3 non-expanded

```
void N1(inC_N1 *inC, outC_N1 *outC)
{
  N2(inC->Input1, &outC->Context_1);
  outC->Output1 = N3(outC->Context_1.Output1, inC->Input2);
}
```

Call to the N2 operator code

Call to the N3 operator code

N2 non-expanded, N3 expanded

```
void N1(inC_N1 *inC, outC_N1 *outC)
{
  N2(inC->Input1, &outC->Context_1);
  outC->Output1 = outC->Context_1.Output1 & inC->Input2;
}
```

Call to the N2 operator code

Code for the N3 operator

Expansion: CPU vs. Memory (C Code)

Example

N2 and N3 expanded

```
void N1(inC_N1 *inC, outC_N1 *outC)
{
  kcg_bool tmp;

  if (outC->init) {
    outC->init = kcg_false;
    tmp = kcg_true;
  }
  else {
    tmp = inC->Input1;
  }
```

Code for the N2 operator

```
 outC->Output1 = tmp & inC->Input2;
}
```

Code for the N3 operator

Expansion: CPU vs. Memory (Ada)

Ada

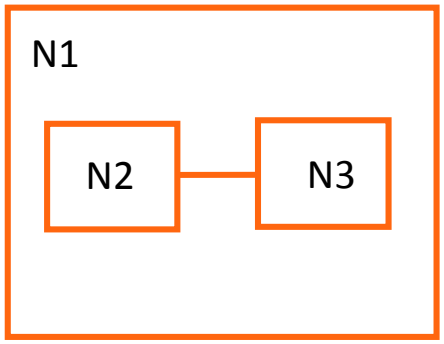
Example

Functional (non-expanded)

Each SCADE Suite operator is generated as a C function.

Expanded

Each SCADE Suite operator call is expanded.



SCADE Suite description

No expansion (Call Mode)	Expansion (inline mode)

`N2 () begin…end N2`
`N3 () begin…end N3`
`N1 ()`
`…`
`begin`
`  …`
`  N2 ();`
`  N3 ();`
`  …`
`end N1`
	`N1 ()`
	`  …`
	`begin`
	`  Code for N2 and N3,`
	`  possibly optimized`
	`  …`
	`end N1`

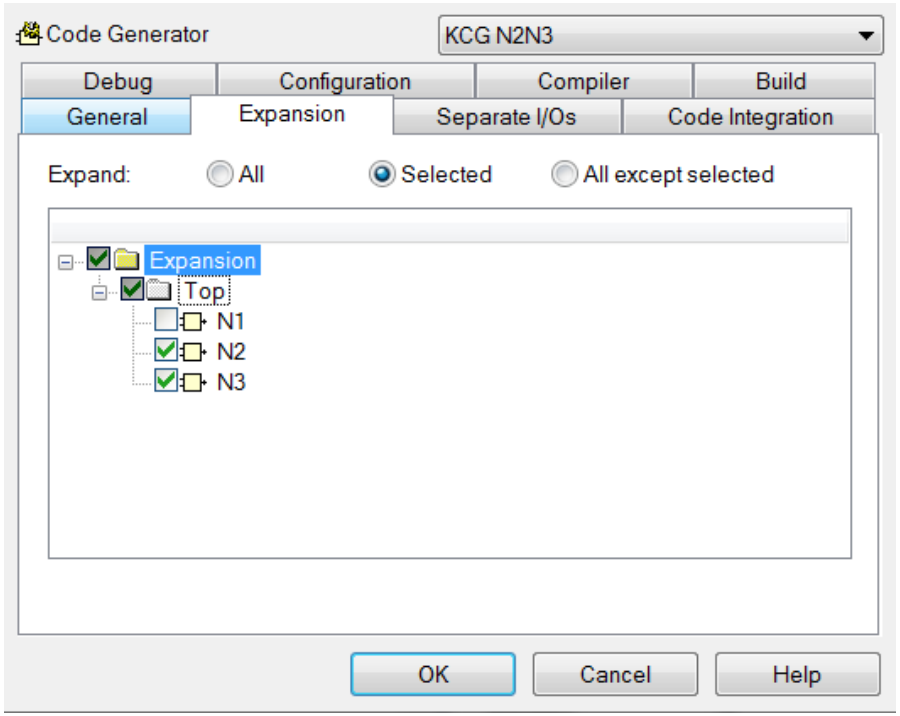
ANSYS

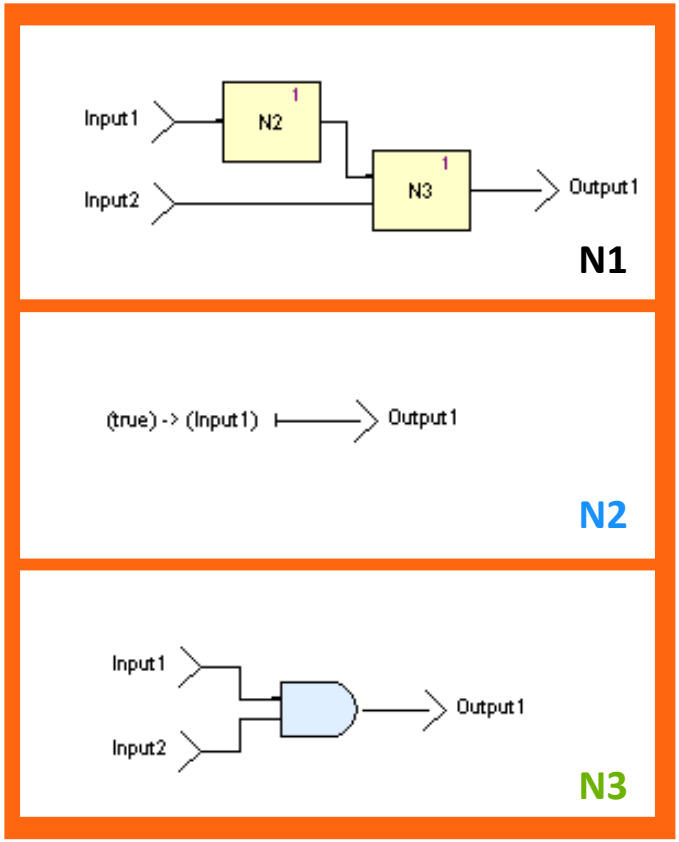
Expansion: CPU vs. Memory (Ada)

Ada

Example

Expansion per operator (except always root node)





Expansion: CPU vs. Memory (Ada)

Ada

Example

N2 and N3 non-expanded

```
-- Top::N1/
procedure N1(
  -- Input1/, _L1/
  Input1 : in Type1;
  -- Input2/, _L2/
  Input2 : in Boolean;
  Ctx : in out Context_N1)
is

begin
  -- _L3=(Top::N2#1)/
  N2(Input1  =>  Input1, Ctx  =>  Ctx.SubCtx_N2_1);
  -- Ctx.Output1 : --- _L4 (Top::N3#1)/
  N3(Input1  =>  Ctx.SubCtx_N2_1.Output1, Input2  =>  Input2);
end N1;
```

Call to the N2 operator code

Call to the N3 operator code

N2 non-expanded, N3 expanded

```
-- Top::N1/
procedure N1(
  -- Input1/, _L1/
  Input1 : in Type1;
  -- @1/Input2/, @1/_L2/, Input2/, _L2/
  Input2 : in Boolean;
  Ctx : in out Context_N1)
is

begin
  -- _L3=(Top::N2#1)/
  N2(Input1  =>  Input1, Ctx  =>  Ctx.SubCtx_N2_1);
  Ctx.Output1 := Ctx.SubCtx_N2_1.Output1 and Input2;
end N1;
```

