Modellbasierte Softwareentwicklung (MODSOFT)

Part II

Domain Specific Languages

Semantics

Prof. Joachim Fischer /
Dr. Markus Scheidgen / Dipl.-Inf. Andreas Blunk

{fischer,scheidge,blunk}@informatik.hu-berlin.de

LFE Systemanalyse, III.310

Agenda

prolog (1 VL) — **Introduction:** languages and their aspects, modeling vs. programming, meta-modeling and the 4 layer model

0. (2 VL) — **Eclipse/Plug-ins:** eclipse, plug-in model and plug-in description, features, *p2*-repositories, *RCP*s

1. (2 VL) — **Structure:** *Ecore*, *genmodel*, working with generated code, constraints with *Java* and *OCL*, *XML/XMI*

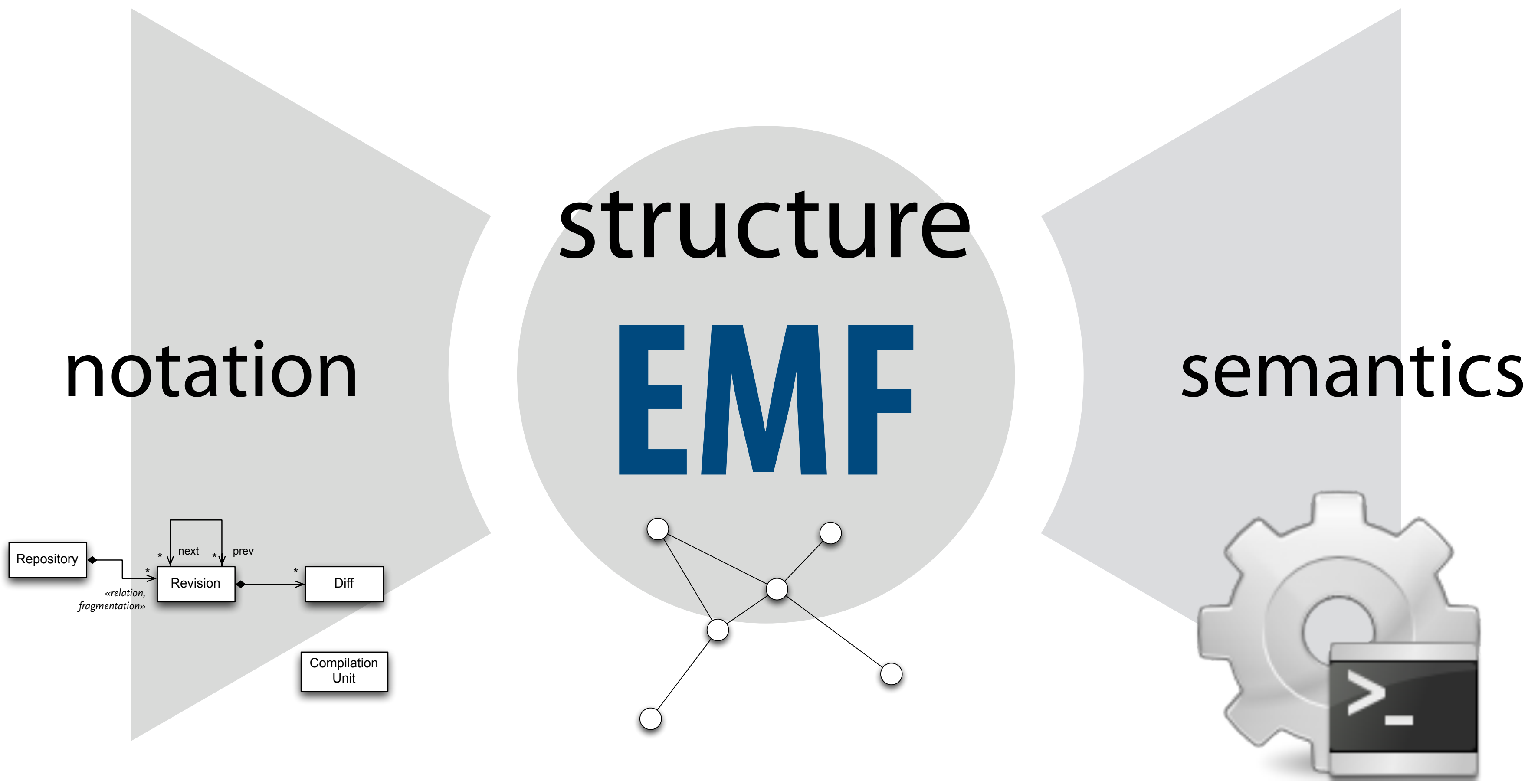
2. (3 VL) — **Notation:** Customizing the tree-editor, textural with *XText*, graphical with *GEF* and *GMF*

➡ 3. (4 VL) — **Semantics:** interpreters with Java, code-generation with *Java* and *XTend*, model-transformations with *Java* and *ATL*

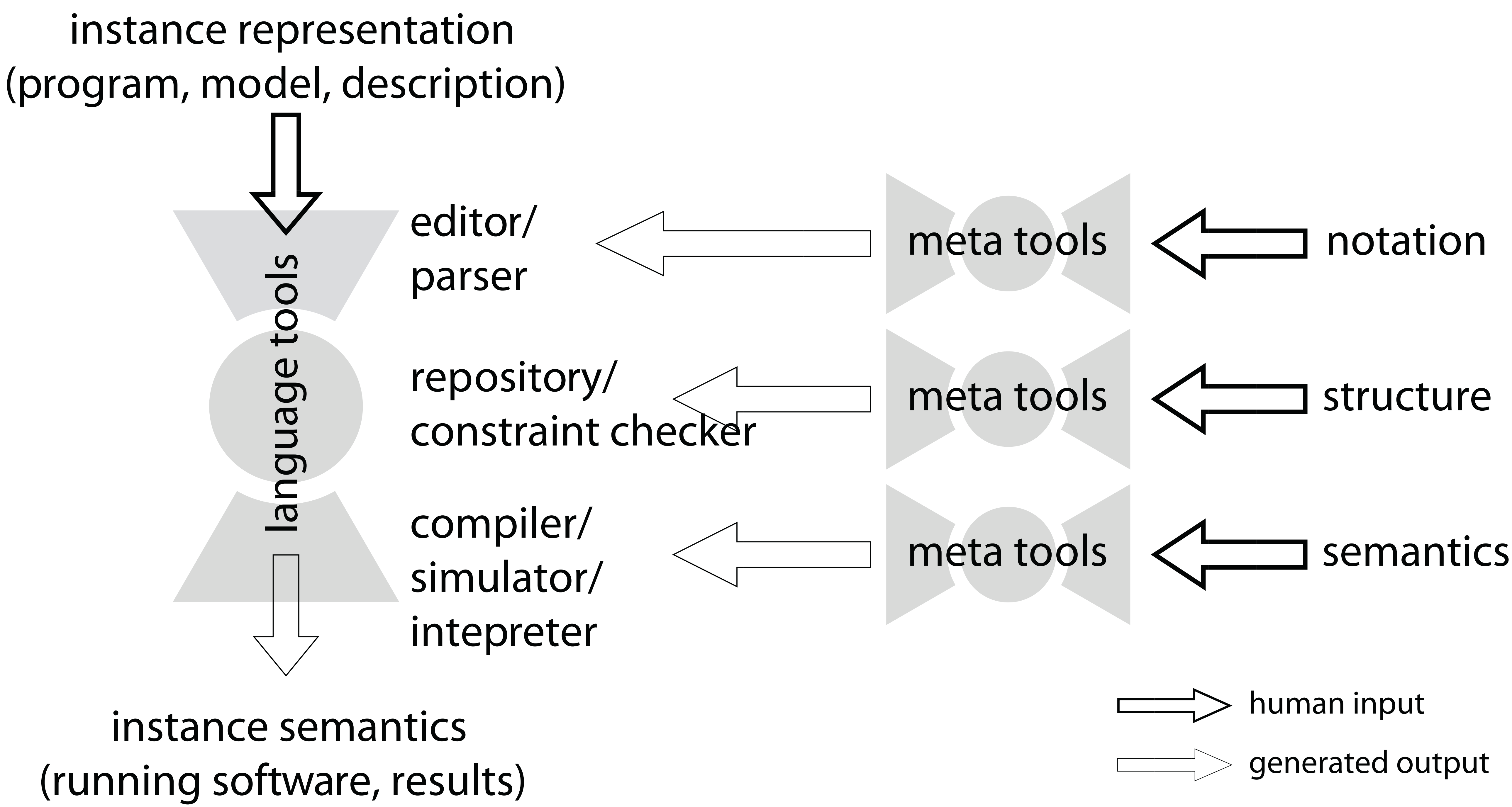
epilog (2 VL) — **Tools:** persisting large models, model versioning and comparison, model evolution and co-adaption, modular languages with *XBase*, *Meta Programming System* (MPS)

Previously on MODSOFT

Eclipse Modeling Framework



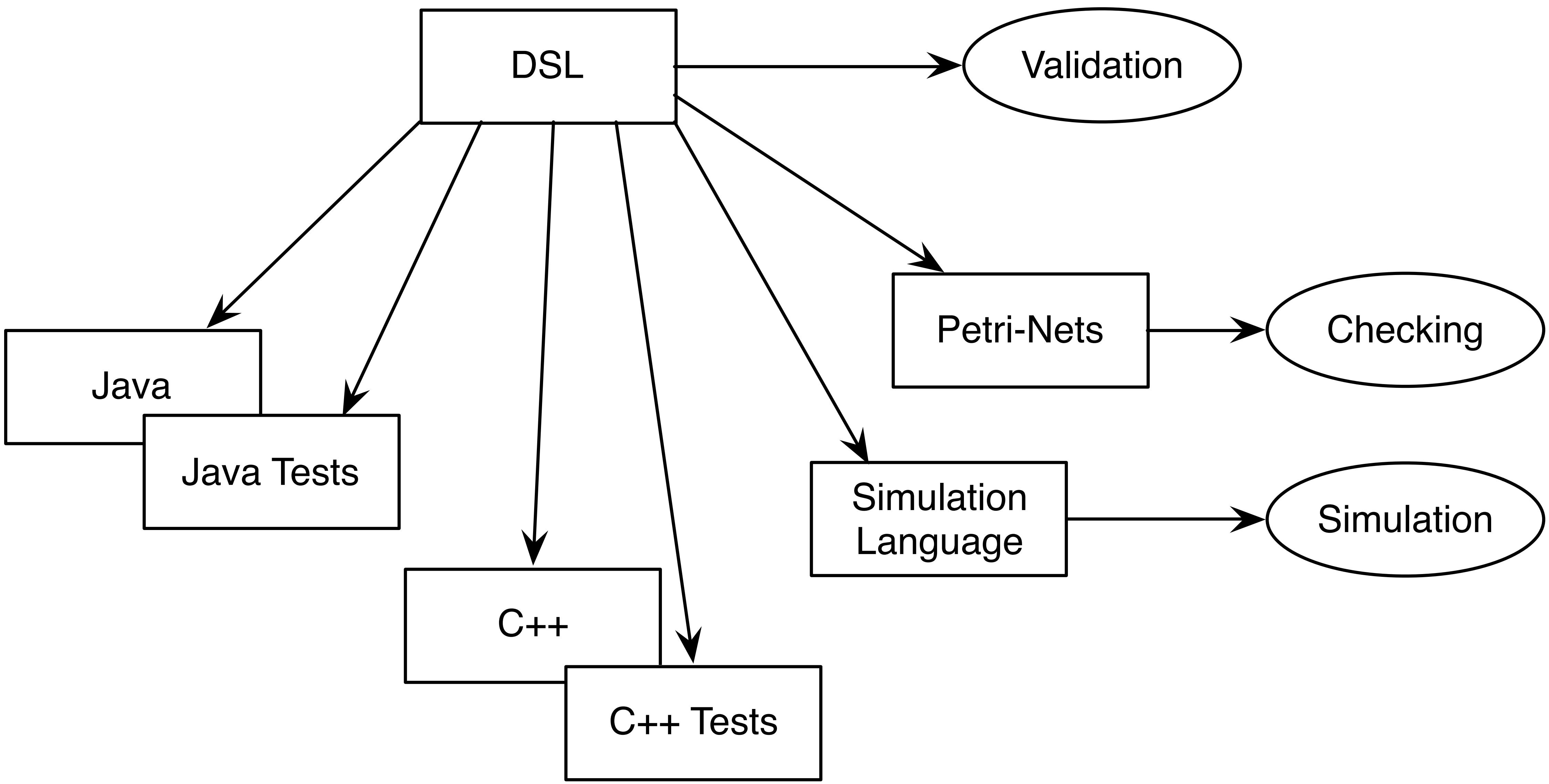
Meta-Languages



Different Types of Semantics

- **Operational Semantics**
- Denotational Semantics
- Axiomatic Semantics
- **Translational Semantics**

Semantics and DSLs

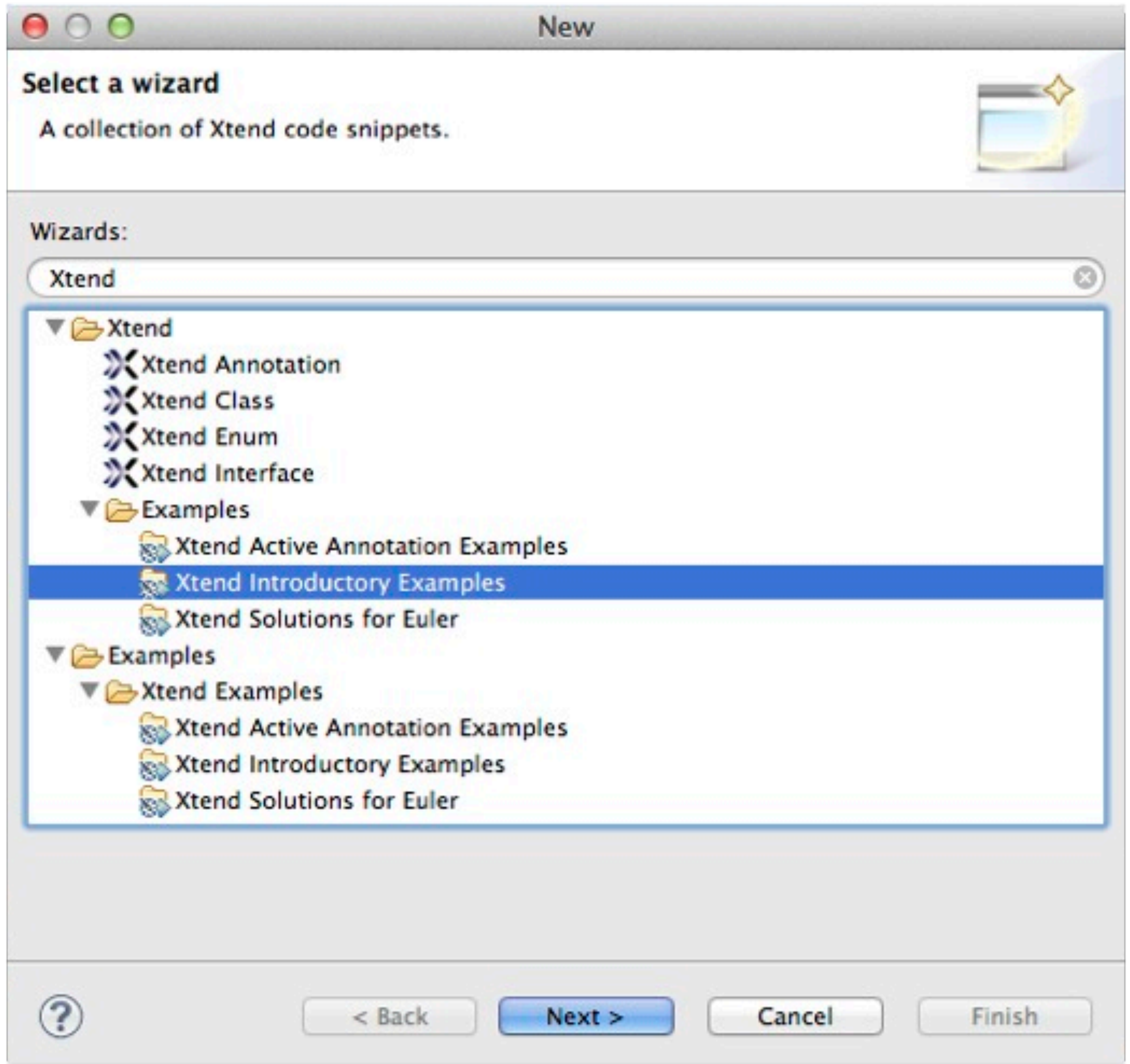


Xtend

Xtend

- External DSL
- transparently compiles into Java
- written with Xtext
- Xtext code can be called from Java code, Java code can be called from within Xtext code

