Transforming models with ATL

The ATLAS Transformation Language

Frédéric Jouault

ATLAS group (INRIA & LINA), University of Nantes, France
http://www.sciences.univ-nantes.fr/lina/atl/

Context of this work

- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

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

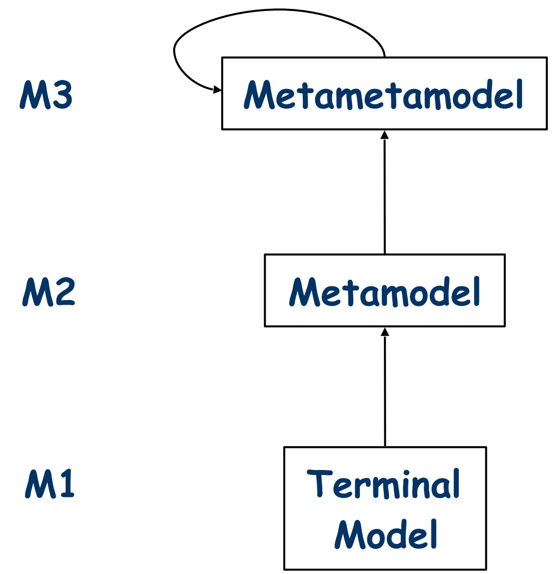
Contents

- Introduction
- Description of ATL
- Example: Class to Relational
- Additional considerations
- Conclusion

