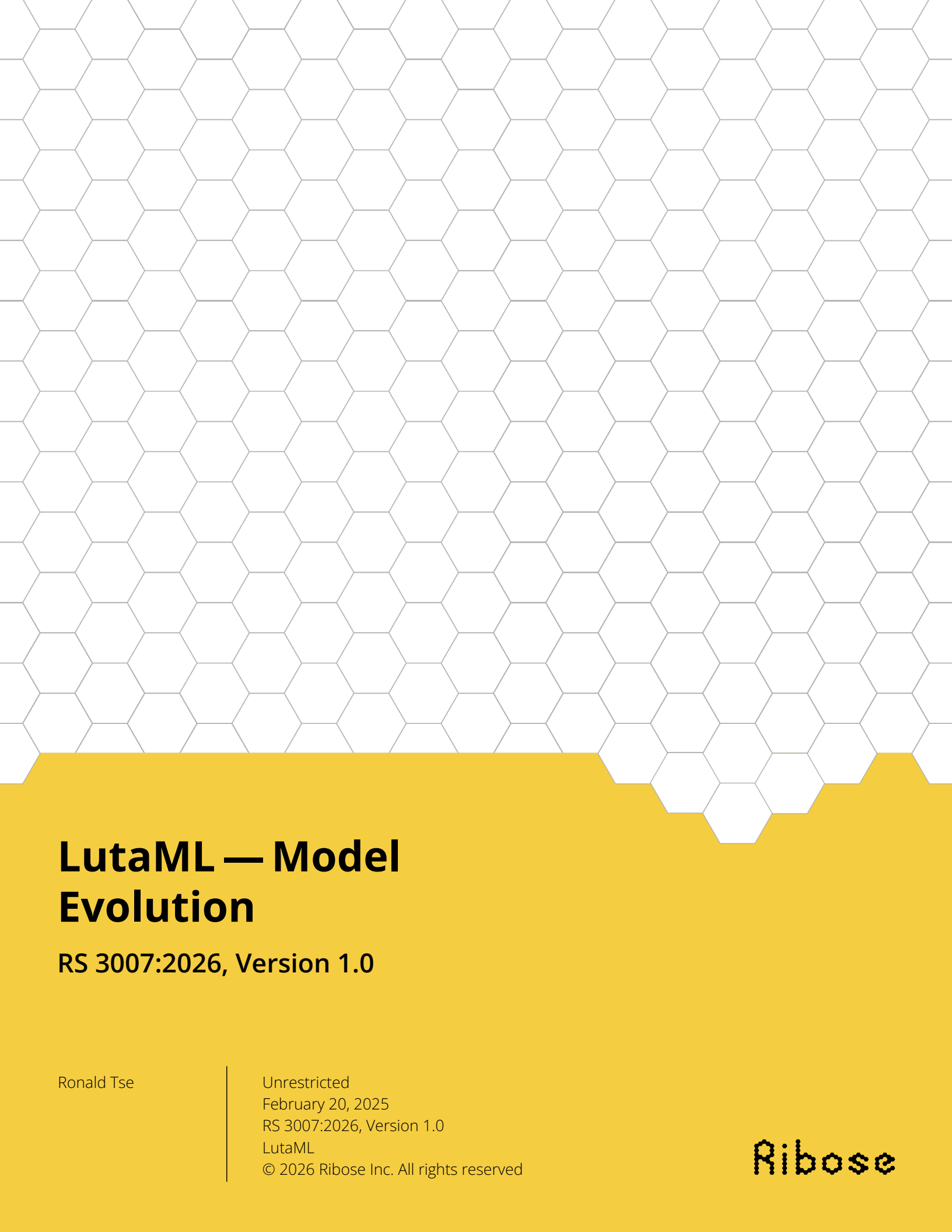




LutaML — Model Evolution

RS 3007:2026, Version 1.0

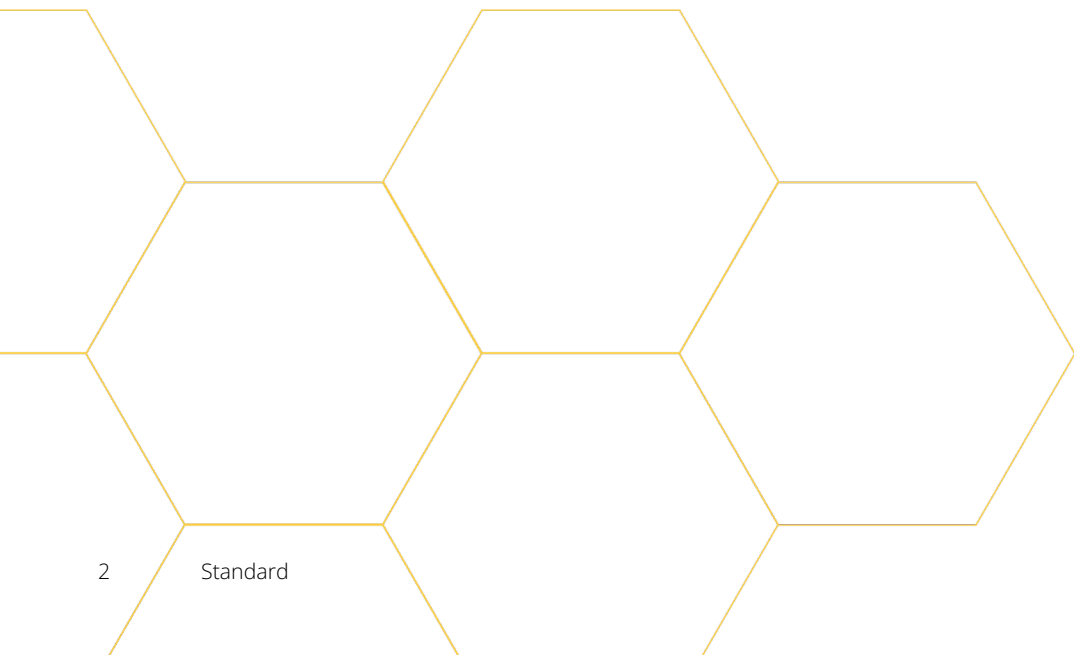
Ronald Tse

Unrestricted
February 20, 2025
RS 3007:2026, Version 1.0
LutaML
© 2026 Ribose Inc. All rights reserved

Ribose

Contents

- Introduction 4
- 1 Version surfaces 4
- 2 Change levels 5
 - 2.1 Patch (x.y.z → x.y.z+1) 5
 - 2.2 Minor (x.y.z → x.y.(z+1)) 5
 - 2.3 Major (x.y.z → x.(y+1).0) 5
- 3 Co-release discipline 5
- 4 Version declaration 6
- 5 Runtime model evolution (lutaml-model DSL) 6
 - 5.1 Version declaration 6
 - 5.2 Version detection 7
 - 5.3 Migration steps 7
 - 5.4 Scope and guarantees 8



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.

167-169 Great Portland Street
5th Floor
London
W1W 5PF
United Kingdom

copyright@ribose.com
www.ribose.com

Introduction

This document defines how LutaML models evolve over time. It covers the version surfaces, what each change level means for consumers, and the co-release discipline for keeping them aligned.

1 Version surfaces

A LutaML model has three independently versioned surfaces:

- Grammar artifact version: the compiled PARG grammar. Governed by `lutaml-lml`'s artifact version (semver; capture renames/removals = major, new alternatives = minor, literal fixes = patch).
- Model version: declared on the models header (`models <Name> version "x.y.z"`, per RS 3001 `lutaml-lml#16`). This is the **model schema** version — what downstream code generators (`pubid-ts`, `pubid-rs`) materialize from.
- Corpus version: the conformance corpus (fixtures) that verify model compatibility. Bumped in lockstep with model version.