Xtend Examples



Xtend Basics

```
package example1

class HelloWorld {
    def static void main(String[] args) {
        println('Hello World!')
    }
}
```

Rich Strings

```
class BottleSong {

    @Test def void singIt() {
        println(singTheSong(99))
    }

    def singTheSong(int all) '''
        «FOR i : all .. 1»
            «i.Bottles» of beer on the wall, «i.bottles» of beer.
            Take one down and pass it around, «(i - 1).bottles» of beer on the wall.

        «ENDFOR»
        No more bottles of beer on the wall, no more bottles of beer.
        Go to the store and buy some more, «all.bottles» of beer on the wall.
    '''

}

class BottleSupport {

    def static bottles(int i) {
        switch i {
            case 0 : 'no more bottles'
            case 1 : 'one bottle'
            default : '''«i» bottles'''
        }.toString
    }

    def static Bottles(int i) {
        bottles(i).toFirstUpper
    }
}
```

```
@Data class Movie {
    String title
    int year
    double rating
    long numberOfVotes
    Set<String> categories
}
```

Higher Order Functions

```
class Movies {

    @Test def void numberOfActionMovies() {
        assertEquals(828, movies.filter[categories.contains('Action')].size)
    }

    @Test def void yearOfBestMovieFrom80ies() {
        assertEquals(1989, movies.filter[(1980..1989).contains(year)].sortBy[rating].last.year)
    }

    @Test def void sumOfVotesOfTop2() {
        val long movies = movies.sortBy[-rating].take(2).map[numberOfVotes].reduce[a, b| a + b]
        assertEquals(47_229, movies)
    }

    val movies = new FileReader('data.csv').readLines.map[ line |
        val segments = line.split('  ').iterator
        return new Movie(
            segments.next,
            Integer.parseInt(segments.next),
            Double.parseDouble(segments.next),
            Long.parseLong(segments.next),
            segments.toSet
        )
    ]
}
```

Xtext + Xtend

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
    greetings+=Greeting*;

Greeting:
    'Hello' name=ID '!';
```

```
class MyDslGenerator implements IGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
        fsa.generateFile('greetings.txt', 'People to greet: ' +
            resource.allContents
                .filter(typeof(Greeting))
                .map[name]
                .join(', '))
    }
}
```

Object Constraint Language (OCL)

Jos Warmer and Anneke Kleppe, JOOP, 1999

Outline

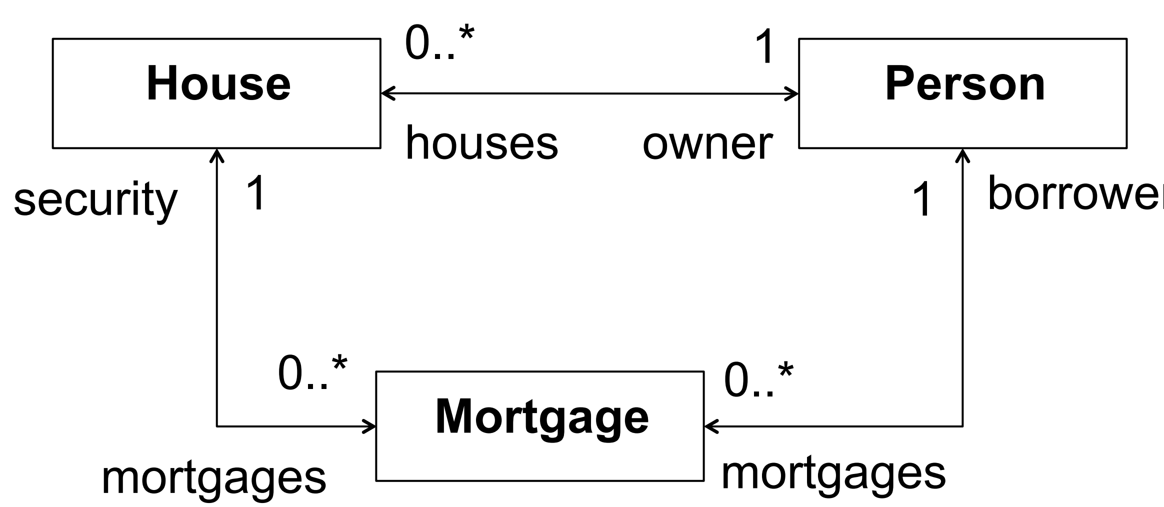
- Motivation
- Basics of OCL
  - Specifying invariants
  - Specifying pre and post-conditions
  - Navigating in OCL expressions
  - Basic values and types
- Collections in OCL

Review

- Protocol
- Documenting protocols
  - Syntactic and semantic interfaces

Object Constraint Language (OCL)

- Motivation
  - UML diagrams don't tell everything
  - Q: What does the following class diagram tell?



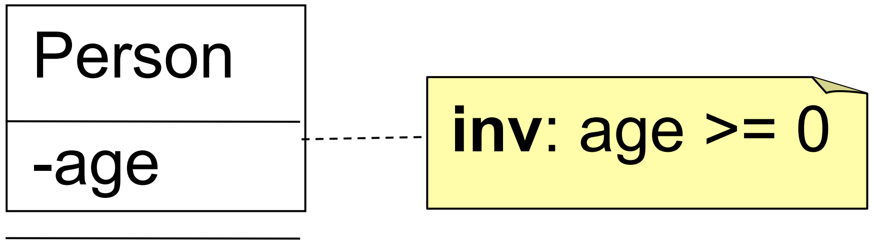
OCL – What Is It?

- Standard "add-on" to UML
  - OCL expressions dependent on types from UML diagrams
  - defined by Object Management Group (OMG)
- Language for expressing additional information (e.g., constraints and business rules) about UML models
- Characteristics
  - Constraint and query language
  - Math foundation (set and predicate) but no math symbols
  - Strongly typed, declarative, and no side effect
  - High level of abstraction (platform independence)

Basics of OCL

- Associating OCL expressions to UML models
  - Directly to diagrams as notes
  - Separate accompanying texts, e.g.,

```
context Person
inv: age >= 0
```



- Specifying invariants
  - State conditions that must be always be met by all instances of context types (classes or interfaces)

Basics of OCL – Invariants

```
context Company inv:
self.numberOfEmployees > 50
```

self: contextual instance, an instance to which the OCL expression is attached

```
context c: Company inv:
c.numberOfEmployees > 50
```

An explicit specification of contextual instance, c

```
context c: Company inv enoughEmployees:
c.numberOfEmployees > 50
```

an optional label

Specifying Pre and Post-conditions

- Pre and post-conditions
  - Conditions that must be true at the moment when an operation begins and ends its execution.

```
context Account::deposit(amt: Integer): void
pre: amt > 0
post: balance = balance@pre + amt
```

pre-value, referring to previous value

```
context Account::deposit(amt: Integer): void
pre argumentOk: amt > 0
post balanceIncreased: balance = balance@pre + amt
```

optional label

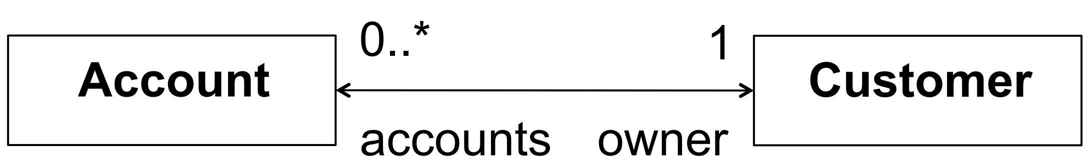
Referring to Pre-value and Result

- @pre: denotes the value of a property at the start of an operations
- result: denotes the result of an operation

```
context Account::payInterest(rate: Real): void
post: balance = balance@pre + calcInterest@pre(rate)

context Account::getBalance(): Integer
post: result = balance
```

Navigating in OCL Expressions

- Use dot notation to navigate through associations
  - Direction and multiplicity matter
  - Use role names or class names




```
context Account
  inv: self.owner …  --comment
       self.customer …
```

single line (--) or multiple lines (/* … */)

```
context Customer
  /* multiline comment */
  inv: self.accounts->size() …
       self.account …
```

Arrow notation for collection operations

Types in OCL

- Two different kinds
  - Predefined types (as defined in standard library)
    - Basic types: Integer, Real, String, Boolean
    - Collection types: Set, OrderedSet, Bag, Sequence
  - User-defined types: classes, interfaces, and enumerations.
- Value vs. object types
  - Immutable vs. mutable types
  - All predefined types are value types, i.e., there is no mutation operation defined.

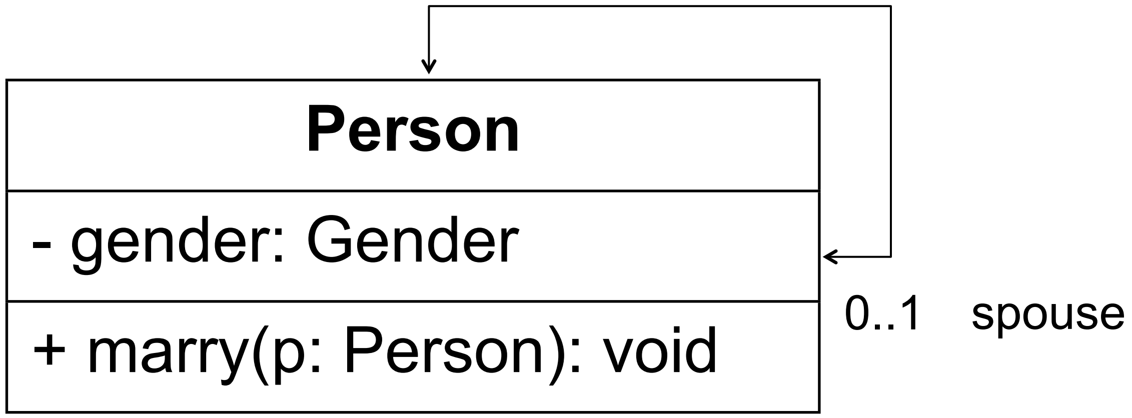
Basic Values and Types

- Several built-in types and operations

Type	Values	Operations
Boolean	false, true	or, and, xor, not, =, <>, implies
Integer	-10, 0, 10, ...	=, <>, <, >, <=, >=, +, -, *, /, mod(), div(), abs(), max(), min(), round(), floor()
Real	-1.5, 3.14, ...	
String	‘Carmen’	=, <>, concat(), size(), toLower(), toUpper(), substring()

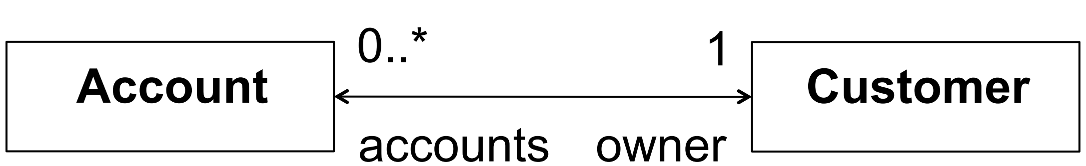
Exercise

- Write pre and post-conditions



Collections in OCL

- Why?
  - Multiple objects produced by navigating associations
  - If multiplicity > 1, collections based on properties
    - Set: {unique} (default)
    - OrderedSet: {unique, ordered}
    - Bag: {notUnique}
    - Sequence: {notUnique, ordered}




```
context Account
inv: self.owner.name <> ''

context Customer
inv: self.accounts->size() > 0
```

Standard Collection Types

- Parameterized with elements types, e.g., Set(Account)
- Value/immutable types, not reference types
- One abstract and four concrete types
  - Collection
  - Set, OrderedSet, Bag, Sequence
  - Determined based on properties of associations, e.g., unique, ordered, and sorted.

Collection Types

- Properties

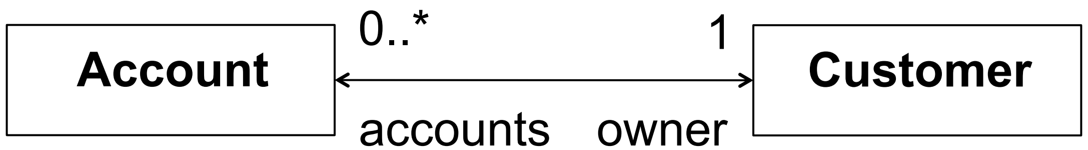
Type	Duplicate?	Ordered?
Set	N	N
OrderedSet	N	Y
Bag	Y	N
Sequence	Y	Y

*Ordered doesn't mean sorted.

- Literals
  - Set{10, 100}
  - OrderedSet{'apple', 'orange'}
  - Bag{10, 10, 100}
  - Sequence{10, 10, 100}, Sequence{1..10}, Sequence{1..(5 + 5)}
  - Set{Set{1}, Set{10}}

Collection Operations

- Large number of predefined operations
- Arrow notation, e.g., c->size()
  - Rationale: allow same-named, user-defined operations, e.g., c.size()




