LutaML — Transforms

RS 3003:2025, Version 1.0

Ronald Tse

Unrestricted
February 20, 2025
RS 3003:2025, 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

1. Scope

This document specifies the transformation capabilities in LutaML Model, which enable mapping between different model representations while maintaining data integrity and structure.

It defines:

- Value transformation interfaces
- Model transformation patterns
- Nested transformation rules
- Bidirectional transformation capabilities

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. transform PREFERRED

operation that defines mapping rules between different model representations

3.2. value transform PREFERRED

transform (3.1) that operates on individual attribute values

3.3. model transform PREFERRED

transform (3.1) that operates on entire model structures

3.4. source model PREFERRED

model instance containing the original data to be transformed

3.5. target model PREFERRED

model instance that will receive the transformed data

4. Principles in transformations

4.1. General

A LutaML model defines the internal information organization structure of an information model.

In order to allow external users to interact with the model, it is necessary to provide a way to transform the defined model into other models:

- Serialization models that represent the model in a specific serialization format. e.g. JSON, XML, YAML.
- Other LutaML models that represent information differently.
- Information models in another modelling language that represent information differently.

NOTEThe “transform” referred here is a mapping between a source LutaML model and a target LutaML model.

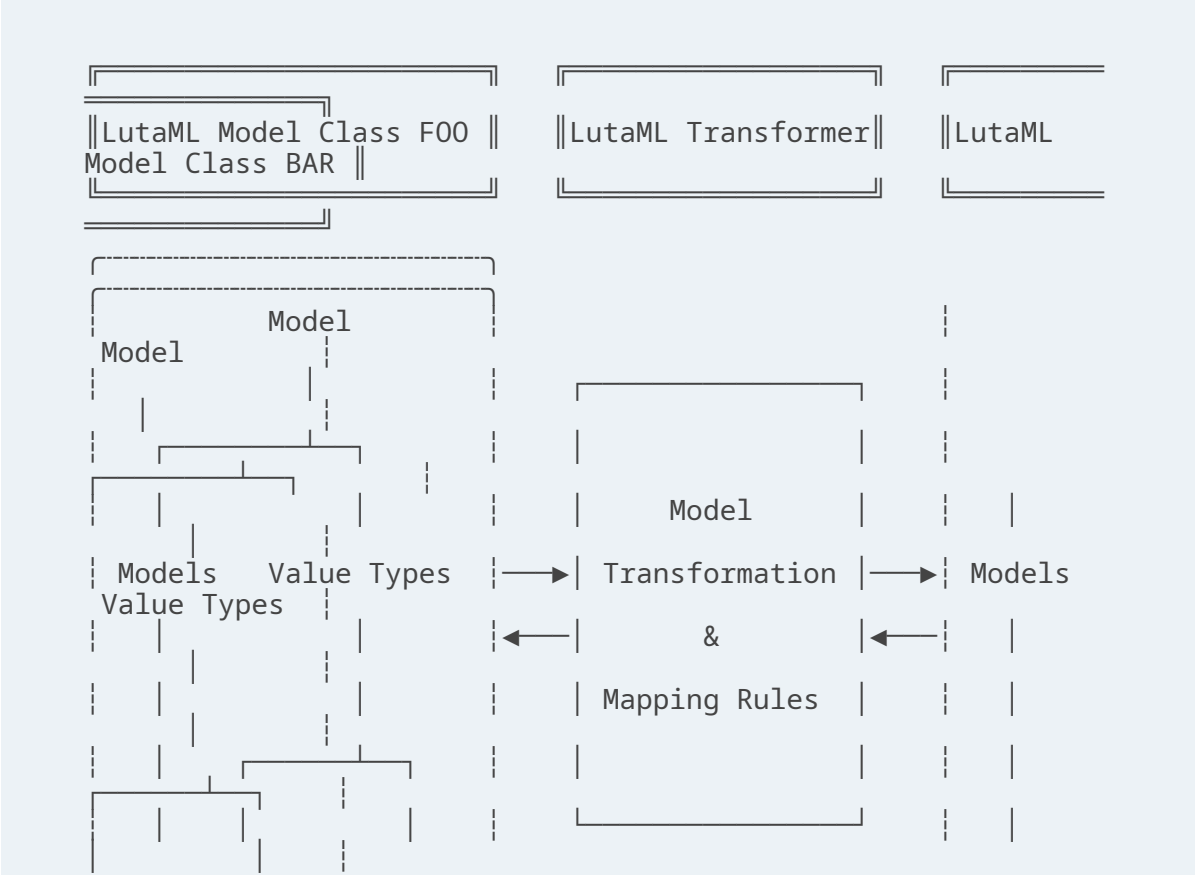
4.2. Architecture

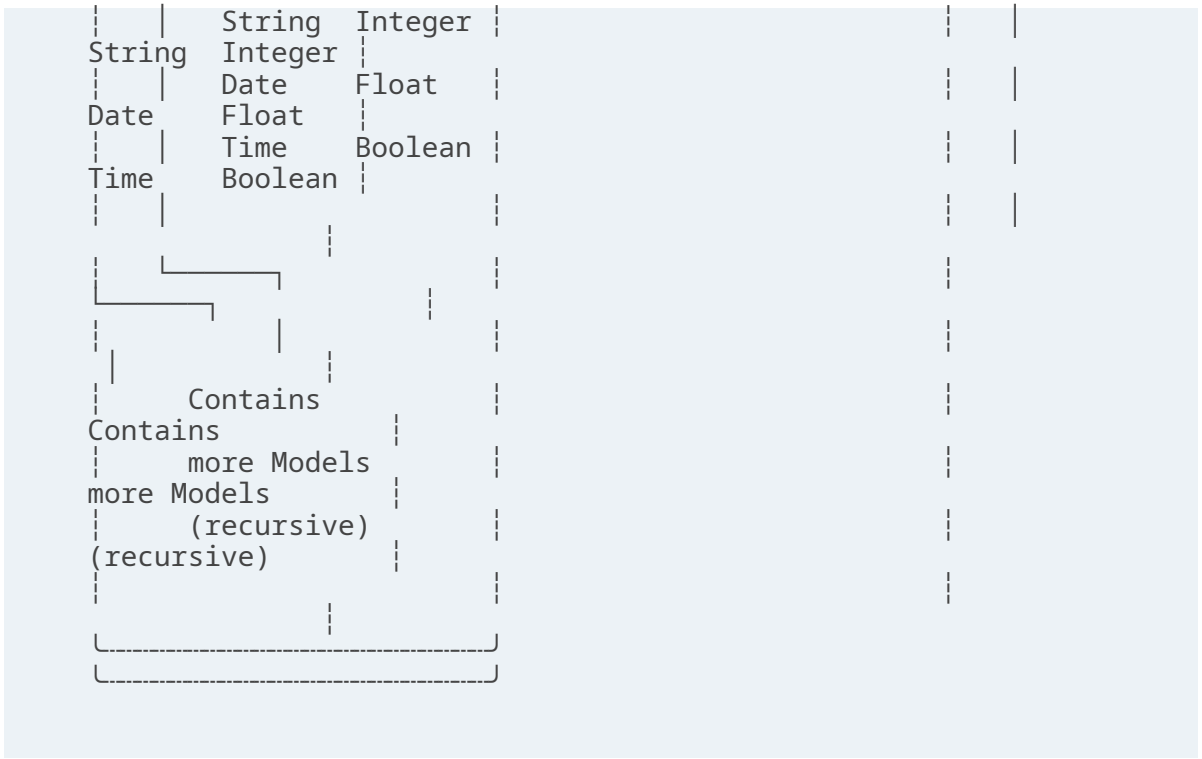
In LutaML, a transform is a first-order object that defines:

- source model
- target model
- mapping between the two models
- any value transforms or processing logic

A transform can be unidirectional or bidirectional.

FIGURE 1: Model transformation of a LutaML Model to another LutaML Model





4.3. Model and value transforms

A value transform is a transformation that operates on individual attribute values, converting from one type or format to another.

