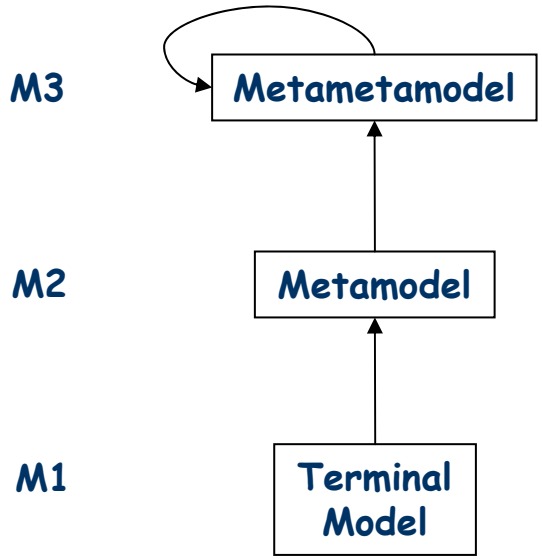
Transforming Models with ATL

Frédéric Jouault
AtlanMod (INRIA & EMN)
frederic.jouault@{inria,emn}.fr

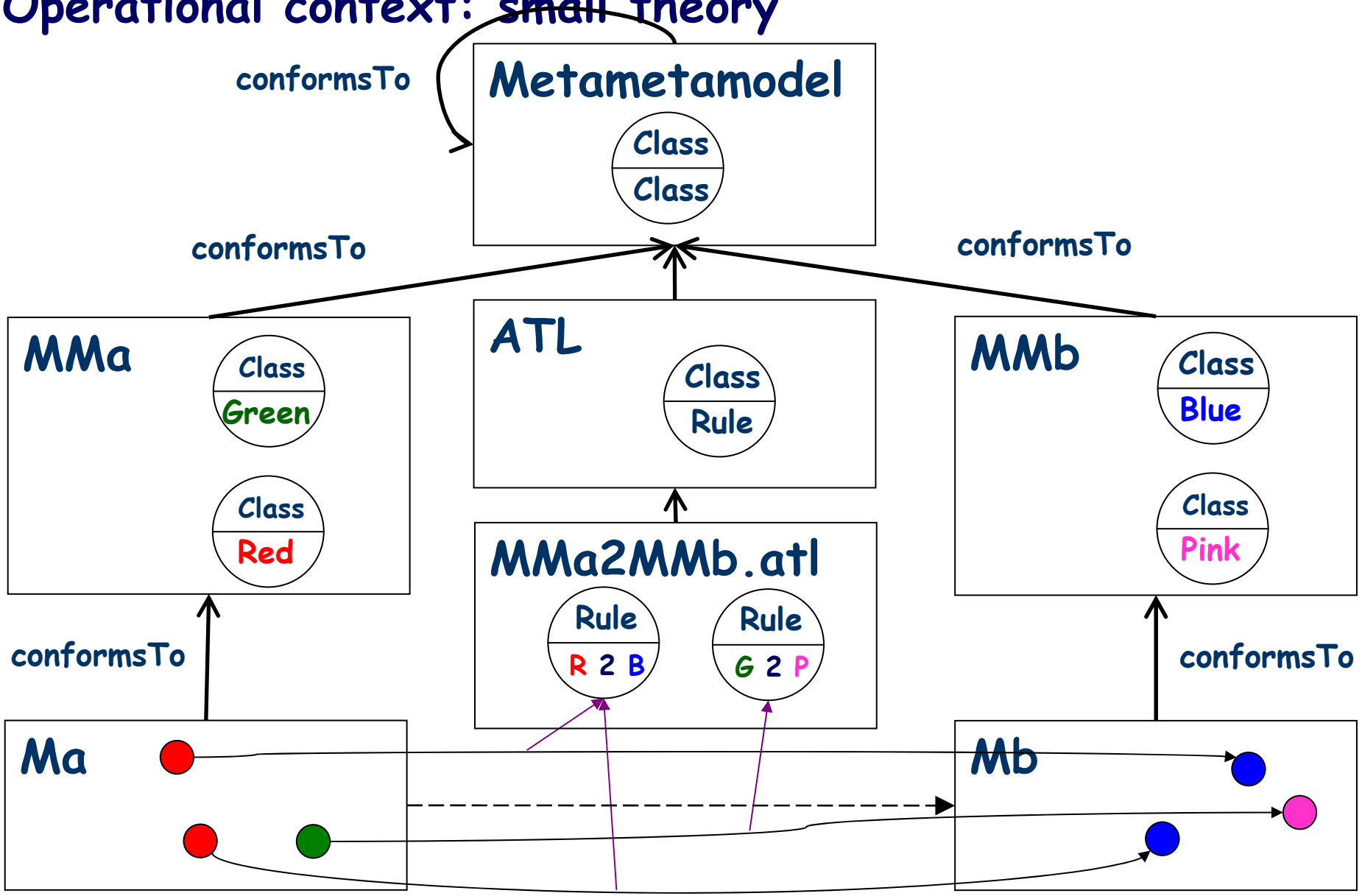
Definitions

- A model transformation is the automatic creation of target models from source models.

