

Practice of Model Transformation and Model Weaving in the Eclipse Modeling Project with ATL and AMW



Part 1 ATL: the ATLAS Transformation Language

Mikaël Barbero (1)

Marcos Didonet Del Fabro (1)

Juan M. Vara (2)

(1) ATLAS Group (INRIA & LINA), University of Nantes (France)
(2) Kybele Research Group, University Rey Juan Carlos, Madrid (Spain)

Prerequisites

To be able to understand this lecture, a reader should be familiar with the following concepts, languages, and standards:

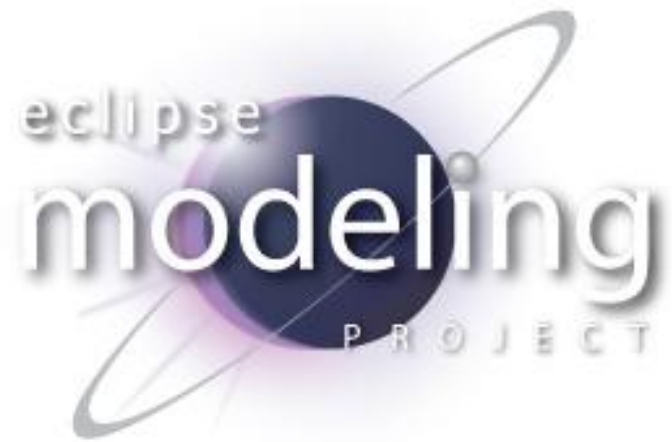
- Model Driven Engineering (MDE)
- The role of model transformations in MDE
- UML
- OCL
- MOF
- Basic programming concepts

Contents

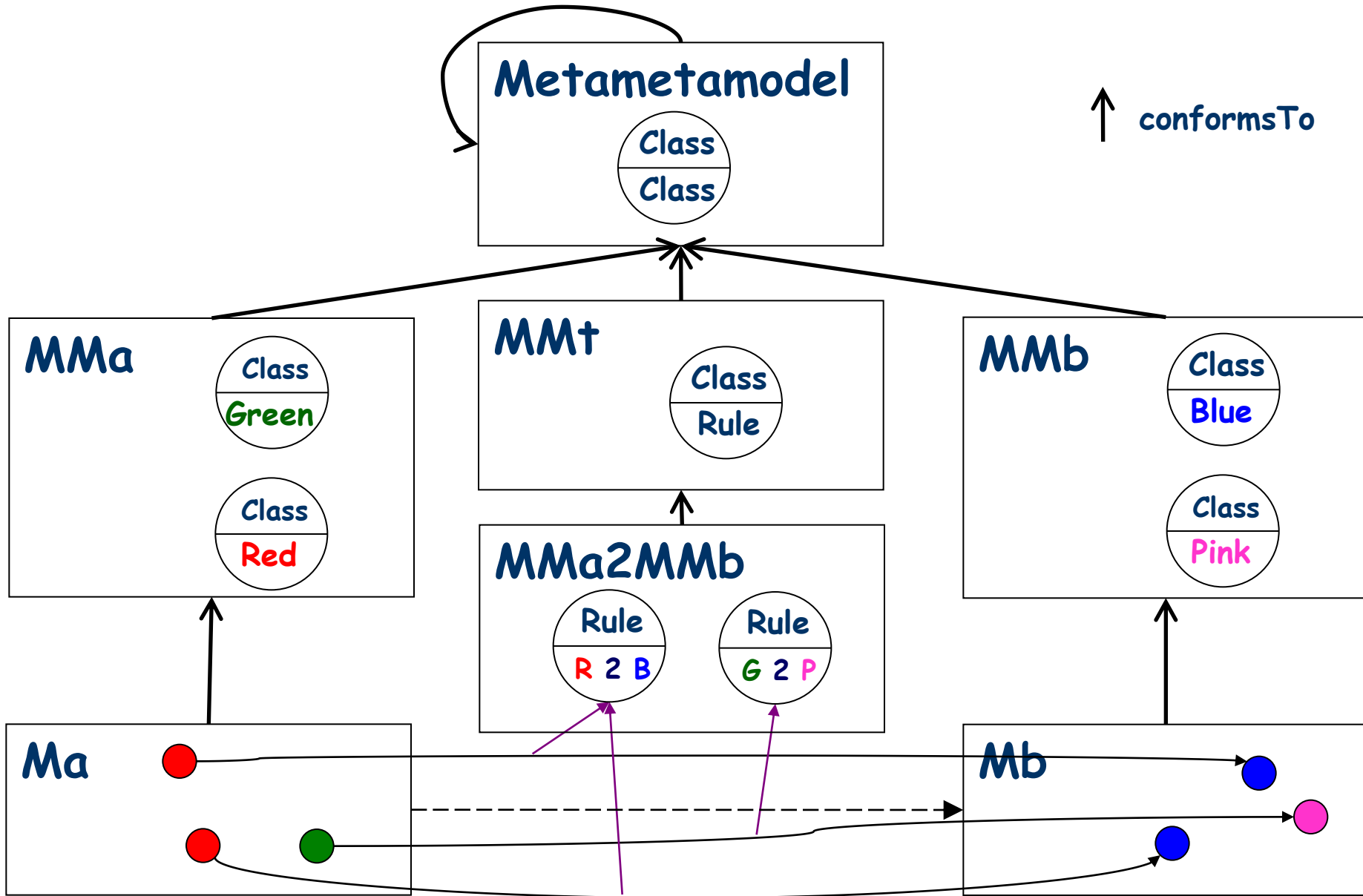
- Introduction
 - Definitions
 - Operational context
- Description of ATL
- Example: Hello World → Families to Persons
- Example: Class to Relational
- Example: Ports
- Conclusion

Introduction: ATL

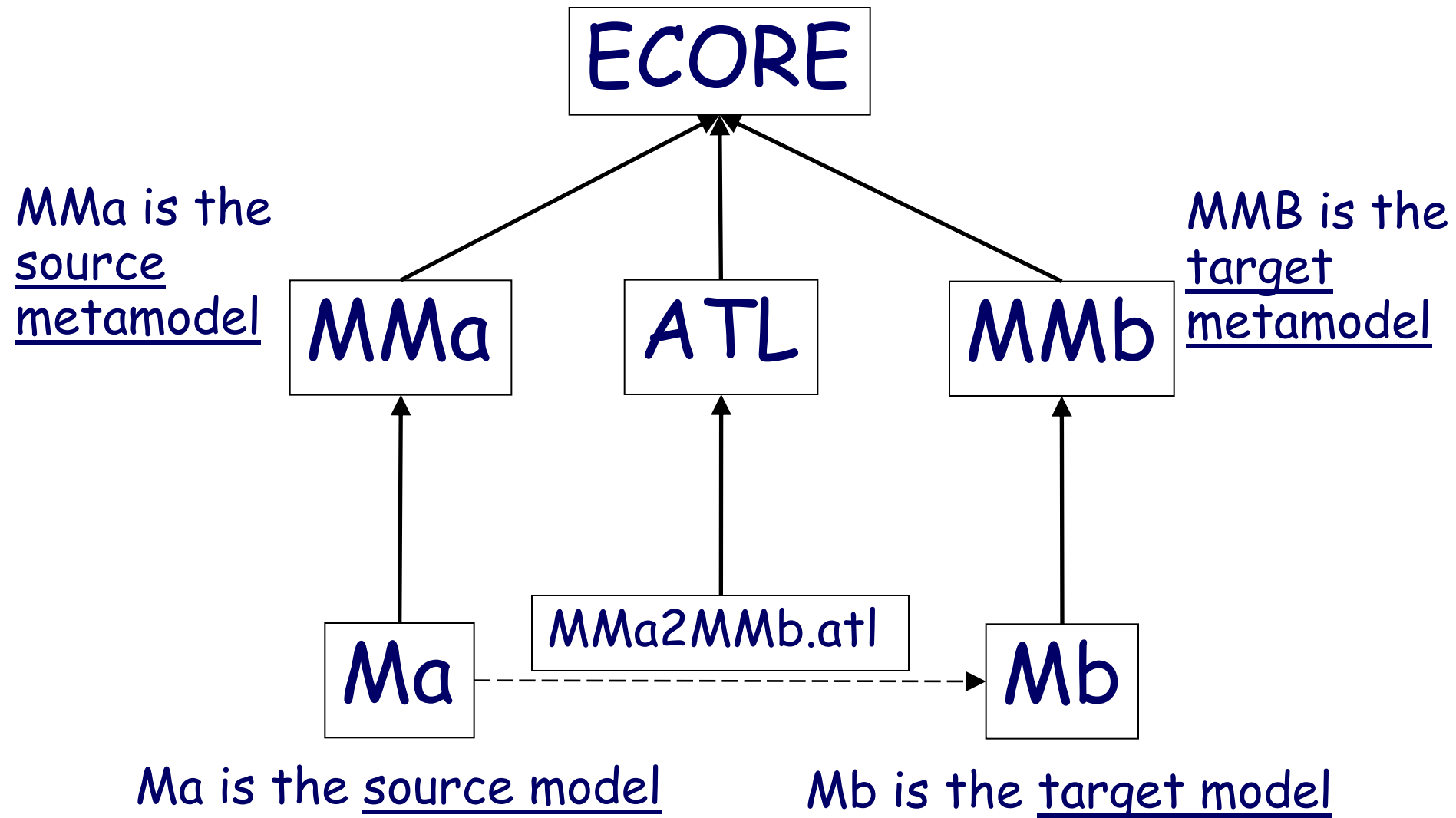
- ATL is a language and a Virtual Machine dedicated to model transformation
- ATL is a component of Eclipse Model-to-Model (M2M) project within the Eclipse Modeling Project (EMP)



Introduction: Model Transformation



Operational context of ATL



Contents

- Introduction
- Description of ATL
 - Resources
 - Overview
 - Source pattern
 - Target pattern
 - Execution order
- Example: Hello World → Families to Persons
- Example: Class to Relational
- Example: Ports
- Conclusion

History

- 1990 : first works on model transformation
- 1998 : initial publication for a Ph.D. thesis at the University of Nantes
- 1998 - 2004 : Implementation
 - CARROLL/MOTOR project (CEA, Thales, INRIA)
 - Collaborative projects : ModelWare, ModelPlex, OpenEmbeDD
- 2004 : Eclipse GMT integration
- 2006 : Industrial solution inside of the Eclipse M2M project

Resources

- Active community
 - Newsgroup: <news://news.eclipse.org/eclipse.modeling.m2m>
 - Wiki: <http://wiki.eclipse.org/ATL>
 - Bugzilla
 - Use cases: <http://www.eclipse.org/m2m/atl/usecases/>
 - Transformations zoo (i.e., a library):
<http://www.eclipse.org/m2m/atl/atlTransformations/>
- Other links :
 - Project page: <http://www.eclipse.org/m2m/atl/>

ATL transformation zoo

ATL Transformations

There are currently 102 model transformations scenarios in this zoo.

ATL Transformations list

- [Ant to Maven: Documentation, Source files](#)
- [AssertionModification: Documentation, Source files](#)
- [ATL to BindingDebugger: Documentation, Source files](#)
- [ATL to Problem: Documentation, Source files](#)
- [ATL to Tracer: Documentation, Source files](#)
- [BibTeXML to DocBook: Documentation, Source files](#)
- [Book to Publication: Documentation, Source files](#)
- [CatalogueModelTransformations: Documentation, Source files](#)
- [Class to Relational: Documentation, Source files](#)
- [Code Clone Tools to SVG: Documentation, Source files](#)
- [CPL to SPL: Documentation, Source files](#)
- [Disaggregation: Documentation, Source files](#)
- [DSL to EMF: Documentation, Source files](#)
- [EliminateRedundantInheritance: Documentation, Source files](#)
- [EMF2KM3 \(Ecore to KM3\): Documentation, Source files](#)

ATL Use Cases

ATL Use Cases

This section provides a set of ATL model transformation use cases covering different domains of application. These use cases are concrete examples of how model to model transformation (M2M) can be applied. Some of these use cases will be simple ones, some others will be more elaborated and will reuse basic use cases or parts of other ones.

Other Eclipse projects have use case section as well for example **AMW** or **MoDisco** (GMT components).

A general description is given for each of these use cases, as well as some more precise documentations. For many of them, prototypes have already been implemented and are directly downloadable from this website.

The general status of each of the provided use cases is indicated by one of the following icon:



Specification Only



Partially Implemented



Reaching Completion

List of use cases

DSLs coordination for Telephony

This work presents a case study of implementing two telephony languages: SPL and CPL. They are partially based on similar vocabularies. However they are very different and have been designed to be used by different people. The use case shows how M2M transformations may be used to map programs conforming to SPL or CPL at different abstraction levels.



