

Open Neural Network Exchange (ONNX) — Part 1: Core

Preparatory Standard

Warning for drafts

This document is a draft and is not a standard. It is distributed for review and comment, is subject to change without notice, and may not be referred to as a standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Table of contents

Foreword.....	v
Introduction.....	vi
1. Scope.....	1
2. Normative references.....	1
3. Terms and definitions.....	2
4. Conformance.....	4
4.1. Requirement terminology.....	4
4.2. Conformance classes.....	4
4.3. Conformance profiles.....	4
4.4. Operator set coverage.....	4
4.5. Conformance statement.....	4
4.6. Relationship to Parts 2 and 3.....	5
5. Information model.....	5
5.1. General.....	5
5.2. Entities and their relationships.....	5
5.3. Model.....	5
5.4. Names and scoping.....	6
5.5. Metadata.....	7
5.6. Domains as the extension point.....	7
6. Type system.....	7
6.1. General.....	7
6.2. Element types.....	7
6.3. Reduced-precision element types.....	8
6.4. String elements and character encoding.....	10
6.5. Shapes and dimensions.....	10
6.6. Broadcasting.....	11
6.7. Denotation.....	11
6.8. Composite types.....	12
6.9. Type equality and assignability.....	13
7. Graph structure.....	13
7.1. Graph, inputs and outputs.....	13
7.2. Initializers.....	13
7.3. Nodes.....	14
7.4. Attributes.....	14
7.5. Topological constraints.....	14
7.6. Subgraphs and scoping.....	15
8. Graph semantics.....	15
8.1. Evaluation model.....	15
8.2. Evaluation order and concurrency.....	15
8.3. Control flow.....	15
8.4. Functions.....	16
8.5. Determinism.....	16
8.6. Error semantics.....	16
9. Shape and type inference.....	16
9.1. General.....	16
9.2. Obligations on operator definitions.....	16
9.3. Symbolic dimensions.....	17
9.4. Strictness and failure.....	17
9.5. Relationship to validation.....	17
10. Model validation.....	17
10.1.General.....	17
10.2.Structural checks.....	18
10.3.Type and shape checks.....	18
10.4.Operator set checks.....	18
10.5.Severity and reporting.....	18

11. Numerical behaviour.....	18
11.1.General.....	18
11.2.Floating-point arithmetic.....	18
11.3.Special values.....	18
11.4.Accumulation and reassociation.....	19
11.5.Tolerances.....	19
11.6.Non-deterministic operators.....	19
12. Serialization.....	19
12.1.General.....	19
12.2.Primitive encodings.....	19
12.3.Fields.....	21
12.4.Unrecognized fields.....	22
12.5.Limits.....	22
12.6.Message definitions.....	22
12.7.Relationship to Protocol Buffers.....	22
12.8.Canonical form.....	22
13. External data.....	23
13.1.General.....	23
13.2.Reference resolution.....	23
13.3.Integrity.....	23
14. Versioning and compatibility.....	24
14.1.Version axes.....	24
14.2.IR version evolution.....	24
14.3.Operator set evolution.....	25
14.4.Compatibility obligations.....	25
14.5.Deprecation and removal.....	26
15. Security and resource limits.....	26
15.1.General.....	26
15.2.Threat model.....	26
15.3.External data and path resolution.....	26
15.4.Resource bounds.....	27
15.5.Handling untrusted models.....	27
16. Maintenance and extension.....	27
16.1.Custom domains.....	27
16.2.Experimental operators.....	27
16.3.Adding an operator.....	28
16.4.Amending this document.....	28
Annex A Conformance statement.....	29
Annex B Message definitions.....	30
B.1. General.....	30
B.2. Model, graph and tensor messages.....	30
B.3. Operator set messages.....	37
B.4. Map and sequence messages.....	37
B.5. Derivation.....	38
Annex C Known gaps and divergences.....	39
C.1. Open gaps.....	39
C.2. Gaps closed from the upstream documentation.....	40
Annex D Correspondence with the ONNX reference implementation.....	42
Bibliography.....	43

Foreword

This document has been prepared by the ONNX Standardization Working Group.

ONNX 1 consists of the following parts:

- Part 1: Core (this document)
- Part 2: Operator sets
- Part 3: Conformance test package

The division is deliberate. The core changes with the IR version, roughly once a year; the operator sets change with every ONNX release. Keeping them in one document would give the core a new edition each time an operator is added.

Status of this document —

This is an unofficial working draft. It has no standing within the ONNX project, the Linux Foundation, IEC, ISO or any other standards development organization, and it MUST NOT be cited as a normative reference.

It exists to demonstrate what a normative, Metanorma-authored ONNX specification would look like, as input to a discussion with the ONNX Steering Committee.

Introduction

The Open Neural Network Exchange (ONNX) format is in wide industrial use as an interchange representation for machine learning models. Its present specification is maintained as a set of Markdown documents alongside the reference implementation, which is well suited to rapid evolution but does not provide the properties expected of a standard: a stable identifier, an unambiguous separation of normative from informative text, testable conformance requirements, and a controlled change process.

This document restates the ONNX format as a normative specification. It is written in Metanorma so that it can be rendered in the house styles of successive publishers as ONNX moves along the intended path:

- 1) a specification published under a Linux Foundation Joint Development Foundation (JDF) project, giving ONNX a specification process with an explicit IPR regime;
- 2) subsequently, submission of that specification through a Publicly Available Specification (PAS) or equivalent route towards an ISO or ISO/IEC deliverable.

Accordingly this document follows ISO/IEC Directives, Part 2 drafting conventions from the outset, even though its current publisher is not an SDO.

This part specifies the framework: what a model is, what its values may be, what evaluating it means, and what an implementation must do to conform. It names no individual operator. The operator definitions themselves are in Part 2, and the test vectors that make conformance checkable are in Part 3.

The technical content is intended to describe ONNX as it is, not as it might be. Where the reference implementation and the present prose specification disagree, that divergence is recorded rather than silently resolved; see Annex C.

Open Neural Network Exchange (ONNX) — Part 1: Core

1. Scope

This document specifies the core of the Open Neural Network Exchange (ONNX) format: the information model, the type system, the evaluation semantics, the rules for shape and type inference and for model validation, the binary encoding, the versioning scheme, and the conformance requirements.

This document specifies:

- the entities of an ONNX model and their relationships;
- the data types and shapes that values in a model may take;
- the structure a graph must have to be well formed;
- the evaluation semantics of a graph;
- the obligations on shape and type inference;
- what a validating implementation must detect;
- the numerical behaviour an implementation must exhibit;
- the binary serialization of a model and the handling of external data;
- the versioning and compatibility rules governing the version axes;
- the security considerations and resource limits that apply to untrusted models;
- the conformance classes and profiles that an implementation may claim.

This document does not specify:

- the definitions of individual operators, which are in [ONNX 1-2](#);
- the conformance test vectors, which are in [ONNX 1-3](#);
- the internal representation, scheduling or optimization strategy of a runtime;
- training procedures, optimizers or loss functions, except where they are expressed as operators within a graph;
- hardware requirements, performance characteristics or resource limits beyond those stated in [Clause 15](#);
- the ONNX reference implementation, its API or its source distribution; see Annex D for the correspondence.

2. Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ONNX 1-2, *Open Neural Network Exchange (ONNX) — Part 2: Operator sets*

ONNX 1-3, *Open Neural Network Exchange (ONNX) — Part 3: Conformance test package*

IEEE 754-2019, *IEEE Standard for Floating-Point Arithmetic*

ISO/IEC 9899, International Organization for Standardization (committee). *Information technology — Programming languages — C*. Fifth edition. Geneva: International Organization for Standardization and International Electrotechnical Commission. <https://www.iso.org/standard/82075.html>.

ISO/IEC 10646, International Organization for Standardization (committee). *Information technology — Universal coded character set (UCS)*. Sixth edition. Geneva: International Organization for Standardization and International Electrotechnical Commission. <https://www.iso.org/standard/76835.html>.

IETF RFC 8259, Internet Engineering Task Force (committee). *The JavaScript Object Notation (JSON) Data Interchange Format*. 2017. RFC Publisher. <https://www.rfc-editor.org/info/rfc8259>.

IETF RFC 2119, BRADNER, S. *Key words for use in RFCs to Indicate Requirement Levels*. 1997. RFC Publisher. <https://www.rfc-editor.org/info/rfc2119>.

NOTE Protocol Buffers is deliberately not a normative reference. It has no citable specification with a stable identifier, and this document therefore states the binary encoding itself, in [Clause 12](#), rather than citing one. An implementation may still be built on a Protocol Buffers library; see [Clause 12.7](#).

3. Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1. model

self-contained description of a computation, comprising a top-level [graph](#), the [operator set](#) versions it depends upon, and producer and metadata information

3.2. graph

directed acyclic graph of [nodes](#) together with its declared inputs, outputs, initializers and value type information

3.3. node

single invocation of an [operator](#), binding named values to the operator's formal inputs and outputs and fixing its [attributes](#)

3.4. initializer

named tensor value carried in a [graph](#), being either a constant of that graph or the default value of an input of it

3.5. subgraph

[graph](#) that is the value of an [attribute](#) of a [node](#)

3.6. function

[operator](#) together with an implementation of it in terms of other operators

Note 1 to entry: The implementation is a fallback, not a definition of the operator's numerical behaviour; see [Clause 8.4](#).

3.7. dimension variable

name denoting an extent that is not fixed when a [model](#) is written, and that is the same wherever the name occurs in that model

3.8. operator

named computation with a specified signature and semantics, identified by the pair of its [domain](#) and its name

Note 1 to entry: Operators are defined in [ONNX 1-2](#). This document specifies the form such a definition takes and the rules it obeys, not the operators themselves.

3.9.**domain**

namespace that disambiguates operator names

Note 1 to entry: The empty string denotes the default ONNX domain.

3.10.**operator set**

opset

set of operator definitions belonging to one [domain](#), identified by that domain and a monotonically increasing integer version

3.11.**attribute**

named value that is fixed at the time the [model](#) is constructed and that does not participate in dataflow

3.12.**tensor**

dense multidimensional array of elements of a single [element type](#), having a [shape](#)

3.13.**element type**

member of the enumeration of scalar types that the elements of a [tensor](#) may take

3.14.**shape**

ordered sequence of dimensions of a [tensor](#), each of which is a non-negative integer, a symbolic name, or unknown

3.15.**conforming implementation**

implementation that satisfies all requirements of one or more of the conformance classes defined in [Clause 4](#)

3.16.**producer**

program that emits a [model](#)

3.17.**consumer**

program that reads a [model](#) in order to evaluate, transform or inspect it

3.18.**validation**

determination of whether a [model](#) satisfies the requirements of this document, without evaluating it

3.19.**conformance profile**

named set of requirements, narrower than a conformance class, that an implementation may claim

3.20.

test vector

model, together with input values and expected output values, published in [ONNX 1-3](#) for the purpose of checking conformance

4. Conformance

4.1. Requirement terminology

In this document, the key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY” and “OPTIONAL” are to be interpreted as described in [IETF RFC 2119](#).

