TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Domain Specific Language Design (2IMP20)

Metamodeling. Model Validation. OCL

Loek Cleophas, Ivan Kurtev

Agenda

Metamodeling

- Metamodels, metalanguages, metametamodels
- Standards: MOF, Ecore
- Metamodeling architectures

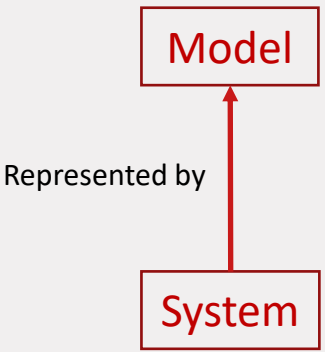
Model validation

Object Constraints Language (OCL)

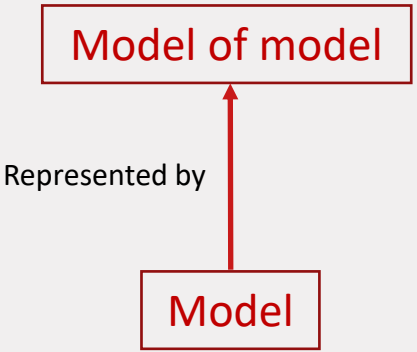
Metamodeling

In metamodeling, the entities being modeled are models, or more specifically, models expressed in a given modeling language

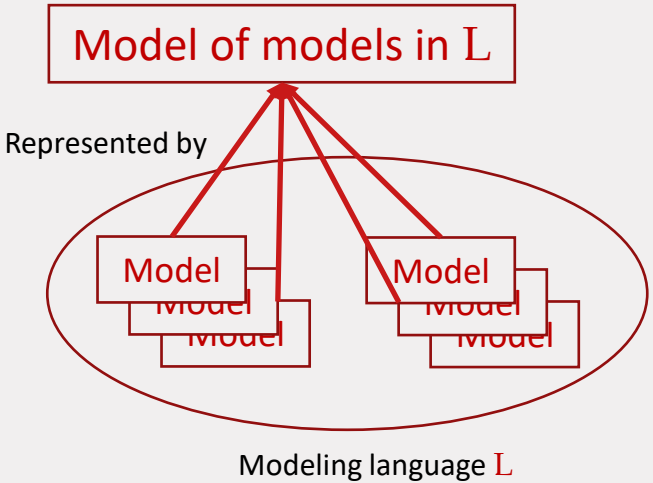
- modeling language understood as a set of models
- in the first lecture, we were actually doing metamodeling



General relation between a model and a system being modeled



A model can be a subject of modeling (enabling model of a model)



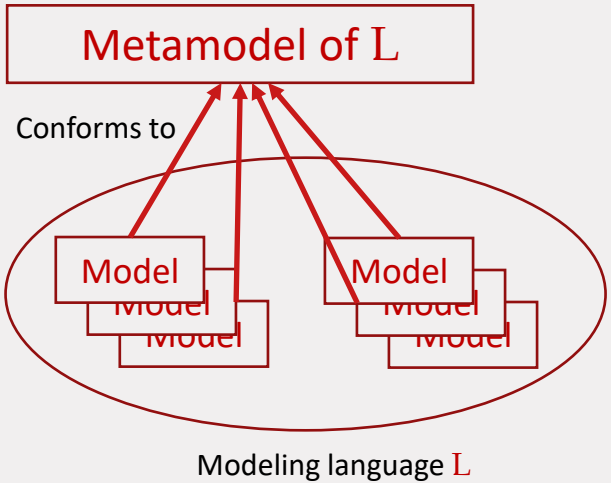
Modeling language L

TU/e

Metamodel

Metamodel:

- The model of all models belonging to a language
- Definition of the *abstract syntax* of a language (the language concepts and the relations among them)
- Captures the universal properties shared by all models and their model elements



Conforms to:

- The relation between a model and its metamodel

Metamodel and Grammar

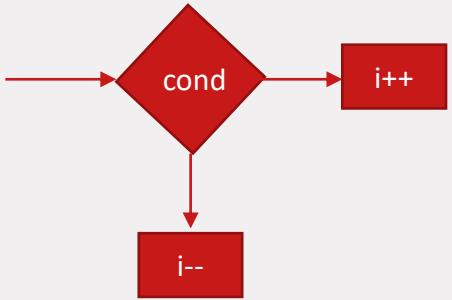
Metamodels play a role similar to the role of a grammar for a language definition

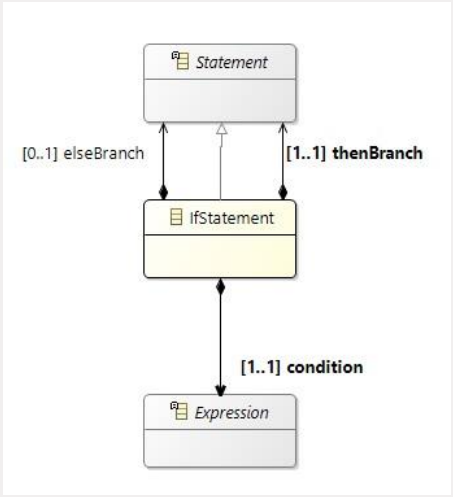
- By instantiating the classes in the metamodel, models in the language can be created (compare to the application of grammar rules to obtain a program)
- For a given model, the check if the *conformsTo* relation holds determines if the model belongs to the language (compare to the notion of *acceptor* and *parser* in Grammarware)

Abstract and Concrete Syntax

```
if(cond){
  i++
} else {
  i--
}
```

```
if cond then
  i++
else
  i--
endif
```






```
Stm ::= if(Exp, Stm, Stm) | …

Exp ::= true | false | Exp and Exp | …
```

Concrete syntax:

- Definition of the concrete appearance of sentences/models
- Can be textual, diagrammatical, tabular,…, a combination of them

Abstract syntax:

- Definition of the structure of the language elements regardless their appearance in a concrete syntax
- Can be expressed in many formalisms (recall *Rascal approach* based on data types)

TU/e

The Role of Metamodel in Language Definition

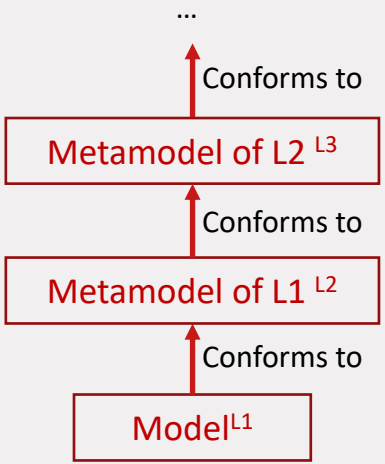
In the Modelware approach to **DSL development**, the language **metamodel is the key** artefact

- Start the language development by first defining the metamodel
- An explicit capturing of the domain

This is one of the main takeaways from the Modelware part of the course

Metamodeling Languages

- A metamodel is itself a model and can be expressed in a language
- This language may have a metamodel
- … repeat the two lines above



Where do we stop?

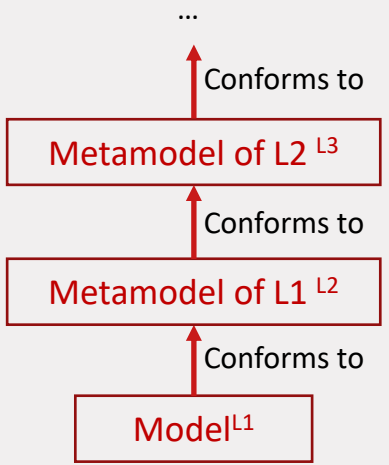
M L Model *M* expressed in language *L*

TU/e

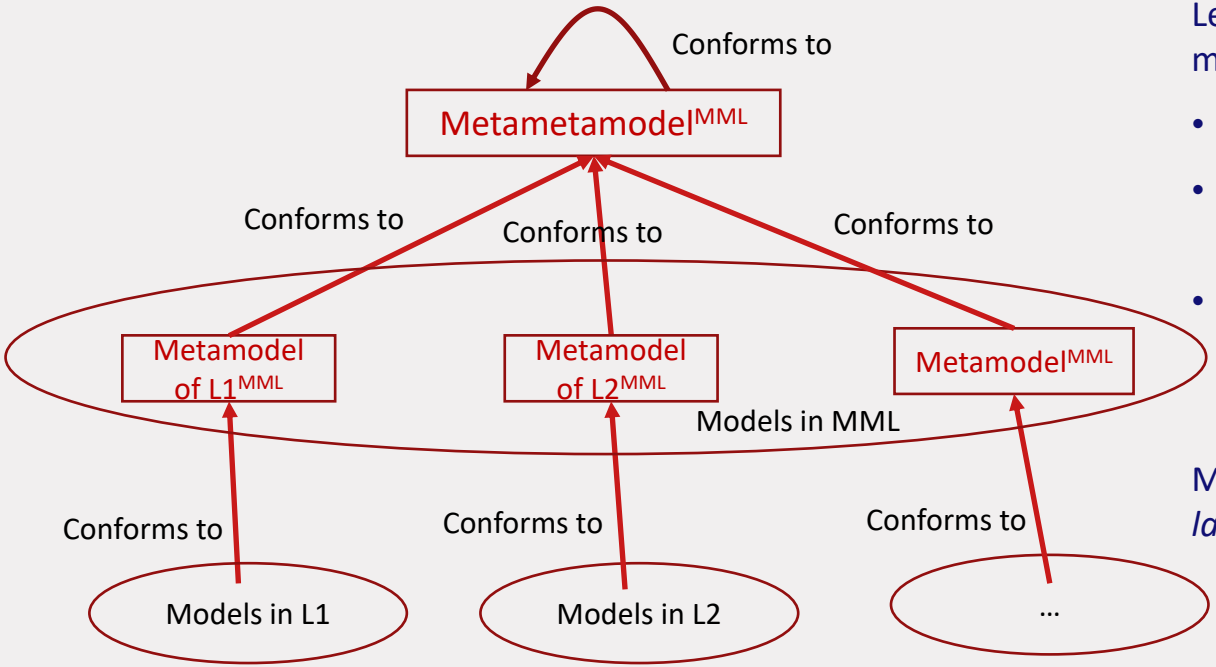
Metamodeling Languages

Where do we stop? The options are:

- Stop at a certain level, do not give an explicit metamodel (example further in the lecture)
- Continue *ad infinitum* (impractical)
- Make a *self-reflective* 'conforms to'



Metamodeling Languages



Let's denote as *MML*, a DSL for metamodels

- Metamodels are expressed in MML
- MML metamodel can therefore be expressed in MML
- The metamodel of MML is called *metametamodel*

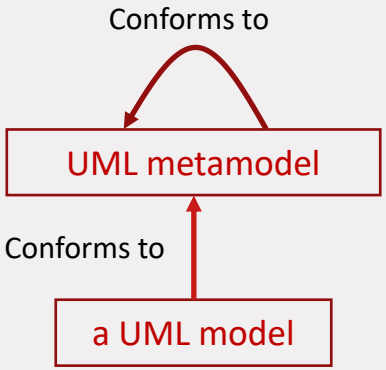
MML is regarded as *metamodeling language*