Models Validation through Petri nets

This work presents a use case of model transformation using ATL rules to validate MDD's models. This use case considers a simplified process description language, SimplePDL. It then presents a property-driven approach in which SimplePDL process models are translated into Petri nets. SimplePDL behavioral properties are expressed on corresponding Petri nets in LTL (*Linear Temporal Logic*). The Tina toolkit and, in particular, its model-checker, are used to validate process models by checking the expressed properties. This use case has been done by **Benoît Combemale (IRIT)**. This is a collaboration between **IRIT** and **LAAS** labs in Toulouse, France.



Sharing Rules Between OCL/UML and SWRL/OWL

This work presents an implementation of sharing rules between two rule languages from different domains: OCL (Object Constraint Language) together with UML and SWRL (Semantic Web Rule Language) together with OWL. For this integration we used the R2ML (REVERSE I1 Rule Markup Language) metamodel as pivotal metamodel. The R2ML is a general Web rule markup language and it can represent different rule types: integrity, reaction, derivation and production. This work has been done by **Milan Milanovic** in collaboration between the **GOOD OLD AI Laboratory at University of Belgrade, School of Interactive Arts and Technology at Simon Fraser University Surrey** in Canada and **Chair of Internet Technology at Brandenburg University of Technology at Cottbus** in Germany.



ATL as a MT Language

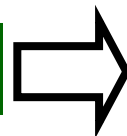
- An ATL program (**module**) is composed of **rules**
 - Define how **SOURCE** elements are mapped to **TARGET** elements
 - Navigate source elements
 - Create and initialize target elements
- Developed over **Eclipse** (Modeling Framework)
 - Provides with an ATL IDE (syntax highlighting, code completion and the like)
- Recommended programming style: **declarative**

ATL Module Structure

- ATL Module

- Header → attributes relative to the transformation

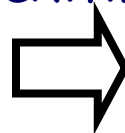
```
module module_name;
create output_models [from|refines] input_models;
```



```
module Book2Publication;
create OUT : Publication from IN : Book;
```

- [Import] → import some exiting libraries

```
uses extensionless_library_file_name;
```



```
uses strings;
```

- Helpers ~ JAVA methods

```
helper context Integer def : max(x : Integer) : Integer = ...;
```

```
helper context Integer def : double() : Integer = self * 2;
```

- Rules

- How target models are generated from the source one

```
rule Author {
  from
    a : MMAuthor!Author
  to
    p : MMPerson!Person (
      name <- a.name,
      surname <- a.surname
    )
}
```

ATL in a nutshell

- Source models and target models are distinct:
 - **Source** models are **read-only** (they can only be navigated, not modified),
 - **Target** models are **write-only** (they cannot be navigated)
 - Referring a target element implies referring the corresponding source element
- The language is a **declarative**-imperative hybrid:
 - Declarative part:
 - **Matched** rules with automatic traceability support,
 - Side-effect free navigation (and query) language: OCL 2.0
 - Imperative part:
 - **Called** rules,
 - Action blocks.

ATL overview - Matched rules

- Core of an ATL transformation
- No explicit call, but matching
- A declarative rule specifies
 - a source pattern to be **matched** in the source models,
 - a target pattern to be created in the target models for each match during rule **application**.
 - the way generated target model elements must be initialized from each matched source elements
- A source model element should not be matched by more than one ATL matched rule

An (almost) 1:1 correspondence between source and target elements

```
rule Author {  
  from  
    to      a : MMAuthor!Author  
           p : MMPerson!Person (  
             name <- a.name,  
             surname <- a.surname  
           )  
}
```

ATL overview - Imperative rules

- An imperative rule is basically a procedure:
 - It is called by its name,
 - It may take arguments,
 - It can contain:
 - A declarative target pattern,
 - An action block (i.e. a sequence of statements),
 - Both.

No
Input Pattern

The imperative code section
is executed after initialization
of the allocated element

```
rule NewPerson (na: String, s_na: String) {  
  to  
    p : MMPerson!Person (  
      name <- na  
    )  
  do {  
    p.surname <- s_na  
  }  
}
```

ATL overview (continued)

- Applying a declarative rule means:
 - Creating the specified target elements,
 - Initializing the properties of the newly created elements.
- There are three types of declarative rules:
 - **Standard** rules that are applied **once** for each match,
 - A given set of elements may only be matched by one standard rule,
 - **Lazy** rules that are applied **as many times** for each match **as** it is referred to from other rules (possibly never for some matches),
 - **Unique lazy** rules that are applied at most once for each match and only if it is referred to from other rules.

ATL overview - Lazy rules

- Called explicitly
- Creates a new output-element on every call

```
lazy rule String2Comment {  
  from s : String  
  to t : UML2!Comment(  
    body <- s)  
}
```

ATL overview - Unique Lazy rules

- Returns the same output for the same input
- Interesting, if there is already an existing rule for an element, that has to be generated in some cases.

```
unique lazy rule Class2TypedTable {  
  from  
    c : UML!Class(c.isSpecialParent())  
  to  
    tt : ORDB4ORA!TypedTable  
      (Name <- c.name.debug('is in Class2TypedTable')  
        + 's<<persistent>>')  
}
```

Declarative rules: source pattern

- The **source pattern** is composed of:
 - A labeled set of types coming from the source metamodels,
 - A **guard** (Boolean expression) used to filter matches.
- A **match** corresponds to a set of elements coming from the source models that:
 - Are of the types specified in the source pattern (one element for each type),
 - Satisfy the guard.

Declarative rules: target pattern

- The **target pattern** is composed of:
 - A labeled set of types coming from the target metamodels,
 - For each element of this set, a set of **bindings**.
 - A binding specifies the initialization of a property of a target element using an expression.
- For each match, the target pattern is applied:
 - Elements are created in the target models (one for each type of the target pattern),
 - Target elements are initialized by executing the bindings:
 - First evaluating their value,
 - Then assigning this value to the corresponding property.

Execution order of declarative rules

- Declarative ATL frees the developer from specifying **execution order**:
 - The order in which rules are matched and applied is not specified.
 - Remark: the match of a lazy or unique lazy rules must be referred to before the rule is applied.
 - The order in which bindings are applied is not specified.
- The execution of declarative rules can however be kept **deterministic**:
 - The execution of a rule cannot change source models
 - ➔ It cannot change a match,
 - Target elements are not navigable
 - ➔ The execution of a binding cannot change the value of another.

ATL Execution

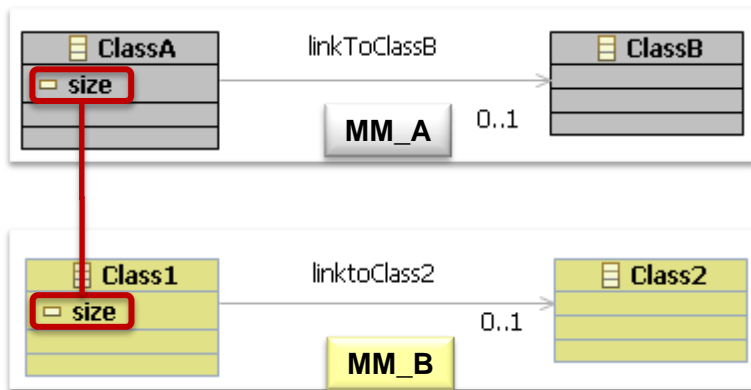
- The execution of an ATL module is organized into three successive phases:
 - A module initialization phase
 - Module attributes initialization
 - Entry point rule
 - A matching phase of the source model elements
 - Testing of matching condition of the declared matched rules
 - If matching, target model elements allocated
 - Traceability links
 - Rule R - Source Pattern P - Target Pattern T
 - A target model elements initialization phase
 - Initialization of the target model elements that have been generated during the previous step

Target Model Elements Initialization

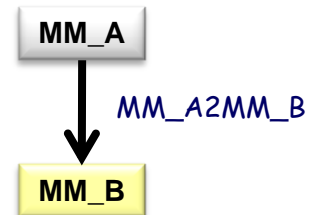
- There are four main cases:
 - Assigning a primitive type value
 - Assigning to a reference a target model element generated by the current rule
 - Assigning to a reference the default target model element of another rule
 - Assigning to a reference a non-default target model element of another rule

Target Model Elements Initialization

- There are four main cases:
 - Assigning a primitive type value

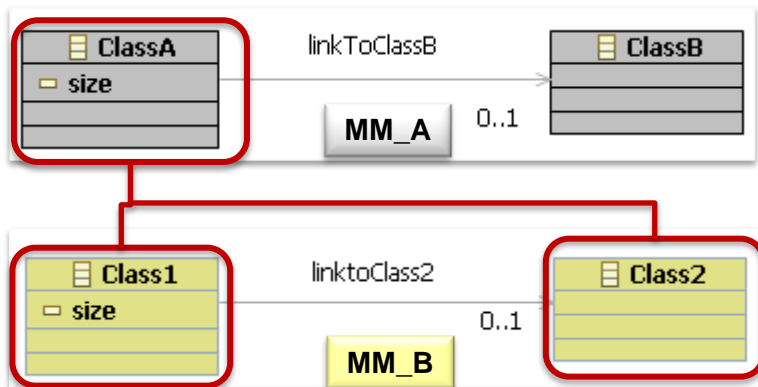


```
rule Case0 {
  from
    input : MM_A!ClassA
  to
    output : MM_B!Class1 (
      size <- input.size * 2
    )
}
```

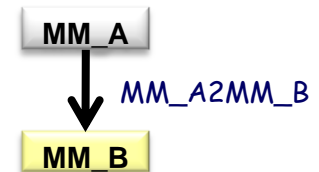


Target Model Elements Initialization (References)

- There are four main cases:
 - Assigning to a reference a target model element generated by the current rule
 - The reference feature is here simply initialized with the local target pattern element that corresponds to the target model element

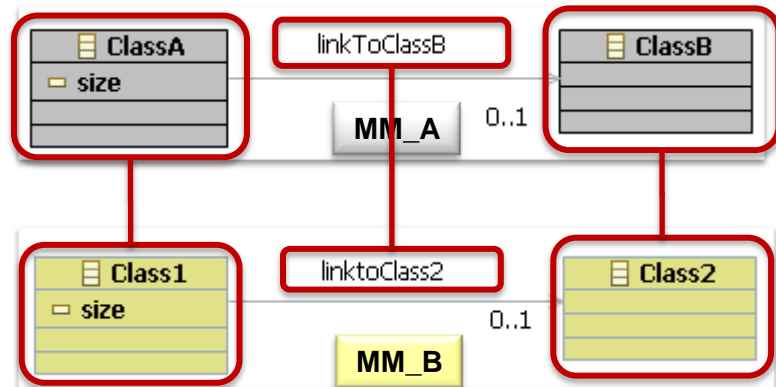


```
rule Case1 {
  from
    input : MM_A!ClassA
  to
    output_1 : MM_B!Class1 (
      linkToClass2 <- output_2
    ),
    output_2 : MM_B!Class2 (
      ...
    )
}
```



Target Model Elements Initialization (References)

- There are four main cases:
 - Assigning to a reference the default target model element of another rule
 - The reference is here initialized with the source model element that is matched by rule *Case2_R2* when generating the target model element *MM_B!Class2*

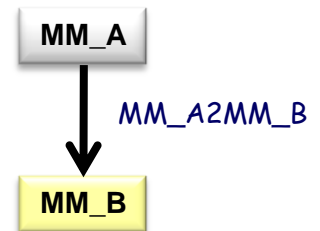


```

rule Case2_R1 {
  from
    input : MM_A!ClassA
  to
    output_1 : MM_B!Class1 {
      linkToClass2 <- i.linkToClassB
    }
}

rule Case2_R2 {
  from
    input : MM_A!ClassB
  to
    output_1 : MM_B!Class2 {
      ...
    }
}

```



Target Model Elements Initialization (References)

- **resolveTemp(var, target_pattern_name)** operation:
 - To point to any of the target model elements that will be generated from a given source model element
 - *var*: ATL variable that contains the source model element from which the searched target model element is produced
 - *target_pattern_name*: a string value that encodes the name of the target pattern element that maps the provided source model element into the searched target model element
 - Note that we do not specify which the matched rule is
 - A source model element should not be matched by more than one matched rule
 - The concerned matched rule can be univoquely derived from the specified source model element

Target Model Elements Initialization (References)