Editorial note —

ISO/IEC Directives Part 2 uses “shall/should/may/can” and disallows RFC 2119 capitalisation. Retain RFC 2119 for the JDF stage; convert to Directives Part 2 verbal forms before any ISO submission. Tracked as an editorial migration task, not a technical one.

4.2. Conformance classes

This document defines the conformance classes listed in Table 1. An implementation claiming conformance to this document SHALL state which classes it claims and, for each class it claims, the [operator set](#) versions it supports.

Table 1 — Conformance classes

Class	Applies to	Summary of obligations
Producer	A program that writes ONNX models	Emits only models that satisfy Clause 12 , Clause 5 and Clause 7 .
Validating consumer	A program that validates ONNX models	Detects and reports every condition classified as an error in Clause 10 .
Evaluating consumer	A runtime that executes ONNX models	Computes results within the tolerances of Clause 11 for every operator in the operator sets it claims.

4.3. Conformance profiles

A conformance profile names a subset of the requirements of this document and of [ONNX 1-2](#) that an implementation may claim in place of full support.

Editorial note —

The profiles themselves are not yet defined. Candidates raised so far are a core inference profile excluding training and control flow, a profile restricted to statically shaped models, and a profile for the `ai.onnx.ml` domain. Profiles must be defined together with the test selection rules in [ONNX 1-3](#), since a profile that no test package can check is not a conformance claim. To be settled with the Steering Committee.

4.4. Operator set coverage

A [conforming implementation](#) is not required to support every operator defined in an operator set. It SHALL, however, report unambiguously which operators it does not support, and it SHALL NOT silently substitute an approximation of an unsupported operator.

4.5. Conformance statement

An implementation claiming conformance SHALL publish a conformance statement containing the information required by Annex A.

4.6. Relationship to Parts 2 and 3

This document specifies the form an operator definition takes and the rules an [operator set](#) obeys; it defines no operator. Operator definitions are in [ONNX 1-2](#).

A claim of conformance to this document alone is a claim about structure, encoding and validation. A claim about evaluation is necessarily also a claim against one or more operator sets of [ONNX 1-2](#), checked with the test vectors of [ONNX 1-3](#).

5. Information model

5.1. General

This clause introduces the entities of an ONNX [model](#) and their relationships. [Clause 7](#) states the constraints a well-formed graph satisfies; [Clause 8](#) states what evaluating one means.

5.2. Entities and their relationships

A model contains exactly one top-level [graph](#).

A graph contains an ordered sequence of [nodes](#), together with its declared inputs, outputs, initializers and value type information.

A node invokes one [operator](#), identified by [domain](#) and name, binds named values to that operator's formal inputs and outputs, and fixes its [attributes](#).

A value is produced by exactly one of: a graph input, an initializer, a node in the same graph, or an enclosing scope.

5.3. Model

5.3.1. Contents

Table 2 states the contents of a model and the obligation on each.

Table 2 — Contents of a model

Item	Obligation	Description
IR version	mandatory	The version of this document against which the model was produced; see Clause 14 .
Operator set imports	mandatory	The operator sets the model requires; see Clause 5.3.2 .
Graph	mandatory	The top-level graph, evaluated to execute the model.
Functions	optional	Functions local to the model; see Clause 8.4 .
Producer name and version	optional	The tool that produced the model, for diagnostic purposes only.
Model domain	optional	A namespace for the model itself, distinct from an operator domain.
Model version	optional	An identifier assigned by the producer , opaque to this document.
Documentation string	optional	Human-readable documentation.
Metadata properties	optional	Named metadata values; see Clause 5.5 .

A model domain SHOULD be a reverse domain name based on the identity of the organization responsible for the model.

5.3.2. Operator set imports

An [operator set](#) is identified by a pair of a [domain](#) and a version. The version is a positive integer that increases monotonically as versions of that operator set are published.

A model SHALL declare, for every [domain](#) used by any node reachable from its top-level graph, the operator set version of that domain that the model requires. The default domain is denoted by the empty domain name and is imported implicitly.

A domain other than the default domain SHALL be named, and SHOULD be named by a reverse domain name based on the identity of the organization responsible for it.

Every operator invoked by a node SHALL be declared by one of the operator sets the model imports. A [consumer](#) that does not implement every operator of every imported operator set SHALL reject the model; see [Clause 4](#).

5.4. Names and scoping

5.4.1. Namespaces

Names are organized into the namespaces of Table 3. Within a namespace, a name SHALL be unique within a given graph.

Table 3 — Namespaces

Namespace	Names in it
Attribute	the attributes of one operator
Value	node inputs and outputs, graph inputs and outputs, initializers
Node	the nodes of a graph
Graph	the graphs of a model domain
Operator	the operators of a domain
Shape	the dimension variables of a model; see Clause 6.5.2

A name SHALL NOT be empty, except where an empty name is used to leave an operator input or output unsupplied; see [Clause 7.3.3](#).

A name SHOULD adhere to the identifier syntax of [ISO/IEC 9899](#).

5.4.2. Definitions and uses

An occurrence of a name of the Value namespace as a graph input, as an initializer or as a node output is a definition. An occurrence as a node input or as a graph output is a use.

A name used in a graph SHALL have exactly one definition, save that the same name MAY be defined both as a graph input and as an initializer; see [Clause 7.2](#).

5.4.3. Visibility in subgraphs

A subgraph is a graph appearing as the value of an attribute of a node.

A node input in a subgraph MAY use a name defined in an enclosing graph, whether as a node output, a graph input or an initializer of that graph. Names are therefore visible inwards.

Shadowing is not permitted: a node output, a graph input or an initializer of a subgraph SHALL have a name distinct from every name of the Value namespace that is visible in that subgraph from an enclosing graph.

NOTE Taken together, the two rules above make the definition that a use in a subgraph refers to unambiguous without stating a search order: at most one definition of any name is visible at any point.

5.5. Metadata

Metadata SHALL NOT affect the result of evaluating a model. A consumer MAY ignore any metadata it does not recognize.

A model MAY carry named metadata properties whose keys are distinct. Table 4 lists the keys this document defines; a key not listed is not reserved, and a producer MAY use one of its own.

Table 4 — Metadata property keys

Key	Value
model_author	The names of the authors of the model, or of their organizations, separated by commas.
model_license	The well-known name of the licence under which the model is made available, or a URL at which it may be read.
Image.BitmapPixelFormat	The channel order and bit depth of the pixel data of a value denoted IMAGE: one of Gray8, Rgb8, Bgr8, Rgba8 or Bgra8.
Image.ColorSpaceGamma	The colour space of such a value: Linear or SRGB.
Image.NominalPixelRange	The range in which the pixel values of such a value are stored: one of NominalRange_0_255, Normalized_0_1, Normalized_1_1 or NominalRange_16_235.

The Image . keys apply to every value of the model whose type carries the IMAGE denotation; see [Clause 6.7](#). Their keys and values are compared without regard to case.

5.6. Domains as the extension point

The [domain](#) is the sole extension point of the information model. An operator outside the domains defined by [ONNX 1-2](#) is a custom operator; see [Clause 16](#). A custom domain MAY also introduce value types of its own; see [Clause 6.8.4](#).

6. Type system

6.1. General

Every value in an ONNX [graph](#) has a type. Types are either tensor types, sparse tensor types, sequence types, map types, optional types or opaque types.

Primitive numeric, string and Boolean types SHALL be used only as the element types of tensors; they are not themselves value types.

6.2. Element types

Table 5 is the complete enumeration of element types. The tag of each type is part of the wire format and is therefore normative: a consumer SHALL interpret the tag as the type named in the same row, and SHALL NOT assign a meaning to a tag not listed.

The tag 0 (UNDEFINED) SHALL NOT appear as the element type of a value.

Table 5 — Element types and their wire tags

Tag	Name	Description
1	FLOAT	binary32 as specified in IEEE 754-2019
2	UINT8	8-bit unsigned integer
3	INT8	8-bit two's complement integer
4	UINT16	16-bit unsigned integer
5	INT16	16-bit two's complement integer
6	INT32	32-bit two's complement integer
7	INT64	64-bit two's complement integer

Tag	Name	Description
8	STRING	sequence of octets; see Clause 6.4
9	BOOL	Boolean
10	FLOAT16	binary16 as specified in IEEE 754-2019
11	DOUBLE	binary64 as specified in IEEE 754-2019
12	UINT32	32-bit unsigned integer
13	UINT64	64-bit unsigned integer
14	COMPLEX64	complex number with FLOAT real and imaginary parts
15	COMPLEX128	complex number with DOUBLE real and imaginary parts
16	BFLOAT16	1 sign bit, 8 exponent bits, 7 mantissa bits; not an IEEE 754-2019 format
17	FLOAT8E4M3FN	see Clause 6.3
18	FLOAT8E4M3FNUZ	see Clause 6.3
19	FLOAT8E5M2	see Clause 6.3
20	FLOAT8E5M2FNUZ	see Clause 6.3
21	UINT4	4-bit unsigned integer, range [0, 15]
22	INT4	4-bit two's complement integer, range [− 8, 7]
23	FLOAT4E2M1	see Clause 6.3
24	FLOAT8E8M0	see Clause 6.3
25	UINT2	2-bit unsigned integer, range [0, 3]
26	INT2	2-bit two's complement integer, range [− 2, 1]
27	FLOAT6E2M3	see Clause 6.3
28	FLOAT6E3M2	see Clause 6.3

NOTE BFLOAT16 is the binary32 format truncated to 16 bits. It has the exponent range of binary32 and the reduced precision of a 7-bit significand, and is therefore not one of the [IEEE 754-2019](#) interchange formats.

6.3. Reduced-precision element types

6.3.1. General

ONNX defines floating-point element types narrower than the [IEEE 754-2019](#) interchange formats, and integer element types narrower than eight bits. This subclause states their value encoding and their packing into octets. The packing is part of the wire format and is therefore normative.

Support for these types is not required of every [conforming implementation](#); see [Clause 4](#).

6.3.2. Narrow floating-point formats

Table 6 states, for each narrow floating-point type, the width of its sign, exponent and mantissa fields, its exponent bias, and whether it encodes infinities and NaN.

Table 6 — Narrow floating-point element types

Type	Sign	Exp.	Mant.	Bias	Infinities	NaN
FLOAT8E4M3FN	1	4	3	7	no	yes
FLOAT8E4M3FNUZ	1	4	3	8	no	yes
FLOAT8E5M2	1	5	2	15	yes	yes
FLOAT8E5M2FNUZ	1	5	2	16	no	yes
FLOAT6E2M3	1	2	3	1	no	no
FLOAT6E3M2	1	3	2	3	no	no
FLOAT4E2M1	1	2	1	1	no	no

A type whose name ends in UZ SHALL NOT encode a negative zero: the encoding that would otherwise denote it denotes NaN instead.

Float8E8M0 is an 8-bit type consisting of an exponent field only, with no sign bit and no mantissa. It is used as the shared scale of a microscaling block format and SHALL NOT be used to represent a signed quantity.

Conversion from a narrow floating-point type to a wider one SHALL be exact. Conversion to a narrow floating-point type SHALL round to nearest, ties to even.