Metamodeling Languages: MOF

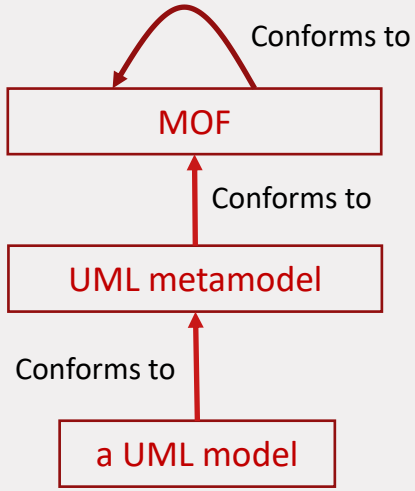
- The concept of *metamodeling language* is recognized in many technologies
- In the area of Model Driven Engineering, a standard called *Meta Object Facility* (MOF) is defined and serves as metamodeling language
- MOF is standardized by Object Management Group (OMG), the organization that manages the development of UML and many other MDE related technologies
  - http://www.omg.org/
  - Current version of MOF: 2.5.1 (as of October 2016)

Evolution of MOF

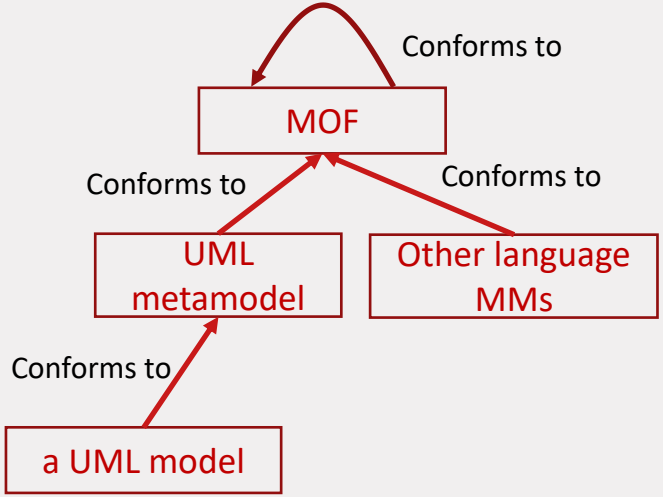
First, it was recognized that UML abstract syntax can be defined in UML just using class diagrams



A simplified version of class diagrams was extracted to form the MOF

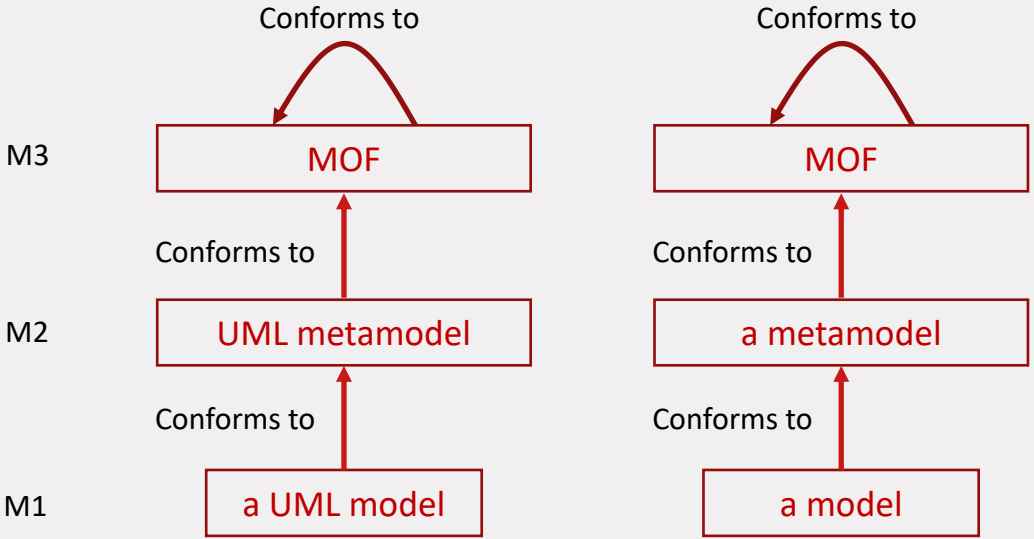


MOF started being used to express the metamodels of other languages (e.g. OCL, QVT, …)



TU/e

MOF Metamodeling Architecture



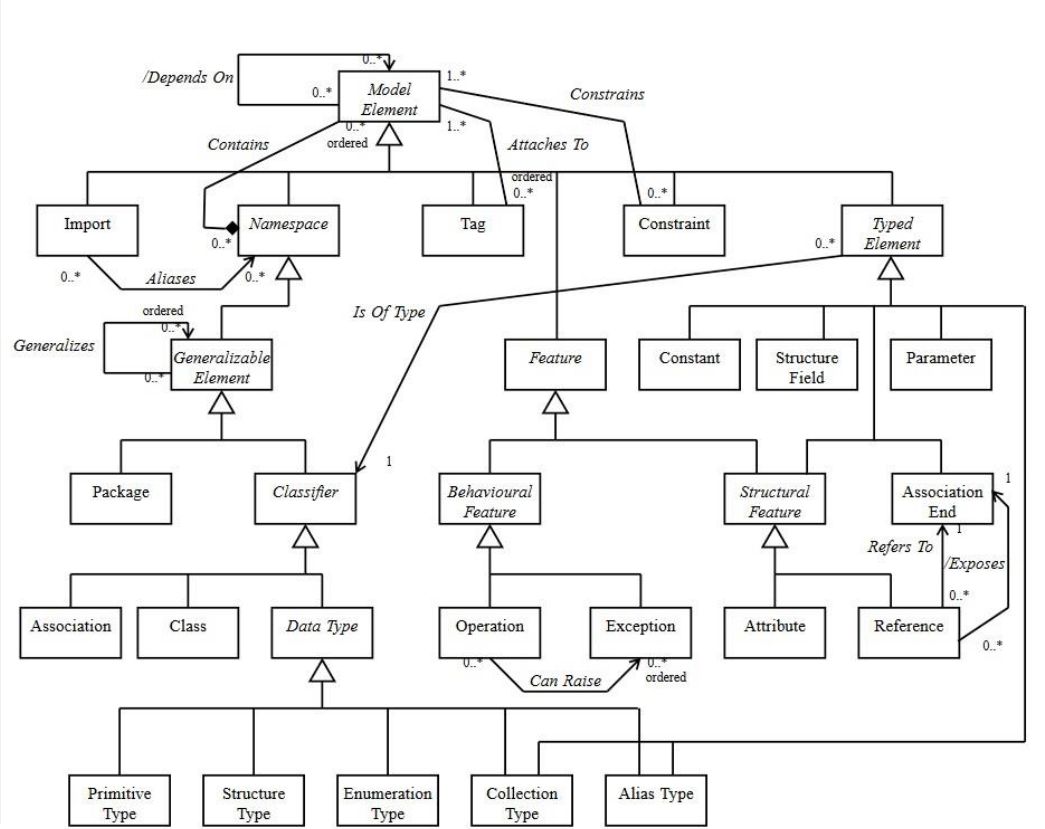
Three-layers metamodeling architecture:

- M1: the level of models
- M2: the level of metamodels
- M3: the metametamodel (MOF)

Unification principle:

- Everything is a model expressed in a language and conforms to a metamodel of that language
- All models can be treated uniformly with a single set of tools

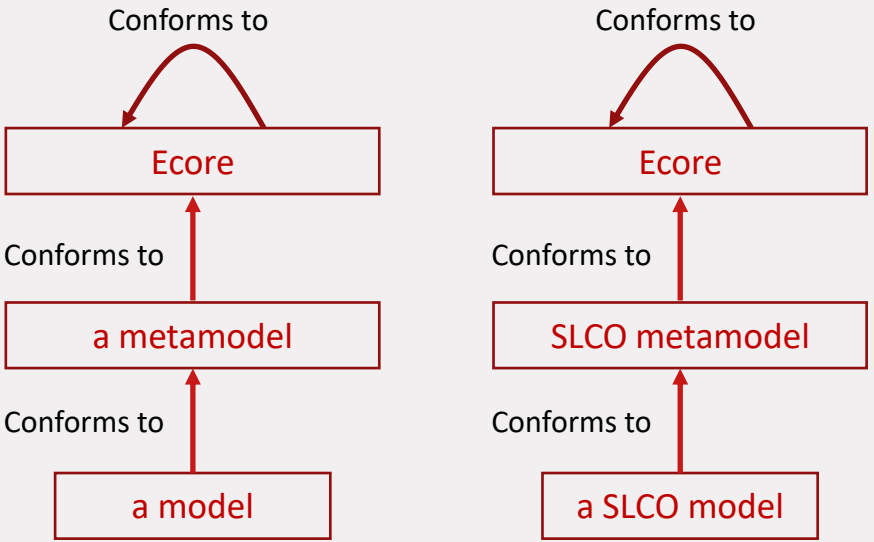
Looking into MOF



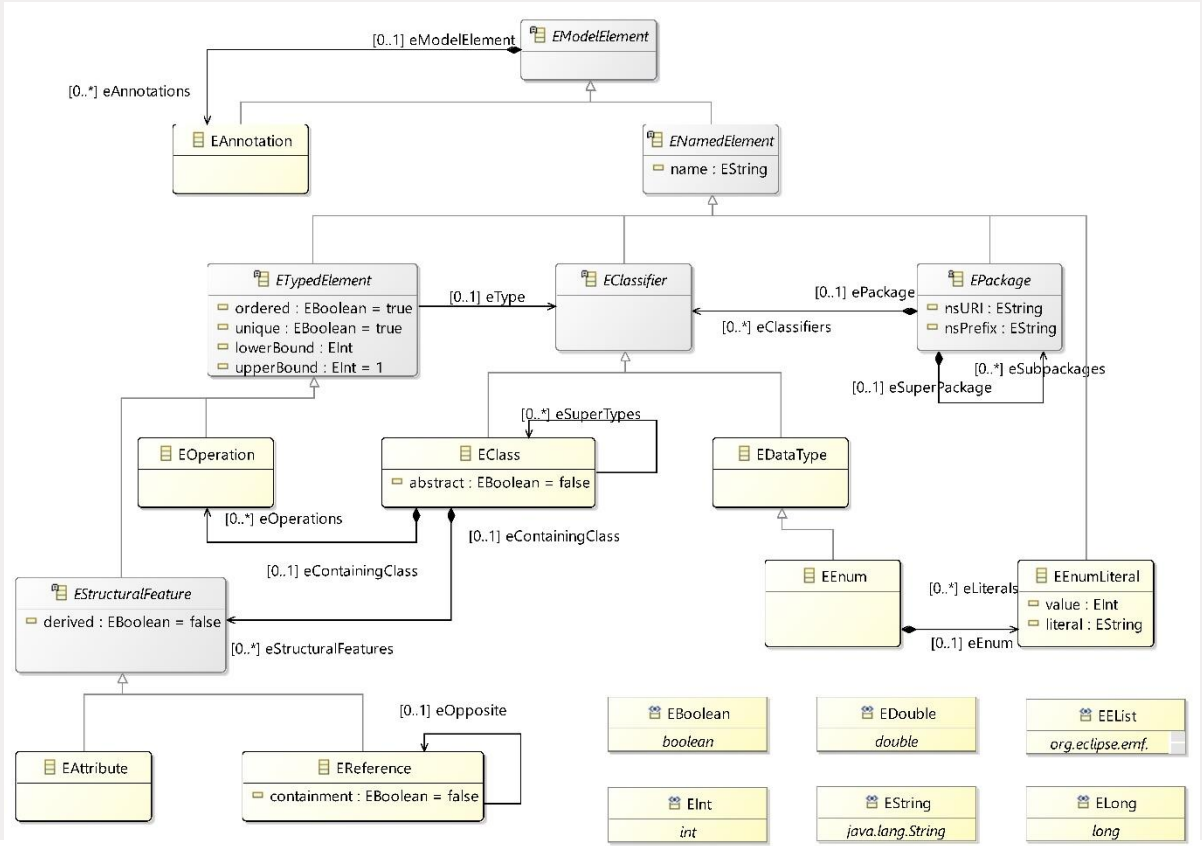
- The diagram shows MOF 1.4
- MOF 2.x is much more complex, aligned with UML 2.x
  - Beyond the scope of this lecture