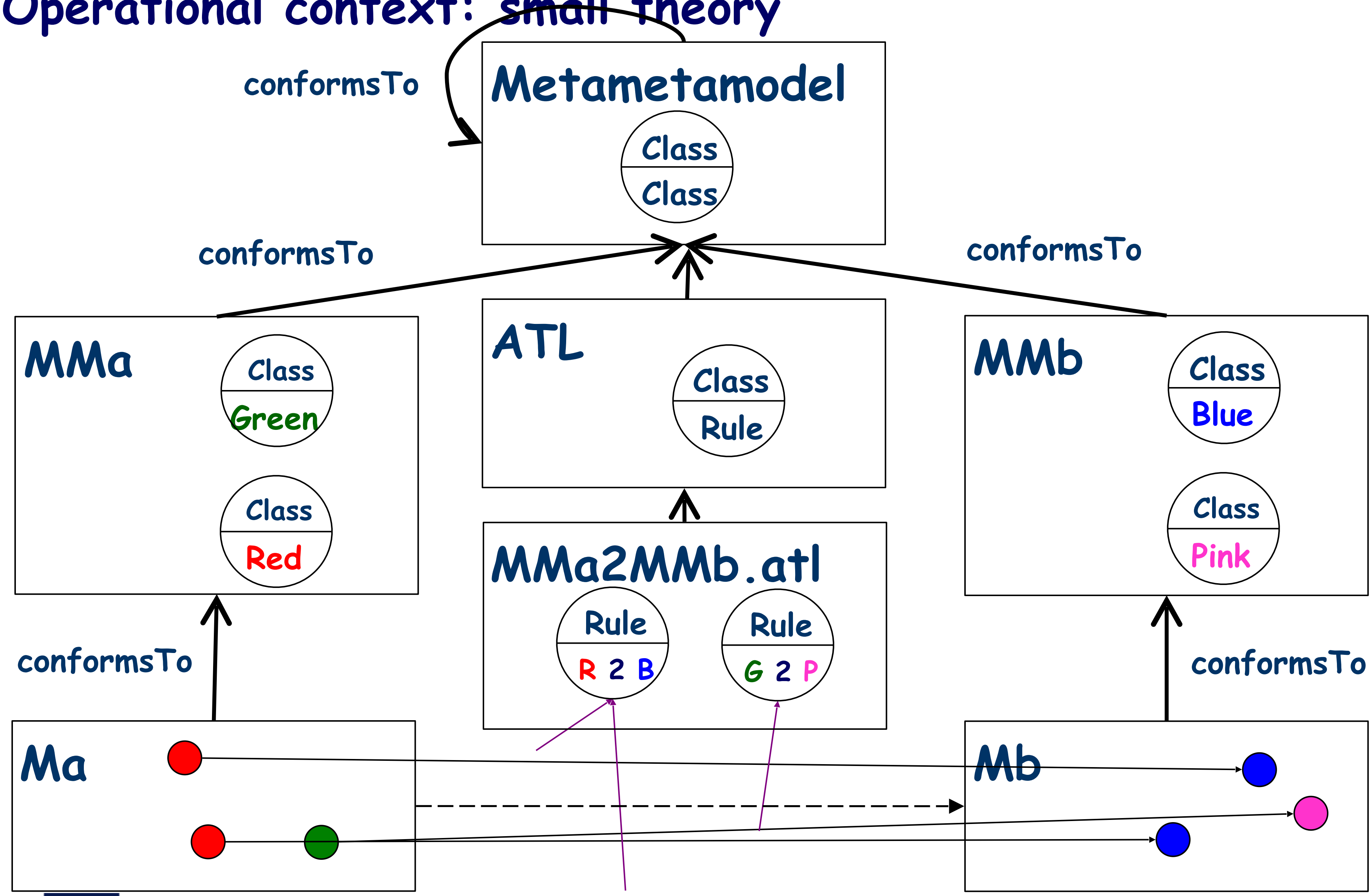
Contents

Definitions

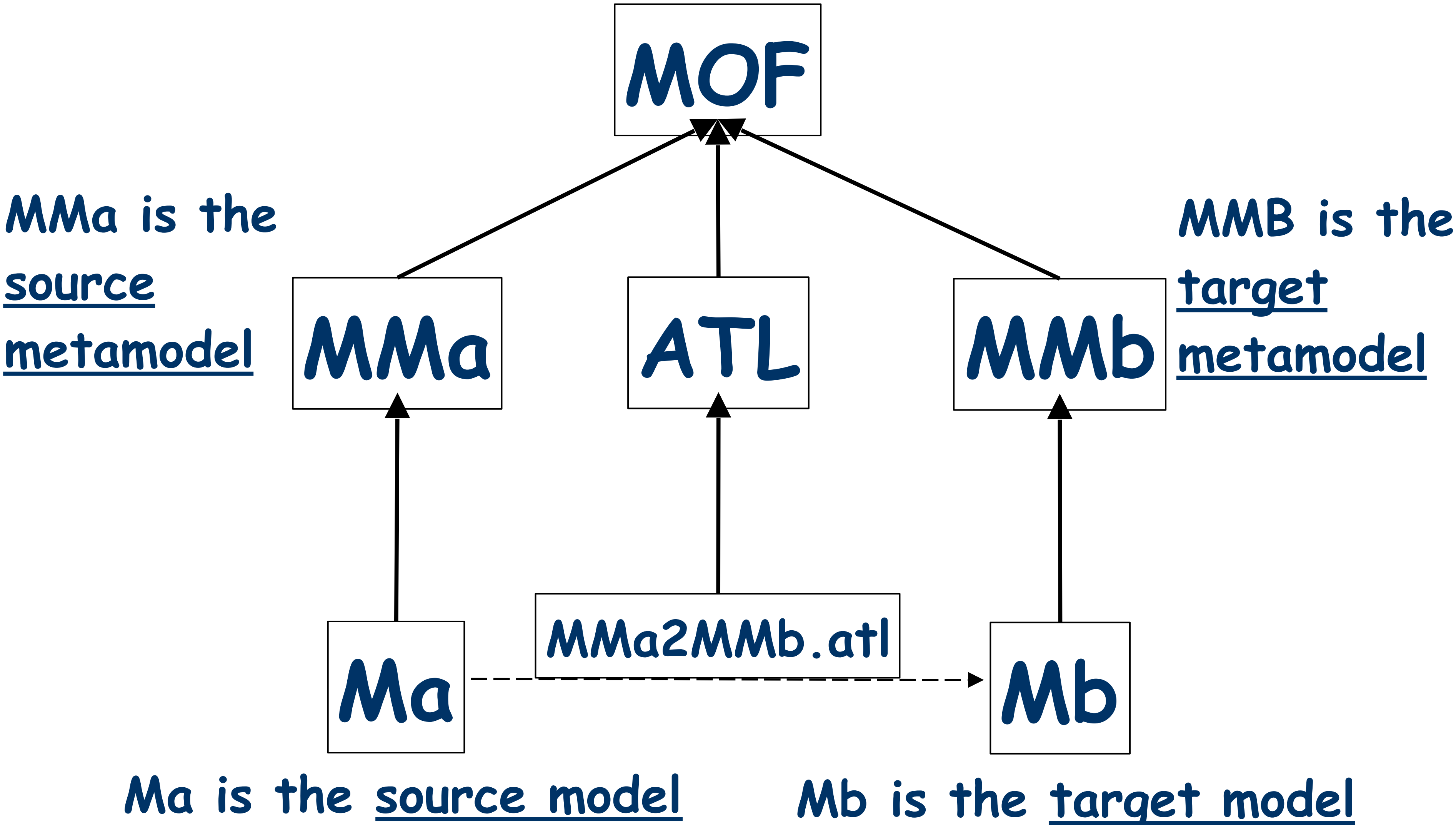
- A model transformation is the automatic creation of target models from source models.
- Model transformation is not only about M1 to M1 transformations:
  - M1 to M2: promotion,
  - M2 to M1: demotion,
  - M3 to M1, M3 to M2, etc.



