



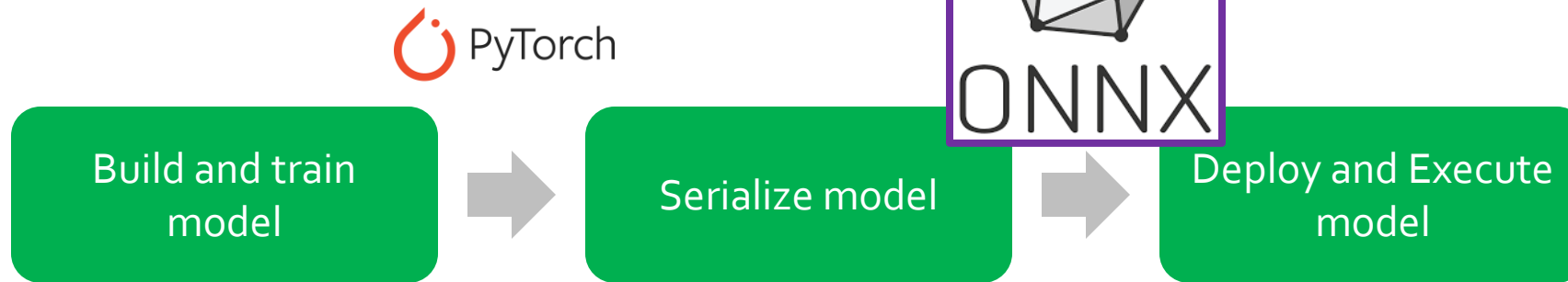
SONNX WG

Towards an ONNX profile for
critical systems

Eric Jenn (IRT), Jean Souyris (Airbus), Henri Belfy (Thales), Loïc Correnson (CEA),
Mariem Turki (IRT), Nicolas Valot (Airbus), and Franck Vedrine (CEA)

- The Context, objectives, and issues addressed...
- The Working Group (briefly)...
- Some first results...
- Status and next steps...

A Typical workflow



```
import torch
import torch.nn as nn
```

```
# Define model directly (no class)
torch_model = nn.Sequential(
    nn.Linear(4, 3),
    nn.ReLU()
)
torch_model.eval()
```

```
# Random but fixed weights for reproducibility
with torch.no_grad():
    torch_model[0].weight.copy_(torch.randn_like(torch_model[0].weight))
    torch_model[0].bias.copy_(torch.randn_like(torch_model[0].bias))
```

```
dummy = torch.randn(2, 4) # [batch=2, features=4]
```

```
# Export to ONNX
pt_onnx_path = "from_pytorch.onnx"
torch.onnx.export(
    torch_model, dummy, pt_onnx_path,
    input_names=["x"], output_names=["y"],
    opset_version=15, do_constant_folding=True
)
```

- A set of operators and a graph execution semantics
- An API
- An Intermediate Representation (IR) described using Protobuf
- A “reference implementation” coded in Python
- A runtime (ONNXruntime) [managed as a separate project in [ONNX Runtime | Home](#)]

The screenshot displays the ONNX Operators documentation page. The top section, titled "ONNX Operators", lists all operators and provides usage guides, parameters, examples, and version history. Below this, a code editor shows Python code for using the ONNX reference implementation:

```
import numpy as np
from onnx.reference import ReferenceEvaluator

X = np.array(...)
sess = ReferenceEvaluator("model.onnx")
results = sess.run(None, {"X": X})
print(results[0]) # display the first result
```

Below the Python code, a table shows the results of the execution for the "ArgMin" operator. The table has columns for the operator name, the input, and the output. The output is a 7x1 array.

Operator	Input	Output
ArgMin	And	7, 1

Below the table, a code editor shows C++ code for using the ONNX reference implementation:

```
// Additional named attributes.
repeated AttributeProto attribute = 5;

// A human-readable documentation for this node. Markdown is allowed.
optional string doc_string = 6;

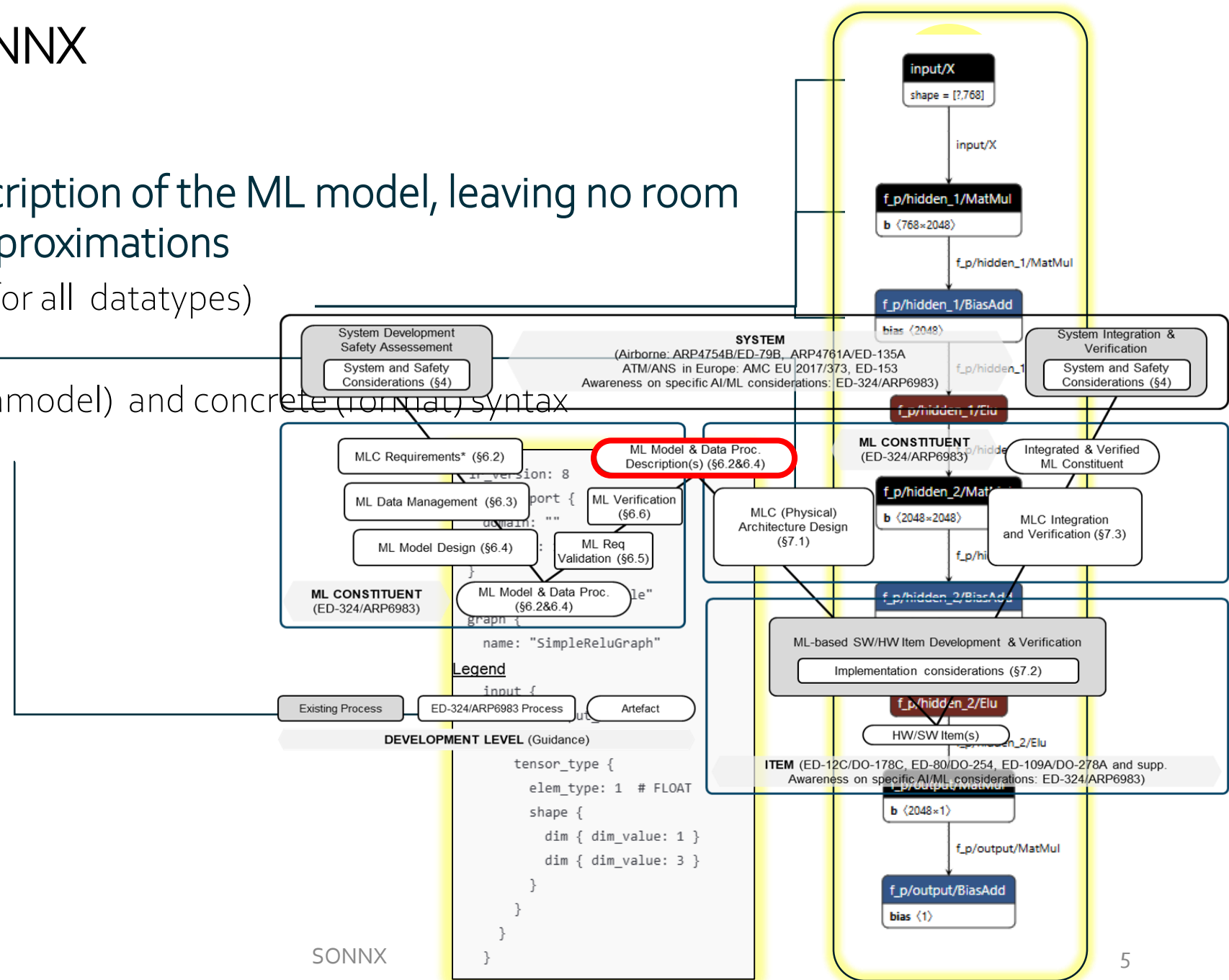
// Named metadata values; keys should be distinct.
repeated StringStringEntryProto metadata_props = 9;

// Configuration of multi-device annotations.
repeated NodeDeviceConfigurationProto device_configurations = 10;
}
```

The bottom of the screenshot shows the SONNX logo and the page number 4.