```
context Account
inv: not owner->isEmpty()
```

```
context Account
inv: not owner.isEmpty()
```

Collection Operations

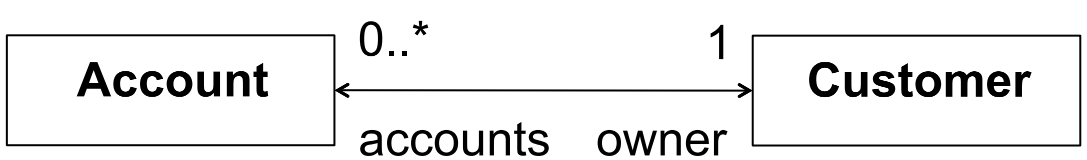
- Defined on all collection types

Operation	Description
count(o)	Number of occurrences of *o* in the collection (*self*)
excludes(o)	Is *o* not an element of the collection?
excludesAll(c)	Are all the elements of *c* not present in the collection?
includes(o)	Is *o* an element of the collection?
includesAll(c)	Are all the elements of *c* contained in the collection?
isEmpty()	Does the collection contain no element?
notEmpty()	Does the collection contain one or more elements?
size()	Number of elements in the collection
sum()	Addition of all elements in the collection

- Type-specific operations
  - append, including, excluding, first, last, insertAt, etc.

Iteration Operations

- Loop over elements by taking one element at a time
- Higher-order functions
- Iterator variables
  - Optional variable declared and used within body
  - Indicate the element being iterated
  - Always of the element type, thus, type declaration is optional




```
context Customer
inv: self.accounts->forAll(a: Account |a.owner = self)
inv: accounts->forAll(a | a.owner = self)
inv: accounts->forAll(owner = self)
```

Iteration Operations

Operation	**Description**
any(expr)	Returns any element for which *expr* is true
collect(expr)	Returns a collection that results from evaluating *expr* for each element of *self*
exists(expr)	Has at least one element for which *expr* is true?
forAll(expr)	Is *expr* true for all elements?
isUnique(expr)	Does *expr* has unique value for all elements?
iterate(x: S; y: T| expr)	Iterates over all elements
one(expr)	Has only one element for which *expr* is true?
reject(expr)	Returns a collection containing all elements for which *expr* is false
select(expr)	Returns a collection containing all elements for which *expr* is true
sortedBy(expr)	Returns a collection containing all elements ordered by *expr*

Iteration Operations

```
accounts->any(a: Account | a.balance > 1000)
accounts->collect(name) -- all the names
accounts->exists(balance > 5000)
accounts->forAll(balance >= 0)
accounts->isUnique(name)
accounts->iterate(a: Account; sum: Integer = 0 |
             sum + a.balance)
accounts->one(name = 'Carmen')
accounts->reject(balance > 1000)
accounts->select(balance <= 1000)
accounts->sortedBy(balance)
```

Select vs. Collect

- Q: Difference between select and collect?
- Note that the dot notation is short for collect, e.g.,

```
context Bank
inv: self.customers.accounts->forAll(balance > 0)
inv: self.customers->collect(accounts)
                   ->forAll(balance > 0)
```

Note that results are flattened for "collect" and not for "collectNested".

The Iterate Operation

- Most fundamental and generic loop operation
- All other loop operations are special cases

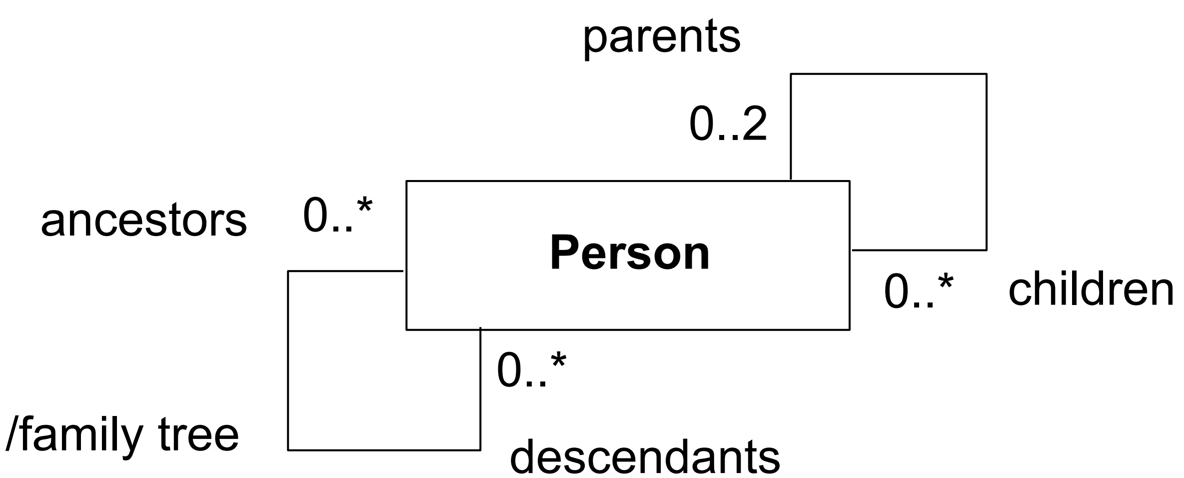
```
iterate(elem: T1; result: T2 = expr | expr-elem-result)
```

- Example

```
Set{1,2,3}->sum()

Set{1,2,3}->iterate(i:Integer; r:Integer=0 | r + i)
```

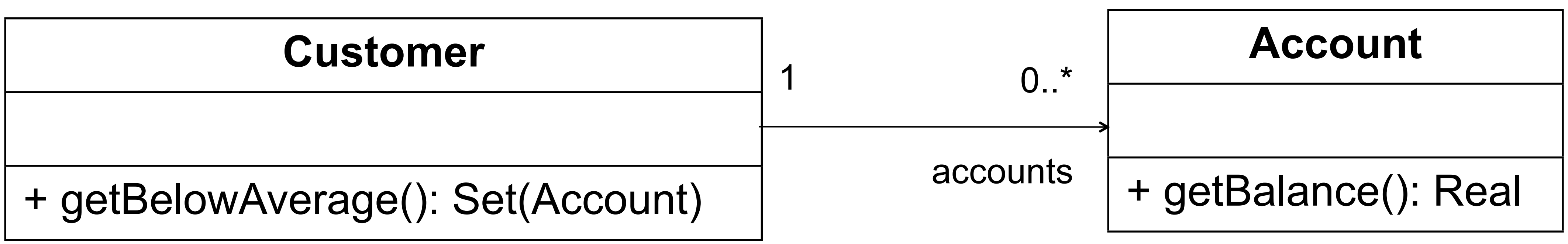
Exercise

- Formulate constraints for the parents/children and the derived associations.



Exercise

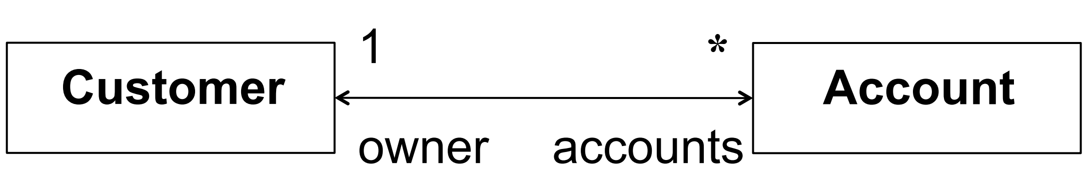
- Write the pre- and post-conditions of the getBelowAverage operation that returns all the accounts of a customer of which balances are below the average balance of the customer's accounts.



Informal Description

- Motivation
  - To escape from formality, but why?
  - To mix formal and informal texts in constraints.
- Approach

```
context Customer::getBelowAverage(): Set(Account)
pre: not accounts->isEmpty()
post: result = accounts->select(a: Account |
    a.getBalance() < informally("Avg of all account balances"))
```



Atlas Transformation Language (ATL)

Freddy Allilaire, Frédéric Jouault, 2007

Overview

- This presentation describes a very simple model transformation example, some kind of ATL "hello world".
- It is intended to be extended later.
- The presentation is composed of the following parts:
  - Prerequisites.
  - Introduction.
  - Metamodeling.
  - Transformation.
  - Conclusion.

Prerequisites

- In the presentation we will not discuss the prerequisites.
- The interested reader may look in another presentation to these prerequisites on:
  - MDE (MOF, XMI, OCL).
  - Eclipse/EMF (ECORE).
  - AMMA/ATL.

Introduction

- The goal is to present a use case of a model to model transformation written in ATL.
- This use case is named: “Families to Persons”.
- Initially we have a text describing a list of families.
- We want to transform this into another text describing a list of persons.

Goal of the ATL transformation we are going to write

Transforming this ...

...
Family March
  Father: Jim
  Mother: Cindy
  Son: Brandon
  Daughter: Brenda
... other Families

... into this.

...
Mr. Jim March
Mrs. Cindy March
Mr. Brandon March
Mrs. Brenda March
... other Persons

Let's suppose these are not texts, but models
(we'll discuss the correspondence
between models and texts later).

Input of the transformation is a model

Family March
Father: Jim
Mother: Cindy
Son: Brandon
Daughter: Brenda

Family Sailor
Father: Peter
Mother: Jackie
Son: David
Son: Dylan
Daughter: Kelly

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://
www.omg.org/XMI" xmlns="Families">
  <Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <sons firstName="David"/>
    <sons firstName="Dylan"/>
    <daughters firstName="Kelly"/>
  </Family>
</xmi:XMI>
```

This is the text.

This is the corresponding model.
It is expressed in XMI,
a standard way to represent models.

Output of the transformation should be a model

Mr. Dylan Sailor
Mr. Peter Sailor
Mr. Brandon March
Mr. Jim March
Mr. David Sailor
Mrs. Jackie Sailor
Mrs. Brenda March
Mrs. Cindy March
Mrs. Kelly Sailor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Male fullName="Dylan Sailor"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Jim March"/>
  <Male fullName="David Sailor"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