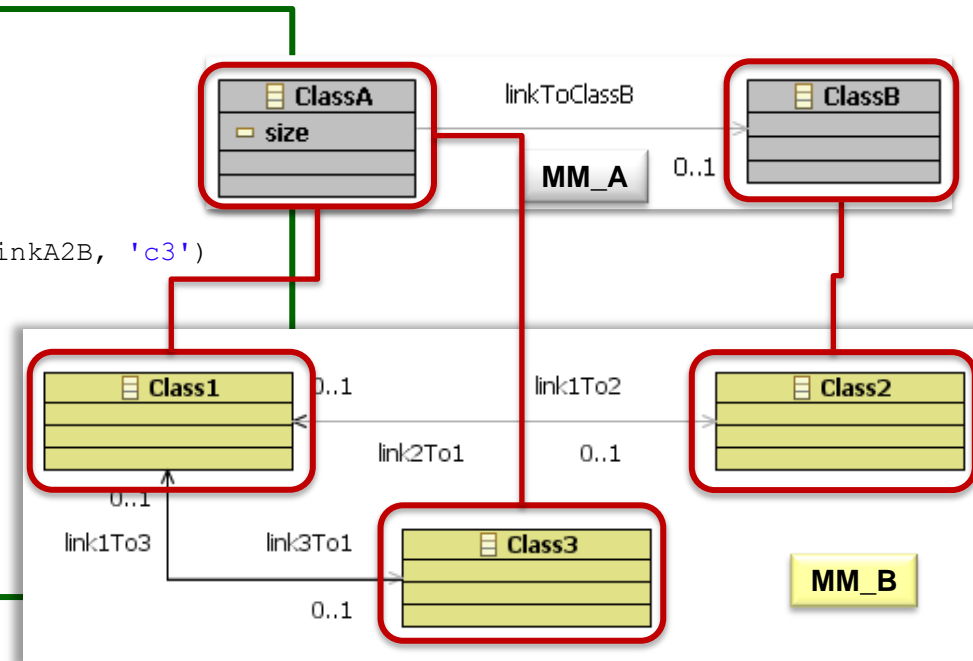
- There are four main cases:
 - Assigning to a reference a non-default target model element of another rule
 - The reference is here initialized by invoking the operation **resolveTemp()** with the source model element ("i.linkToClassB") and the name of the target pattern ("output_n") as arguments.

```

rule Case3_R1 {
from
  input : MM_A!ClassA
to
  output : MM_B!Class1 (
    name <- input.name,
    link1To2 <- input.linkA2B,
    link1To3 <- thisModule.resolveTemp(input.linkA2B, 'c3')
  )
}

rule Case3_R2 {
from
  input : MM_A!ClassB
to
  c2 : MM_B!Class2 (name <- 'C2_' + input.name),
  c3 : MM_B!Class3 (name <- 'C3_' + input.name)
}

```



Contents

- Introduction
 - Definitions
 - Operational context
- Description of ATL
- Example: Hello World → Families to Persons
- Example: Class to Relational
- Example: Ports
- Conclusion

Use Case 1

Families to Persons

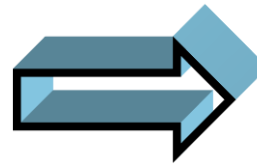
Families to Persons: Overview

- Initially we have a text describing a list of families.
- We want to transform this list into another one describing a list of persons.

Transforming this ...

... into this.

...
Family: Simpsons
Father: Homer
Mother: Marge
Son: Bart
Daughter: Lisa
... other Families



...
Mr. Homer Simpson
Mrs. Marge Simpson
Mr. Bart Simpson
Mrs. Lisa Simpson
... other Persons

Let's suppose these are not text, but models.

Input of the transformation is a model

Family: Simpson
Father: Homer
Mother: Marge
Son: Bart
Daughter: Lisa
Daughter: Maggie
Family: Skywalker
Father: Anakin
Mother: Padme
Son: Luke
Daughter: Leia

This is the text.

```
<?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="Skywalker">
    <father firstName="Anakin"/>
    <mother firstName="Padme"/>
    <sons firstName="Luke"/>
    <daughters firstName="Leia"/>
  </Family>
  <Family lastName="Simpsons">
    <father firstName="Homer"/>
    <mother firstName="Marge"/>
    <sons firstName="Bart"/>
    <daughters firstName="Lisa"/>
    <daughters firstName="Maggie"/>
  </Family>
</xmi:XMI>
```

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. Anakin Skywalker
Mrs. Padme Skywalker
Mr. Luke Skywalker
Mrs. Leia Skywalker
Mr. Homer Simpson
Mrs. Marge Simpson
Mr. Bart Simpson
Mrs. Lisa Simpson
Mrs. Maggie Simpson

This is the text.

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<xmi:XMI xmi:version="2.0"  
  xmlns:xmi="http://www.omg.org/XMI"  
  xmlns="Persons">  
  <Man fullName="Anakin Skywalker"/>  
  <Woman fullName="Padme Skywalker"/>  
  <Man fullName="Luke Skywalker"/>  
  <Woman fullName="Leia Skywalker"/>  
  <Man fullName="Homer Simpsons"/>  
  <Woman fullName="Marge Simpsons"/>  
  <Man fullName="Bart Simpsons"/>  
  <Woman fullName="Lisa Simpsons"/>  
  <Woman fullName="Maggie Simpsons"/>  
</xmi:XMI>
```

This is the corresponding model
(The corresponding XMI file is named
"sample-Persons.ecore").

Each model conforms to a metamodel

Source metamodel

conformsTo

```
<?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="Skywalker">
    <father firstName="Anakin"/>
    <mother firstName="Padme"/>
    <sons firstName="Luke"/>
    <daughters firstName="Leia"/>
  </Family>
  <Family lastName="Simpsons">
    <father firstName="Homer"/>
    <mother firstName="Marge"/>
    <sons firstName="Bart"/>
    <daughters firstName="Lisa"/>
    <daughters firstName="Maggie"/>
  </Family>
</xmi:XMI>
```

Source model
"sample-Families.xmi"

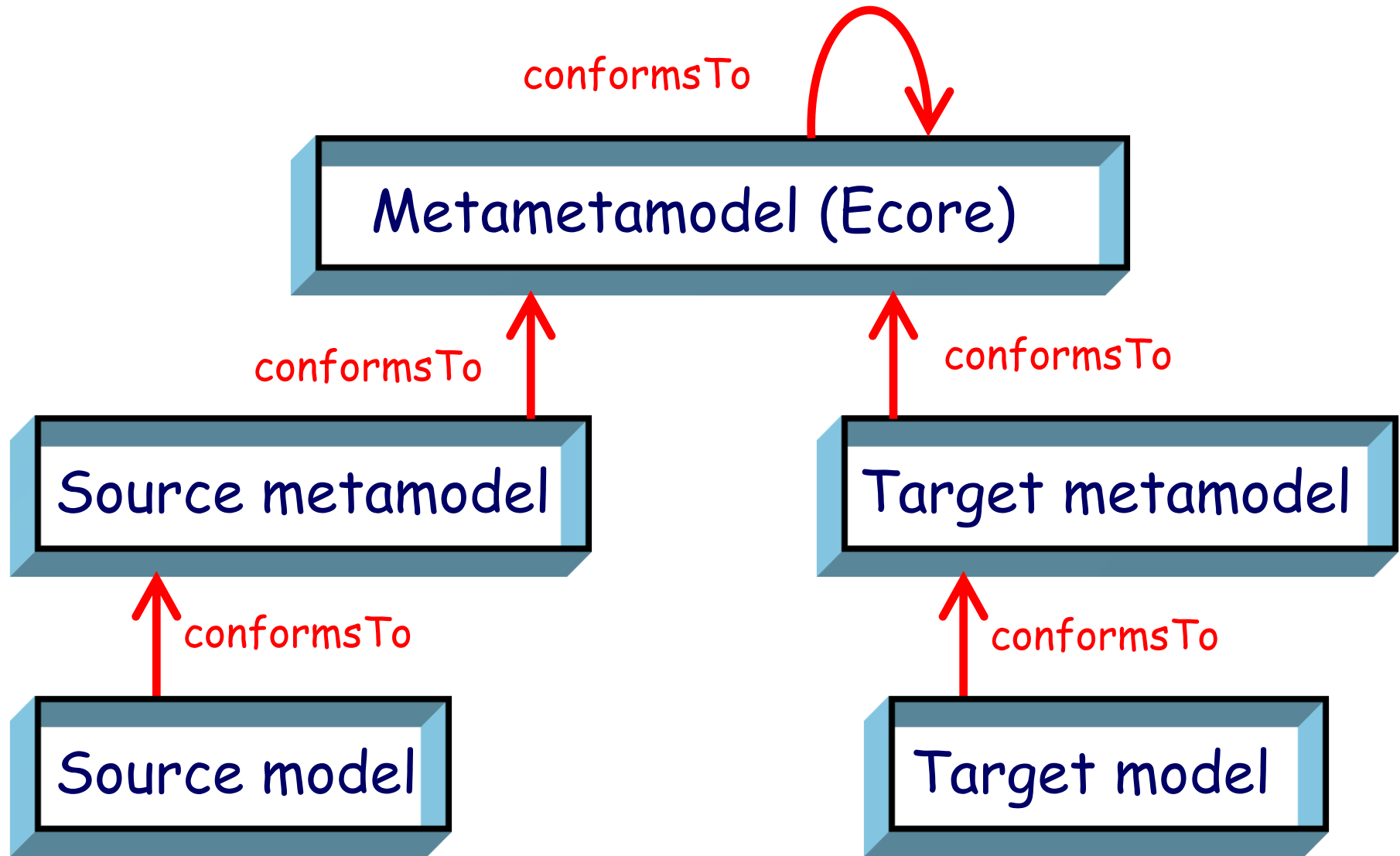
Target metamodel

conformsTo

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Man fullName="Anakin Skywalker"/>
  <Woman fullName="Padme Skywalker"/>
  <Man fullName="Luke Skywalker"/>
  <Woman fullName="Leia Skywalker"/>
  <Man fullName="Homer Simpsons"/>
  <Woman fullName="Marge Simpsons"/>
  <Man fullName="Bart Simpsons"/>
  <Woman fullName="Lisa Simpsons"/>
  <Woman fullName="Maggie Simpsons"/>
</xmi:XMI>
```

Target model
"sample-Persons.xmi"

The general picture

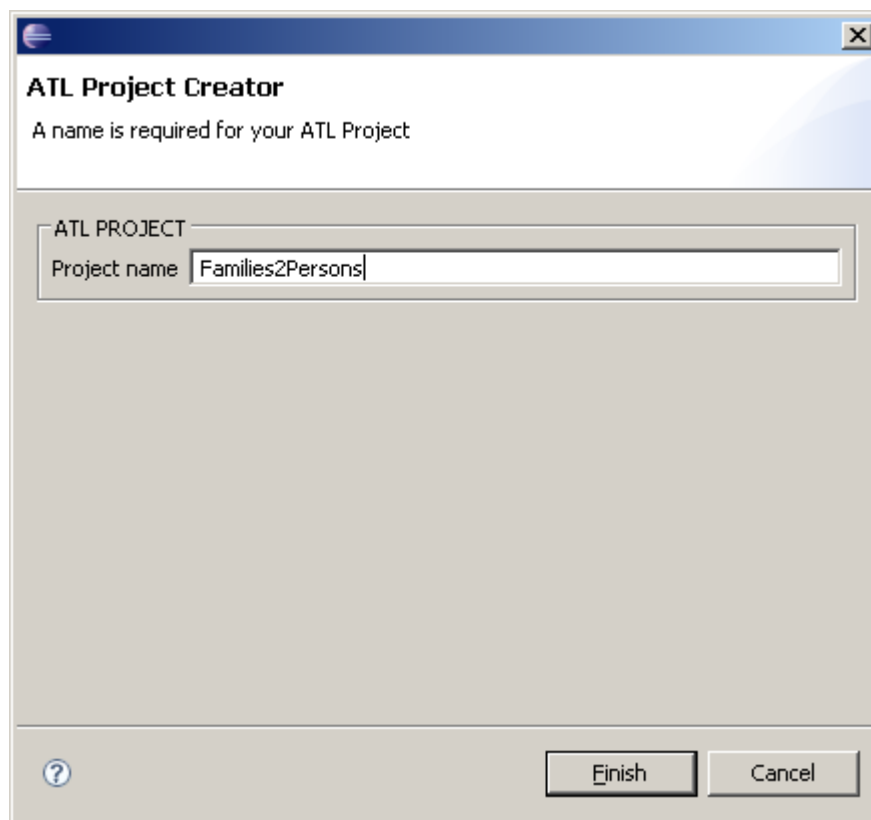


What we need to provide

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

Families to Persons: project creation

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



Definition of the source metamodel "Families"

What is "Families":

A collection of families.