MOF and Ecore

- Ecore is the metametamodel/metalanguage of EMF (used in the previous lecture)
- All EMF metamodels conform to Ecore
- Ecore is an implementation of MOF
  - Closer to MOF 1.4 rather than MOF 2.x



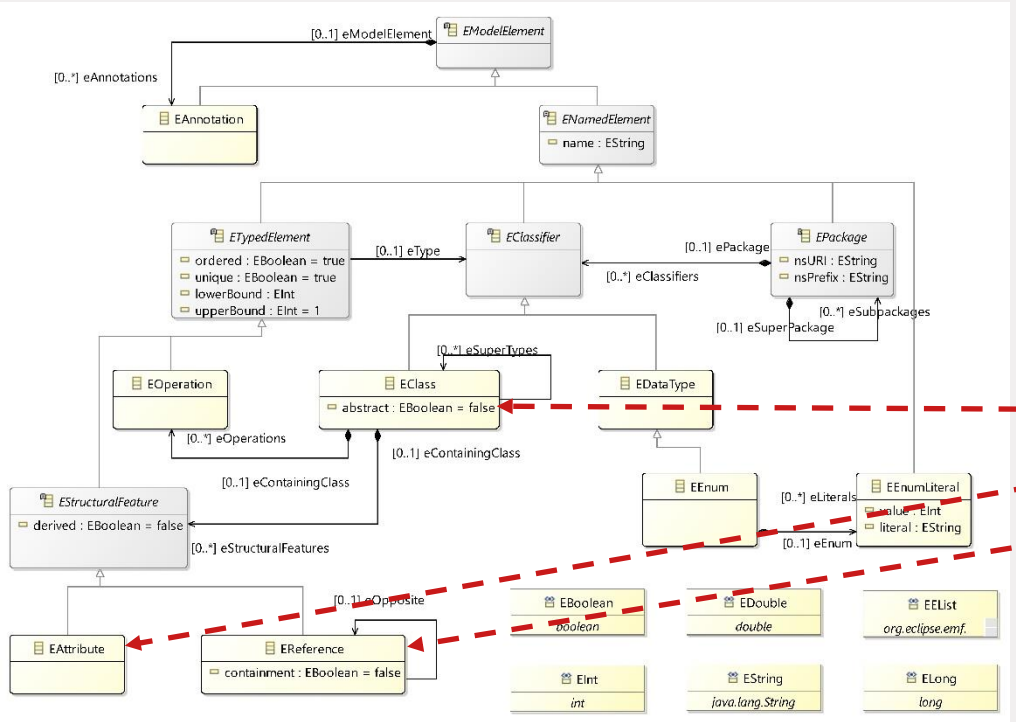
Ecore Contents



- These are the Ecore main classes
- Some details are skipped (e.g. class operations, EObject class, Resource factories)

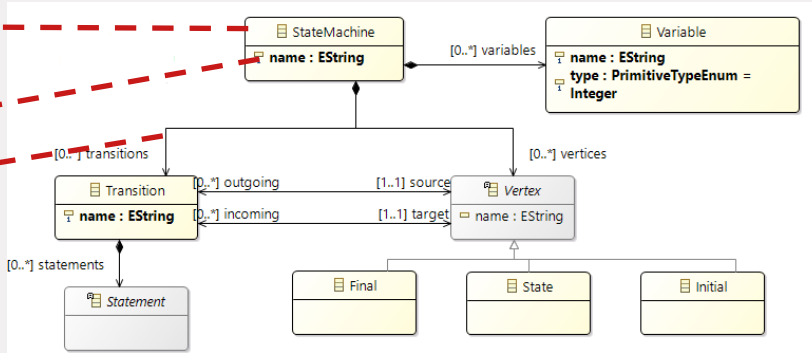
Compare this to the MOF 1.4 diagram shown previously

Ecore Metamodels



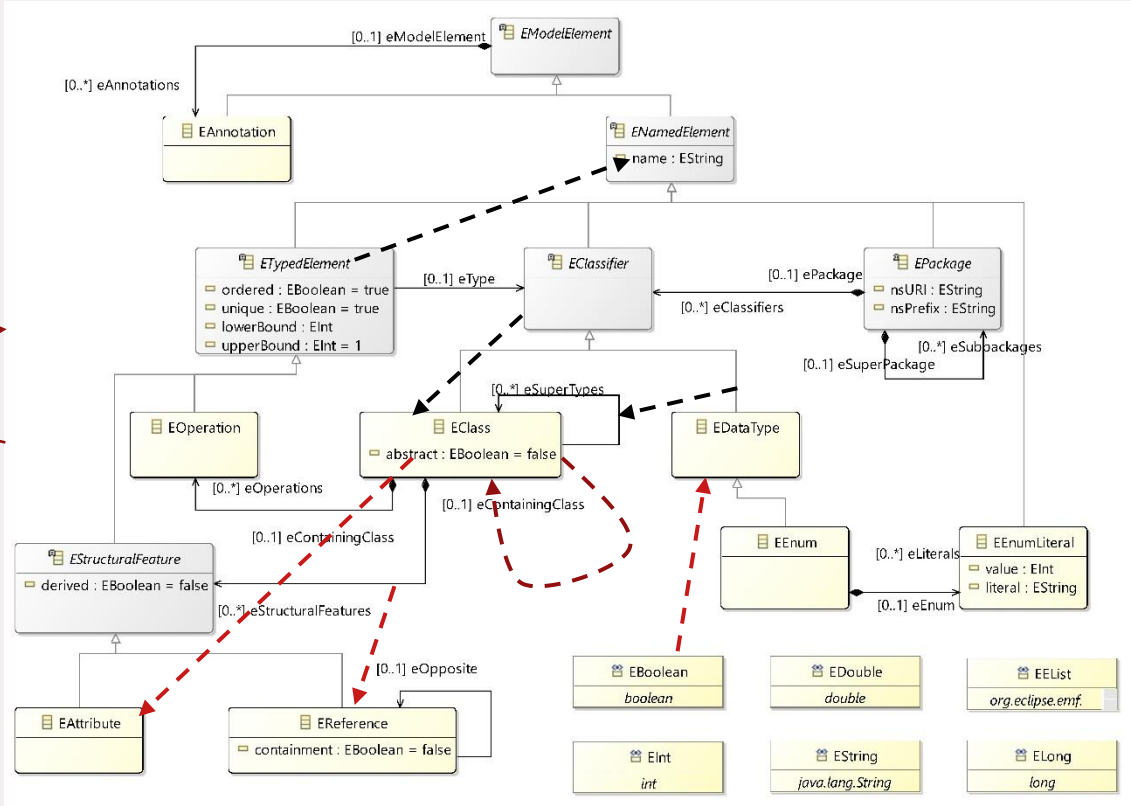
Elements of an Ecore metamodel instantiate elements of Ecore (illustration based on a fragment of SLCO metamodel)

Instance of



Reflective “conforms to” Relation

Conforms to



All elements of Ecore are instances of some element in Ecore

Note:

- Observe that some elements are types (in the sense of instantiable) and others are not (they are individuals)
- Example: the link from a class to its superclass is actually an instance of the reference *eSuperTypes* defined in *EClass.* Both are represented as lines but have rather different semantics

Formalization of Models and Metamodels

- Models can be formalized as graphs
- *Conforms to* can be defined based on a function that maps elements from one graph (a model) to elements in another graph (the metamodel)

Definition 1 (Graph): directed multigraph is a triple $G = (N_G, E_G, \Gamma_G)$, where N_G is a finite set of nodes, E_G is a finite set of edges, $\Gamma_G : E_G \rightarrow N_G \times N_G$ is a function that maps edges to the pair of source and target nodes.

Note: *multigraph* is a graph that permits more than one edge between two given nodes

Note: observe that the graphs are not labeled

Formalization of Models and Metamodels

Definition 2 (Model): a model is a triple $M = (G, \omega, \mu)$, where:

- $G = (N_G, E_G, \Gamma_G)$ is a directed multigraph,
- ω is a model called *reference model* for M, associated to a graph $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$
- $\mu : N_G \cup E_G \rightarrow N_\omega$ is a function mapping nodes and edges of G to nodes of G_ω

The relation between a model and its reference model is called *conformsTo*.

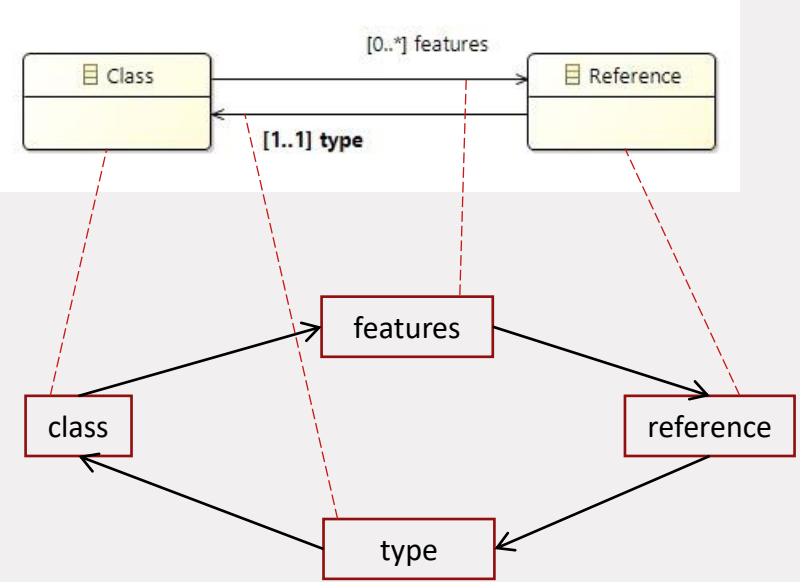
Formalization of Models and Metamodels

- Elements of ω are called *metaelements*
- μ is not injective: several model elements can be mapped to the same metaelement
- μ is not surjective: not all metaelements need to be associated to a model element

Definition 3:

- *Metametamodel* is a model that is its own reference model
- *Metamodel* is a model such that its reference model is a metametamodel
- *Terminal model* is a model such that its reference model is a metamodel

Example: Minimal Metametamodel



The graph representation of the model given in the diagram and the correspondence to the graphical elements. Nodes have labels for illustrative purposes.