Call to the N2 operator code

Code for the N3 operator

Expansion: CPU vs. Memory (Ada)

Ada

Example

N2 and N3 expanded

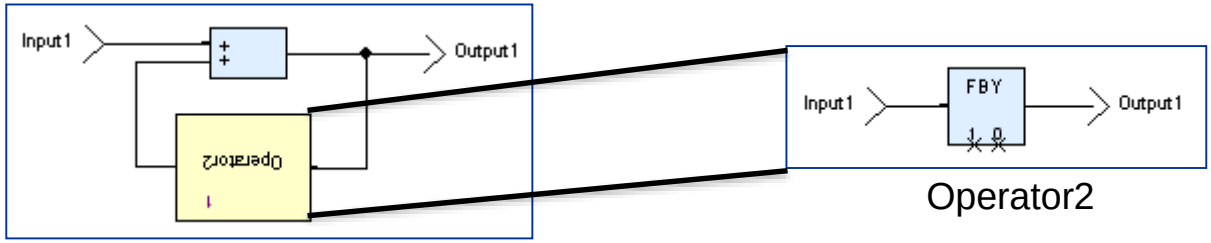
```
-- Top::N1/
procedure N1(
  -- @1/Input1/, Input1/, _L1/
  Input1 : in Type1;
  -- @2/Input2/, @2/_L2/, Input2/, _L2/
  Input2 : in Boolean;
  Ctx : in out Context_N1)
is

begin
  Ctx.Output1 := (Ctx.init or Input1) and Input2;
  Ctx.init := False;
end N1;

-- Expanded instances for: Top::N1/
-- @1: (Top::N2#1)
-- @2: (Top::N3#1)
```

Code for N2 and N3 operator

Expansion vs Causality Loop

Example

Feedback case:



KCG will detect a causality error if the Operator2 is not expanded: SCADE language makes no assumptions on the content of an operator: => Non information of which input is connected to which output

=> Non visibility on FBY that cut the feedback (“loop”) by a delay

KCG will generate the code without error if the Operator2 is expanded to have that direct “loop”

Lab 11: KCG options

Lab

Objective:

Understand the KCG options.

Time: 10 min

Requirements:

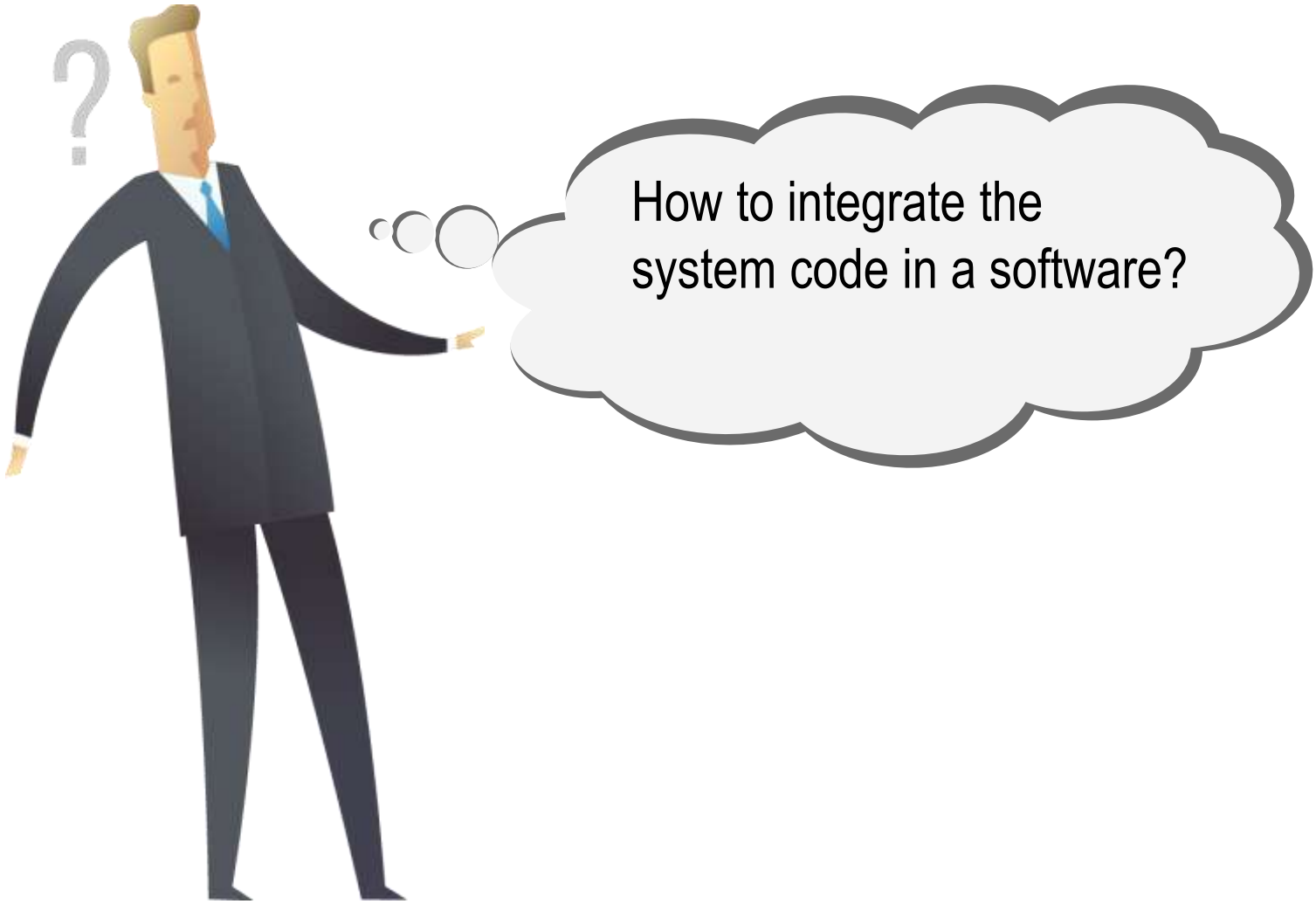
Generate KCG code of your median design, with different options:

- Change the target directory,
- Modify the expansion options,
- Generate the context as global,
- Change the name length and significance length,