Where the magnitude of a value exceeds the largest finite value the target type represents, the result is either that largest finite value with the sign of the operand, or NaN. Which of the two applies is a property of the operator performing the conversion and is stated in its definition in [ONNX 1-2](#); this document does not fix it.

Editorial note —

The rules above are stated as the upstream documentation states them, and two things are unsatisfactory about that. The first is the sentence immediately above: whether a conversion saturates is settled per operator rather than per type, by an attribute of Cast in the default domain, so the same pair of types converts differently in two places in one model. A type system should decide this, and the operator should not be able to vary it. The second is that the rules are not stated once for all types in [Clause 11](#), and the tolerance that [ONNX 1-3](#) needs in order to pass or fail a vector in these types does not follow from them. That remains the blocking gap recorded in Annex C.

6.3.3. Narrow integer formats

INT2, UINT2, INT4 and UINT4 denote integers of the width and range given in Table 5. Conversion from a narrow integer type to a wider one SHALL be exact. Conversion to a narrow integer type SHALL round to nearest, ties to even.

Editorial note —

The result of converting a value that lies outside the range of the target type is unstated. The upstream documentation says the rounded value is truncated, without saying whether that means clamping to the range or discarding the leading bits; the two differ in sign as well as in magnitude. State one.

6.3.4. Packing

A tensor carries its elements either as an octet string, called raw storage here, or in a field typed for its elements, called typed storage. The two pack a narrow element type differently, and a tensor uses one or the other, not both.

In raw storage, the elements SHALL be a bit stream, in order, each element occupying the next free bits of the current octet counting from its least significant bit. The storage size in octets of a tensor of N elements of a type of width w bits SHALL be the least integer not less than $N * w / 8$. Any bits of the final octet not occupied by an element SHALL be zero, and a consumer SHALL ignore them.

EXAMPLE — Storage size in raw storage

A tensor of 7 INT4 elements occupies 4 octets, of which the four most significant bits of the fourth octet are padding. A tensor of 7 Float6E2M3 elements occupies 6 octets, of which the two most significant bits of the sixth octet are padding.

In typed storage, a narrow element type uses a sequence of 32-bit integers, and Table 7 states how many elements each integer carries and which of its bits they occupy. Every other bit of such an integer SHALL be zero.

Table 7 — Narrow element types in typed storage

Element width	Elements per integer	Bits occupied
2 bits	4	bits 0 to 1, 2 to 3, 4 to 5 and 6 to 7, in element order
4 bits	2	bits 0 to 3 and 4 to 7, in element order
6 bits	1	bits 0 to 5

A producer **SHOULD** use raw storage for a 6-bit element type. Typed storage costs at least one octet per element there, so it is not a packing at all.

A floating-point element type narrower than 32 bits **SHALL** be written to typed storage as the unsigned integer whose bits are those of the element.

6.4. String elements and character encoding

Editorial note —

The reference implementation treats STRING elements as arbitrary byte strings, while much tooling assumes UTF-8. Decide whether this document requires [ISO/IEC 10646](#) UTF-8 encoding or specifies octet strings with an encoding attribute. This is a normative change either way and **MUST be raised with the Steering Committee. The upstream documentation does not settle it.**

6.5. Shapes and dimensions

6.5.1. General

A shape is an ordered sequence of dimensions. Each dimension is one of:

- a non-negative integer, denoting a fixed extent;
- a dimension variable, denoting an extent that is not statically fixed;
- unknown.

The number of dimensions in a shape is the rank of the tensor. A tensor with a shape of length zero is a scalar. A dimension of extent zero is permitted; a tensor having such a dimension contains no elements.

A tensor type whose shape is absent denotes a tensor of unknown rank. This is distinct from a tensor of rank zero, and distinct from a tensor whose rank is known and whose extents are unknown.

The inputs and outputs of a top-level [graph](#) **SHALL** have a shape, so that their rank is stated, even where the individual extents are not.

6.5.2. Dimension variables

A dimension variable is a name. Two occurrences of the same dimension variable **SHALL** denote the same extent at evaluation.

Dimension variables are scoped to the [model](#), not to a graph: an occurrence in a subgraph and an occurrence in an enclosing graph denote the same extent. This is what allows a model to relate an extent inside a subgraph to an extent outside it.

A dimension that is neither an integer nor a dimension variable is unknown, and is unrelated to any other unknown dimension.

NOTE Because dimension variables are model-scoped, a value of a sequence type whose element shape names a variable is a sequence of tensors that all have that extent. A sequence whose elements differ in an extent has to omit that dimension from its element type, and one whose elements differ in rank has to omit the shape entirely.

The name of a dimension variable SHOULD adhere to the identifier syntax of [ISO/IEC 9899](#).

6.6. Broadcasting

6.6.1. General

Two or more tensors of different shapes MAY be combined element by element where their shapes are broadcastable in one of the two senses defined below. An operator definition in [ONNX 1-2](#) states which, if either, of the two applies to it.

6.6.2. Multidirectional broadcasting

A set of tensors is multidirectionally broadcastable where, after each shape of rank lower than the greatest rank in the set has been extended on the left with dimensions of extent 1 until all shapes have that rank, every position holds, across the set, only extents that are equal to one another or equal to 1.

The result has the greatest rank in the set, and its extent at each position is the greatest extent at that position.

EXAMPLE — Multidirectional broadcasting

Shapes (2, 3, 4, 5) and () give (2, 3, 4, 5); shapes (1, 4, 5) and (2, 3, 1, 1) give (2, 3, 4, 5); shapes (3, 4, 5) and (2, 1, 1, 1) give (2, 3, 4, 5).

6.6.3. Unidirectional broadcasting

A tensor *B* is unidirectionally broadcastable to a tensor *A* where, after the shape of *B* has been extended on the left with dimensions of extent 1 until it has the rank of *A*, every position holds an extent of *B* that is either equal to the extent of *A* at that position or equal to 1.

The result has the shape of *A*. Unidirectional broadcasting is not symmetric: that *B* is broadcastable to *A* does not imply the converse.

6.6.4. Broadcasting and dimension variables

Where an extent is a dimension variable, it is equal to another extent only where that extent is the same dimension variable. A dimension variable is not known to be equal to 1, so a shape containing one is broadcastable only where the rule above is satisfied without appeal to the value of the variable.

6.7. Denotation

6.7.1. General

A type MAY carry a type denotation, and a dimension MAY carry a dimension denotation. A denotation ascribes a meaning to a type or an axis for the benefit of a [producer](#) or a [consumer](#).

A denotation SHALL NOT affect the result of evaluating a model. A consumer MAY ignore a denotation, and MAY reject a model whose denotations are inconsistent with the operators applied to the values concerned.

6.7.2. Type denotation

Table 8 lists the defined type denotations.

Table 8 — Type denotations

Denotation	Meaning
TENSOR	a tensor carrying no further ascribed meaning
IMAGE	an image; see Clause 5.5 for the properties that describe it
AUDIO	an audio signal
TEXT	a block of text

6.7.3. Dimension denotation

Table 9 lists the defined dimension denotations.

Table 9 — Dimension denotations

Denotation	Meaning
DATA_BATCH	the batch axis of the data
DATA_CHANNEL	the channel axis of the data
DATA_TIME	a time axis
DATA_FEATURE	a feature axis, including the spatial axes of an image
FILTER_IN_CHANNEL	the input-channel axis of a filter
FILTER_OUT_CHANNEL	the output-channel axis of a filter
FILTER_SPATIAL	a spatial axis of a filter

An operator that permutes, removes or introduces axes propagates the denotations of its input axes to the corresponding output axes.

NOTE Upstream describes denotation as an experimental mechanism. It is stated here because the encoding is part of the wire format and a consumer has to know what the values mean; nothing in this document requires a producer to supply a denotation.

6.8. Composite types

6.8.1. Sequence

A sequence is an ordered, homogeneous collection of values of a single type.

6.8.2. Map

A map is an unordered association of keys of an integral or string element type to values of a single type.

6.8.3. Optional

An optional value either holds a value of its element type or is absent. The element type is a tensor, sparse tensor, sequence or map type.

An optional type is distinct from the type of its element: a value of type optional tensor of INT64 is not a value of type tensor of INT64. The operators that convert between the two are defined in [ONNX 1-2](#).

NOTE This type is what [Clause 7](#) calls a dynamic-optional. It is not the same mechanism as an unsupplied operator input, which is described in [Clause 7.3.3](#).

6.8.4. Opaque

An opaque type is identified by a pair of a [domain](#) and a name. The name is required; an absent or empty domain denotes the default domain.

The meaning of an opaque type is defined by the operators of its domain, and this document ascribes none to it. A consumer that does not implement that domain SHALL treat a value of the type as unanalysable, and SHALL NOT infer anything from it beyond its identity.

NOTE An opaque type is how a custom domain introduces a kind of value of its own — a handle to something a pair of its operators pass between them — without a change to this document.

6.9. Type equality and assignability

Editorial note —

State when two types are equal, and when a value of one type may be supplied where another is expected. Shape is part of the type, so this clause decides whether a tensor of unknown shape is assignable to a formal input of known shape, which in turn decides how much [Clause 9](#) is obliged to prove. The upstream documentation does not state this; it is a rule the working group has to settle.

7. Graph structure

7.1. Graph, inputs and outputs

A [graph](#) SHALL have a name, and SHALL declare its inputs, its outputs and, optionally, its initializers and value type information.

A top-level graph SHALL declare the name, type and shape of each of its inputs and outputs. The shape states the rank; the individual extents need not be stated. See [Clause 6.5](#).

A subgraph SHALL declare the names of its inputs and outputs, and MAY declare their types.

A graph MAY declare the type and shape of values that are neither its inputs nor its outputs. Such a declaration is a constraint on the value, not a definition of it.

7.2. Initializers

An initializer is a named tensor value carried in the graph. The order of the initializers of a graph is not significant.

Where an initializer has the same name as an input of the same graph, it specifies a default value for that input. A [consumer](#) MAY allow the caller to supply a value for that input, overriding the initializer, and MAY allow the caller to omit it, in which case the value of the initializer is used.

Where an initializer has a name that is not the name of an input of the same graph, it specifies a constant. A caller SHALL NOT supply a value for it.

NOTE It follows that a producer that intends a value to be constant states it as an initializer only, and does not also declare it as a graph input.

In a subgraph, a name SHALL NOT be both an input and an initializer, unless the operator whose attribute carries the subgraph states otherwise. The inputs a control-flow operator supplies to a subgraph are matched to the inputs of that subgraph positionally, so an initializer of a subgraph that is also declared as an input SHALL be declared after every input the operator supplies.

Editorial note —

The subgraph rule above holds for models produced against IR version 4 and later. Earlier models were permitted to use one name as both a subgraph input and a subgraph initializer in order to carry a constant, and models in circulation rely on it. State whether this document admits those models, and if so under what compatibility rule; see [Clause 14](#).

7.3. Nodes

7.3.1. General

A [node](#) SHALL identify the [operator](#) it invokes by domain and name.

The number and types of a node's inputs and outputs, and the set and types of its attributes, SHALL satisfy the signature of the operator as defined in the operator set version declared by the enclosing model.