Each family has a lastName and is composed of members:

A father

A mother

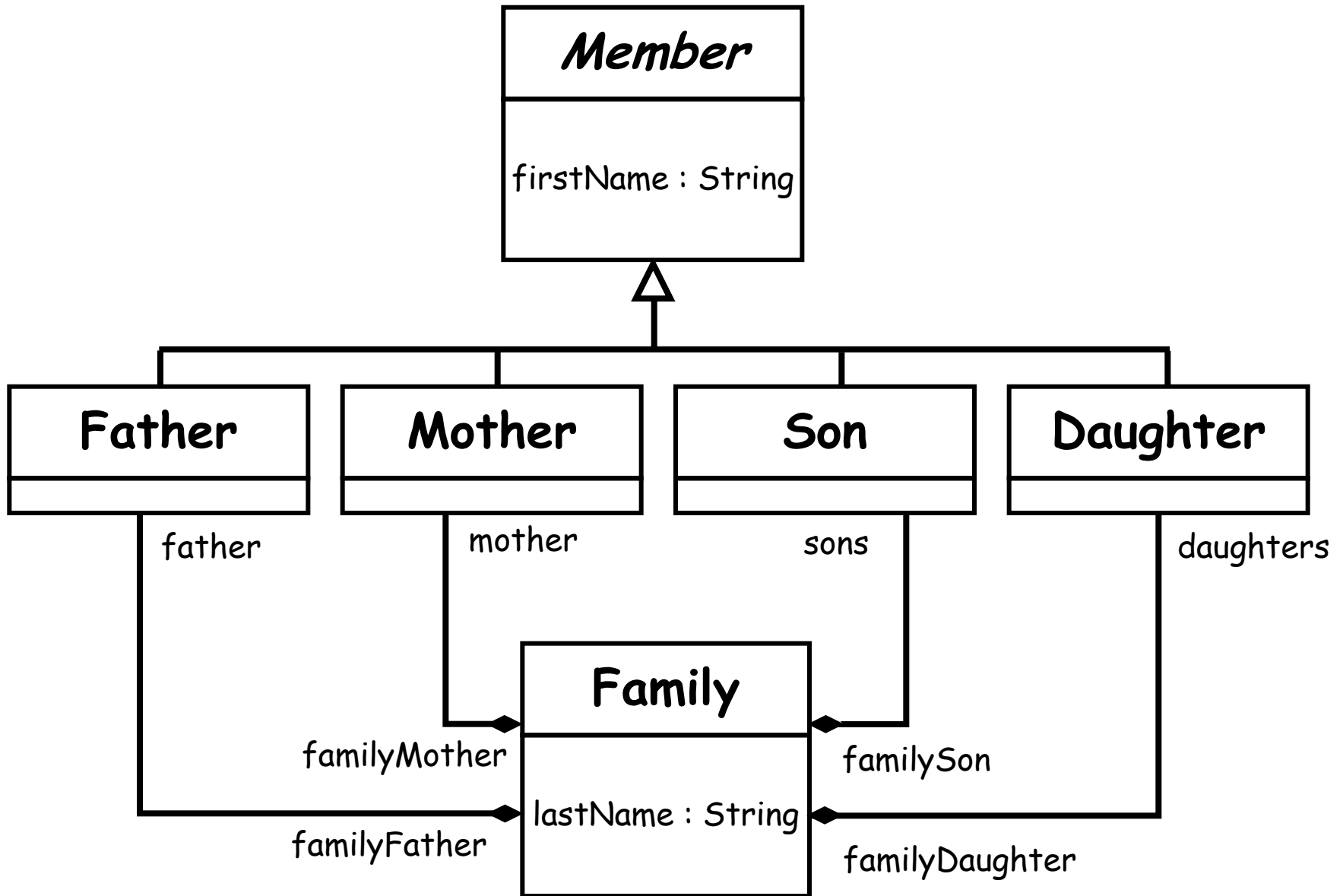
Several sons

Several daughters

Each family member has a first name.

Family: Simpson
Father: Homer
Mother: Marge
Son: Bart
Daughter: Lisa
Daughter: Maggie
Family: Skywalker
Father: Anakin
Mother: Padme
Son: Luke
Daughter: Leia

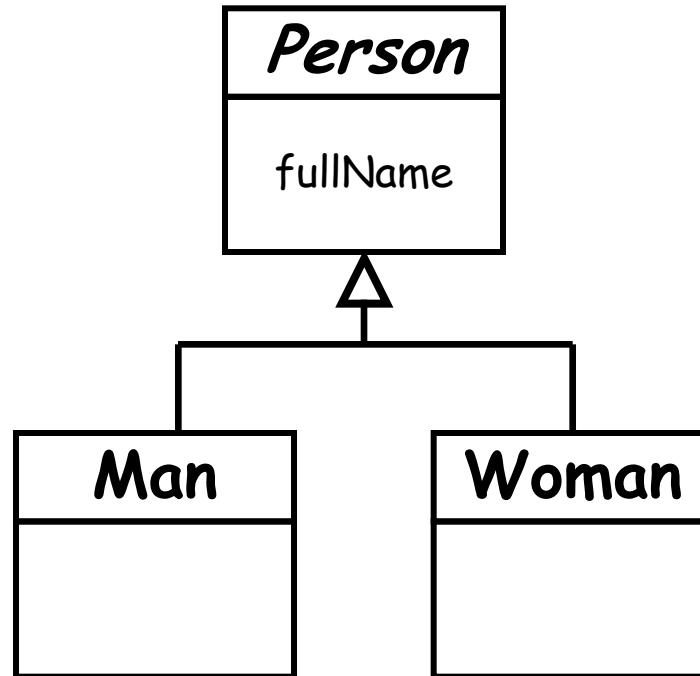
"Families" metamodel (visual presentation)



Demo

Writing a metamodel with KM3

"Persons" metamodel (visual presentation)

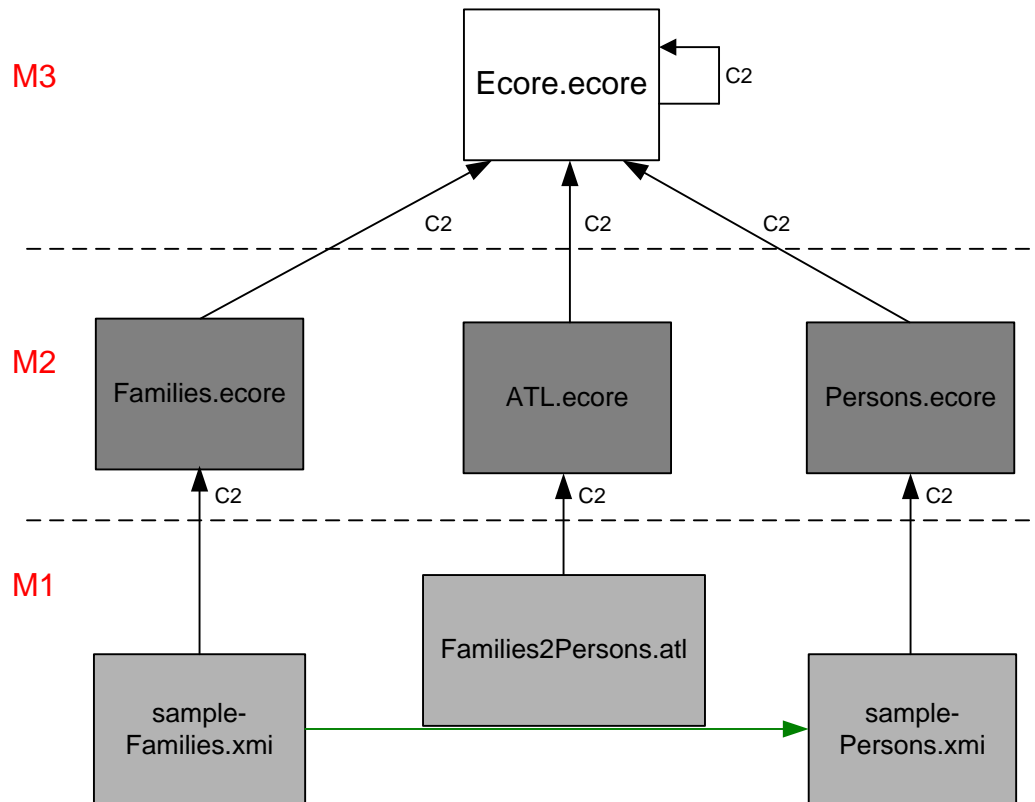


Demo

Writing a metamodel with KM3

The big picture

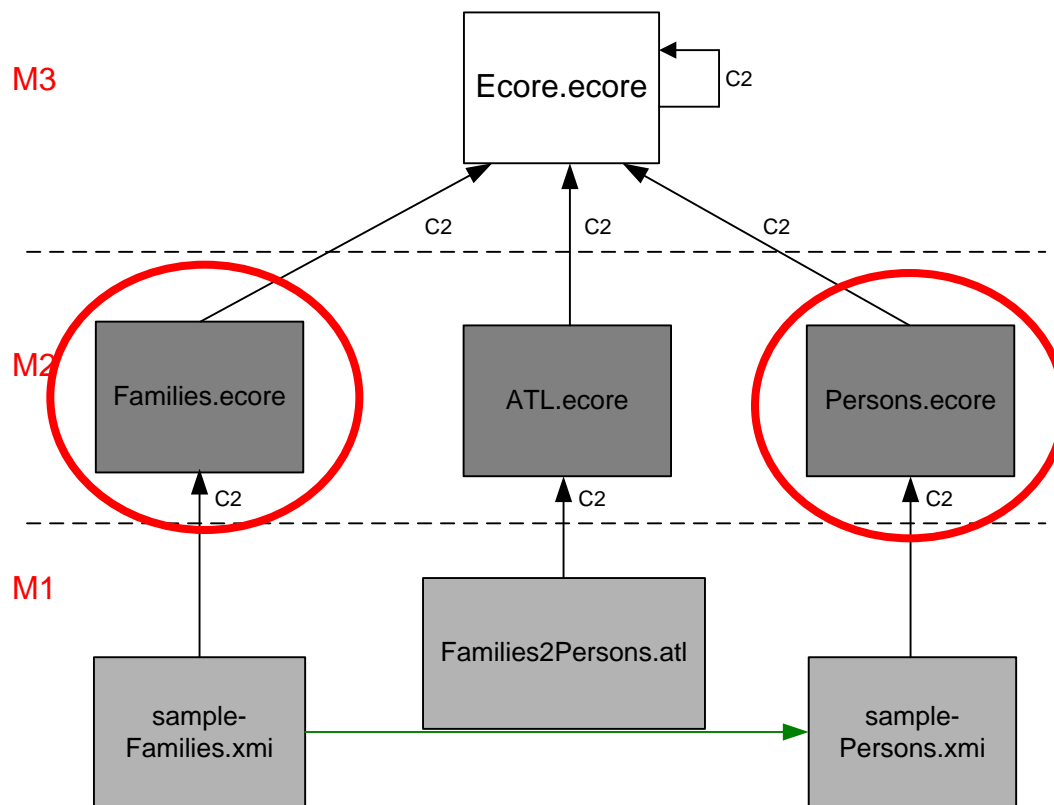
Eclipse Modeling Framework (EMF)



1. Our goal in this use case is to write the ATL transformation, stored in the "Families2Persons" file.
2. Prior to the execution of this transformation the resulting file "sample-Persons.xmi" does not exist. It is created by the transformation.
3. Before defining the transformation itself, we need to define the source and target metamodels ("Families.km3" and "Person.km3") and compile them to Ecore.
4. We take for granted that the definition of the ATL language is available (supposedly in the "ATL.ecore" file).
5. Similarly we take for granted that the environment provides the recursive definition of the metamodel (supposedly in the "Ecore.ecore" file).

