LutaML — Path

RS 3002:2025, Version 1.0

Ronald Tse

Unrestricted
February 20, 2025
RS 3002:2025, Version 1.0
LutaML

Ribose

Contents

Ribose Inc.

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

copyright@ribose.com
www.ribose.com

Introduction

"LutaML Path" is a query language for LutaML Models. It allows for referencing and locating elements within information models defined using the LutaML Model system.

There are two types of LutaML Path syntaxes that share a common syntax:

- Model definition query (e.g. LutaML Model and UML definitions) (see Clause 6)
- Instance data query (information model instances) (see Clause 7)

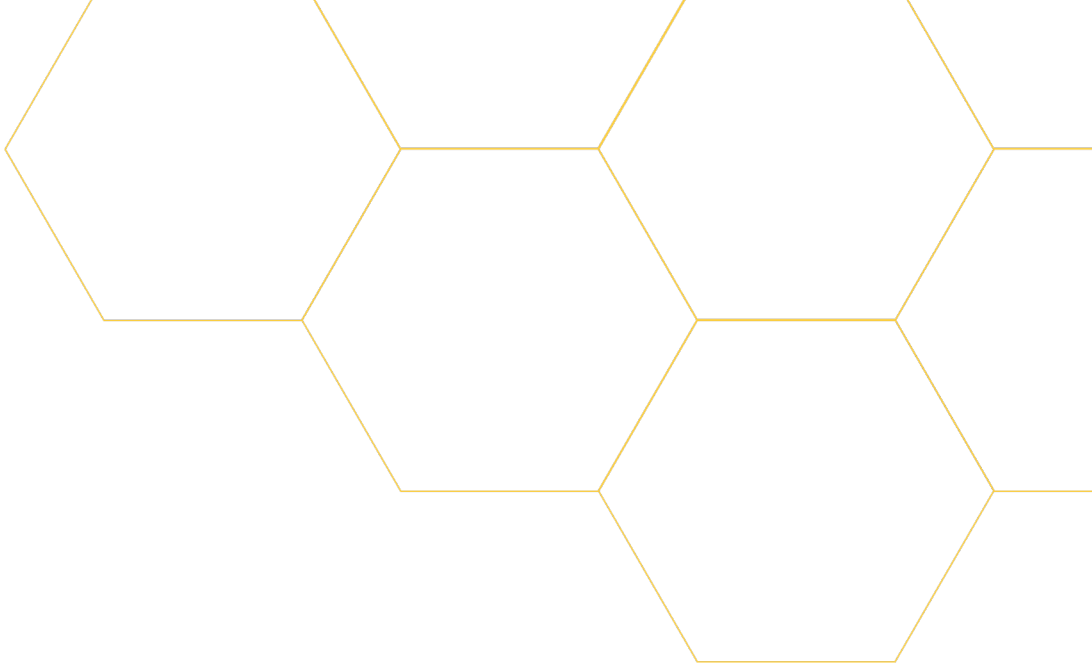
1. Scope

This document specifies the path expression capabilities in LutaML Model, which enable:

- Navigation through model hierarchies
- Querying model instances and definitions
- Selection of model elements and attributes using path expressions
- Filtering model instances based on predicates

2. Normative references

There are no normative references in this document.



3. Terms and definitions

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

3.1. path expression PREFERRED

query pattern that describes a traversal through a model hierarchy to locate specific elements or attributes

3.2. model navigation PREFERRED

process of traversing through model relationships using path expressions

3.3. model query PREFERRED

expression that selects model elements based on specific criteria

3.4. path segment PREFERRED

individual component within a path expression that represents a single step in model navigation

3.5. predicate PREFERRED

filtering expression used to select specific model instances based on criteria

3.6. path root PREFERRED

starting point for path navigation within a model hierarchy

4. Principles of path expressions

4.1. General

5. Common syntax

5.1. General

The LutaML path syntax is designed to be concise and expressive. It supports both absolute and relative paths, wildcards, and pattern matching.

The path syntax is inspired by the Object Constraint Language (OCL) and extends beyond it to provide a mechanism for uniquely identifying model elements within the LutaML Model hierarchy.

LutaML Path fully supports Unicode characters in package and element names.

5.2. Expression

5.2.1. General

An expression can be a single element, a hierarchy, or a pattern.

The path syntax supports the following features:

- Wildcard matching
- Pattern matching
- Path segments with a regular expression-like syntax

In LutaML Models, models are typically referred using TitleCase and attributes in snake_case.

5.2.2. Wildcard matching

Wildcard characters are used to match multiple elements.

The path syntax supports the following wildcards:

Wildcard match character *

matches any sequence of characters. When used as double asterisks **, it matches any nested depth of the hierarchy.

