

An aerial night photograph of the TU/e campus in Eindhoven, showing several modern glass-walled buildings illuminated from within. The image is overlaid with a semi-transparent red filter. The text is positioned on the left side of the image.

Domain Specific Language Design (2IMP20)

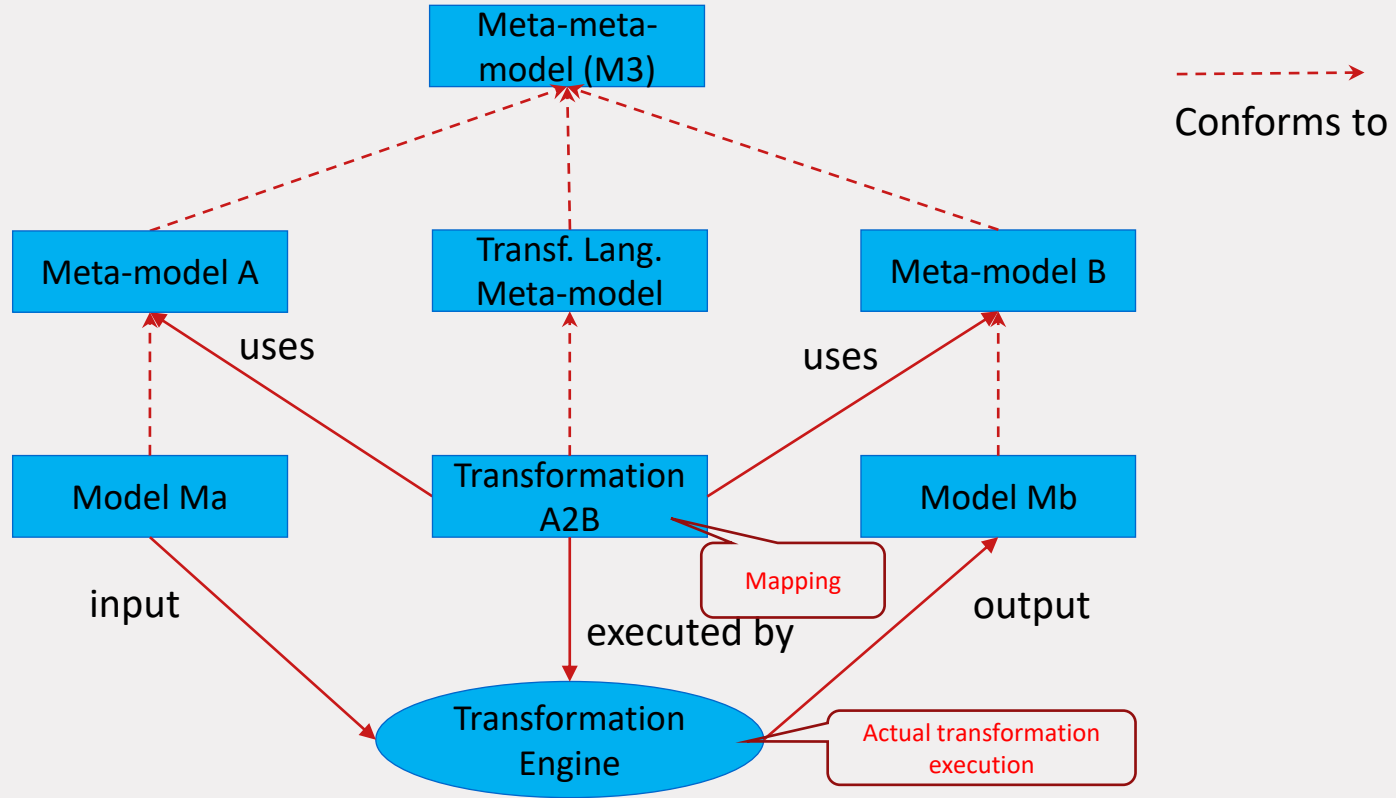
Model Transformations. ETL

Loek Cleophas, Ivan Kurtev

Agenda

- Model management with Epsilon platform
- Epsilon Transformation Language (ETL)
- Design and Testing of Model Transformations

Model Transformation Pattern



Model Transformation Languages

ATL

Xtend

QVT Relations

QVT Operational Mappings

QVT Core

Mola

Henshin

Rascal

Stratego/XT

VIATRA

Tefkat

ETL (Epsilon Transformation Language)

GrGen

...