Common cases include:

- Converting between different data types (e.g., string to date)
- Splitting or combining values

EXAMPLE 1: Transforming:

- source: a string date “2025-03-15”
- target: a LutaML Value type “Date”.

A model transform operates on entire model structures, allowing for comprehensive mapping of various attributes and relations within the model.

Common cases include:

- Converting between different model representations
- Handling nested object transformations
- Processing collections

EXAMPLE 2: Transforming:

- source: a model with attributes `title`, `author`, `publication_date`
- target: a model with attributes `name`, `creator`, `date`, which correspond to the source model attributes.

A source or target model here could also be a value type. For instance, extracting structured information (a model) from a string (a value type).

EXAMPLE 3: Transforming:

- source: a string set of names `"Gottlieb;von;Peter;Arnold"`
- target: a structured name model that contains given names (`["Peter", "Arnold"]`), conjunction (`"von"`) and last name (`"Gottlieb"`).

4.4. Directionality

A transform can operate in one or both directions.

A unidirectional transform is a transformation that operates in one direction only, and cannot be reversed.

A bidirectional transform is a transformation that operates in both directions, allowing for reversible transformations.

4.5. Common use cases

4.5.1. Between model representations

This use case is demonstrated by the Plurimath gem's handling of MathML conversion (plurimath/plurimath#304).

In Plurimath:

- Plurimath maintains an internal math model (the `Formula` class) for mathematical semantics
- The mml gem models the MathML language specification, and provides MathML XML serialization

When a MathML XML document is loaded, it is transformed into a `Plurimath::Formula` model instance in these steps:

1. MathML XML ⇒ The `Mml` LutaML model within the `mml` gem
2. The `Mml` LutaML model ⇒ The `Plurimath::Formula` LutaML model

This architecture means Plurimath::Math does not directly handle serialization, but can transform into the `Mml` model when serialization is needed.

When a MathML XML document is saved, the process is reversed:

1. The `Plurimath::Formula` LutaML model ⇒ The `Mml` LutaML model
2. The `Mml` LutaML model within the `mml` gem ⇒ MathML XML

4.5.2. Between serialization models

A common requirement is the need to handle multiple serialization formats for the same data model.

The `modspec gem` provides a LutaML model for the OGC Modular Specification (ModSpec) requirements model, and supports XML and YAML serialization outputs (the "Native ModSpec XML/YAML format").

The `mn-requirements gem` needs to provide a Metanorma Requirements XML serialization format for the identical ModSpec model (the "Metanorma Requirements XML format").

In encoding Metanorma Requirements in ModSpec, the user supplies Native ModSpec YAML which is meant to be transformed into Metanorma Requirements XML.

The transformation process is:

1. ModSpec YAML ⇒ ModSpec LutaML model
2. ModSpec LutaML model ⇒ Metanorma Requirements LutaML model
3. Metanorma Requirements LutaML model ⇒ Metanorma Requirements XML (the "Native Metanorma Requirements XML format")

In reverse, when the user wants to extract ModSpec YAML from Metanorma Requirements XML:

1. Metanorma Requirements XML ⇒ Metanorma Requirements LutaML model
2. Metanorma Requirements LutaML model ⇒ ModSpec LutaML model
3. ModSpec LutaML model ⇒ ModSpec YAML

4.5.3. Between versioned models

A common use case involves transforming between different versions of the same model as it evolves over time.

The Relaton LutaML model demonstrates this pattern:

- Model version information is stored in the `schema-version` attribute of serialized formats of Relaton.
- When an older version of the Relaton serialization is parsed, it is first interpreted by the appropriate version of the Relaton serialization LutaML model, and then transformed into the latest version of the Relaton data model.
- A version-to-version transform handles model changes

EXAMPLE: Relaton XML/YAML version attributes:

```
<bibdata type="standard" schema-version="1.2.9">
  ...
  <ext schema-version="1.0.3">
    ...
  </ext>
</bibdata>

id: ISO1231994
type: standard
schema_version: 1.2.9
...
ext:
  schema_version: 1.0.3
  ...
```

For transformations across multiple versions, transformations must be applied sequentially in historical order (e.g., “1.0.1” → “1.0.2” → “1.0.3”).

4.6. Model transformation patterns

4.6.1. General

There are several common model transformation patterns:

- Generic-to-specific transformation
- Specific-to-generic transformation
- Many-to-many transformation

4.6.2. Re-mapping attributes

When transforming between models, it is common to re-map attributes between different models without changing value types.

EXAMPLE: Converting a “title” attribute in a “Publication” model to a “name” attribute in a “CatalogEntry” model.

4.6.3. Generic-to-specific transformation

Transforms a general model into a more specific one.

EXAMPLE: Converting a general “car” model into a specialized “taxi” model.

4.6.4. Specific-to-generic transformation

Transforms a specific model into a more general one.

EXAMPLE: Converting a specialized “taxi” model into a general “car” model.

4.6.5. Many-to-many transformation

Transforms a model that can be represented in multiple ways.

EXAMPLE: An amphibious vehicle model that can transform into both “car” and “boat” models.

4.7. Directionality

4.7.1. General

Transforms can be configured to operate in one or both directions.

The reversibility of a transform depends on two things:

- whether any mapping rules are one-way transforms
- whether the `reverse_transform` do block is defined

4.7.2. Simple transforms (bidirectional)

When a transform is defined with only a `transform` do block that contains bidirectional mapping rules, the transform is bidirectional.

FIGURE 2

```
class SimpleBidirectionalTransform < Lutaml::Model::Transform
  source_model :source_model
  target_model :target_model

  transform do
    # mapping without value transform logic
  end
end
```

4.7.3. Single direction transform

When a transform is defined with only a `transform` do block that contains unidirectional mapping rules, the transform is unidirectional.

FIGURE 3

```
class UnidirectionalTransform < Lutaml::Model::Transform
  source_model :source_model
  target_model :target_model

  transform do
```

```
      # mapping with value transform logic
    end
  end
```

4.7.4. Explicit bidirectional transform

When a transform is defined with a `transform` do block that contains unidirectional mapping rules, but also a `reverse_transform` do block that contains reverse unidirectional mapping rules, the transform is bidirectional.

FIGURE 4

```
class ExplicitBidirectionalTransform < Lutaml::Model:
:Transform
  source_model :source_model
  target_model :target_model

  transform do
    # mapping with value transform logic
  end

  reverse_transform do
    # mapping with value transform logic
  end
end
```

5. Value transforms

5.1. General

Value transforms operate on individual attribute values, converting from one type or format to another.

5.2. Structure

A value transform:

- Inherits from `Lutaml::Value::Transform`
- Defines source and target value types
- Implements the `transform` method

- Implement `reverse_transform` method if bidirectional

Syntax:

FIGURE 5

```
class ValueTransformClass < Lutaml::Value::Transform
  source_value :source_type <1>
  target_value :target_type <2>

  transform do |source_value|
    # transformation logic
  end

  reverse_transform do |target_value|
    # reverse transformation logic
  end
end
```

source_value	Specifies the source value type. This can be a primitive type or a class that inherits from `Lutaml::Value`.
target_value	Specifies the target value type. This can be a primitive type or a class that inherits from `Lutaml::Value`.
transform	Defines the transformation logic.
reverse_transform	Defines the reverse transformation logic for bidirectional transforms.

EXAMPLE

```
# Transforms a string into a Date model
class DateFormatTransform < Lutaml::Value::Transform
  source_value :string
  target_value :date_with_time

  transform do |source_value|
    Date.parse(source_value)
  end

  reverse_transform do |target_value|
    target_target_value.strftime('%Y-%m-%d')
  end
end
```

Given:

```
DateFormatTransform.transform('2021-01-01')
# => #<Date: 2021-01-01 ((2459216j,0s,0n),+0s,2299161j)>

DateFormatTransform.reverse_transform(Date.new(2021, 1, 1))
```

```
# => "2021-01-01"
```

6. Model transforms

6.1. General

Model transforms operate on entire model structures, mapping attributes between different model representations.

