ONNX format comments for safety critical profile

# Introduction

This document objective: to highlight the issues in the current ONNX specification and address new needs for safety profile.

Some comments are linked to MLMD storage format concerns which are not linked to ONNX Operator semantics: is it in the perimeter of the working group SOW ?

# References

ONNX IR specification <https://github.com/onnx/onnx/blob/main/docs/IR.md>

ONNX Operator specification <https://onnx.ai/onnx/operators/>

ONNX protobuf specification <https://github.com/onnx/onnx/blob/main/onnx/onnx.proto>

ONNX broadcasting specification <https://github.com/onnx/onnx/blob/main/docs/Broadcasting.md>

# Function vs Operator

## Opset resolution

ONNX opset for ai.onnx domain define *Functions* and *Operators*.

There is no clear rationale to choose between each possibility.

Ex:

[Relu](https://onnx.ai/onnx/operators/onnx__Relu.html) (opset < 14) is an Operator

[Relu](https://onnx.ai/onnx/operators/onnx__Relu.html) (opset >= 14) is a Function (using Max Operator which was already existing at opset < 14)

Sigmoid, Tanh are Operators although they could be implemented as Functions using Exp.

In the current meta model (onnx.proto), there is a possibility to define a Function in the ai.onnx domain, or locally in the Model itself.

Ex: the Mish or Silu activations functions are not (yet) in the ai.onnx domain, but can be locally defined and used in the Model.

The existence of Function is a design concern, which brings some inconsistencies to opset reference.

Ex: The model can reference an opset V1, and a local Function can reference an opset V2. The ONNX IR specification reports that the inconsistency shall be resolved at runtime. **This has to be resolved at design time in the safety profile.**

In the [ONNX protobuf](#m49488docl2):

 // 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 priotity or this is treated as error) is defined by

// the runtimes.



For the semantic specification, do we authorize the formal expression to use predefined operators ? In this case, there shall be a clear distinction between the used names.

Ex: Exp for Exp operator, and e for Euler constant.

For each new opset release, we need also to carefully review all operators specification in front of the referred operators (which might have been modified).

My recommendation : add a statement in the profile that all opset import directive shall be consistent in the model.

Ex: if there is a local Function importing an opset Vx, then the model shall import opset Vx, and conversely.

## Function polymorphism

The [IR document](#3n8jmqgk5v2) authorize Functions to be polymorph (as Operators are):

*The value\_info field (added in IR version 10) allows a model to store type and shape information about the values used in a function, including its inputs and outputs. Note that this is optional, and ONNX allows functions to be polymorphic.*

Do we authorize Functions to be polymorphic in the safety profile ? (My recommendation: yes)

Rationale: if specifying Mish or Silu activation as a local function. We do not want to restrict it to a particular tensor shape.

## Function overloading

[IR document](#3n8jmqgk5v2) v10 add the possibility to embed several implementations of the same function:

*IR version 10 adds the field overload, and the triple (name, domain, overload) acts as a unique-id across functions stored in a model. This is intended to support cases where distinct function-bodies are required for distinct calls to the function within the model. An opset version is not explicitly identified in a FunctionProto, but it is implicitly determined by the opset version of the domain included in the model.*

Do we support this feature in a safety profile ?

# Attributes

[ONNX IR](#3n8jmqgk5v2) defines *Attributes*, which are supposed to be constant parameter data.

Actually *Attributes* are able to contain an array of *Graph*, which brings complexity.

This is typically used for ControlFlow Operators

Ex: <https://onnx.ai/onnx/operators/onnx__If.html>

The Attribute object can be specified by name or by reference (ref [ONNX protobuf](#m49488docl2))

 // if ref\_attr\_name is not empty, ref\_attr\_name is the attribute name in the parent function.

// In this case, this AttributeProto does not contain data, and it's a reference of attribute

// in parent scope.

// NOTE: This should ONLY be used in function (sub-graph). It's invalid to be used in the main graph.

optional string **ref\_attr\_name** = 21;

As Attributes, Tensors might be declared at several levels (Model, Functions), the naming of each object shall be unique.

The existence of **ref\_attr\_name** suggests that all Function attributes shall be defined in the Function, and in case the Attribute object fields are defined in the parent Model, the **ref\_attr\_name** field shall be used instead of the **name** field.