The Epsilon Platform

Research platform by University of York (UK) that supports *model management tasks*

- <https://www.eclipse.org/epsilon/>

Epsilon Transformation Language (ETL) is part of the Epsilon Platform

Model Management Tasks

Validation, transformation and generation are common tasks that can be applied to models:

- Transforming models
- Generating text from models
- Refactoring models
- Merging models
- Validating models
- Comparing models
- Migrating models as their metamodel changes
- Querying and modifying models
- Workflows of tasks

Model Management with Epsilon

These tasks can be performed by using existing technologies. In the Modelware world, such technologies are usually built on top of EMF

- OCL for validation, QVTo for transformation, Acceleo for code generation, etc.
- This approach requires specification of a complex workflow; tool interoperability and integration are challenging

Epsilon platform aims at providing a *coherent set* of tools for model management that share *common syntax* and are designed to *interoperate* and to be *integrated*

Epsilon Platform

Epsilon is a family of integrated domain-specific languages for managing models:

- EOL (base language): similar to OCL but with extra features. Reused in other Epsilon DSLs
- ETL (transformations)
- EGL (generation)
- EVL (validation)
- ...

Strong integration with EMF

Epsilon Transformation Language

Overview

- Model-to-model transformation language
 - Can operate on EMF models and metamodels
- Hybrid language (declarative and imperative parts)
- Arbitrary number of source/target models
- Support for traceability

Epsilon Transformation Language

ETL will be explained on the basis of the transformation problem for UML class models to relational schemas

- The same problem was already solved with QVTo

An ETL transformation consists of *rules*:

- the main unit of transformation logic
- conceptually correspond to *mapping*
- rules have a *single source* element and *multiple target* elements

ETL Transformations and Rules

```
pre {  
  "Running Class to Relational with ETL".println();  
  var schema : new Relational!Schema;  
}
```

Block to be executed
before the rules.

Target root model
element is created

```
post {  
  "Finished Class to Relational with ETL".println();  
}
```

Block to be executed after
rules are executed

ETL rule

```
rule Class2Table  
  transform c : Class!Class  
  to t : Relational!Table, pk : Relational!Column {  
    t.name = c.name;  
    t.schema = schema;  
    pk.name = "objectId";  
    pk.type = getIntegerType();  
    t.col.add(pk);  
    t.key.add(pk);  
  }
```

Imperative block in EOL to
set the properties of the
target objects

ETL Rules

```
rule PrimSingleValAttr2Column
  transform a : Class!Attribute
  to c : Relational!Column {
    guard : not a.multiValued and a.type.isKindOf(Class!DataType)
    c.name = a.name;
    c.type ::= a.type;
    c.owner ::= a.owner;
  }
```

Guard: the rule is applied
only if the guard is true

Rules are declarative specifications of mappings:

- From a single source to multiple target elements
- Rule guard further restricts the source elements to which the mapping applies
- No explicit order of rule invocation

ETL Rules

```
rule PrimSingleValAttr2Column
  transform a : Class!Attribute
  to c : Relational!Column {
    guard : not a.multiValued and a.type.isKindOf(Class!DataType)
```

```
    c.name = a.name;
```

Assignments of values
for the features of the
target elements

```
    c.type ::= a.type;
```

Special assignment: assigns the
value obtained after transforming
the right-hand side

```
    c.owner ::= a.owner;
```

```
}
```

The value will be the table obtained
from the class given by a.owner

ETL Rule Execution

For all rules

- all the elements in the input model that match the rule source are found based on the source type and the guard (if any)

A rule is executed on all its matches in the source model

- for every match, the target part is instantiated, and property values are set

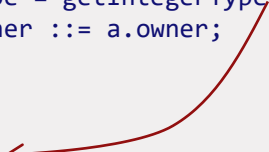
No explicit execution order of rules and order of a rule application on its matches

- the transformation engine determines the order
- in this example, rules do not call each other explicitly

ETL Operations

```
rule ClassSingleValAttr2Column
  transform a : Class!Attribute
  to fk : Relational!Column {
    guard : not a.multiValued and a.type.isKindOf(Class!Class)
    fk.name = a.name + 'Id';
    fk.type = getIntegerType();
    fk.owner ::= a.owner;
  }

operation getIntegerType() : Relational!Type {
  return Class!DataType.allInstances()->select(e | e.name = "Integer")->first().equivalent();
}
```



ETL *operations* are similar to QVTo queries

- explicitly called from rules

Lazy and Non-lazy Rules

In the example we have only seen *non-lazy* ETL rules

- Non-lazy rule is executed by the transformation engine on all the matches in the input model
- Non-lazy rule is executed once on a match

ETL supports *lazy* rules:

- Not scheduled for execution by the engine; have to be called explicitly from another rule
- Useful when a single source element has to be transformed multiple times, each time producing a new result

Internal Trace and Source-Target Resolution

ETL transformation engine maintains an internal trace

- Pairs of (source element, tuple of target elements)

ETL provides the **equivalents()** and **equivalent()** built-in operations that automatically resolve source elements to their transformed counterparts in the target models

equivalents()

- When invoked on a source element returns all the results obtained by rules applied on that element
- Can be parameterized with rule names. In this case, only the elements produced by the given rules are returned. If a lazy rule is passed as a parameter then it is executed

Internal Trace and Source-Target Resolution

`equivalent()`

- Returns only the first element from the ones that would be returned by `equivalents()`
- Useful especially if we know that a source element is transformed by one mapping only

`::=` is shorthand for `.equivalent()`

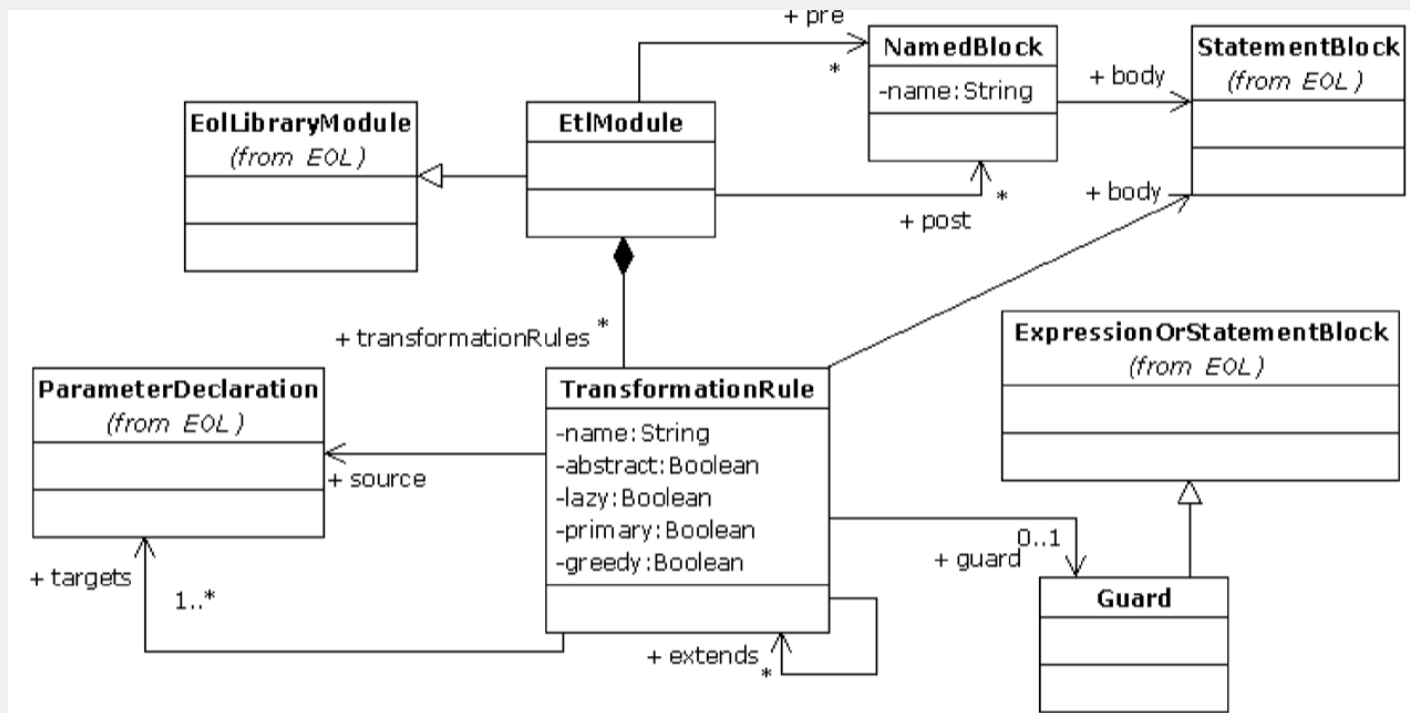
```
rule ClassSingleValAttr2Column
  transform a : Class!Attribute
  to fk : Relational!Column {

    guard : not a.multiValued and a.type.isKindOf(Class!Class)
    fk.name = a.name + 'Id';
    fk.type = getIntegerType();
    fk.owner ::= a.owner;
  }
```

Shorthand for
`fk.owner = a.owner.equivalent()`

Rule Abstract Syntax

Abstract syntax of ETL rules (part of ETL metamodel)