A node's inputs and outputs are matched to the formal inputs and outputs of its operator by position. Its attributes are matched by name.

A node MAY have a name. A node name is for diagnostic purposes and SHALL NOT affect the result of evaluating the graph.

7.3.2. Variadic inputs and outputs

The last formal input or output of an operator MAY be variadic, in which case the operator definition states its minimum arity N . A node invoking such an operator SHALL supply N or more inputs, respectively outputs, in that position.

7.3.3. Unsupplied inputs and outputs

An operator definition MAY mark a formal input as optional, in which case a node invoking the operator MAY leave it unsupplied. An operator definition MAY mark a formal output as optional, in which case a consumer MAY forgo computing it where the node leaves it unsupplied.

An input or output is left unsupplied in one of two ways: by omitting it, which is available only where it and every input or output after it is also omitted; or by giving an empty name in its position.

A node SHALL give a name for each output that is computed, and SHALL NOT give a name for an output that is not.

NOTE This mechanism is distinct from the optional type of [Clause 6.5](#). Leaving an input unsupplied does not change the type of anything; the optional type is a type, and the operators that construct and destruct it are defined in [ONNX 1-2](#).

7.4. Attributes

An [attribute](#) SHALL have a name that is unique within its node.

An attribute value SHALL NOT refer to a value produced by dataflow: it is fixed when the graph is constructed.

The value of an attribute is one of: a tensor, a sparse tensor, a numeric scalar, a string, a type, a graph, or a list of values of one of those kinds. The type system of attribute values is therefore not the type system of [Clause 6.5](#), which applies to values carried by dataflow.

7.5. Topological constraints

The nodes of a graph SHALL be topologically sorted: every node SHALL appear after the nodes producing the values it consumes.

The graph SHALL be acyclic.

Every value consumed by a node SHALL be produced by exactly one of: a graph input, an initializer, an earlier node in the same graph, or an enclosing scope; see [Clause 5.4.2](#).

A graph SHALL be in single static assignment form: every node output name SHALL be unique within the graph.

7.6. Subgraphs and scoping

Subgraphs appear as attribute values, and are evaluated by the operators that carry them; in the default domain these are If, Loop and Scan.

The names a subgraph may use from an enclosing graph, and the prohibition on shadowing, are stated in [Clause 5.4.3](#). The names of a subgraph's own inputs and initializers are constrained by [Clause 7.2](#).

The acyclicity requirement above applies to each graph separately. A subgraph SHALL NOT be reachable from itself through the attribute values of its own nodes.

Editorial note —

This clause now states the structural rules completely. What remains unwritten is the evaluation semantics of the operators that carry subgraphs: the branch selection of If, the termination condition and loop-carried dependencies of Loop, and the scan axis handling of Scan. Those belong to [Clause 8](#), and the upstream documentation states them only in the operator definitions, which is to say in [ONNX 1-2](#).

8. Graph semantics

8.1. Evaluation model

Evaluating a [graph](#) means computing a value for each of its declared outputs, given a value for each of its declared inputs.

A [model](#) denotes a stateless function. Evaluating the same model on the same inputs SHALL yield the same outputs, save where an operator of the model is permitted to be non-deterministic; see [Clause 8.5](#).

A graph has no side effects. In particular, an operator that would otherwise carry state across invocations expresses that state as an explicit output, which the caller passes to the next invocation as an input.

8.2. Evaluation order and concurrency

Evaluation is defined by the dataflow relation between nodes. This document does not prescribe an evaluation order beyond the constraint that a node's inputs are available before the node is evaluated; in particular, a [conforming implementation](#) MAY evaluate nodes concurrently, may elide nodes whose outputs are unused, and may fuse or reassociate computations, provided that the results remain within the tolerances of [Clause 11](#).

8.3. Control flow

The If, Loop and Scan operators evaluate subgraphs.

Editorial note —

Specify the evaluation semantics of each: the branch selection of If, the termination condition and loop-carried dependencies of Loop, and the scan axis handling of Scan. The structural constraints are in [Clause 7.6](#); this clause owes the semantics.

8.4. Functions

A function is an [operator](#) together with an implementation of it in terms of other operators, called its function body. The body is a topologically sorted list of nodes, as a graph is.

A function is identified by a triple of a [domain](#), a name and an overload. The overload distinguishes function bodies that are required to differ between call sites; where a model needs no such distinction the overload is empty. A function does not carry an operator set version of its own: the version is that of its domain, as imported by the model.

The parameters of a function are its inputs, its outputs and its attributes. An attribute parameter MAY have a default value. Where a node in the body uses an attribute parameter, that use is replaced by the value the call site supplies; where the call site supplies none, it is replaced by the default value; where there is no default value, the attribute is omitted from that node.

A function body is a fallback implementation, not a definition of the operator's numerical behaviour. A [conforming implementation](#) MAY compute the operator by any means, and the tolerances of [Clause 11](#) apply to the operator rather than to the evaluation of its body.

NOTE A function therefore serves two purposes at once. A consumer that does not implement the operator can still evaluate the model by expanding the body; a consumer that does implement it is not held to the body's rounding.

A function body MAY omit the types of the values it uses: a function MAY be polymorphic in the types of its parameters.

8.5. Determinism

Editorial note —

State which operators are permitted to be non-deterministic (random number generation, and reductions whose order is unspecified in floating point) and what a producer may assume. Without this clause, "within tolerance" is not testable.

8.6. Error semantics

A consumer that detects a violation of a requirement of this document SHALL report it and SHALL NOT produce a result. A consumer MAY continue validation in order to report further violations.

9. Shape and type inference

9.1. General

Shape and type inference derives the types and shapes of values that a [model](#) does not state explicitly.

Inference is normative rather than a convenience: consumers rely on it, and a [producer](#) is permitted to omit value type information for values other than graph inputs and outputs.

9.2. Obligations on operator definitions

Every operator definition in [ONNX 1-2](#) SHALL state the type and shape of each of its outputs as a function of the types and shapes of its inputs and of its attributes.

9.3. Symbolic dimensions

Where an output extent equals an input extent, the operator definition SHALL say so, so that a symbolic dimension propagates.

An implementation MAY introduce a fresh dimension variable for an extent it cannot determine, and every occurrence of that variable denotes the same extent as any other ([Clause 6.5.2](#)).

An implementation SHALL NOT conclude that two distinct dimension variables denote the same extent, and SHALL NOT conclude that a dimension variable denotes any particular integer.

NOTE It follows that inference does not compute over extents. Concatenating a tensor of extent 5 with one of extent 7 gives 12; concatenating one of extent 5 with one of extent N gives a fresh variable, not $N + 5$.

9.4. Strictness and failure

Inference is not required to be exact. Where an implementation cannot determine the type or the shape of a value, it SHALL yield unknown for that value rather than a value that is not entailed by the model.

An inference that yields unknown SHALL NOT of itself be a failure, and SHALL NOT prevent evaluation.

An inference that yields a result contradicting a type or shape the model states for the same value SHALL be reported under [Clause 10](#).

Editorial note —

What remains unstated is which operators are required to be inferable at all. Upstream does not require an inference function of every operator, and where one is absent the values downstream of that operator are simply unknown. Under [Clause 9.2](#) this document requires one of every operator, which is the stronger position and the one a standard should take; the gap between the two is what Annex C records.

9.5. Relationship to validation

Inference results are inputs to the type and shape checks of [Clause 10](#). An inference that returns unknown SHALL NOT by itself cause a validation error.

10. Model validation

10.1. General

Validation determines whether a [model](#) satisfies the requirements of this document, without evaluating it.

A validating consumer SHALL perform the checks of this clause and SHALL report every condition it classifies as an error.

Editorial note —

This clause is new in this structure. The draft previously defined a validating consumer conformance class and then never said what it must detect, which made the class unclaimable. The checks below are the intended shape of the clause; each needs to be stated as a testable condition and given a severity before the committee draft. Upstream's checker is the obvious source, but its behaviour is implementation rather than specification and diverges from the prose in places.

10.2. Structural checks

A validating consumer SHALL detect:

- a graph that is not acyclic;
- a node that consumes a value with no producer;
- a value produced more than once within one graph;
- a name that is empty;
- a node whose operator is not defined in any declared operator set.

10.3. Type and shape checks

A validating consumer SHALL detect a node whose input types do not satisfy the type constraints of its operator definition.

Editorial note —

State how far shape checking is obliged to go. Checking every statically known extent is stronger than upstream's checker performs; requiring nothing makes the check vacuous. The likely answer is that a mismatch between two statically known extents is an error and that unknown extents are not checked, but this must be decided together with [Clause 9](#).

10.4. Operator set checks

A validating consumer SHALL detect:

- a model that uses a domain it does not declare;
- a node whose operator is not present in the declared version of its domain;
- an attribute that is not defined for the operator, or a required attribute that is absent.

10.5. Severity and reporting

Each condition detected SHALL be reported as an error or a warning. An error means the model does not conform to this document. A warning means the model conforms but relies on behaviour this document leaves open.

A validating consumer SHOULD report all conditions it detects, not only the first.

11. Numerical behaviour

11.1. General

This clause states the numerical obligations on an evaluating consumer. It applies to every operator of [ONNX 1-2](#) unless that operator's definition states otherwise.

11.2. Floating-point arithmetic

Arithmetic on the FLOAT and DOUBLE element types SHALL follow [IEEE 754-2019](#) for the corresponding binary formats, subject to the reassociation permitted by [Clause 11.4](#).

11.3. Special values

Editorial note —

State the behaviour for NaN, the infinities and signed zero, per operator category rather than per operator. $\max(\text{NaN}, 0)$ is not well defined by [IEEE 754-2019](#) without stating which

of `maxNum` or `maximum` is intended, and that choice has to be made once here rather than repeated in every operator definition in [ONNX 1-2](#).

11.4. Accumulation and reassociation

An implementation MAY reassociate and may accumulate in a wider precision than the element type, provided the result remains within the tolerance of [Clause 11.5](#).

Editorial note —

Reassociation and the tolerance are two statements of one obligation and must be written together: a tolerance loose enough to permit any accumulation order is too loose to be a conformance criterion, and one tight enough to pin the order forbids ordinary optimizations. State the permitted reassociation, then derive the tolerance from it.

11.5. Tolerances

An evaluating consumer SHALL document the tolerance it achieves for each element type it supports.

Editorial note —

State the tolerance model. The reference implementation's test suite uses relative tolerance $1e-3$ and absolute tolerance $1e-7$ for float32, but this is a test-harness convention rather than a specified requirement, and it is not stated for float16, bfloat16 or the 8-bit and narrower float types. This clause is the single largest gap between current practice and a testable standard, and it is the clause [ONNX 1-3](#) depends on: a test vector without a stated tolerance cannot be passed or failed.

11.6. Non-deterministic operators

An operator whose definition in [ONNX 1-2](#) declares it non-deterministic is exempt from [Clause 11.5](#). The conformance criterion for such an operator SHALL be stated in its definition.