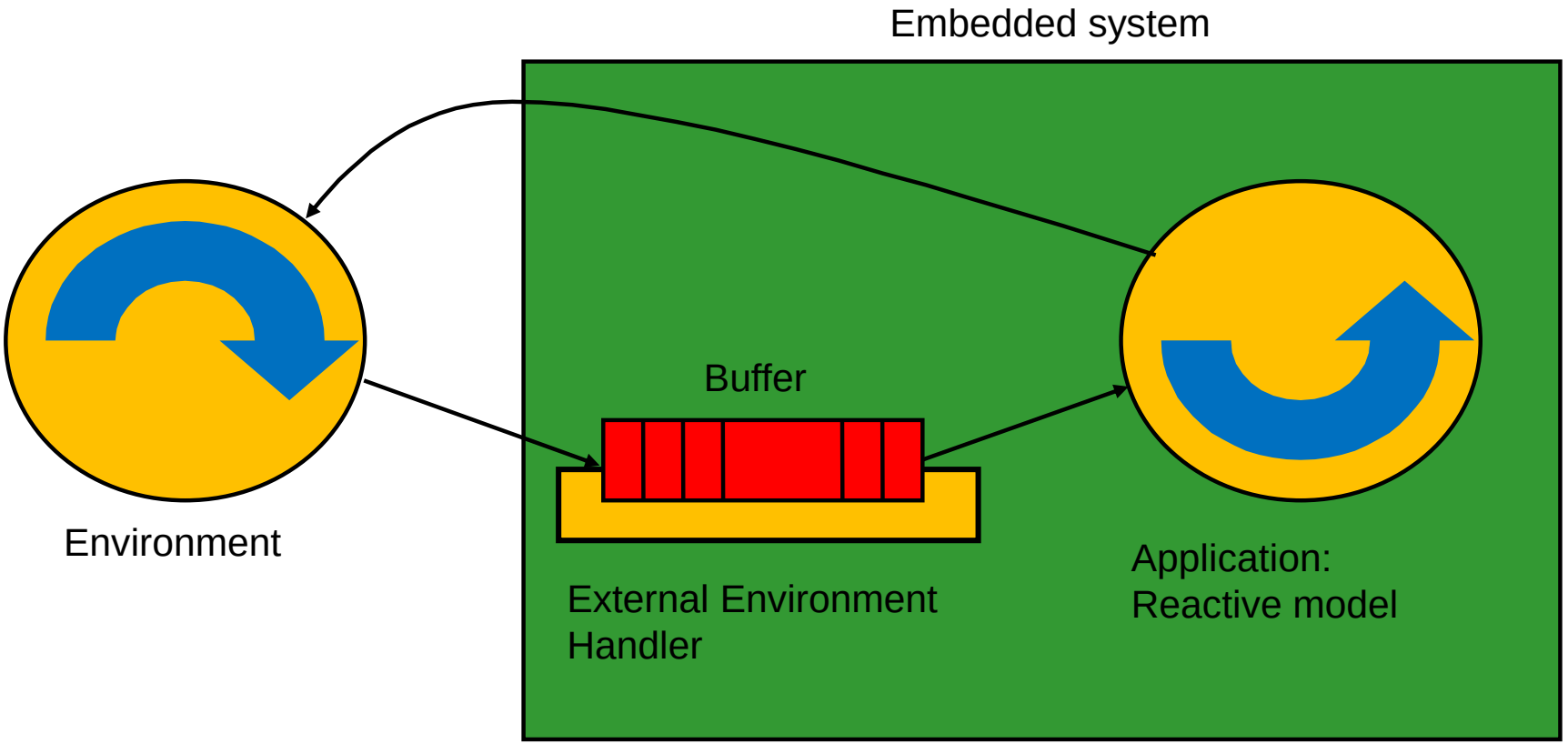
Select and prevent the optimization of a variable,

…

For each generation, observe the generated files (in KCG or KCGAda directory).


How to integrate the system code in a software?

Global Embedded System Architecture



Module Integration

Integration in an existing software:

- Write the main cyclic loop,
- Call the generated functions.

```
call the generated initialization function
begin_loop
  wait for an event
  treat the inputs
  call the generated cyclic function
  treat the outputs
end_loop
```

Module Integration

Module integration is not unique but depends on the Code Generator options:

- Generated files vary according to SCADE Suite KCG options,
- Communication between the generated code and the main application depends on the SCADE Suite KCG options.

Module integration also depends on the basic types implementation:

- The user could provide a user configuration file allowing to map the scade basic types (kcg_bool, kcg_char, …) on specific C/Ada types (float, double, signed short, signed long, unsigned short…)

C Mapping of Basic Scade Types

A default kcg_types.h file is generated in the target code folder to map basic Scade types (kcg_float32, kcg_float64, kcg_intN, kcg_uintN, …) to C types (with N=8,16,32,64)

Example:

```
#ifndef kcg_float32
#define kcg_float32 kcg_float32
typedef float kcg_float32;
#endif /* kcg_floaat32 */

#ifndef kcg_int8
#define kcg_int8 kcg_int8
typedef signed char kcg_int8;
#endif /* kcg_int8 */
```

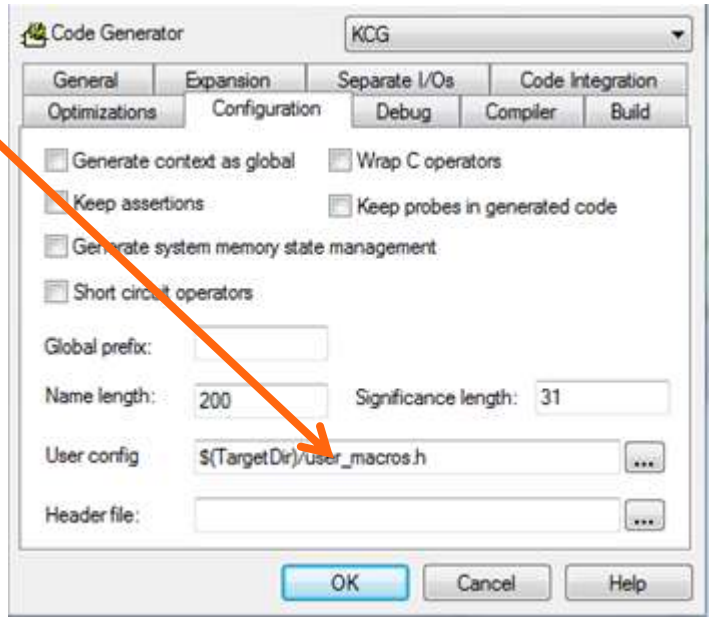
C Mapping of Basic Scade Types

Users may redefine basic types (except char) by providing the symbolic definition of the type (#define or typedef) in a user configuration file

The path to this user configuration file has to be provided as an option to SCADE Suite KCG

Example:

```
#ifndef kcg_int8
#define kcg_int8 kcg_int8
typedef signed short int kcg_int8;
#endif /* kcg_int8 */

#ifndef kcg_float64
#define kcg_float64 kcg_float64
typedef float kcg_float64;
#endif /* kcg_float64 */
```



Module Integration (C Code)

Example

Communication between the generated code and the main application: input context and output context

```
typedef struct {
  kcg_bool /* System::SystemSimul::On */ On;
  kcg_bool /* System::SystemSimul::Resume */ Resume;
  kcg_bool /* System::SystemSimul::Set */ Set;
  kcg_bool /* System::SystemSimul::QuickAccel */ QuickAccel;
  kcg_bool /* System::SystemSimul::QuickDecel */ QuickDecel;
  tPercent_CarType /* System::SystemSimul::Accel */ Accel;
  tPercent_CarType /* System::SystemSimul::Brake */ Brake;
} inC_SystemSimul_System;

typedef struct {
  /* -------------------------  outputs  ------------------------- */
  tSpeed_CarType /* System::SystemSimul::CruiseSpeed */ CruiseSpeed;
  tCruiseState_CruiseControl /* System::SystemSimul::CruiseState */ CruiseState;
  kcg_float32 /* System::SystemSimul::CarSpeed */ CarSpeed;
  /* ---------------------  no local probes  --------------------- */
  /* ----------------  no initialization variables  ---------------- */
  /* ----------------------  no local memory  ---------------------- */
  /* --------------------  sub nodes' contexts  -------------------- */
  outC_CarModel_Car /* 1 */ Context_1;
  outC_CruiseControl_CruiseControl /* 2 */ Context_2;
  /* ----------------- no clocks of observable data ----------------- */
} outC_SystemSimul_System;
```