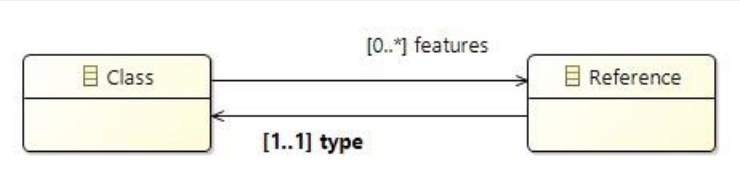
In the example, metamodels contain only classes and directed binary associations among classes.

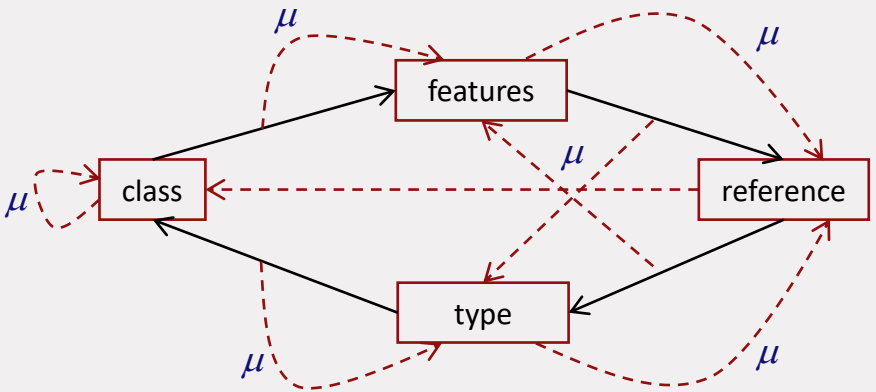
For simplicity we do not define attributes, primitive types, grouping constructs, …

Multiplicity and role names of references is supported but not illustrated in the example

TU/e

How Does a Metametamodel Conform to Itself?





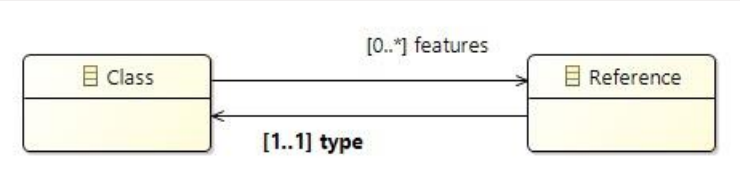
μ is defined as follows:

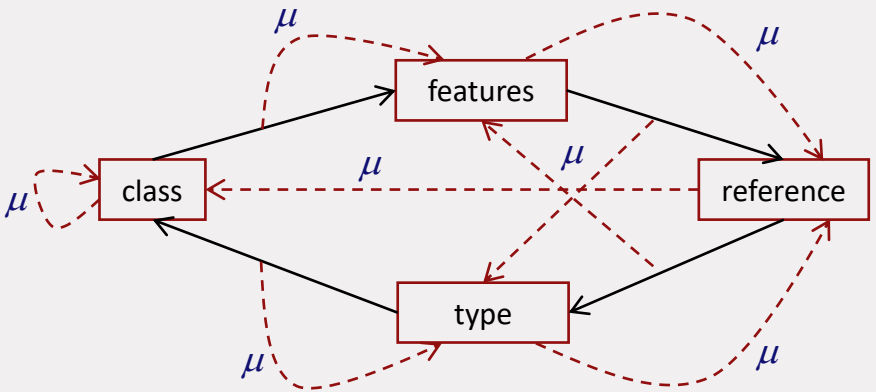
class → class
reference → class
features → reference
type → reference

(class, features) → features
(features, reference) → type
(reference, types) → features
(type, class) → type

Dashed arrows represent the mappings in μ (the *instanceOf*)

How Does a Metametamodel Conform to Itself?





μ has to obey the conditions:

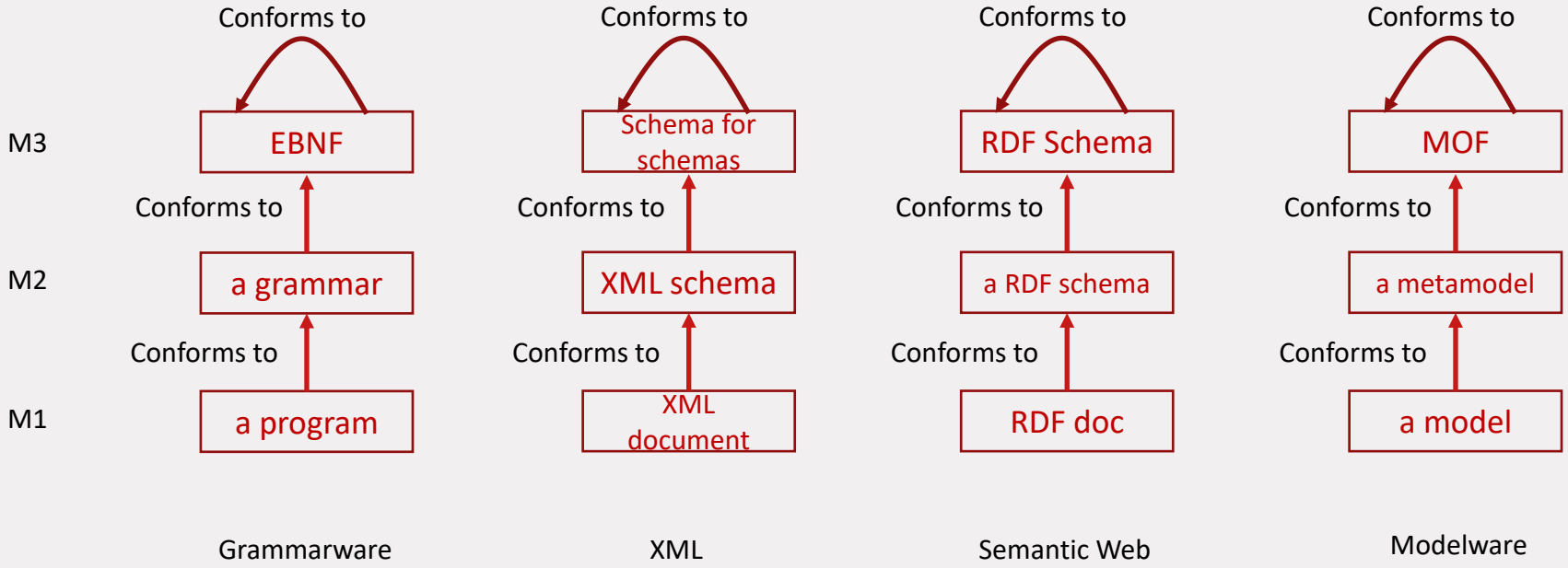
- If $\mu(x)$ = class then for all edges (x, y): $\mu(x, y)$ = features and $\mu(y)$ = reference
- If $\mu(x)$ = reference then exists unique edge (x, y) and $\mu(x, y)$ = type and $\mu(y)$ = class

In essence, this defines the semantics of instantiation relation

- A metamodeling language defines the conditions for μ

It is fairly easy to extend the example by defining generalization relation among classes

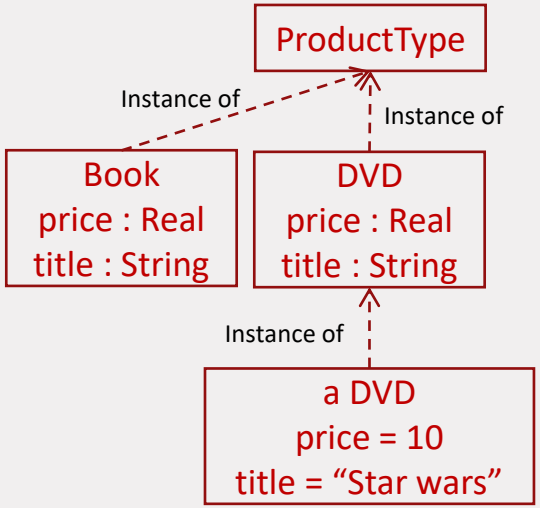
Metamodeling Architectures in other Technologies

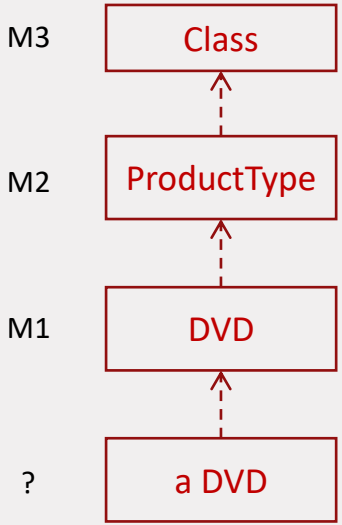


The three-layers organization is present in several other technologies. From a certain perspective, XML and Semantic Web can also be regarded as frameworks that allow definition of DSLs

Some Challenging Modeling Problems

Imagine a language where users can define their own product types and products instances of these type. We also want to state that all product types has attributes *price* and *title*





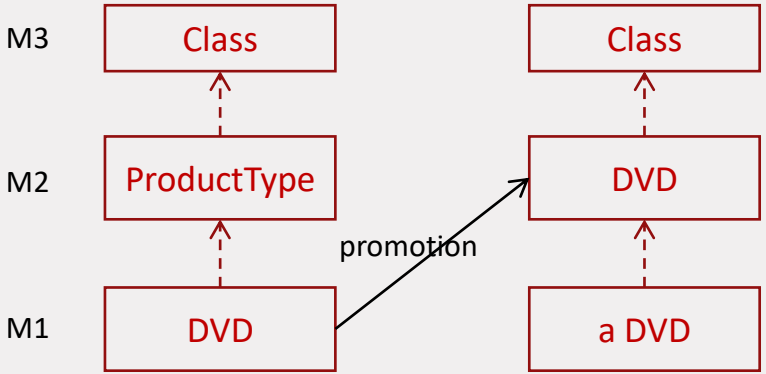
Starting from the fixed metametamodel at M3, we can define language elements at M2 and instantiate them at M1

Where do we allocate the concrete DVD?