Operational context: small theory



Operational context of ATL



Contents

ATL overview

- 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).
- 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.
- Recommended programming style: declarative

ATL overview (continued)

- 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.
- 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.

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.

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.

Contents

- Introduction
- Description of ATL
- Example: Class to Relational
  - Overview
  - Source metamodel
  - Target metamodel
  - Rule Class2Table
  - Rule SingleValuedAttribute2Column
  - Rule MultiValuedAttribute2Column
- Additional considerations
- Conclusion

Example: 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.

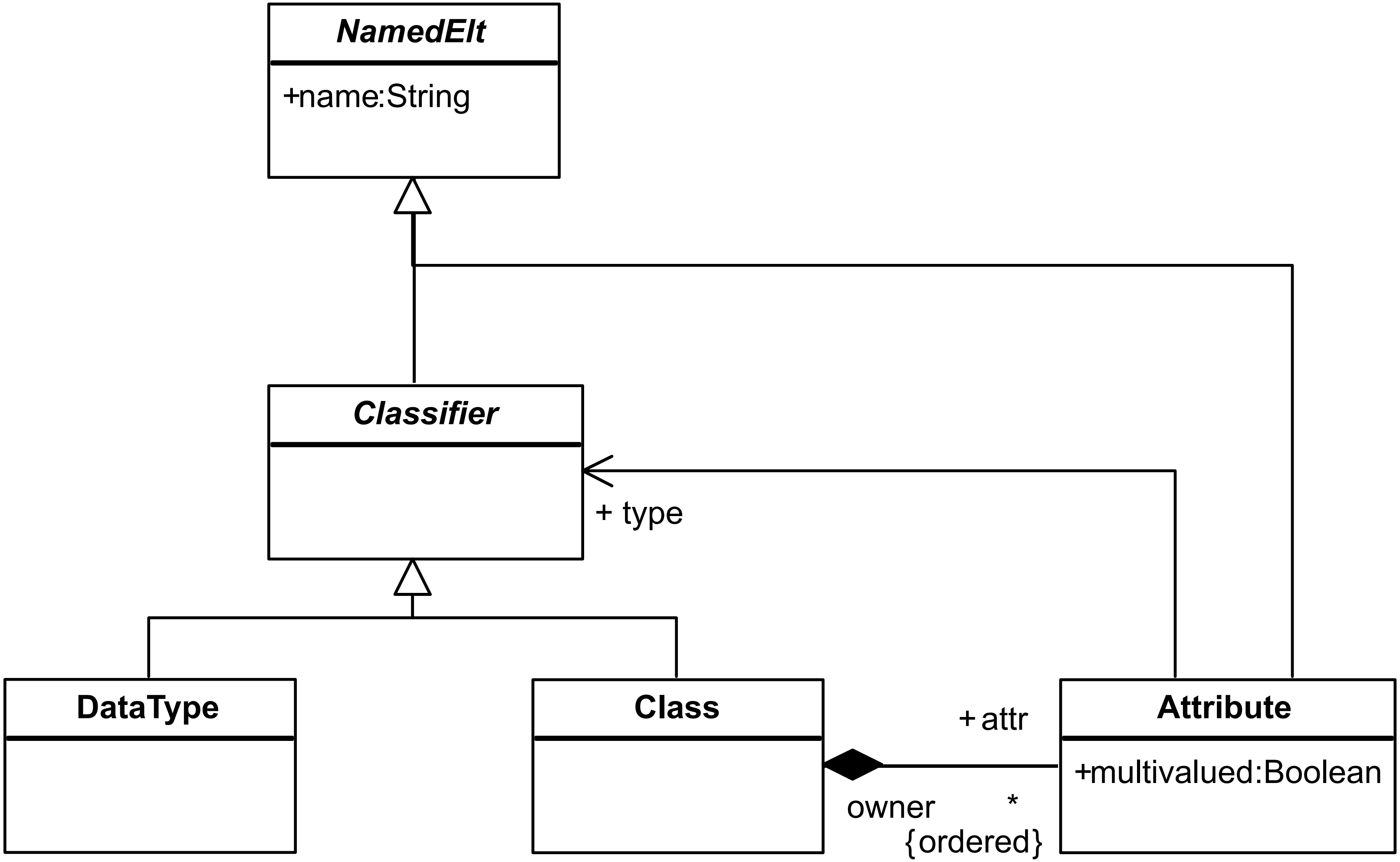
➔ ATL declaration of the transformation:

```
module Class2Relational;
create Mout : Relational from Min : Class;
```