## Attributes vs Input trend

In the [IR document](#3n8jmqgk5v2):

*There are two distinct ways to pass information to operators – inputs and attributes. Inputs represent graph inputs or values computed elsewhere in the graph, while attributes are used for values that are constants in the graph. This distinction may be highly relevant to achieving good performance for some implementations, while completely irrelevant to others.*

*…*

*Attribute values are runtime constants, in that their values are determined when a model graph is constructed and therefore not computed at runtime.*

In Operator specifications, Attributes tend to be replaced by Input to increase dynamicity.

Ex: <https://onnx.ai/onnx/operators/onnx__Dropout.html>, the ratio attribute in opset 10 was moved to input in recent opsets.

Do we follow this trend in a safety profile, or the opposite (more static), which would break compatibility with the legacy profile ?

# Documentation & traceability

The meta model proposes several features to document the model.

In several class, the following fields can be used for this purpose

 // A human-readable documentation for this value. Markdown is allowed.

optional string **doc\_string** = 3;

// Named metadata values; keys should be distinct.

repeated StringStringEntryProto **metadata\_props** = 4;



These fields are used by onnx exporter to log the model origins. They are also used by optimizers to document what was performed (merge layer x with layer y)

Type and shape of intermediate tensors are typically not explicitly defined.

They can be inferred by some algorithm but are not part of the specification.

ValueInfo is an attribute which is used to define an explicit type for a Tensor, and add documentation. The ValueInfo can be specified at several levels and might bring inconsistencies.

My recommendation 1: some rule(s) shall be added to specify/restrict the scope of ValueInfo field.

My recommendation 2: to force the documentation of all intermediate Tensor using ValueInfo, at least in the Model (and eventually in Functions if not supporting polymorphism)

There is no particular field for traceability.

Using **doc\_string** field for this purpose is subject to conflict with exporter / optimizer tools.

Using **metadata\_props** with a key pattern like “trace\_”… can provide trace data.

Another possibility is to request for the addition of a new property in the meta model.

## MLMID identification

The ARP 6983 draft 5 recommended practice suggests that the MLMD is decomposed into MLMID, and in §7.1.4 Reviews and Analyses of ML MID:

Traceability – The objective is to ensure that the MLMD was developed into the ML MIDs.

The **metadata\_props** can be used to store MLMID identification, or

# Integrity

There is no support for integrity of model structure nor model parameter.

Nevertheless, there might be 2 kinds of indirect support:

* **external\_data** file
* **metadata\_props** field

## Exernal data

 // Data can be stored inside the protobuf file using type-specific fields or raw\_data.

// Alternatively, raw bytes data can be stored in an external file, using the external\_data field.

// external\_data stores key-value pairs describing data location. Recognized keys are:

// - "location" (required) - POSIX filesystem path relative to the directory where the ONNX

// protobuf model was stored

// - "offset" (optional) - position of byte at which stored data begins. Integer stored as string.

// Offset values SHOULD be multiples 4096 (page size) to enable mmap support.

// - "length" (optional) - number of bytes containing data. Integer stored as string.

// - "checksum" (optional) - SHA1 digest of file specified in under 'location' key.

repeated StringStringEntryProto **external\_data** = 13;

In this case, the checksum field contains the SHA1 of the data file. Using external data without checksum does not bring identification of the data file (path location is not sufficient to strictly identify the data file), and the offset and length field might lead to invalid data interpretation.

The checksum atomicity is per file.

My recommendation: we have to remove ‘optional’ to make it mandatory.

## metadata property

The **metadata\_props** field is able to store the checksum. It can also specify another algorithm than SHA1. It authorizes the use of a single onnx file. The checksum atomicity shall be per tensor (else we have to specify the processing order among several tensors).

This method seems to offer more flexibility, and enables MLMID local checksums that might be verified on each specific device (end to end integrity).

# DataStorage

Constant model parameters are stored using protobuf format. There is 2 possibilities to store data:

* typed array
* raw data

In the [ONNX protobuf specification](#m49488docl2),

// Depending on the data\_type field, exactly one of the fields below with

// name ending in \_data is used to store the elements of the tensor.

// For float and complex64 values

// Complex64 tensors are encoded as a single array of floats,

// with the real components appearing in odd numbered positions,

// and the corresponding imaginary component appearing in the

// subsequent even numbered position. (e.g., [1.0 + 2.0i, 3.0 + 4.0i]

// is encoded as [1.0, 2.0 ,3.0 ,4.0]

// When this field is present, the data\_type field MUST be FLOAT or COMPLEX64.

repeated float **float\_data** = 4 [packed = true];

// For int32, uint8, int8, uint16, int16, uint4, int4, bool, float8 and float16 values

// float16 and float8 values must be bit-wise converted to an uint16\_t prior

// to writing to the buffer.