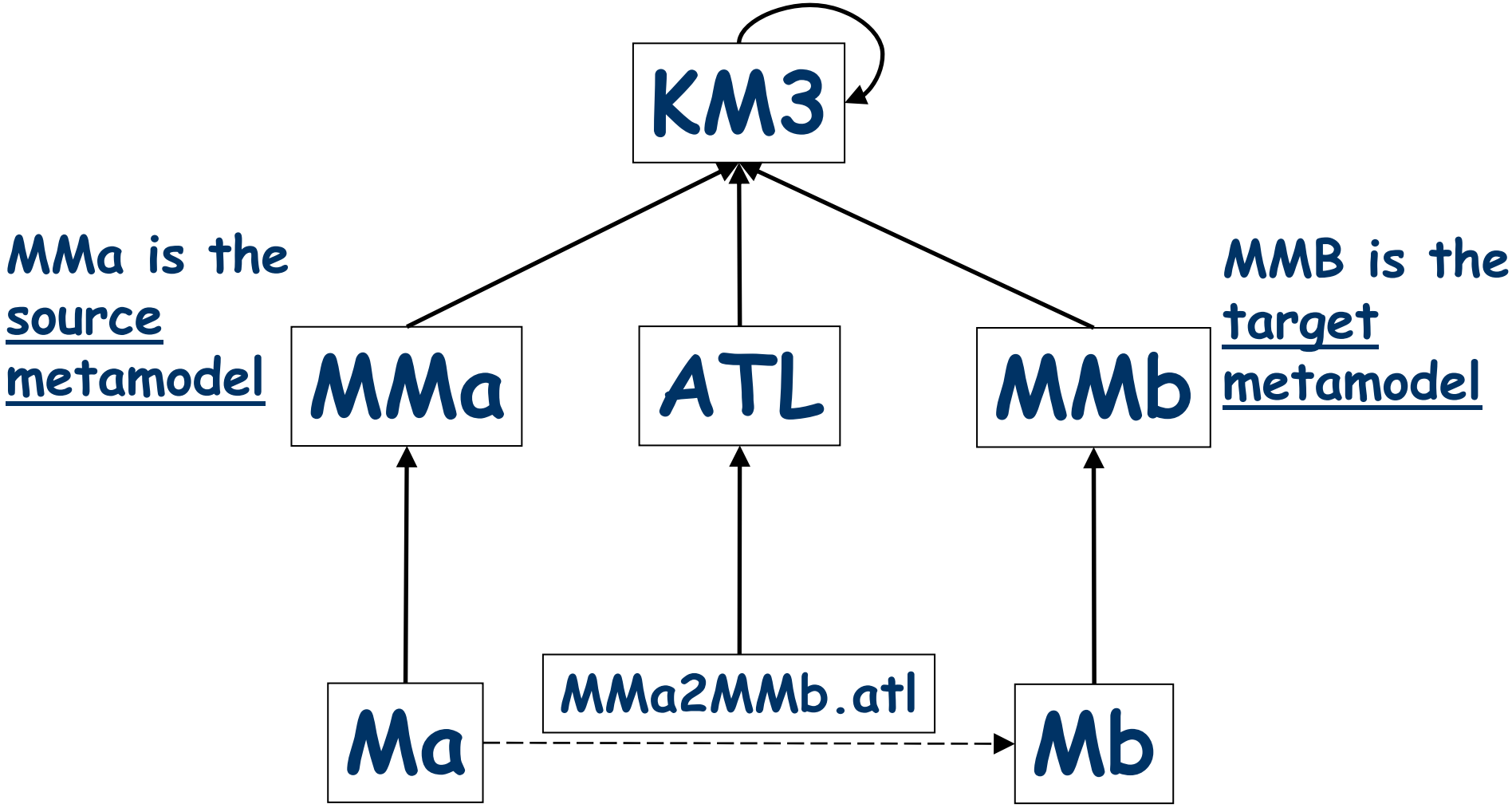
- Model transformation is not only about M1 to M1 transformations:
  - M1 to M2: promotion,
  - M2 to M1: demotion,
  - M3 to M1, M3 to M2, etc.



Operational context: small theory



Operational context of ATL



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, unless the **nodefault** modifier is used.
  - Unique lazy rules that are applied at most once for each match and only if it is referred to from other rules.
  - Lazy rules that are applied as many times for each match as it is referred to from other rules (possibly never for some matches).

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 rule 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.

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.