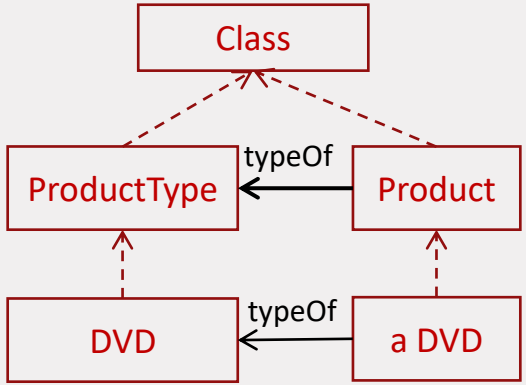
The three-layered architecture can’t handle this directly

Possible Solutions

Define a DSL for product types, then transform a model with user types to a metamodel. This strategy is called *model promotion* (model at M1 is promoted to a metamodel at M2). We know that elements in M1 are types but they can't be instantiated further, thus promotion needed



Type-Object pattern: in the DSL metamodel, define metaclasses for user types and their instances. Then user types and their objects will be just ordinary model elements at M1 level



Possible Solutions

Promotion approach:

- an extra transformation needs to be defined
- every time a new user type is introduced, the promotion must be performed. In effect this results in a new version of the DSL

Type-Object pattern:

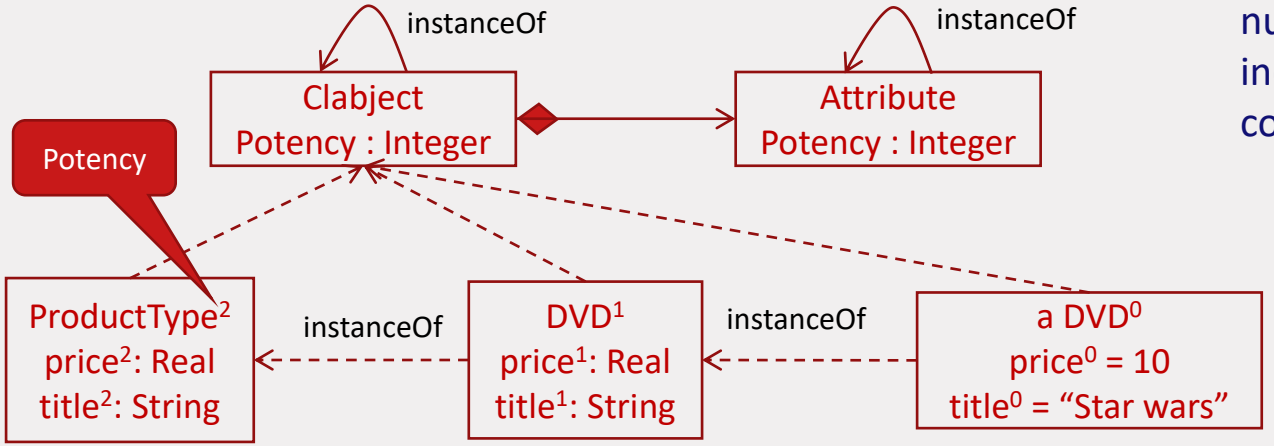
- Concrete types and objects are just instances at M1. Their meaning is not explicit
- *typeOf* relation is just a normal reference. Tools need to implement its meaning (in essence, the meaning of instantiation)

Examples like this demonstrate some limitations in the 3-layers metamodeling architectures

Multi-level Modeling

An approach called *multi-level modeling* allows an arbitrary number of instantiations to be present in user models. An element is at the same time a *class* that can be instantiated, and an *object* that is an instance of a class.

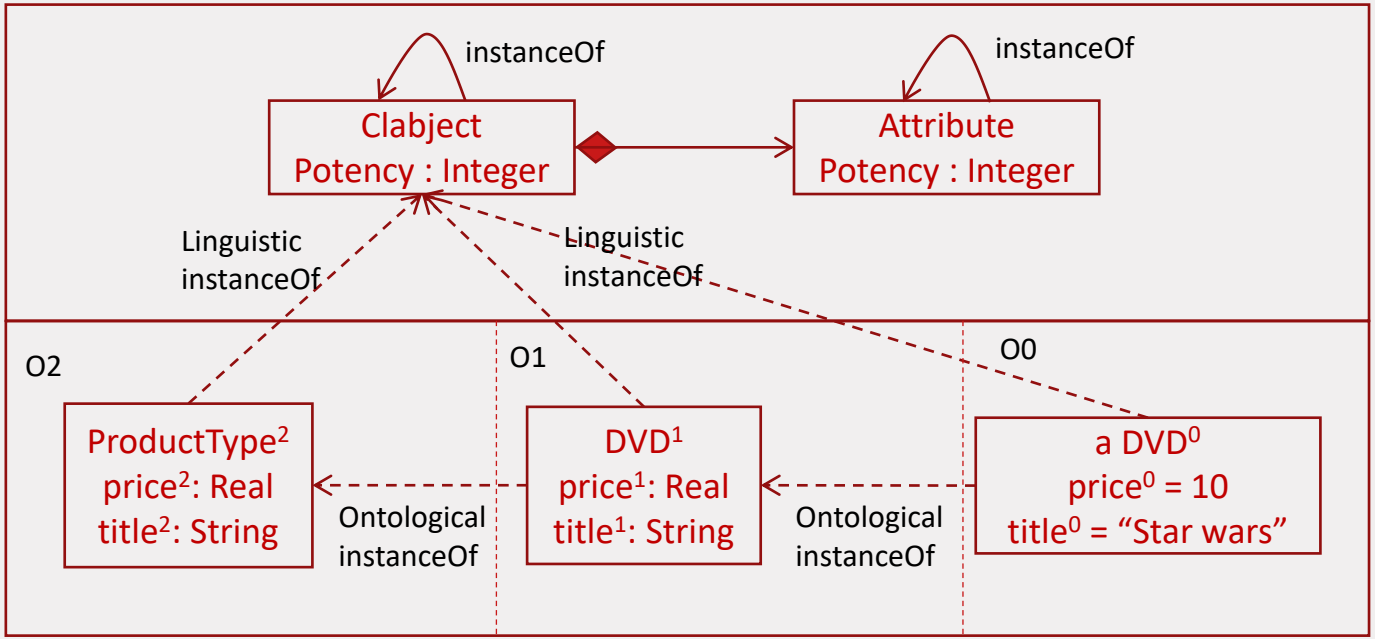
Such elements are called *Clabjects*



Clabjects and attributes are assigned a non-negative number called *potency* that indicates how many times the construct can be instantiated

TU/e

Multi-level Modeling

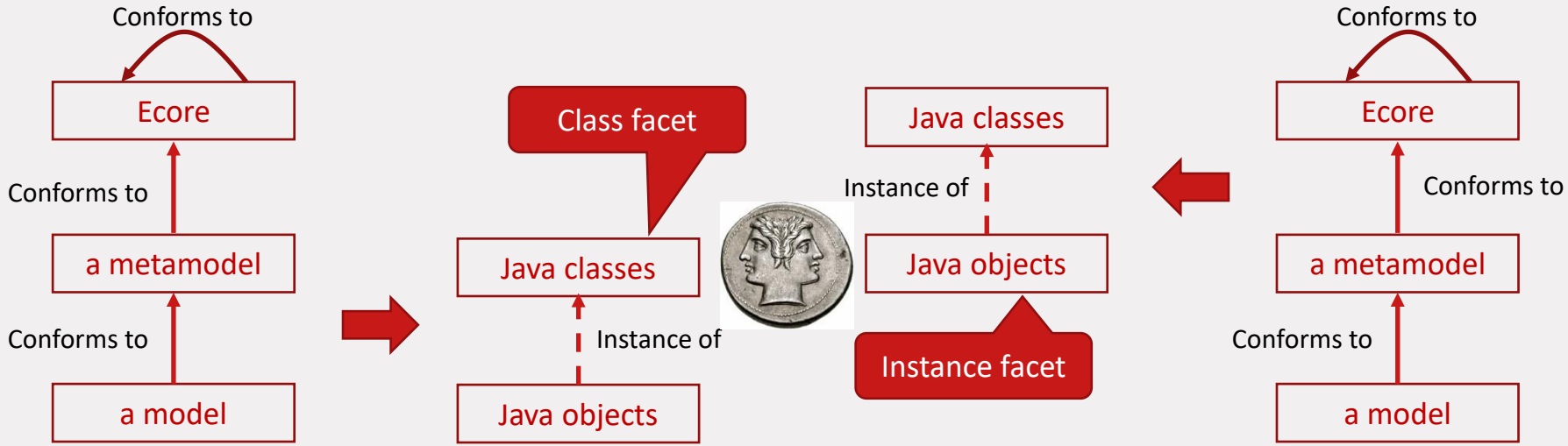


- Instantiation between M1 and M2 is called *linguistic instanceOf*
- Instantiation between clabjects (at M1) is called *ontological instanceOf*
- We may have arbitrary number of levels at M1, stratified over ontological instantiation: O0, O1, …

- The metamodeling architecture has two levels (M2, M1)
- M2 is not necessarily self-conformant
- DSL metamodels and models are defined in M1

TU/e

Ecore Revisited



Using the EMF model code generator, every metamodel that *conforms to Ecore* can be mapped to Java classes that implement it. Models are implemented as instances of these classes

Since *Ecore conforms to itself*, we can derive Java classes from Ecore in the same way. They are located in package *org.eclipse.emf.ecore* and allow dynamic creation of metamodels in Java (as objects!). This also illustrates the *clabject* nature of all metamodel elements

Summary

- Three-layers metamodeling architectures are observed in several technologies
- EMF Ecore follows the three-layers pattern and is de facto an industrial standard for model-based DSL development
- The metameta level is essential and captures the fundamental features of all models
  - Uniform model representation (objects, properties and classes)

- We presented a formalization of models and metamodels based on graphs. Other formalizations have been proposed, based on: algebraic structures, set theory, first-order logic

Summary

- Three-layers metamodeling architecture has known limitations
- Alternative organizations of the metalevels have been proposed
  - Multi-level metamodeling with clabjects
  - Architecture with nested levels has been proposed in the literature, with explicit definition of instantiation relation
- The model-metamodel conceptual distinction is present in other non-EMF based Modelware technologies like JetBrains MPS

Summary

Some research challenges

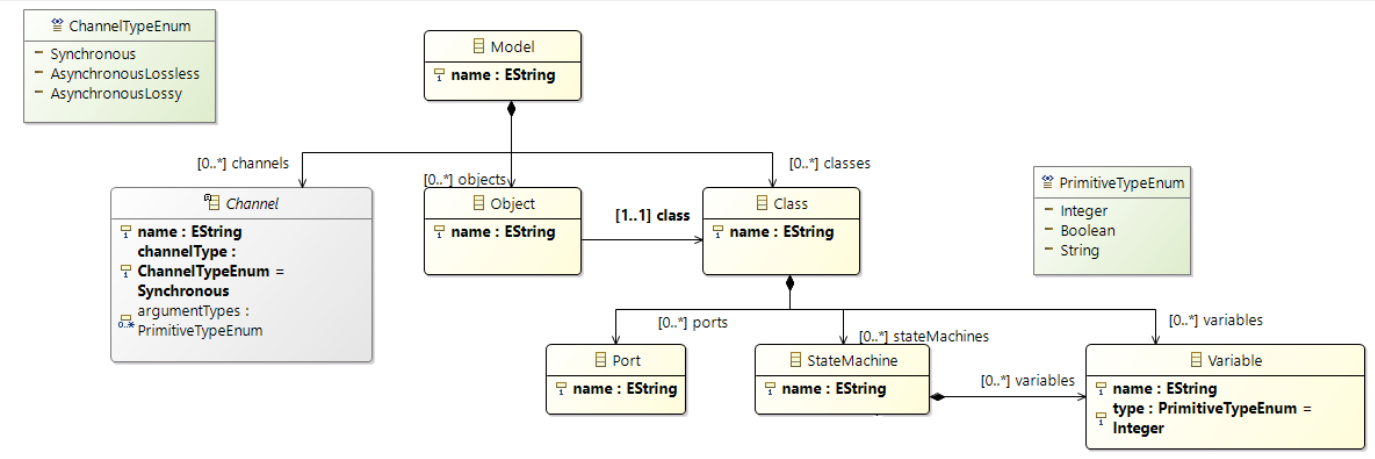
- Need for uniformly defined modularity for both models and metamodels. The issue should be addressed at the level of the metametamodel
  - Currently, every DSL (metamodel respectively) needs to define its own modularization approach and constructs
  - How do we define and compose fragments of DSLs/metamodels?
- Evolution of models following the evolution of their metamodel (co-evolution)
  - What happens with existing models if the metamodel changes?
- Expressivity issues as demonstrated in the multi-level model example
  - How adequate is, for example, Ecore to support DSL definition tasks?
- Model management