Rule Concrete Syntax

Concrete syntax of ETL rules

```
(@abstract)?  
(@lazy)?  
rule <name>  
  transform <sourceParameterName>:<sourceParameterType>  
  to (<rightParameterName>:<rightParameterType>  
    (, <rightParameterName>:<rightParameterType>)*  
  (extends (<ruleName>,)*<ruleName>)? {  
  
    (guard (:expression) | ({statement+}))?  
  
    statement+  
  }
```

Rule inheritance is supported (not illustrated)

- Inheriting rules can specialize the source type
- Inheriting rules can add new target elements
- Abstract rules can be defined for the purpose of reuse

ETL Transformation Execution

Execution order of ETL transformations:

- Pre-blocks
- **Non-abstract, non-lazy rules**
- Post-blocks

Pay attention that rules are defined declaratively, and the execution semantics mentioned above does not say the exact order of matching and execution (order of rules, order of processing the matches)

ETL Transformation Execution

What happens behind the scenes?

- Execution scheduling of ETL rules
1. Find all matches for all *non-abstract, non-lazy* rules
 2. For every match create the target elements (no feature setting yet)
 - Create a record in the execution trace
 3. For every newly created target element, assign values to the features
 - Values are based either on primitive values or on other target elements (created in the previous step) or on elements provided by explicitly called lazy rules
 - Invoke lazy rules if explicitly called
 - Consult the internal trace when *equivalents()* and *equivalent()* are used

Other ETL Features: Interactive Transformations

User can provide input during transformation execution:

```
rule Tree2Node
  transform t : Tree!Tree to n : Graph!Node {

    guard : UserInput.confirm("Transform tree " +
                              t.label + "?", true)

    n.label = t.label;
    var target : Graph!Node ::= t.parent;
    if (target.isDefined()) {
      var edge = new Graph!Edge;
      edge.source = n;
      edge.target = target;
    }
  }
```

Model Transformation Styles

Transformation languages can be classified according to their *style*: the degree of explicitly specifying the rule execution order and model navigation

- Declarative:
 - rules: correspondence between source and target patterns
 - no explicit execution order
 - decoupled rules: no explicit rule calls; traversal over the source model is usually local
 - more complex problems are often difficult to express
- Imperative:
 - statements are used to express the transformation logic; explicit rule calls; explicit execution order
 - declarative languages can solve some of these tasks based on the dependencies among model elements

Model Transformation Styles

- Hybrid:
 - a mix of declarative and imperative features, e.g. declarative rules and imperative statements for assigning values to model element features
 - typical scenario: start in a declarative way, resort to imperative way when the specific transformation task gets more difficult

ETL is a hybrid language

QVTo is an imperative language (however, with declarative style of mapping definition)

Usually, all styles provide support for implicit creation and query of the transformation trace

Implementing the Example Problem in GPL

The Class2Relational was implemented in QVTo and in ETL

- Can also be implemented in a Java using EMF API

All the typical transformation tasks have to be handled explicitly and from scratch

- More code needs to be written compared to the solutions with transformation languages

Summary on Epsilon

- Platform for model management
- Model management tasks handled by DSLs (basic model operations, validation, transformation, merging, migration, generation workflow, code generation)
- Overcomes integration and interoperability issues in model management
- ETL is the transformation language of Epsilon
 - Hybrid transformation language

Resources on Epsilon

The Epsilon book

<http://www.eclipse.org/epsilon/doc/book/>



Model Transformations: Final Considerations

- Other classifications of languages
- Higher order transformations
- Design of model transformations
- Testing of model transformations

Direction of Model Transformations

We distinguish between *uni-directional* and *bi-directional* transformations

- Uni-directional: the same transformation cannot be applied on the output to get the input back
- Bi-directional: the same transformation can be applied symmetrically to its input/output

Conceptually, not all transformations are bi-directional

Directionality impacts the design of *transformation languages*

Model Transformation Applications

Model transformations are not:

- necessarily semantics (or information) preserving
 - they can be, but there are useful transformations that are “lossy”
- necessarily refinements
 - they can be (especially update-in-place transformations) but many useful ones are not
- necessarily specified in a way that allows interesting properties to be checked on them

