Specification of ONNX operator: add

COMMERCIAL AIRCRAFT

Salomé Marty Laurent - 1YYWA

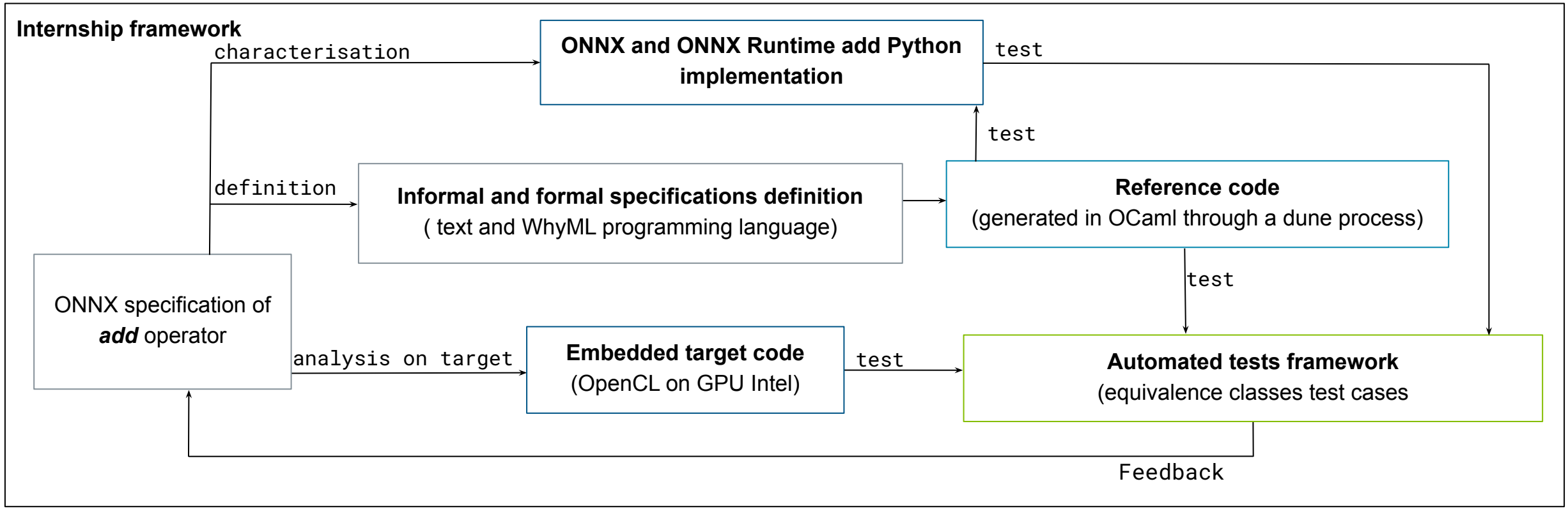
17/07/2025

AIRBUS

Agenda

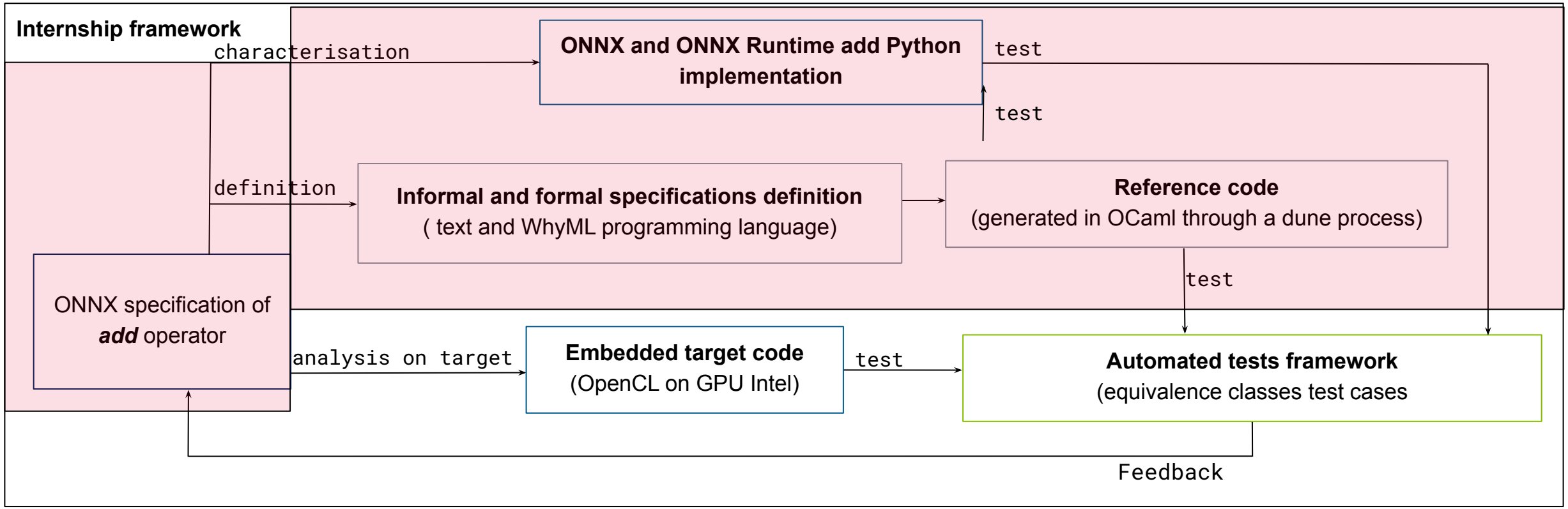
- Global workflow
- General ONNX specification
- SONNX specification
- Reference code and validation tests
- Conclusion

Global workflow



- Closed loop process outlined above for the ONNX ***add*** operator serves as a template.
- Future work will involve repeating this methodology for additional key ONNX operators such as ***reLU, maxPool, and resize***, progressively building a library of formally verified and tested implementations.

Today's topic



- Today's topic, highlighted by the red outline, centers on studying the ONNX specification of ***add***, then looking at the informal and formal specifications (our SONNX specifications) for the ONNX ***add*** operator and finishing by the reference code generation.