Families to Persons Architecture

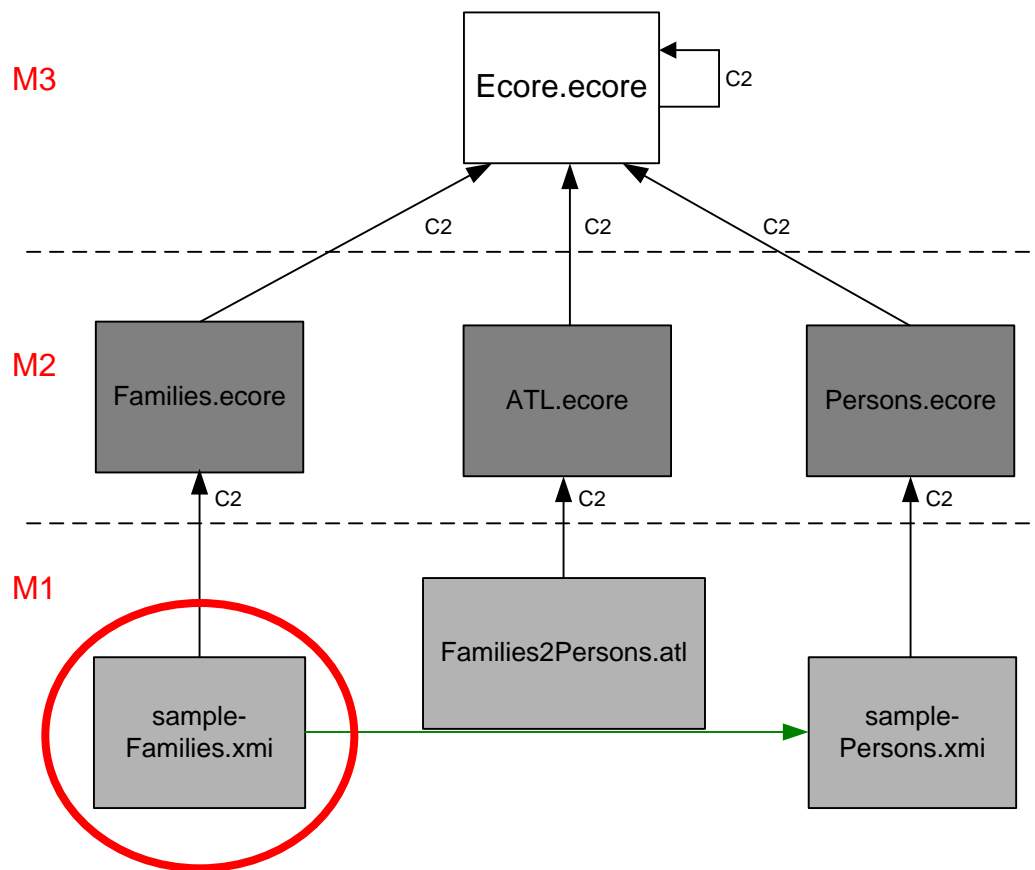
Eclipse Modeling Framework (EMF)



1. *Families* and *Persons* metamodels have been created previously.
2. They have been written in the KM3 metamodel specification DSL (Domain Specific Language).
3. They have been compiled to Ecore

Families to Persons Architecture

Eclipse Modeling Framework (EMF)

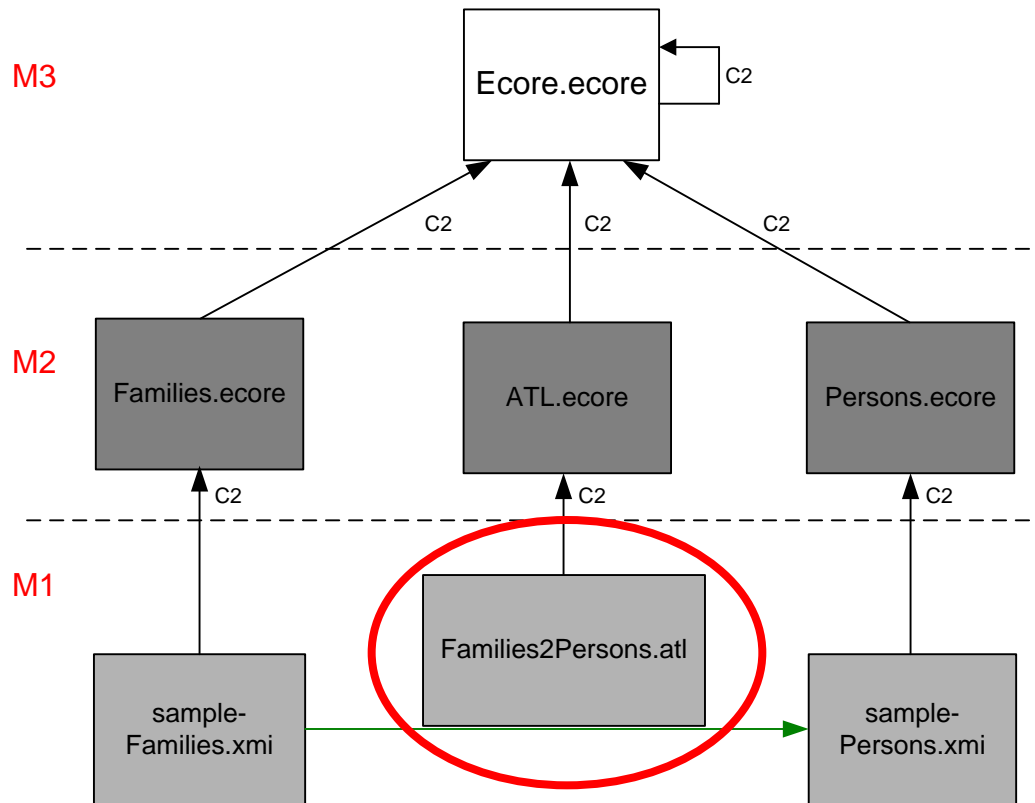


1. The following file is the sample that we will use as source model in this use case:

```
<?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="Skywalker">
    <father firstName="Anakin"/>
    <mother firstName="Padme"/>
    <sons firstName="Luke"/>
    <daughters firstName="Leia"/>
  </Family>
  <Family lastName="Simpsons">
    <father firstName="Homer"/>
    <mother firstName="Marge"/>
    <sons firstName="Bart"/>
    <daughters firstName="Lisa"/>
    <daughters firstName="Maggie"/>
  </Family>
</xmi:XMI>
```

Families to Persons Architecture

Eclipse Modeling Framework (EMF)



1. Now, let us start the creation of the ATL transformation *Families2Persons.atl*.
2. We suppose the ATL environment is already installed.

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

ATL File Wizard

HEAD

Container: \\Families2Persons [Browse...]

ATL Module Name: Families2Persons

ATL File Type: module

IN

Model: IN Metamodel: Families [ADD]

Model: IN Metamodel: Families

OUT

Model: OUT Metamodel: Persons [ADD]

Model: OUT Metamodel: Persons

LIB

LIB [ADD]

LIB

[?] [Finish] [Cancel]

Families:
Name of the source metamodel in the transformation

Persons:
Name of the target metamodel in the transformation

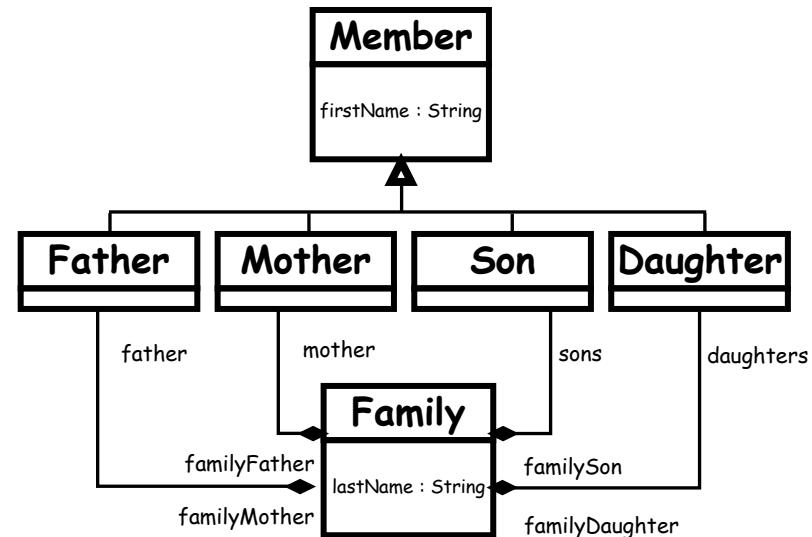
Families to Persons: header section

- 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:

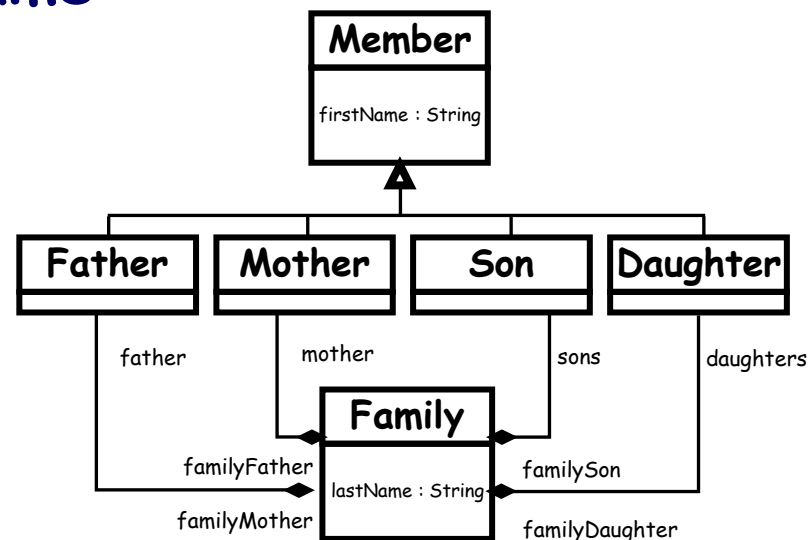


```

helper context Families!Member def: isFemale() : Boolean =
  if self.oclIsTypeOf(Families!Mother) or
    self.oclIsTypeOf(Families!Daughter) then
    true
  else
    false
  endif;
  
```

Families to Persons: helper "getName"

- 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: getName() : String =
  if self.oclIsTypeOf(Families!Father) then
    'Mr. ' + self.firstName + ' ' + self.familyFather.lastName
  else
    if self.oclIsTypeOf(Families!Mother) then
      'Mrs. ' + self.firstName + ' ' + self.familyMother.lastName
    else
      if self.oclIsTypeOf(Families!Son) then
        'Master ' + self.firstName + ' ' + self.familySon.lastName
      else
        'Miss ' + self.firstName + ' ' + self.familyDaughter.lastName
      endif
    endif
  endif;

```

Families to Persons: writing the rules

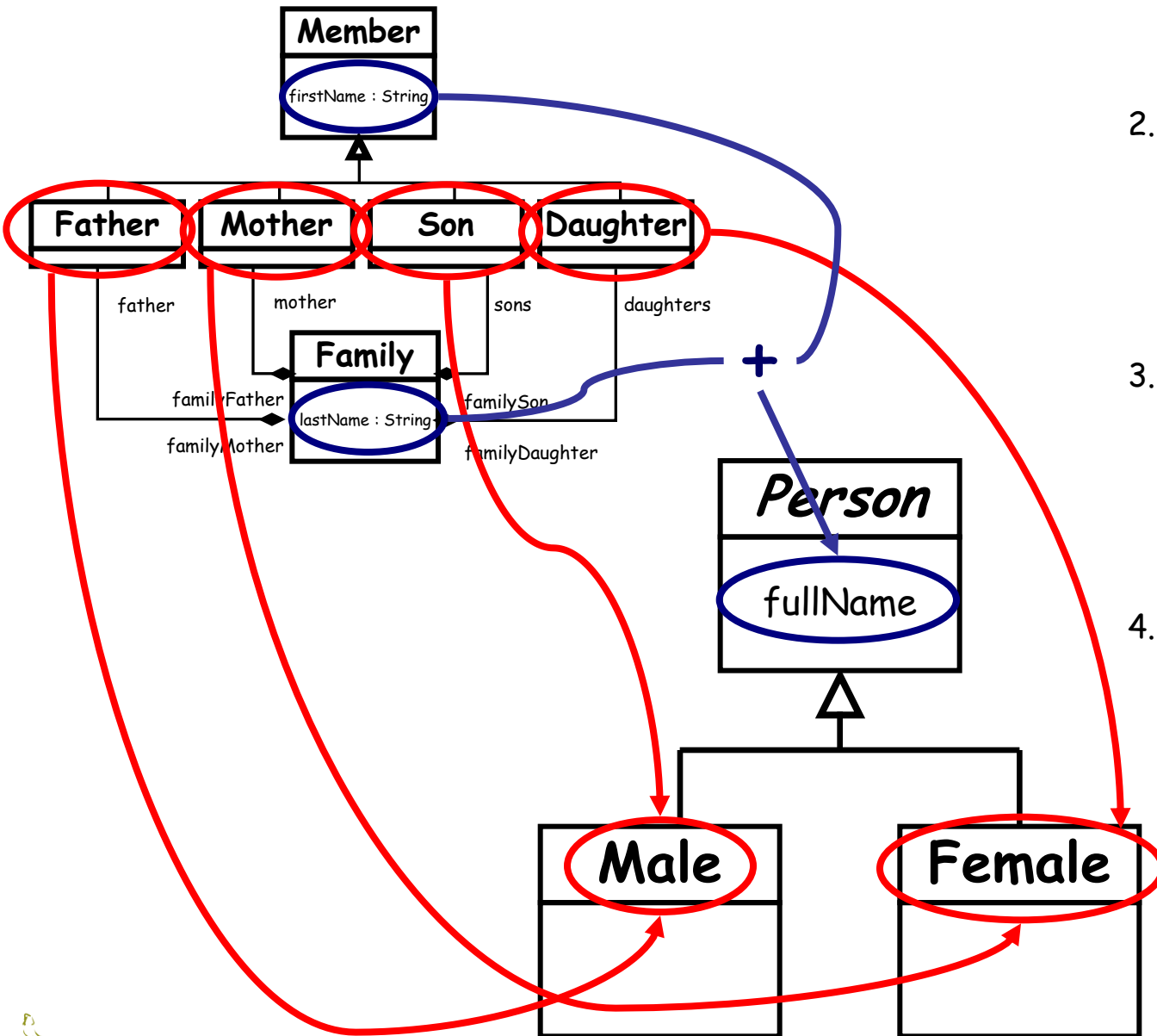
- After the helpers we now write the rules:
 - Member to Male