Model Validation

- Model constraints
- Model validation
- Object Constraints Language

Need for Constraints beyond Metamodels

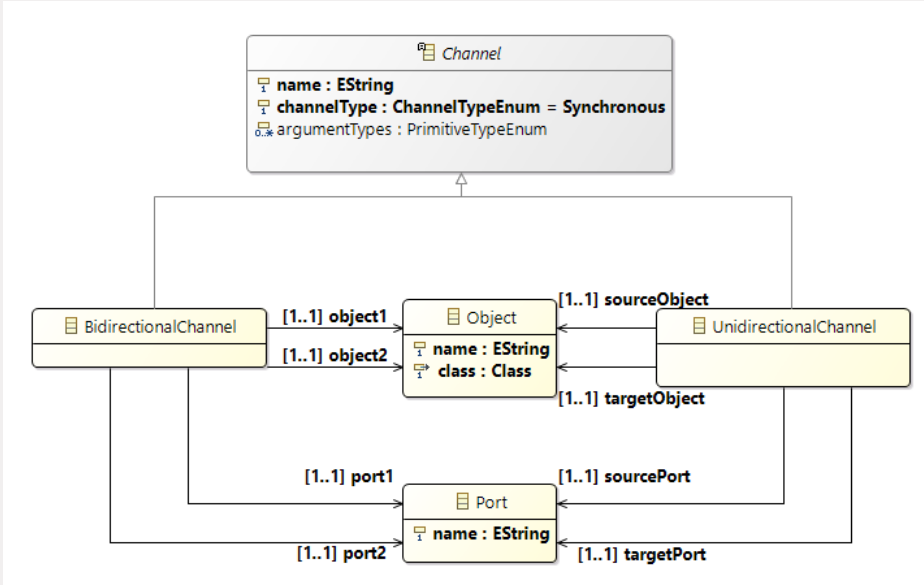
Consider again the metamodel of SLCO



We require:

- Every *class/object/channel* to have unique name within the containing model
- Every *port/state machine/variable* to have unique name within its class

Need for Constraints beyond Metamodels



We require:

- The *sourcePort* of a unidirectional channel to be a port in the class of the *sourceObject*
- *sourceObject* and *targetObject* are different for a channel

Model Constraints

Languages like Ecore and UML allow a restricted set of constraints to be expressed in the metamodels and class diagrams respectively:

- Allowed relations between objects
- Multiplicities on relation ends
- Types for attribute values

The previous slides show that there are other important constraints that need an explicit specification but **cannot be captured** by the metalanguage

If such a constraint does not hold in a model then this model does not belong to the language (although it may conform to the metamodel)!

Model Constraints

Constraints are also known as:

- Well-formedness constraints
- *Validity constraints* (term adopted in this lecture)
- Part of the static semantics of a DSL

Validity constraints further restrict the set of valid models that belong to a DSL

- in addition to requiring that a model conforms to the metamodel

Model validity constraints should be considered as an integral part of the language abstract syntax, although they are not expressed in the metalanguage

Model Constraints

Validity constraints are typically *defined* in the context of meta elements in the *metamodel* and *checked* (evaluated) on a concrete *model*

Apart from constraints on models, we often need to specify:

- Default values of attributes
- Values of derived attributes (calculated from other model elements)
- Pre- and post-conditions of operations
- Operation bodies (recall that Ecore allows class operations)

Object Constraints Language

Validity constraints can be expressed in a DSL, a dedicated *constraints language*

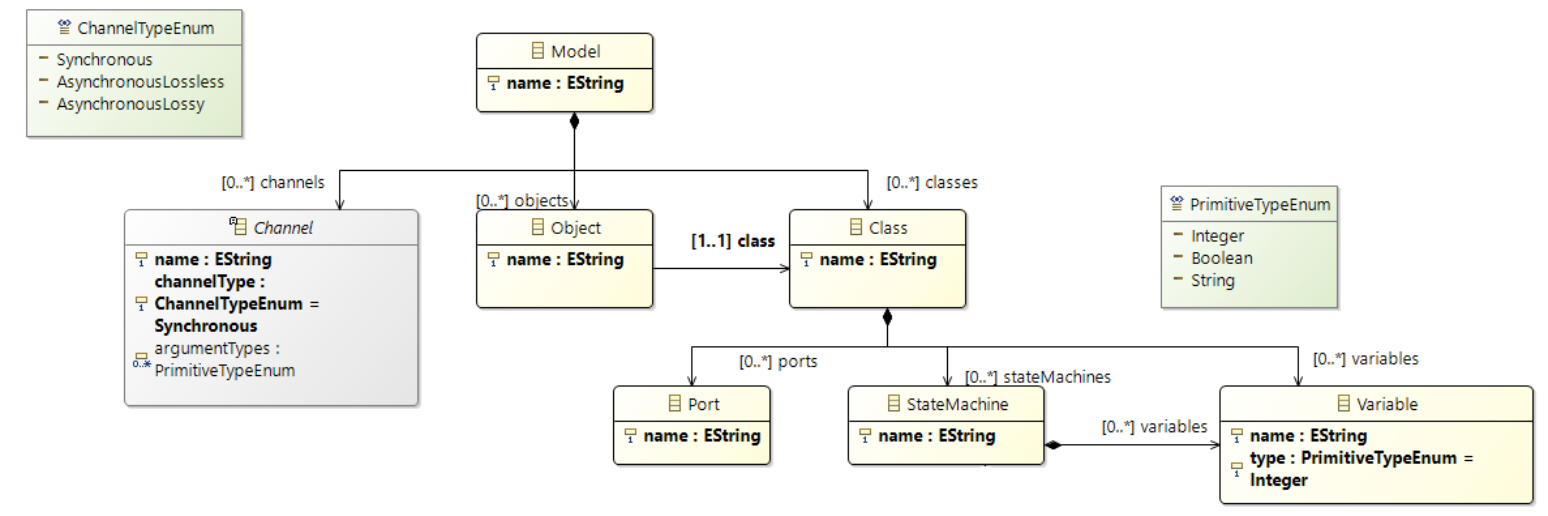
Object Constraints Language (OCL):

- DSL for defining constraints on Ecore models and (originally) UML models
- OCL is an OMG standard

Latest version

- 2.4 (from 2014), document *formal-14-02-03*
- *https://www.omg.org/spec/OCL/About-OCL/*

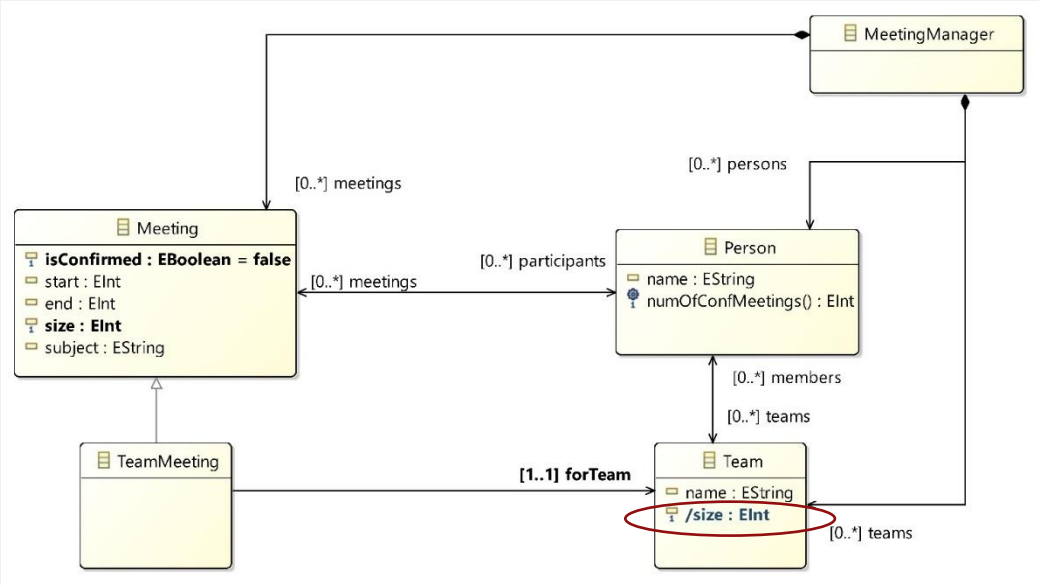
OCL Examples



Invariant for all instances of a given class:

```
context Model
inv uniqueObjNames: self.objects->forAll(o1, o2| o1 <> o2 implies o1.name <> o2.name)
```

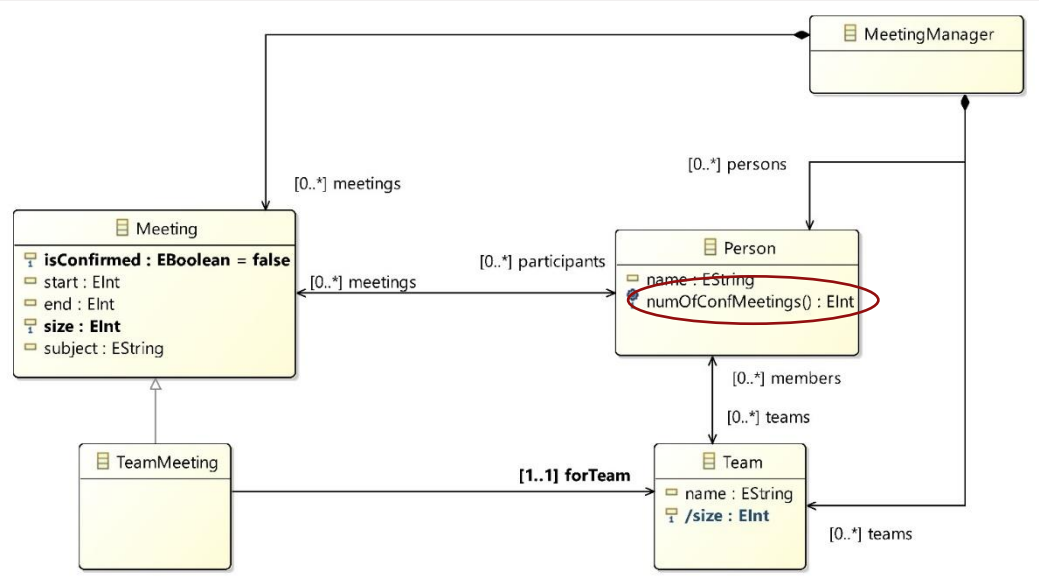
OCL Examples



The value of *size* attribute in class *Team* is the number of team members (derived attribute)

```
context Team::size : Integer
    derive: self.members->size()
```

OCL Examples



For a given person, operation *numOfConfMeetings* returns the number of meetings that have been confirmed

```
context Person::numOfConfMeetings() : Integer
    body: self.meetings->select(isConfirmed)->size()
```

Overview of OCL

OCL is a typed language

- every expression has a type
- evaluation produces a value of this type

OCL is a declarative side-effect free language

- does not contain imperative constructs (e.g. assignment)
- underlying model and OCL values are not changed during evaluation of OCL expressions