6.2. Base requirements

A model transform:

- Inherits from `Lutaml::Model::Transform`
- Specifies source and target models
- Defines mapping rules within a transform block
- Declares directionality

Syntax:

FIGURE 6

```
class TransformClass < Lutaml::Model::Transform
  source_model :source_model <1>
  target_model :target_model <2>

  transform do
    # mapping rules
  end

  reverse_transform do
    # reverse mapping rules
  end
end
```

`source_model` Specifies the source model class.

`target_model` Specifies the target model class.

EXAMPLE

```
class PublicationTransform < Lutaml::Model::Transform
```

```
  source_model Publication
  target_model CatalogEntry

  transform do
    # mapping rules
    map from: 'title', to: 'title'
    map from: 'author', to: 'creator'
  end

  reverse_transform do
    # reverse mapping rules
    map from: 'target.title', to: 'source.title'
    map from: 'target.creator', to: 'source.author'
  end
end
```

Given:

```
publication = Publication.new(title: 'The Art of War', author: 'Sun
Tzu')
transformed = PublicationTransform.transform(publication)
# => #<CatalogEntry:0x00007f9b1b8b3b10 @title="The Art of War",
@creator="Sun Tzu">

publication_transformed = PublicationTransform.reverse_
transform(transformed)
# => #<Publication
#      @title="The Art of War",
#      @author="Sun Tzu">
```

6.3. Model mapping rules

6.3.1. Direct attribute mapping

Maps source attributes to target attributes with identical names without value modification.

This type of mapping is bidirectional by default.

Syntax:

FIGURE 7

```
    map from: 'path_from_source', to: 'path_at_target',
    directional: :bidirectional # default
    # or simply
    map from: 'path_from_source', to: 'path_at_target'
```

The `from:` and `to:` parameters are in the LutaML Path syntax.

EXAMPLE 1: In LutaML Path syntax, given a source model of `{ name("John Doe"), email("john@example.com") }`, the path to the `name` attribute is `name`, and the path to the `email` attribute is `email`.

EXAMPLE 2 — Direct attribute mapping example

```
class Publication < Lutaml::Model::Serializable
  attribute :title, :string
  attribute :author, :string
end

class CatalogEntry < Lutaml::Model::Serializable
  attribute :title, :string
  attribute :author, :string
end

class PublicationTransform < Lutaml::Model::Transform
  source_model Publication
  target_model CatalogEntry

  transform do
    map from: 'title', to: 'title'
    map from: 'author', to: 'author'
  end
end
```

6.3.2. Attribute renaming

Maps source attributes to differently named target attributes.

This type of mapping is bidirectional by default.

Syntax:

FIGURE 8

```
    map from: 'old_name', to: 'new_name'
```

EXAMPLE

```
class Person < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :year_born, :string
end

class User < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :birth_year, :string
end

class UserTransform < Lutaml::Model::Transform
  source_model Person
  target_model User
```

```
  transform do
    map from: 'year_born', to: 'birth_year'
  end
end
```

6.3.3. Value transformation mapping

Maps source attributes to target attributes with value transformation.

The directionality of a transformation mapping depends on the directionality of the value transform.

Syntax:

FIGURE 9

```
map from: 'attribute', to: 'attribute', transform:
TransformClass
```

EXAMPLE 1

```
class Person < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :birth_date, :string
end

class User < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :birth_date, :date_with_time
end

# This is a uni-directional transform
class DateFormatTransform < Lutaml::Value::Transform
  source_value :string
  target_value :date_with_time

  transform do |source_value|
    Date.parse(source_value)
  end
end

class UserTransform < Lutaml::Model::Transform
  source_model Person
  target_model User

  transform do
    # This is a uni-directional mapping
    map from: 'birth_date', to: 'birth_date', transform:
DateFormatTransform
  end
end
```

EXAMPLE 2

```
# This is a bidirectional transform
class DateFormatTransform < Lutaml::Value::Transform
```

```
  source_value :string
  target_value :date_with_time

  transform do |source_value|
    Date.parse(source_value)
  end

  reverse_transform do |target_value|
    target_target_value.strftime('%Y-%m-%d')
  end
end

class UserTransform < Lutaml::Model::Transform
  transform do
    # This becomes a bidirectional mapping
    map from: 'birth_date', to: 'birth_date', transform:
DateFormatTransform
  end
end
```

7. Nested transforms

7.1. General

Nested transforms handle complex object hierarchies, allowing transformation of nested attributes and objects.

7.2. Structure

A nested transform:

- Defines mappings for nested attributes using dot notation
- Handles collections appropriately
- Supports value transforms within nested mappings

7.3. Nested attribute mapping

Maps attributes within nested objects.

The directionality of a nested mapping depends on the directionality of the transform.

Syntax:

FIGURE 10

```
# Value to value transformation
map from: 'source_attribute', to: 'destination_attribute'

# Cross-model-value transformation
## Value to model transformation
map from: 'source_model.attribute', to: 'destination_
attribute'
## Model to value transformation
map from: 'source_attribute', to: 'destination_model.
attribute'

# Model to model transformation
map from: 'source_model.[...]attribute', to: 'destination_
model.[...]attribute'
```

There are three scenarios in transforming nested attributes:

1. Value to value transformation
2. Cross model-value transformation
3. Model to model transformation

7.4. Value to value transformation

We can apply a value-to-value transformation if both the source and destination attributes are values.

Suppose we have the following source and destination models.

FIGURE 11

```
class UnstructuredDateTime < Lutaml::Model::Value
  attribute :value, :string
end

class StructuredDateTime < Lutaml::Model::Value
  attribute :date, :string
  attribute :time, :string
end

class OldDigitalTimepiece < Lutaml::Model::Serializable
  attribute :raw_time, UnstructuredDateTime
end
```

```
class NewDigitalTimepiece < Lutaml::Model::Serializable
  attribute :detailed_time, StructuredDateTime
end
```

First define a value transform:

FIGURE 12

```
class DateTimeTransform < Lutaml::Value::Transform
  source_value UnstructuredDateTime
  target_value StructuredDateTime

  transform do |source_value|
    # In ISO 8601-1:2019, a basic date-time string can be
"{YYYY}{MM}{DD}T{hh}:{mm}:{ss}"
    date, time = source_value.value.split('T')
    StructuredDateTime.new(date: date, time: time)
  end

  reverse_transform do |target_value|
    UnstructuredDateTime.new(value: "#{target_
value.date}T#{target_value.time}")
  end
end
```

Then define a model transform:

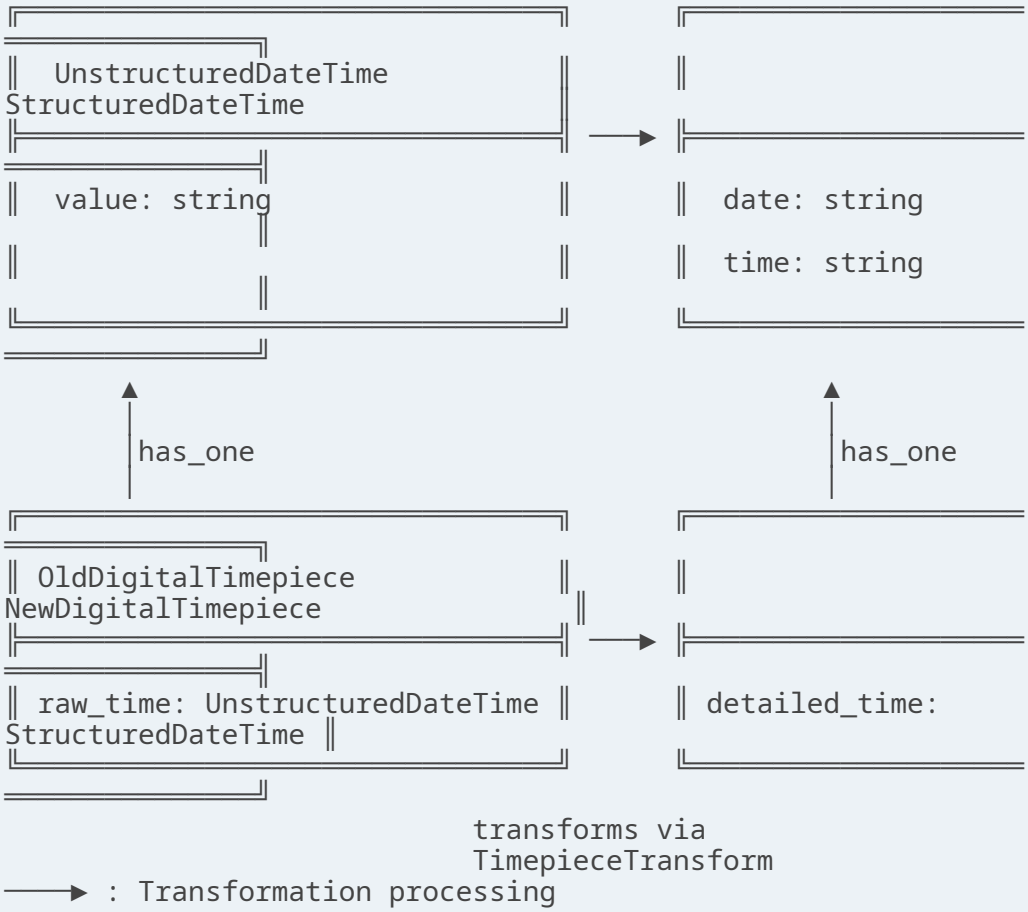
FIGURE 13

```
class TimepieceTransform < Lutaml::Model::Transform
  source_model OldDigitalTimepiece
  target_model NewDigitalTimepiece

  transform do
    map from: 'time', to: 'time', transform:
DateTimeTransform
  end
end
```