Each model conforms to a metamodel

Source metamodel

conformsTo

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://
www.omg.org/XMI" xmlns="Families">
  <Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <sons firstName="David"/>
    <sons firstName="Dylan"/>
    <daughters firstName="Kelly"/>
  </Family>
</xmi:XMI>
```

Source model "sample-Families.ecore"

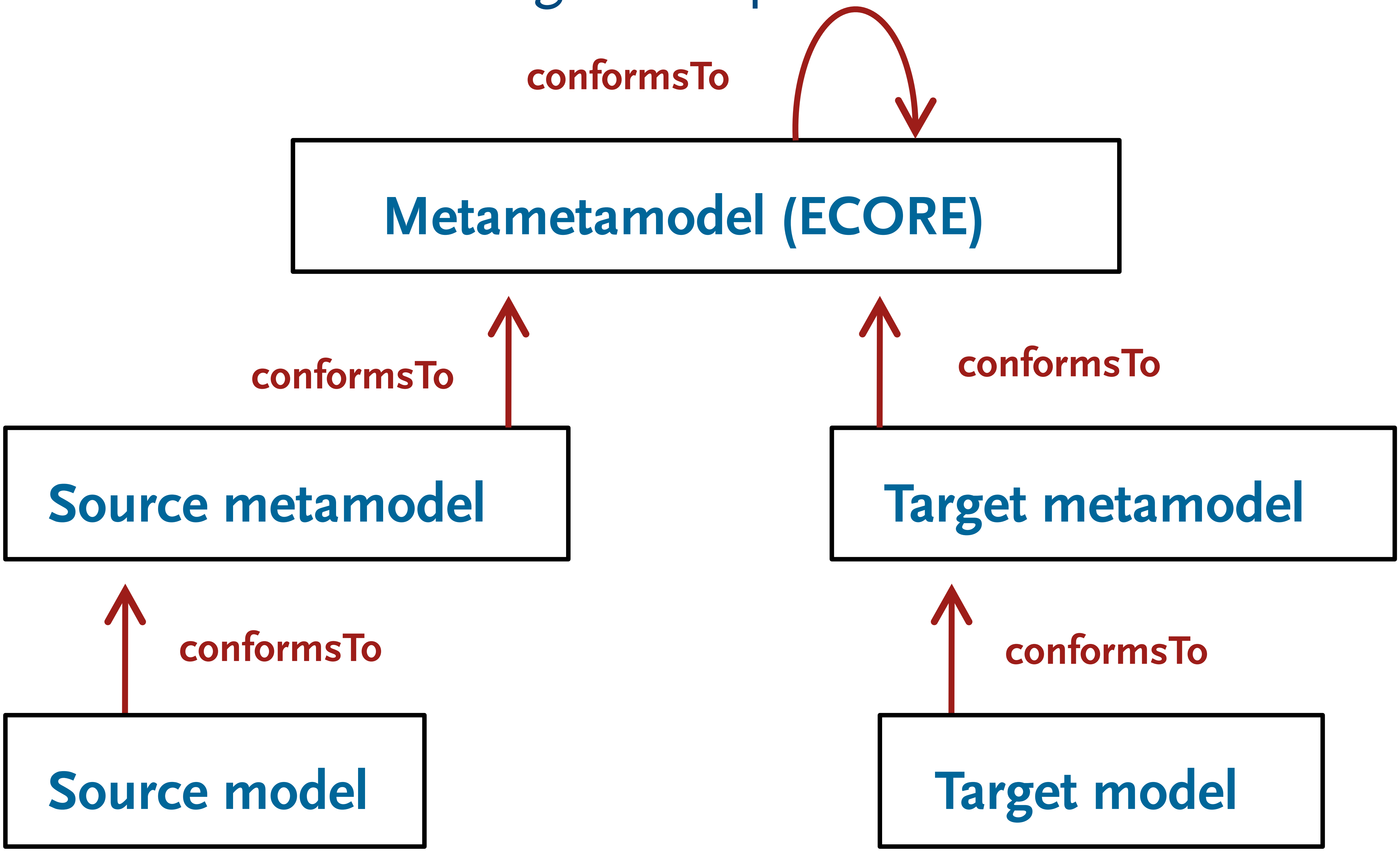
Target metamodel

conformsTo

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Male fullName="Dylan Sailor"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Jim March"/>
  <Male fullName="David Sailor"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

Target model "sample-Persons.ecore"

The general picture



What we need to provide

- In order to achieve the transformation, we need to provide:
  - A source metamodel in KM3 ("Families").
  - A source model (in XMI) conforming to "Families".
  - A target metamodel in KM3 ("Persons").
  - A transformation model in ATL ("Families2Persons").
- When the ATL transformation is executed, we obtain:
  - A target model (in XMI) conforming to "Persons".

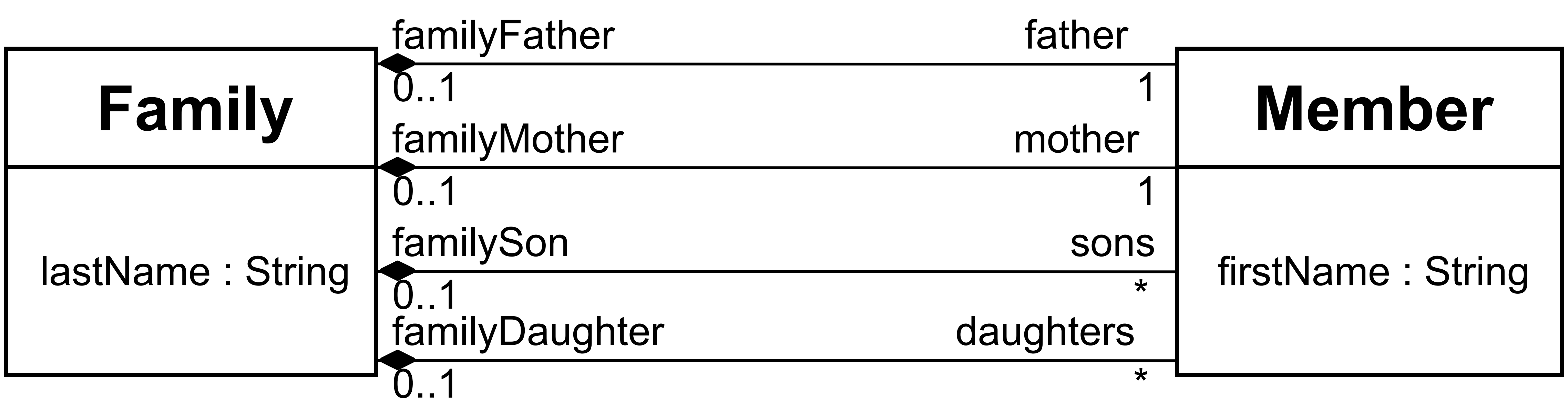
Definition of the source metamodel "Families"

What is “Families”:

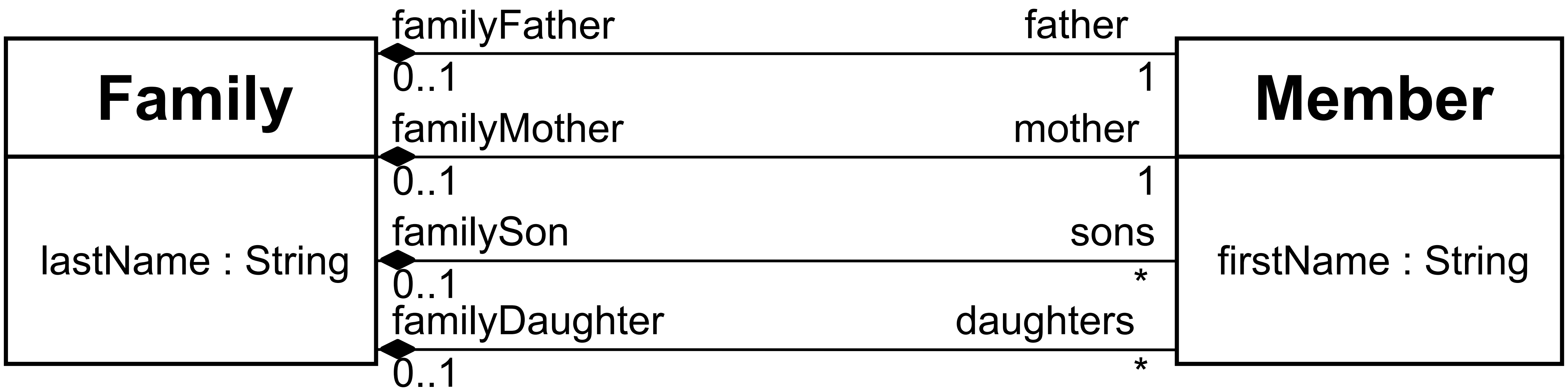
- A collection of families.
- Each family has a name and is composed of members:
  - A father
  - A mother
  - Several sons
  - Several daughters
- Each family member has a first name.

Family March
- Father: Jim
- Mother: Cindy
- Son: Brandon
- Daughter: Brenda

Family Sailor
- Father: Peter
- Mother: Jackie
- Sons: David, Dylan
- Daughter: Kelly

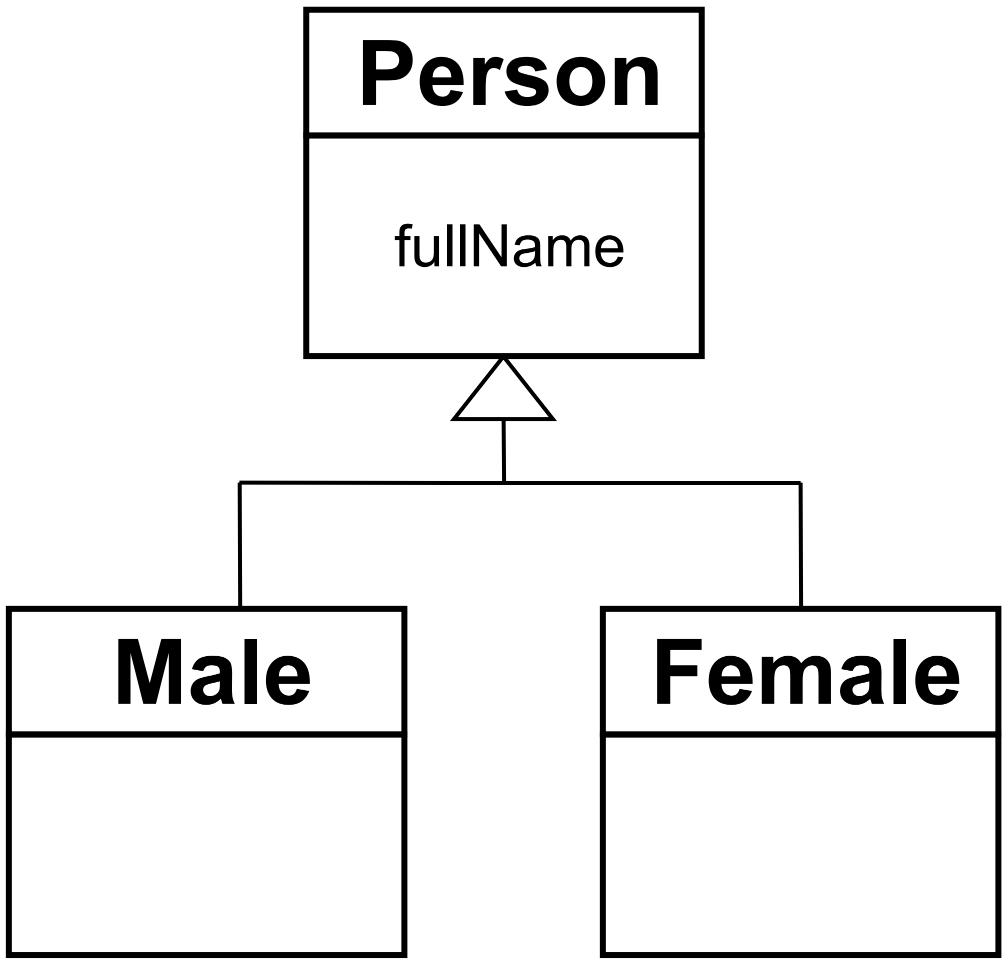


"Families" metamodel (visual presentation and KM3)



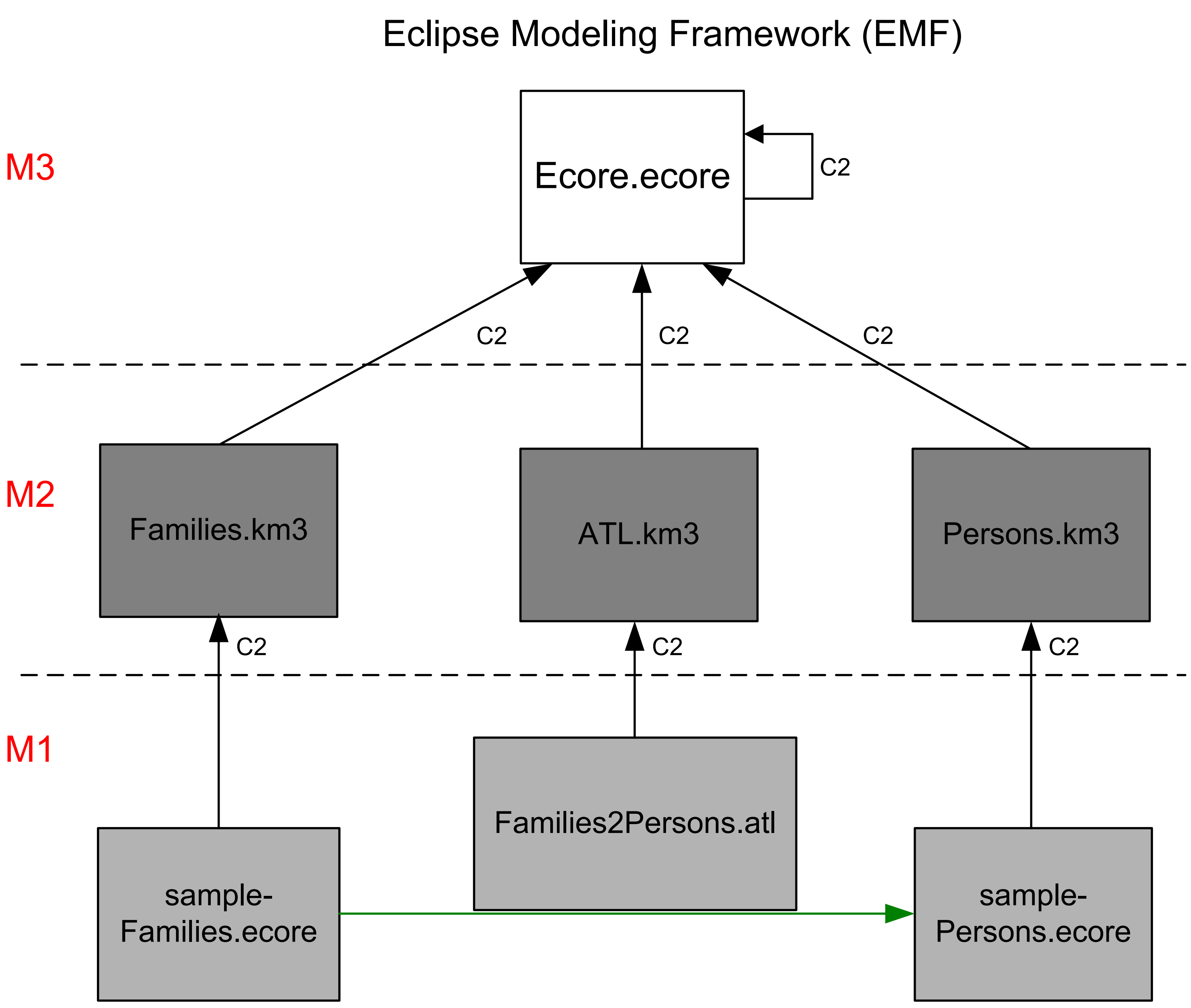

```
package Families {

	class Family {
		attribute lastName : String;
		reference father container : Member oppositeOf familyFather;
		reference mother container : Member oppositeOf familyMother;
		reference sons[*] container : Member oppositeOf familySon;
		reference daughters[*] container : Member oppositeOf familyDaughter;
	}

	class Member {
		attribute firstName : String;
		reference familyFather[0-1] : Family oppositeOf father;
		reference familyMother[0-1] : Family oppositeOf mother;
		reference familySon[0-1] : Family oppositeOf sons;
		reference familyDaughter[0-1] : Family oppositeOf daughters;
	}

}

package PrimitiveTypes {
	datatype String;
}
```

"Persons" metamodel (visual presentation and KM3)



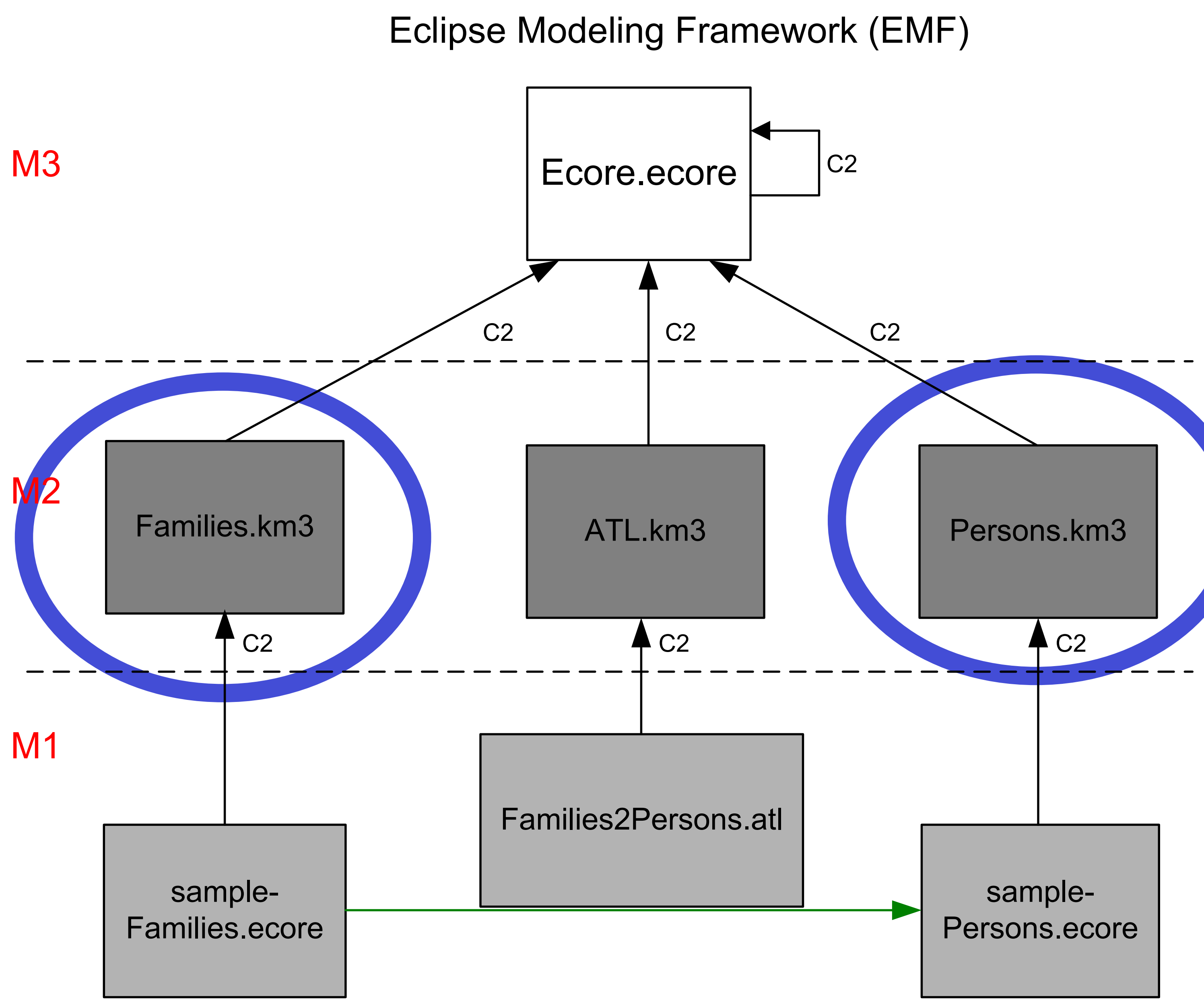

