EINDHOVEN UNIVERSITY OF TECHNOLOGY

Domain Specific Language Design (2IMP20)

Model Transformations. QVT Operational

Loek Cleophas, Ivan Kurtev

Agenda

- Introduction to Model Transformations
- QVT Operational (QVTo)

Everything is a Model (Revisited)

Artefacts are treated uniformly as models

- everything is a model expressed in a language

Processing models:

- in Modelware, the main *operation* on models is *model transformation*
- Model transformations automate many engineering tasks
  - Code generation, reverse engineering, data translation, model updates, etc.

Model Transformation Applications

- Elaboration(refinement): generating detailed models or code from abstract models
- Synchronization: ensuring consistency between models at the same or different levels of abstraction
- View creation: producing query-based views
- Model evolution (including refactoring)
- Abstraction: generating less detailed models from more detailed ones

TU/e

Model Transformations

Specify the way to produce *target models* from a number of *source models*

Define the way source model elements must be matched and navigated in order to create target model elements

A model transformation is a *mapping* of a set of models onto another set of models or onto themselves, where a mapping defines correspondences between source and target model elements

Model Transformations

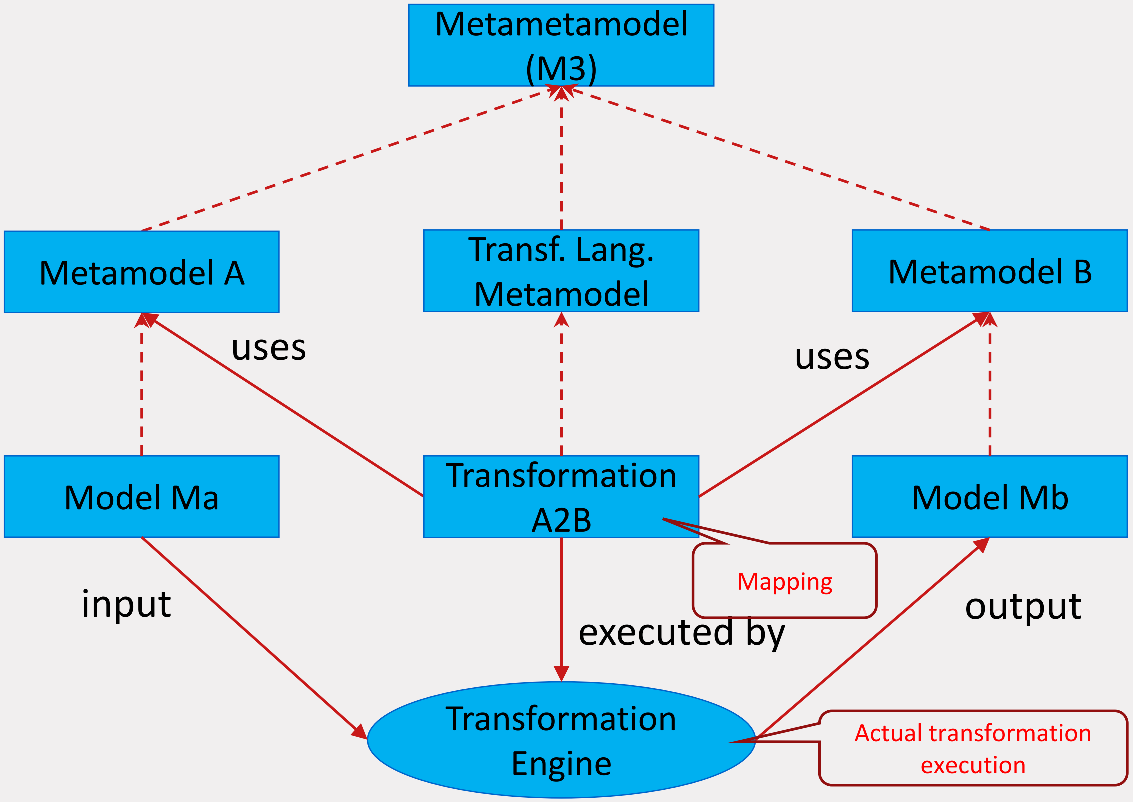
A model transformation takes input and

- changes the input (model update, in-place update), or
- produces output

Model transformations can be implemented in:

- GPL (like Java, Python)
- DSL for model transformations, aka *model transformation language*
  - If model transformations are repetitive tasks, we can benefit from a dedicated language

Model Transformation Pattern



A concrete transformation is defined at the level of metamodels and executed over models conforming to the metamodels

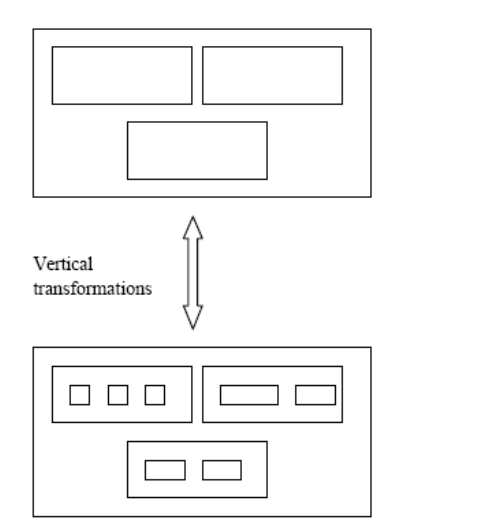
Model Transformation Categories

Two main categories of model transformations

- *Vertical* or *horizontal* (concerning the level of abstraction of the source and target models)
- *Endogenous* or *exogenous* (concerning the source and target languages)