EXAMPLE 1

```
# Wildcard matching
```

```
## Matches any class starting with "Base"
Base*

## Matches any class ending with "Model"
*Model

## Matches any class starting with "Base" in the
"Core" package
Core::Base*

## Matches any class ending with "Model" in the
"Core" package
Core::*Model

## Matches any class in the "Core" package
Core::*
```

Any match character ?

matches any single character

EXAMPLE 2

```
# Single character matching
## Matches any class starting with "Bas" and one
more character, such as "Base"
Bas?

## Matches any class ending with "Model" and one
more character, such as "BaseModel"
*Model?
```

5.2.3. Pattern matching

The path syntax supports several kinds of patterns. These patterns are used to match elements based on specific criteria. The patterns are similar to regular expressions but are more concise and easier to read.

The path syntax is case-sensitive and follows the source model's character cases.

Set match expression `[` character expression `]`

The delimiters of `[` and `]` are used to define a set of characters. The set can contain any characters, including Unicode characters.

Character match expression `[abc]`

matches any character in the set.

Negative match expression `[!…]`

matches any character not in the set.

Range match expression `[a-z]`

The `-` character is used to define a range of characters. It matches any character in the range according to Unicode code points.

Alternatives match expression `{expression1,expression2}` matches any of the comma-separated patterns

EXAMPLE

```
# Pattern matching
## Matches any class ending with "ase", such as "Base", "Case" but not
"Vase"
[BC]ase

## Matches any class starting with "Base" or "Case"
{Base,Case}

## Matches any class starting with "Base" or "Case" and ending with
"Radius"
{Base,Case}*Radius

# Range matching
## Matches any class of "Vase" and "vase"
[Vv]ase

# Negative matching
## Matches any class ending with "ase" but not "Vase" and "Case"
[!CV]ase

# Unicode matching
## Matches any class starting with "建物"
建物*

## Matches exactly "ドア" or "窓"
{ドア,窓}
```

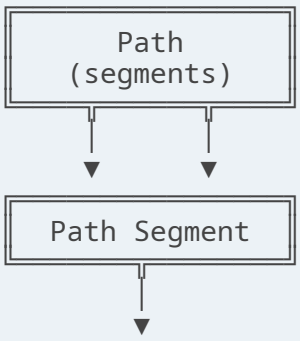
5.3. Path

5.3.1. General

A path is a sequence of path segments separated by hierarchy separators.

A path segment is a single element or a pattern that matches a single element.

FIGURE 1



5.3.2. Hierarchy separators

The hierarchy separators are used to separate path segments within a path:

`::` for model definitions

`.` for attributes and instance data.

The separators can be escaped with a backslash inside an expression. Single colons (`:`) are not used as separators and behave as part of the segment.

EXAMPLE 1

```
# Referencing the "Rectangle::Shape" object
::Rectangle\::Shape

# Accessing the "width.length" attribute of the "Rectangle" object
::Rectangle.width\.length
```

The leading hierarchy separator, indicating absolute paths, cannot be escaped.

EXAMPLE 2

```
\::Rectangle::Shape # This is invalid

\.width.length # This is invalid
```

5.3.3. Absolute and relative paths

The path syntax supports both absolute and relative paths:

- Absolute paths start with `::` and begin at the model root
- Relative paths start without `::` and are resolved from the current context

The target element type may be a class, property, operation, or any other model element.

The separator can be escaped with a backslash: `\::`, if the package name contains a double colon.

EXAMPLE

```
# Absolute path
## Locates a model called "Rectangle" in the "Shapes" package at root
::Shapes::Rectangle

## Locates an attribute called "width" in the "Rectangle" class at root
::Shapes::Rectangle.width

## Locates a model called "図形" in the "Geometry" package at root
::Geometry::図形

## Locates an attribute called "高" in the "図形" class
```



```
::Geometry::図形.高

# Relative path
## Locates a model called "Rectangle" in the current package
Shapes::Rectangle

## Locates an attribute called "width" in the current class
Shapes::Rectangle.width

## Locates a model called "図形" under the "Geometry" model in the
current model
Geometry::図形

## Locates an attribute called "高" in the "図形" model
Geometry::図形.高
```

6. LutaML path for model definitions

6.1. Overview

The LutaML path for model definition query syntax (“LutaML model path”) is used to reference elements within model definitions. These paths are used to locate classes, properties, operations, and other model elements within the model hierarchy.

While the LutaML path syntax is designed to work with LutaML Models, it can also be used with UML models.

It implements a path notation similar to the Object Constraint Language (OCL) to locate UML model elements across package hierarchies.

The UML element path specification extends the OCL 2.4 specification to provide a mechanism for uniquely identifying model elements (classes, interfaces, enumerations, etc.) within the UML package hierarchy. It provides both relative and absolute path references.