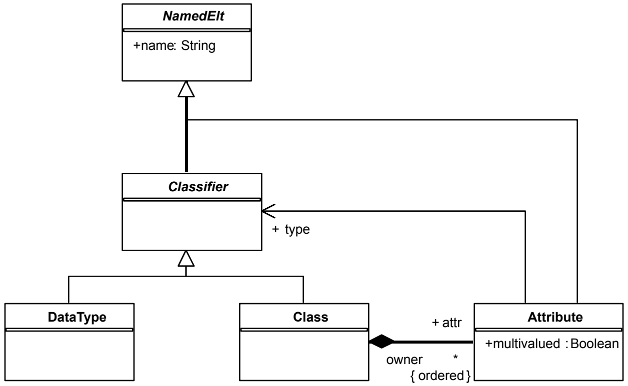
- The transformation excerpts used in this presentation come from:

http://www.eclipse.org/gmt/atl/atlTransformations/#Class2Relational

Source: the Class metamodel



The Class Metamodel in KM3*

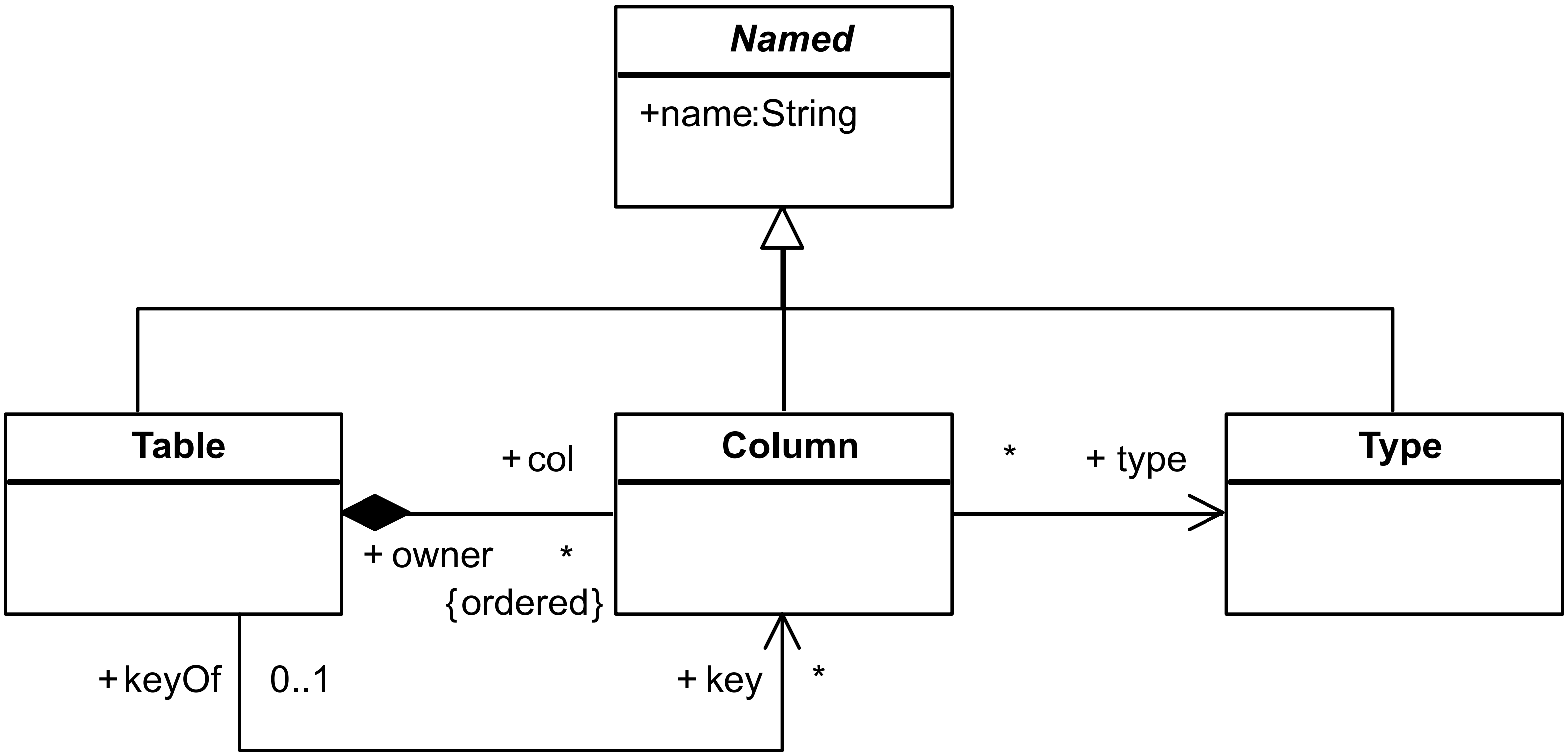



```
package Class {

        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;
        }
}
```