12. Serialization

12.1. General

An ONNX [model](#) is serialized as a sequence of octets. This clause specifies that sequence: the encoding of each kind of value, the framing of the fields of a message, and the treatment of a field a [consumer](#) does not recognize. [Annex B](#) states the messages and their fields.

The encoding is specified here in full. A [producer](#) or consumer can be implemented from this document alone, and this document does not require any other specification in order to be implemented.

NOTE The encoding defined here is the one a Protocol Buffers implementation produces for the schema from which [Annex B](#) is derived. Using such an implementation is therefore a convenient way to satisfy this clause, and the way most existing tools do satisfy it, but it is an implementation choice and not a requirement. See [Clause 12.7](#).

12.2. Primitive encodings

12.2.1. General

Four encodings carry every value. Table 10 names them and gives the wire type of each, which is the number that identifies the encoding in a field key ([Clause 12.3](#)).

Table 10 — Primitive encodings and their wire types

Wire type	Encoding	Used for
0	variable-length integer	int32, int64, uint64, bool, enumerations
1	64-bit	double
2	length-delimited	string, bytes, embedded messages, packed repeated fields
5	32-bit	float

The wire types 3 and 4 SHALL NOT appear. A consumer that encounters one SHALL reject the model.

NOTE Wire types 3 and 4 delimit a construct the format no longer uses. They are excluded so that a consumer can skip an unrecognized field by its wire type alone, which [Clause 12.4](#) requires.

12.2.2. Variable-length integer

A variable-length integer encodes a non-negative integer as a sequence of octets. The integer is divided into groups of seven bits, least significant group first. Each group occupies the seven low-order bits of one octet. The high-order bit of an octet is 1 where a further octet follows and 0 in the last octet.

The value encoded is at most $2^{64} - 1$, so the sequence is at most ten octets. A consumer SHALL reject a sequence of more than ten octets.

EXAMPLE — Variable-length integers

1 is 01. 300 is ac 02: the groups are $300 = 44 + 2 \cdot 128$, the first octet carries 44 with the continuation bit set, the second carries 2.

12.2.3. Signed integers

A value of a signed integer type is encoded as the variable-length integer of its representation as a 64-bit two's complement integer.

A negative value therefore occupies ten octets, whatever its magnitude and whether its type is int32 or int64.

EXAMPLE — A negative integer

-1 is ff ff ff ff ff ff ff ff ff 01, being the encoding of $2^{64} - 1$.

A consumer reading a field of type int32 SHALL reject a value outside the range of a 32-bit two's complement integer.

12.2.4. Fixed-width values

A 32-bit value is four octets, least significant octet first. A value of type float is the binary32 representation of [IEEE 754-2019](#) in that order.

A 64-bit value is eight octets, least significant octet first. A value of type double is the binary64 representation of [IEEE 754-2019](#) in that order.

The octets are those of the representation, so a signalling NaN, a quiet NaN and a negative zero are each preserved exactly.

12.2.5. Length-delimited values

A length-delimited value is a variable-length integer giving a length in octets, followed by that many octets.

The octets are:

- for `string`, the UTF-8 encoding of the character sequence, as specified in [ISO/IEC 10646](#), without a byte order mark;
- for `bytes`, the octets themselves;
- for an embedded message, the encoding of that message under this clause;
- for a packed repeated field, the encodings of its elements, in order, each without a field key ([Clause 12.3.3](#)).

A consumer SHALL reject a length that extends beyond the end of the enclosing message.

12.3. Fields

12.3.1. Field key

Each field is preceded by a field key, which is the variable-length integer of

$$(t \times 8) + w \tag{1}$$

where t is the tag of the field, as given in [Annex B](#), and w is its wire type, from Table 10.

A tag is at least 1. A field whose tag is at most 15 therefore has a key of one octet, and one whose tag is at least 16 has a key of two or more.

EXAMPLE — Field keys

The field of tag 1 and wire type 0 has the key `08`. The field of tag 20 and wire type 0 has the key `a0 01`.

12.3.2. Order and repetition of fields

The fields of a message MAY appear in any order. A producer SHOULD write them in order of increasing tag.

Where a field that is not repeated appears more than once, the value is that of the last occurrence, save that occurrences of an embedded message field are merged, field by field, under this same rule.

12.3.3. Repeated fields

A repeated field is encoded in one of two forms.

In the packed form, the field appears once, with wire type 2, and its length-delimited value holds the encodings of the elements in order, each without a field key. Only a repeated field whose elements are of a type with wire type 0, 1 or 5 may take this form.

In the unpacked form, the field key and value appear once per element, in order.

A producer SHOULD write a repeated field in the form [Annex B](#) records for it.

A consumer SHALL accept both forms for any repeated field, and SHALL accept a field that appears in both forms in one message, concatenating the elements in order of appearance.

EXAMPLE — The two forms

A repeated `int64` field of tag 5 holding 1 and 300 is `2a 03 01 ac 02` packed, and `28 01 28 ac 02` unpacked.

12.4. Unrecognized fields

A consumer that encounters a field whose tag it does not recognize SHALL skip it and SHALL continue reading the message. The length to skip is determined by the wire type in the field key alone:

- wire type 0: read the variable-length integer and discard it;
- wire type 1: skip eight octets;
- wire type 5: skip four octets;
- wire type 2: read the length and skip that many octets.

A consumer SHALL NOT reject a model for carrying a field it does not recognize.

NOTE This is what allows a model written against a later IR version to be read by a consumer written against an earlier one, so far as the fields they have in common allow. [Clause 14](#) states when that is enough for the model to be evaluated.

A consumer MAY retain the octets of a field it does not recognize. A consumer that rewrites a model and does not retain them SHALL NOT represent the result as the same model.

12.5. Limits

A consumer MAY impose a maximum size on a message and on the model as a whole. A consumer that does so SHALL document the limit and SHALL report a model exceeding it under [Clause 10](#); see [Clause 15.4](#).

Editorial note —

No maximum is stated here. The reference implementation applies one of 2 gigabytes to a single message, which is also the point beyond which a length does not fit the field it is read into in some implementations, and the external data mechanism of [Clause 13](#) exists so that a model can exceed it. State whether this document fixes that maximum or leaves it implementation-defined; if it is fixed, [Annex B](#) has to say which messages it applies to.

12.6. Message definitions

The messages, their fields, and for each field its tag, wire type and obligation are given normatively in [Annex B](#).

A tag identifies a field within its message. A tag SHALL NOT be reused for a different field, and a tag recorded as reserved SHALL NOT be used at all.

NOTE The obligation is what a schema cannot express, and is the reason [Annex B](#) is a table rather than a schema. In the schema this standard was derived from, every field is syntactically optional.

12.7. Relationship to Protocol Buffers

The encoding of [Clause 12.2](#) to [Clause 12.4](#) is that of Protocol Buffers, restricted to the constructs [Annex B](#) uses. An implementation MAY therefore be built on a Protocol Buffers library, by compiling a schema whose messages, fields, tags and types are those of [Annex B](#).

This document does not require that, and does not depend on any specification of Protocol Buffers. Where the two differ, this document governs a model claimed to conform to it.

NOTE The restriction is substantial and is what makes the encoding short enough to state here. Groups are excluded; no field uses a zig-zag encoded or fixed-width integer type; and the schema uses seven scalar types in all.

12.8. Canonical form

Editorial note —

Decide whether this document defines a canonical serialization. This clause admits encodings that differ octet for octet while carrying the same content: fields may appear in any order, a repeated field may be packed or not, and a signed value may be written in fewer octets than ten only where it is non-negative. A canonical form is a prerequisite for signing and for reproducible model hashes, both of which have been requested by adopters.

13. External data

13.1. General

The contents of a tensor MAY be stored outside the model file. A tensor SHALL declare, as its data location, either that its contents are carried in the model file or that they are external.

A tensor whose data location is external SHALL NOT also carry its contents in the model file. Where no data location is declared, the contents are carried in the model file.

The octets of external contents have the same layout as the raw contents of a tensor carried in the model file.

One external file MAY hold the contents of more than one tensor.

13.2. Reference resolution

An external reference is a set of key-value pairs. Table 11 states the keys this document defines and the obligation on each. A consumer SHALL reject a reference carrying a key not listed.

Table 11 — Keys of an external data reference

Key	Obligation	Value
location	mandatory	The path of the file holding the contents, relative to the directory holding the model file.
offset	optional	The position, in octets, at which the contents begin in that file. Where absent, the contents begin at the start of the file.
length	optional	The number of octets of contents. Where absent, the contents extend to the end of the file.
checksum	optional	A digest of the file named by location; see Clause 13.3 .

The value of location SHALL be a relative path, and SHALL NOT contain a parent-directory component. A consumer SHALL reject a reference whose location is an absolute path, contains a parent-directory component, or resolves outside the directory holding the model file. See [Clause 15](#) for the resolution rules a consumer applies in order to establish that.

A consumer that cannot resolve a reference, or that resolves it to a file shorter than offset plus length requires, SHALL report the failure under [Clause 10](#) and SHALL NOT evaluate the model.

NOTE A consumer may record the directory from which it loaded the model alongside the reference, so that resolution does not depend on the working directory at the time of evaluation. That record is not part of the model, and a producer does not write it.

A producer SHOULD align offset to a multiple of the page size of the target platform, so that a consumer can map the file into memory rather than copying it.

13.3. Integrity

A reference MAY carry a checksum, being the SHA-1 digest of the file named by its location.

Where a checksum is present, a consumer SHALL verify it before using the contents, and SHALL reject the model where it does not match.

Editorial note —

Two questions remain. SHA-1 is the digest the upstream format specifies, and it is no longer fit for the integrity of data from an untrusted source; this document should permit a stronger digest and state how the choice is encoded. And the checksum covers the whole file rather than the octets the reference selects, so it does not bind a tensor to its contents where one file holds several tensors. Until both are settled, external contents can change the meaning of a model that is otherwise byte-identical, which is the gap Annex C records.

14. Versioning and compatibility

14.1. Version axes

14.1.1. General

ONNX has three independent version axes:

IR version	the version of the structural and serialization rules given in Clause 5 , Clause 7 and Clause 12 .
Operator set version	the version of an operator set , per domain , as published in ONNX 1-2 .
Model version	an identifier assigned by the producer , opaque to this document.

14.1.2. The IR version this edition specifies

This edition specifies IR version 14, as published in ONNX release 1.23.0.

Table 12 is the version history.

Table 12 — IR versions, the release each was published in, and what each introduced

IR version	ONNX release	Introduced
8	1.10.0	the sparse tensor and optional types; model-local functions
9	1.14.0	attribute default values in a function; the four FP8 types
10	1.16.0	UINT4 and INT4; the function overload; metadata properties
11	1.18.0	FLOAT4E2M1; multi-device configuration
12	1.19.0	FLOAT8E8M0
13	1.20.0	UINT2 and INT2
14	1.23.0	FLOAT6E2M3 and FLOAT6E3M2; the opaque type outside the ONNX-ML build

NOTE The release is the publication event. The schema source carries a publication date against each IR version as well, and for IR version 14 it still reads “TBD” although 1.23.0 shipped it; the date in the schema is not maintained and is not relied on here.