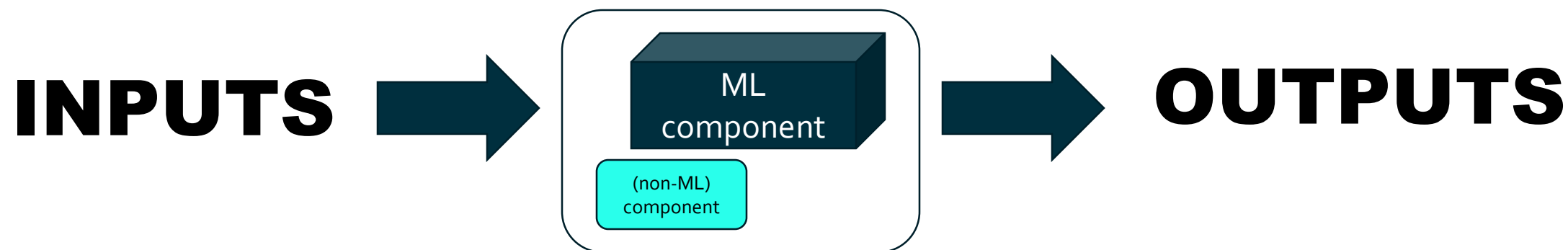
Challenges for ONNX

- Provide an accurate description of the ML model, leaving no room to interpretation and approximations
 - The operator semantics (for all datatypes)
 - The graph semantics
 - The ONNX abstract (metamodel) and concrete (normal) syntax



Why do we care?

Why not an end-to-end test?



If the system is tested **in any possible conditions** and **all tests pass** then **the component is correct**

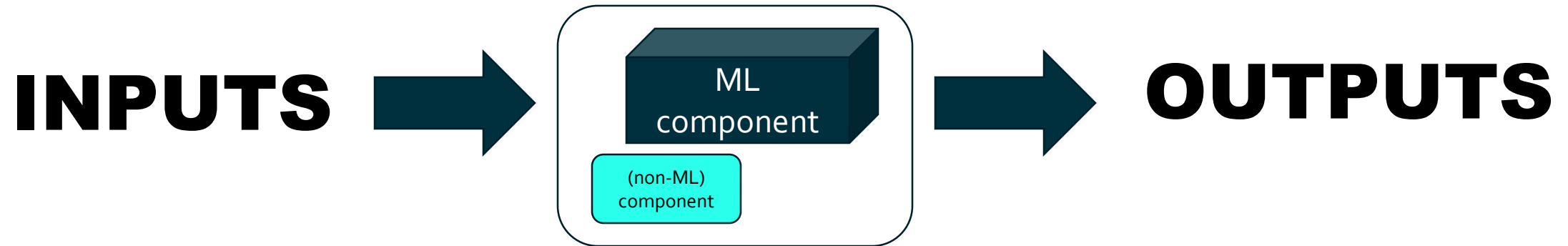
=> We don't care about how it is implemented...



- All values
- All internal states
- All temporal conditions that may affect the outputs
- ...

Why do we care?

Why not an end-to-end test?



Confidence relies on a rigorous development process, from spec to validation

Fine.

But is there **anything** to improve?

1st question: Do we understand what's "going on under the hood"?

Data types, null tensors,...

- **Non ML**
 - What is exactly an addition when using integers?
 - What is a float?
 - What are the IEEE special values?
- **What is a null tensor?**
 - What happens when a tensor becomes null?
 - Can a null tensor be "revived"?

2nd question: is there any observable issue?

ONNX failed conversion survey

- Are there empirical evidences of incompleteness, inconsistencies, etc.?
- Converters fail...
 - See Wenxin Jiang, Arav Tewari, et al, [Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters](#), Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1466–1478, Vien 2024
- ... often with the bad mode...
- ... but no root cause leading to the spec...

Finding 4. Location: Most failures are in *Node Conversion* (74%).

Finding 5. Symptom: The most common symptoms in DL model converters are *Crash* (56%) and *Wrong Model* (33%).

Finding 6. Causes: *Crashes* are largely due to *Incompatibilities* and *Type Problems*. *Wrong models* are largely due to *Type Problems* and *Algorithmic Errors*.

3rd question: are there known issues

Opset resolution, naming ambiguity

Problem: *ambiguity in opset resolution*

- An ONNX Function is a design artefact used to:
 1. define a composition of operators (ex: Relu Function is defined through Max Operator)
 2. define a composition of Nodes in the Graph as a reusable sub-graph (local function)
- Opsets are referenced in the Model element, and in each Function definition.
- Ex : Model import Opset v15,
Model local function Relu import Opset v14.



- The Opset resolution is not specified:
 - // The (domain, name, overload) tuple must be unique across the function protos in this list.
 - // In case of any conflicts the behavior (whether the model local functions are given higher priority,
 - // or standard operator sets are given higher priority or this is treated as error) **is defined by**
 - // **the runtimes.**

From [onnx/onnx-ml.proto at main · onnx/onnx · GitHub](#) , line 498-501

3rd question: are there known issues

ONNX github issues

- See discussion <https://github.com/onnx/onnx/issues/3651>
- See issues labelled topic: spec clarification

Describe the bug

When `noop_with_empty_axes == 1` & `axes` is empty, in ONNX spec, it will return input tensor directly.
But in reference in onnx, it is mismatch. it returned `np.square` of the input tensor

```
Xavier Dupré, 13个月前 | 2 authors (Xavier Dupré and others)
class ReduceSumSquare_18(OpRunReduceNumpy):
    def run(self, data, axes=None, keepdims=1, noop_with_empty_axes=0): # type: ignore
        if self.is_axes_empty(axes) and noop_with_empty_axes != 0: # type: ignore    liqun Fu, 17
            return (np.square(data),)

    axes = self.handle_axes(axes)
    keepdims = keepdims != 0 # type: ignore
```

This is complicated. Agree that there is a mismatch, but is the bug in the specification or implementation?

My personal interpretation is that this is a bug in the specification, not implementation, for the following reason: the attributes serve to define the set of axes being reduced: specifically, it is a flag to allow the empty list to indicate that all axes must be reduced (or that no axes must be reduced). Now, even if zero axes are reduced, it makes sense to compute the square. `ReduceSumSquare` is not actually a reduction-op: it is a reduction-op `Sum` applied to the square of the input.

Note: as of 2025, this issue has been corrected.

3rd question: are there known issues

ONNX github issues

- Rounding and numerical precision

- [DequantizeLinear](#) (#6132)

- $y = (x - x_zero_point) * x_scale$, with x and x_zero_point with the same dtype. What happens if $x - x_zero_point$ is outside the range of dtype?

The [onnxruntime](#) and [reference implementation](#) behave differently.

- Operator semantics

- [RandomNormal](#), [RandomUniform](#) (# 6408)

- The operator mentions a seed attribute, but doesn't said anything about its behavior. If the operator is stateless, the same value will be generated each time it is called. If it is statefull, it will generate different values, but according to the same sequence.
 - The onnxruntime and reference implementation behave differently.

3rd question: are there known issues

Laconic and lacunar documentation

Laconic: *what is a convolution?*

Conv - 22

[↑ Back to top](#)

Summary

The convolution operator consumes an input tensor and a filter, and computes the output.

(Excerpt of ONNX doc.)

No specification
of the operation

Ceil - 13

Summary

Ceil takes one input data (Tensor) and produces one output data (Tensor) where the ceil is, $y = \text{ceil}(x)$, is applied to the tensor elementwise. If x is integral, $+0$, -0 , NaN, or infinite, x itself is returned.

More or less
Reflexive definition

Lacunar: *What is the value used for padding in a convolution?*

- Uhhh... zero?