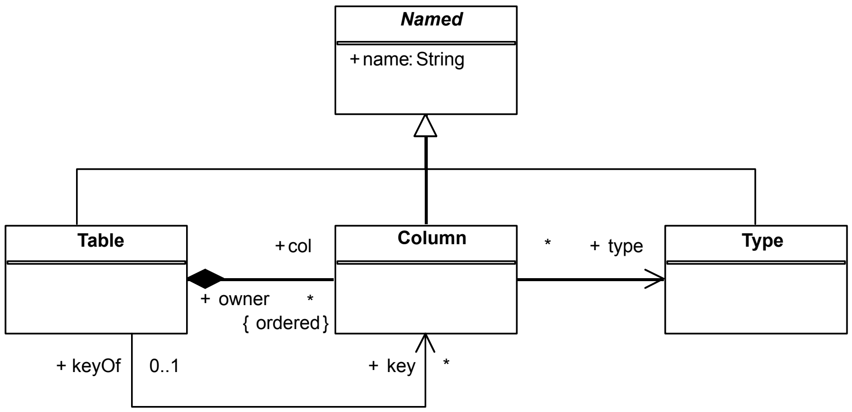
UNIVERSITÉ DE NANTES

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

The 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 {}
```

Example: Class to Relational, overview

- Informal description of rules
  - Class2Table:
    - A table is created from each class,
    - The columns of the table correspond to the single-valued attributes of the class,
    - A column corresponding to the key of the table is created.
  - SingleValuedAttribute2Column:
    - A column is created from each single-valued attribute.
  - MultiValuedAttribute2Column:
    - A table with two columns is created from each multi-valued attribute,
    - One column refers to the key of the table created from the owner class of the attribute,
    - The second column contains the value of the attribute.

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
	to
		t : Relational!Table (
			name <- c.name,
			col <- c.attr->select(e |		-- a binding
					not e.multiValued		-- 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 (  -- another target
            name <- 'Id'               -- 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
		)
}
```

Contents

Other ATL features: rule inheritance

- Rule inheritance, to help structure transformations and reuse rules and patterns:
  - A child rule matches a subset of what its parent rule matches,
    - → All the bindings of the parent still make sense for the child,
  - A child rule specializes target elements of its parent rule:
    - Initialization of existing elements may be improved or changed,
    - New elements may be created,
  - Syntax:

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

Other ATL features: refining mode

- Refining mode for transformations that need to modify only a small part of a model:
  - Since source models are read-only target models must be created from scratch,
  - This can be done by writing copy rules for each elements that are not transformed,
    - ➔ This is not very elegant,
  - In refining mode, the ATL engine automatically copies unmatched elements.
- The developer only specifies what changes.
- ATL semantics is respected: source models are still read-only.
  - ➔ An (optimized) engine may modify source models in-place but only commit the changes in the end.
- Syntax: replace `from` by `refining`

```
module A2A; create OUT : MMA refining IN : MMA;
```