ONNX ***add*** operator specification

Definition:

Performs element-wise binary addition (with Numpy-style broadcasting support).This operator supports multidirectional (i.e., Numpy-style) broadcasting;

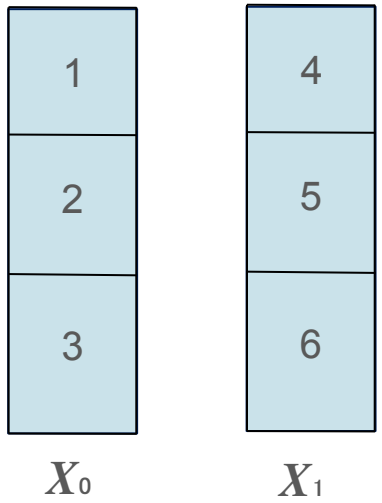
- Focus on ONNX ***add*** operator **:**
  - **Version 14**
- **Inputs / output :**
  - **Binary operator**
  - **Input(s) :** 2 tensors
  - **Output :** tensor
- **Main data types supported :**
  - **12 types :** `tensor(bfloat16) … tensor(uint8)` including `tensor(int8)`, `tensor(uint8)`, `tensor(int16) and tensor(uint16)`

- To fully grasp the ONNX ***add*** operator, its specification is best understood alongside the ONNX Runtime implementation and the Numpy library.
- The official ONNX definition alone is incomplete, and its limitations will be highlighted when compared to our SONNX informal specification.

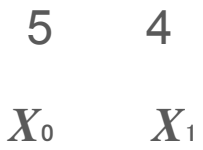
Example

- Inputs (***2 tensors***)