C Module Integration

Example

```
// Include the header file for the system_simul
#include "stdio.h"
#include "SystemSimul_System.h"

int main(int argc, char* argv[])
{
 // Declare 2 variables of the system input/output context type
inC_SystemSimul_System  inC;
outC_SystemSimul_System outC;

// Call the init function
  SystemSimul_init_System(&outC);

// Make a Loop
do {
 inC.Accel = 0.8;
 inC.Set = kcg_true;

// Call the system C generated function //

 SystemSimul_System(&inC,&outC);

 printf("CruiseSpeed = %lf\n",outC.CruiseSpeed);
 printf("VehiculeSpeed = %lf\n",outC.VehiculeSpeed);

 }
 while (1);
 return 0;
}
```

Generated file to include

Contexts for the generated function

Call the generated init function

Treatment of the inputs

Main Loop

Call the generated cyclic function

Treatment of the outputs

Ada Mapping of Basic Scade Types

Ada

The predefined types definitions are not generated but are contained in **kcg_config.ads** located under <installation>\SCADE\include\Ada to map basic Scade types (kcg_Float32, kcg_Float64, kcg_IntN, kcg_UintN, …) to Ada types

with N=8,16,32,64

```
subtype Kcg_Float32 is Interfaces.IEEE_Float_32
subtype Kcg_Float64 is Interfaces.IEEE_Float_64
subtype Kcg_IntN is Interfaces.Integer_N;
subtype Kcg_UintN is Interfaces.Unsigned_N;
subtype Kcg_Char is Character;
subtype Kcg_Size is Integer;
```

ANSYS

Ada Module Integration

Ada

Example

```
-- Include package files to use
with kcg_Config;
with Kcg_Types; with CartType; with System_1;

procedure Main()
 is
 -- Declare the system input parameters and output context
ctx: Context_SystemSimul;
brake; accel : CarType.tPercent;
quickDecel; quickAccel; set_1 : Boolean;
resumeCmd; on : Boolean;

begin
 -- Call the init function
     SystemSimul_Init(Ctx => ctx);
-- Make a Loop
while (Stop /=True) loop
 accel := 0.8;
 set_1: = False;
 .....
-- Call the cyclic generated function
 SystemSimul_Init(....; Ctx => ctx);

 Ada.Text_IO.Put(Kcg_Config.Kcg_Float64'Image(ctx.CruiseSpeed));
 Ada.Text_IO.Put(Kcg_Config.Kcg_Float64'Image(ctx.VehiculeSpeed));

end loop;
end Main;
```

Generated files to include

Context for the generated function

Declaration of inputs parameters

Call the generated init function

Main Loop

Treatment of the inputs

Call the generated cyclic function

Treatment of the outputs

ANSYS

Exercise 5 (C Code)

Exercise

Objective:

Create a program calling the C code generated by SCADE Suite.

Requirement:

Time: 15 min

Copy the Exercise 5 from the Prerequisite folder:

- The SCADE model computes the sliding mean, minimum and maximum values of a real flow over 3 cycles

Generate the corresponding C code

Move Make.bat, main.c and kcg_assign.h (C folder) to the project level

Complete main.c and build it with all the generated files

Execute the binary you obtain and try to run a nominal behavior

Exercise 5: Prerequisite (C Code)

Exercise

Step 1 – Generate The Code

Generate the code according to the following requirements:

- C files are generated in the folder EmbeddedCode
- Do not perform any expansion except for libraries

Exercise 5: Prerequisite (C Code)

Exercise

Step 2 – Complete The Main C Program

```
// TODO: Include the header file of M3C3
#include

int main(int argc, char* argv[])
{
  // Local variable declarations
  kcg_float64 value;

  // TODO: Declare 2 variables of the M3C3
  // input/output context type

  // TODO: Call the generated initialization function
  // with the address of the system
  // context as a parameter

  // Make a Loop
  do
  {
    // Read the inputs
    printf("Enter a real value : ");
    scanf("%lf",&value);

    // TODO: Fill the M3C3 input context with value

    // TODO: Call the generated cyclic function
    // with the address of the system
    // context as a parameter

    // TODO: Display the outputs
    printf("Sliding values on 3 cycles\n");
    printf("    Mean = %lf\n",);
    printf("    Min  = %lf\n",);
    printf("    Max  = %lf\n",);
  } while (1);
  return 0;
}
```

Exercise 5: Prerequisite (C Code)

Exercise

Step 3 – Create And Execute The Binary

Move Make.bat, main.c and kcg_assign.h (C folder) to the project level

Compile the main C file you completed with all the generated files using Make.bat

Execute the binary you obtain and try to run a nominal behavior

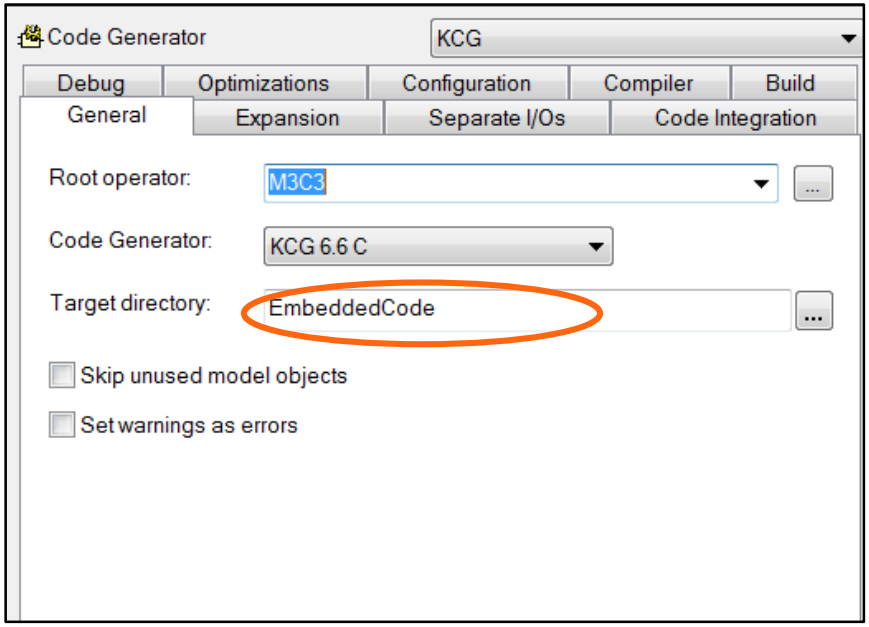
Tips

Insert into Make.bat:

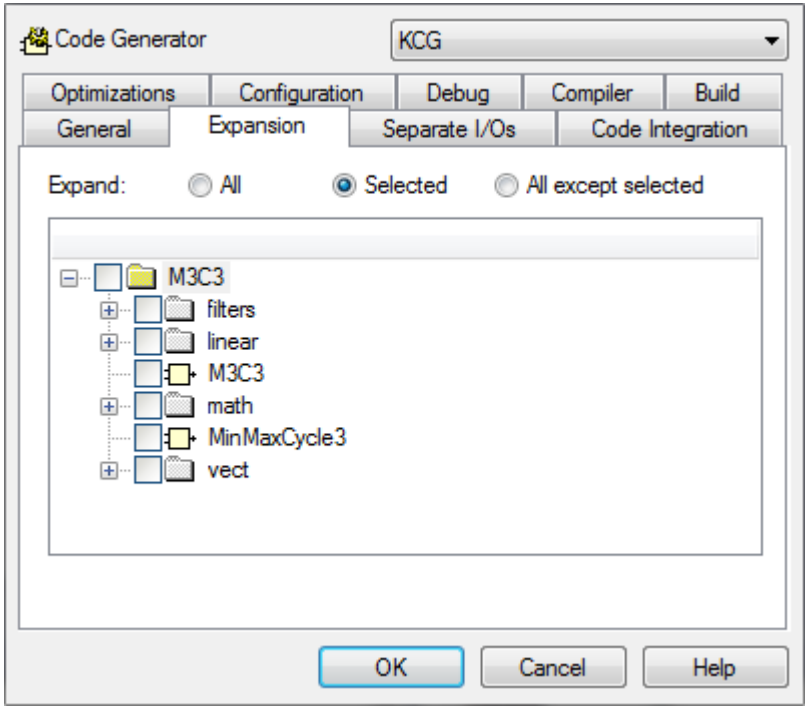
“gcc -I. -IEmbeddedCode - "<SCADE Suite installdir>" main.c EmbeddedCode*.c -o EmbeddedCode\main.exe”

Exercise 5 – Step 1: Solution (C Code)

Exercise



1



2

Exercise 5 – Step 2: Solution (C Code)

```c
// Include the header file of M3C3
#include "M3C3.h"
int main(int argc, char* argv[])
{
  // Local variable declarations
  kcg_float64 value;

  // Declare 2 variables of the M3C3
  // input/output context type
  inC_M3C3 inM3C3Context;
  outC_M3C3 outM3C3Context;
  // Call the generated initialization function
  // with the address of the system
  // context as a parameter
  M3C3_init(&outM3C3Context);

  // Make a Loop
  do
  {
    // Read the inputs
    printf("Enter a real value : ");
    scanf("%lf",&value);
    // Fill the M3C3 input context
      inM3C3Context.U = value ;
    // Call the generated cyclic function
    // with the address of the system
    // contexts as a parameter
    M3C3(&inM3C3Context, &outM3C3Context);
    // Treat the outputs
    printf("Sliding values on 3 cycles\n");
    printf("    Mean = %lf\n",outM3C3Context.MeanOn3Cycles);
    printf("    Min  = %lf\n",outM3C3Context.MinOn3Cycles);
    printf("    Max  = %lf\n",outM3C3Context.MaxOn3Cycles);
  } while (1);
  return 0;
}
```

Exercise 5 - Step 3: Solution

```
Enter a value:
1
Sliding values on 3 cycles:
Mean:  1.00000000000000E+00
Min:  1.00000000000000E+00
Max: 1.00000000000000E+00
Enter a value:
2
Sliding values on 3 cycles:
Mean:  1.33333333333333E+00
Min:  1.00000000000000E+00
Max: 2.00000000000000E+00
Enter a value:
3
Sliding values on 3 cycles:
Mean:  2.00000000000000E+00
Min:  1.00000000000000E+00
Max: 3.00000000000000E+00
Enter a value:
```

Exercise

Exercise 5 (Ada)

Ada

Exercise

Objective:

Create a program calling the Ada code generated by SCADE Suite.

Requirement:

Time: 15 min

Copy the Exercise 5 from the Prerequisite folder:

- The SCADE model computes the sliding mean, minimum and maximum values of a floating flow over 3 cycles

Generate the corresponding Ada code

Move MakeAda.bat, main.adb (Ada folder) to the project level

Complete main.adb

Compile it with all the generated files

Execute the binary you obtain and try to run a nominal behavior

ANSYS

Exercise 5: Prerequisite (Ada)

Ada

Exercise

Step 1 – Generate The Code

Generate the code according to the following requirements:

- Ada files are generated in the folder EmbeddedCodeAda
- Do not perform any expansion except for libraries

Exercise 5: Prerequisite (Ada)

Ada

Exercise

Step 2 – Complete The Main Procedure

```
-- TODO  Include package files to use

procedure Main()
is
   -- Local variable declarations
   U: Kcg_Config.Kcg_Float64:=0.0;

   -- TODO: inputs and outputs declaration

package Kcg_Float64_Text_IO is  new Ada.Text_IO.Float_IO
(Kcg_Config.Kcg_Float64);
begin

 -- TODO: Call the generated initialization function

--   Make a Loop
  while (U/=-1.0) loop
    -- Read the inputs
    Ada.Text_IO.Put_Line("Enter a real value: ");
    Kcg_Float64_Text_IO.Get(U);

    -- TODO: Call the generated cyclic function
    -- TODO: Display the outputs

   Ada.Text_IO.Put_Line("Sliding values on 3 cycles");
   Ada.Text_IO.Put_Line("Mean = "&);
   Ada.Text_IO.Put_Line("Min  = "&);
   Ada.Text_IO.Put_Line("Max  = "&);
end loop;

end Main;
```

ANSYS

Exercise 5: Prerequisite (Ada)

Ada

Exercise

Step 3 – Create And Execute The Binary

Move MakeAda.bat, main.dab (Ada folder) to the project level

Compile the main body file you completed with all the generated files using MakeAda.bat

Execute the binary you obtain and try to run a nominal behavior

Tips

Insert into MakeAda.bat:

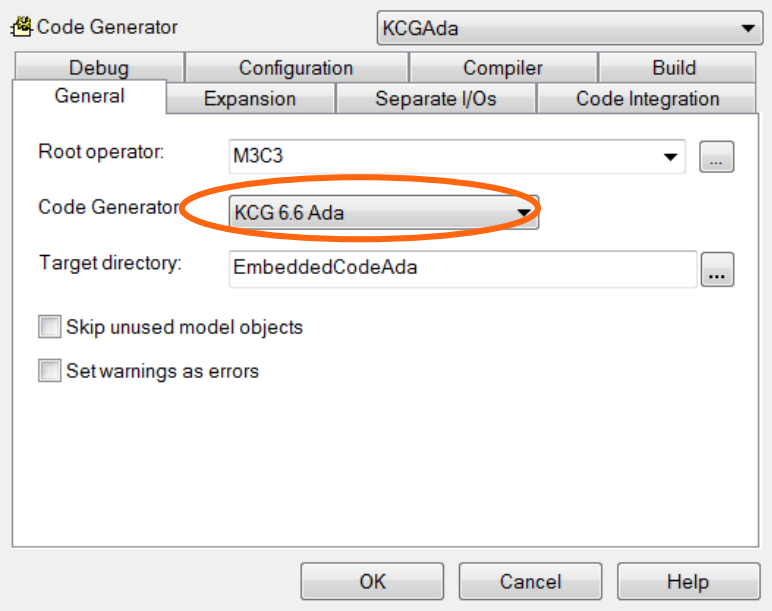
“gnatmake -I“<installation>/SCADE R17/SCADE/include/Ada" -
“<installation>/SCADE R17/SCADE/lib/Ada" -l"EmbeddedCodeAda" -D
"./obj" main.adb”