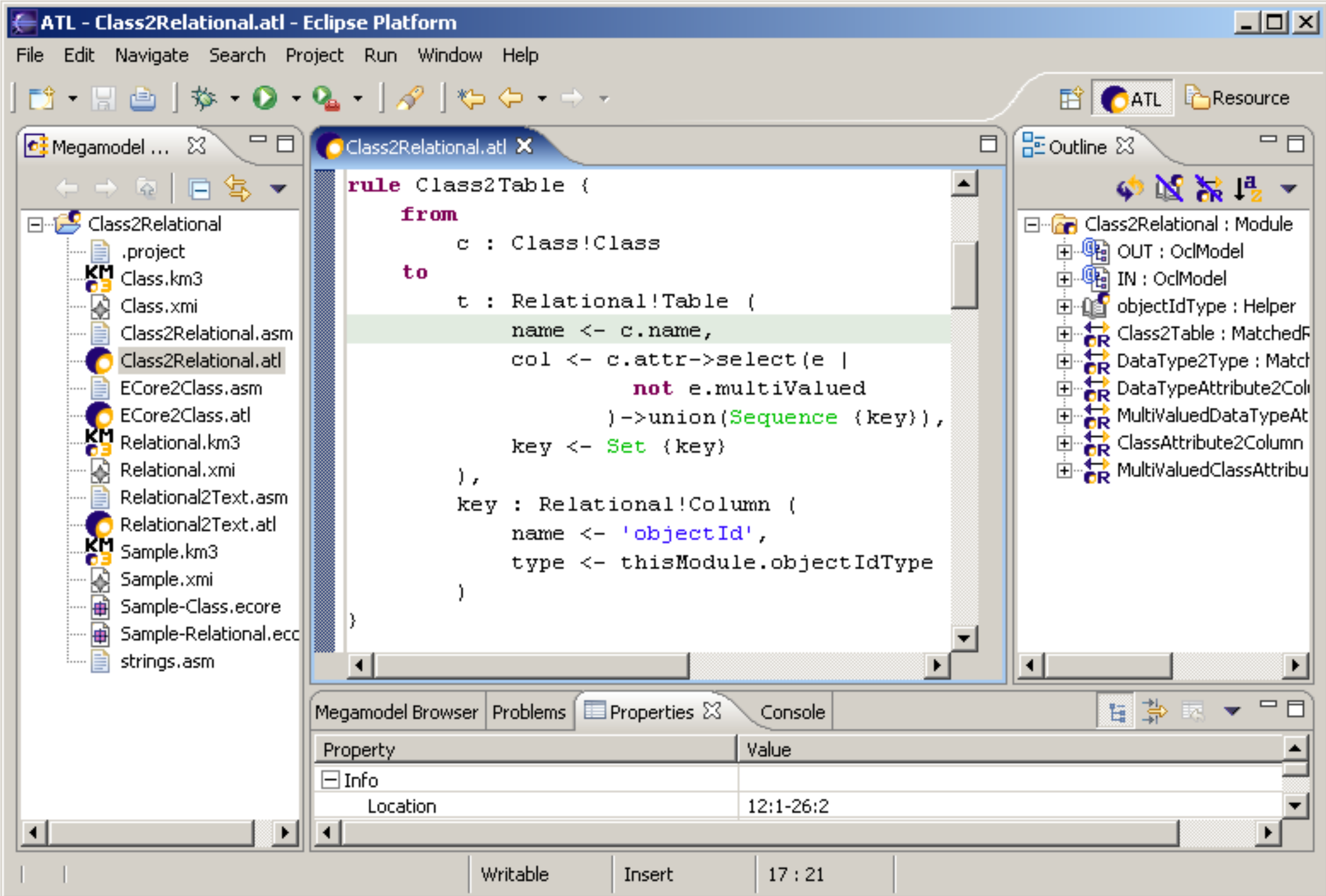
ATL in use

- ATL has been used in a large number of application domains.
- A library of transformations is available at http://www.eclipse.org/gmt/atl/atlTransformations/
  - More than 40 scenarios,
  - More than 100 single transformations.
- About 100 sites use ATL for various purpose:
  - Teaching,
  - Research,
  - Industrial development,
  - Etc.