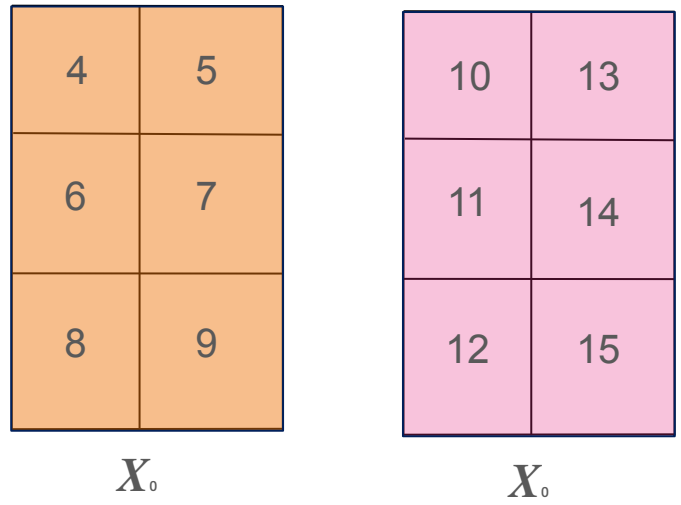
Case 1: two input tensors, vectors



Case 2: two input tensors, scalars



Case 3: two input tensors, matrices



- These illustrations show input tensors with different dimensions.

Informal specification

- ***Add*** operator is a strictly binary operator using ***Scalar*** type

Inputs constraints :

- **Size:** All inputs must have the same size. This is a **restriction** because broadcasting is not supported for the SONNX scope.
- **Type:** All inputs must have the same data type**.**

Output constraints :

- Same **dimension** as inputs.
- Same **type** as inputs

Mathematical semantics :

Let i the index covering all dimensions of the tensors.

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

add.md

SONNX ***add*** operator signature

$$C = Add(A, B)$$

ONNX ***add*** operator limitations

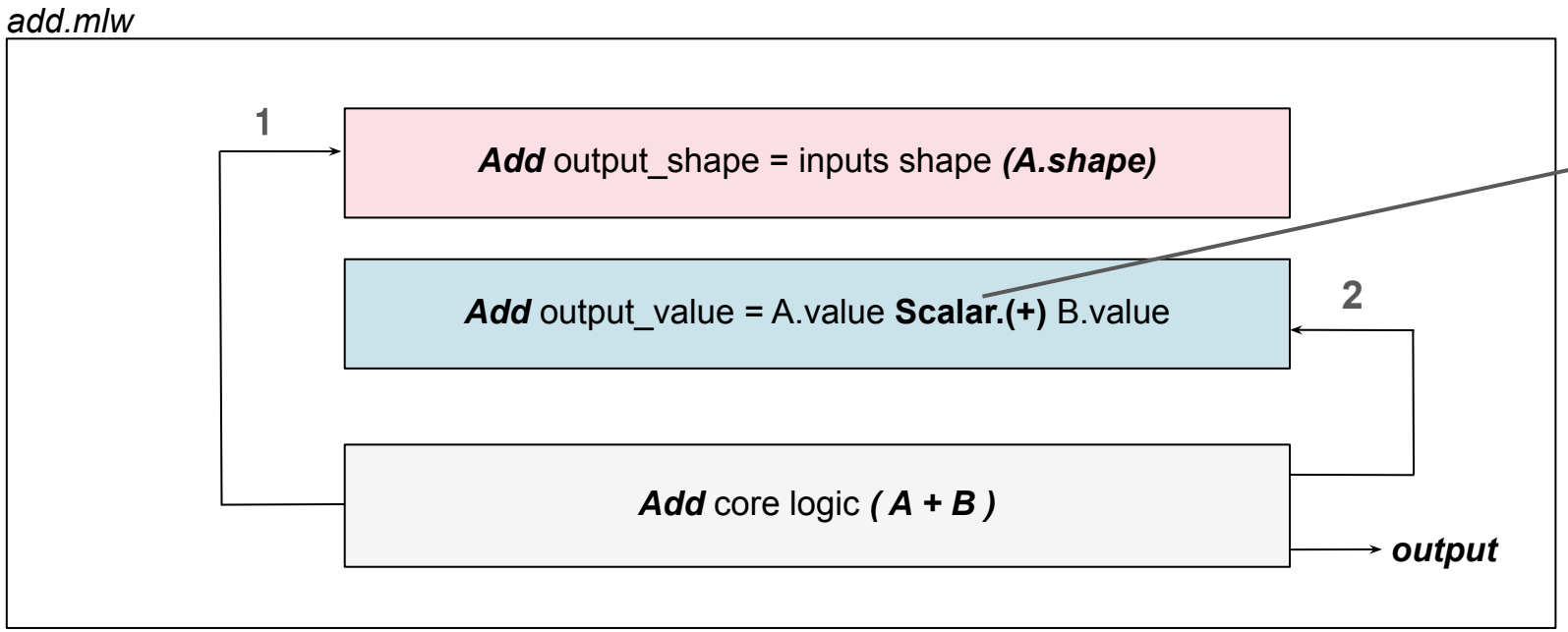
- **Tensor shape:** The specification lacks detail on the valid range for input tensor dimensions.
- **Floating-point behavior:** It does not define rounding methods or how to handle overflow (e.g., results becoming `inf`).
- **Integer Overflow:** It fails to specify that integer types should wrap around on overflow.
- **Binary Operation:** The operator is strictly binary; it cannot be used to sum multiple (more than two) inputs in a single operation.