Usage of OCL

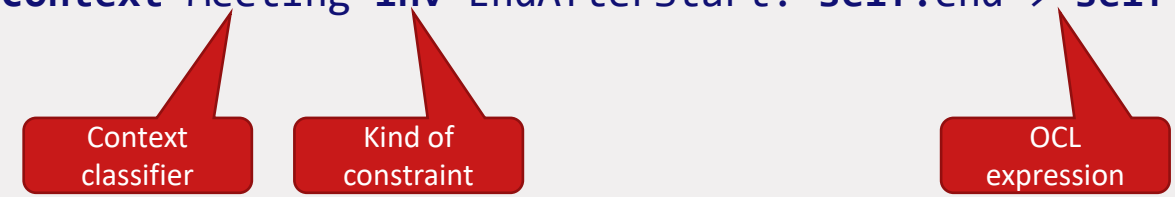
- Originally, OCL was intended for writing constraints that complement UML models and later MOF metamodels (~ 1995 at IBM)

- OCL is often embedded in other languages:
  - Model query languages
  - Model transformation languages like QVT
  - Model navigation

- There are experimental extensions of OCL that add imperative constructs (Imperative OCL)

OCL Syntax

```
context Meeting inv EndAfterStart: self.end > self.start
```



- OCL expressions are specified in a given **context**
- Context is a classifier in the scope of a given metamodel, e.g.: *class*, *attribute*, *operation*
- **self** is a predefined variable that refers to an instance of the context

Kinds of OCL Constraints

Invariant

- the constraint always holds for all instances of the context

```
context Team inv AtLeastOneMeeting:
     TeamMeeting.allInstances()->select(forTeam = self)->size() > 0
```

An optional message if the constraint is violated can be given:

```
context Team inv AtLeastOneMeeting('A team must participate in at least one meeting'):
     TeamMeeting.allInstances()->select(forTeam = self)->size() > 0
```

```
context Team inv AtLeastOneMeeting('Team ' + self.name + 'must participate in at least one
meeting'):
     TeamMeeting.allInstances()->select(forTeam = self)->size() > 0
```

Kinds of OCL constraints

Derived attribute value:

```
context Team::size : Integer
      derive: self.members->size()
```

Initial attribute value:

```
context Meeting::isConfirmed : Boolean
     init: false
```

Operation body, pre- and post- conditions:

```
context Person::numOfConfMeetings() : Integer
      body: self.meetings->select(isConfirmed)->size()
      pre: self.teams->size() > 0
      post resultOk: result > 0
```

Kinds of OCL constraints

Helper variables and operations

- sometimes it is useful to define new attributes and operations on top of an existing model and use them in OCL expressions

```
context MeetingManager
    def: nrOfConfMeetings : Integer =
       self.meetings->select(isConfirmed)->size()
    def: personKnown(n : String) : Boolean =
       self.persons->exists(p | p.name = n)
```

Attention: these helpers **are not** features of the *MeetingManager* class (the metamodel stays unchanged)

Overview of OCL Types

Primitive types

- Boolean, Integer, String, Real, …

Collections

- Set, OrderedSet, Bag, Sequence

Tuples

Enumeration types

Model element types

- Classifiers from context models, for example, Ecore classes, UML classes

OCL Primitive Types

Type	Values	Operations
Integer	Math set Z	+, -, /, *, abs(), …
Boolean	true, false	not, or, and, xor, implies
Real	Math set R	+, -, /, *, …
String	Sequences of characters	concat(), size(), substring(),…
OclInvalid	invalid	
OclVoid	null, invalid	

Access to Features

OCL expressions can access features of objects

Features may be:

- Attributes
- Association/reference ends
- Side-effect free operations

OCL uses the ‘dot’ notation to access attributes and references: `self.meetings`

OCL Collection Types

- *Set*, *OrderedSet*, *Bag*, *Sequence*
- *Collection* is a super type of all the above
- Collection types may indicate the type of elements, e.g. `Set(Integer)`

Syntax for calling an operation (*arrow accessor*, ->):

- `_value -> _opName(…)`

Example:

```
{1, 2, 3}->size() --returns 3
```

OCL Set Type

- Set in the mathematical sense
- No order of elements
- Duplicated elements count as one element (see below)

Set equality:

```
Set{1, 1} = Set{1}
Set{1, 2} = Set{2, 1}
Set{1, 2, 1} = Set{1, 2}
```

OCL OrderedSet

- Elements are ordered
- OrderedSet is **not** a subtype of Set

OrderedSet equality:

```
OrderedSet{1, 2} ≠ OrderedSet{2, 1}
```

But(!):

```
OrderedSet{1, 2, 1} = OrderedSet{1, 2}
```

OCL Bag

- Elements are unordered
- Duplications are allowed

Bag equality:

```
Bag{1, 1} ≠ Bag{1}
Bag{1, 2} = Bag{2, 1}
Bag{1, 2, 1} = Bag{2, 1, 1}
```

OCL Sequence

- Elements are ordered
- Duplications are allowed

Sequence equality:

```
Sequence{1, 2} ≠ Sequence{2, 1}
Sequence{1, 2} = Sequence{1, 2}
Sequence{1, 2, 1} ≠ Sequence{1, 2}
```

Collections Subtyping

Type	Is subtype of	Condition
Collection(T1)	Collection(T2)	If T1 subtype of T2
Set(T1)	Collection(T2)	If T1 subtype of T2
OrderedSet(T1)	Collection(T2)	If T1 subtype of T2
Bag(T1)	Collection(T2)	If T1 subtype of T2
Sequence(T1)	Collection(T2)	If T1 subtype of T2

Subtyping relation is transitive and reflexive

Collection Iterators

select and **reject** return a new collection by applying certain conditions on the elements of a given collection

Select:

```
Collection->select(v : Type | boolExp(v) )
```

Returns a collection with the elements that satisfy *boolExp*

Reject:

```
Collection->reject(v : Type | boolExp(v))
```

Returns a collection with the elements that **do not** satisfy *boolExp*

Collection Iterators

forAll:

```
Collection->forAll(v : Type | boolExp(v) )
```

true if *boolExp* holds for every element of *Collection*

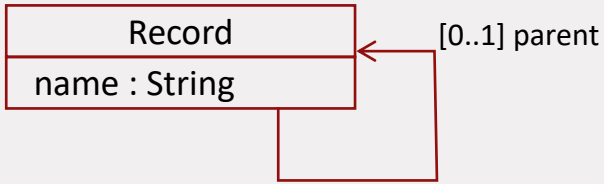
exists:

```
Collection->exists(v : Type | boolExp(v))
```

true if there is at least one element in *Collection* that satisfies *boolExp*

```
context MeetingManager
    inv uniqueNames:
        self.persons->forAll(p1| self.persons->forAll(p2 |
                    p1 <> p2 implies p1.name <> p2.name))
```

Closure

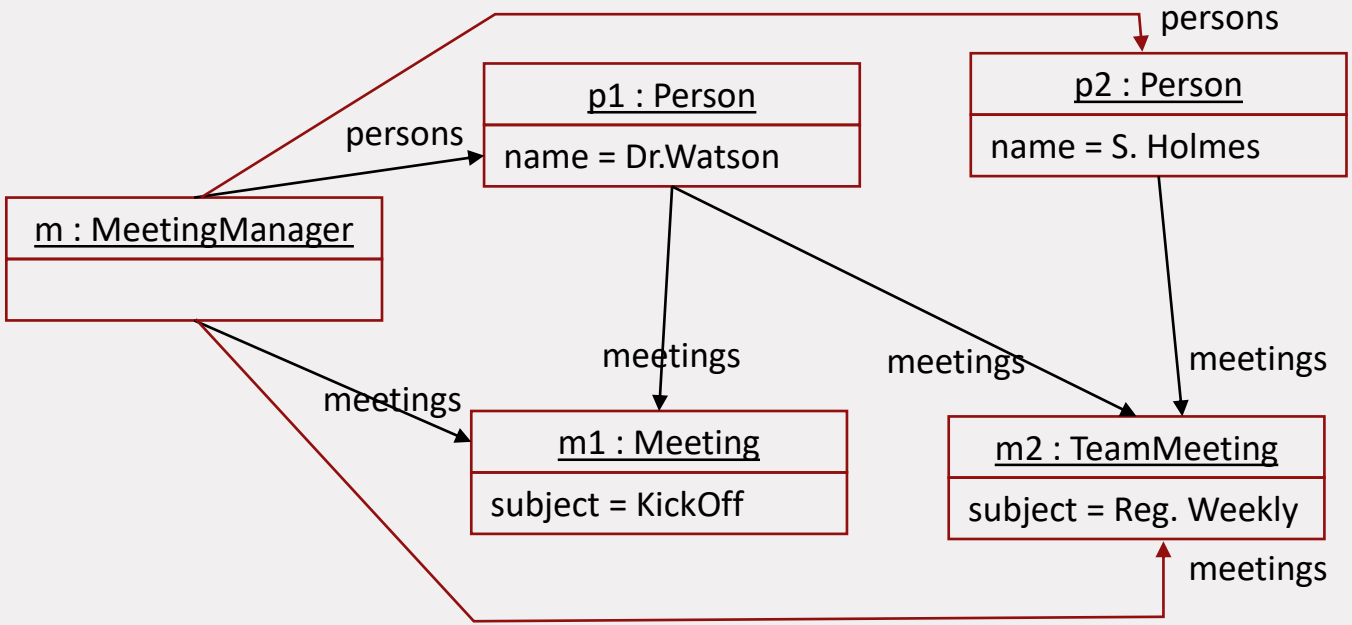


Consider the definition of class *Record*. We want to specify that the inheritance hierarchy (following reference *parent*) is acyclic.

OCL operation **closure** can be used (stands for reflective transitive closure):

```
context Record
     inv not_acyclic: self.parent->closure(r | r.parent)->excludes(self)
```

Types of Navigation Expressions




```
m.persons = {p1, p2} --result is an ordered set
m.persons.meetings = (m1, m2, m2) --result is a sequence!
m.persons.name = (Dr.Watson, S.Holmes)
m.persons.meetings.subject = (KickOff, Reg. Weekly, Reg. Weekly)
```


TU/e

Special Values

Null

- **null** indicates absence of value
- **null** is of type **OclVoid**
- a feature **f** with lower bound 0 may have no value

  `a.f` returns **null** if the feature is not set

Invalid

- **invalid** indicates error when evaluating an expression
- **invalid** is of type **OclInvalid**

  `Sequence{}->first()` returns **invalid**

  `a.oclAsType(_someType)` returns **invalid** if *a* cannot be cast to *_someType*

Special Values

Methods for checking if a value is **null** or **invalid**:

- **oclIsUndefined()** returns true if the object is null or invalid
- **oclIsInvalid()** returns true if the object is invalid

```
Sequence{}->first().oclIsUndefined()      true
Sequence{}->first().oclIsInvalid()        true
null.oclIsUndefined()                     true
null.oclIsInvalid()                       false
invalid.oclIsInvalid()                    true
invalid.oclIsUndefined()                  true
```

Attention: **null** is a valid value! It can be an element of collections

Special Values

If **invalid** is part of an expression, this expression evaluates to **invalid**

… with the following exceptions:

```
true or _anything -> true
     true or invalid      true
     invalid or true      invalid

false and _anything -> false
false implies _anything -> true
_anything implies true -> true
```

If-then-else-endif OCL expression is valid if the chosen branch is valid

```
     if true then 1 else invalid endif --returns 1
```