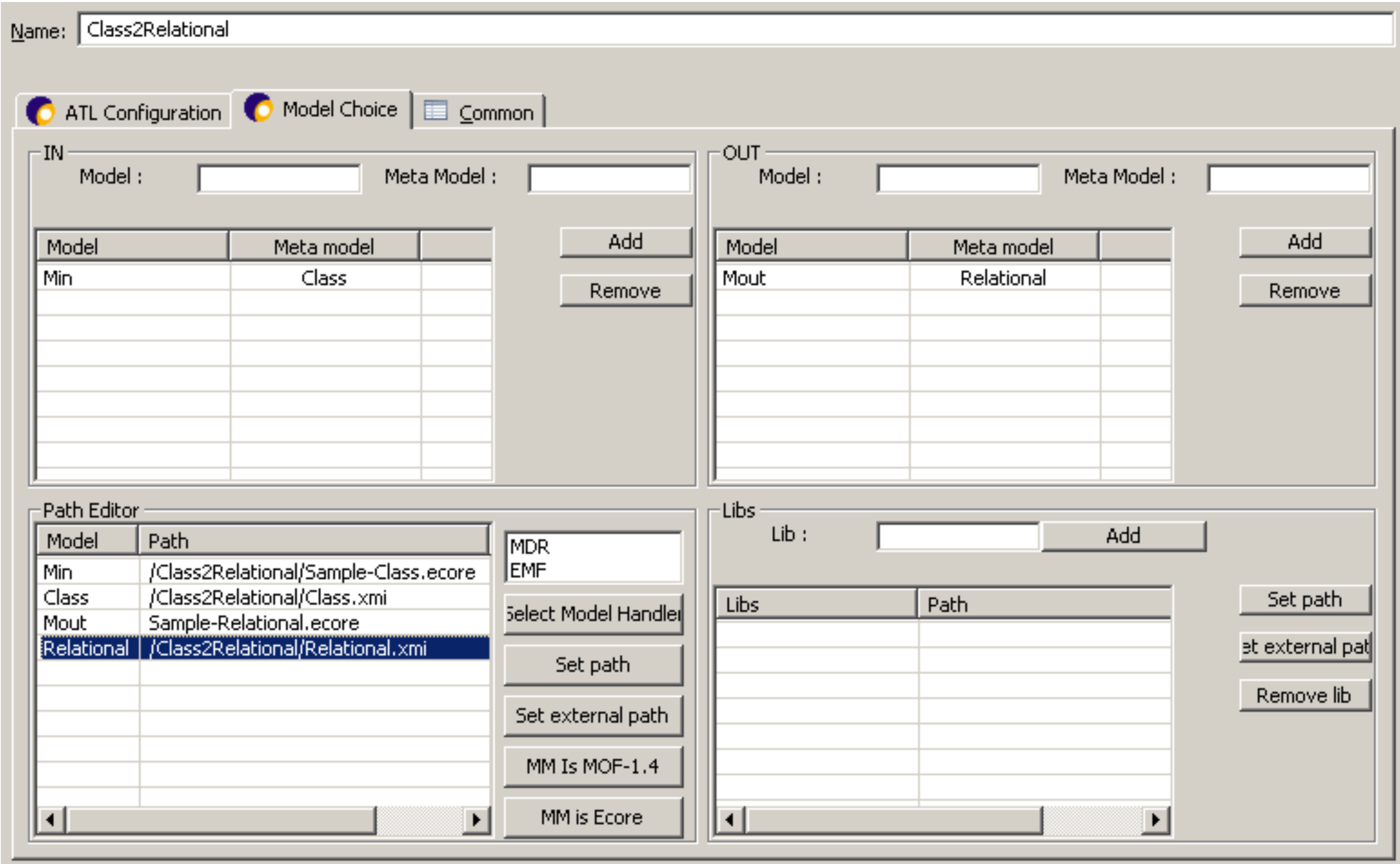
ATL in use

- ATL tools and documentation are available at http://www.eclipse.org/gmt/atl/
  - Execution engine:
    - Virtual machine,
    - ATL to bytecode compiler,
  - Integrated Development Environment (IDE) for:
    - Editor with syntax highlighting and outline,
    - Execution support with launch configurations,
    - Source-level debugger.
  - Documentation:
    - Starter's guide,
    - User manual,
    - Installation guide,
    - Etc.

