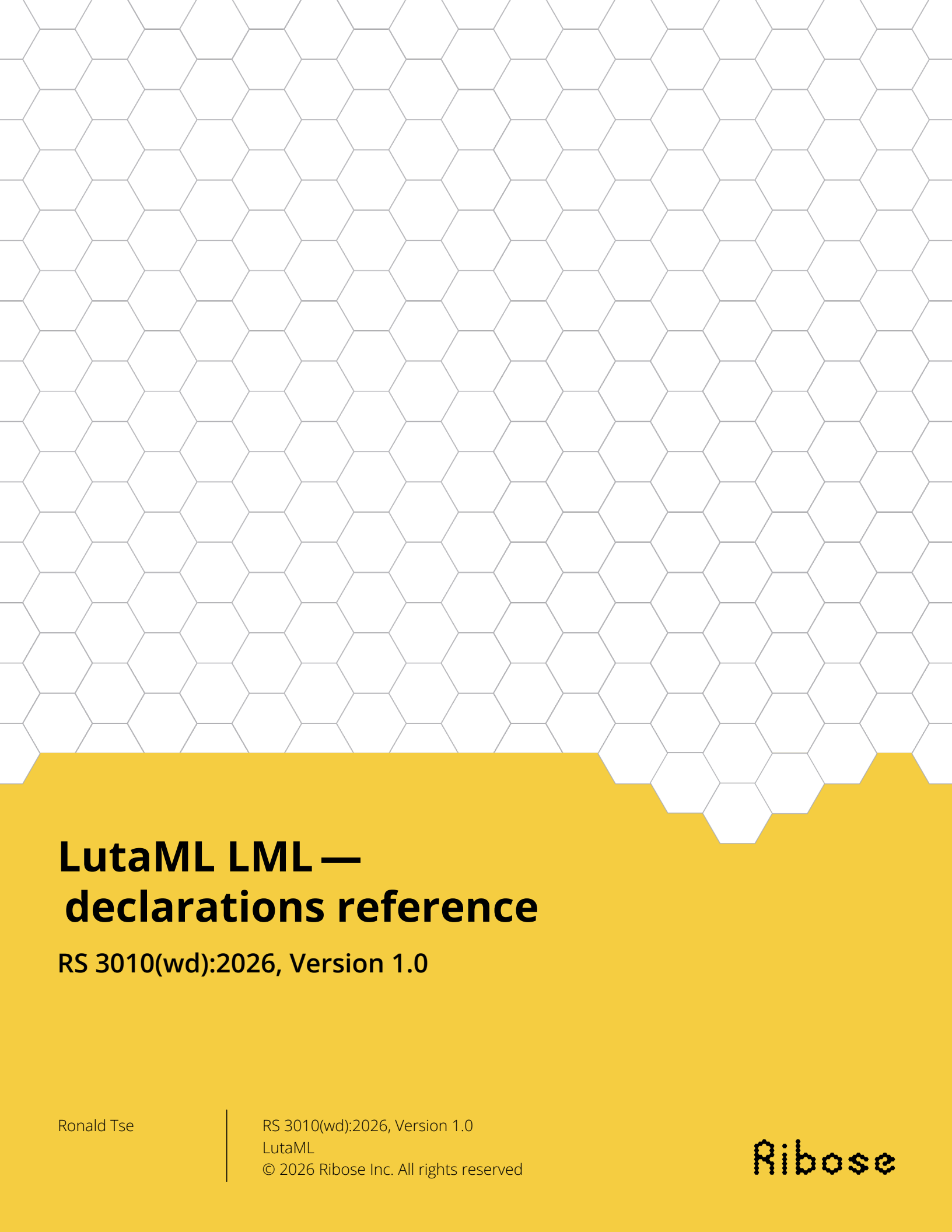




LutaML LML — declarations reference

RS 3010(wd):2026, Version 1.0

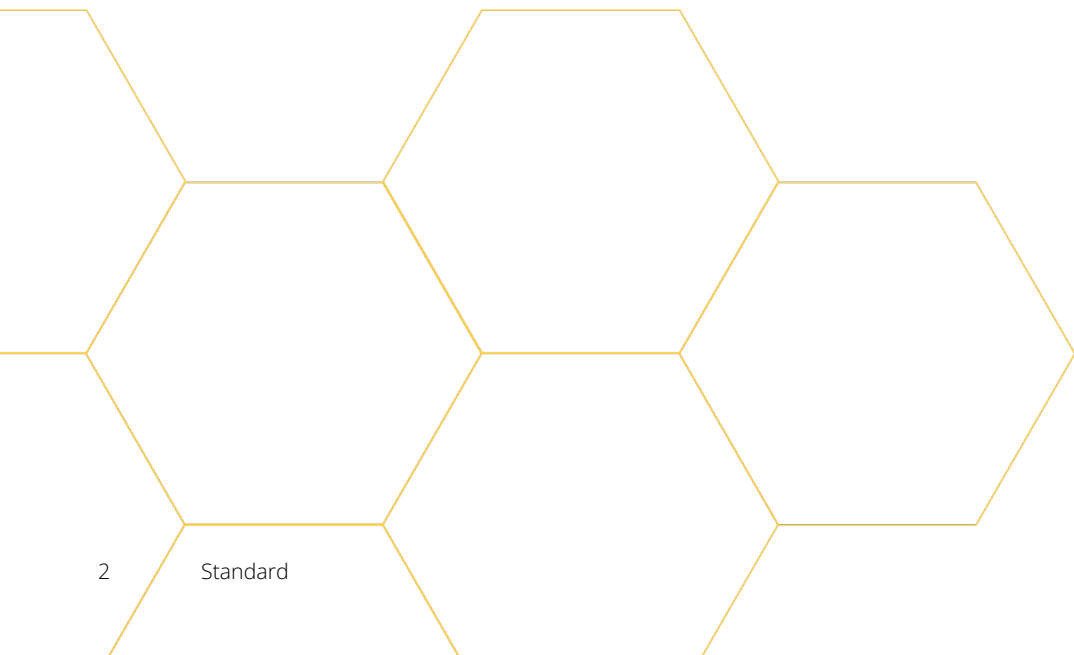
Ronald Tse

RS 3010(wd):2026, Version 1.0
LutaML
© 2026 Ribose Inc. All rights reserved

Ribose

Contents

- 1 Scope 4
- 2 Serialization mappings 4
- 3 Mapping blocks 5
- 4 Instance-type override (isa) 6
- 5 Attribute defaults and nil policy 6
- 6 Derived fields 7
- 7 Grammar-backed string formats 7
- 8 Enum member payloads 8
- 9 Table-driven enums 8
- 10 Operations 9



Warning for Drafts

This document is not a Ribose Standard. It is distributed for review and comment, and 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.

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from the address below.

Ribose Inc.

Suite 1, 8/F, New Henry House
10 Ice House Street
Central
Hong Kong

copyright@ribose.com

www.ribose.com

1 Scope

This document specifies the **declaration constructs** of the LutaML Model Language (LML) as implemented by `lutaml-lml 0.1.8`: serialization mapping blocks, instance-type overrides, attribute defaults and nil policy, derived fields, grammar-backed string formats, and enum member payloads including table-driven members. The base models-and-instances syntax is specified in RS 3001 and the accompanying `lutaml-lml` syntax guide.

2 Serialization mappings

Serialization mappings bind classes to their serialization wire forms. They are declared as **sibling constructs linked to their class by name** — mapping `<format> <class>` — or inside the class body, where the enclosing class is the implied target.

```
`lml models Ceramics { namespace CeramicsNs { uri "http://example.com/ceramics" prefix "cer" }
```