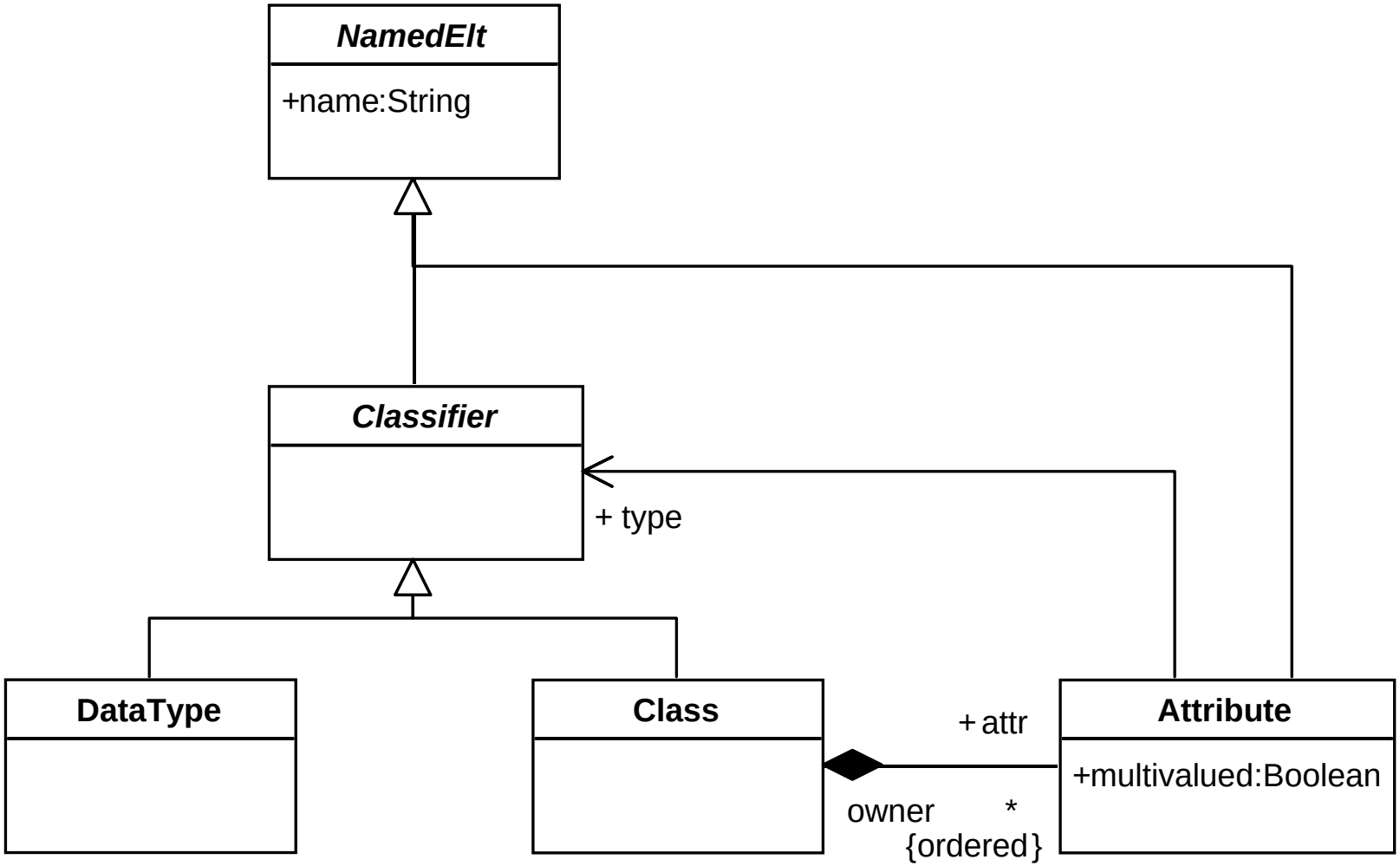
➔ ATL declaration of the transformation:

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

- The transformation excerpts used in this presentation come from:

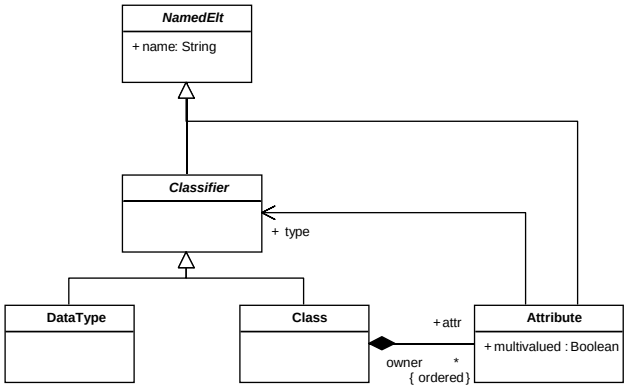
http://www.eclipse.org/m2m/atl/atlTransformations/#Class2Relational