// uint4 and int4 values must be packed to 4bitx2 prior to writing to the buffer, the first element is stored in

// the 4 LSB and the second element is stored in the 4 MSB.

// When this field is present, the data\_type field MUST be

// INT32, INT16, INT8, INT4, UINT16, UINT8, UINT4, BOOL, FLOAT16, BFLOAT16, FLOAT8E4M3FN, FLOAT8E4M3FNUZ, FLOAT8E5M2, FLOAT8E5M2FNUZ

repeated int32 **int32\_data** = 5 [packed = true];

// For strings.

// Each element of string\_data is a UTF-8 encoded Unicode

// string. No trailing null, no leading BOM. The protobuf "string"

// scalar type is not used to match ML community conventions.

// When this field is present, the data\_type field MUST be STRING

repeated bytes **string\_data** = 6;

// For int64.

// When this field is present, the data\_type field MUST be INT64

repeated int64 **int64\_data** = 7 [packed = true];

// Serializations can either use one of the fields above, or use this

// raw bytes field. The only exception is the string case, where one is

// required to store the content in the repeated bytes string\_data field.

//

// When this raw\_data field is used to store tensor value, elements MUST

// be stored in as fixed-width, little-endian order.

// Floating-point data types MUST be stored in IEEE 754 format.

// Complex64 elements must be written as two consecutive FLOAT values, real component first.

// Complex128 elements must be written as two consecutive DOUBLE values, real component first.

// Boolean type MUST be written one byte per tensor element (00000001 for true, 00000000 for false).

// uint4 and int4 values must be packed to 4bitx2, the first element is stored in the 4 LSB and the second element is stored in the 4 MSB.

//

// Note: the advantage of specific field rather than the raw\_data field is

// that in some cases (e.g. int data), protobuf does a better packing via

// variable length storage, and may lead to smaller binary footprint.

// When this field is present, the data\_type field MUST NOT be STRING or UNDEFINED

optional bytes **raw\_data** = 9;

The protobuf data storage for integer (**int32\_data)** uses variable length encoding (VARINT) for integer, which leads to a compressed model data file, and complex encoding/decoding algorithm.

This has no effect on float data which is encoded as constant length I32 protobuf data type. It is equivalent to use **float\_data** or **raw\_data** in this case.

The protobuf I32 data type is little-endian. There is no support for big endian data storage.

Do we need to add big endian support (powerpc) ?

Do we authorize VARINT data storage, or force the use of raw\_data, and specify our own integer encoding scheme (constant size, at the cost of a bigger model file) ?

# Operator Polymorphism, variadic input

## data type

When performing an addition between 2 tensors, we suppose that both tensors are of the same data type and the same shape. In this case the specification is straightforward, but this leads to specifying the semantic for each data type, and therefore to particularize Operators per data type (Add\_float, Add\_int8, Add\_int16…).

Do we support polymorphic operators ? How to formally specify it ?Else we break legacy compatibility.

## tensor shape

The problem gets even more complex and intractable if operator inputs are of different shapes.

In recent opsets, element wise operators (Add, Mul…) provide shape [broadcasting](#472s5cmv4wa3) capability.

It is a logic which implicitly modifies the shape of one input (extends its dimensions), in order to be identical to the other input shape, before actually performing the computation.

Shall we support broadcasting (legacy ONNX compatibility) ? How to formally specify it ?

Else ???

## variadic input

Some operators enables variadic input ex: <https://onnx.ai/onnx/operators/onnx__Max.html>

Do we support them as is ? Do we bound the number of input ?

# Protobuf version and IR support

ONNX is supported with protobuf v2 and v3. Do we also support both ?

ONNX IR version evolves. Do we support safety profile for old IR versions ? How will we manage to update the safety profile with newer ONNX IRs ?

# Safety profile opset definition

The [IR document](#3n8jmqgk5v2) identifies the existence of a meta model to define opsets

*Operator Sets*

*Each model MUST explicitly name the operator sets that it relies on for its functionality. Operator sets define the available operators and their version. Each model defines the imported operator sets by their domains. All models implicitly import the default ONNX operator set.*

*Each operator set SHALL be defined in a separate document, also using protobuf as the serialization format. How operator set documents are found at runtime is implementation-dependent.*

*Note: As of the publication of this document, no ONNX implementation is known to process operator set documents.*

This could be a convenient formal way to specify the safety profile opset (if different from the legacy one). The resulting protobuf file could be used as a configuration file by code generator tools, or embedded runtime.