```
rule Member2Male {  
  from  
    m : Families!Member (not m.isFemale())  
  to  
    p : Persons!Male (  
      fullName <- m.getName()  
    )  
}
```

- Member to Female

```
rule Member2Female {  
  from  
    m : Families!Member (m.isFemale())  
  to  
    p : Persons!Female (  
      fullName <- m.getName()  
    )  
}
```

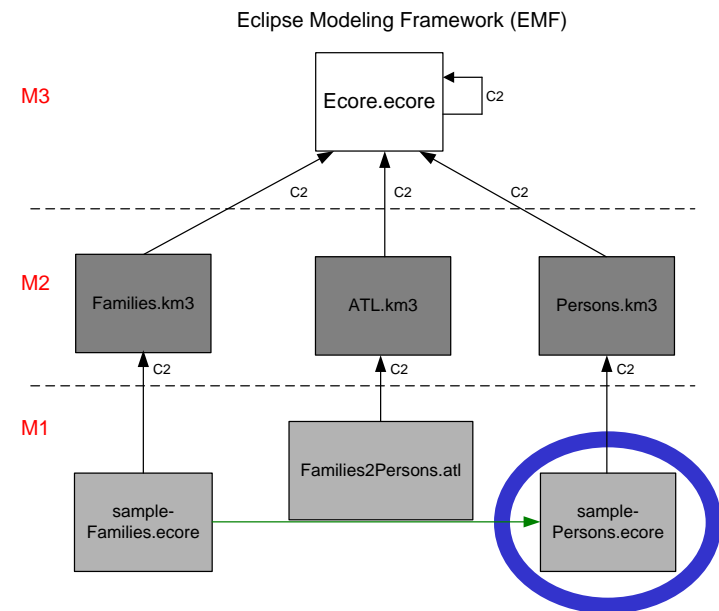
What do we want to do ?



1. For each instance of the class "Member" in the IN model, create an instance in the OUT model.
2. 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.
3. 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.
4. 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.

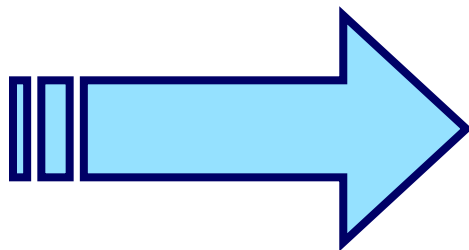
Families to Persons Architecture

1. 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 - 1

```
module Families2Persons;
```

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

The screenshot shows the ATL Launch Configuration dialog in Eclipse. The configuration is named "Families2PersonsComplete". The ATL file is "/Families2PersonsComplete/ATLFile/Families2Persons.atl". The configuration is divided into several sections:

- Metamodels:**
 - Families:** /Families2PersonsComplete/Metamodels/Families.ecore (highlighted with a yellow circle and line)
 - Persons:** /Families2PersonsComplete/Metamodels/Persons.ecore (highlighted with a red circle and line)
- Source Models:**
 - IN:** /Families2PersonsComplete/Models/sample-Families.xmi (highlighted with a blue circle and line)
- Target Models:**
 - OUT:** /Families2PersonsComplete/Models/sample-Persons.xmi (highlighted with a green circle and line)

The left sidebar shows a list of configurations, with "Families2PersonsComplete" selected. The bottom of the dialog has "Apply", "Revert", "Run", and "Close" buttons.

Demo

Writing and launching your
very first ATL transformation

Contents

- Introduction
 - Definitions
 - Operational context
- Description of ATL
- Example: Hello World → Families to Persons
 - Rule Inheritance
- Example: Class to Relational
- Example: Ports
- Conclusion

Rule Inheritance

- Definition:

- A matched rule R2 inherits from another matched rule R1:
 - R1 is the parent rule
 - R2 is the child rule, or subrule
- A parent rule may be abstract (i.e. cannot be applied directly)

- Syntax:

```
abstract rule R1 {  
    -- ...  
}  
rule R2 extends R1 {  
    -- ...  
}
```

- When to use:

- When several rules share a common part
 - code reuse
- To specify polymorphic rules
 - the parent rule specifies source and target element names and types of all its children

Rule Inheritance

- Usage rules:
 - A child rule matches a subset of what its parent rule matches:
 - The source pattern must have the same number of elements,
 - Each child source element must correspond to a unique parent source element,
 - Each child element type must conform to type of corresponding parent element (i.e. be the same or a subtype),
 - Guards are anided.

```
rule Member2Person {  
  from  
    m : Families!Member  
  to  
    p : Persons!Person (fullName <- m.getName())  
}
```

```
rule Member2Male extends Member2Person {  
  from  
    m : Families!Member (not m.isFemale())  
  to  
    p : Persons!Male  
}  
  
rule Member2Female extends Member2Person {  
  from  
    m : Families!Member  
  to  
    p : Persons!Female  
}
```

Rule Inheritance

- Usage rules:
 - A child rule specializes target elements of its parent rule:
 - Target elements can be added in child rule,
 - Child target elements with corresponding parent elements (i.e. with the same variable name) can:
 - Have a different type (but must be a subtype of parent type),
 - Have more bindings,
 - Redefine bindings.

```
rule Member2Person {  
  from  
    m : Families!Member  
  to  
    p : Persons!Person (fullName <- m.getName())  
}
```

```
rule Member2Male extends Member2Person {  
  from  
    m : Families!Member(not m.isFemale())  
  to  
    p : Persons!Male  
}  
  
rule Member2Female extends Member2Person {  
  from  
    m : Families!Member  
  to  
    p : Persons!Female  
}
```

Rule Inheritance

- Usage rules:
 - Only one child rule can match
 - There may be at most one *default* subrule (including parent rule if not abstract): a rule without guard, which is matched by default.

```
rule Member2Person {  
  from  
    m : Families!Member  
  to  
    p : Persons!Person (fullName <- m.getName())  
}
```

```
rule Member2Male extends Member2Person {  
  from  
    m : Families!Member(not m.isFemale())  
  to  
    p : Persons!Male  
}  
  
rule Member2Female extends Member2Person {  
  from  
    m : Families!Member  
  to  
    p : Persons!Female  
}
```

Rule Inheritance

- Informal semantics:
 - Matching:
 1. Root rules (i.e. without parent) are matched,
 2. For each potential match, every subrule with a guard is tested,
 3. The one that matches, if any, is selected,
 4. If none matches, the *default** rule, if any, is selected,
 5. If selected rule is not a leaf (i.e. if it has subrules) then goto 2
 6. Target elements are created by using the most specific types.
 - Applying
 - Most specific bindings are used to initialize target elements (redefined bindings are not executed)

* *default* rule: subrule without guard or parent rule if not abstract

Demo

Using ATL Rule Inheritance

What we have learnt

- Writing metamodels with KM3 and compiling them to Ecore
- Writing a first ATL transformation
- Using rule inheritance
- Launching an ATL transformation

Contents

- Introduction
 - Definitions
 - Operational context
- Description of ATL
- Example: Hello World → Families to Persons
- Example: Class to Relational
 - Overview
 - Source metamodel
 - Target metamodel
 - Rule Class2Table
 - Rule SingleValuedAttribute2Column
 - Rule MultiValuedAttribute2Column
- Example: Ports
- Conclusion

Use Case 2

Class to Relational

Class to Relational: Overview

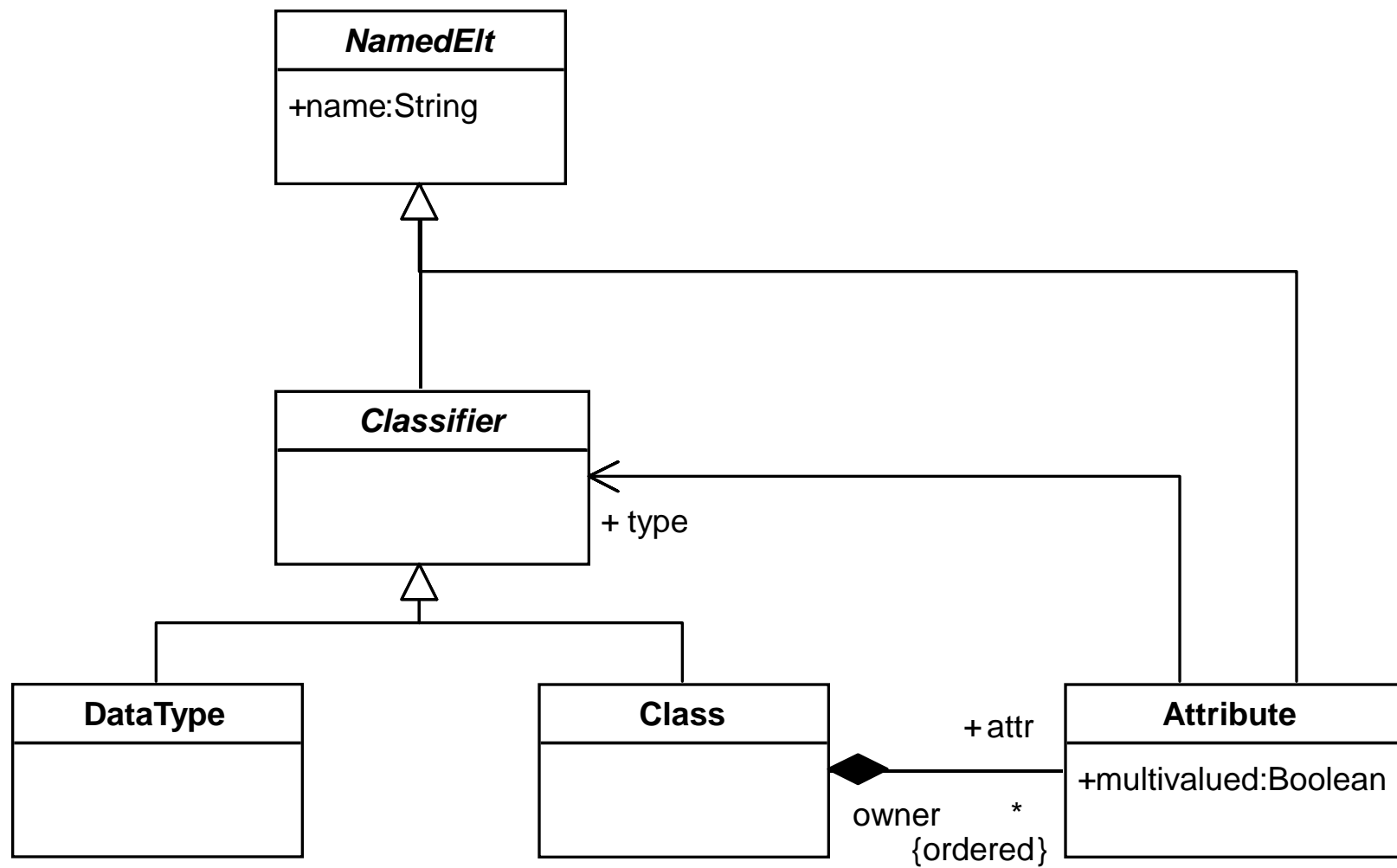
- The source metamodel *Class* is a simplification of class diagrams.
- The target metamodel *Relational* is a simplification of the relational model.

➔ ATL declaration of the transformation:

```
module Class2Relational;
```

```
create OUT : Relational from IN : Class;
```

"Class" metamodel



The Class Metamodel in KM3*

```
package Class {
```

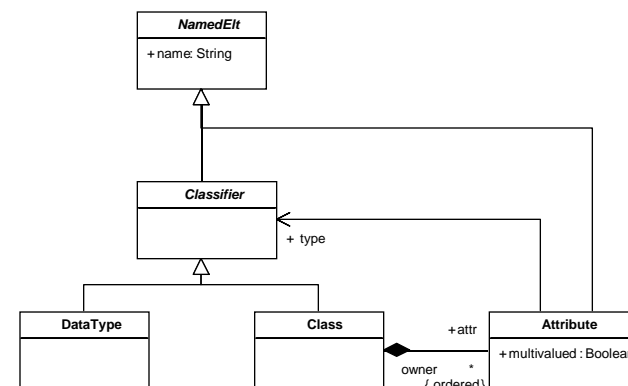
```
  abstract class NamedElt {
    attribute name : String;
  }
```

```
  abstract class Classifier extends NamedElt {}
```

```
  class DataType extends Classifier {}
```

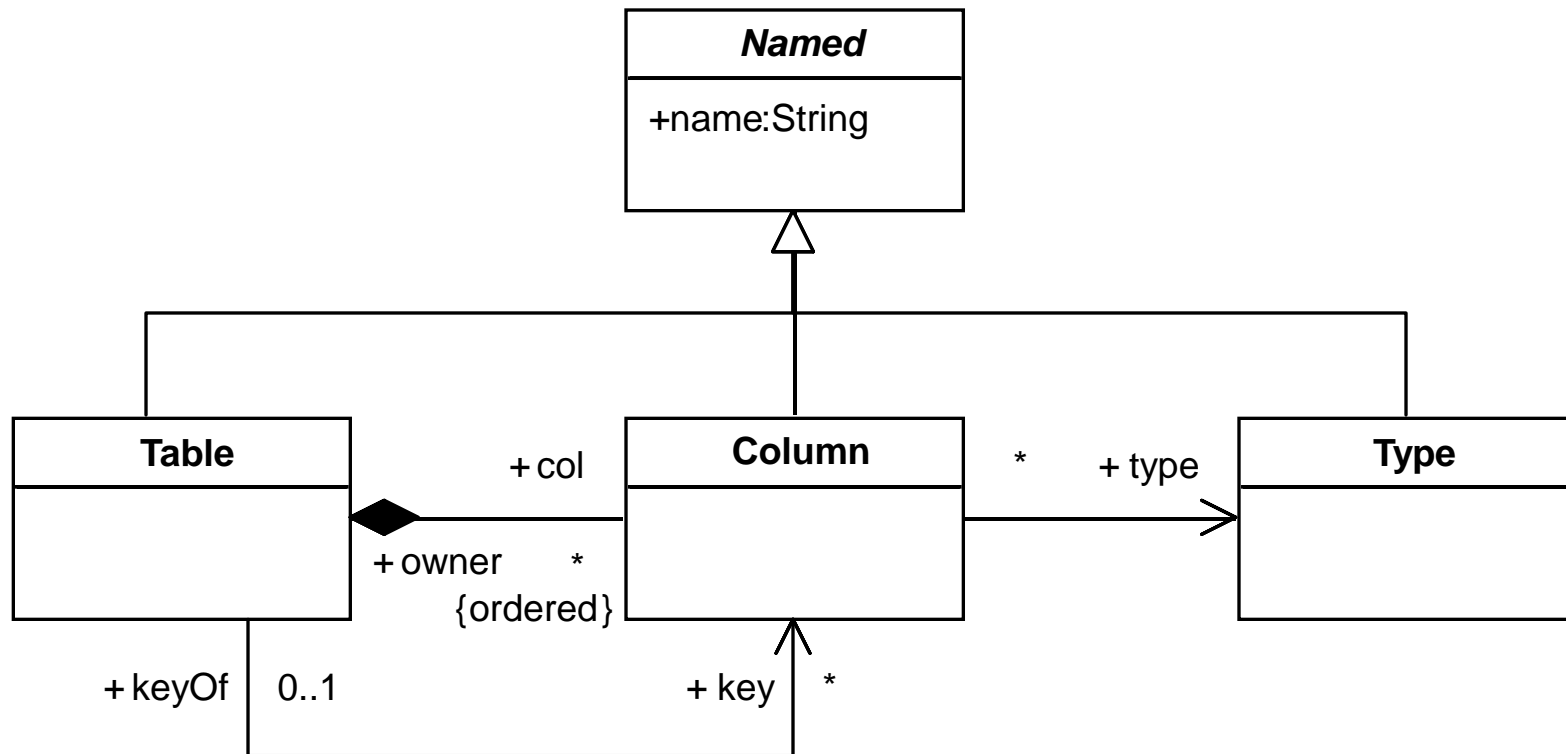
```
  class Class extends Classifier {
    reference attr[*] ordered container : Attribute oppositeOf owner;
  }
```

```
  class Attribute extends NamedElt {
    attribute multiValued : Boolean;
    reference type : Classifier;
    reference owner : Class oppositeOf attr;
  }
}
```



*For more information on KM3 see <http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/>

"Relational" metamodel



The Relational Metamodel in KM3

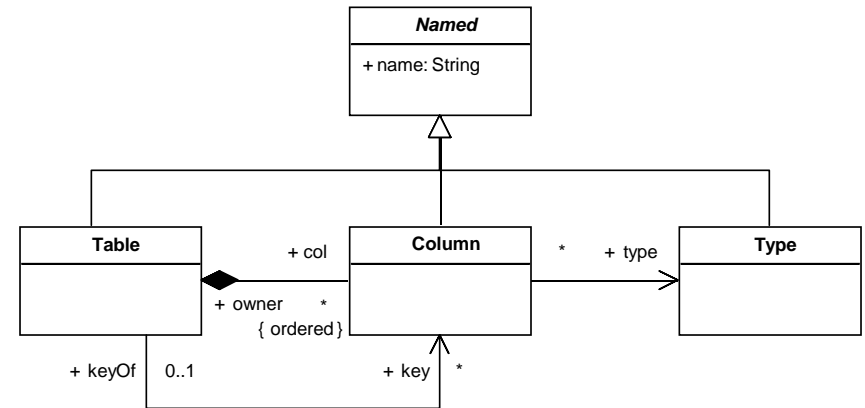
```
package Relational {
```

```
  abstract class Named {
    attribute name : String;
  }
```

```
  class Table extends Named {
    reference col[*] ordered container : Column oppositeOf owner;
    reference key[*] : Column oppositeOf keyOf;
  }
```

```
  class Column extends Named {
    reference owner : Table oppositeOf col;
    reference keyOf[0-1] : Table oppositeOf key;
    reference type : Type;
  }
```

```
  class Type extends Named {}
}
```

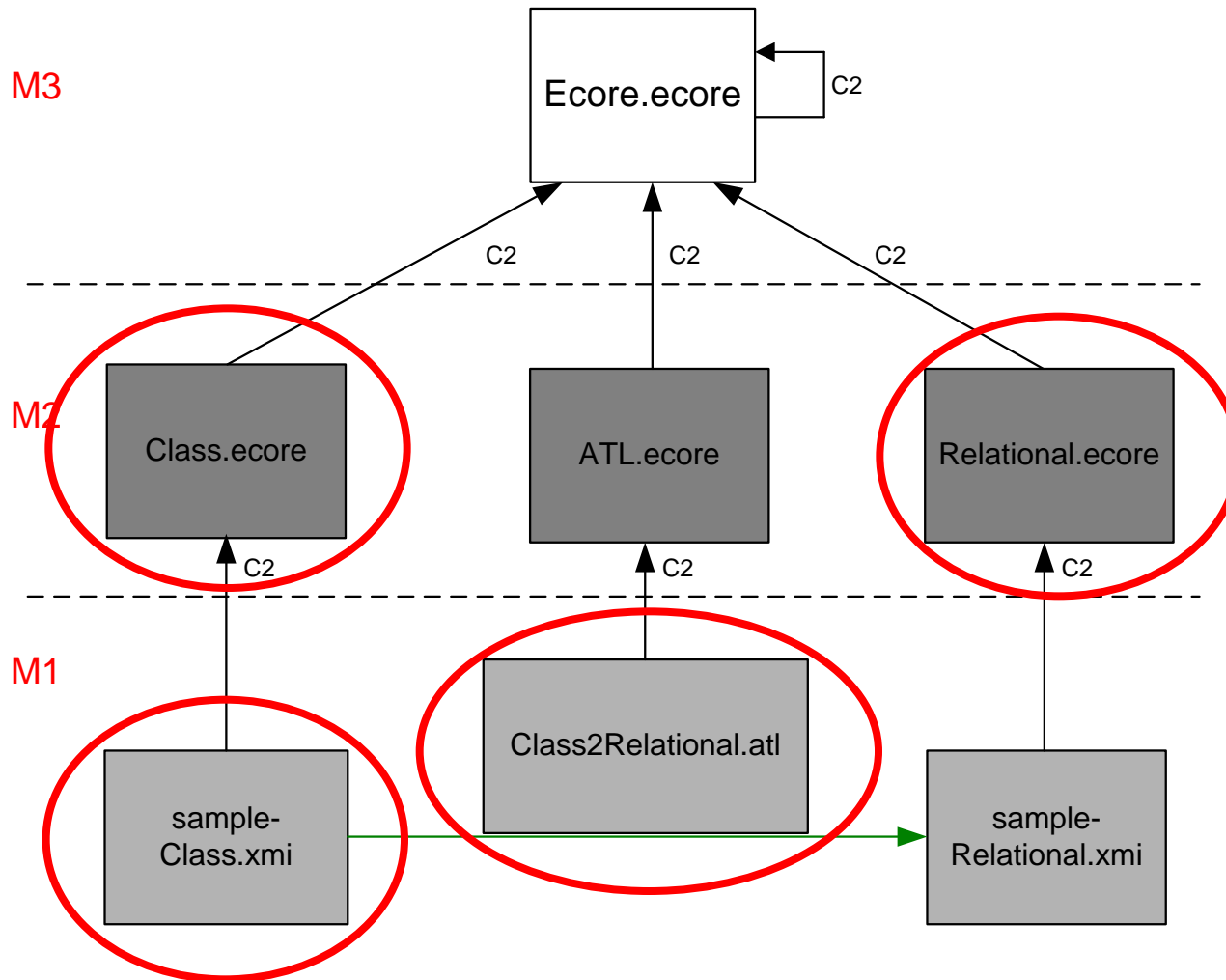


Demo

"Class" and "Relational"
metamodel in KM3

The big picture

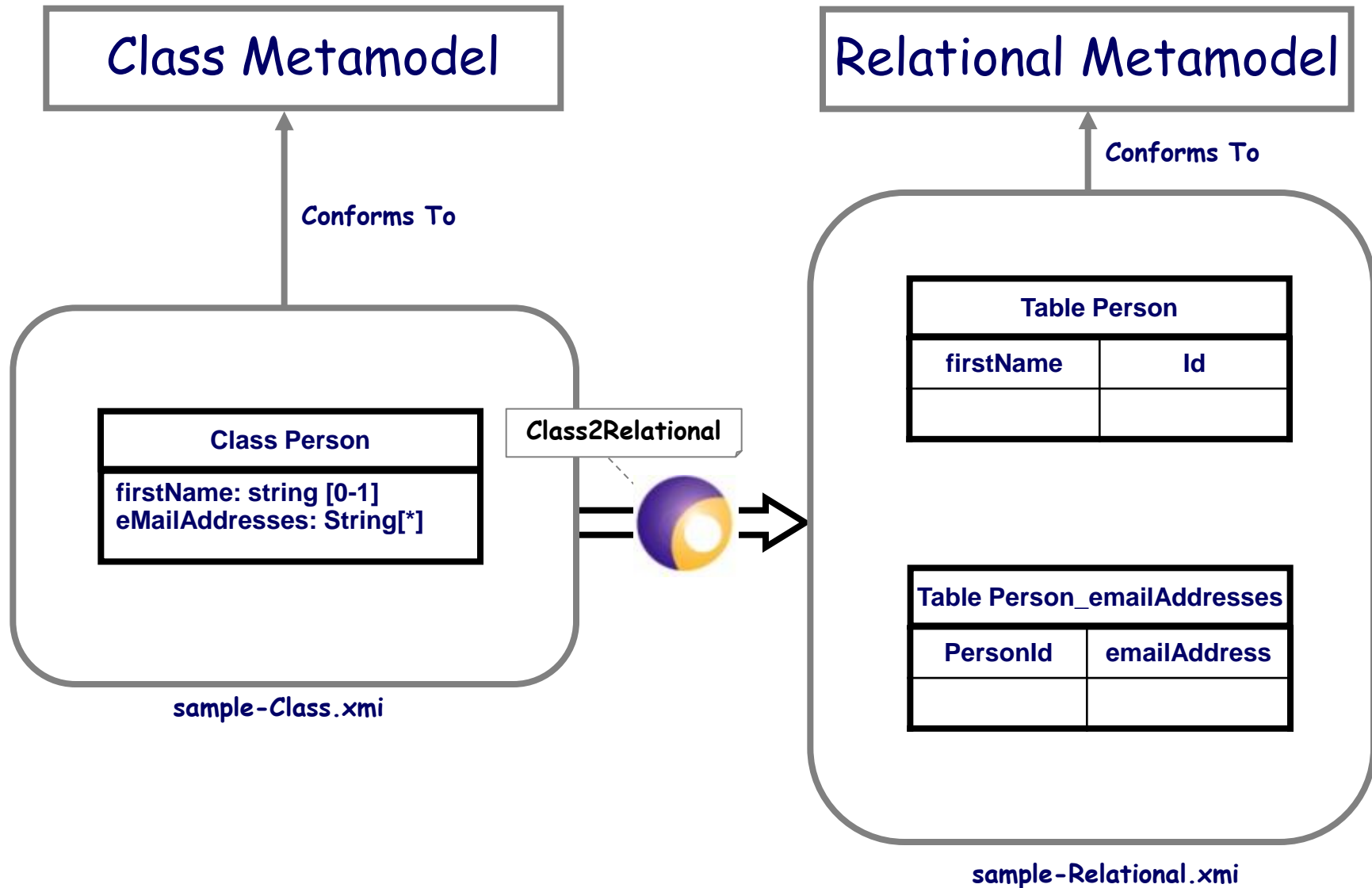
Eclipse Modeling Framework (EMF)



Rules specifications

- Class2Table: a **Table** is created for each **Class**
 - The **name** of the Table is the **name** of the Class
 - The **columns** of the table correspond to **the single-valued attributes of the class**
 - Each Table owns a **key** containing a unique identifier
- SingleValuedAttribute2Column: a **Column** is created for each **single-valued Attribute**
- MultiValuedAttribute2Column: a **Table** is created for each **multi-valued Attribute**, which contains two **columns**:
 - The identifier of the table created from the class owner of the Attribute
 - The value.

Rules specifications - multivalued atts



Example: Class to Relational, rule Class2Table

- A **Table** is created for each **Class**:

```
rule Class2Table {  
  from           -- source pattern  
  c : Class!Class  
  
  to             -- target pattern  
  t : Relational!Table  
}
```

Example: Class to Relational, rule Class2Table

- The **name** of the Table is the **name** of the Class:

```
rule Class2Table {  
  from  
    c : Class!Class  
  to  
    t : Relational!Table (  
      name <- c.name -- a simple binding  
    )  
}
```

Example: Class to Relational, rule Class2Table

- The **columns** of the table correspond to **the single-valued attributes of the class**:

```
rule Class2Table {
```

```
  from
```

```
    c : Class!Class
```

We are referring to the set of columns
that map the set of attributes of this class

```
  to
```

```
    t : Relational!Table (
```

```
      name <- c.name,
```

```
      col <- c.attr->select(e | not e.multiValued)
      -- a binding using complex navigation
```

```
    )
```

```
  }
```

- Remark: attributes are automatically resolved into columns by automatic traceability support.

Example: Class to Relational, rule Class2Table

- Each Table owns a **key** containing a unique identifier:

```
rule Class2Table {  
  from  
    c : Class!Class  
  to  
    t : Relational!Table (  
      name <- c.name,  
      col <- c.attr->select(e |  
        not e.multiValued  
      )->union(Sequence {key}),  
      key <- Set {key}  
    ),  
    key : Relational!Column (name <- 'Id' )  
    -- another target pattern element for the key  
}
```

Example: Class to Relational, rule SingleValuedAttribute2Column

- A Column is created for each **single-valued Attribute**:

```
rule SingleValuedAttribute2Column {  
  from  -- the guard is used for selection  
  a : Class!Attribute (not a.multiValued)  
  to  
  c : Relational!Column (  
    name <- a.name  
  )  
}
```

Example: Class to Relational, rule MultiValuedAttribute2Column

- A **Table** is created for each **multi-valued** Attribute, which contains two columns:
 - The **identifier** of the table created from the class owner of the Attribute
 - The **value**.

```
rule MultiValuedAttribute2Column {  
  from  
    a : Class!Attribute (a.multiValued)  
  to  
    t : Relational!Table (  
      name <- a.owner.name + '_' + a.name,  
      col <- Sequence {id, value}  
    ),  
    id : Relational!Column (  
      name <- 'Id'  
    ),  
    value : Relational!Column (  
      name <- a.name  
    )  
}
```

Demo

"Class2Relational"
ATL transformation

What we have learnt

- Using automatic traceability support in ATL
- Writing simple OCL for navigating models
- Writing helpers

In/Out ports: Overview

- This use case deals with situations where the source element meta type could not be simply matched with the target meta type.
- The only way to resolve the target meta type is to browse source model.
- We will discover how:
 - to use matched rules and lazy (matched) rules,
 - to avoid some imperative constructs
 - to make a first code optimization.

➔ ATL declaration of the transformation:

```
module TypeA2TypeB;
```

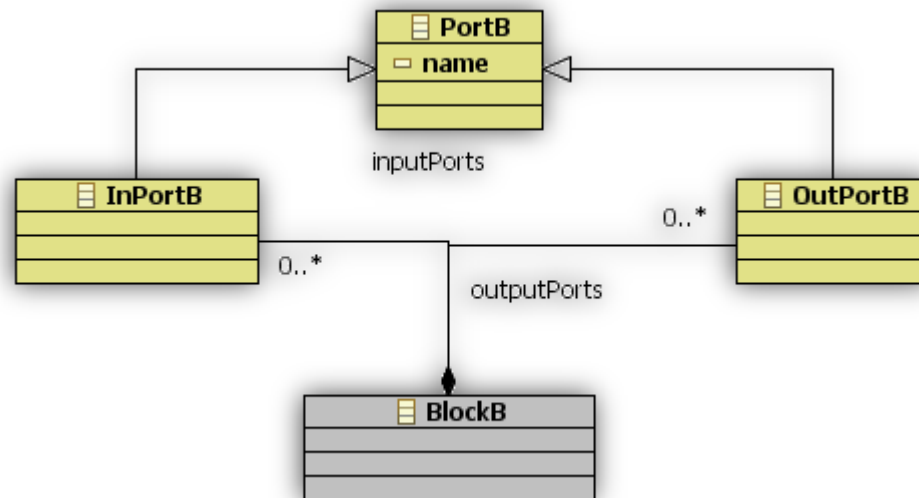
```
create OUT : TypeB from IN : TypeA;
```

In/Out ports: Metamodels



- The metamodel *A* describes a block/port structure where a block can have several input and/or output ports.
- The port type direction is distinguished via its container relation's name: `inputPorts` or `outputPorts`.
- Both input and output ports are represented via the `PortA` concept.

In/Out ports: Metamodels



- The metamodel B describes the same basic concepts but differentiates input and output ports via two new concepts: InPortB and OutPortB.

In/Out ports - Transformation Overview

- These transformation principles are pretty straightforward as both metamodels strictly represent the same block/port concept.
- For a *BlockA*, we create a *BlockB* and:
 - for each input port of *aBlock.inputPorts*, we create an *InPortB*.
 - for each output port of *aBlock.outputPorts*, we create an *OutPortB*.

In/Out ports - First Implementation

- A first idea to transform A to B could be to use the **distinct** clause as shown in the following piece of code

```
module TypeA2TypeB;
create inB : TypeB from inA : TypeA;

rule BlkA2BlkB {
  from
    blkA : TypeA!BlockA
  to
    blkB : TypeB!BlockB {
      inputPorts <- inPts,
      outputPorts <- outPts
    },
    inPts : distinct TypeB!InPortB foreach(ptA in blkA.inputPorts) {
      name <- ptA.name),
    outPts : distinct TypeB!OutPortB foreach(ptA in blkA.outputPorts) {
      name <- ptA.name)
}
```

- Each element is "manually" transformed by iterating on each source port element)

In/Out ports: 2nd - Explicit called lazy rules

- In ATL, developers should avoid bad habits of imperative programming i.e. explicit creation of elements.
 - They should focus on the **What** and not the **How**.
- What kind of rule(s) do we have to write?
 - The common way to replace **distinct-foreach** instruction is to use **lazy rules** instead.
 - So the previous code could be updated as follow:

```
module TypeA2TypeB;
create inB : TypeB from inA : TypeA;

rule BlkA2BlkB {
  from
    blkA : TypeA!BlockA
  to
    blkB : TypeB!BlockB {
      inputPorts <- blkA.inputPorts->collect(e | thisModule.PortA2InPortB(e)),
      outputPorts <- blkA.outputPorts->collect(e | thisModule.PortA2OutPortB(e))
    }
}
```

```
lazy rule PortA2InPortB {
  from
    s : TypeA!PortA
  to
    t : TypeB!InPortB {
      name <- s.name
    }
}
```

```
lazy rule PortA2OutPortB {
  from
    s : TypeA!PortA
  to
    t : TypeB!OutPortB {
      name <- s.name
    }
}
```

In/Out ports: 3rd - Matched Lazy Rules

- Lazy rules should be used when it is not possible to directly match an element from the source model.
- In our case, an input port is created from an input port and an output is created from an output port.
- It clearly appears that a simple solution is possible by using only **automatic traceability links**
 - We can avoid to explicitly call matched rules dealing with Port as done in the previous code with lazy rules).

```
rule PortA2InPortB {
  from
    s : TypeA!PortA (TypeA!BlockA.allInstances()->select(e | e.inputPorts->includes(s))->notEmpty())
  to
    t : TypeB!InPortB (
      name <- s.name
    )
}
```

```
rule PortA2OutPortB {
  from
    s : TypeA!PortA (TypeA!BlockA.allInstances()->select(e
  to
    t : TypeB!OutPortB (
      name <- s.name
    )
}
```

```
rule BlkA2BlkB {
  from
    blkA : TypeA!BlockA
  to
    blkB : TypeB!BlockB (
      inputPorts <- blkA.inputPorts,
      outputPorts <- blkA.outputPorts
    )
}
```

In/Out ports: 4th - Matched Lazy Rules Optimized

- **PortA2OutPortB** has a guard filtering only output ports and **PortA2InPortB** has a guard filtering only input ports from the input model.
- When this transformation will be applied on big models, evaluation of the guard could be time-consuming.
- Some optimizations in each guard could be done by using **refImmediateComposite()** operation.
 - **refImmediateComposite()** is a reflective operation that returns the immediate composite (e.g. the immediate container).

```
rule PortA2InPortB {  
  from  
    s : TypeA!PortA (s.refImmediateComposite().inputPorts->includes(s))  
  to  
    t : TypeB!InPortB (  
      name <- s.name  
    )  
}
```

```
rule BlkA2BlkB {  
  from  
    blkA : TypeA!BlockA  
  to  
    blkB : TypeB!BlockB (  
      inputPorts <- blkA.inputPorts,  
      outputPorts <- blkA.outputPorts  
    )  
}
```

```
rule PortA2OutPortB {  
  from  
    s : TypeA!PortA (s.refImmediateComposite().outputPorts->includes(s))  
  to  
    t : TypeB!OutPortB (  
      name <- s.name  
    )  
}
```

What we have learnt

- Using matched rules.
- Using lazy (matched) rules.
- Avoiding some imperative constructs.
- Making a first code optimization.

Contents

- Introduction
 - Definitions
 - Operational context
- Description of ATL
- Example: Hello World → Families to Persons
 - Rule Inheritance
- Example: Class to Relational
- Example: Ports
- Conclusion

Conclusion

- ATL has a simple declarative syntax:
 - Simple problems are generally solved simply.
- ATL supports advanced features:
 - Complex OCL navigation, lazy rules, refining mode, rule inheritance, etc.
 - Many complex problems can be handled declaratively.
- ATL has an imperative part:
 - Any problem can be handled.

Summary

- We have presented here a "hello world" level basic ATL transformation as well as a more elaborated ones.
- 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>

End of part 1: Model Transformation with ATL

- Thanks
 - Questions?
 - Comments?