14.2. IR version evolution

The IR version is a positive integer that increases monotonically.

A change that alters the structure or the meaning of a model, or that breaks a producer or consumer written against the previous version, SHALL increment the IR version. A change that does neither SHALL NOT.

NOTE 1 A change can break a producer or consumer without altering a single octet of any model — renaming a field of the serialized form is the standard example. Such a change increments the IR version.

A consumer SHOULD accept a model that omits a field the consumer does not require. A field that a given IR version requires SHALL be identified as such in [Annex B](#), and a producer SHALL supply every such field.

An increment of the IR version SHALL NOT of itself change any operator set. An increment of an operator set version SHALL NOT of itself change the IR version.

NOTE 2 The independence stated above is why this standard is in parts. The IR version is the cadence of this part; the operator set versions are the cadence of [ONNX 1-2](#).

14.3. Operator set evolution

An operator set version SHALL NOT be changed once published.

An operator is identified by a triple of a [domain](#), a name and the operator set version at which that definition of it was introduced. Where the definition of an operator changes, the changed definition is a new operator with a new such version, and the previous definition remains what it was.

Where the inventory of an operator set changes, whether by addition, by removal or by a change to the definition of an operator it contains, a new operator set version SHALL be published.

Table 13 states the changes to an operator that require a new version of it.

Table 13 — Changes requiring a new version of an operator

Change	New version required
Adding, removing or renaming an attribute, including adding an optional attribute whose default value reproduces the previous behaviour	yes
Adding, removing or reordering an input or an output	yes
Adding or removing a type admitted by an input or an output, or changing the type of an attribute	yes
Admitting behaviour the previous definition did not, under an unchanged signature	yes
Resolving an ambiguity in the previous definition in favour of what implementations already do	no

NOTE The third row is symmetric: widening a type constraint requires a new version of the operator just as narrowing one does. What the previous row's prohibition protects is the previous definition, which stays available at its own version.

A model binds each of its nodes to an operator definition by the operator set version the model imports; see [Clause 5.3.2](#). How a consumer binds that definition to an implementation is outside the scope of this document.

14.4. Compatibility obligations

A consumer that claims support for operator set version n of a domain SHALL accept models declaring any version $m \leq n$ of that domain, and SHALL evaluate each node according to the operator definition in force at version m .

Editorial note —

The above states backward compatibility as an obligation. Current runtimes do not universally meet it. Either the obligation is weakened, or it is stated as written and the gap is recorded in Annex C as a known divergence. The latter is recommended: a standard states the requirement, and a conformance statement records what an implementation achieves.

14.5. Deprecation and removal

An operator MAY be marked deprecated in an operator set version. Deprecation SHALL NOT remove the operator, and a consumer SHALL continue to evaluate deprecated operators.

15. Security and resource limits

15.1. General

A [model](#) is data from an untrusted source unless the consumer knows otherwise. This clause states the obligations that follow.

15.2. Threat model

A [consumer](#) defends against a model crafted so that processing it causes the consumer to read or write a file the caller did not intend, to consume unbounded memory, processor time or disk, or to recur without bound.

Table 14 states the attacks a consumer is obliged to defend against and the clause that states the defence.

Table 14 — Threats and the defences required

Threat	Description	Defence
Path traversal	An external data reference escapes the model directory through a parent-directory component or an absolute path.	Clause 15.3
Symbolic link traversal	A component of an external data path is a symbolic link resolving outside the model directory.	Clause 15.3
Hard link exposure	A file in the model directory is a hard link to a file outside it, and so passes a check that considers symbolic links alone.	Clause 15.3
Resource exhaustion	A model declares a tensor, a dimension, a loop bound or an external file so large that evaluating it exhausts memory, disk or time.	Clause 15.4
Unbounded recursion	Subgraphs nest so deeply that evaluating them exhausts the evaluation stack.	Clause 15.4

The following are outside the scope of this document: defects in the implementation of an individual operator; the integrity of the channel by which a model was obtained; and the behaviour of a consumer processing a file that is not an ONNX model.

15.3. External data and path resolution

A consumer SHALL resolve the location of an external data reference ([Clause 13.2](#)) against the directory holding the model file, and SHALL reject a reference that resolves outside that directory. Resolution SHALL take account of every component of the path, not the final component alone.

A consumer SHALL reject a reference whose path, or any component of it, is a symbolic link.

A consumer SHALL reject a reference naming a file that has more than one hard link.

A consumer SHALL establish that the file it has opened is the file the resolution identified, and SHALL NOT rely on the path alone; a path checked before opening can name a different file by the time it is opened.

A consumer MAY be configured to resolve references outside the model directory. Such a configuration SHALL be explicit and SHALL NOT be the default.

Rejection under this subclause is a validation failure under [Clause 10](#).

NOTE These requirements are stated as obligations on the outcome, not on the mechanism. The upstream implementation meets them with a layered check — canonical containment, a symbolic link test, a constrained open, and a hard link count — and the constrained open is the only layer that is not subject to a change of the file system between the check and the open.

15.4. Resource bounds

A consumer MAY impose a limit on the memory it allocates for a model, on the extent of a dimension, on the depth to which subgraphs nest, on the number of iterations of a loop, and on the size of an external data file.

A consumer that imposes such a limit SHALL document it, and SHALL report a model that exceeds it as a refusal under [Clause 10](#) rather than failing in any other way.

A consumer SHALL validate the declared size of an external data reference against the size of the file before allocating for it.

NOTE A consumer that refuses a model for exceeding a documented limit conforms; one that crashes does not. The limits themselves are implementation-defined, and a conformance statement ([Annex A](#)) records them.

15.5. Handling untrusted models

A consumer SHALL validate a model under [Clause 10](#) before evaluating it.

A consumer SHOULD make the resolution of external data an explicit, opt-in step when the model is untrusted.

16. Maintenance and extension

16.1. Custom domains

A model MAY use operators in a [domain](#) not defined by [ONNX 1-2](#).

A consumer that does not implement such a domain SHALL report the operator as unsupported under [Clause 4](#), and SHALL NOT evaluate the model.

16.2. Experimental operators

This document does not recognize an experimental status for an operator. Every operator of an operator set published under [ONNX 1-2](#) is subject without exception to [Clause 14.3](#).

An operator whose definition is not yet settled SHALL be published in a [domain](#) of its own, and is a custom operator under the preceding subclause.

NOTE Upstream reached the same position. The serialized form of an operator set carries a status of EXPERIMENTAL or STABLE, and an operator marked STABLE may not change; but experimental operators have not been accepted into the ONNX domains since release 1.5, and the operators that had been are withdrawn. The field remains in the format, and this document gives it no effect.

A producer SHOULD replace a withdrawn experimental operator with the operators that supersede it. [ONNX 1-2](#) records the correspondence.

16.3. Adding an operator

The procedure by which an operator enters an [operator set](#) is specified in [ONNX 1-2](#). The constraints that procedure must respect are in [Clause 14](#).

16.4. Amending this document

Editorial note —

State the amendment process for this part, and its relationship to the release cadence of [ONNX 1-2](#). The division into parts exists so that an operator added upstream does not force a new edition of this part; the amendment process is what makes that real rather than aspirational.

Annex A (normative) Conformance statement

A conformance statement claiming conformance to this document SHALL supply the information in Table A.1.

Table A.1 — Content of a conformance statement

Item	Obligation
Identity and version of the implementation	mandatory
Conformance classes claimed (see Clause 4)	mandatory
Conformance profiles claimed, if any	mandatory
IR versions supported	mandatory
For each domain, the operator set versions of ONNX 1-2 supported	mandatory
For each operator set version, the operators not supported	mandatory
For each element type supported, the numerical tolerance achieved	mandatory
Result of the test package of ONNX 1-3 , by version	mandatory
Known divergences from this document	mandatory
External data handling policy, including path resolution	conditional, if external data is supported
Documented resource limits (see Clause 15)	mandatory

Annex B (normative) Message definitions

B.1. General

This annex states the message definitions of the serialized form: 35 messages and 166 fields. [Clause 12](#) states how the octets of a message are laid out; this annex says which fields there are.

The tag of a field identifies it within its message and SHALL NOT be reused. The wire type is the encoding of its value, per Table 10; together the two make the field key ([Clause 12.3.1](#)).

A field marked mandatory SHALL be present. A field marked optional MAY be absent, and a [consumer](#) SHALL accept a message in which it is. A field marked repeated holds zero or more values, and a [producer](#) SHOULD write it in the form recorded here, packed or not; a consumer SHALL accept either ([Clause 12.3.3](#)). A field marked deprecated SHALL NOT be written by a producer, and a consumer that reads one SHALL ignore it.

The obligations are those in force at the IR version stated in [Clause 14.1.2](#).

NOTE 23 of the 166 fields are mandatory. A schema cannot express the distinction: in the syntax this standard was derived from every field is optional, and the obligation is carried in the comments. Stating it in a table is the reason this annex exists.

Table B.1 — Subclauses of this annex and the schema file each restates

Subclause	Source	Messages
B.2 Model, graph and tensor messages	upstream/onnx/proto/onnx-ml.proto	30
B.3 Operator set messages	upstream/onnx/proto/onnx-operators-ml.proto	2
B.4 Map and sequence messages	upstream/onnx/proto/onnx-data.proto	3

B.2. Model, graph and tensor messages

B.2.1. Version

Table B.2 — Values of Version

Name	Value
_START_VERSION	0
IR_VERSION_2017_10_10	1
IR_VERSION_2017_10_30	2
IR_VERSION_2017_11_3	3
IR_VERSION_2019_1_22	4
IR_VERSION_2019_3_18	5
IR_VERSION_2019_9_19	6
IR_VERSION_2020_5_8	7
IR_VERSION_2021_7_30	8
IR_VERSION_2023_5_5	9
IR_VERSION_2024_3_25	10
IR_VERSION_2025_05_12	11
IR_VERSION_2025_08_26	12
IR_VERSION_2025_11_06	13
IR_VERSION	14

B.2.2. AttributeProto

Table B.3 — Fields of AttributeProto

Field	Tag	Wire type	Type	Obligation
name	1	2	string	mandatory

Field	Tag	Wire type	Type	Obligation
ref_attr_name	21	2	string	optional
doc_string	13	2	string	optional
type	20	0	AttributeType	mandatory
f	2	5	float	mandatory
i	3	0	int64	optional
s	4	2	bytes	optional
t	5	2	TensorProto	optional
g	6	2	GraphProto	optional
sparse_tensor	22	2	SparseTensorProto	optional
tp	14	2	TypeProto	deprecated
floats	7	5	float	repeated
ints	8	0	int64	repeated
strings	9	2	bytes	repeated
tensors	10	2	TensorProto	repeated
graphs	11	2	GraphProto	repeated
sparse_tensors	23	2	SparseTensorProto	repeated
type_protos	15	2	TypeProto	repeated

Tags and names reserved in AttributeProto, which SHALL NOT be used: 12, 16 to 19; "v".