Formal specification : add.mlw

- Document **add.mlw** was defined using **Scalar** module which leverages Why3 standard libraries **ieee_float** and **mach.int**.
  https://www.why3.org/stdlib/ieee_float.html , https://www.why3.org/stdlib/mach.int.html

Concat.mlw architecture:

- Based on ***concat*** operator and L. Correnson ONNX ***where*** operator work.



Scalar.mlw formal definition :

- The ***Scalar*** module introduces the **"+"** operator and provides a common numerical type for use with other operators.

AIRBUS

Formal specification : scalar.mlw

- The **scalar.mlw** module is being developed in collaboration with M. Turki from IRT de Saint Exupéry to serve as a common foundation for various numerical operators, such as convolution, add, and matmul.

Scalar module goals:

- The primary goal of the ***Scalar*** module is to accurately represent diverse data types and their basic operations. While it is not part of the official ONNX library, it was created to provide a flexible abstraction that can be applied to all numerical operators. The module is built upon a foundation of pre-existing standard libraries.

Add operator focus

Data types needed for ONNX:

- bfloat16,
- double,
- float (by default float32)
- float16,
- int16,
- int32,
- int64,
- Int8,
- uint16,
- uint32,
- uint64,
- uint8

→ **ieee_float** and **mach.int**

Covered by libraries
Not covered

Data types covered in Why3:

- bfloat16,
- double,
- float (by default float32)
- float16,
- int16,
- int32,
- int64,
- Int8,
- uint16,
- uint32,
- uint64,
- uint8

→

Properties and operations needed:

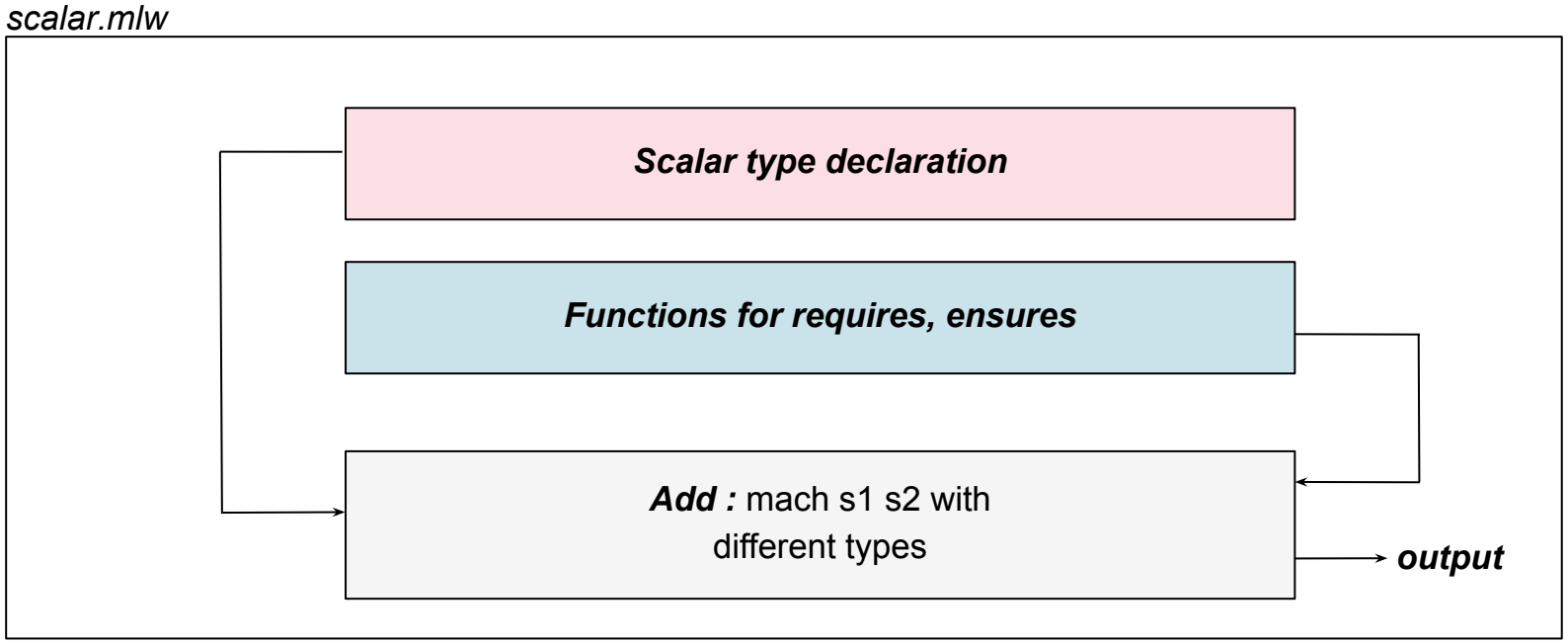
- Zero value
- Arithmetic operations (addition, subtraction, multiplication, division)
- Boolean comparison
- Algebraic properties as commutativity and associativity

Formal specification : scalar.mlw

- For the module **scalar.mlw**, we considered two approaches: one using a generic **`Scalar`** type that is instantiated for different data types, versus another, more direct approach that implements logic separately for each type.

Scalar.mlw architecture :

- This direct, “naive” implementation of the Scalar module is a stepping stone for the improved version using an abstract generic type.



Traceability mechanisms

Specifications coherence :

- Link between informal specification and the formal one.
- Ease for the user of reading both specifications and understanding their coherence.

Key points (E1… E9) :

- Major notions in the informal specification that need to be found in (or reflected in) the formal specification.
- In theory, these points guide the development of the formal specification.

Traceability through testing

- We repurpose examples from the informal specification as tests for our reference code.
- This creates a direct link from the initial requirements to the final implementation.

Alignment of informal and formal specifications :

- Our goal is to write the formal code in a direct, almost "naive" way that closely mirrors the informal specification. This involves a key principle:
- Every variable, concept, or term introduced in the informal specification must have a clear and direct counterpart in the formal code.
- Reuse of the same variable names to refer to definitions introduced in the informal specification.
- Function names also refer to these concepts.

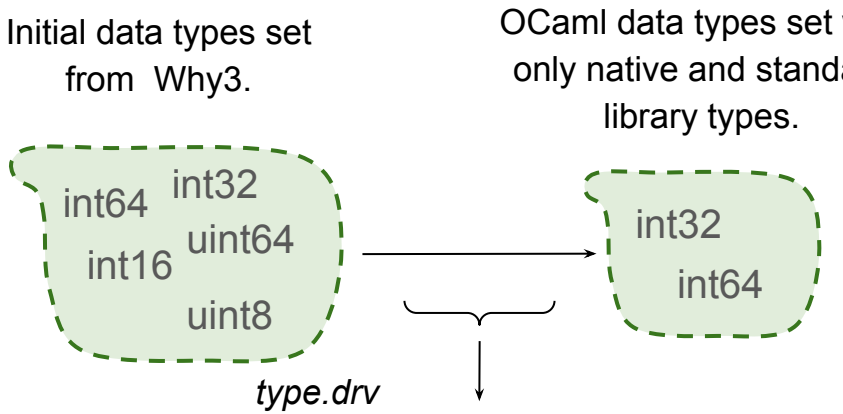
Reference code generation

Serves as a reference standard for testing :

- The generated OCaml code provides a definitive reference for testing the correctness of others ***add*** operator implementations. It preserves and guarantees the conservation if the semantics from formal specification to the reference code.
- Crucial for comprehensive testing processes and serves as a “ground truth” before achieving the proof process.