3rd question: are there identified “issues”

Handling of special values

- Clip operator

$\text{Clip}(A, L, M)$:

For all element a of tensor A :

$$\text{Clip}(a, L, M) = \min(M, \max(a, L))$$

$$A = \begin{bmatrix} -\infty & 0.0 & \infty & NaN \end{bmatrix}$$

$$L = NaN \quad M = NaN$$



$$Y = \begin{bmatrix} -\infty & 0.0 & \infty & NaN \end{bmatrix}$$



3rd question: are there identified “issues”

Handling of special values

Summary

Element-wise max of each of the input tensors (with Numpy-style broadcasting support). All inputs and outputs must have the same data type. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check [Broadcasting in ONNX](#).

■ Max operator

Max(NaN, X) = ?

IEEE754 `maximum(x, y)`

is x if $x > y$, y if $y > x$, **and a quiet NaN if either operand is a NaN [...]**



IEEE754 `maximumNumber(x, y)`

is x if $x > y$, y if $y > x$, **and the number if one operand is a number and the other is a NaN [...]**

3rd question: are there identified "issues"

Handling of special values

MaxPool operator

$$S, Ind = \text{MaxPool}(E)$$

- Data type: double
- Shape of $E = [1, 1, 3, 3]$
- kernel_shape = [2,2]
- pads = [0,0,0,0]
- dilation = [1,1]
- strides = [1,1]
- Shape of $Y = [1, 1, 2, 2]$
- Shape of $Ind = [1, 1, 2, 2]$

$$X = \left[\left[\left[\begin{array}{cc} -inf & -inf \\ -inf & -inf \end{array} \right] \right] \right]$$

$$Y = \left[\left[\left[\begin{array}{cc} -1.79769313e + 308(????) & 4.56432533e + 000 \\ 3.46789489e + 000 & 5.23979851e + 000 \end{array} \right] \right] \right]$$

$$Indices = \left[\left[\left[\begin{array}{cc} -4(Bug????) & 2 \\ 7 & 8 \end{array} \right] \right] \right]$$

Why not -inf?

Uhh? 42?

4th question: Do we have some specific constraints?

Default values

- Conv operator

Attributes

- auto_pad - STRING** (default is 'NOTSET'):
auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used.

Conv operator

4th question: Do we have some specific constraints?

Graph execution order

Problem: "ambiguity" in operator execution

- No functional ambiguity (the function is completely determined by the graph) but...
- ...Operator are executed according to dataflow constraints, which determine a **partial** order...

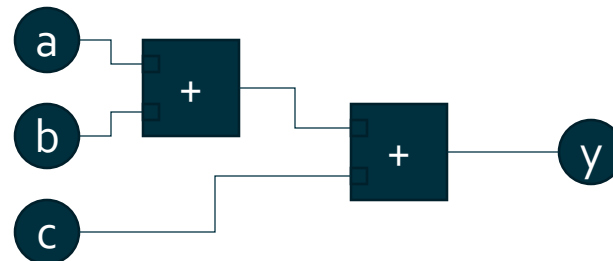
ONNX runtime

- Default execution order uses `Graph::ReverseDFS()` to generated topological sort
- Priority-based execution order uses `Graph::KahnsTopologicalSort` with per-node priority

- Note that there is no problem with associativity

$$y = a + b + c \stackrel{?}{=} (a + b) + c \stackrel{?}{=} a + (b + c)$$

Associativity is imposed
by the graph



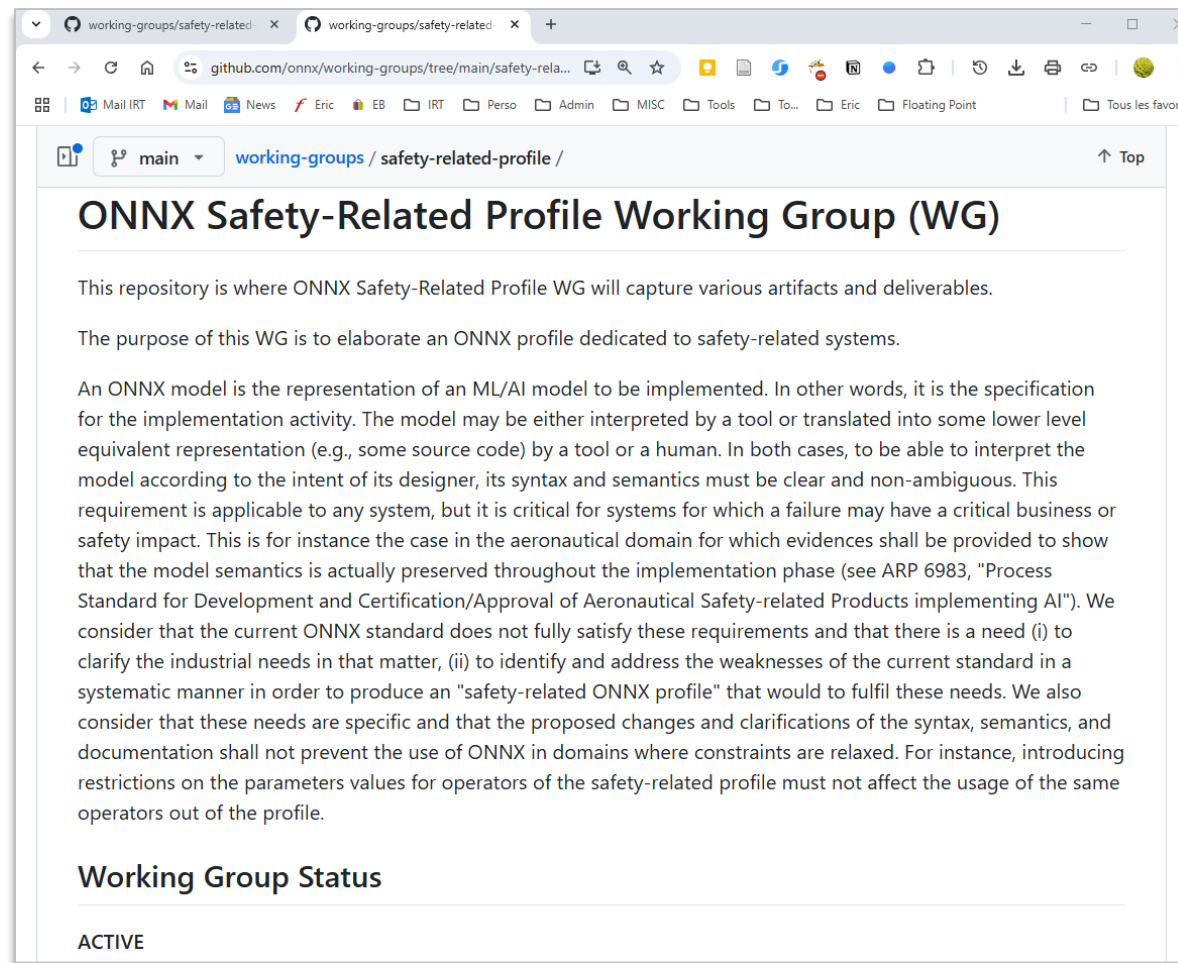
SONNX

So, there are some issues...
Back to the SONNX workgroup...