Table B.4 — Values of AttributeProto.AttributeType

Name	Value
UNDEFINED	0
FLOAT	1
INT	2
STRING	3
TENSOR	4
GRAPH	5
SPARSE_TENSOR	11
TYPE_PROTO	13
FLOATS	6
INTS	7
STRINGS	8
TENSORS	9
GRAPHS	10
SPARSE_TENSORS	12
TYPE_PROTOS	14

B.2.3. ValueInfoProto**Table B.5 — Fields of ValueInfoProto**

Field	Tag	Wire type	Type	Obligation
name	1	2	string	mandatory
type	2	2	TypeProto	mandatory
doc_string	3	2	string	optional
metadata_props	4	2	StringStringEntryProto	repeated

B.2.4. NodeProto**Table B.6 — Fields of NodeProto**

Field	Tag	Wire type	Type	Obligation
input	1	2	string	repeated
output	2	2	string	repeated
name	3	2	string	optional
op_type	4	2	string	optional
domain	7	2	string	optional
overload	8	2	string	optional

Field	Tag	Wire type	Type	Obligation
attribute	5	2	AttributeProto	repeated
doc_string	6	2	string	optional
metadata_props	9	2	StringStringEntryProto	repeated
device_configurations	10	2	NodeDeviceConfigurationProto	repeated

B.2.5. IntIntListEntryProto

Table B.7 — Fields of IntIntListEntryProto

Field	Tag	Wire type	Type	Obligation
key	1	0	int64	optional
value	2	0	int64	repeated

B.2.6. NodeDeviceConfigurationProto

Table B.8 — Fields of NodeDeviceConfigurationProto

Field	Tag	Wire type	Type	Obligation
configuration_id	1	2	string	mandatory
sharding_spec	2	2	ShardingSpecProto	repeated
pipeline_stage	3	0	int32	optional

B.2.7. ShardingSpecProto

Table B.9 — Fields of ShardingSpecProto

Field	Tag	Wire type	Type	Obligation
tensor_name	1	2	string	mandatory
device	2	0	int64	repeated
index_to_device_group_map	3	2	IntIntListEntryProto	repeated
sharded_dim	4	2	ShardedDimProto	repeated

B.2.8. ShardedDimProto

Table B.10 — Fields of ShardedDimProto

Field	Tag	Wire type	Type	Obligation
axis	1	0	int64	mandatory
simple_sharding	2	2	SimpleShardedDimProto	repeated

B.2.9. SimpleShardedDimProto

Table B.11 — Fields of SimpleShardedDimProto

Field	Tag	Wire type	Type	Obligation
dim_value	1	0	int64	one of the group
dim_param	2	2	string	one of the group
num_shards	3	0	int64	mandatory

Exactly one of dim_value, dim_param SHALL be present, being the group dim.

B.2.10. TrainingInfoProto

Table B.12 — Fields of TrainingInfoProto

Field	Tag	Wire type	Type	Obligation
initialization	1	2	GraphProto	optional
algorithm	2	2	GraphProto	optional
initialization_binding	3	2	StringStringEntryProto	repeated

Field	Tag	Wire type	Type	Obligation
update_binding	4	2	StringStringEntryProto	repeated

B.2.11. ModelProto

Table B.13 — Fields of ModelProto

Field	Tag	Wire type	Type	Obligation
ir_version	1	0	int64	optional
opset_import	8	2	OperatorSetIdProto	repeated
producer_name	2	2	string	optional
producer_version	3	2	string	optional
domain	4	2	string	optional
model_version	5	0	int64	optional
doc_string	6	2	string	optional
graph	7	2	GraphProto	optional
metadata_props	14	2	StringStringEntryProto	repeated
training_info	20	2	TrainingInfoProto	repeated
functions	25	2	FunctionProto	repeated
configuration	26	2	DeviceConfigurationProto	repeated

B.2.12. DeviceConfigurationProto**Table B.14 — Fields of DeviceConfigurationProto**

Field	Tag	Wire type	Type	Obligation
name	1	2	string	mandatory
num_devices	2	0	int32	mandatory
device	3	2	string	repeated

B.2.13. StringStringEntryProto**Table B.15 — Fields of StringStringEntryProto**

Field	Tag	Wire type	Type	Obligation
key	1	2	string	optional
value	2	2	string	optional

B.2.14. TensorAnnotation**Table B.16 — Fields of TensorAnnotation**

Field	Tag	Wire type	Type	Obligation
tensor_name	1	2	string	optional
quant_parameter_tensor_names	2	2	StringStringEntryProto	repeated

B.2.15. GraphProto**Table B.17 — Fields of GraphProto**

Field	Tag	Wire type	Type	Obligation
node	1	2	NodeProto	repeated
name	2	2	string	optional
initializer	5	2	TensorProto	repeated
sparse_initializer	15	2	SparseTensorProto	repeated
doc_string	10	2	string	optional
input	11	2	ValueInfoProto	repeated
output	12	2	ValueInfoProto	repeated
value_info	13	2	ValueInfoProto	repeated
quantization_annotation	14	2	TensorAnnotation	repeated

Field	Tag	Wire type	Type	Obligation
metadata_props	16	2	StringStringEntryProto	repeated
Tags and names reserved in GraphProto, which SHALL NOT be used: 3, 4, 6 to 9; "ir_version", "producer_version", "producer_tag", "domain".				

B.2.16. TensorProto

Table B.18 — Fields of TensorProto

Field	Tag	Wire type	Type	Obligation
dims	1	0	int64	repeated
data_type	2	0	int32	optional
segment	3	2	Segment	optional
float_data	4	2	float	repeated, packed
int32_data	5	2	int32	repeated, packed
string_data	6	2	bytes	repeated
int64_data	7	2	int64	repeated, packed
name	8	2	string	optional
doc_string	12	2	string	optional
raw_data	9	2	bytes	optional
external_data	13	2	StringStringEntryProto	repeated
data_location	14	0	DataLocation	optional
double_data	10	2	double	repeated, packed
uint64_data	11	2	uint64	repeated, packed
metadata_props	16	2	StringStringEntryProto	repeated

Table B.19 — Values of TensorProto.DataType

Name	Value
UNDEFINED	0
FLOAT	1
UINT8	2
INT8	3
UINT16	4
INT16	5
INT32	6
INT64	7
STRING	8
BOOL	9
FLOAT16	10
DOUBLE	11
UINT32	12
UINT64	13
COMPLEX64	14
COMPLEX128	15
BFLOAT16	16
FLOAT8E4M3FN	17
FLOAT8E4M3FNUZ	18
FLOAT8E5M2	19
FLOAT8E5M2FNUZ	20
UINT4	21
INT4	22
FLOAT4E2M1	23
FLOAT8E8M0	24
UINT2	25
INT2	26
FLOAT6E2M3	27
FLOAT6E3M2	28

Table B.20 — Values of TensorProto.DataLocation

Name	Value
DEFAULT	0
EXTERNAL	1

B.2.17. TensorProto.Segment**Table B.21 — Fields of TensorProto.Segment**

Field	Tag	Wire type	Type	Obligation
begin	1	0	int64	optional
end	2	0	int64	optional

B.2.18. SparseTensorProto**Table B.22 — Fields of SparseTensorProto**

Field	Tag	Wire type	Type	Obligation
values	1	2	TensorProto	optional
indices	2	2	TensorProto	optional
dims	3	0	int64	repeated

B.2.19. TensorShapeProto**Table B.23 — Fields of TensorShapeProto**

Field	Tag	Wire type	Type	Obligation
dim	1	2	Dimension	repeated

B.2.20. TensorShapeProto.Dimension**Table B.24 — Fields of TensorShapeProto.Dimension**

Field	Tag	Wire type	Type	Obligation
dim_value	1	0	int64	one of the group
dim_param	2	2	string	one of the group
denotation	3	2	string	optional

Exactly one of dim_value, dim_param SHALL be present, being the group value.

B.2.21. TypeProto**Table B.25 — Fields of TypeProto**

Field	Tag	Wire type	Type	Obligation
tensor_type	1	2	Tensor	one of the group
sequence_type	4	2	Sequence	one of the group
map_type	5	2	Map	one of the group
optional_type	9	2	Optional	one of the group
sparse_tensor_type	8	2	SparseTensor	one of the group
opaque_type	7	2	Opaque	one of the group
denotation	6	2	string	optional

Exactly one of tensor_type, sequence_type, map_type, optional_type, sparse_tensor_type, opaque_type SHALL be present, being the group value.

B.2.22. TypeProto.Tensor**Table B.26 — Fields of TypeProto.Tensor**

Field	Tag	Wire type	Type	Obligation
elem_type	1	0	int32	mandatory

Field	Tag	Wire type	Type	Obligation
shape	2	2	TensorShapeProto	optional

B.2.23. TypeProto.Sequence

Table B.27 — Fields of TypeProto.Sequence

Field	Tag	Wire type	Type	Obligation
elem_type	1	2	TypeProto	mandatory

B.2.24. TypeProto.Map

Table B.28 — Fields of TypeProto.Map

Field	Tag	Wire type	Type	Obligation
key_type	1	0	int32	mandatory
value_type	2	2	TypeProto	mandatory

B.2.25. TypeProto.Optional

Table B.29 — Fields of TypeProto.Optional

Field	Tag	Wire type	Type	Obligation
elem_type	1	2	TypeProto	mandatory

B.2.26. TypeProto.SparseTensor

Table B.30 — Fields of TypeProto.SparseTensor

Field	Tag	Wire type	Type	Obligation
elem_type	1	0	int32	mandatory
shape	2	2	TensorShapeProto	optional

B.2.27. TypeProto.Opaque

Table B.31 — Fields of TypeProto.Opaque

Field	Tag	Wire type	Type	Obligation
domain	1	2	string	optional
name	2	2	string	optional

Tags and names reserved in TypeProto.Opaque, which SHALL NOT be used: 3; "parameters".

B.2.28. OperatorSetIdProto**Table B.32 — Fields of OperatorSetIdProto**

Field	Tag	Wire type	Type	Obligation
domain	1	2	string	mandatory
version	2	0	int64	mandatory

B.2.29. OperatorStatus

Table B.33 — Values of OperatorStatus

Name	Value
EXPERIMENTAL	0
STABLE	1

B.2.30. FunctionProto

Table B.34 — Fields of FunctionProto

Field	Tag	Wire type	Type	Obligation
name	1	2	string	optional
input	4	2	string	repeated
output	5	2	string	repeated
attribute	6	2	string	repeated
attribute_proto	11	2	AttributeProto	repeated
node	7	2	NodeProto	repeated
doc_string	8	2	string	optional
opset_import	9	2	OperatorSetIdProto	repeated
domain	10	2	string	optional
overload	13	2	string	optional
value_info	12	2	ValueInfoProto	repeated
metadata_props	14	2	StringStringEntryProto	repeated

Tags and names reserved in FunctionProto, which SHALL NOT be used: 2; "since_version"; 3; "status".