```
package Persons {

    abstract class Person {
        attribute fullName : String;
    }

    class Male extends Person { }

    class Female extends Person { }

}

package PrimitiveTypes {
    datatype String;
}
```

The big picture



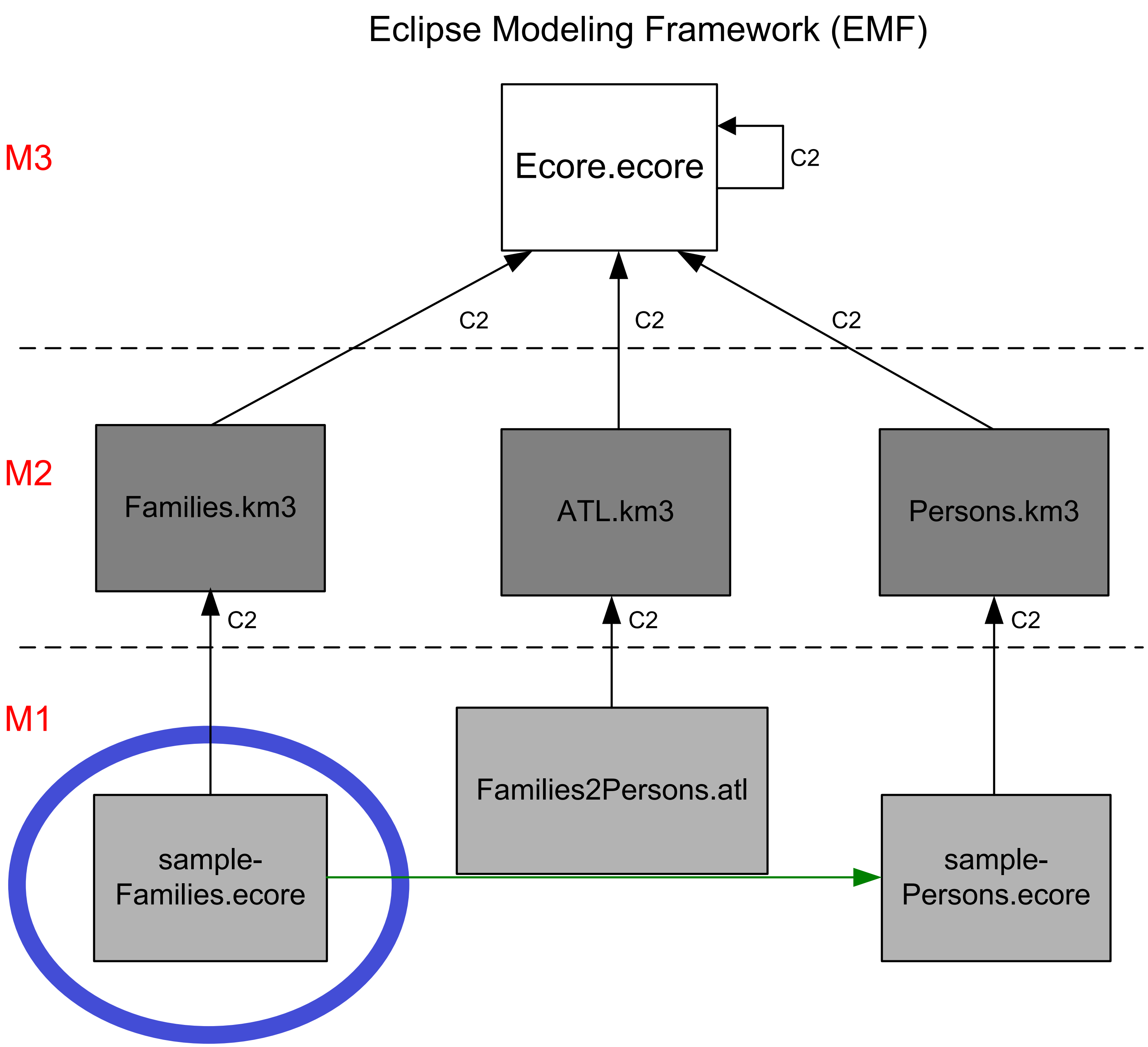
- Our goal in this mini-tutorial is to write the ATL transformation, stored in the "Families2Persons" file.
- Prior to the execution of this transformation the resulting file "sample-Persons.ecore" does not exist. It is created by the transformation.
- Before defining the transformation itself, we need to define the source and target metamodels ("Families.km3" and "Person.KM3").
- We take for granted that the definition of the ATL language is available (supposedly in the "ATL.km3" file).
- Similarly we take for granted that the environment provides the recursive definition of the metametamodel (supposedly in the "Ecore.ecore" file).

Families to Persons Architecture



- Families and Persons metamodels have been created previously.
- They have been written in the KM3 metamodel specification DSL (Domain Specific Language).

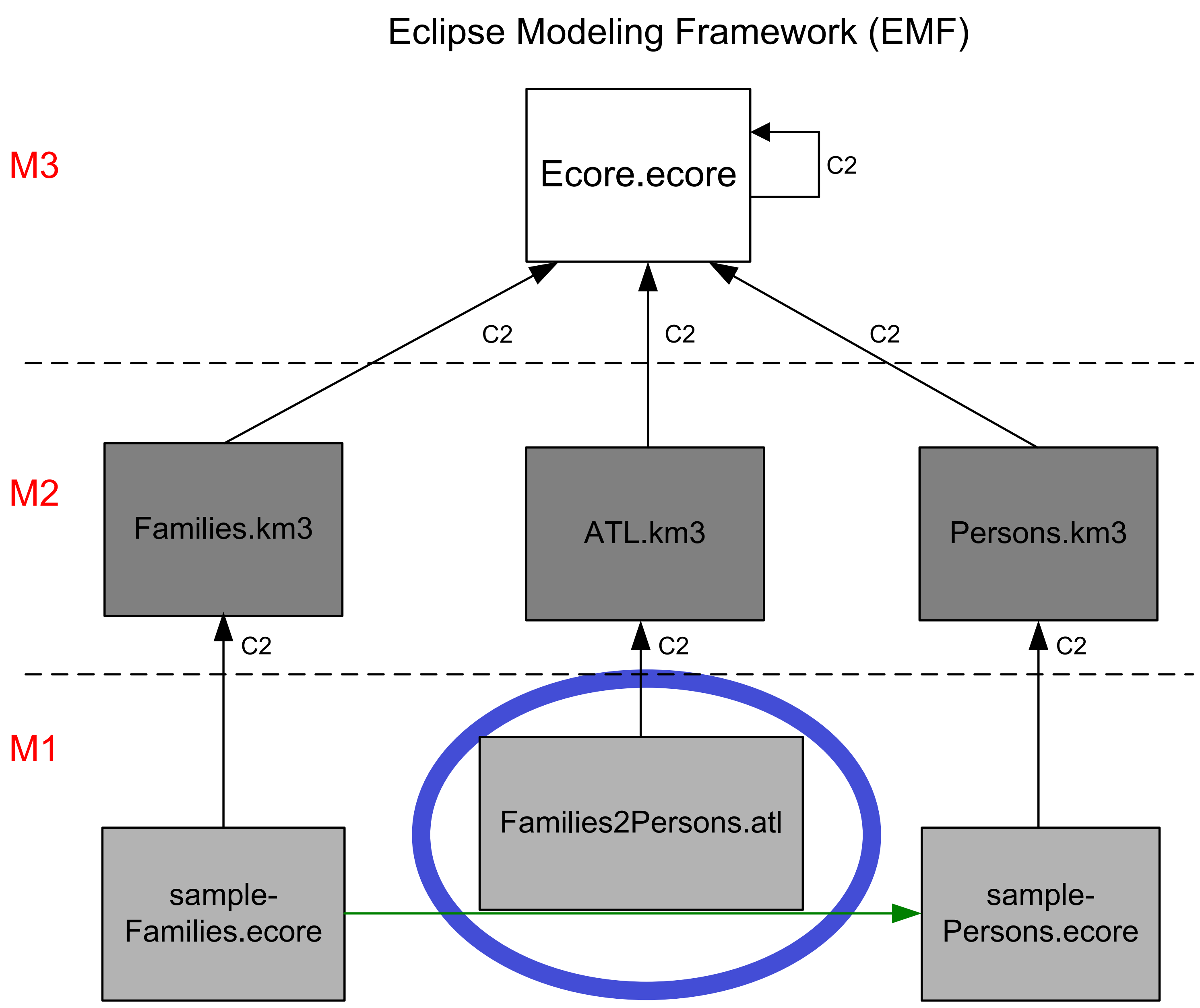
Families to Persons Architecture



- The following file is the sample that we will use as source model in this use case:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://
www.omg.org/XMI" xmlns="Families">
  <Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <sons firstName="David"/>
    <sons firstName="Dylan"/>
    <daughters firstName="Kelly"/>
  </Family>
</xmi:XMI>
```

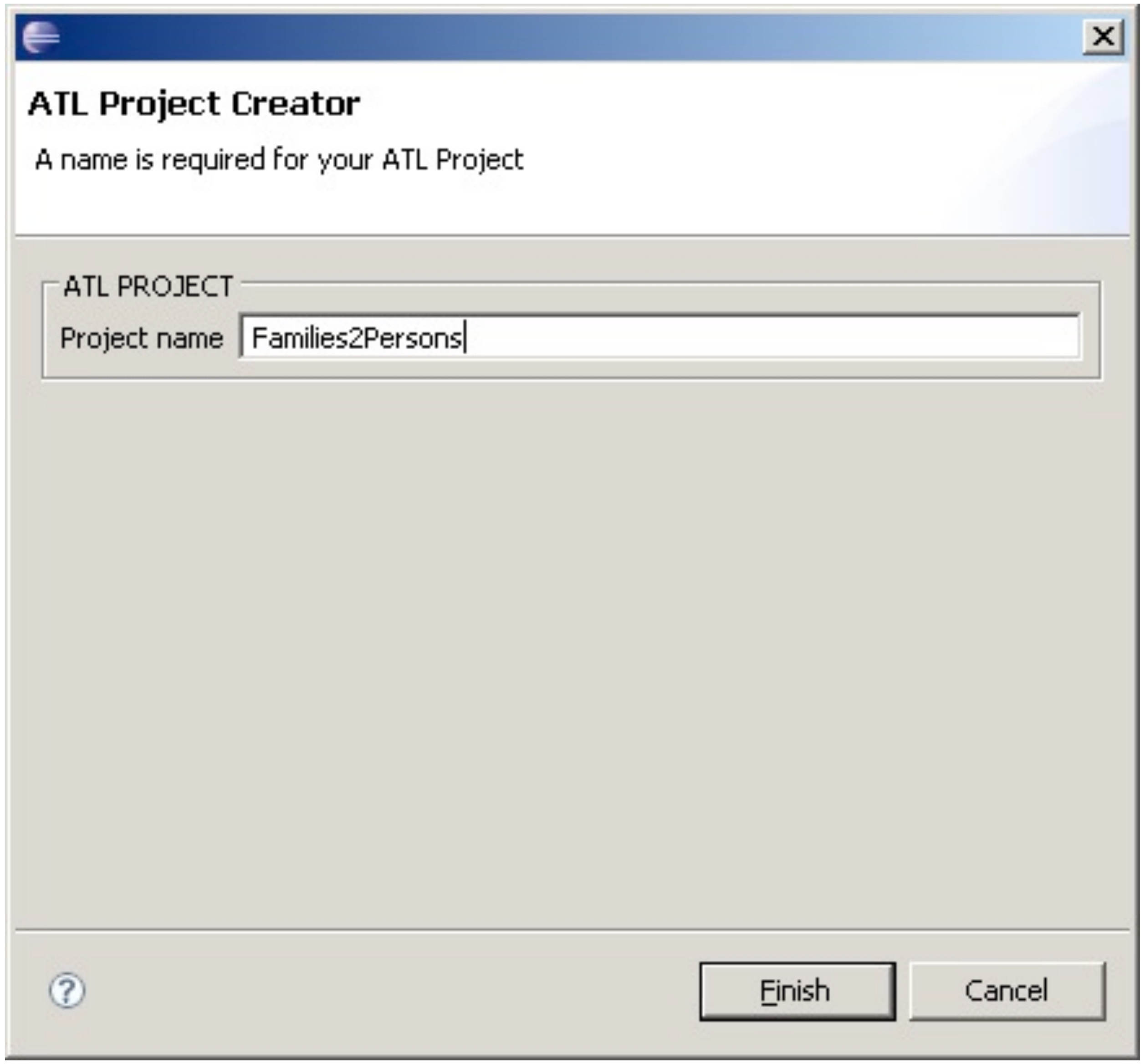
Families to Persons Architecture



- Now, let us start the creation of the ATL transformation Families2Persons.atl.
- We suppose the ATL environment is already installed.
- The creation of the ATL transformation will follow several steps as described in the next slides.

Families to Persons: project creation

- First we create an ATL project by using the ATL Project Wizard.

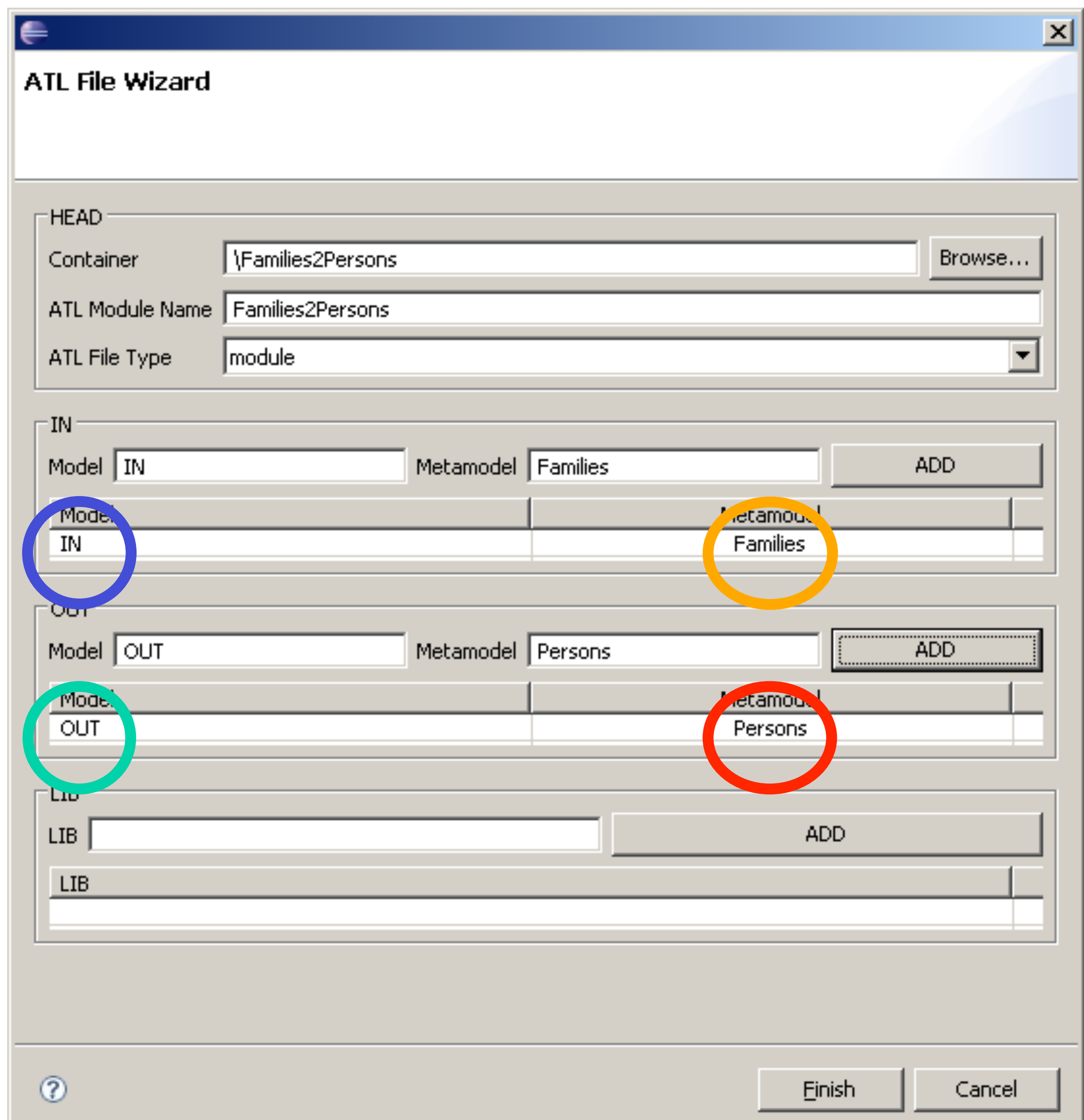


Families to Persons: ATL transformation creation

- Next we create the ATL transformation. To do this, we use the ATL File Wizard. This will generate automatically the header section.