ANSYS

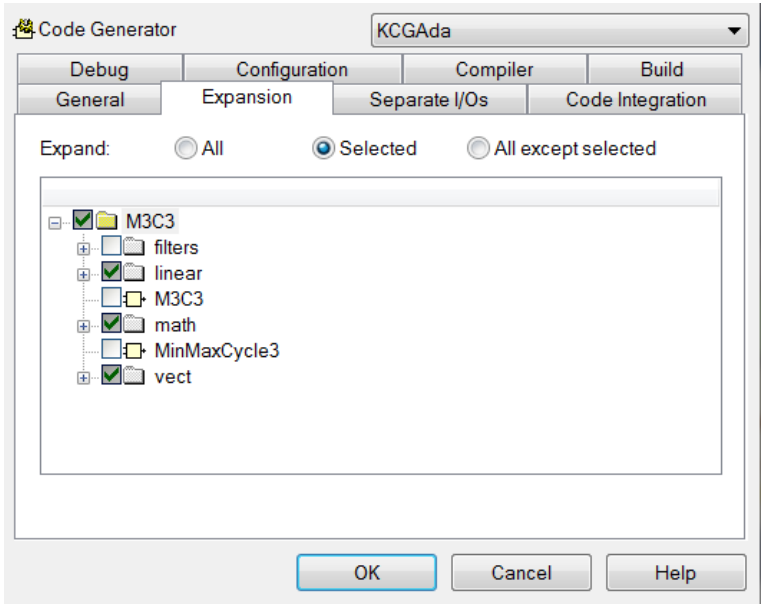
Exercise 5 – Step 1: Solution (Ada)

Ada

Exercise



1



2

Exercise 5 - Step 3: Solution

```
Enter a value:
1
Sliding values on 3 cycles:
Mean:  1.00000000000000E+00
Min:  1.00000000000000E+00
Max: 1.00000000000000E+00
Enter a value:
2
Sliding values on 3 cycles:
Mean:  1.33333333333333E+00
Min:  1.00000000000000E+00
Max: 2.00000000000000E+00
Enter a value:
3
Sliding values on 3 cycles:
Mean:  2.00000000000000E+00
Min:  1.00000000000000E+00
Max: 3.00000000000000E+00
Enter a value:
```

Exercise

How can I use legacy code?

Is it possible to use the SCADE Suite Simulator with imported code?

What do we Need?

Import torben function in the target code in SCADE Suite:

- `elem_type torben(elem_type m[],int n)`, C code
- `torben(m:Kcg_Types.Float64_Range_0_3; n:Integer) return Kcg_Config.Kcg_Float64`, Ada code

Test the SCADE Suite median design

Improve the genericity of the median operator

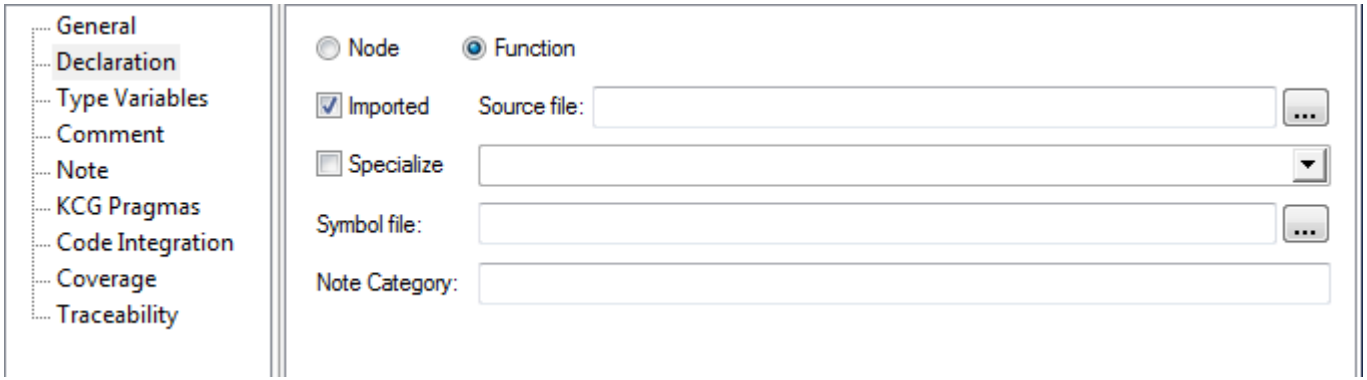
Imported Operators

They are used to implement in the host language operators not suited for SCADE Suite modeling or already existing such as legacy code:

Right-click on Operators folder in Scade View

Select *Insert > Imported Operator*

The operator can be a function or a node (with memory):



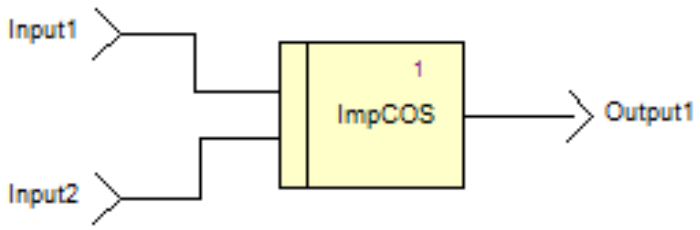
Imported Operators

The operator's name and interface are defined in SCADE Suite.

Its body is defined in the host language.

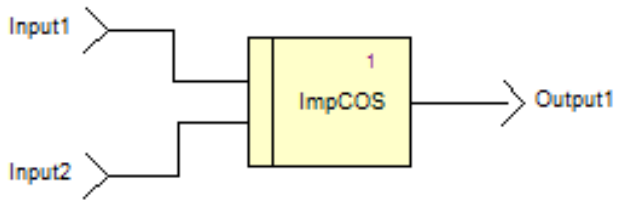
It graphically appears as a box with a vertical bar on the left hand side.

It is called in following the data flow order.



Imported Operators: Functions (C Code)

Outputs only depend on inputs:



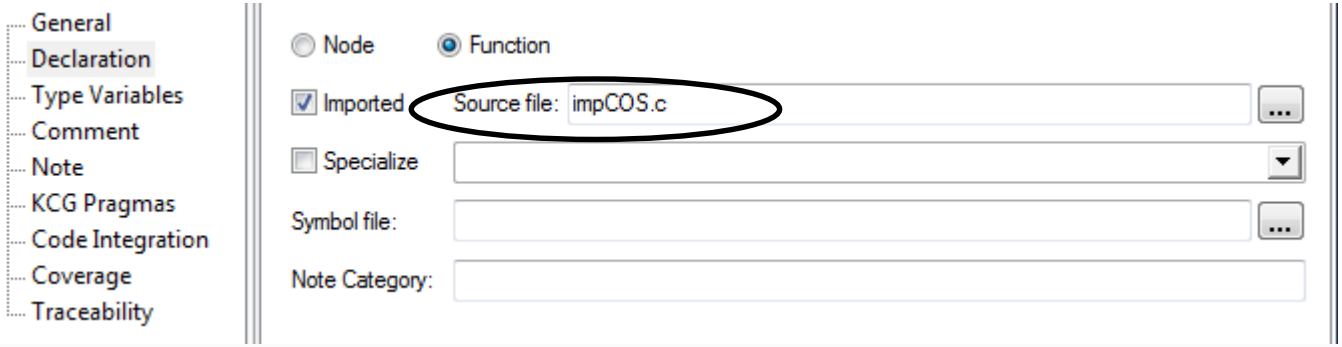

```
kcg_float32 ImpCOS(kcg_float32 Input1,kcg_float32 Input2)
{
  kcg_float32 Output1;
  Output1 = cos (Input1 + Input2);
  return Output1;
}
```

C Imported Function Implementation

KCG generates a kcg_imported_functions.h file containing the function prototypes for the imported function operators

```
#ifndef ImpCOS
/* ImpCOS */
extern kcg_float32 ImpCOS(
  /* ImpCOS::Input1 */ kcg_float32 Input1,
  /* ImpCOS::Input2 */ kcg_float32 Input2);
#endif /* ImpCOS */
#endif
```