B.3. Operator set messages

B.3.1. OperatorProto

Table B.35 — Fields of OperatorProto

Field	Tag	Wire type	Type	Obligation
op_type	1	2	string	mandatory
since_version	2	0	int64	mandatory
status	3	0	OperatorStatus	optional
doc_string	10	2	string	optional

B.3.2. OperatorSetProto

Table B.36 — Fields of OperatorSetProto

Field	Tag	Wire type	Type	Obligation
magic	1	2	string	mandatory
ir_version	2	0	int64	mandatory
ir_version_prerelease	3	2	string	optional
ir_build_metadata	7	2	string	optional
domain	4	2	string	optional
opset_version	5	0	int64	optional
doc_string	6	2	string	optional
operator	8	2	OperatorProto	repeated
functions	9	2	FunctionProto	repeated

B.4. Map and sequence messages

B.4.1. SequenceProto

Table B.37 — Fields of SequenceProto

Field	Tag	Wire type	Type	Obligation
name	1	2	string	optional
elem_type	2	0	int32	optional
tensor_values	3	2	TensorProto	repeated
sparse_tensor_values	4	2	SparseTensorProto	repeated
sequence_values	5	2	SequenceProto	repeated

Field	Tag	Wire type	Type	Obligation
map_values	6	2	MapProto	repeated
optional_values	7	2	OptionalProto	repeated

Table B.38 — Values of SequenceProto.DataType

Name	Value
UNDEFINED	0
TENSOR	1
SPARSE_TENSOR	2
SEQUENCE	3
MAP	4
OPTIONAL	5

B.4.2. MapProto**Table B.39 — Fields of MapProto**

Field	Tag	Wire type	Type	Obligation
name	1	2	string	optional
key_type	2	0	int32	optional
keys	3	0	int64	repeated
string_keys	4	2	bytes	repeated
values	5	2	SequenceProto	optional

B.4.3. OptionalProto**Table B.40 — Fields of OptionalProto**

Field	Tag	Wire type	Type	Obligation
name	1	2	string	optional
elem_type	2	0	int32	optional
tensor_value	3	2	TensorProto	optional
sparse_tensor_value	4	2	SparseTensorProto	optional
sequence_value	5	2	SequenceProto	optional
map_value	6	2	MapProto	optional
optional_value	7	2	OptionalProto	optional

Table B.41 — Values of OptionalProto.DataType

Name	Value
UNDEFINED	0
TENSOR	1
SPARSE_TENSOR	2
SEQUENCE	3
MAP	4
OPTIONAL	5

B.5. Derivation

The tables above are derived from the Protocol Buffers schema of the ONNX reference implementation, vendored in the repository at `upstream/onnx/proto/` at the release recorded in `upstream/onnx/SOURCE.txt`. That schema is an informative aid, not a normative reference: the tables above are normative, and where a comment in the schema carries a normative statement, that statement is in the clause it belongs to. See [Clause 12.7](#).

Annex C (informative) Known gaps and divergences

C.1. Open gaps

This annex records, for the working group's use, the points at which the present ONNX specification and its reference implementation do not yet supply what a normative standard requires. It is informative and is expected to shrink to nothing before this document is submitted anywhere.

Every clause carrying an editorial note has at least one row here. A row may cover several related notes in the same clause; a clause with a note and no row is an omission.

NOTE A gap recorded here is a gap in the source material, not a clause left unwritten out of inattention. Where the ONNX documentation states a rule, this document states it; the rows below are what it does not state, or states in a form a standard cannot adopt.

Table C.1 — Known gaps between current ONNX practice and this document

Clause	Gap	Severity
Clause 11.5	No specified numerical tolerance; the test suite's tolerances are a harness convention. Clause 6.3 now states the conversion and saturation rules of the narrow types, but a tolerance does not follow from them, and ONNX 1-3 cannot pass or fail a vector without one.	Blocking
Clause 10	The checks a validating consumer owes are drafted but not yet stated as testable conditions with severities.	Blocking
ONNX 1-2 Clauses 5, 6	The generated operator clauses supply shape inference, determinism and error conditions for 0 of 222 operators: the upstream documentation states none of them. They cannot be generated and have to be written per operator.	Blocking
Clause 6.3 (conversion)	Whether a conversion to a narrow floating-point type saturates or yields NaN is settled per operator, by an attribute of Cast, rather than per type, so one model can convert the same pair of types both ways. And the result of converting an out-of-range value to a narrow integer type is unstated: the source says the value is truncated without saying to what.	Major
Clause 6.4	STRING elements are octet strings in the implementation and UTF-8 by assumption in much tooling. The upstream documentation does not settle it.	Major
Clause 6 (type equality)	When two types are equal, and when a value of one type may be supplied where another is expected, is unstated; this bounds what Clause 9 must prove. The upstream documentation does not state it either.	Major
Clause 8 (control flow)	The evaluation semantics of If, Loop and Scan are unwritten here. The structural rules are now in Clause 7.6 ; upstream states the semantics only in the operator definitions, which is to say in ONNX 1-2 , so Clause 8 of this part and Clause 5 of that one have to be reconciled.	Major
Clause 8 (determinism)	No statement of which operators may be non-deterministic. Clause 8 now requires a model to be a stateless function save where an operator is permitted to be otherwise; the set of such operators is still unstated.	Major
Clause 11 (special values, reassociation)	NaN and infinity behaviour, and the permitted reassociation, are unwritten; the tolerance cannot be derived until they are.	Major
Clause 14 (compatibility)	Backward compatibility is stated as an obligation here; runtimes do not universally meet it.	Major

Clause	Gap	Severity
Clause 14 , ONNX 1-2	Clause 14.5 requires a consumer to keep evaluating a deprecated operator, but upstream stops documenting the inputs, outputs and type constraints of one once deprecated. Four operators are in that state, so their signature cannot be implemented from the documentation at all.	Major
Clause 12 (limits)	No maximum message size is stated. The rest of the encoding is now specified in Clause 12 , but the reference implementation applies a limit of 2 gigabytes to a single message and this document neither fixes that nor leaves it explicitly to the implementation.	Minor
Annex B	The annex states which fields are mandatory at one IR version, being the one Clause 14.1.2 names. The schema source carries no history of the obligations, so a model produced against an earlier IR version cannot be checked against this annex, and Clause 14.4 requires that it can be.	Major
Clause 13.3	The digest an external reference may carry is SHA-1, which is not fit for data from an untrusted source, and it covers the file rather than the octets a reference selects. Resolution itself is now specified in Clause 13.2 .	Major
Clause 9	Which operators are required to be inferable is unstated. Clause 9.2 requires an inference function of every operator; upstream requires one of none, and leaves the values downstream of an operator without one unknown. The two positions have to be reconciled.	Major
Clause 4 (profiles)	No conformance profile is defined, so the class system has no granularity.	Minor
Clause 12.8	No canonical serialization, blocking model signing and reproducible hashes.	Minor
Clause 16 (amendment)	The amendment process for this part, and its relationship to the release cadence of ONNX 1-2 , is unwritten. The status of experimental operators is now settled in Clause 16 .	Minor
Clause 7.2	The relationship between an initializer and a graph input is now specified for models produced against IR version 4 and later. Whether this document admits the earlier reading, under which one name in a subgraph could be both, is unsettled.	Minor
Clause 6.6 , ONNX 1-2	Broadcasting is now specified in Clause 6.6 , but the operator prose of ONNX 1-2 still refers to it in plain text rather than by cross-reference, because a reference from one part of this standard to a clause of another is not resolved by the toolchain.	Minor
Clause 4 (verbal forms)	The document uses RFC 2119 capitalised keywords; ISO/IEC Directives Part 2 requires “shall/should/may/can”. An editorial migration, not a technical one.	Minor

C.2. Gaps closed from the upstream documentation

Table C.2 records the rows removed from the table above, and the upstream document each was closed from. It is kept so that a reviewer can check the transcription rather than take it on trust, and so that a refresh of `upstream/onnx/` can be checked against the clauses that depend on it.

Table C.2 — Gaps closed, and the source each was closed from

Clause	Source
Clause 6.2 (complete enumeration)	<code>onnx-m1.proto, TensorProto.DataType</code>
Clause 6.3.4 (typed storage)	<code>onnx-m1.proto, TensorProto.int32_data</code>
Annex B (the whole annex)	<code>upstream/onnx/proto/, generated by scripts/generate-schema.rb</code>

Clause	Source
Clause 12 (the binary encoding)	upstream/onnx/proto/, for the types and tags; the encoding itself is checked against a reference implementation by scripts/check-encoding.rb
Table 12 (the IR version history)	docs/Versioning.md, "Released Versions"; onnx-ml.proto, Version
Clause 6.3 (narrow types and their packing)	docs/docsgen/source/technical/: float4.md, float6.md, float8.md, int2.md, int4.md
Clause 6.6	docs/Broadcasting.md
Clause 6.5.2 (model-wide scope)	docs/IR.md, "Static tensor shapes"
Clause 6.7	docs/TypeDenotation.md, docs/ DimensionDenotation.md
Clause 6.8.4	docs/ONNXTypes.md
Clause 5.3 , Clause 5.3.2	docs/IR.md, "Models", "Operator Set Identifiers", "Operator Sets"
Clause 5.4 , Clause 5.4.3	docs/IR.md, "Names Within a Graph", "Nodes"
Clause 5.5	docs/IR.md, "Optional Metadata"; docs/MetadataProps. md
Clause 7.2 (IR version 4 and later)	docs/IR.md, "Nodes"
Clause 7.3.3 (variadic and unsupplied inputs and outputs)	docs/IR.md, "Variadic Inputs and Outputs", "Optional Inputs and Outputs"
Clause 8.4	docs/IR.md, "Functions"
Clause 8 (model as a stateless function)	docs/IR.md, "Model Semantics"
Clause 13.2	docs/ExternalData.md, "TensorProto: data_location and external_data fields"
Clause 15.2 , Clause 15.3 , Clause 15.4	docs/ExternalDataSecurity.md
Clause 14 (IR version evolution, operator set evolution)	docs/Versioning.md
Clause 9 (strictness, symbolic dimensions)	docs/ShapeInference.md, "Limitations"
Clause 16 (experimental operators)	docs/ManagingExperimentalOps.md

NOTE The baseline is the vendored copy in upstream/onnx/, whose release and commit are recorded in upstream/onnx/SOURCE.txt. Nothing above was taken from the upstream repository outside that copy.

Annex D

(informative)

Correspondence with the ONNX reference implementation

This document is a restatement of the ONNX format as documented by the ONNX project. It is not derived from the reference implementation's source, and the implementation is not reproduced here: it is large, it moves with every release, and a copy inside a standard would age badly.

The documentation this part was written against is vendored in the repository under `upstream/onnx/`, at the release recorded in its provenance file, so that every statement in this document can be checked against a fixed tree.

Where this document and the reference implementation disagree, the divergence is recorded in Annex C rather than resolved silently. Resolving such a divergence is a decision for the ONNX Steering Committee, not for the editor.

Bibliography

- [1] ISO/IEC DIR 2, ISO and IEC.
- [2] ISO/IEC JTC 1 PAS, *Publicly Available Specification (PAS) transposition process*
- [3] Protocol Buffers, *Protocol Buffers*, <https://protobuf.dev>
- [4] ONNX, *Open Neural Network Exchange*, <https://onnx.ai>
- [5] JDF, *Joint Development Foundation*, <https://www.jointdevelopment.org>