IN:
Name of the source model in the transformation

OUT:
Name of the target model in the transformation



Families:
Name of the source metamodel in the transformation

Persons:
Name of the target metamodel in the transformation

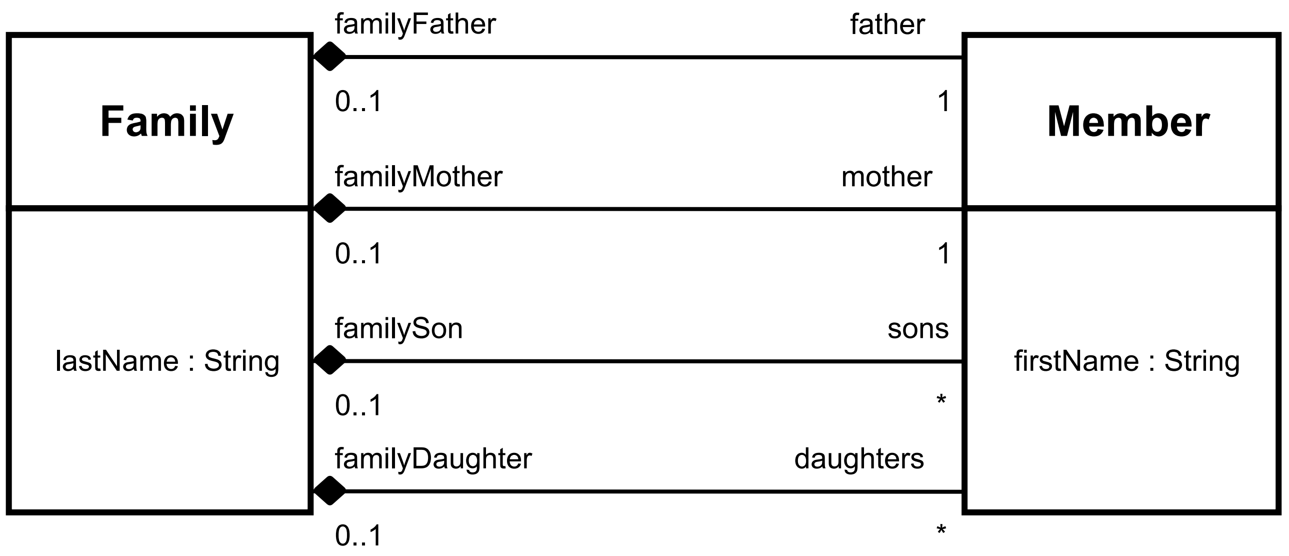
Families to Persons: header section

- The header section names the transformation module and names the variables corresponding to the source and target models ("IN" and "OUT") together with their metamodels ("Persons" and "Families") acting as types. The header section of "Families2Persons" is:

```
module Families2Persons;
create OUT : Persons from IN : Families;
```

Families to Persons: helper "isFemale()"

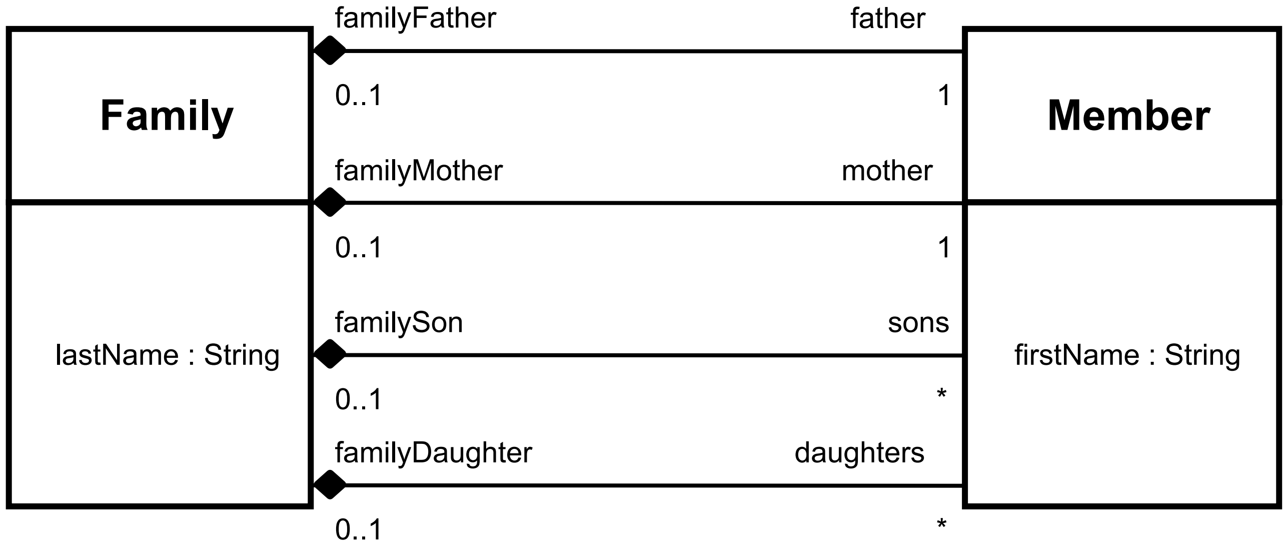
- A helper is an auxiliary function that computes a result needed in a rule.
- The following helper "isFemale()" computes the gender of the current member:




```
helper context Families!Member def: isFemale() : Boolean =
      if not self.familyMother.oclIsUndefined() then
            true
      else
            if not self.familyDaughter.oclIsUndefined() then
                  true
            else
                  false
            endif
      endif;
```

Families to Persons: helper "familyName"

- The family name is not directly contained in class “Member”. The following helper returns the family name by navigating the relation between “Family” and “Member”:




```
helper context Families!Member def: familyName : String =
	if not self.familyFather.oclIsUndefined() then
		self.familyFather.lastName
	else
		if not self.familyMother.oclIsUndefined() then
			self.familyMother.lastName
		else
			if not self.familySon.oclIsUndefined() then
				self.familySon.lastName
			else
				self.familyDaughter.lastName
			endif
		endif
	endif;
```

Families to Persons: writing the rules

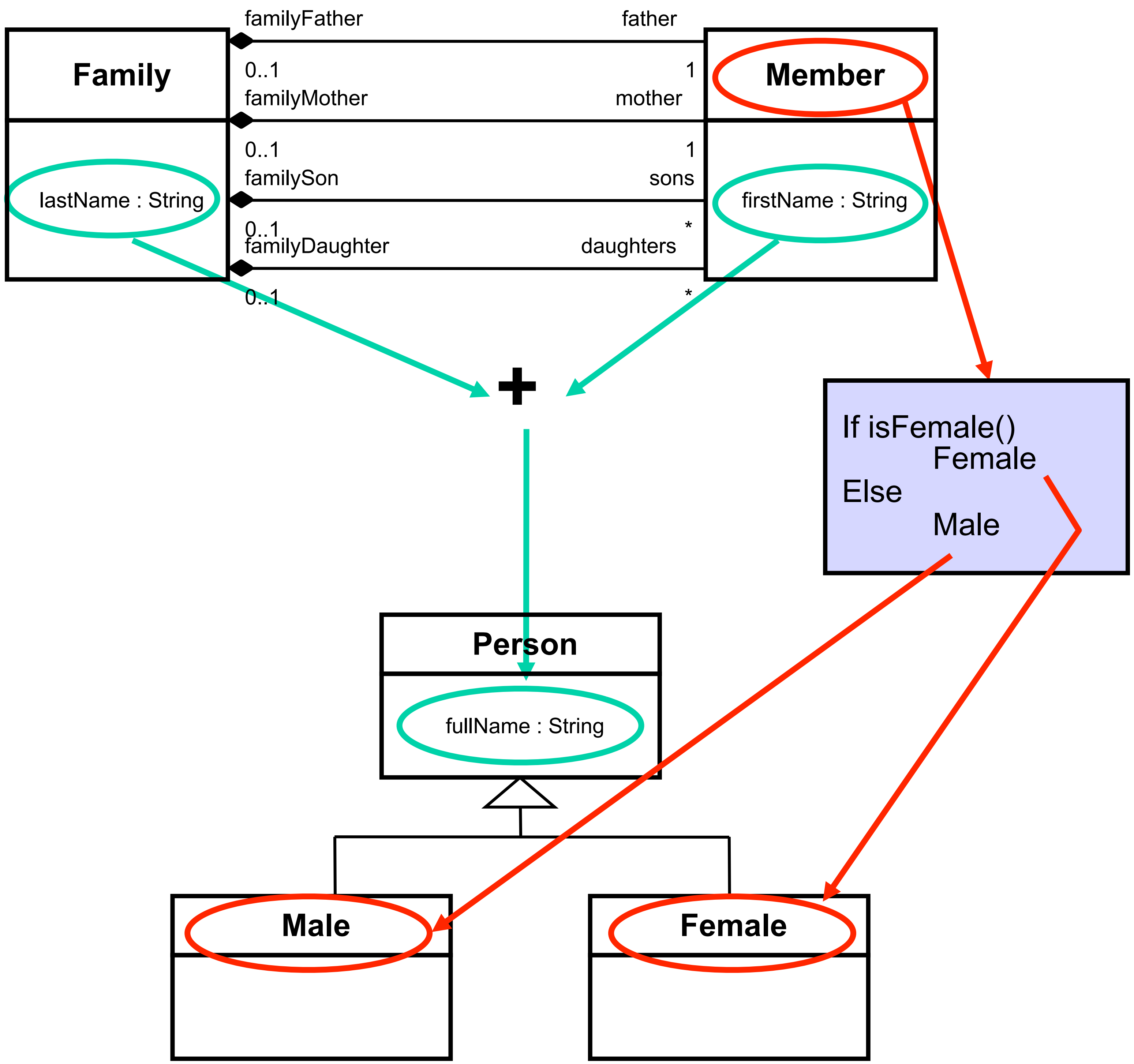
- After the helpers we now write the rules:
  - Member to Male

```
rule Member2Male {
      from
            s : Families!Member (not s.isFemale())
      to
            t : Persons!Male (
                  fullName <- s.firstName + ' ' + s.familyName
            )
}
```

  - Member to Female

```
rule Member2Female {
      from
            s : Families!Member (s.isFemale())
      to
            t : Persons!Female (
                  fullName <- s.firstName + ' ' + s.familyName
            )
}
```

Summary of the Transformation



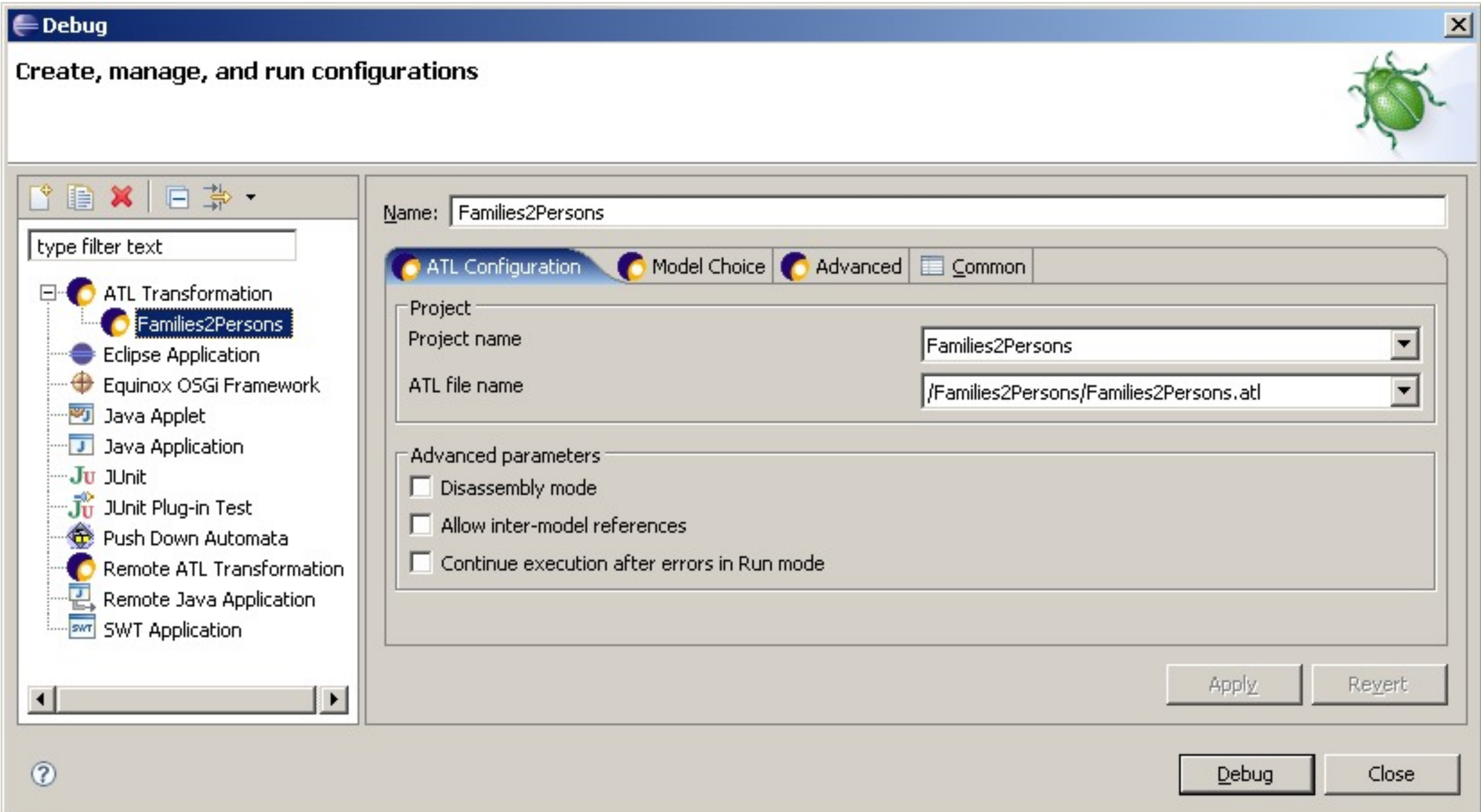
- For each instance of the class "Member" in the IN model, create an instance in the OUT model.
- If the original "Member" instance is a "mother" or one of the "daughters" of a given "Family", then we create an instance of the "Female" class in the OUT model.
- If the original "Member" instance is a "father" or one of the "sons" of a given "Family", then we create an instance of the "Male" class in the OUT model.
- In both cases, the "fullname" of the created instance is the concatenation of the Member "firstName" and of the Family "lastName", separated by a blank.