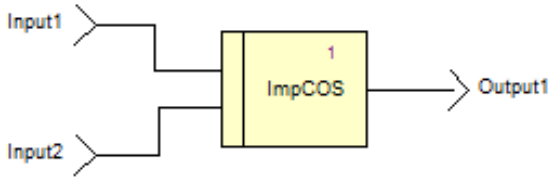
In the Properties window, specify the name of the source file for the simulation purpose



Imported Operators: Functions (Ada)

Ada

One Output only depends on inputs:



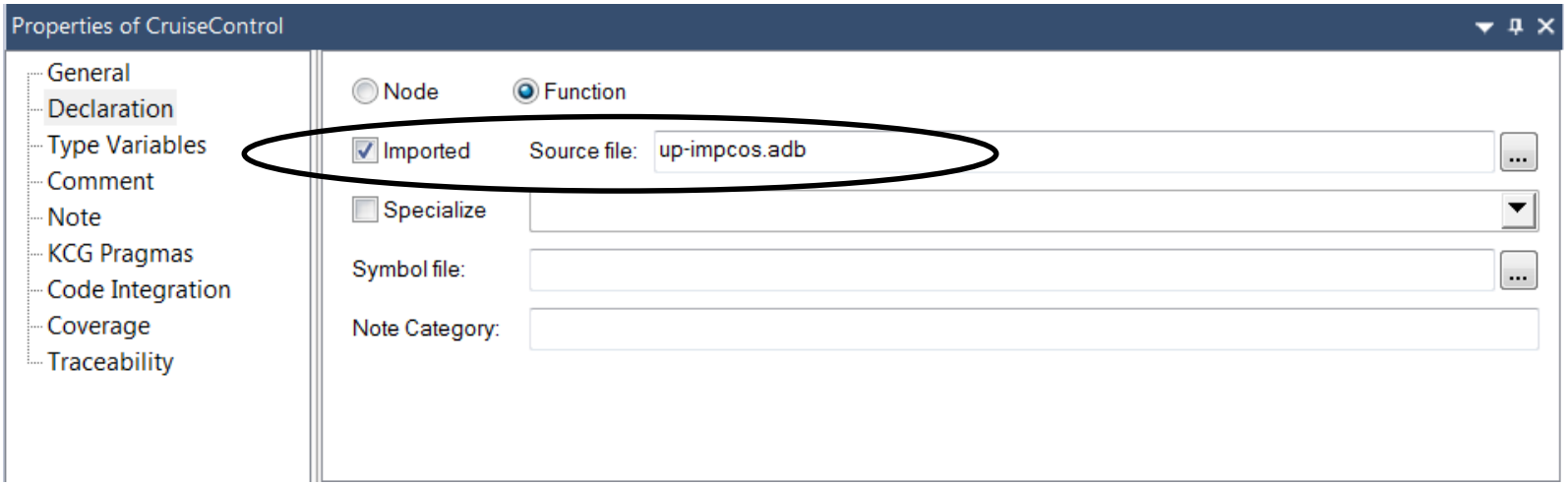

```
function ImpCOS(
  Input1 : in Kcg_Config.Kcg_float32;
  Input2 : in Kcg_Config.Kcg_float32) return Kcg_Config.Kcg_float32
is
  Output1 : Kcg_Config.Kcg_float32;
Begin
   Output1:= cos (Input1 + Input2);
  return Output1;
end ImpCOS;
```

Imported Function Implementation (Ada)

Ada

KCG generates .dads and .dadb files containing the function prototypes for the imported function operators, users must complete and move them to .adb and .ads files

In the Properties window, specify the name of the body source file for the simulation purpose



Lab 12: Imported Operator

Lab Support p.71-76

Objective:

Import and simulate code

Time: 15 min

Requirements:

Use the `torben`() function (`median_c.c` or `median_a.adb` file) to test your design

Add a test with the imported function and simulate

Lab

Lab 12: C Wrapping Function

Lab Solution

You need to encapsulate the torben function to use in your SCADE design

In Prerequisites\median_c.c file, below torben definition function, create the function:

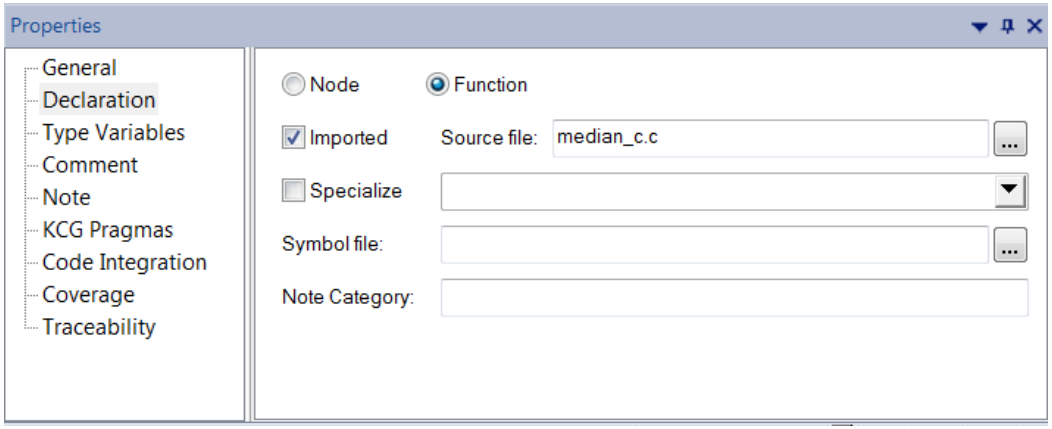
```
kcg_float64 torben_real_4_Up(kcg_float64 m[4])
{
   elem_type array[4];
   int i;
   for (i=0; i<4; i++)
     array[i] = m[i];
   return (kcg_float64) torben(array, 4);
}
```

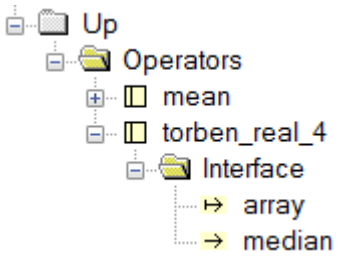
Lab 12: Imported SCADE Function (C Code)

Lab Solution

Create the imported function (into Up package)

- torben_real_4(array: float64^4) returns (median: float64)





Lab 12: Ada Wrapping Function

Ada

Lab Solution

You need to encapsulate the torben function to use in your SCADE design:

In Prerequisites\median_a.adb file, below torben definition function, create the function:

```
-- Up::torben_real_4/
function torben_real_4(
  -- array/
   array_1 : in Kcg_Types.Float64_Range_0_3) return Kcg_Config.Kcg_Float64
is

  median_2 : Kcg_Config.Kcg_Float64;

begin
  median_2 := torben(m=>array_1,n=>3);
return median_2;

end torben_real_4;
```