Complexity of the extraction process & configuration :

- The process relies heavily on dune, a build system that requires considerable time to master. It also need a specific driver for data types for the ***add*** specific case.




```
module mach.int.Int32
  syntax type int32 "int32"
  syntax val (+) "Int32.add %1 %2"
  syntax val (-) "Int32.sub %1 %2"
  syntax val (=) "Int32.eq %1 %2"
  syntax val to_int "Int32.to_int %1"
end
```

Specific *driver* for add operator

add.mlw

```
let add_tensors (a: tensor scalar) (b: tensor scalar) :
tensor scalar
    =
 let result_shape = a.shape in {
       shape = result_shape;
       value = fun (i: index) ->  Scalar.add val_a val_b}
```

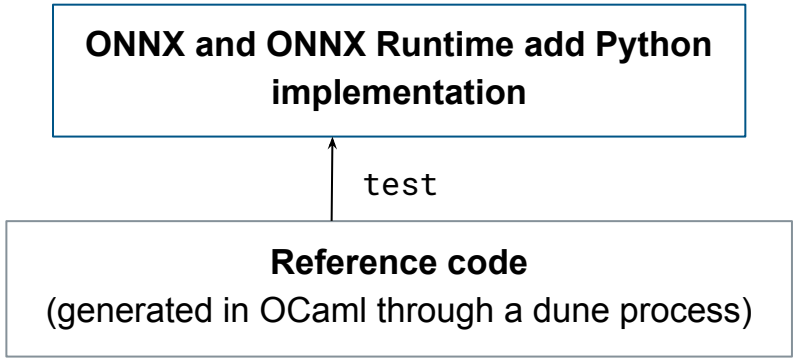
Opadd__add.ml

```
let add (a: Scalar__Scalar.scalar Tensor__Tensor.tensor) (b:
Scalar__Scalar.scalar Tensor__Tensor.tensor) :
  Scalar__Scalar.scalar Tensor__Tensor.tensor =
  let result_shape = a.Tensor__Tensor.shape in
  { Tensor__Tensor.shape = result_shape; Tensor__Tensor.value =
    (fun (i: (int) list) -> Scalar__Scalar.add val_a val_b)) }
```

Validation tests

Test to check the correctness of our definition of *add* :

- To validate our Why3 formal specification of the ***add*** operator *prior* to undertaking the full proof process, we first evaluated its OCaml implementation.
- Our strategy involved comparing this implementation behavior against the standard ONNX and ONNX Runtime Python versions using a diverse set of test cases to ensure consistency.
- We also tested the overflow behavior of **float32** and **int32** to determine if custom library handlers were necessary.



- We adapted its *make test* command and dune files, implementing specific *test.expected* file to verify the output for floating-point and integer values.
- However, these test cases are insufficient to evaluate the correctness of the add implementation and need to be complemented with the use of the automated test framework.

Overflow tests example

Int32 overflow test :

```
x = 2^31 - 1 // 2,147,483,647
y = 1

result = x Int32.+ y
```

	ONNX Runtime	OCaml
Result	-2147483648	-2147483648

- Results are identical for both ONNX Runtime and OCaml implementation after overflow test for **Int32,** performing wrap around process.