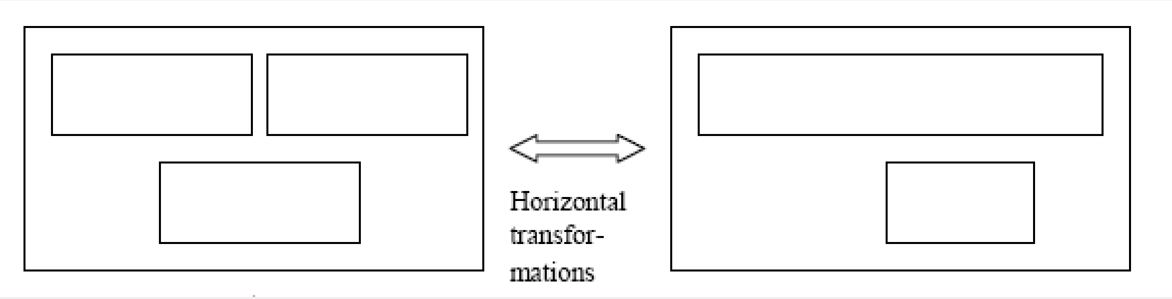
Vertical Transformations

- Source model is at a different level of abstraction than the target model
- High-level to low-level representation or vice versa
- Examples of vertical transformation
  - refinement (specialization)
  - abstraction (generalization)



Horizontal Transformations

- Source model has the same level of abstraction as target model
  - not to be confused with “meta levels”
- Do not preserve the source model structure
- Examples of horizontal transformation
  - refactoring
  - merging



Endogenous and Exogenous Transformations

Endogenous transformations:

- between same metamodels

Exogenous transformations:

- between different metamodels

Summary on Transformation Categories

Taxonomy

	Horizontal	Vertical
Endogenous	*Refactoring*	*Refinement*
Exogenous	*Migration*	*Code generation*

Example Transformation Problem

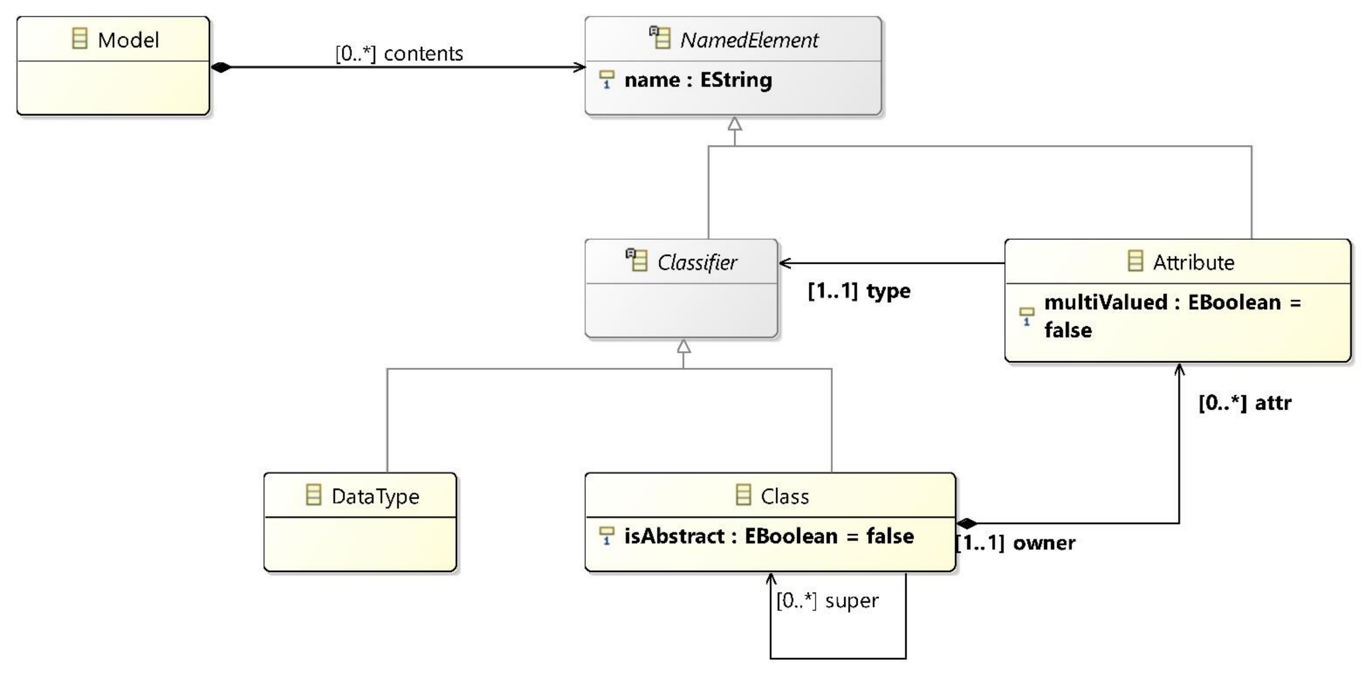
From a given *UML class* model, generate *relational database* schema

- For every class, create a table
- For every class attribute create one or more columns (and tables if needed)
- Take care of multi- and single-valued attributes
- For simplicity only concrete classes are considered, no generalization/specialization

What are the challenges in such a transformation? What are the common tasks in transformations?