Example: Class to Relational - Source Metamodel



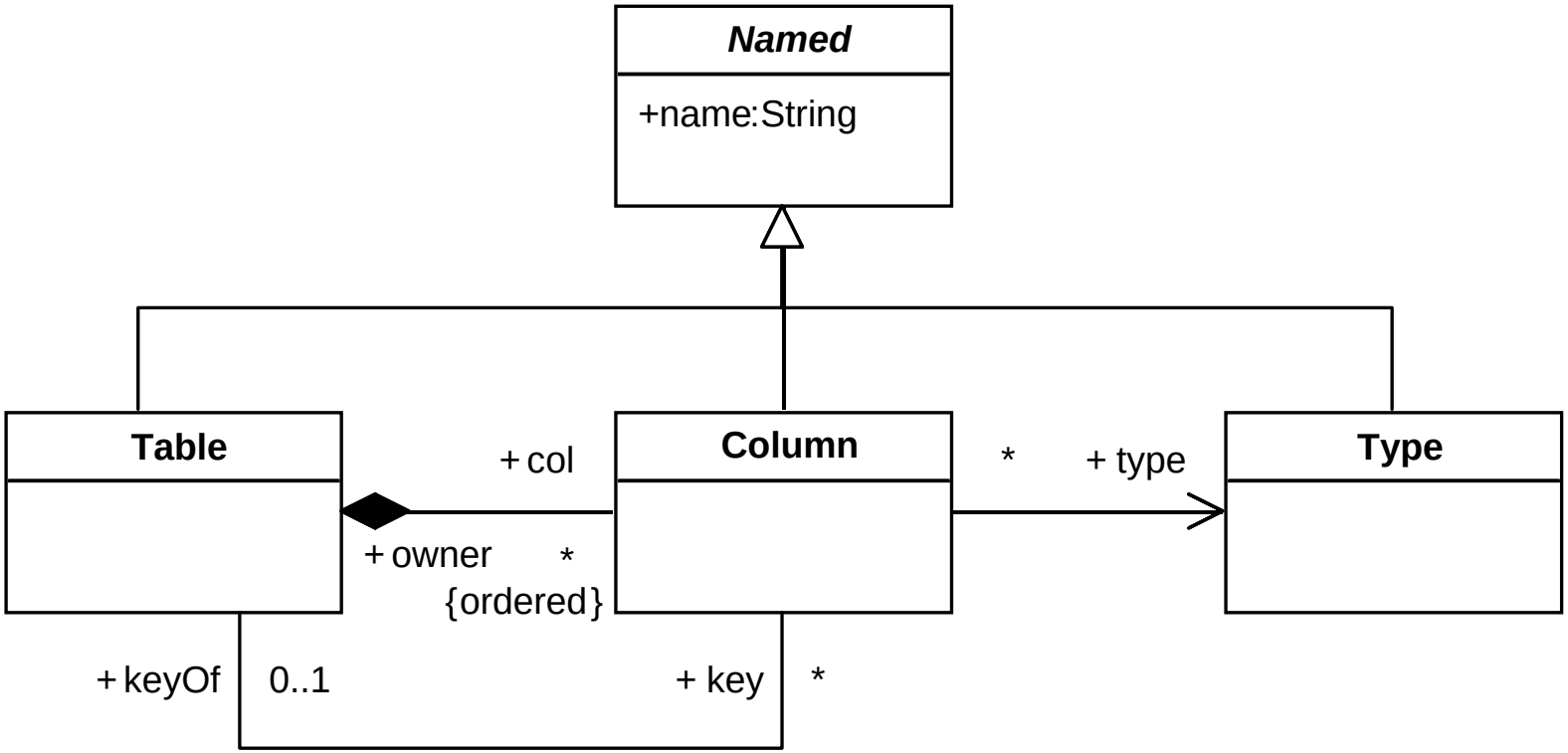
The Class Metamodel in KM3*

```
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;
        }
}
```

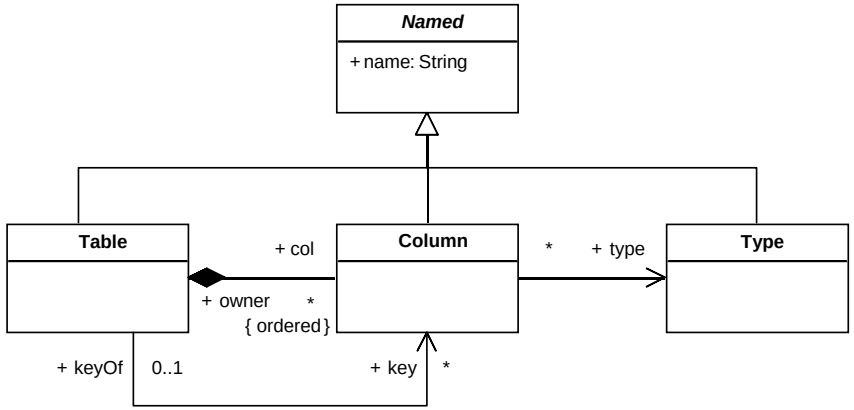


*For more information on KM3 see http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/