Float32 overflow test :

```
x = 3.4^38
y = 1.0^38

result = x Float32.+ y
```

	ONNX Runtime	OCaml
Result	inf	44000000000000001276 2122340980846755840.0 000000

The results in OCaml differ from our expectations, as they do not show an overflow. **How can we explain this discrepancy?**

type.drv

```
module ieee_float.Float32
  syntax type t "float"
  syntax val (.+) "%1 +. %2"
  syntax val (.-) "%1 -. %2"
  syntax val (.*) "%1 *. %2"
  syntax val (./) "%1 /. %2"
  syntax val eq   "%1 = %2"
end
```

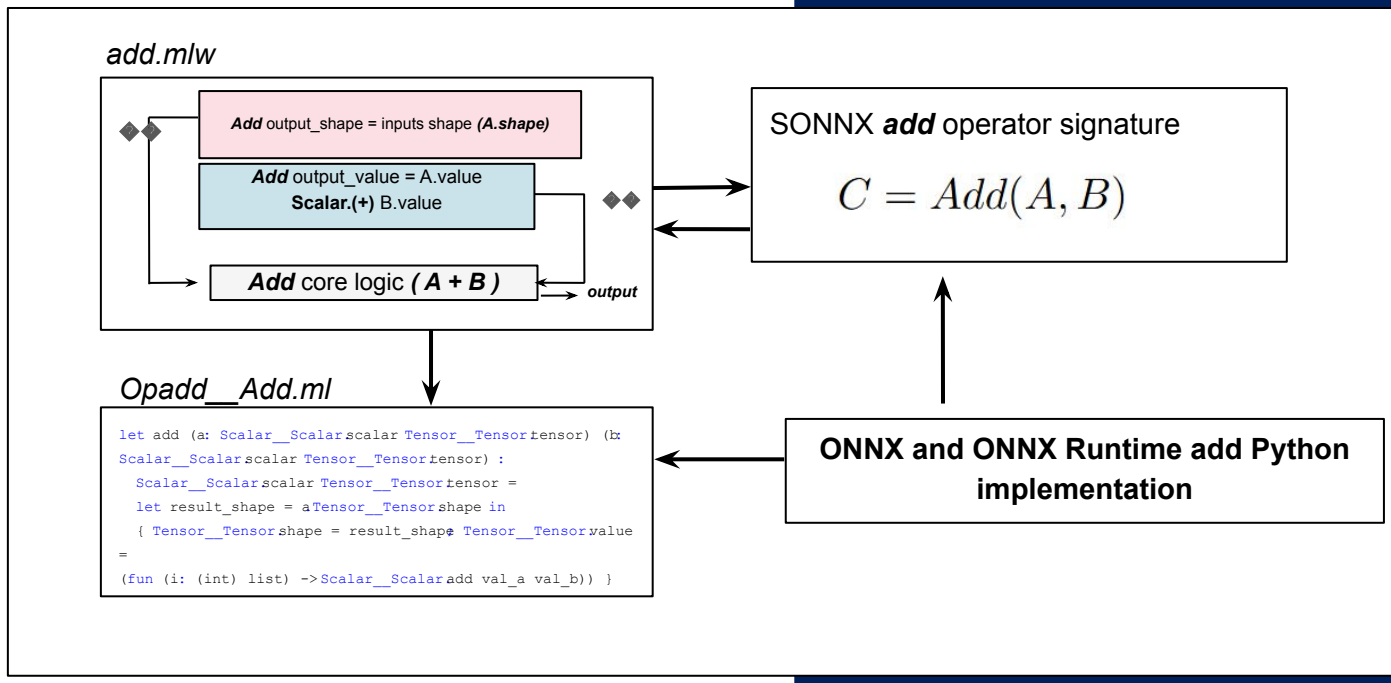
OCaml lacks a native Float32. WhyML implementation currently maps Float32 to OCaml's standard float type, which is a 64-bit float (Float64).

Using an external library could be an answer ? ocaml-float32 (community library)

Conclusion

Key takeaways :

- The ***add*** operator's complexity stems from handling numerical data types and their machine representation (rounding and overflow issues).
- Refining the informal specification for conciseness without sacrificing accuracy demonstrated a clear trade-off between verbosity and precision.
- The formal specification, though potentially seeming complex at first glance, faithfully translates the key points from its informal counterpart.



Process overview diagram

- Rather than being independent, the formal and informal specifications are mutually reinforcing and complementary.
- Consistent syntax across formal and informal specifications, which can be applied to other operators, helps ensure and demonstrate overall specification coherence.
- The generated OCaml code provides a reliable standard for testing but is effort-intensive due to the custom driver *type.drv* required for data types cases.