6.2. Hierarchical paths

An element path can be specified in these forms:

- Single element: `ElementName`
- Relative path: `Package1::Package2::ElementName`
- Absolute path: `::Package1::Package2::ElementName`

The absolute path variant starts with `::` to indicate the path begins at the model root.

FIGURE 2

```
# Japanese package and class names
建物::窓::ガラス
::建築モデル::建物::窓

# Mixed language names
building::窓::Window
geometry::図形::円

# Patterns with Unicode
建物::部品*
*部::Base*
```

6.3. Path segment patterns

Path segment wildcards can be used to match package hierarchy.

- Single segment: `Package1::*::Element` matches Element in any subpackage of Package1
- Multiple segments: `Package1::**::Element` matches Element in Package1 or any nested depth

FIGURE 3

```
# Wildcard matching
Package1::*::Element
Package1::**::Element
```

6.4. Resolution rules

- Single element name or pattern matches in any package
- Relative paths are resolved from current context
- Absolute paths are resolved from model root
- Path segments must match patterns exactly
- Empty segments are invalid
- Multiple matches are allowed with wildcards/patterns

- Without wildcards/patterns, first match is used for multiple matches

7. LutaML path for instance data

7.1. Overview

The LutaML path for instance data query syntax ("LutaML instance path") is used to navigate and query data within model instances. These paths are used to access attributes, filter data, and navigate complex structures within model instances.

Model instances are instances of data that conform to a model definition.

The LutaML model data syntax uses dot notation and filters to navigate and query data within model instances.

Model instances can be queried using path expressions to access attributes, filter data, and navigate complex structures.

Model are typically referred using TitleCase and attributes in snake_case.

An instance path expression when resolved provides two types of return values:

- A single element (e.g. attribute value)
- A collection of elements (e.g. filtered data)

7.2. Hierarchical paths

An instance path can be specified in these forms:

- Collection of model instances: `{model_path}`
- Attribute: `{class_path}.{attribute}`

EXAMPLE

```
# Simple attribute access
obj.title
obj.edition.number
```

7.3. Wildcard matching

Wildcard symbols can be used to match characters in the path.

These include:

- path segments
- attribute names
- attribute values in conditions

EXAMPLE

```
# Wildcard matching path segments
obj.*.docidentifier[type!='ISBN']

# Wildcard matching attribute names
obj.contributor.*[type='author']

# Wildcard in path segments
## Matches all contributors with the name 'ISO'
## e.g. matches obj.contributor.publisher.organization.name('ISO')
obj.contributor.**[name='ISO']

# Wildcard usage in conditions
## Matches all authors from standards organizations
obj.contributor[role.type='author' && organization.type='stand*']
```

7.4. Filtering

7.4.1. General

Filters can be applied to model instances to query data based on specific criteria. Filters are enclosed in square brackets `[]` and can contain conditions.

A filter “condition” is a comparison expression that evaluates to a boolean value.

Syntax:

FIGURE 4

```
attribute[condition]
```

The result of a filter is a collection of elements that match the condition.

7.4.2. Conditions

Conditions are used to filter data based on specific criteria.

A filter “condition” is a comparison expression that evaluates to a boolean value.

Conditions can include:

- Value comparison: `property='value'`
- Multiple conditions: `condition1 && condition2`

- List membership: `property in ('value1', 'value2')`
- Logical operators: `condition1 && (condition2 || condition3)`
- Negation: `!condition`
- Existence: `exists`

Logical operators that can conjoin conditions:

- Comparison: `=`, `!=`, `>`, `<`, `>=`, `<=`
- Logical: `&&`, `||`, `!`
- List membership: `in`

EXAMPLE

```
# Condition for attribute value
obj.contributor[role.type='publisher']
# => Returns all contributors with role type 'publisher'

# Multiple conditions
obj.contributor[role.type='author' && organization.type='standards']
# => Returns all authors from standards organizations

# List membership
obj.docidentifier[type in ('ISBN','ISSN','DOI')]
# => Returns all document identifiers of type ISBN, ISSN, or DOI

# Existence
obj.contributor[exists]
# => Returns all contributors

# Negation
obj.docidentifier[type!='ISBN']
# => Returns all document identifiers not of type ISBN

# Complex conditions
obj.contributor[role.type='author' && organization.type='standards']
# => Returns all authors from standards organizations
```

7.4.3. Applying filters

Filters are applied to model instances to query data based on specific criteria.

Filters are enclosed in square brackets `[]` and can contain conditions.

EXAMPLE 1

```
# Filter by date type
## Steps:
## 1. The first portion of the path navigates to the date element
## 2. The filter condition is applied to the type attribute
obj.date[type='updated']

# Filter by document identifier type
## Steps:
```

```
## 1. The first portion of the path navigates to the docidentifier
element
## 2. The filter condition is applied to the type attribute
obj.docidentifier[type!='ISBN']
```

Filters can be chained to navigate complex structures and query data based on specific criteria.