Example: Class to Relational – Target Metamodel

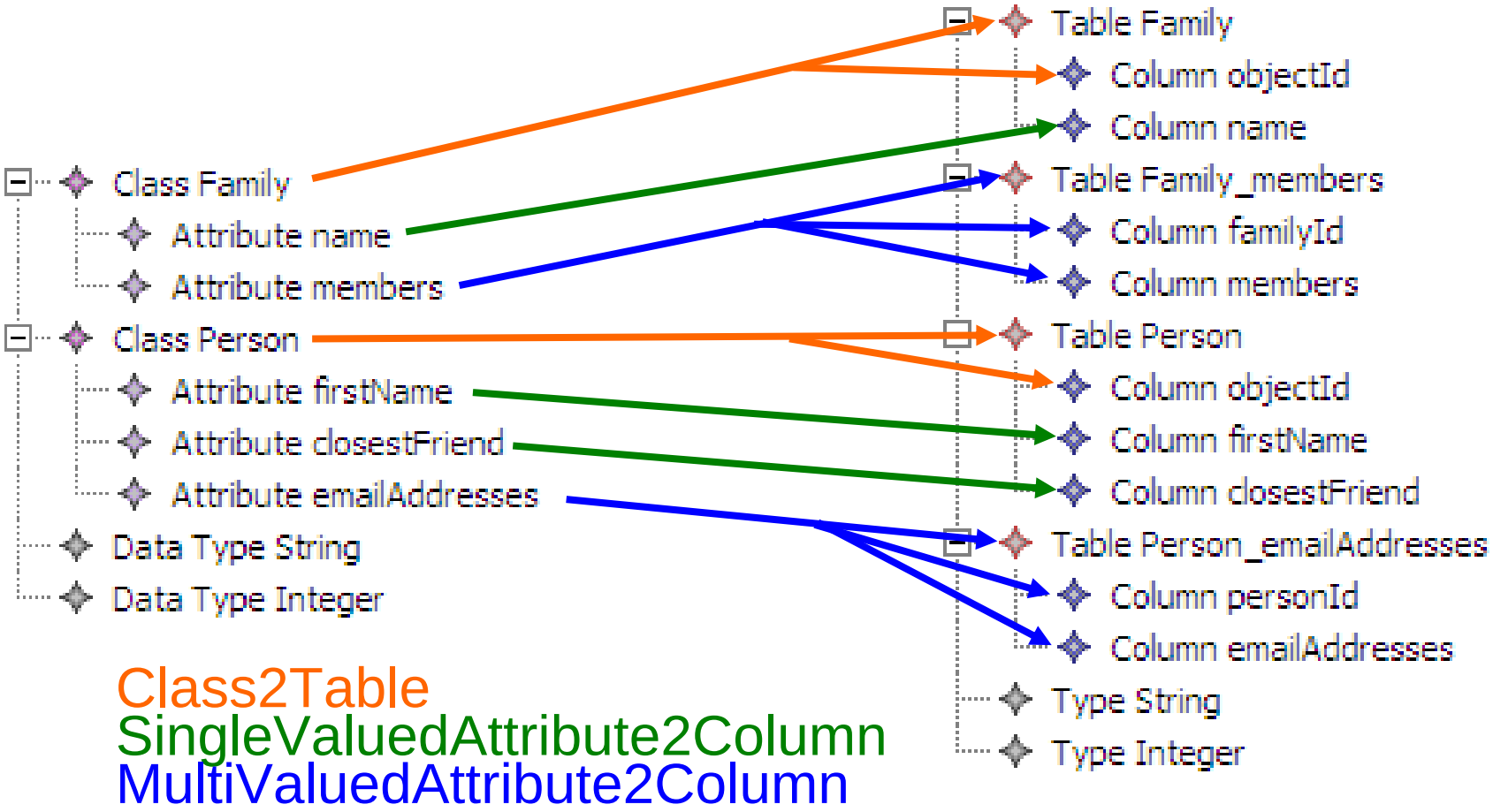


The Relational Metamodel in KM3

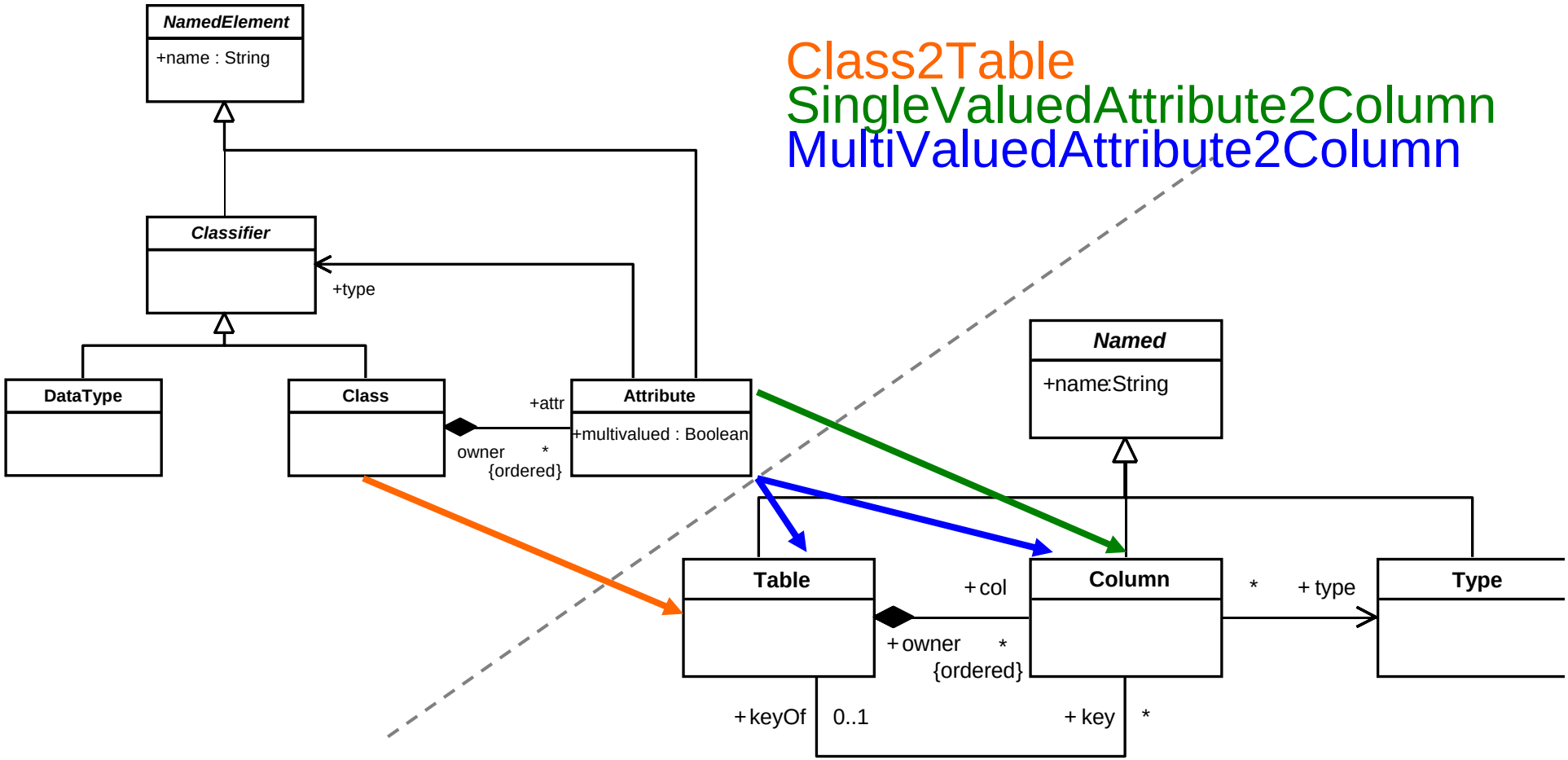



```
package Relational {

        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 {}
}
```