The resulting transformation is demonstrated with this diagram.

FIGURE 14

```
                              transforms via
                              DateTimeTransform
```



7.5. Cross model-value transformation

A cross-model-value transformation occurs when one of the source and destination attributes is a model and the other is a value.

Suppose we have a set of deeper source and destination models.

FIGURE 15

```
class UnstructuredDateTimeWithOffset < Lutaml::Model::Value
  attribute :value, :string
end

class StructuredDateTimeWithOffset < Lutaml::Model::Model
  attribute :date_time, StructuredDateTime
  attribute :offset, TimeOffset
end

class TimeOffset < Lutaml::Model::Value
  attribute :value, :string
end
```

```
class OldDigitalTimepiece < Lutaml::Model::Serializable
  attribute :raw_time, UnstructuredDateTimeWithOffset
end

class NewDigitalTimepiece < Lutaml::Model::Serializable
  attribute :detailed_time, StructuredDateTimeWithOffset
end
```

Here, the `UnstructuredDateTimeWithOffset` model is a value, and the`StructuredDateTimeWithOffset` model is a model.

A cross-model-value transform is defined to transform between`UnstructuredDateTimeWithOffset` (value) and `StructuredDateTimeWithOffset` (model).

FIGURE 16

```
class DateTimeTransform < Lutaml::Value::Transform
  source_value UnstructuredDateTimeWithOffset
  target_value StructuredDateTimeWithOffset

  # Split the string into date, time, and offset
  transform do |source_value|
    match, date, time, offset = source_value.value.match(/^(.
+?)T(.+?)(?:\+(.+))?$/)
    StructuredDateTimeWithOffset.new(
      date_time: StructuredDateTime.new(date: date, time:
time),
      offset: TimeOffset.new(value: offset || '00:00')
    )
  end

  reverse_transform do |target_value|
    UnstructuredDateTimeWithOffset.new(
      value:[
        target_value.date_time.date,
        "T",
        target_value.date_time.time,
        "+",
        target_value.zone.offset
      ].join("")
    )
  end
```

```
    end
```

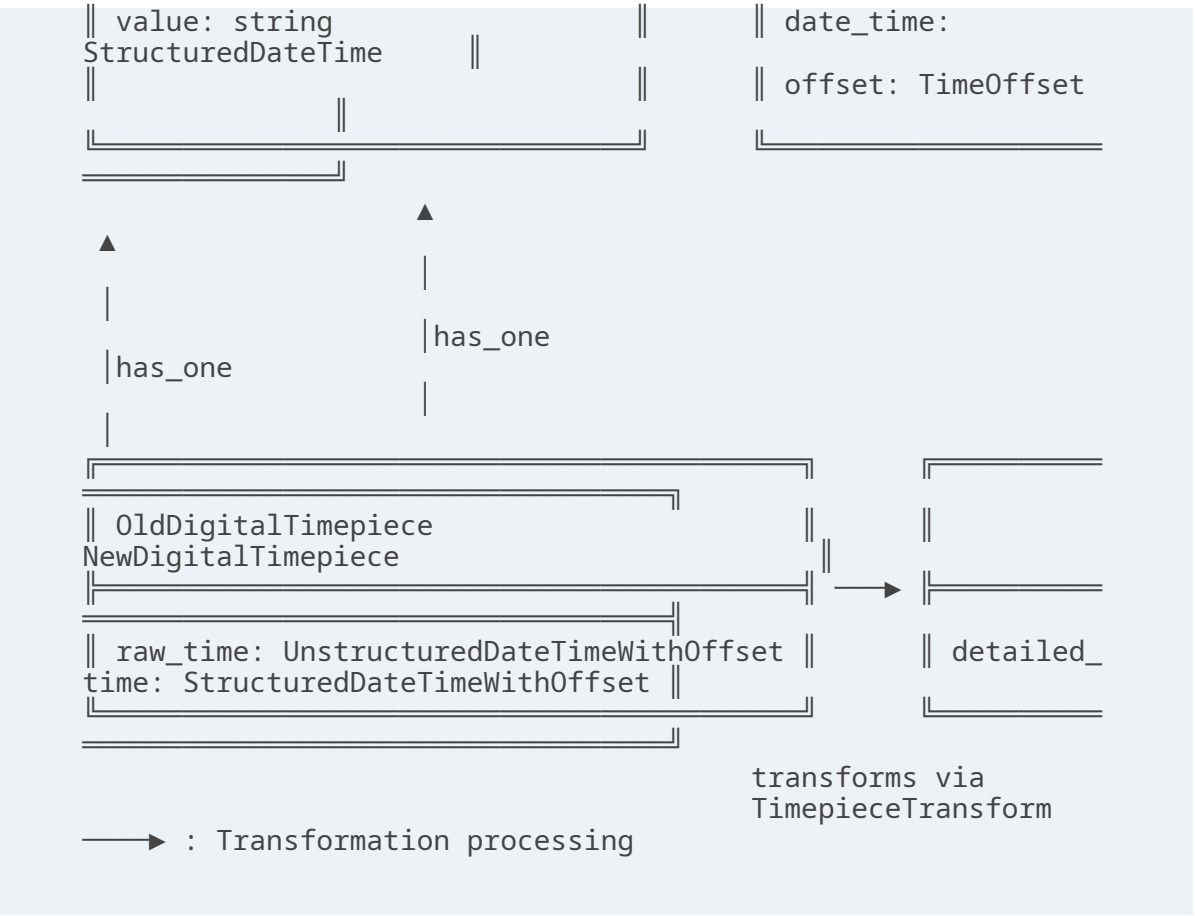
Then define a model transform.

FIGURE 17

```
class TimepieceTransform < Lutaml::Model::Transform
  source_model OldDigitalTimepiece
  target_model NewDigitalTimepiece

  transform do
    map from: 'raw_time', to: 'detailed_time', transform:
DateTimeTransform
  end
end
```

The resulting transformation is demonstrated with this diagram.

FIGURE 18



7.6. Model-to-model transformation

Model-to-model transformations handle mapping between models.

In a model-to-model transformation, a Value Transform may be used to transform the value of an attribute.

When approaching model to model transformation, the mapping can be placed at the desired level of transformation.

1. Creating the mapping at the most basic level, where either the mapping source or destination is an attribute accessible at root.

   EXAMPLE 1: In the mapping, a `time` path is a root attribute, referring to the attribute of `{source|target}.time`.