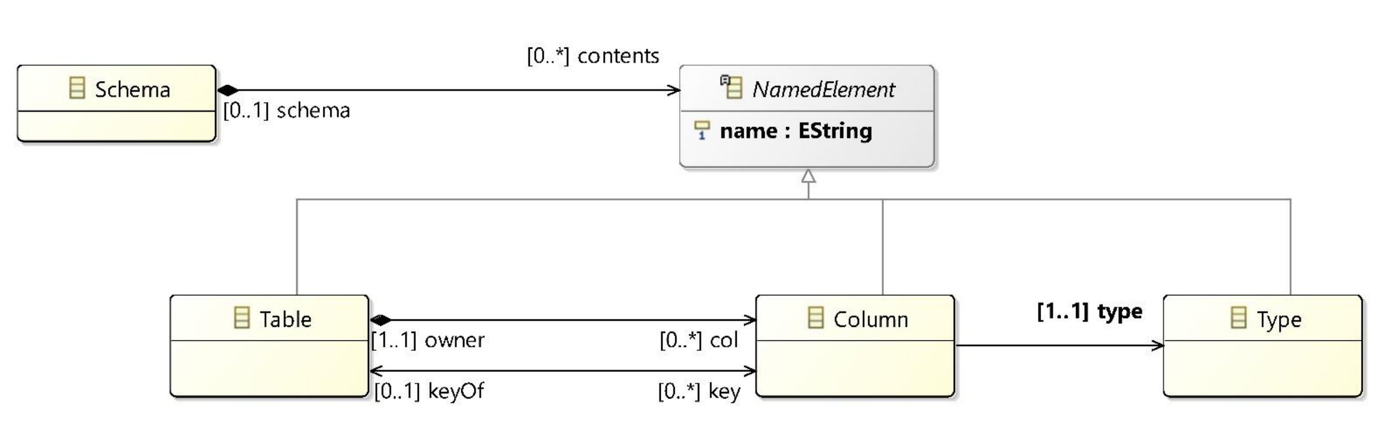
Source Metamodel

Simple language for class models



Target Metamodel

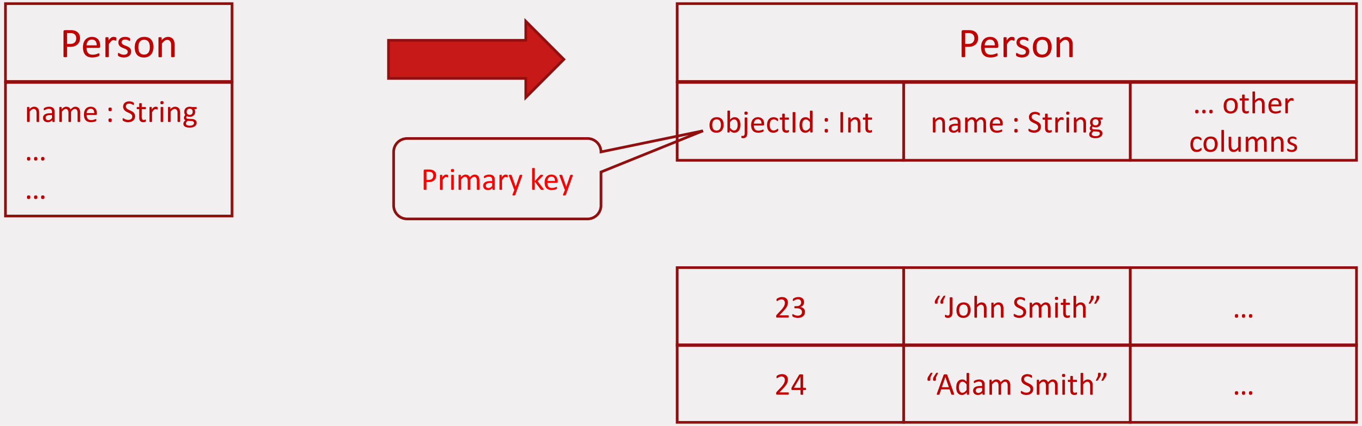
Simple language for relational database schemas



Class to Table

Class *Person* and a single valued attribute of primitive type

Table *Person*, a primary key column (*objectId*) and a column from the single valued attribute of primitive type



Example of mapping classes to tables. Also shows the use of the mapping for single valued attributes of primitive types