The SONNX working group

Overview

- ❑ Self-funded...
- ❑ Around 20 bi-weekly meetings
- ❑ Multiple working sessions on formal specification, reviews, etc.
- ❑ Core team of industrial and academic participants (10-15p)



(link to the repo at the end of the presentation)

(D1.a) Safety-related Profile **Scope** Definition (2024/11/01)

(D1.c) **Consolidated needs** for all industrial domains (2025/01/01)

(D2.a) ONNX safety-related Profile **requirements** (2025/02/01)

(D3.a) ONNX Safety-related profile – graph (2025/05/01)

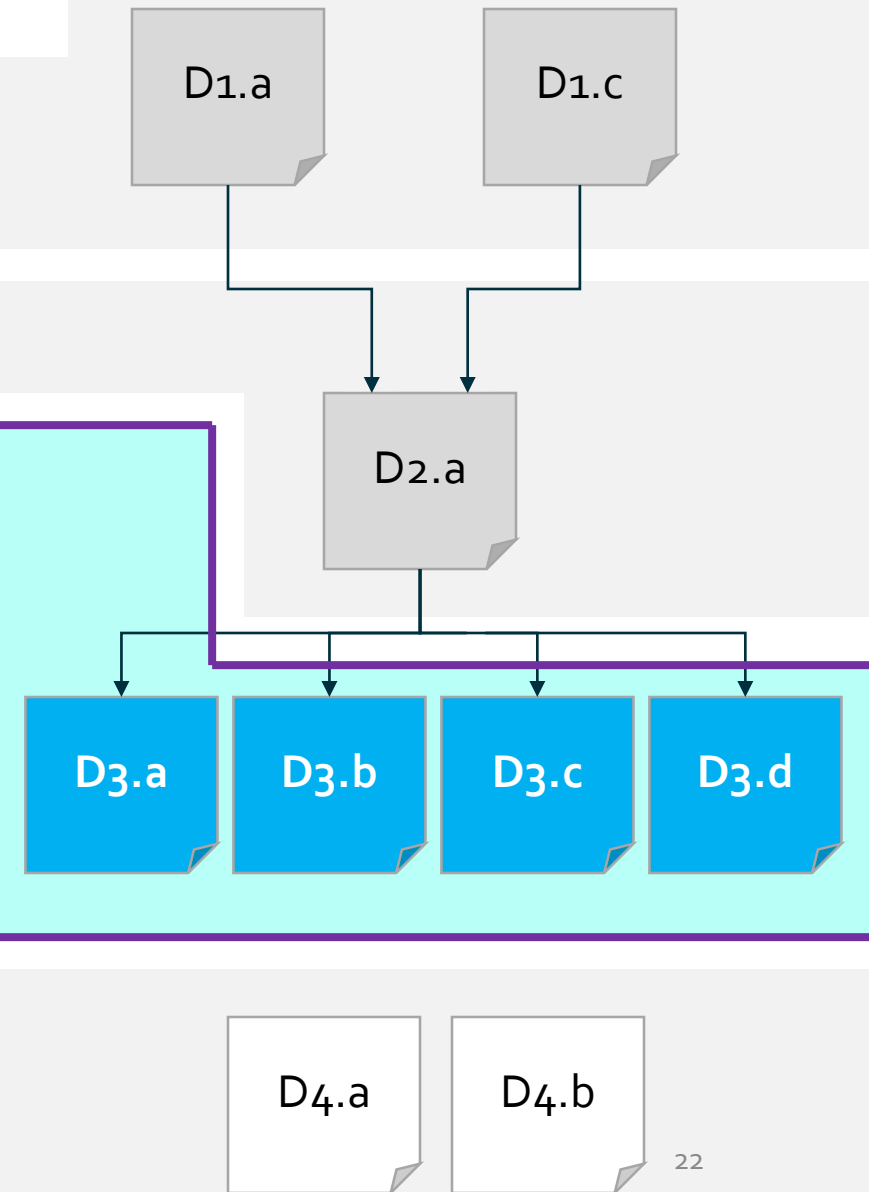
(D3.b) ONNX Safety-related profile – operators (2025/12/31)

(D3.c) ONNX Safety-related profile – format (2025/12/31)

(D3.d) ONNX Safety-related profile reference implementation (2025/12/31)

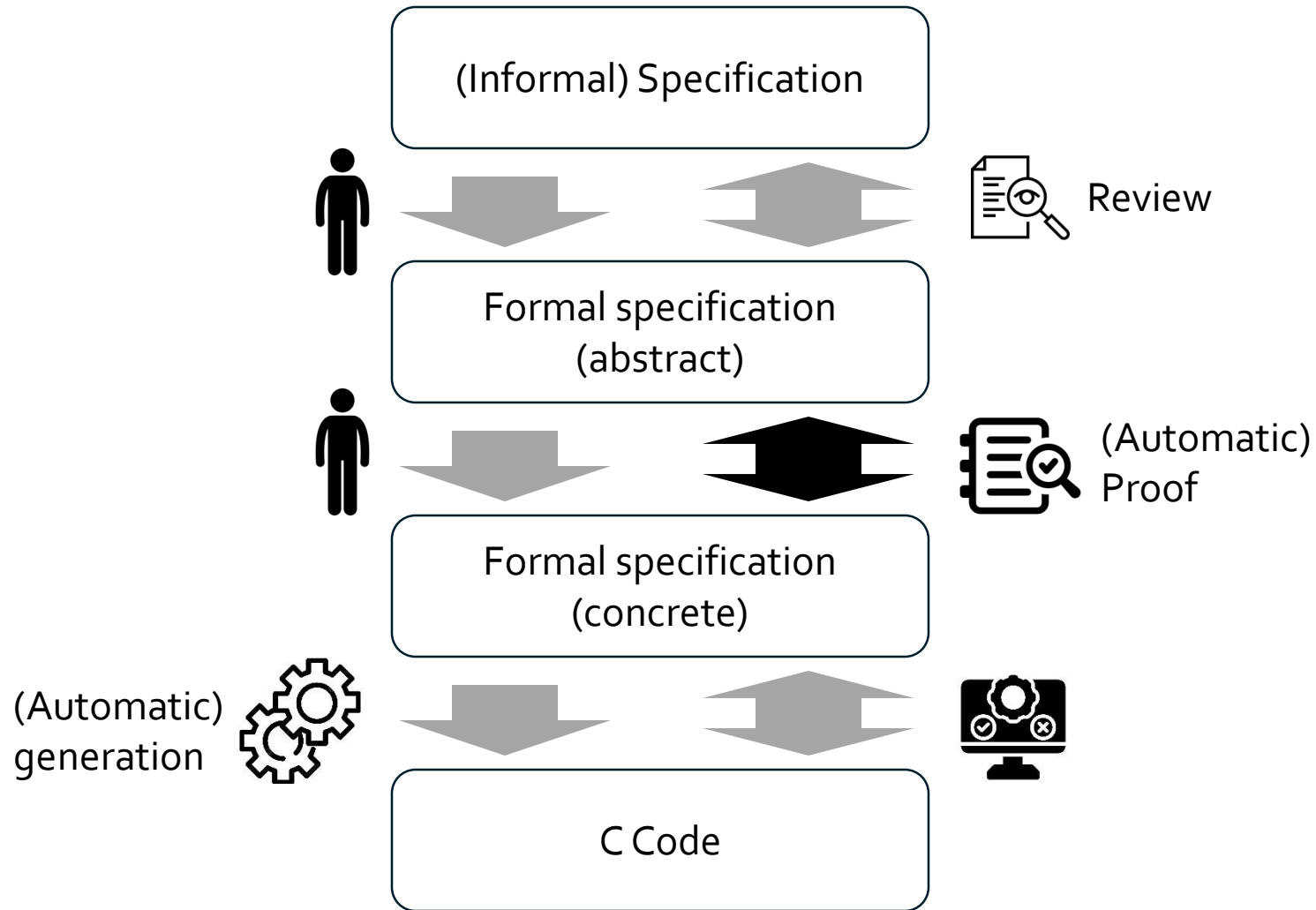
(D4.a) ONNX Safety-related profile **verification** report

(D4.b) ONNX Safety-related profile **validation** report



The SONNX workflow

From the specification to the implementation



The SONNX workflow

The **conv** operator – (Informal) specification

conv operator

Contents

- Convolution operator for type real.