2. Creating the mapping at the model-to-model level, where both source and destination are models, . Given a deep model hierarchy, the transformation can be applied at a shallow level (e.g. close to the underlying value) or at a deeper level (e.g. close to the top-level model, very nested).

   EXAMPLE 2: A path of `clock.time` refers to an inner model attribute, referring to the attribute of `{source|target}.clock.time`.

A path of `computer.clock.time` refers to a deeper model attribute, referring to the attribute of `{source|target}.computer.clock.time`.

The consideration on where the mapping is placed depends on the accessibility, appropriateness and authority of the desired transformation.

A basic level value-to-value or cross-model-value transformation may not be practical when the source and destination models are not managed or owned by the party performing the transformation.

To illustrate such a scenario, we add one more model layer to the same model as in 7.5 to define `ComputerClock` and`WallClock`. Then move the transformation logic to the model level.

FIGURE 19

```
class UnstructuredDateTimeWithOffset < Lutaml::Model::Value
  attribute :value, :string
end

class StructuredDateTimeWithOffset < Lutaml::Model::Model
  attribute :date_time, StructuredDateTime
  attribute :offset, TimeOffset
end

class TimeOffset < Lutaml::Model::Value
  attribute :value, :string
end

class OldDigitalTimepiece < Lutaml::Model::Serializable
  attribute :raw_time, UnstructuredDateTimeWithOffset
end

class NewDigitalTimepiece < Lutaml::Model::Serializable
  attribute :detailed_time, StructuredDateTimeWithOffset
end

class WallClock < Lutaml::Model::Serializable
  attribute :timepiece, OldDigitalTimepiece
end

class ComputerClock < Lutaml::Model::Serializable
  attribute :timepiece, NewDigitalTimepiece
end

class DateTimeTransform < Lutaml::Value::Transform
  source_value UnstructuredDateTimeWithOffset
  target_value StructuredDateTimeWithOffset

  # Split the string into date, time, and offset
  transform do |source_value|
    match, date, time, offset = source_value.value.match(/^(.
+?)T(.+?)(?:\+(.+))?$/)
```

```
      StructuredDateTimeWithOffset.new(
        date_time: StructuredDateTime.new(date: date, time:
time),
        offset: TimeOffset.new(value: offset || '00:00')
      )
    end

    reverse_transform do |target_value|
      UnstructuredDateTimeWithOffset.new(
        value:[
          target_value.date_time.date,
          "T",
          target_value.date_time.time,
          "+",
          target_value.zone.offset
        ].join("")
      )
    end
  end
```

Now define a model transform between `ComputerClock` and `WallClock`.

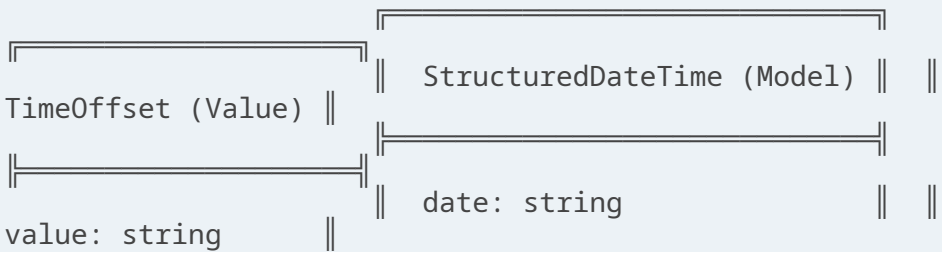
FIGURE 20

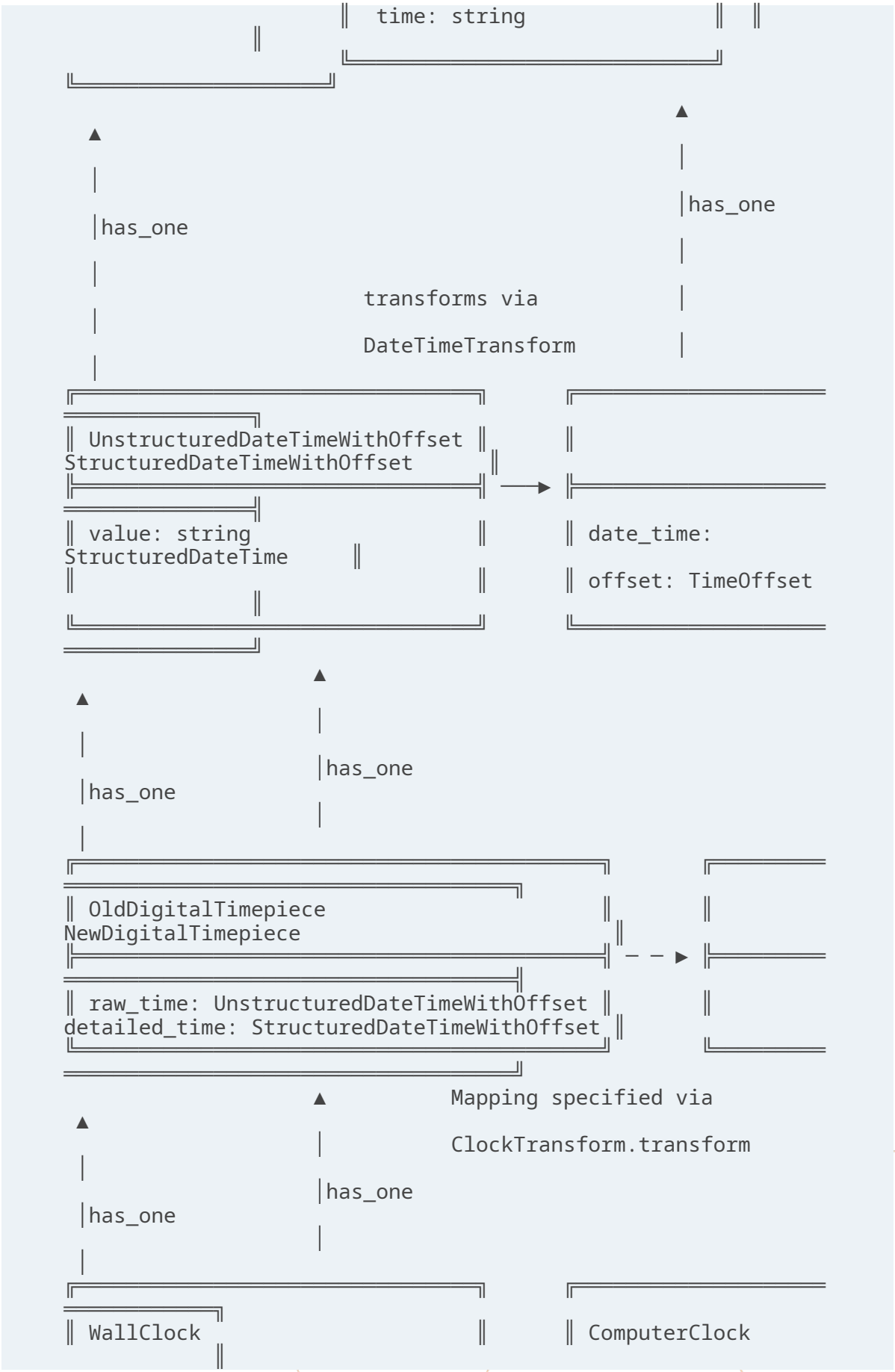
```
class ClockTransform < Lutaml::Model::Transform
  source_model WallClock
  target_model ComputerClock

  transform do
    map from: 'timepiece.raw_time', to: 'timepiece.detailed_
time', transform: DateTimeTransform
  end
end
```