Attribute to Column

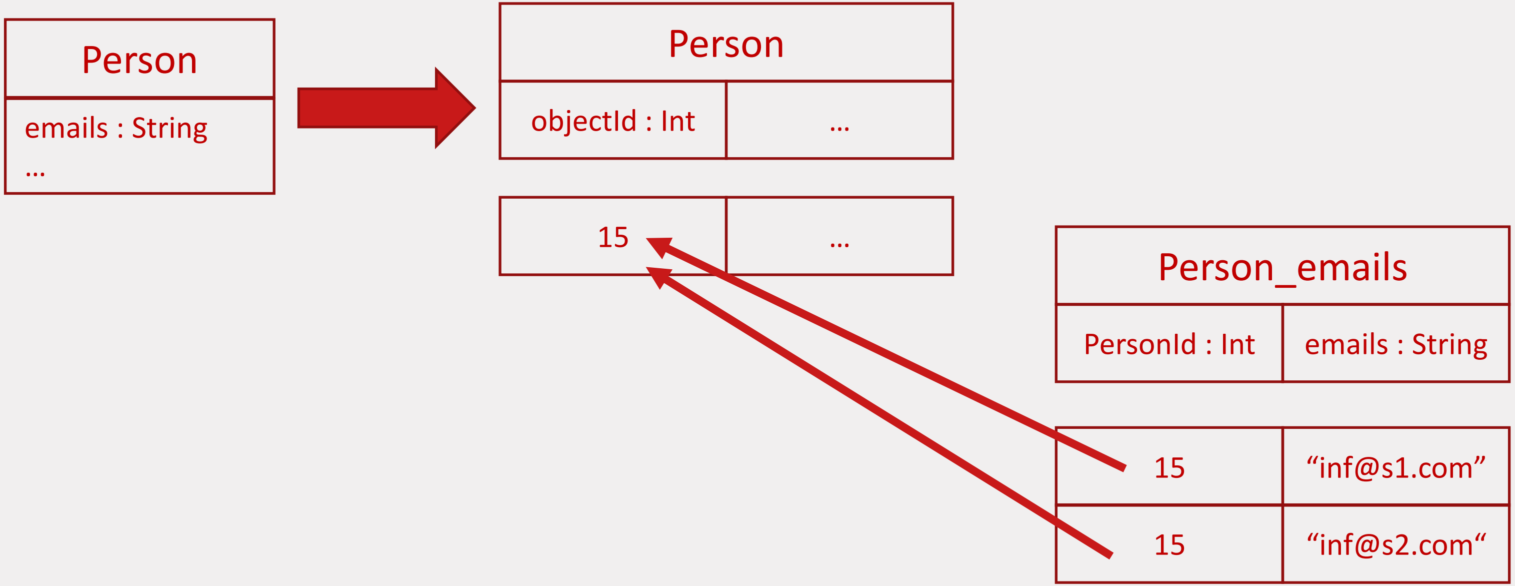
Single valued attribute of class type

Column of type *Int* with values that identify rows in another table

Attribute to Column and Table

Multi valued attribute of primitive type

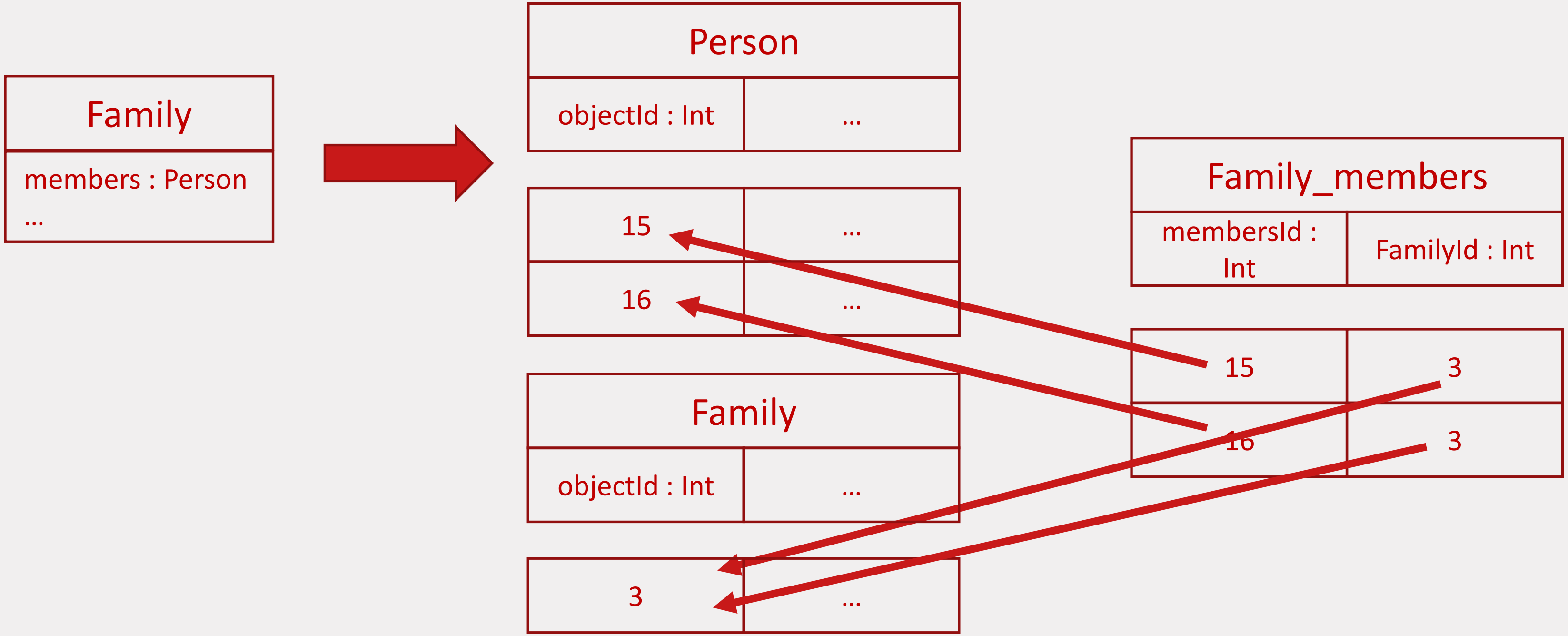
New table with two columns



Attribute to Column and Table

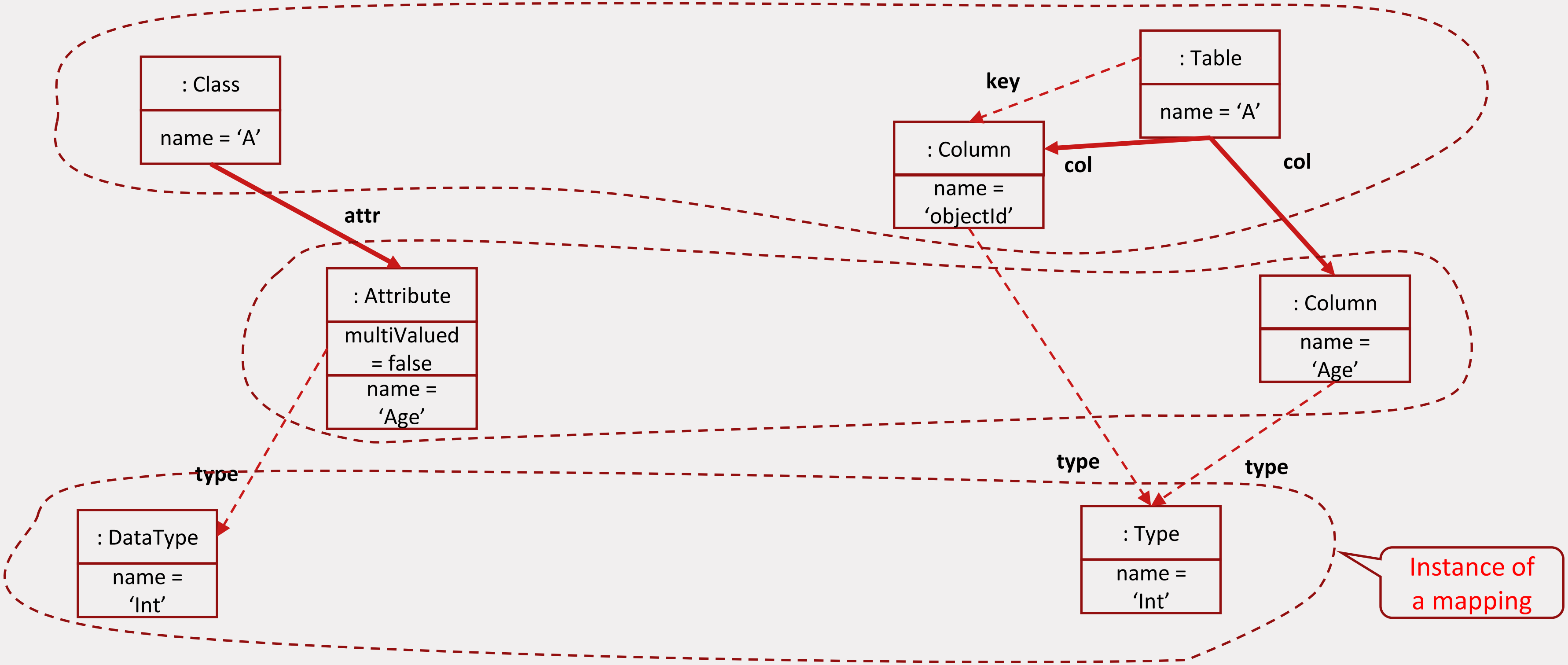
Multi valued attribute of class type

New table with two columns



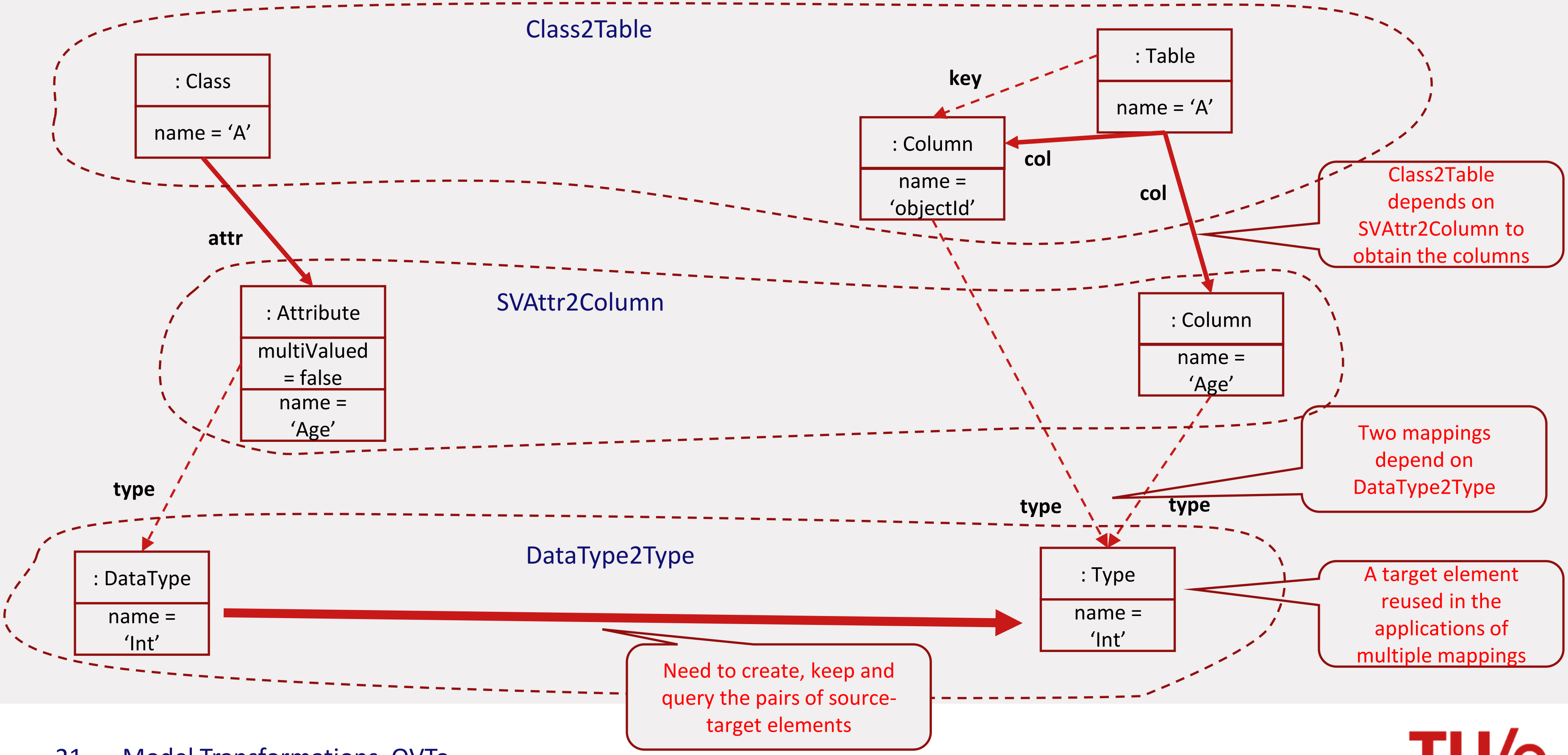
TU/e

Transformation Logic: Example



Occurrences of *patterns* in the *source* model cause instantiation of *patterns* in the *target* model. The relation of source and target pattern is captured in a *mapping*

Mapping Dependencies



Transformation Logic

Pattern occurrences in the source model cause instantiation of patterns in the target model

- Captured in a *mapping* with *source* (from) part and *target* (to) part
- In general, *mapping* (aka *transformation rule*, *relation*) is the main *decomposition unit* in the transformation logic

Mapping *dependencies* reflect the connections among the source and target patterns across two distinct mappings

Mapping *applications* need to be orchestrated:

- Determine the order of matching of source patterns and mapping applications. Sometimes application of a mapping on a source element depends on the result of the application of the same mapping on a different source