Higher Order Transformations

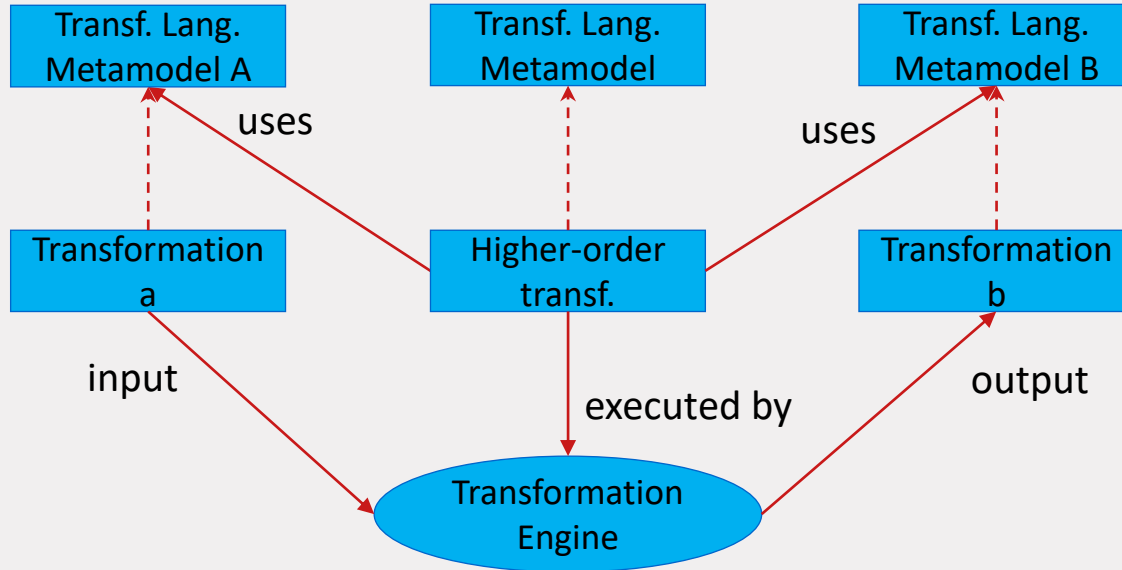
A transformation expressed in a language for which we have a metamodel is a model itself

- The transformation can be an input or output model of another model transformation (possibly in another transformation language)

Higher order transformation

- A transformation that produces another transformation
- Sometimes less strict meaning: a transformation that has another transformation as input and/or output

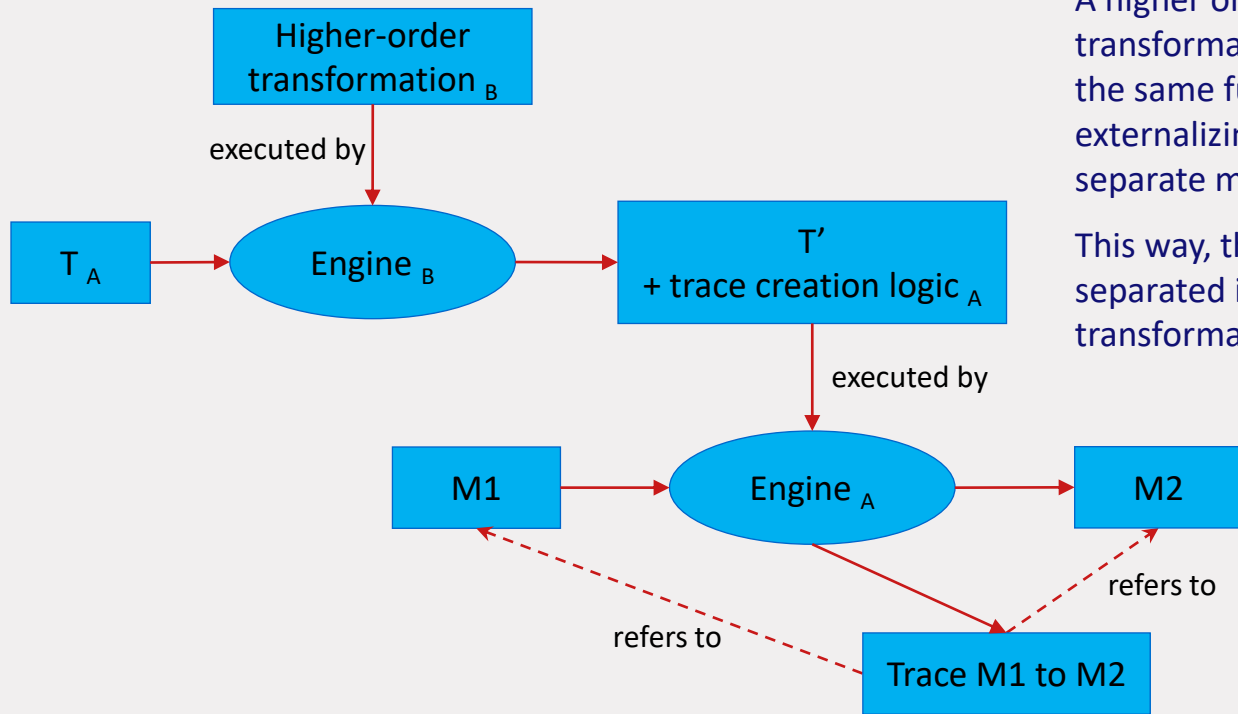
Higher Order Transformations



An example higher order transformation pattern:

- Translation between two transformation languages
- The higher order transformation can be written in one of the two languages or in a 3d one

An Example Higher Order Transformation



A higher order transformation takes a transformation T and produces T' that has the same functionality plus logic for externalizing the execution trace as a separate model.

This way, the trace creation logic is separated in the higher order transformation

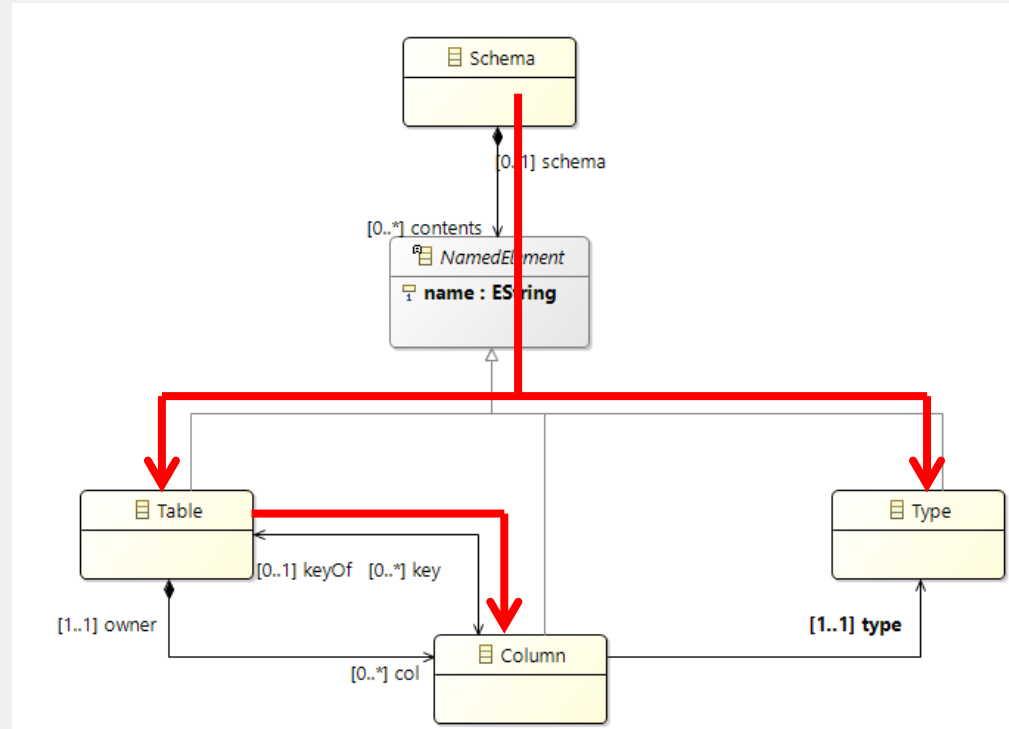
Designing Model Transformations

Given source and target metamodels, how do we *decompose* the transformation logic?

One possibility is to use the structure of the source/target metamodel

- Source-driven decomposition: how source elements are translated to target elements
- Target-driven decomposition: how to construct the result (e.g. top-down)

Target-driven Decomposition



A model is a *tree*. Start from the *root* class:

- How do we create the root element?