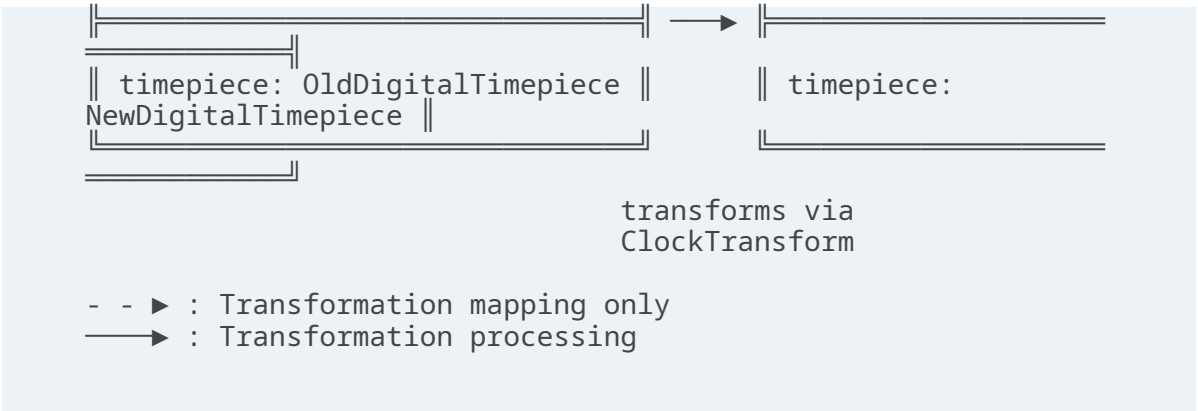
Notice that different from the 7.5, the transformation is now applied at the model level, where the mapping is defined at a model-to-model level where both source and destination do not own the attributes involved in transformation.

The resulting transformation is demonstrated with this diagram.

FIGURE 21







8. Collection transforms

8.1. General

Collection transforms handle mapping between collections of objects.

8.2. Models to models

The `map_each` command is used to specify that the source attribute is a collection.

Syntax:

FIGURE 22

```
map_each from: 'collection_path', to: 'attribute_or_
collection_path', transform: CollectionTransform
```

EXAMPLE

```
class Publication < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :authors, Author, collection: true
end

class CatalogEntry < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :contributors, Contributor, collection: true
end

class Author < Lutaml::Model::Serializable
  attribute :name, :string
```

```
end

class Contributor < Lutaml::Model::Serializable
  attribute :name, :string
end

# This is a model to model transform
class AuthorTransform < Lutaml::Model::Transform
  source_model Author
  target_model Contributor

  transform do
    map from: 'name', to: 'name'
  end
end

class PublicationTransform < Lutaml::Model::Transform
  source_model Publication
  target_model CatalogEntry

  transform do
    map_each from: 'authors', to: 'contributors', transform:
AuthorTransform
  end
end
```

8.3. Splitting models into a collection

Collection transforms can also split a single model into multiple entries in a collection. For instance, consider a transform that takes a publication's authors and converts them into a collection of contributors.

The `target_model …, collection: true` syntax is used to specify that the target attribute is a collection.

Syntax:

FIGURE 23

```
    map_each from: 'attribute', to: 'collection_path',
    transform: CollectionTransform
```

EXAMPLE

```
class PublicationV1 < Lutaml::Model::Serializable
  attribute :title, :string
  attribute :contributor_information, :string
end

class PublicationV2 < Lutaml::Model::Serializable
  attribute :title, :string
  attribute :contributors, Contributor, collection: true
end
```

```
class Contributor < Luatml::Model::Serializable
  attribute :name, :string
end

class ContributorTransform < Lutaml::Model::Transform
  source_value :string
  target_model Contributor, collection: true

  transform do |source_value|
    source_value.split(',').map do |name|
      Contributor.new(name: name)
    end
  end
end

class PublicationTransform < Lutaml::Model::Transform
  source_model PublicationV1
  target_model PublicationV2

  transform do
    map from: 'contributor_information', to: 'contributors',
transform: ContributorTransform
  end
end
```

8.4. Joining a collection into an attribute

Collection transforms can also join a collection of objects into a single attribute in the target model.

Syntax:

FIGURE 24

```
    map_each from: 'collection_path', to: 'attribute',
    transform: CollectionTransform
```

EXAMPLE

```
class StandardsPublication < Lutaml::Model::Serializable
  attribute :title, :string, collection: true
end

class BibliographyEntry < Lutaml::Model::Serializable
  attribute :title, :string
end

class TitleAggregationTransform < Lutaml::Model::Transform
  source_model :string, collection: true
  target_model :string

  transform do |source_values|
```

```
    source_values.join(', ')
  end
end

class StandardsPublicationTransform < Lutaml::Model::Transform
  source_model StandardsPublication
  target_model BibliographyEntry

  transform do
    map_each from: 'title', to: 'title', transform:
TitleAggregationTransform
  end
end
```

Annex A
(normative)

Tutorial: Complex transformation scenario

This tutorial demonstrates a complete transformation scenario using a museum's art collection as an example.

Consider the following model trees.

The "De Lutam'l Art Museum" has a collection of ceramic pieces managed in a register used for generic art information.

The register has the following fields:

FIGURE A.1

```
# First model tree
class GenericArtInformation < Lutaml::Model::Serializable
  attribute :title, :string
  attribute :description, :string
  attribute :artist, CreatorInformation
  attribute :creation_date, :string
  attribute :place_of_work, :string
end

class CreatorInformation < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :bio, :string
  attribute :website, :string
  attribute :year_born, :integer
  attribute :year_died, :integer
```

```
end
```

This is an example of a YAML file that represents the first model tree:

FIGURE A.2: In the `GenericArtInformation` model

```
---
- title: "Translucent Vase"
  description: |
    A tall and beautiful translucent vase created in the
celadon color.

    Dimensions: 10x10x10 cm
    Fire temperature: 1000°C
    Clay type: Porcelain
  artist:
    name: "Masaaki Shibata"
    bio: |
      Masaaki Shibata is a Japanese ceramic artist.

      Awards: Japan Ceramic Society Award, 2005.

      Skills: Glazing, painting
    website: "https://www.masaakishibata.com"
    year_born: 1947
    year_died: null
  creation_date: "2010-01-01"
  place_of_work: Tokyo, Japan
- title: "Blue and White Bowl"
  description: |
    A blue and white bowl with a floral pattern.

    Dimensions: 20x20x20 cm
    Fire temperature: 1200°C
    Clay type: Stoneware
    Glaze: Blue and white
  artist:
    name: "Lucie Rie"
    bio: |
      Lucie Rie was an Austrian-born British studio potter.

      Awards: Potter's Gold Medal, 1987.

      Skills: Throwing, glazing
    website: "https://www.lucierie.com"
    year_born: 1902
    year_died: 1995
  creation_date: "1970-01-01"
  place_of_work: London, UK
- title: "Ceramic Sculpture"
  description: |
    A ceramic sculpture in form of a golden fish.
```

```
    Dimensions: 30x10x20 cm
    Fire temperature: 800°C
    Clay type: Earthenware
    Glaze: Gold
  artist:
    name: "Peter Voulkos"
    bio: |
      Peter Voulkos was an American artist of Greek descent.

      Awards: National Medal of Arts, 2001.

      Skills: Throwing, hand-building, glazing

    website: "https://www.petervoulkos.com"
    year_born: 1924
    year_died: 2002
  creation_date: "1980-01-01"
  place_of_work: Portopolous, Greece
```

The museum wants to transform the generic art information model into a ceramic art information model to better manage the ceramic pieces.

FIGURE A.3

```
# Second model tree
class CeramicArtInformation < Lutaml::Model::Serializable
  attribute :title, :string
  attribute :description, :string
  attribute :artist, CeramicCreatorInformation
  attribute :creation_date, :date_with_time
  attribute :location, :string
  attribute :dimensions, Dimensions
  attribute :fire_temperature, :integer
  attribute :fire_temperature_unit, :string, values: %w[°C
°F]
  attribute :clay_type, :string
  attribute :glaze, :string
end

class Dimensions < Lutaml::Model::Serializable
  attribute :height, :integer
  attribute :width, :integer
  attribute :depth, :integer
end

class CeramicCreatorInformation < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :bio, :string
  attribute :website, :string
  attribute :year_of_birth, :integer
  attribute :year_of_death, :integer
  attribute :techniques, :string, collection: true
  attribute :awards, :string, collection: true
```

```
    end
```

We need to create a `Lutaml::Model::Transform` class that will transform the first model tree into the second model tree.