Families to Persons Architecture

- Once the ATL transformation “Families2Persons” is created, we can execute it to build the OUT model.

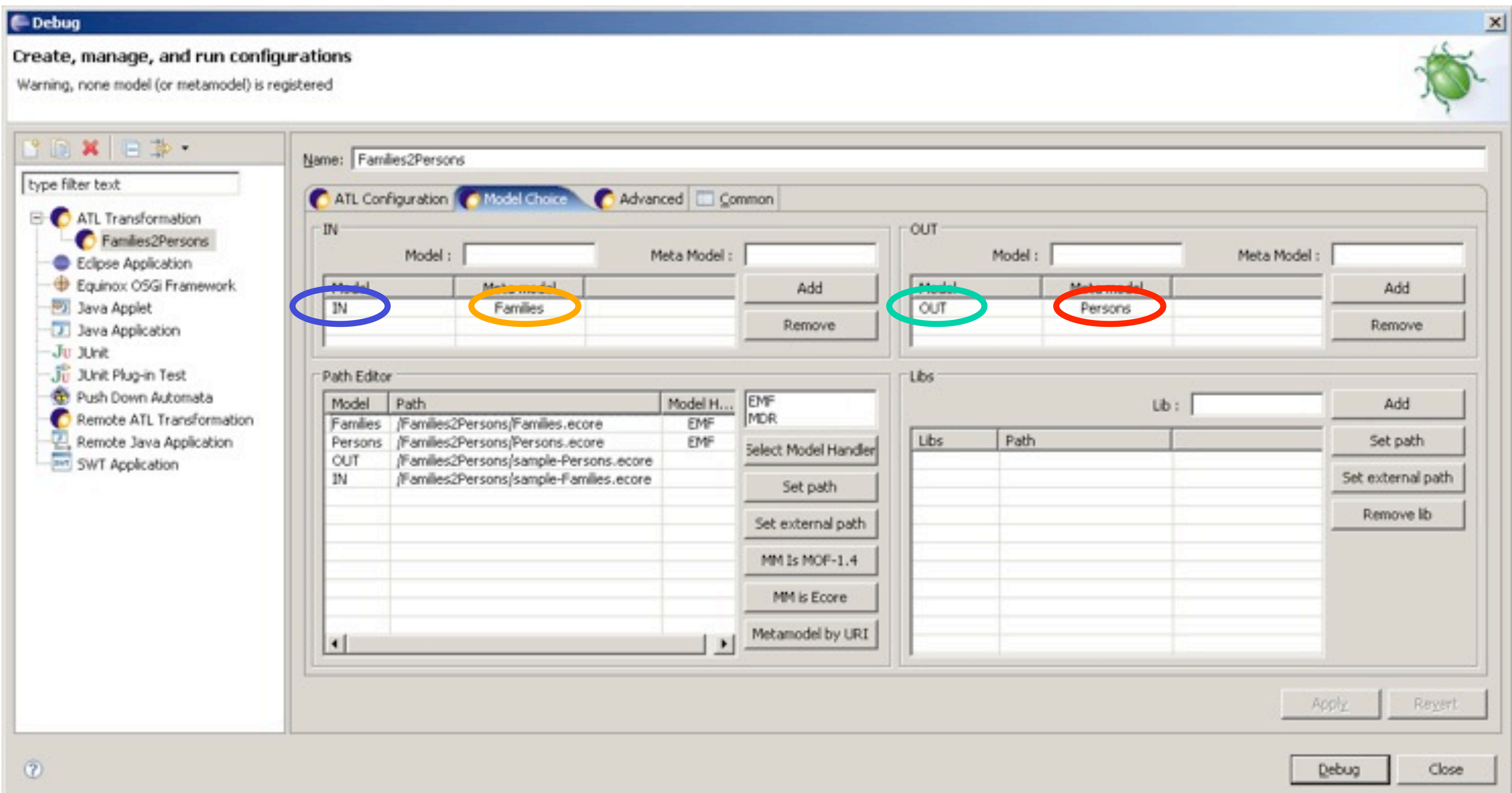
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Male fullName="Dylan Sailor"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Jim March"/>
  <Male fullName="David Sailor"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

ATL Launch Configuration



ATL Launch Configuration

```
module Families2Persons;
create OUT : Persons from IN : Families;
```



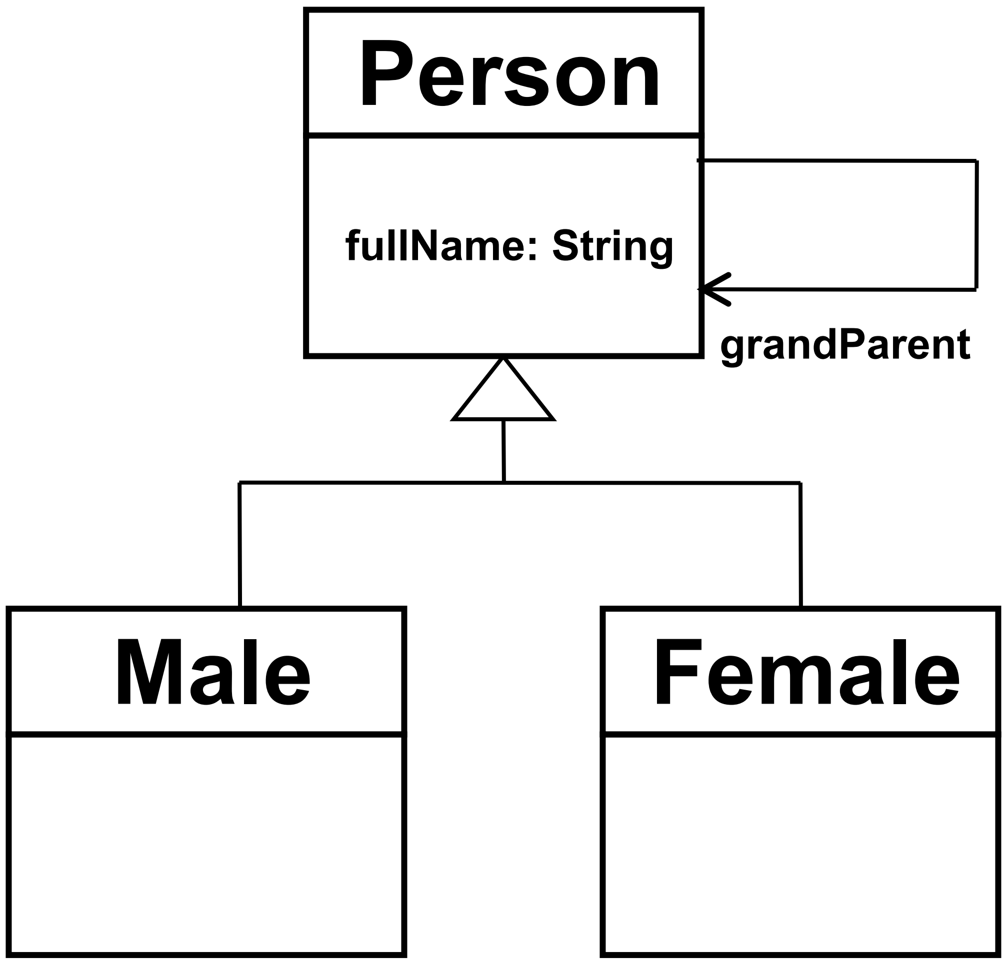
Summary

- We have presented here a "hello world" level basic ATL transformation.
- This is not a recommendation on how to program in ATL, just an initial example.
- Several questions have not been answered
  - Like how to transform a text into an XMI-encoded model.
  - Or how to transform the XMI-encoded result into text.
- For any further questions, see the documentation mentioned in the resource page (FAQ, Manual, Examples, etc.).

ATL Resource page

- ATL Home page
  - http://www.eclipse.org/m2m/atl/
- ATL Documentation page
  - http://www.eclipse.org/m2m/atl/doc/
- ATL Newsgroup
  - news://news.eclipse.org/eclipse.modeling.m2m
- ATL Wiki
  - http://wiki.eclipse.org/index.php/ATL

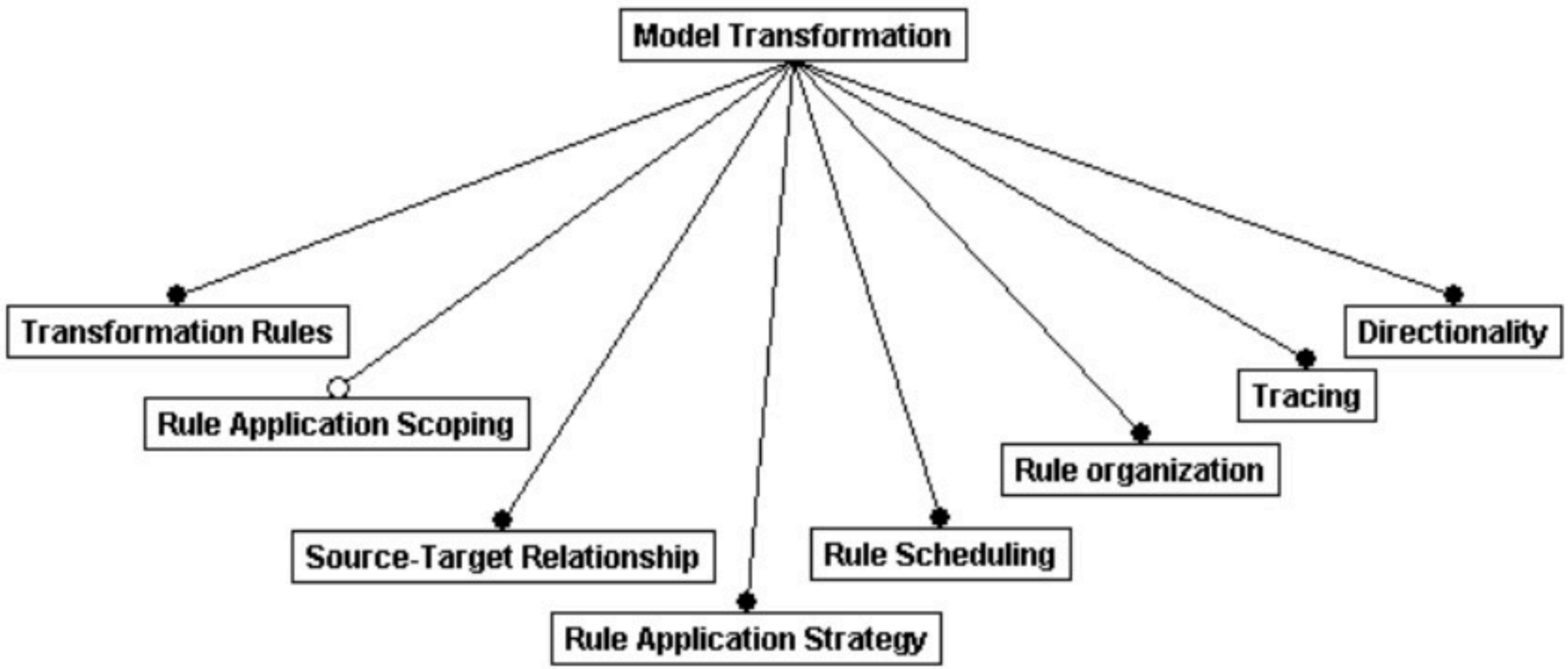
Working on the example



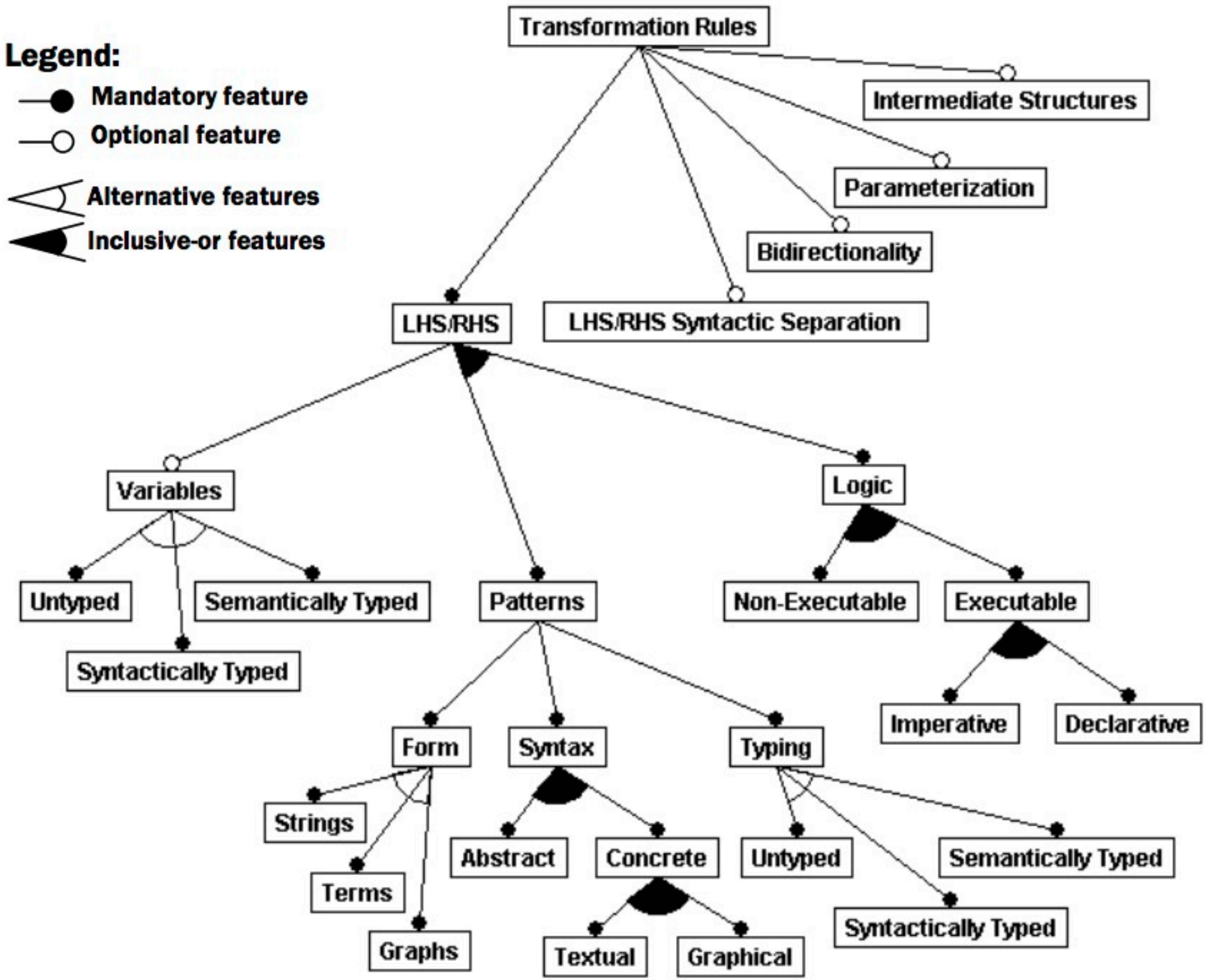
- There are a lot of exercise questions that could be based on this simple example.
- For example, modify the target metamodel as shown and compute the "grandParent" for any Person.