Tuples

Tuples group named values together, in the same way as records in some languages

Tuple types:

- `Tuple(company : Company, nrEmployees : Integer)`

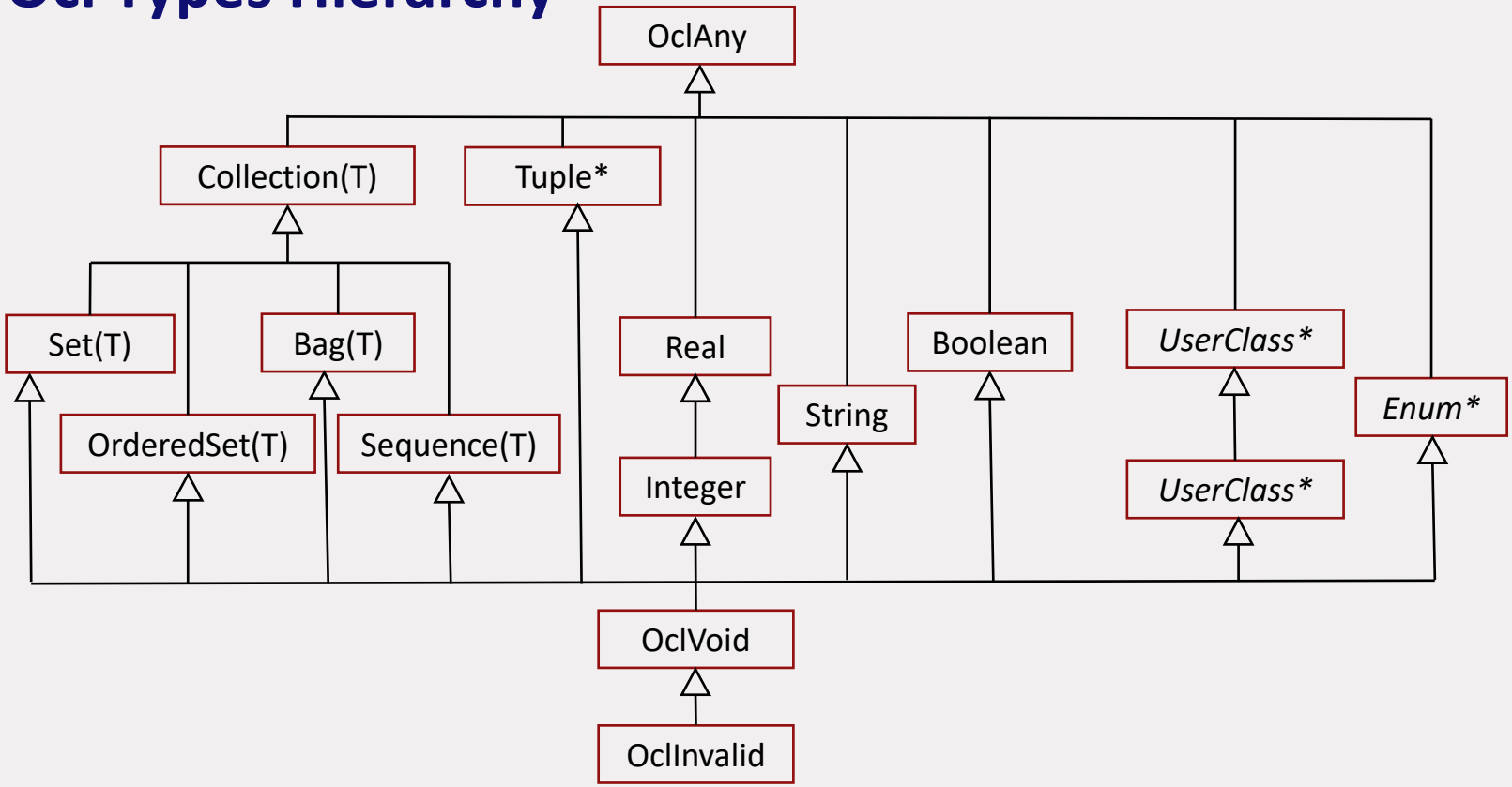
Tuple literals:

- `Tuple{name : String = “John”, age : Integer = 21}`
- `Tuple{name = “John”, age = 21}`

Field access:

- `Tuple{name = “John”, age = 21}.age` -- results in 21

Ocl Types Hierarchy



Operations of OclAny

The following is a selection of operations defined for *OclAny*, therefore applicable to all objects

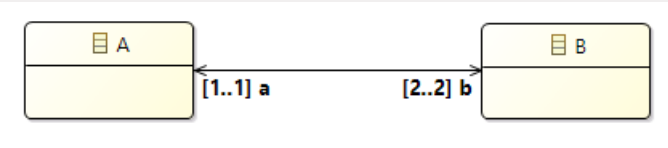
```
=(obj : OclAny) : Boolean
<>(obj : OclAny) : Boolean
oclIsUndefined : Boolean
oclIsInvalid : Boolean

oclAsType(t : Classifier) : T
oclIsTypeOf(t : Classifier) : Boolean -- true if the object is instance of t
oclIsKindOf(t : Classifier) : Boolean -- true if the object is instance of t or
                                         one of its subclasses
oclType() : Classifier
```

Consistency of Constraints

It is possible that for a given metamodel *M* and a set of constraints *C*, no model exists that conforms to *M* and satisfies *C*

- the combination <M, C> is *inconsistent*
- in less severe cases, only some classes from *M* may be 'dead' (no model can contain their instances)

Example:




```
B.allInstances()->size() = 1
```

- `_Type.allInstances()` returns the set of all instances of the type in a given model
- not allowed for infinite types, e.g. `Integer.allInstances()`

Redundancy of Constraints

Furthermore, it is possible that a given constraint is *redundant*:

- The satisfaction of a constraint is always implied by other constraints over all models in a language

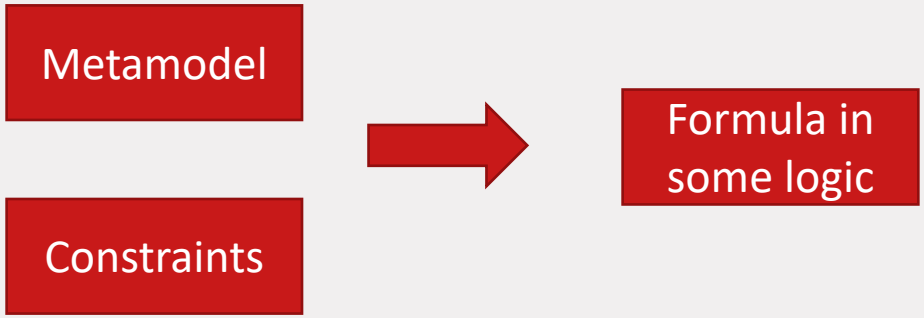
Generally, it is difficult to manually check the consistency and redundancy in non-trivial realistic cases

Analysis of Metamodels and Constraints

OCL has underlying mathematical semantics based on set theory

The same has been done for metamodels

- Open the possibility for automated checks
- There are research prototypes that perform such checks (tools USE and EFinder)



The idea is to use the semantics in order to translate the abstract syntax (metamodel plus constraints) to a logic formula

- Reduces the consistency check to the satisfiability of a formula (a NP-complete problem!)
- Check if the abstract syntax is consistent, i.e. there is at least one valid model
- Check for dead classes (classes that can never be instantiated)
- Completion of partial models

TU/e

Ocl Tools

OCL tools shall allow:

- Creating OCL expressions
- Evaluating OCL expressions

Evaluation of OCL expressions generally depends on existing models

- Constraints are evaluated over existing model elements
- OCL queries suppose a model exists

OCL tools, therefore, are typically integrated into other modeling tools

Eclipse Ocl

Built on top of EMF and Ecore

Eclipse OCL supports

- Embedding OCL into Ecore metamodels via **OCLInEcore**
- Specifying OCL in a separate document complementary to Ecore metamodels via **Complete OCL** editor
- Interactive evaluation of OCL expressions via **Interactive OCL Console**
- Evaluation via Java API

Ocl in Ecore

OCLInEcore:

Textual editor for Ecore metamodels and OCL constraints

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;

package meetings : meetings = 'http://www.example.org/meetings'
{
    class Meeting
    {
        attribute isConfirmed : Boolean[1];
        attribute start : ecore::EInt[1];
        attribute end : ecore::EInt[1];
        attribute size : ecore::EInt[1];
        property participants#meetings : Person[*|1] { ordered };
        attribute subject : String[?];
        invariant EndAfterStart: self.end > self.start;
    }
    class Team
    {
        attribute name : String[?];
        property members#teams : Person[*|1] { ordered };
        attribute size : ecore::EInt[1] { derived }
        {
            derivation: members->size();
        }
    }
```

Complete Ocl

OCLConstraints.ocl

```
1  import 'meetings.ecore'
2
3  package meetings
4
5  context Meeting
6
7      inv EndAfterStart: self.end > self.start
8
9      inv: self.oclIsTypeOf(TeamMeeting) implies self.participants->includesAl
10
11 context Meeting::isConfirmed : Boolean
12     init: false
13
14 context Team::size : Integer
15
16     derive: members->size()
17
18 context TeamMeeting
19
20     inv AllTeamMembersInMeeting: participants->includesAll(forTeam.members)
21
22 context Person::numOfConfMeetings() : Integer
23     post: result = meetings->select(isConfirmed)->size()
24
25     body: self.meetings->select(isConfirmed)->size()
```

The metamodel for which we define the constraints

Invariant

Derived value

Operation body

- Textual editor for OCL constraints
- In contrast to OClInEcore, it stores the constraints in a document separate from the metamodel

This is the recommended practice in this course

Demo

Tools: Eclipse OCL

- Complete OCL
- Model validation
- OCL Console

All demo activities are described in the *Tool Guide*

OCL Evaluation

Shortcomings of OCL:

- Lack of severity levels when a constraint is violated: for example, *error* vs *warning*
- No support for user input
- No support for error repairing
- Lack of context flexibility
  - Definition context is the same as error context

Tools and standardization

- Still immature tools
- Inconsistencies between the official spec and the implementations

Alternatives to OCL

Validation and specification languages

- Epsilon Validation Language (EVL), part of Epsilon platform, University of York (UK)
- Using some existing specification languages that offer analysis capabilities and have formal semantics (e.g. Alloy and Alloy Analyzer, logic-based languages and theorem provers)

Validity constraints implemented in a GPL

- Modern languages already offer concise constructs for the typical tasks performed in OCL
- Example will be shown further in the course (lecture on Xtext)
- Still, constraints expressed in GPL are rather difficult to analyze for consistency and redundancy

Summary

Validity constraints

- are needed because the metamodeling language often cannot express all restrictions on the allowed model structure
- should be considered as an integral part of the language abstract syntax
- often expressed in a DSL

OCL

- OMG standard for writing model constraints
- can be used as a navigation and query language for models
- used in other MDE technologies, e.g. model transformation languages

Resources

On metamodeling:

- Atkinson, C., Kühne, T. *The Essence of Multilevel Metamodeling*, <<UML>> 2001, Springer
- *Software Languages*, Chapter 3 (an alternative formalization of models and metamodels)

OCL

- Eclipse OCL documentation as part of the Eclipse Help
- https://projects.eclipse.org/projects/modeling.mdt.ocl