- Ensure that when a target object is needed, it is created beforehand (or on demand, or even at a later moment) from the proper source

Transformation Logic

The result of applying a mapping once on a given source pattern can be used multiple times from other mappings

- A tuple of source elements, mapping, and the produced target elements need to be created and maintained
- A set of such tuples is called *internal transformation trace*
- The trace is populated at runtime and can be queried

Implementing a Model Transformation

The mentioned concerns need to be addressed regardless of the way the transformation is implemented

If a GPL is used, the developer is responsible for explicitly:

- encode the identification of pattern occurrences in the input models
- encode the traversal of the source model and the order of mapping applications
- create, maintain and use the internal trace

If a transformation DSL is used, the transformation engine can perform some of these tasks automatically. The language is expected to provide suitable constructs

- possibly deduce an execution order based on dependencies among mappings (mapping scheduling)
- create and maintain the internal trace transparently

Model Transformation Languages

ATL

Xtend

QVT Relations

QVT Operational

QVT Core

Mola

Henshin

Stratego/XT

VIATRA

Tefkat

ETL (Epsilon)

GrGen

…

In the last 15 years a number of model transformation languages have been proposed in the academia and industry

QVT Operational

Query Views Transformations (QVT):

- OMG standard
- Initially three languages: QVT Core, QVT Relations, *QVT Operational (QVTo)*

QVTo features:

- operates on EMF models
- uses OCL for model navigation
- main goal: model modification and transformation
- requires an explicit and complete algorithm for model-to-model mapping

QVT Operational

Main constructs:

- Transformation declaration
- Imperative operations (mappings, helpers, queries, constructors)
- Intermediate data
- Object creation and update mechanism
- Expressions for querying the internal transformation trace

Transformation Structure: UML to Relational

```
modeltype UML uses 'http://glt.tue.nl/uml';
modeltype Relational uses 'http://glt.tue.nl/relational';

transformation uml2relational(in uml : UML, out Relational);

main() {
  uml.rootObjects()[UML::Model]->map model2schema();
}

mapping UML::Model::model2schema() : Relational::Schema {
  contents := self.contents->map classifier2target()->
                 union(getMultiValAttributes()-> map multiValAttr2table())
}
```

Metamodel declarations: metamodels identified by URI and bound to an alias

Transformation signature

Entry point

Mappings, helpers, queries, …

Usage of OCL for navigation

Explicit mapping application on a given source

Entry Point

```
main() {
   uml.rootObjects()[UML::Model]->map model2schema();
}
```

main operation is the entry point for the execution of a transformation.