Every element *contains* other elements

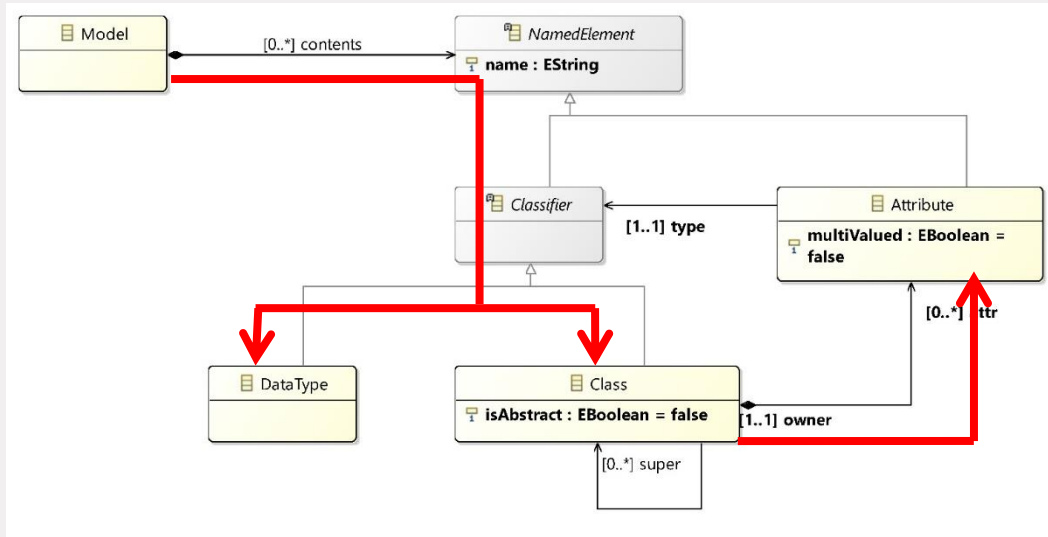
- How do we obtain the directly contained elements?

Every element *refers* to other elements

- How do we obtain these elements (think of order or target element creation)?

This is the approach used in the QVTo example

Source-driven Decomposition



- Model is mapped to Schema
- DataType is mapped to Type
- Class is mapped to Table and an Attribute
- 4 different cases for attributes (use guards)

This is the approach used in the ETL example

Chain of Model Transformations

Sometimes the source and target metamodels are too 'distant':

- big structural differences
- different levels of abstraction

It might be difficult to implement the transformation logic in a single transformation

- Use a decomposition of *transformation steps* that form a *transformation chain*

Usually, non-trivial transformations in industry are implemented as a chain

Chain of Model Transformations



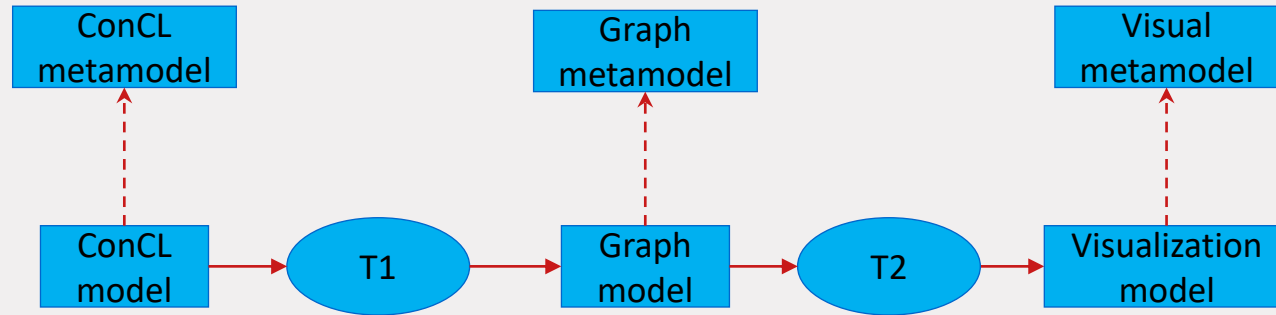
- From design perspective every step handles a single *aspect* of the logic
- Chaining may require developing metamodels for the intermediate models (but some may exist)
- Sometimes intermediate generic structures may be used (tuples, sets, collections, etc.) instead of a full-fledged metamodel
 - QVTo allows definitions of *intermediate classes* for this purpose, without creating a metamodel

Chain of Model Transformations: Example

Assignment 2 can be perceived as a first step in a transformation chain for visualizing ConCL models:

In the first step we are interested in capturing the *structural aspect*. Target model is just a *graph*

- The second step deals with the *visual aspect*. In some cases, it may exist and may be reused