Sample Source and Target Models



ClassDiagram2Relational.atl – Overview



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 (1 Of 4)

- For each Class, create a Table :

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

Example: Class to Relational - Rule Class2Table (2 Of 4)

- 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 (3 Of 4)

- 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 (4 Of 4)

- 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

- For each single-valued Attribute create a Column:

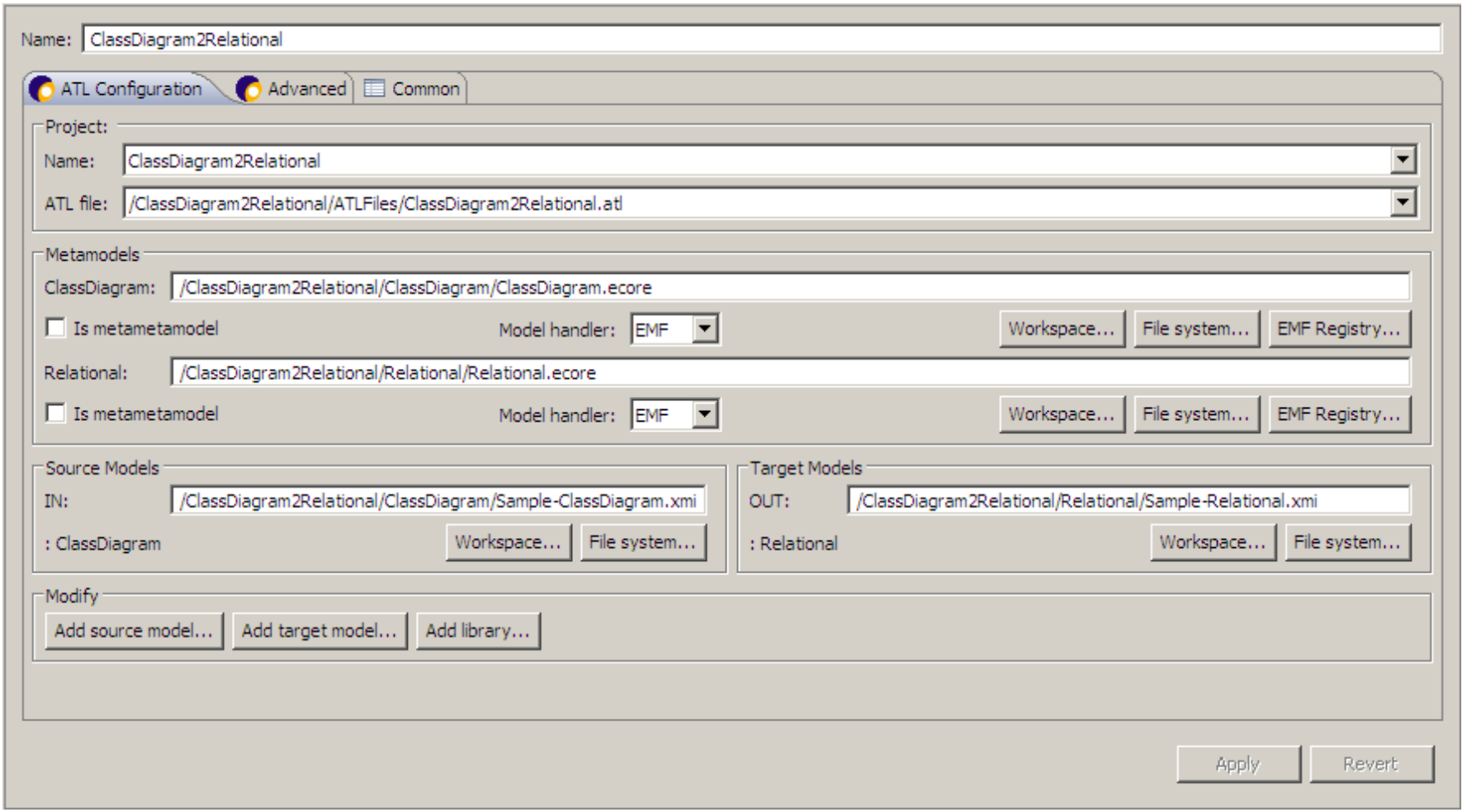
```
rule SingleValuedAttribute2Column {
    from        -- the guard is used for selection
        a : Class!Attribute (not a.multiValued)
    to
        c : Relational!Column (
                name <- a.name
        )
}
```

Example: Class to Relational - Rule MultiValuedAttribute2Column

- For each multi-valued Attribute create a Table, 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
                )
}
```

ATL Launch Configuration



Launching ATL using ANT

```
<project name="ClassDiagram2Relational" default="main">
	<property name="source" value="ClassDiagram/Sample-ClassDiagram.xmi"/>
	<property name="target" value="Relational/Sample-Relational.xmi"/>

	<target name="main" depends="loadMetamodels">
		<am3.loadModel modelHandler="EMF"
			name="myClassDiagram" metamodel="ClassDiagram"
			path="${source}"/>