Typically refers to model parameters in the signature (*uml*) and invokes top-level mappings (*model2schema* in the example):

- For every instance of *Model* in the input model, apply *model2schema*

Mappings

Define a mapping between one or more source model elements into one or more target model elements

- can have zero or more parameters
- the types of the parameters and the result of a mapping can also be OCL types: tuples, collections, primitive values, …

The most common case:

```
mapping <context_classifier>::<name> ( <params> ) : <return_type> {
    <mapping body>
}
```

Case with multiple results:

```
mapping <context_classifier>::<name> ( <params> ) : <id> : <return_type>, <id> : <return_type>, … {
    <mapping body>
}
```

Mappings

Context classifier: the type of the input

Single element will be created of type Column

```
mapping UML::Attribute::singleValAttr2column() : Relational::Column {
  if (self.type.oclIsKindOf(UML::DataType)) {
    name := self.name;
    type := self.type.oclAsType(UML::DataType).map datatype2type();
  } else {
    name := self.name + "Id";
    type := getIntegerType();
  }
}
…
```

name and **type** are features of the result

self is bound to the context source element

```
mapping UML::DataType::datatype2type() : Relational::Type {
  name := self.name;
}
```

Calling Mappings

```
mapping UML::Attribute::singleValAttr2column() : Relational::Column {
  if (self.type.oclIsKindOf(UML::DataType)) {
    name := self.name;
    type := self.type.oclAsType(UML::DataType).map datatype2type();
  }
……
```

Explicit call of another mapping over a **single** context element

```
mapping UML::Model::model2schema() : Relational::Schema {
  contents := self.contents->map classifier2target()->
              union(getMultiValAttributes()-> map multiValAttr2table())
}
```

Explicit call of another mapping over a **collection**. Implicit iteration is performed

Parameter Direction

```
mapping UML::Class::someMapping(in c : UML::Class) : Relational::Table {
  ……
}

mapping UML::Class::someMapping(out c1 : UML::Class, inout c2 : UML::Class) : Relational::Table {
  ……
}

mapping inout  UML::Model::someMapping() : UML::Model {
        name := self.name + '123';
        self.name := result.name +  '456';
}
```

Direction kind

- *in* – object passed for read-only access, the default direction
- *inout* – object passed for update, retains its value
- *out* – parameter receives new value (not necessarily newly created object)

Mapping’s When Clause

When clause:

```
mapping UML::Attribute::singleValAttr2column() : Relational::Column
when { not self.multiValued} {
………
}
```

WHEN clause contains a Boolean condition

Invocation:

- *standard* mode: When-clause acts as a *guard* that filters input parameters

```
a.map singleValAttr2column();
```

Mapping not executed; **null** is returned

- *strict* mode: When-clause acts as a pre-condition that must always hold

```
a.xmap singleValAttr2column();
```

Mapping not executed; Exception thrown

Helpers

```
helper UML::Attribute::clone() : UML::Attribute {

   return object UML::Attribute {
     name := self.name;
     type := self.type.oclAsType(UML::DataType).map datatype2datatype();
   }

}
```

A *helper* is an operation that performs a computation on one or more source objects and provides a result

- May have parameters
- May have side effects

Helpers can be defined in the context of a metamodel classifier and can be called as if they are operations of this classifier

Queries

```
query getMultiValAttributes() : Sequence(UML::Attribute) {
  return UML::Attribute.allInstances()->select(e | e.multiValued)->asSequence()
}
```

A *query* is a helper without side effects

Explicit Object Instantiation

```
mapping UML::Class::class2table() : Relational::Table {
   name := self.name;
   var pk := object Relational::Column {name := 'ObjectId'; type := getIntegerType();};
   col := Sequence{pk}->union(self.attr->select(e | not e.multiValued)->map singleValAttr2column());
   key := Sequence{pk};
}
```

A metamodel class can be instantiated using the **object** keyword

Observe the usage of a local variable *pk* (**var** declaration)

Mapping Body: Full Details

General form:

```
mapping UML::Class::myMapping() : Relational::Table {
        init {
           var tmp := self.map otherMapping();
           if (self.name = 'AAA') then {
              result := object Table {};
           }
        }
        population {
           object result : Table {
              name := self.name;
           }
        }
        end {
           assert (result.name <> null);
        }
}
```

Optional **Init** section: computation *before* the instantiation of outputs

Implicit **Instantiation** section: instantiation of out parameters

Population section: assignment of features of outputs

End (termination) section: computation before exiting the body

Predefined variables in mappings:

- *self* – refers to the context
- *result* – refers to the result

TU/e

Mapping Body

Mapping body without *population* keyword:

```
mapping UML::Class::myMapping() : Relational::Table {

        init {
           var tmp := self.map otherMapping();
           if (self.name = 'AAA') then {
              result := object Table {};
           }

        }

        name := self.name


        end {
           assert (result.name <> null);
        }

}
```

Direct access to properties of the result within the population section without the 'population' keyword!