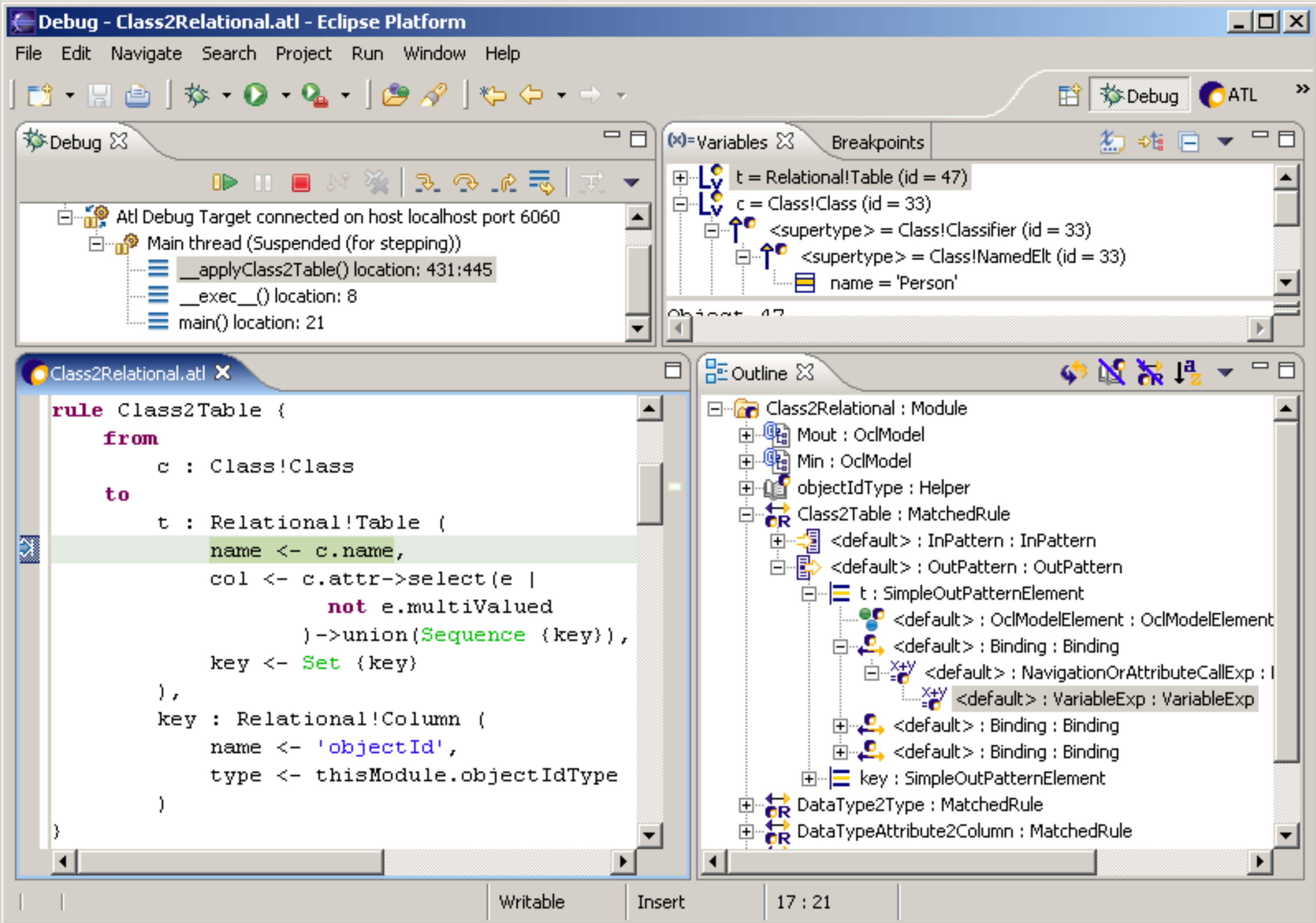
ATL Development Tools: perspective, editor and outline



ATL Development Tools: launch configuration



ATL Development Tools: source-level debugger



Contents

- Introduction
- Description of ATL
- Example: Class to Relational
- Additional considerations
- 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.

End of the presentation

- Thanks
  - Questions?
  - Comments?

AMMA@lina.univ-nantes.fr
ATLAS group, INRIA & LINA, Nantes