2 Change levels

2.1 Patch ($x.y.z \rightarrow x.y.z+1$)

Safe changes that require no consumer action:

- grammar literal fixes (typo corrections, error message improvements)
- documentation updates
- corpus fixture additions that do not change model semantics

2.2 Minor ($x.y.z \rightarrow x.y.(z+1)$)

Additions that preserve backward compatibility:

- new `class`, `enum`, `data_type`, `primitive`, `package`, or `association` declarations in existing models
- new attributes on existing classes (consumers must tolerate unknown fields)
- new values members on existing enums
- new grammar alternatives (previously invalid LML text now parses)

2.3 Major ($x.y.z \rightarrow x.(y+1).0$)

Breaking changes that require consumer migration:

- removal or rename of a class, attribute, or enum value
- change of an attribute's type or cardinality
- grammar capture renames or removals (the artifact artifact changes shape)
- semantic changes to hydration behavior

3 Co-release discipline

The model schema version, corpus version, and grammar artifact version move **together** — one release, no mixed states.

A co-release consists of:

1. a models <Name> version "x.y.z" bump (if the model schema changed)
2. a conformance corpus update reflecting the change
3. a grammar artifact rebuild if the grammar changed
4. a semver-consistent lutaml-lml release

Conformance runners must verify all three versions agree before running the suite.

4 Version declaration

The version is declared in the LML source itself:

FIGURE 1

```
models PubidModels version "1.0.0" {  
  class IsoIdentifier { }  
}
```

This is recorded on Document#model_versions (name → version) and is the single source of truth for the model schema version.

5 Runtime model evolution (lutaml-model DSL)

The LML governance above governs the **artifact**. The lutaml-model Ruby DSL implements the **runtime** counterpart: models declare their version, families of versioned classes share a document detector, and explicit wire-level steps migrate documents between versions.

5.1 Version declaration

FIGURE 2

```
class MyModel < Lutaml::Model::Serializable
```

```

    version 1
    attribute :name, :string
end

class MyModelV2 < MyModel
  version 2
  attribute :created, Lutaml::Model::Type::DateTime
end

```

version stamps the model schema version. Subclasses inherit the nearest ancestor's version; `MyModelV2.model_version` reads it. A **family** is the ancestor chain of version-declared classes.

5.2 Version detection

A family head declares a detector that receives the parsed document and returns the version the document was written against:

FIGURE 3

```

class MyModelV2 < MyModel
  version 2
  detect_version do |doc|
    doc["schema_version"] ? doc["schema_version"].to_i : 1
  end
end

```

`from_json`, `from_yaml` and the other key-value entry points route to the detected version's class before parsing, raising `UnsupportedVersionError` when no family member declares the version. Detection is opt-in: without a detector, parsing is unchanged.

5.3 Migration steps

An evolution `from:` block declares wire-level steps on the target (newest) class. Steps apply in declaration order to a copy of the source document's wire hash; the evolved hash then parses as the target model.

TABLE 1

Step	Arguments	Semantics
wire_rename	from_key, to:	Accept a former wire name for the current attribute; the current name wins when both appear. Not re-emitted on serialization.

Step	Arguments	Semantics
wire_move	from_key, to:	Relocate a former top-level key into a nested position (dot path or segment array); intermediates are created.
wire_merge	from_keys, to:, with:	Combine former keys into one. with: receives the source values in declared order; without it, the sources must all be hashes and are merged in order.
wire_drop	*keys	Retire former keys explicitly.

FIGURE 4

```
class MyModelV2 < MyModel
  version 2

  evolution from: MyModel do
    wire_rename "author", to: "creator"
    wire_merge %[first_name last_name], to: "full_name",
              with: ->(first, last) { [first,
last].compact.join(" ") }
    wire_drop "deprecated_field"
  end
end

doc = MyModelV2.evolve(old_document_or_instance, from:
MyModel)
```

evolve walks every declared family link between the source version and the target (chains compose transitively), never mutates the caller's hash, and parses the evolved wire through the hash format. Migration is explicit; detection alone **accommodates** (routes), evolve **migrates**.

5.4 Scope and guarantees

- Key-value formats (JSON, YAML, TOML, hash). XML documents keep their structure through the parse complement; XML structural moves are not covered by the wire steps.
- The source hash is never mutated.
- Composed chains are the verified path: upcast then serialize yields a current-version document, and re-parse is idempotent.