Disjuncts

```
mapping UML::NamedElement::classifier2target() : Relational::NamedElement
disjuncts UML::DataType::datatype2type, UML::Class::class2table{}

mapping UML::DataType::datatype2type() : Relational::Type {
  name := self.name;
}

mapping UML::Class::class2table() : Relational::Table {
  name := self.name;
  ……
}
```

- Applying *classifier2target* to a data type will invoke *datatype2type*
- Applying *classifier2target* to a class will invoke *class2table*

When a mapping based on *disjunction* of other mappings is invoked:

1. The actual parameter values are checked for compatibility with the disjuncted mappings
2. when-clauses of the disjuncted mappings are evaluated (if any)
3. If no parameter match is found and/or no when-clause is true, null is returned
4. Otherwise, the first matching mapping with a true when-clause is executed

Internal Trace and QVTo

The internal trace of the QVTo execution engine:

- contains information about mapped objects during transformation execution
- consists of trace records

A trace record is created when a mapping is executed

- trace records keep reference to the executed mapping and the mapping parameter values
- a trace record is created after the implicit instantiation section of the mapping is finished

Trace

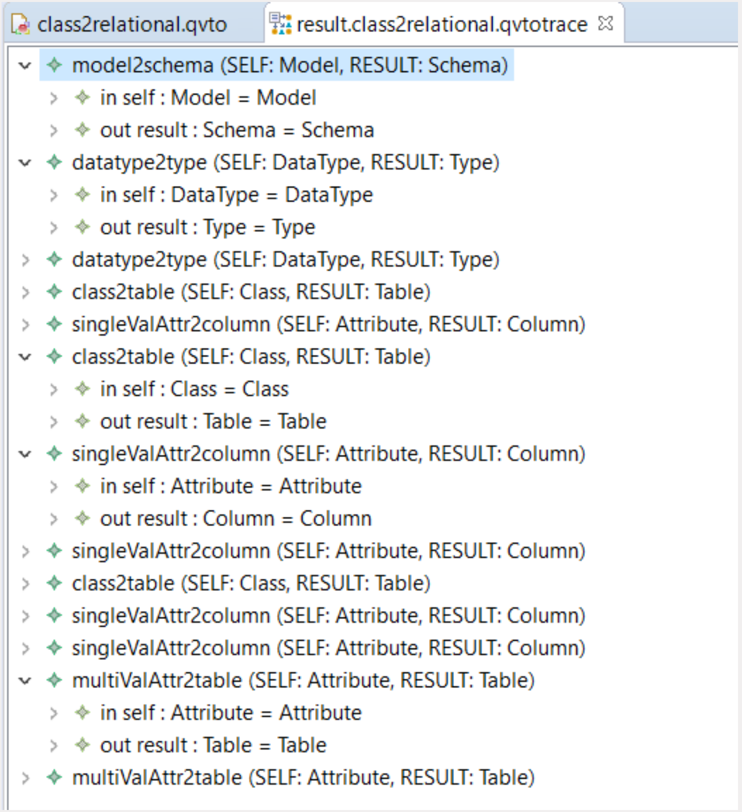
mapping	param1	param2	...
mapping	param1	param2	...
mapping	param1	param2	...
mapping	param1	param2	...

Usage:

- Prohibits duplicate execution with the same parameters
- Used in *resolve* expressions
- May be serialized after the transformation execution

Internal Trace and QVTo

The trace can be serialized as a model after the transformation execution



mapping	param1	param2	...
mapping	param1	param2	...
mapping	param1	param2	...
mapping	param1	param2	...

Internal Trace and QVTo

```
mapping UML::DataType::datatype2type() : Relational::Type {
  name := self.name;
}

…

mapping UML::Attribute::singleValAttr2column() : Relational::Column {
  if (self.type.oclIsKindOf(UML::DataType)) {
    name := self.name;
    type := self.type.oclAsType(UML::DataType).map datatype2type();
  }
…

query getIntegerType() : Relational::Type {
  return ……… .map datatype2type();
}
```

Mapping *datatype2type* can be called multiple times over the same source object

- For example, multiple attributes may be of the same type
- We want to reuse the Relational model type

When a mapping is invoked, the internal trace is consulted first:

- If *no record exists* for the mapping and the input, the mapping is executed for the first time and a trace record is created
- If *a record exists* for the mapping and the input, the already produced result is returned (no repetitive execution for the same context)

Tip: if you want a given source element to be processed multiple times, every time producing a new result:

- Use a helper

Or

- Pass an unique artificial parameter that will cause a new trace record to be created for every visit

Querying the Internal Trace in QVTo

A *resolve expression* is an expression that inspects trace records to retrieve source or target objects which participated in the previous mapping executions

- **resolve** – resolves target objects for a given source object
- **inv** (***invresolve***) – inversed resolve. Resolves source objects for a given target object
- **one** (***resolveOne***) – returns one target object for a given source
- **in** (***resolveIn***)– inspects trace records for a given mapping only
- **late** (***late resolve***) – performs resolution and assignment to some model object property after the transformation execution

TU/e

QVTo Expressions and Statements

QVTo and OCL:

- Assignments
- Variables
- Loops (while, forEach)
- Loop interrupt constructs (break, continue)
- Conditional execution workflow
- Convenient shorthand notation
- Mutable collections

Other Features

- Abstract mappings
- Implicit disjuncts (in effect, mapping overloading)
- Mapping inheritance
- Libraries
- Intermediate structures
    - Definition of classes that reside outside of the metamodels but can be used within a transformation

Summary on QVTo

- Part of the OMG QVT standard
- Implementation based on Eclipse and EMF
- Imperative language
    - Mappings have imperative body
    - Mappings are not scheduled for execution by the transformation engine
    - Traversal of the input models is explicit
    - Order of mapping application is explicit

Resources on QVTo

Eclipse web site:

- https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

OMG official standard

- https://www.omg.org/spec/QVT/About-QVT/
- https://www.omg.org/spec/QVT/1.3/PDF (see Chapter 8)