Conv (real)

Signature

$Y = \text{conv}(X, W, [B])$ where

- X : input tensor
- W : convolution kernel
- B : optional bias
- Y : output tensor

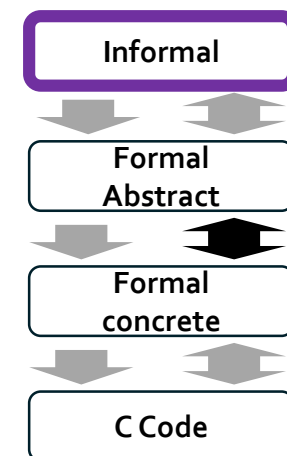
Restrictions

The following restrictions apply to the **conv** operator for the SONNX profile:

Restriction	Statement	Origin
R1	Input tensor x has 2 spatial axes	Transient
R2	Attribute <code>auto_pad</code> is set to <code>NOTSET</code>	No default values
R3	Attribute <code>group</code> is set to 1 (standard convolution) or to the number of channels of the input tensor x (depthwise convolution)	Transient

Simplification
of the WG's
work

Development
assurance



The SONNX workflow

The **conv** operator – (Informal) specification

Informal specification

Operator `conv` computes the convolution of the input tensor `x` with the kernel `w` and adds bias `b` to the result. Two types of convolutions are supported: *standard convolution* and *depthwise convolution*.

Standard convolution

A *standard convolution* applies a kernel (also called "filter") to the input tensor, aggregating information accross both spatial axes and channels. For a given output channel, the kernel operates accross all input channels and all contributions are summed to produce the output. This corresponds to the case where attribute `group` = 1.

The mathematical definition of the operator is given hereafter:

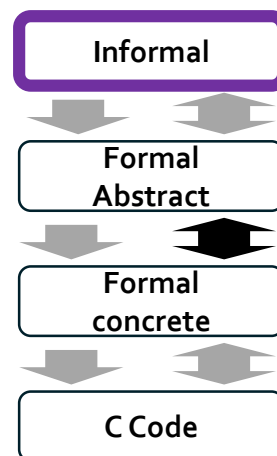
$$Y[b, c, m, n] = \sum_{i=0}^{dW_1-1} \sum_{j=0}^{dW_2-1} \sum_{z=0}^{dW_3-1} (X_p[b, i, m \cdot \text{strides}[0] + j, n \cdot \text{strides}[1] + z] \cdot W_d[c, i, j, z]) + B_b[c]$$

Where

- $b \in [0, dY_0 - 1]$ is the batch index. dY_0 is the batch size of output `y`.
- $c \in [0, dY_1 - 1]$ is the data channel. dY_1 is the number of output channels.
- $m \in [0, dY_2 - 1]$ is the index of the first spatial axis of the kernel.
- $n \in [0, dY_3 - 1]$ is the index of the second spatial axis of the kernel.
- dW_1 is the number of feature maps of the input tensor `x`.
- dW_2 is the size of the first spatial axis of kernel `w`.
- dW_3 is the size of the second spatial axis of kernel `w`.
- `strides` is an attribute of the operator. It will be described later in this section.
- $X_p = \text{pad}(X, \text{pads})$ is the padded version of the input tensor `x`. Function `pad` applies zero-padding as specified by the `pads` attribute (see ONNX `Pad` operator).
- $W_d = \text{dilation}(W, \text{dilations})$ is the dilated version of the kernel `w`. Function `dilation` expands the kernel by inserting spaces between its elements as specified by the `dilations` attribute. Its definition is given later.
- $B_b = \text{broadcast}(B, (dY_0, dY_1, dY_2, dY_3))$ is the broadcasted version of bias `b`. Function `broadcast` replicates the bias value across the spatial dimensions and batch dimension of the output `y`. It takes as argument the bias `b` and the shape of output `y`. Its definition is given later.

Simple,
"naïve"
formulation

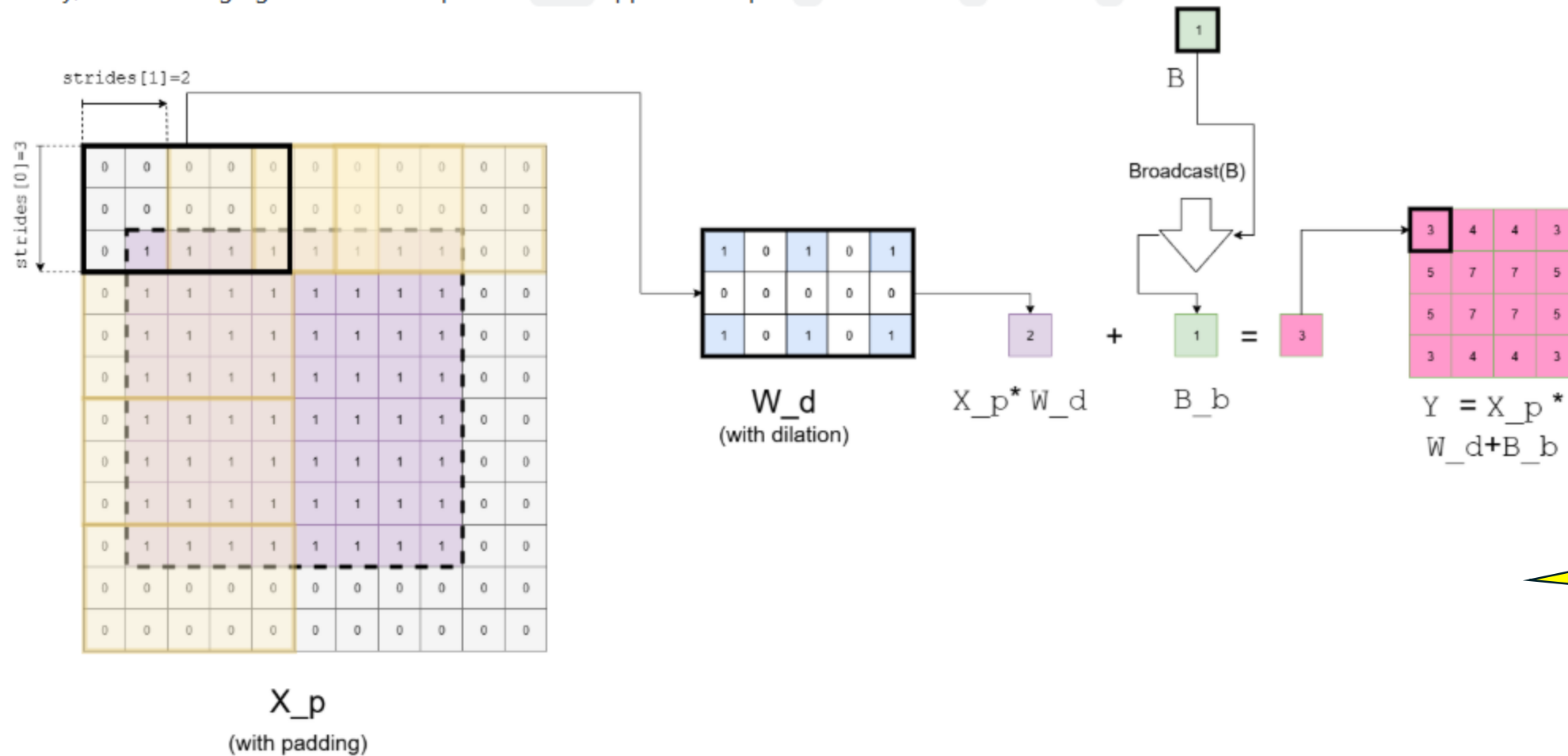
$$Y[b, c, m, n] = \sum_{i=0}^{dW_1-1} \sum_{j=0}^{dW_2-1} \sum_{z=0}^{dW_3-1} (X_p[b, i, m \cdot \text{strides}[0] + j, n \cdot \text{strides}[1] + z] \cdot W_d[c, i, j, z]) + B_b[c]$$



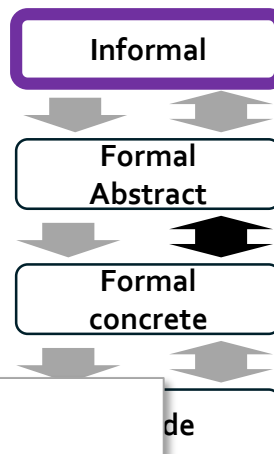
The SONNX workflow

The **conv** operator – (Informal) specification

Finally, the following figure illustrates operator **conv** applied on input **x** with kernel **w** and bias **b**:



Illustration



X : tensor of real

Tensor X is the input tensor on which convolution with kernel W is computed.