FIGURE 1

```
class Ceramic { attribute location, String attribute kiln,  
String attribute notes, String }
```

```
mapping xml Ceramic { element "ceramic" namespace CeramicsNs map_element "location"  
location map_attribute "kiln" kiln map_content notes }
```

```
mapping key_value Ceramic { map "full-name" location map_key to_instance key_name  
map_value as_attribute values }
```

XML mapping rules follow the host implementation's model: `element` declares the element name, `namespace <reference>` applies a context namespace, `map_element/map_attribute/map_content` bind wire names to model attributes, and `mixed_content` declares mixed content. The key-value family (`yaml`, `json`, `toml`, `key_value`) uses `map`, `map_key to_instance`, and `map_value to_instance/as_attribute`. Namespaces are declared once in the enclosing model's scope; mappings reference them by name. When the scope declares exactly one namespace it applies implicitly; several namespaces without an explicit reference is a compile-time error. Prefix-less namespaces act as the default namespace (`xmlns`).

3 Mapping blocks

A mapping block declares serialization wire names for a class's attributes — the serialized key differs from the attribute name.

FIGURE 2

```
class Widget {
  attribute name { type String cardinality 1 }
  mapping yaml { map "full-name", to: "name" }
}
```

TABLE 1

Part	Meaning
yaml	target format; one block per format (yaml, xml, json, toml, csv)
map "{wire}", to: "{attribute}"	a wire-name entry; {wire} is the serialized key, {attribute} the model attribute

The compiler emits the corresponding lutaml-model mapping block. Multiple mapping blocks may appear; each format is emitted once.