Let's first map the fields and group them according to the level of processing needed.

NOTE The "source" refes to the `GenericArtInformation` model, and the "target" refers to the `CeramicArtInformation` model.

TABLE A.1: No processing needed

Source attribute(s)	Target attribute(s)	Value processing needed
`title`	`title`	None
`description`	`description`	None

TABLE A.2: Attribute rename

Source attribute(s)	Target attribute(s)	Value processing needed
`place_of_work`	`location`	None

TABLE A.3: Value type conversion

Source attribute(s)	Target attribute(s)	Value processing needed
`creation_date`	`creation_date`	Convert string to date with time

TABLE A.4: Processing needed

Source attribute(s)	Target attribute(s)	Value processing needed
`fire_temperature`	`fire_temperature`	Extract from `description`
`fire_technique`	`fire_technique`	Extract from `description`
`clay_type`	`clay_type`	Extract from `description`
`glaze`	`glaze`	Extract from `description`
`dimensions`	`dimensions`	Extract specific values and map to Dimensions attributes

TABLE A.5: Nested attribute map

Source attribute(s)	Target attribute(s)	Value processing needed

artist.name	artist.name	None
artist.bio	artist.bio	None
artist.website	artist.website	None

TABLE A.6: Nested attribute rename

Source attribute(s)	Target attribute(s)	Value processing needed
artist.year_born	artist.year_of_birth	None
artist.year_died	artist.year_of_death	None

TABLE A.7: Nested attribute processing

Source attribute(s)	Target attribute(s)	Value processing needed
artist.bio	artist.techniques	Extract from artist bio
artist.bio	artist.awards	Extract from artist bio

The following `Lutaml::Model::Transform` class will transform the first model tree into the second model tree. We build this class incrementally by adding the necessary mappings.

Let's start with mapping the attributes that do not require any processing.

FIGURE A.4

```
class CeramicArtInformationTransform < Lutaml::Model:
:Transform
  source_model GenericArtInformation
  target_model CeramicArtInformation

  transform do
    # Simple mapping
    map from: 'title', to: 'title'
    map from: 'description', to: 'description'
  end
```

```
end
```

Next, we add the mapping for the attributes that require renaming.

FIGURE A.5

```
class CeramicArtInformationTransform < Lutaml::Model:
:Transform
  source_model GenericArtInformation
  target_model CeramicArtInformation

  transform do
    # Simple mapping
    map from: 'title', to: 'title'
    map from: 'description', to: 'description'

    # Rename attributes
    map from: 'place_of_work', to: 'location'
  end
end
```

Now let's add the mapping for the nested attributes.

FIGURE A.6

```
class CeramicArtInformationTransform < Lutaml::Model:
:Transform
  source_model GenericArtInformation
  target_model CeramicArtInformation

  transform do
    # Simple mapping
    map from: 'title', to: 'title'
    map from: 'description', to: 'description'

    # Rename attributes
    map from: 'place_of_work', to: 'location'

    # Nested attribute mapping
    map from: 'artist.name', to: 'artist.name'
    map from: 'artist.bio', to: 'artist.bio'
    map from: 'artist.website', to: 'artist.website'

    # Rename nested attributes
    map from: 'artist.year_born', to: 'artist.year_of_birth'
    map from: 'artist.year_died', to: 'artist.year_of_death'
  end
```

```
    end
```

Next, we add the mapping for the attributes that require value type conversion.

There are two ways we can specify a value transform.

1. By using a Lutaml::Value::Transform class that implements the `transform` and `reverse_transform` methods.
2. By using a block that takes the source value as an argument and returns the transformed value.

In the first manner, we define the `DateFormatTransform` class that converts a string to a date with time.

FIGURE A.7

```
class DateFormatTransform < Lutaml::Value::Transform
  source_value :string
  target_value :date_with_time

  transform do |source_value|
    Date.parse(source_value)
  end

  reverse_transform do |target_value|
    target_value.strftime('%Y-%m-%d')
  end
end
```

Then the mapping is added to the `CeramicArtInformationTransform` class like the following.

FIGURE A.8

```
# Value type conversion
map from: 'creation_date', to: 'creation_date', transform:
DateFormatTransform
```

In the second manner, we can use a block to specify the transformation.

FIGURE A.9

```
# Value type conversion
map from: 'creation_date', to: 'creation_date',
```

```
  transform: -> { |source_value|
    Date.parse(source_value)
  },
  reverse_transform: -> { |target_value|
    target_value.strftime('%Y-%m-%d')
  }
```

The example follows that we follow the first manner.

Next, we add the mapping for the attributes that require processing.

FIGURE A.10

```
class CeramicArtInformationTransform < Lutaml::Model:
:Transform
  source_model GenericArtInformation
  target_model CeramicArtInformation

  transform do
    # Simple mapping
    map from: 'title', to: 'title'
    map from: 'description', to: 'description'

    # Rename attributes
    map from: 'place_of_work', to: 'location'

    # Nested attribute mapping
    map from: 'artist.name', to: 'artist.name'
    map from: 'artist.bio', to: 'artist.bio'
    map from: 'artist.website', to: 'artist.website'

    # Rename nested attributes
    map from: 'artist.year_born', to: 'artist.year_of_birth'
    map from: 'artist.year_died', to: 'artist.year_of_death'

    # Value type conversion
    map from: 'creation_date', to: 'creation_date',
transform: DateFormatTransform

    # Single direction transform only, because the source
information remains
    # unchanged in a reverse migration.
    map from: 'description', to: 'fire_temperature',
transform: :extract_fire_temperature
    map from: 'description', to: 'fire_temperature_unit',
transform: :extract_fire_temperature_unit

    # Extract the clay type from the description.
    # e.g. "Clay type: Porcelain" => "Porcelain"
    map from: 'description', to: 'clay_type', transform: ->
{ |description|
      description.match(/Clay type: ([\w\s]+)/)[1]
    }
```

```
    # Extract the glaze from the description.
    # e.g. "Glaze: Blue and white" => "Blue and white"
    # Notice that the glaze is (optional), so we use a non-
greedy match.
    map from: 'description', to: 'glaze', transform: -> { |
description|
      description.match(/Glaze: (.+?)/)[1] rescue nil
    }

    # Use a separate method to extract dimensions from the
description.
    # Extract the fire temperature from the description.
    map from: 'description', to: 'fire_temperature',
transform: :extract_fire_temperature

    # e.g. "Fire temperature: 1000°C" => 1000
    # NOTE: Fire temperature might not be present.
    def extract_fire_temperature(description)
      description.match(/Fire temperature: (\d+)/)[1]&.to_i
    end

    # Use a separate method to extract dimensions from the
description.
    # Extract the temperature unit from the description.
    map from: 'description', to: 'fire_temperature_unit',
transform: :extract_fire_temperature_unit

    # e.g. "Fire temperature: 1000°C" => "°C"
    # NOTE: Fire temperature might not be present.
    def extract_fire_temperature_unit(description)
      description.match(/Fire temperature: \d+(°\w+)/)
[1]&.to_s
    end

    # Nested attribute with extracted values
    map from: 'artist.bio', to: 'artist.techniques',
transform: :extract_techniques
    map from: 'artist.bio', to: 'artist.awards', transform: :
extract_awards

    # Extract techniques from the bio text.
    # e.g. "Technique: Pottery" => ["Pottery"]
    def extract_techniques(bio)
      bio.scan(/Technique: ([\w\s]+)/).flatten
    end

    # Extract awards from the bio text.
    # e.g. "Award: Best in Show" => ["Best in Show"]
    def extract_awards(bio)
      bio.scan(/Award: ([\w\s]+)/).flatten
    end
  end
```

```
end
```

Now we have to create a transformation for the Dimensions attribute.