The shape

- dX_1
- dX_2
- dX_3

Constraints

- C1

- **C1** : Number of spatial axes of tensor X
 - Statement: The number of spatial axes of tensor X is 2. R1
 - Rationale: This restriction is introduced to reduce the specification effort. It matches the industrial use cases considered in the profile.

- Rationale: This restriction is introduced to reduce the specification effort. It matches the industrial use cases

- C2
- C3

- **C3** : Consistency between the shape of tensors X , W , Y and attributes $pads$, $dilations$ and $strides$

- Statement:

$$\left\lfloor \frac{\alpha - ((dilations[0] \cdot dW_2 - 1) + 1)}{strides[0]} \right\rfloor + 1 = dY_2 \text{ with } \alpha = dX_2 + pads[0] + pads[2]$$

and

$$\left\lfloor \frac{\beta - ((dilations[1] \cdot dW_3 - 1) + 1)}{strides[1]} \right\rfloor + 1 = dY_3 \text{ with } \beta = dX_3 + pads[1] + pads[3]$$

Constraints relating arguments and attributes

- How to specify error conditions
- Examples
 - Addition

y=Add(a: int32, b: int32)

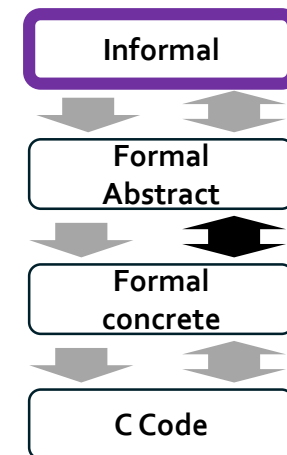
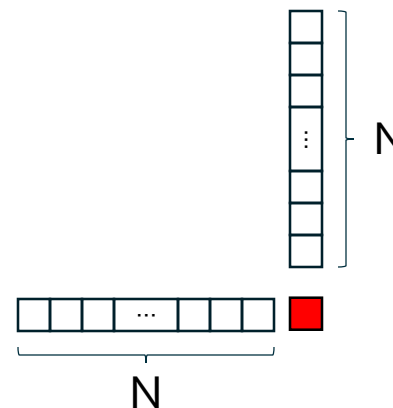
$$-2^{32} \leq a + b \leq 2^{32} - 1$$

Or, more conservatively,

$$-2^{31} \leq a \leq 2^{31} - 1 \text{ and } -2^{31} \leq b \leq 2^{31} - 1$$

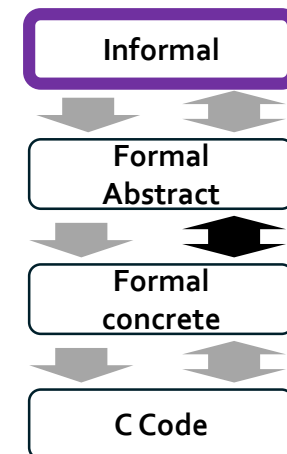
- Matrix multiplication (e.g., MatMulInteger)
Condition can be expressed on the shape of the tensors

$$N > \frac{2^{32} - 1}{128^2} \approx 133141.5$$



Failure conditions

The add operator



Add (real, real)

$$Y[i] = A[i] + B[i]$$

Add (float, float)

$$Y[i] = A[i] \oplus B[i]$$

Add (int, int)

For unsigned values (type `UINTn`):

$$Y[i] = \begin{cases} A[i] + B[i] - k \cdot 2^n & \text{if } A[i] + B[i] > 2^n - 1 \\ A[i] + B[i] & \text{otherwise} \end{cases}$$

For signed values (type `INTn`):

$$Y[i] = \begin{cases} A[i] + B[i] - k_1 \cdot 2^n & \text{if } A[i] + B[i] > 2^{n-1} - 1 \\ A[i] + B[i] + k_2 \cdot 2^n & \text{if } A[i] + B[i] < -2^{n-1} \\ A[i] + B[i] & \text{otherwise} \end{cases}$$

Actual semantics

To be checked ...

The SONNX workflow

The **conv** operator – Formal specification:
abstract **tensors**

(** Formalization of Tensor *)

module Tensor

use int.Int

use map.Map

use list.List

use Range

type data 'a = map (list int) 'a

type tensor 'a = {

 dims : list int ;

 data : data 'a ;

 background : 'a ; (* default value, or value for 0-dimensions tensor *)

}

(** Constant Tensor *)

let ghost function const (v : 'a) (bg : 'a) (ds : list int): tensor 'a

 ensures { result.dims = ds }

 requires { positive ds }

 ensures { result.background = bg }

 ensures { forall k. valid k ds -> result k = v }

 (*proof*)

 = { dims = ds ; data = pure { fun k -> if valid k ds then v else bg } ; background = bg }

 (*qed*)

(** Constant & Null *)

goal zero_is_const: forall e : 'a, ds. positive ds -> zero e ds == const e e ds

end

type tensor 'a = {

 dims : list int ;

 data : data 'a ;

 background : 'a ; (* default value, or value for 0-dimensions tensor *)

}

A abstract map from
an index to a value

Informal

Formal
Abstract

Formal C

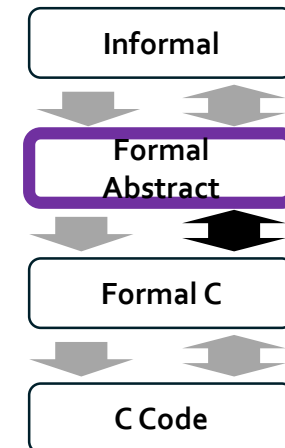
C Code

Why3 platform

Inria

Formal specification and verification

The **conv** operator – Formal specification:
the operation



```

let function conv2d_int (x: tensor int) (w: tensor int) (b: option (tensor int))
    (strides pads dilations: seq int)
    (group_val: int)
    (auto_pad_is_not_set: bool)
    : tensor int

```

- **c1** : Number of spatial axes of tensor **X**
 - Statement: The number of spatial axes of tensor **X** is 2. **R1**
 - Rationale: This restriction is introduced to reduce the specificity considered in the profile.

The formal
expression of
constraints

The informal
expression of
constraints

```

{ Ops4D.n_dim w > 0 }
requires { Ops4D.n_dim x > 0 }

```

```

(* --- Attribute Sequence Length Requirements --- *)
requires { Seq.length strides = 2 }
requires { Seq.length pads = 4 }
requires { Seq.length dilations = 2 }

```

```

ONNX Profile Restrictions ---
requires { group_val = 1 }
requires { auto_pad_is_not_set }

```

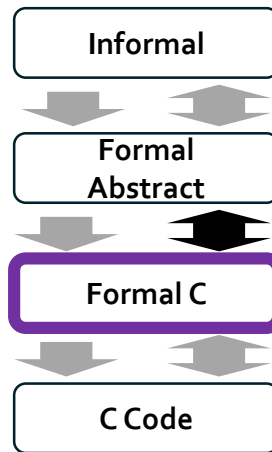
```

(* --- Conditional Bias Tensor Constraints --- *)
requires { match b with
    | None -> true
    | Some b_tensor ->
        dim b_tensor = 1
    end
}