4 Instance-type override (isa)

The `isa` keyword overrides the type of an instance; a type attribute is ordinary data.

FIGURE 3

```
instance S158Checks {  
  isa ValidationChecks  
  type = Type::standard  
}
```

{type} the overridden (instance-of) type `type = ...::` an ordinary data attribute — identifier-kind fields named `type` remain usable as data
The bare instance `{Type} { ... }` header form is unchanged.

5 Attribute defaults and nil policy

Inside an attribute block:

FIGURE 4

```
attribute originator {  
  type String  
  cardinality 0..1  
  default "iso"  
  render_nil false  
}
```

default
"{literal}"

render_nil
{true|false}

literal-only default; applied only when the input is **absent** — an explicit nil stays explicit. A declared literal wins over the optional-marker nil implied by a `0..1` cardinality.

nil policy on serialization; the name mirrors the lutaml-model mapping option, and the compiler applies it to the key-value mapping rules.

6 Derived fields

FIGURE 5

```
derive urn: String
```

Declares the field’s existence and type; the field participates in mappings like any attribute. The **computation** is bound by the binder from the grammar artifact’s derive specifications — LML never defines computation.

7 Grammar-backed string formats

FIGURE 6

```
class IsoIdentifier {
  string_format :pubid_iso {
    artifact "pubid/iso"
    root "iso_identifier"
  }
}
```

Declares that instances of the class serialize to and from an identifier string parsed by a grammar artifact. The declaration is a thin reference: the artifact (a compiled grammar with typed bindings) is the single source of truth for capture maps and render specifications — LML never duplicates them.

TABLE 2

Part	Meaning
:pubid_iso	the registered format name (per flavor — flavors differ in grammar, not just data)
artifact "{ref}"	artifact reference, resolved through the compiler’s artifact_paths map or default_artifact
root "{entry}"	the grammar entry rule

Registration defines `from_{format}` on the compiled model, and the format is registered in the single lutaml-model `FormatRegistry`. A `models-block string_format` acts as the default for classes without their own.

8 Enum member payloads

FIGURE 7

```
enum Stage {
  member_fields { abbreviation String urn_segment String }

  draft10 { abbreviation = "WD" urn_segment = "wd" }
  published { abbreviation = "IS" urn_segment = "published" }
}
```

member_fields { . .. } the typed payload schema shared by all members {member}
{ {field} = {value} ... }:: inline payload lines, validated against the schema at compile time — an undeclared field is an error, a missing optional field is nil

9 Table-driven enums

FIGURE 8

```
enum Stage {
  member_fields { abbreviation String urn_segment String }
  from_table "stages" {
    artifact "pubid/iso"
  }
}
```

Members are expanded at compile time from a table in a grammar artifact's `tables` section — the same data source that preprocessing lookups and render conditions read. Rows are validated against `member_fields`: an unknown field is an error; a missing

optional field is nil; the member name comes from the row's name column. Expansion is compile-time, so materialization is deterministic and no data dependency reaches consumer runtimes.

from_table the table name within the artifact
"{name}" artifact "{ref}": the artifact reference; resolved through the compiler's artifact_paths map or default_artifact

10 Operations

A class may carry operations (methods) as structural metadata: a name, an optional symbol visibility, an optional parameter list, an optional return type with cardinality, and property modifiers. Operations carry no executable semantics; they exist so that UML models round-trip through LML without losing their behavioral surface (Enterprise Architect and XML carry methods on every class).

Syntax:

FIGURE 9

```
class {name} {  
  {visibility} {operation-name}({*parameter}): {return-type}  
  [{cardinality}] {*property}  
}
```

Where,

{visibility} optional; one of + (public), - (private), # (protected), ~ (package); default +
{operation-name} the name of the operation {*parameter}: zero or more parameters, comma-separated {return-type}: optional; the declared return type {cardinality}: optional; the cardinality of the returned collection {*property}: optional; comma-separated property modifiers (for example ordered, unique, query)

A parameter is:

FIGURE 10

```
[{direction} ]{parameter-name}[: {parameter-type}][ =  
{default}]
```

Where,

{direction}

optional; one of in, out, inout; default in

{parameter-name}

the name of the parameter {parameter-type}:: optional; the declared parameter type {default}:: optional; the parameter's default value

EXAMPLE

```
class Ceramic {  
  + getGlazeType(glazeCode: CharacterString = "123"): GlazeType[1..*]  
  {ordered, unique, query}  
  - simpleOp(): String  
  multi(a: String, b: Integer = 5, in c: Boolean)  
}
```