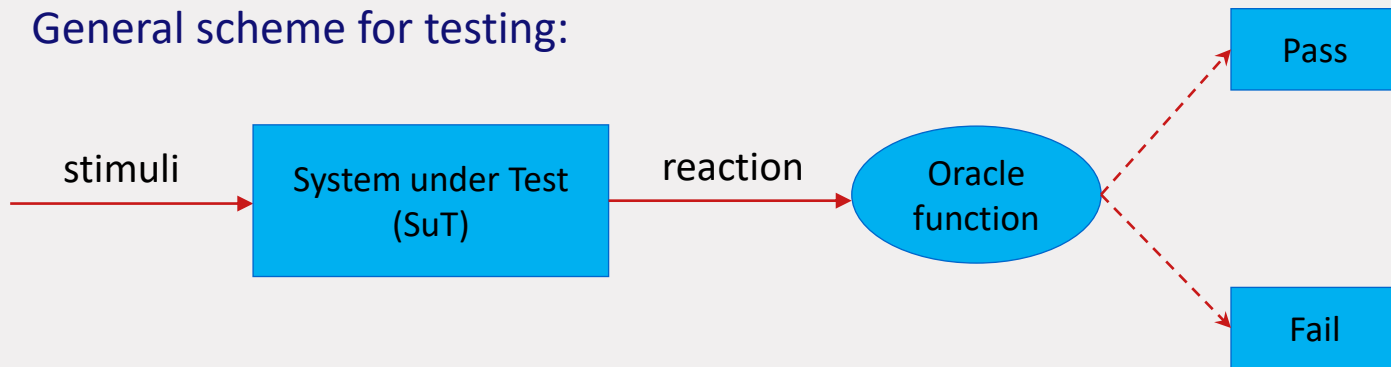
Testing Model Transformations

A model transformation is a software artefact

- Needs to be specified, designed, implemented, tested

How do we test a model transformation?

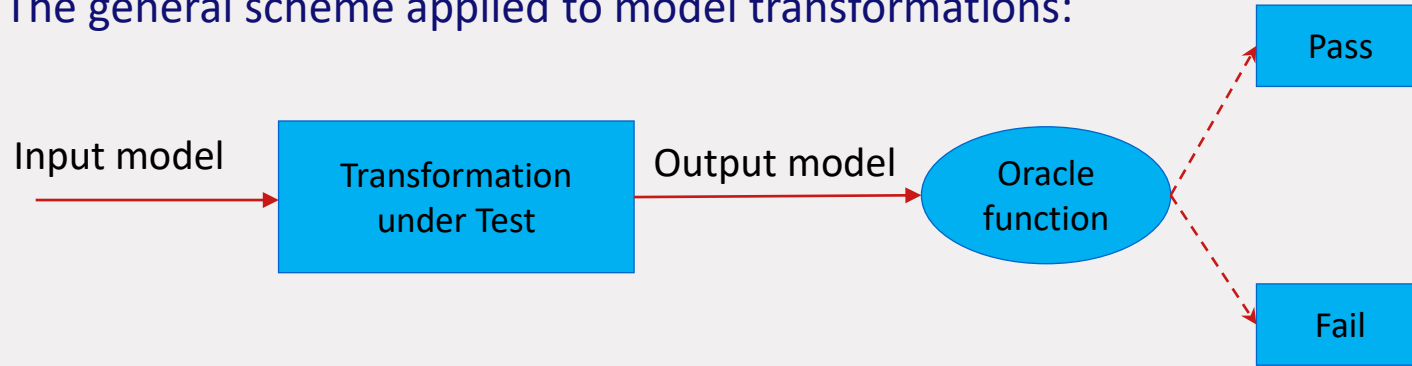
General scheme for testing:



The *system under test* is sent an input (*stimuli*) and produces an *observable reaction*. The reaction is evaluated by a decision procedure called *oracle function* that produces a verdict: the reaction is as expected (*test passes*) or not (*test fails*)

Testing Model Transformations

The general scheme applied to model transformations:



Challenges:

- How to produce the input test models? The set of possible input models is infinite
- What is the oracle function?
- What is a good test suite?

Testing Model Transformations

Constructing input test models

- Preferably in automatic way
- ‘Similar’ input models have to be avoided (*efficient* test suite)
- All ‘interesting cases’ have to be represented (*effective* test suite)
 - Based on metrics for input *metamodel coverage*
 - Based on metrics for *transformation program coverage*

Testing Model Transformations

Possible oracle functions

- Give an explicit expected result model and compare (naïve, time consuming approach)
- Property to be evaluated on the result model or on the pair (input, output)
 - Suitable when the transformation has formally specified requirements
- If the result model is executable, test the result model
 - This approach can be also used when code is generated
- Check if the result model *behaves as* the input model
 - If the input model can be executed/simulated, use it as a reference behavior (back-to-back testing)

Unfortunately, these approaches are not supported yet in a mature way and are not a mainstream industrial practice

Summary on Model Transformations

- Model transformations are a key operation in MDE
- Used in many MDE scenarios (code generation, reverse engineering, refactoring, ...)
- Repetitive task: addressed by DSLs called *model transformation languages*
 - Automation for common transformation problems
 - Can be implemented as a library in a GPL (internal DSL approach)