EXAMPLE 2

```
# Nested navigation with filtering
## Filter by role type
## Steps:
## 1. The first portion of the path navigates to the contributor
element
## 2. The filter condition is applied to the role type attribute
## 3. The last portion of the path navigates to the organization name
attribute
obj.contributor[role.type='publisher'].organization.name

# Filter by multiple conditions
## Filter by role type and organization type
## Steps:
## 1. The first portion of the path navigates to the contributor
element
## 2. The filter conditions are applied to the role type and
organization type attributes
## 3. The last portion of the path navigates to the organization name
attribute
obj.contributor[role.type='author' && organization.type='standards'].
organization.name

# Hierarchical filtering (nested conditions)
## Filter by role type and organization type
## Steps:
## 1. The first portion of the path navigates to the contributor
element
## 2. The filter conditions are applied to the role type and
organization type attributes
## 3. The path navigates to the organization name attribute
## 4. The filter condition is applied to the organization name
attribute
obj.contributor[role.type='author' && organization.type='standards'].
organization[organization.name in ('ISO','IEC')].name
```

7.4.4. Path expressions

Path expressions are used to navigate through model attributes and filter data based on specific criteria.

The path expressions are evaluated against the current model context.

EXAMPLE

```
# Navigate through model attributes
contributor.role.type
organization.name

# Filter by attribute values
```

```
contributor[role.type = 'publisher'].organization.name
date[type = 'updated']
docidentifier[type != 'ISBN']

# Multiple conditions
contributor[role.type = 'author' and organization.type = 'standards']
```

7.5. Resolution rules

- Single element name or pattern matches in any package
- Relative paths are resolved from current context
- Absolute paths are resolved from model root
- Path segments must match patterns exactly
- Empty segments are invalid
- Multiple matches are allowed with wildcards/patterns
- Without wildcards/patterns, first match is used for multiple matches

8. Ruby API

8.1. Introduction

The LutaML Path gem provides a simple API for parsing and matching paths.

> *WARNING*
>
> *It currently only supports the model definition path syntax.*

8.2. How to install

FIGURE 5

```
gem install lutaml-path
```

Or add this line to your application's Gemfile:

FIGURE 6

```
gem 'lutaml-path'
```

8.3. Basic usage

The LutaML Path gem provides a simple API for parsing and matching paths.

The path syntax follows UML namespace conventions using `::` as a separator:

FIGURE 7

```
require 'lutaml/path'

# Model definition path
## Simple element reference
path = Lutaml::Path.parse("Package::Class")

## Absolute path (starts from root namespace)
path = Lutaml::Path.parse("::Root::Package::Class")

## Path with wildcards
path = Lutaml::Path.parse("Package::*::BaseClass*")
```

8.4. Working with patterns

EXAMPLE

```
# Model location matching
# Match any class starting with "Base"
path = Lutaml::Path.parse("Base*")

# Match specific character patterns
path = Lutaml::Path.parse("Package::[A-Z]*::Interface")

# Match multiple alternatives
```

```
path = Lutaml::Path.parse("model::{Abstract,Base}Class")
```

8.5. How to match paths

The parsed path can be used to match against actual element paths:

FIGURE 8

```
path = Lutaml::Path.parse("model::*::BaseClass")

path.match?(["model", "core", "BaseClass"])     # => true
path.match?(["model", "BaseClass"])             # => false
path.match?(["other", "core", "BaseClass"])     # => false
```

9. Understanding absolute and relative paths

- Absolute paths (starting with `::`) must match the entire element path
- Relative paths can match elements at any depth

FIGURE 9

```
absolute = Lutaml::Path.parse("::model::Class")
relative = Lutaml::Path.parse("model::Class")

absolute.match?(["model", "Class"])           # => true
absolute.match?(["root", "model", "Class"])   # => false

relative.match?(["model", "Class"])           # => true
```

```
relative.match?(["root", "model", "Class"])   # => true
```

10. Matching paths with escaped colons

When matching paths with escaped colons, the escaped sequences are treated as part of the segment name:

FIGURE 10

```
path = Lutaml::Path.parse("model::std\\::string")

path.match?(["model", "std::string"])     # => true
path.match?(["model", "std", "string"])   # => false
```

10.1. Examples of UML element references

FIGURE 11

```
# Reference a class in a package
"model::shapes::Rectangle"

# Reference an operation on a class
"model::shapes::Rectangle::area"

# Reference a property in a nested class
"model::university::Student::Address::street"

# Find all classes implementing an interface
"model::*::IShape"

# Match any stereotype application
"model::profiles::UMLProfile::*Stereotype"
```

These paths can be used to locate elements across UML model hierarchies, making it easier to reference and work with model elements programmatically.

11. License

Copyright Ribose.

The `lutaml-path` gem is available as open source under the terms of the MIT License.