FIGURE A.11

```
# Transforms a string into a Dimension model
class DimensionsTransform < Lutaml::Model::Transform
  source_value :string
  target_model :dimensions

  transform do |source_value|
    height, width, depth = source_value.match(/Dimensions:
(\d+)x(\d+)x(\d+)/).captures
    target_model.new(
      height: height.to_i,
      width: width.to_i,
      depth: depth.to_i
    )
  end

  reverse_transform do |target_model|
    "#{target_model.height}x#{target_model.width}x#{target_
model.depth}"
  end
end
```

Then we add the mapping to the `CeramicArtInformationTransform` class.

FIGURE A.12

```
# Extract dimensions from the description.
map from: 'description', to: 'dimensions', transform:
DimensionsTransform
```

Finally, we add the reverse transformation to the `CeramicArtInformationTransform` class.

FIGURE A.13

```
class CeramicArtInformationTransform < Lutaml::Model:
:Transform
  source_model GenericArtInformation
  target_model CeramicArtInformation

  transform do
```

```
    # Simple mapping
    map from: 'title', to: 'title'
    map from: 'description', to: 'description'

    # Rename attributes
    map from: 'place_of_work', to: 'location'

    # Nested attribute mapping
    map from: 'artist.name', to: 'artist.name'
    map from: 'artist.bio', to: 'artist.bio'
    map from: 'artist.website', to: 'artist.website'

    # Rename nested attributes
    map from: 'artist.year_born', to: 'artist.year_of_birth'
    map from: 'artist.year_died', to: 'artist.year_of_death'

    # Value type conversion
    map from: 'creation_date', to: 'creation_date',
transform: DateFormatTransform

    # Single direction transform only, because the source
information remains
    # unchanged in a reverse migration.
    map from: 'description', to: 'fire_temperature',
transform: :extract_fire_temperature
    map from: 'description', to: 'fire_temperature_unit',
transform: :extract_fire_temperature_unit

    # Extract the clay type from the description.
    # e.g. "Clay type: Porcelain" => "Porcelain"
    map from: 'description', to: 'clay_type', transform: ->
{ |description|
      description.match(/Clay type: ([\w\s]+)/)[1]
    }

    # Extract the glaze from the description.
    # e.g. "Glaze: Blue and white" => "Blue and white"
    # Notice that the glaze is (optional), so we use a non-
greedy match.
    map from: 'description', to: 'glaze', transform: -> { |
description|
      description.match(/Glaze: (.+?)/)[1] rescue nil
    }

    # Use a separate method to extract dimensions from the
description.
    # Extract the fire temperature from the description.
    map from: 'description', to: 'fire_temperature',
transform: :extract_fire_temperature

    # e.g. "Fire temperature: 1000°C" => 1000
    # NOTE: Fire temperature might not be present.
    def extract_fire_temperature(description)
      description.match(/Fire temperature: (\d+)/)[1]&.to_i
    end

    # Use a separate method to extract dimensions from the
description.
```

```
    # Extract the temperature unit from the description.
    map from: 'description', to: 'fire_temperature_unit',
transform: :extract_fire_temperature_unit

    # e.g. "Fire temperature: 1000°C" => "°C"
    # NOTE: Fire temperature might not be present.
    def extract_fire_temperature_unit(description)
      description.match(/Fire temperature: \d+(°\w+)/)
[1]&.to_s
    end

    # Nested attribute with extracted values
    map from: 'artist.bio', to: 'artist.techniques',
transform: :extract_techniques
    map from: 'artist.bio', to: 'artist.awards', transform: :
extract_awards

    # Extract techniques from the bio text.
    # e.g. "Technique: Pottery" => ["Pottery"]
    def extract_techniques(bio)
      bio.scan(/Technique: ([\w\s]+)/).flatten
    end

    # Extract awards from the bio text.
    # e.g. "Award: Best in Show" => ["Best in Show"]
    def extract_awards(bio)
      bio.scan(/Award: ([\w\s]+)/).flatten
    end

    # Extract dimensions from the description.
    map from: 'description', to: 'dimensions', transform:
    DimensionsTransform
  end
end

# Transforms a string into a Dimension model
class DimensionsTransform < Lutaml::Model::Transform
  source_value :string
  target_model Dimensions

  transform do |source_value|
    height, width, depth = source_value.match(/Dimensions:
(\d+)x(\d+)x(\d+)/).captures
    target_model.new(
      height: height.to_i,
      width: width.to_i,
      depth: depth.to_i
    )
  end

  reverse_transform do |target_model|
    "#{target_model.height}x#{target_model.width}x#{target_
model.depth}"
  end
end

# Transforms a string into a Date model
class DateFormatTransform < Lutaml::Value::Transform
```

```
    source_value :string
    target_value :date_with_time

    transform do |source_value|
      Date.parse(source_value)
    end

    reverse_transform do |target_value|
      target_value.strftime('%Y-%m-%d')
    end
  end
```

The transformation is now complete.

We can now use the `CeramicArtInformationTransform` class to transform the data from the first model tree to the second model tree.

FIGURE A.14

```
# Load the data from the YAML file
data = YAML.load_file('generic_art_information.yaml')

# Load the generic art information
generic_art_info = GenericArtInformation.from_yaml(data)

# Transform the data
transformed_data =
CeramicArtInformationTransform.transform(generic_art_info)

transformed_data.first.class
# => CeramicArtInformation

# Save the transformed data to a YAML file
File.write('ceramic_art_information.yaml', transformed_
data.to_yaml)
```

The transformed data looks like this.

FIGURE A.15: Data instances in the `CeramicArtInformation` model

```
---
- title: "Translucent Vase"
  description: |
    A tall and beautiful translucent vase created in the
celadon color.

    Dimensions: 10x10x10 cm
    Fire temperature: 1000°C
```

```
    Clay type: Porcelain
  artist:
    name: "Masaaki Shibata"
    bio: |
      Masaaki Shibata is a Japanese ceramic artist.

      Awards: Japan Ceramic Society Award, 2005.

      Skills: Glazing, painting
    website: "https://www.masaakishibata.com"
    year_of_birth: 1947
    year_of_death: null
    techniques:
      - "Glazing"
      - "Painting"
    awards:
      - "Japan Ceramic Society Award, 2005"
  creation_date: "2010-01-01"
  location: Tokyo, Japan
  dimensions:
    height: 10
    width: 10
    depth: 10
  fire_temperature: 1000
  fire_temperature_unit: "°C"
  clay_type: "Porcelain"
  glaze: null
- title: "Blue and White Bowl"
  description: |
    A blue and white bowl with a floral pattern.

    Dimensions: 20x20x20 cm
    Fire temperature: 1200°C
    Clay type: Stoneware
    Glaze: Blue and white
  artist:
    name: "Lucie Rie"
    bio: |
      Lucie Rie was an Austrian-born British studio potter.

      Awards: Potter's Gold Medal, 1987.

      Skills: Throwing, glazing
    website: "https://www.lucierie.com"
    year_of_birth: 1902
    year_of_death: 1995
    techniques:
      - "Throwing"
      - "Glazing"
    awards:
      - "Potter's Gold Medal, 1987"
  creation_date: "1970-01-01"
  location: London, UK
  dimensions:
    height: 20
    width: 20
    depth: 20
  fire_temperature: 1200
```

```
    fire_temperature_unit: "°C"
    clay_type: "Stoneware"
    glaze: "Blue and white"
  - title: "Ceramic Sculpture"
    description: |
      A ceramic sculpture in form of a golden fish.

      Dimensions: 30x10x20 cm
      Fire temperature: 800°C
      Clay type: Earthenware
      Glaze: Gold
    artist:
      name: "Peter Voulkos"
      bio: |
        Peter Voulkos was an American artist of Greek descent.

        Awards: National Medal of Arts, 2001.

        Skills: Throwing, hand-building, glazing

      website: "https://www.petervoulkos.com"
      year_of_birth: 1924
      year_of_death: 2002
      techniques:
        - "Throwing"
        - "Hand-building"
        - "Glazing"
      awards:
        - "National Medal of Arts, 2001"
    creation_date: "1980-01-01"
    location: Portopolous, Greece
    dimensions:
      height: 30
      width: 10
      depth: 20
    fire_temperature: 800
    fire_temperature_unit: "°C"
    clay_type: "Earthenware"
    glaze: "Gold"
```