```

Formal specification and verification

The **conv** operator – Formal specification:
concrete **tensors**



```
type farray = ptr float
```

```
type ctensor = {
  t_rank : int32 ;
  t_dims : iarray ;
  t_data : farray ;
}
```

```
function tensor_dim (t : ctensor) : list int
function tensor_size (t : ctensor) : int = ...
predicate valid_index (k : list int) (t : ctensor) = valid k (tensor_dim t)
predicate empty_tensor (t : ctensor) = t.t_rank == 0
```

```
predicate valid_tensor (t : ctensor) =
  dimension t.t_dims t.t_rank /\
  valid_range t.t_data 0 (tensor_size t) /\
  writable t.t_data
```

```
let ctensor_add (a b r : ctensor)
  requires { valid_tensor a }
  requires { valid_tensor b }
  requires { valid_tensor r }
  requires { tensor a ~= tensor b ~= tensor r }
  ensures { tensor r = opadd (tensor a) (tensor b) }
```

```
type ctensor = {
  t_rank : int32 ;
  t_dims : iarray ;
  t_data : farray ;
}
```

A concrete array
containing the
values

```
let ctensor_add (a b r : ctensor) =
  requires { valid_tensor a }
  requires { valid_tensor b }
  requires { valid_tensor r }
  requires { tensor a ~= tensor b ~= tensor r }
  ensures { tensor r = opadd (tensor a) (tensor b) }
```

The concrete
addition

Relation between
the abstract and the
concrete addition

Formal specification and verification

The **conv** operator – Formal specification:
the operation

```
let function conv2d_output_value (x w: tensor int) (b: option (tensor int))
  (strides pads dilations: seq int)
  (out_shape_param: Shape.shape)
  : (Index.index -> int)
```

The formal
specification of the
operation

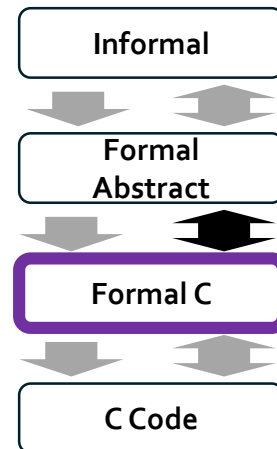
```
let rec function sum_over_c_in (x tensor w_tensor: tensor int)
  (pads_attr dilations_attr strides_attr: seq int)
  (n_for_x c_out_for_w h_out_val w_out_val c_in_iter_for_both: int) : int
```

```
let rec function sum_over_kh (x tensor w_tensor: tensor int)
  (pads_attr dilations_attr strides_attr: seq int)
  (n_for_x c_out_for_w c_in_for_both h_out_val w_out_val kh_iter_for_w: int) : int
```

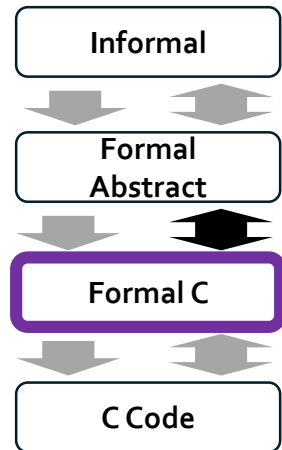
```
let rec function sum_over_kw (x tensor w_tensor: tensor int)
  (pads_attr dilations_attr strides_attr: seq int)
  (n c_out c_in h_out w_out kh kw_iter: int) : int
```

$$Y[b, c, m, n] = \sum_{i=0}^{dW_1-1} \sum_{j=0}^{dW_2-1} \sum_{z=0}^{dW_3-1} (X_p[b, i, m \cdot \text{strides}[0] + j, n \cdot \text{strides}[1] + z] \cdot W_d[c, i, j, z]) + B_b[c]$$

The informal
specification of the
operation



The **conv** operator – Formal specification: the operation



```
let ctensor_conv2d (x : ctensor) (w : ctensor) (b : ctensor)
  (pad_top pad_bottom pad_left pad_right : int32)
  (dil_h dil_w : int32) (str_h str_w : int32)
  (output : ctensor): unit =
```

```
(* Postcondition: The concrete tensor 'output' must be equal to the ghost specification of Conv2D. *)
ensures { tensor output = OPConv2d.opconv2d (tensor x) (tensor w) (tensor b)
  (Int32.to_int pad_top) (Int32.to_int pad_bottom)
  (Int32.to_int pad_left) (Int32.to_int pad_right)
  (Int32.to_int dil_h) (Int32.to_int dil_w)
  (Int32.to_int str_h) (Int32.to_int str_w) }
```

```

for n = 0 to n_batches - 1 do (* Loop over Batch dimension (N) *)
  invariant { true }
  for m = 0 to m_out - 1 do (* Loop over Output Channels (M) *)
    invariant { true }
    for oh = 0 to h_out - 1 do (* Loop over Output Height (oH) *)
      invariant { true }
      for ow = 0 to w_out - 1 do (* Loop over Output Width (oW) *)
        invariant { true }
        (* Initialisation de l'accumulateur pour le produit scalaire (dot product) *)
        let conv_sum = ref (f64 0.0) in
          .../...