ANSYS

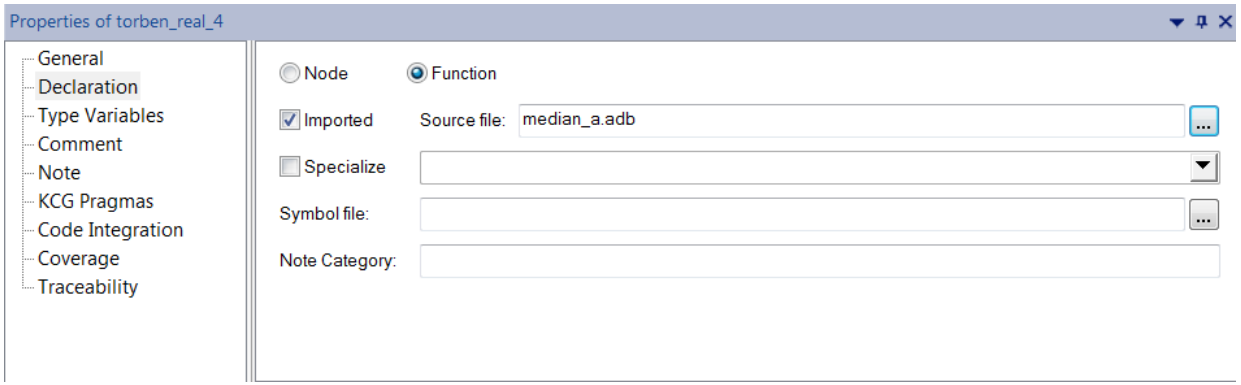
Lab 12: Imported SCADE Function (Ada)

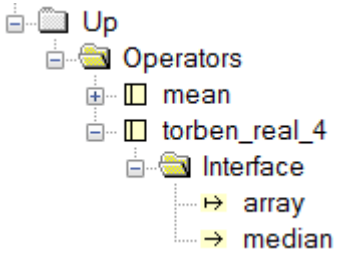
Ada

Lab Solution

Create the imported function into Up package:

- torben_real_4(array: float64^4) returns (median : float64)



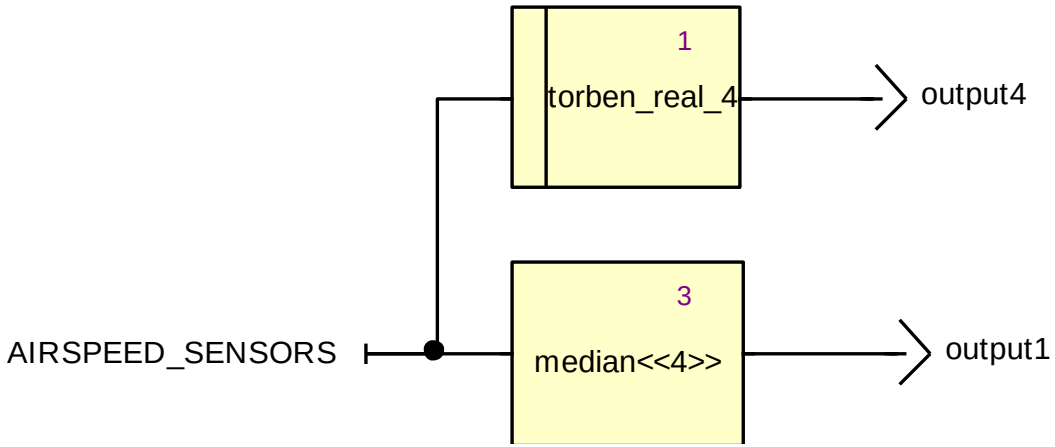


ANSYS

Lab 12: Imported SCADE Function

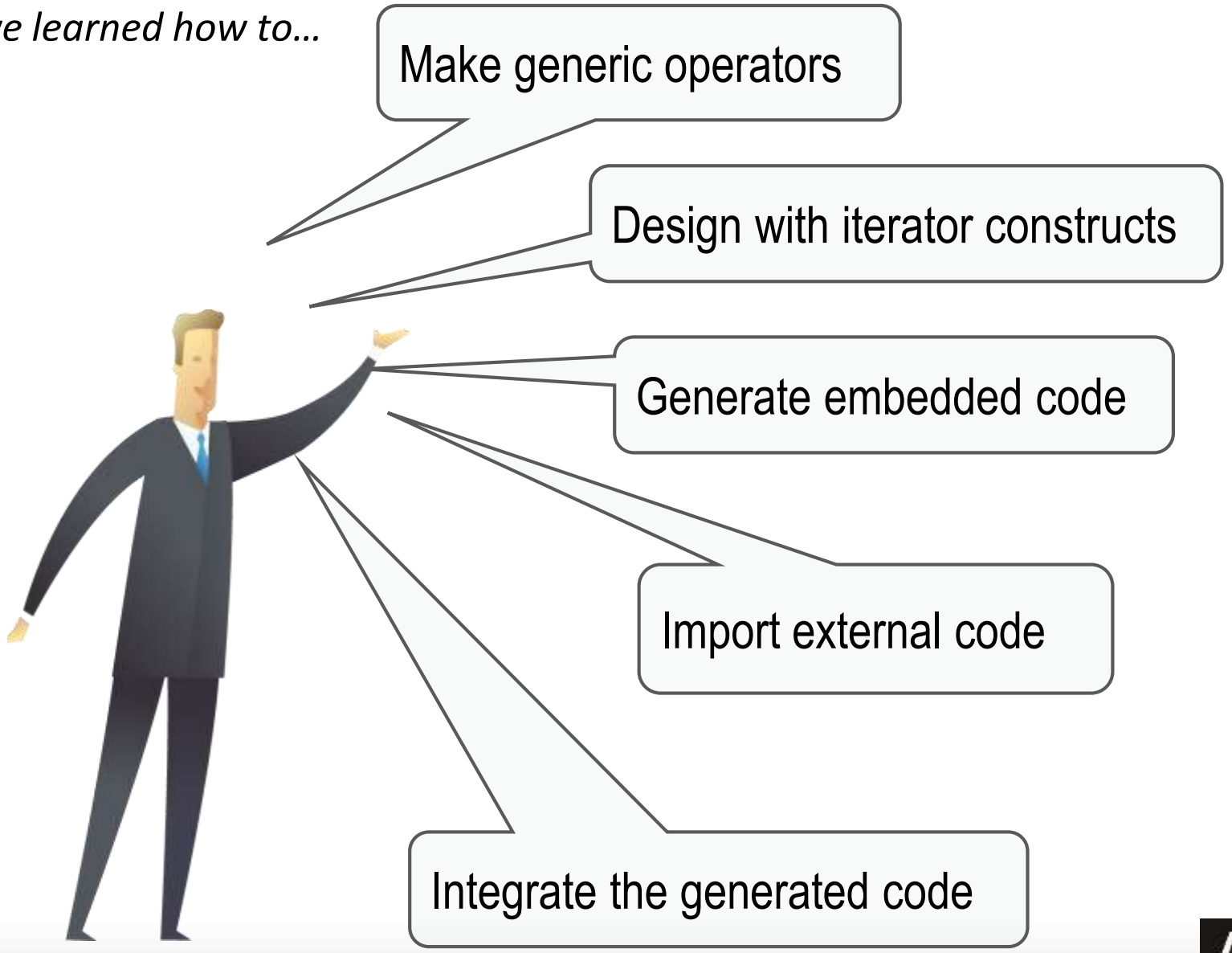
Lab Solution

Test torben_real_4 imported operator in “test” operator:



Conclusion

We have learned how to…



Contacts

Legal Contact
Esterel Technologies SAS
14/15, Place Georges Pompidou
78180 Montigny Le Bretonneux
FRANCE
Phone: +33 1 30 68 61 60
Fax: +33 1 30 68 61 61

Technical Support
Esterel Technologies SAS
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 Technical Support: scade-support@ansys.com
Contact one of our Sales representatives at: scade-sales@ansys.com
Direct general questions about Esterel Technologies to: scade-info@ansys.com
Discover the latest news on our products and technology at: http://www.ansys.com/products/embedded-software

Legal Information