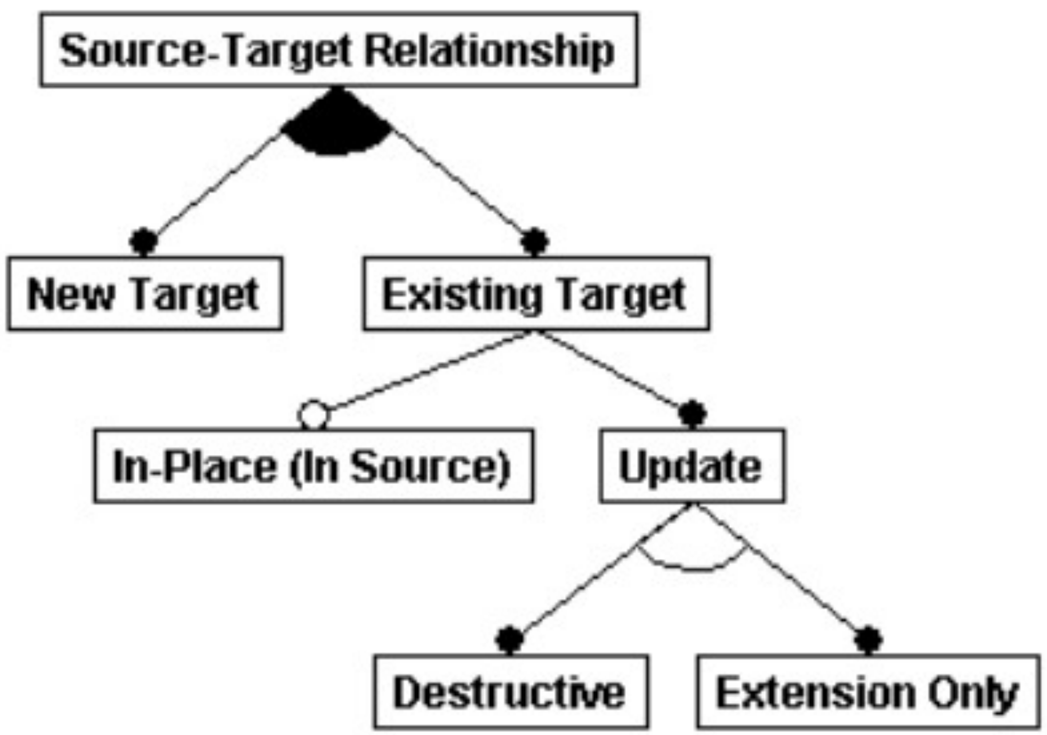
Model-Transformations

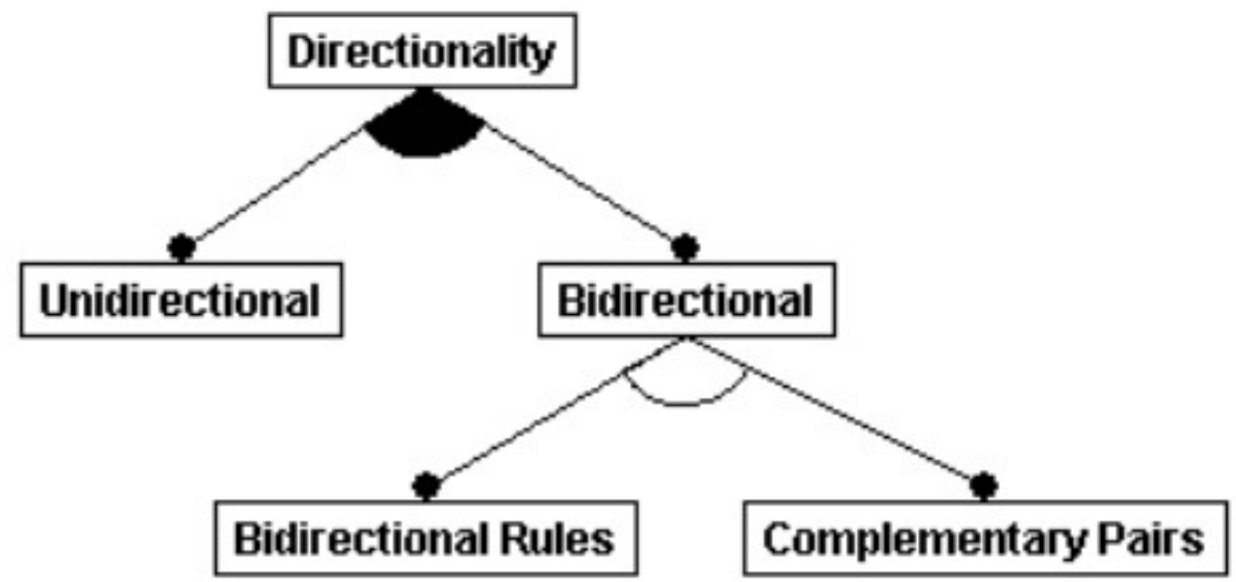
further categorization

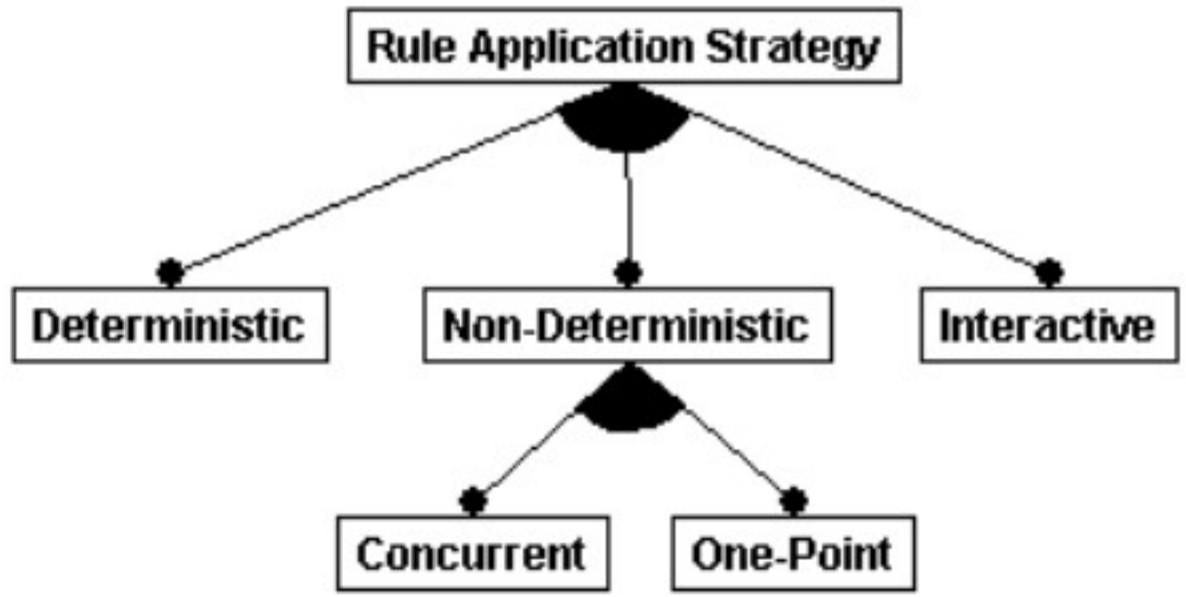


Czarnecki, Helsen: Classification of Model Transformation Approaches, OOSPLA Workshop GTCMDA 2003

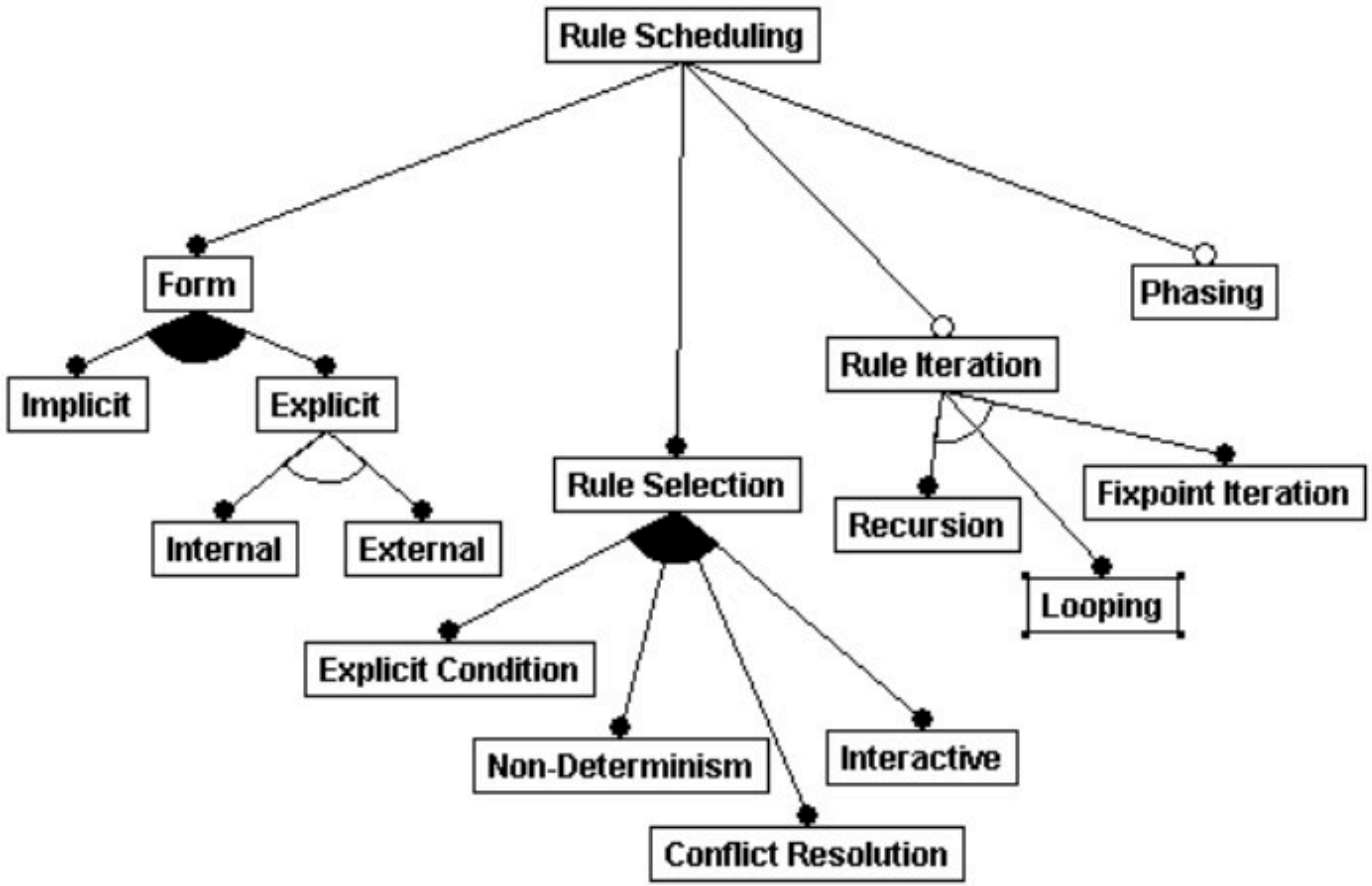

Legend:
Mandatory feature
Optional feature
Alternative features
Inclusive-or features
Transformation Rules
Intermediate Structures
Parameterization
Bidirectionality
LHS/RHS
LHS/RHS Syntactic Separation
Variables
Logic
Untyped
Semantically Typed
Patterns
Non-Executable
Executable
Syntactically Typed
Imperative
Declarative
Form
Syntax
Typing
Strings
Abstract
Concrete
Untyped
Semantically Typed
Terms
Graphs
Textual
Graphical
Syntactically Typed


Source-Target Relationship
New Target
Existing Target
In-Place (In Source)
Update
Destructive
Extension Only


Directionality
Unidirectional
Bidirectional
Bidirectional Rules
Complementary Pairs



Rule Application Strategy
Deterministic
Non-Deterministic
Interactive
Concurrent
One-Point


Rule Scheduling
Form
Implicit
Explicit
Internal
External
Rule Selection
Explicit Condition
Non-Determinism
Conflict Resolution
Interactive
Rule Iteration
Recursion
Looping
Fixpoint Iteration
Phasing

Standard Model-Transformation Language?

- Query View Transformation (QVT), OMG Standard
  - 3 in 1
    - Relational
    - Operational
    - Core
  - obviously there is not “the” model transformation language

External vs. Internal Transformation Languages

- example ATL in Ruby, Scala
- build-in flexibility because model transformation concepts are *added* to an existing GPL, while an external model transformation language *hides* GPL concepts
- Transformation language semantics becomes parameterizable (it is just a GPL Library)

George, Wider, Scheidgen: Type-Safe Model Transformation Languages as internal Scala DSLs, ICMT 2012

Cuadrado, Molina, Tortosa: RubyTL: A Practical, Extensible Transformation Language, ECMDA 2006

Summary

- large number of of model transformation types and model transformation languages
- declarative transformation languages for normative and non-normative specification purposes
- imperative, statically type safe transformation languages, or programming languages for implementation
- no accepted *standard* model transformation languages, internal DSL/GPL hybrids might be *the* approach

Operational vs. Translational

- self-contained
- requires a specific runtime environment almost all the time
- debuggable
- platform specific, requires model processing on that platform
- interpreters can be parameterized for semantic variations
- no generated artifacts, no elaboration of generated artifacts
- no generated artifacts that need to be maintained

- target language dependent
- sometimes requires specific runtime environment
- hard to debug
- “platform independent”, platform does not need to process model
- model transformations can be parameterized for semantic variations
- generated code can be elaborated for semantic variations
- generated code is another asset to maintain

Code-Generation vs. Model-Transformations

- No guaranties that generated artifacts are well-formed or even semantically sound
- In general, no properties can be formally proved
- Structural differences between source and target possible
- Generated artifacts can be syntactically elaborated (there is concrete syntax)

- generated artifacts are at least syntactically sound (no concrete syntax involved)
- In theory and for some techniques, some properties (e.g. retention of properties) can be proved
- Its harder to create structurally different targets with most model transformation languages
- Elaboration of generated artifacts only via external extension

Summary Semantics

- Operational semantics
  - Syntax and runtime structure in EMF
  - EMF-operations as interpreter, Java implementation are the state of the art
- Code Generation
  - Templates or Rich Strings on EMF Models are state of the art
- Model-Transformations
  - many languages for many different types of model transformation
  - GPL/internal DSL hybrids are current standard approach
- Higher-order functions and OCL-style collections go a long way