```

The concrete
specification of the
operation

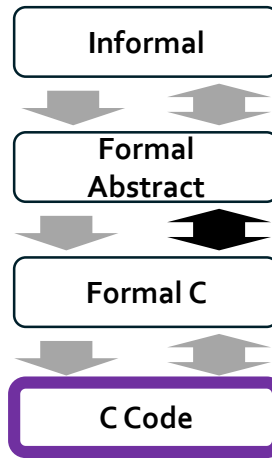
Formal specification and verification

The **conv** operator – Code generation

```
void ctensor_conv2d(struct ctensor x, struct ctensor w, struct ctensor b,
                   int32_t pad_top, int32_t pad_bottom, int32_t pad_left,
                   int32_t pad_right, int32_t dil_h, int32_t dil_w,
                   int32_t str_h, int32_t str_w, struct ctensor output) {
    int32_t n_batches, c_in, h_in, w_in, m_out, kh, kw, h_out, w_out,
    pad_top_i, pad_left_i, dil_h_i, dil_w_i, str_h_i, str_w_i;
    int32_t* x_coords;
    int32_t* w_coords;
    int32_t* b_coords;
    int32_t* output_coords;
    int32_t n, o, m, o1, oh, o2, ow, o3, c, o4, k_h, o5, k_w, o6, ih, iw,
    m_out_i, pad_top_i, pad_left_i, dil_h_i, dil_w_i, str_h_i, str_w_i;

    // ... (omitted code) ...

    if (x_coords && (w_coords && (b_coords && output_coords))) {
        o = n_batches - 1;
        if (0 <= o) {
            for (n = 0; ; ++n) {
                o1 = m_out - 1;
                if (0 <= o1) {
                    for (m = 0; ; ++m) {
                        o2 = h_out - 1;
                        if (0 <= o2) {
                            for (oh = 0; ; ++oh) {
                                o3 = w_out - 1;
                                if (0 <= o3) {
                                    for (ow = 0; ; ++ow) {
                                        conv_sum = ((double) 0.0);
                                        // ... (omitted code) ...
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```



- Naïve implementation
 - Simple, traceable to the specification but very slow...
 - Performance
- Tests shall be conducted on small tensors / kernels

Numerical errors

Example: the **add** operator

Numerical Accuracy

If tensor A_{err} is the numerical error of **A**, tensor B_{err} is the numerical error of **B**, let us consider $C_{\text{err}}^{\text{propag}}$ the propagated error of **Add** and $C_{\text{err}}^{\text{intro}}$ the introduced error of **Add**. Hence the numerical error of **C**,
 $C_{\text{err}} = C_{\text{err}}^{\text{propag}} + C_{\text{err}}^{\text{intro}}$.

Error propagation

For every indexes $I = (i_0, i_1, \dots, i_n)$ over the axes,

- $C_{\text{err}}^{\text{propag}}[I] = A_{\text{err}}[I] + B_{\text{err}}[I]$

Error introduction - floating-point IEEE-754 implementation

The error introduced by the **Add** operator shall be bound by the semi-ulp of the addition result for every tensor component for a normalized result. For a hardware providing m bits for floating-point mantissa, the semi-ulp of **1.0** is $2^{-(m+1)}$. Hence, for every indexes $I = (i_0, i_1, \dots, i_n)$ over the axes,

- $|C_{\text{err}}^{\text{intro}}[I]| \leq \max \left(|A[I] + B[I] + A_{\text{err}}[I] + B_{\text{err}}[I]| \times 2^{-(m+1)}, \frac{\text{denorm-min}}{2} \right)$
- $|C_{\text{err}}^{\text{intro}}[I]| \leq \max \left(|A_{\text{float}}[I] + B_{\text{float}}[I]| \times 2^{-(m+1)}, \frac{\text{denorm-min}}{2} \right)$
- $|C_{\text{err}}^{\text{intro}}[I]| \leq \max \left(|A[I] + B[I]| \times \frac{2^{-(m+1)}}{1 - 2^{-(m+1)}}, \frac{\text{denorm-min}}{2} \right)$

Hybrid unit checker

assertion

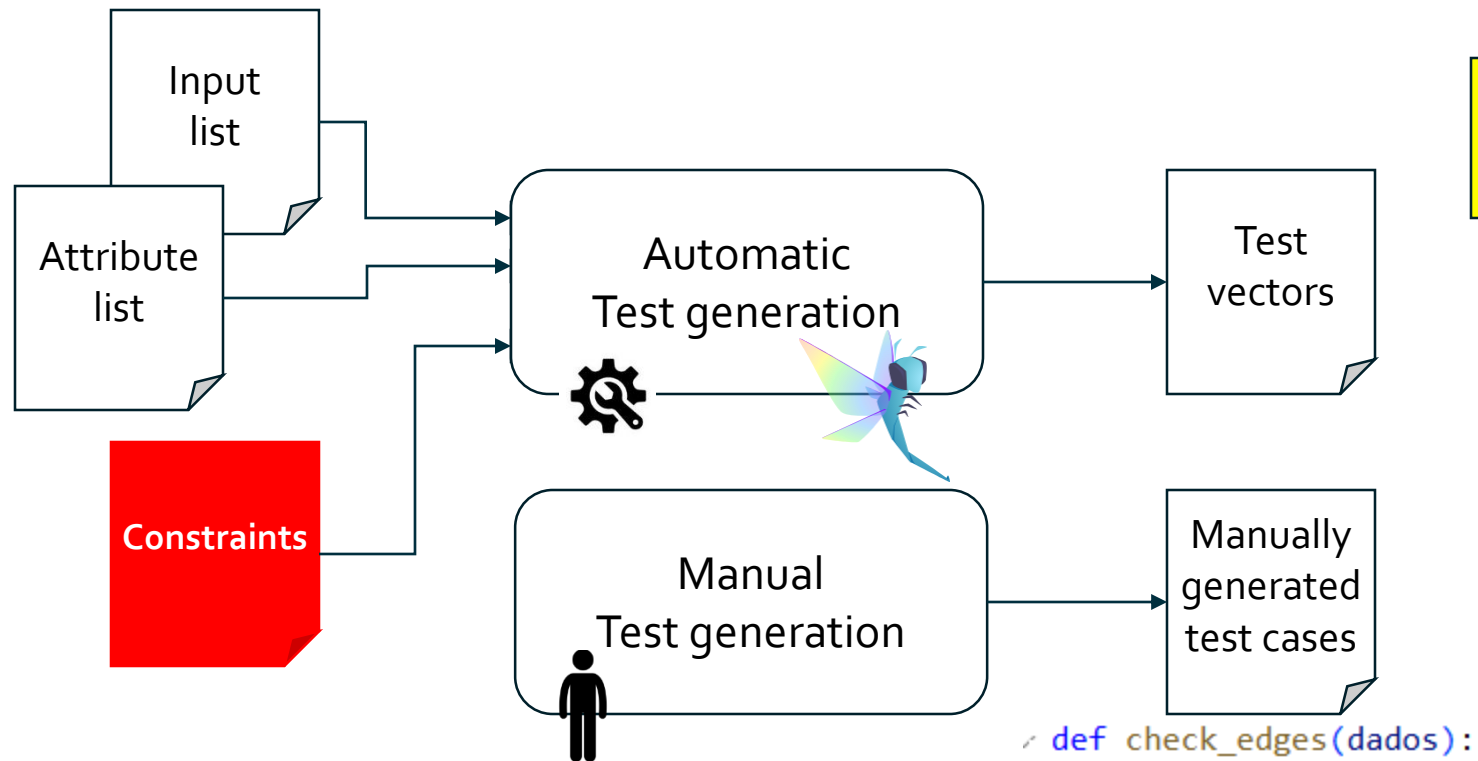
C code

Testing

Generating test vectors with Hypothesis

Draw samples according to some distribution

Use dedicated backends : hypothesis, hypofuzz, **crosshair**



Coverage
guided
fuzzing

SMT solver

Conclusion

Where are we? What's next?

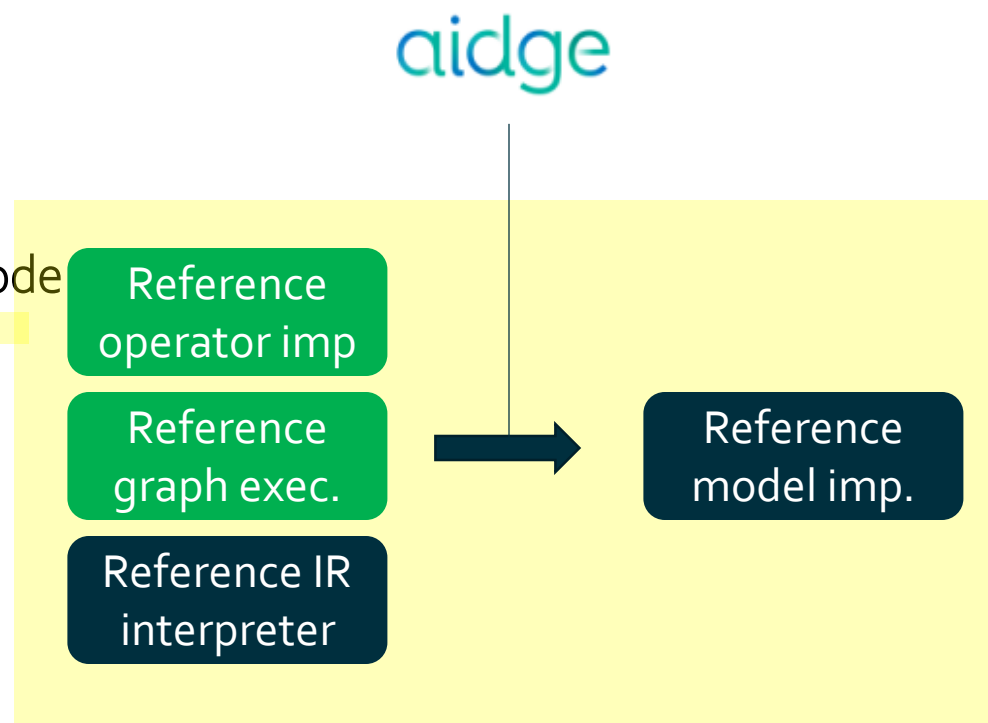
■ Where are we...

- First set of (informal) spec available...
- First code generation
- Formal specific on-going (concat, conv,...)

■ What's next...

- Completion of operator informal and formal spec + proof + code
- Completion graph spec + proof + code gen
- Generation of tests
- Integration in the ONNX ecosystem...

■ Integration to the platform



We Need You!



SONNX github repo

Eric JENN (eric.jenn@irt-saintexupery.com)

Jean SOUYRIS (jean.souyris@airbus.com)