		<am3.atl path="ATLFiles/ClassDiagram2Relational.atl">
			<inModel name="ClassDiagram" model="ClassDiagram"/>
			<inModel name="IN" model="myClassDiagram"/>
			<inModel name="Relational" model="Relational"/>
			<outModel name="OUT" model="myRelational" metamodel="Relational"/>
		</am3.atl>

		<am3.saveModel model="myRelational" path="${target}"/>
	</target>

	<target name="loadMetamodels">
		<am3.loadModel modelHandler="EMF"
			name="ClassDiagram" metamodel="%EMF"
			path="ClassDiagram/ClassDiagram.ecore"/>
		<am3.loadModel modelHandler="EMF"
			name="Relational" metamodel="%EMF"
			path="Relational/Relational.ecore" />
	</target>
</project>
```

Reading about ATL

- Jouault, F., and Kurtev, I.: *Transforming Models with ATL*. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica. 2005.
- http://www.eclipse.org/m2m/atl/doc/
- http://wiki.eclipse.org/ATL

Object Constraint Language (OCL)

- Originally intended to express constraints over UML models, for instance:

  context Person inv: self.age > 0
- Extended to query any model
- Used in several transformation languages (e.g., ATL, QVT) to compute values from the source models
- Specification:
  - The version on which ATL is based is available from: http://www.omg.org/docs/ptc/03-10-14.pdf
  - Section 7: language overview
  - Section 11: standard library
    - Section 11.8: iterator expressions

OCL – Trying Out OCL with AMMA Web Service

- Online:
  - http://www.sciences.univ-nantes.fr/lina/atl/cgi-bin/ammawebservice.sh/query
- Locally:
  - Get & unzip AMMAWebService.zip
  - Launch *run.bat*
  - Open http://localhost:6885/query in your web browser
- Usage:
  - Testing OCL expressions
  - Navigating over models
- Drawbacks:
  - Must use preloaded models
  - Cannot specify a context

OCL – Running ATL Queries Within Eclipse

- Considering a given OCL expression, such as:

  5 mod 2
- Place it in an ATL query, for instance:

  query TestQuery = (5 mod 2).debug('Result');
- Where the call to OclAny.debug(String) : OclAny:
  - Prints the result on the console
  - Does not change the result (it returns the value on which it is called)

  ➔ This operation may be called anywhere (e.g., to print intermediate values)
- Advantages:
  - May query any model
  - May use debugger
- Drawbacks:
  - Cannot specify a context

OCL - Iterator Expressions Returning Collections

- collect
  - Query: Sequence {1, 2, 3, 4}->collect(e | e + 2)
  - Result: Sequence {3, 4, 5, 6}
- reject
  - Query: Sequence {1, 2, 3, 4}->reject(e | e > 2)
  - Result: Sequence {1, 2}
- select
  - Query: Sequence {1, 2, 3, 4}->select(e | e > 2)
  - Result: Sequence {3, 4}
- sortedBy
  - Query: Sequence {3, 4, 1, 2}->sortedBy(e | e)
  - Result: Sequence {1, 2, 3, 4}

OCL - Predicate Iterator Expressions Admitting Multiple Iterators

- exists

  Query: Sequence {1, 2, 3, 4}->exists(e | e > 2)
  Result: true
  Query: Sequence {1, 2, 3, 4}->exists(e | e < 1)
  Result: false
  Query: Sequence {1, 2, 3, 4}->exists(e, f | e = f + 3)
  Result: true

- forAll

  Query: Sequence {1, 2, 3, 4}->forAll(e | e > 2)
  Result: false
  Query: Sequence {1, 2, 3, 4}->forAll(e | e > 0)
  Result: true
  Query: Sequence {1, 2, 3, 4}->forAll(e, f | e <> f)
  Result: false

OCL - Other Iterator Expressions with One Iterator

- any
  - Query: Set{1, 2, 3, 4}->any(e | e > 2)
  - Result: 3
  - Query: Set{1, 2, 3, 4}->any(e | e > 2)
  - Result: 4
- isUnique
  - Query: Sequence {1, 2, 3, 4}->isUnique(e | e mod 2 = 1)
  - Result: false (Sequence {1, 0, 1, 0} contains duplicates)
  - Query: Sequence {1, 2, 3, 4}->isUnique(e | e)
  - Result: true
- one
  - Query: Set{1, 2, 3, 4}->one(e | e > 2)
  - Result: false

OCL - Iterate Expression

- Enables computation of any value from a collection
- Example:

```
Sequence {'one', 'two', 'three'}->iterate(e; acc : String = '(' |
   acc + if acc = '(' then '' else ', 'endif + e
) + ')'
```

Result: '(one, two, three)'

- Description:
  - The accumulator is initialized
  - For each element in the source:
    - The iterator gets the value of this element
    - The body is evaluated
    - The accumulator receives the resulting value
  - The result is the value of the accumulator at the end of the iteration

OCL - Querying Models - Examples

- Names of abstract classes of a KM3 metamodel:

```
KM3!Class.allInstances()->select(e | e.isAbstract)->collect(e |
    e.name)
```

- Elements without child element in an XML model:

```
XML!Element.allInstances()->select(e |
    not e.children->exists(c |
      c.oclIsKindOf(XML!Element)
    )
)
```

- KM3 classes without supertype:

```
KM3!Class.allInstances()->select(e | e.supertypes->size() = 0)
```

or

```
KM3!Class.allInstances()->select(e | e.supertypes->isEmpty())
```

Writing Interpreters in ATL

- An ATL interpreter typically evaluates a model into a value.
  - Example: simple mathematical.
- Evaluation may involve input data that is not part of the model:
  - Example: XPath expressions evaluates over an XML model.
- The interpreter may be part of a larger transformation.
- How to write an evaluator:
  - Define an evaluation helper on each metamodel class to evaluate.

Example: KM3 Metamodel for *Simple Expressions*

```
package Calc {
    abstract class Expression {}
    class IntegerExp extends Expression {
        attribute value : Integer;
    }
    class OperatorExp extends Expression {
        reference left container : Expression;
        attribute operator : String;
        reference right container : Expression;
    }
    datatype Integer;
    datatype String;
}
```

Example: ATL Interpreter for *Simple Expressions*

```
query CalcEval =
    let root : Calc!Expression.allInstances()->any(e |        -- select root
            e.refImmediateComposite().oclIsUndefined()) in
    root.eval().debug('Result');    -- evaluate
-- IntegerExp evaluation:
helper context Calc!IntegerExp def: eval() : Integer =
    self.value;
-- OperatorExp evaluation:
helper context Calc!OperatorExp def: eval() : Integer =
    if self.operator = '+' then
            self.left.eval() + self.right.eval()
    else if self.operator = '*' then
            self.left.eval() * self.right.eval()
    else
            self.operator.debug('UnsupportedOperator')
    endif endif;
```

Example: *Simple Expressions* in Java (1 Of 3)

```
abstract class Expression {
    public abstract int eval();
}

class IntegerExp extends Expression {
    int value;
    public IntegerExp(int value) {
        this.value = value;
    }
    public int eval() {
        return value;
    }
}
```

Example: *Simple Expressions* in Java (2 Of 3)

```
class OperatorExp extends Expression {
    Expression left;          String operator;      Expression right;
    public OperatorExp(Expression left, String operator, Expression right) {
        this.left = left;
        this.operator = operator;
        this.right = right;
    }
    public int eval() {
        int ret = 0;
        if("+".equals(operator)) {
            ret = left.eval() + right.eval();
        } else if("*".equals(operator)) {
            ret = left.eval() * right.eval();
        } else {
            System.err.println("Unsupported operator: " + operator);
            System.exit(1);
        }
        return ret;
    }
}
```

Example: *Simple Expressions* in Java (3 Of 3)

```
public class TestExpression {
    public static void main(String args[]) {
        System.out.println(    // evaluating 1 + 4 * 3
            new OperatorExp(
                new IntegerExp(1),
                "+",
                new OperatorExp(
                    new IntegerExp(4),
                    "*",
                    new IntegerExp(3)
                )
            ).eval()
        );
    }
}
```

Writing Serializers in ATL

- A serializer transforms a source model into a String.
- It is a special kind of interpreter that evaluates into a String representing the evaluated model.
- Pros:
  - Any string may be computed (e.g., XML, HTML).
  - Easily usable from within a transformation (e.g., when parts of the source model correspond to strings in the target model).
- Cons:
  - A TCS syntax specification can already be used for serialization.
  - Some complex syntactic rules (e.g., proper parenthesizing of expressions) are hard to represent (but TCS does it automatically).

Example: ATL Serializer for *Simple Expressions*

```
query CalcEval =
    let root : Calc!Expression.allInstances()->any(e | -- select root
            e.refImmediateComposite().oclIsUndefined()) in
    root.serialize().debug('Result');    -- serialize
-- IntegerExp serialization:
helper context Calc!IntegerExp def: serialize() : String =
    self.value.toString();
-- OperatorExp serialization:
helper context Calc!OperatorExp def: serialize() : String =
    '(' +
            self.left.serialize() + ' ' +
            self.operator + ' ' +
            self.right.serialize() +
    ')';
```

Simple copy

```
-- Source metamodel: MMA
class A1 {
    attribute v1 : String;
    attribute v2 : String;
}
```

```
-- Target metamodel: MMB
class B1 {
    attribute v1 : String;
    attribute v2 : String;
}
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from
    s : MMA!A1
  to
    t : MMB!B1 (
      v1 <- s.v1,
      v2 <- s.v2
    )
}
```

Structure creation

```
-- Source metamodel: MMA
class A1 {
    attribute v1 : String;
    attribute v2 : String;
}
```

```
-- Target metamodel: MMB
class B1 { reference b2 : B2;
           reference b3 : B3;    }
class B2 { attribute v1 : String; }
class B3 { attribute v2 : String; }
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1andB2andB3 {
  from
    s : MMA!A1
  to
    t1 : MMB!B1 (
      b2 <- t2,
      b3 <- t3
    ),
    t2 : MMB!B2 (
      v1 <- s.v1
    ),
    t3 : MMB!B3 (
      v2 <- s.v2
    )
}
```

Structure simplification

```
-- Source metamodel: MMA
class A1 { reference a2 : A2;
           reference a3 : A3;    }
class A2 { attribute v1 : String; }
class A3 { attribute v2 : String; }
```

```
-- Target metamodel: MMB
class B1 {
    attribute v1 : String;
    attribute v2 : String;
}
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from
    s : MMA!A1
  to
    t : MMB!B1 (
      v1 <- s.a2.v1,
      v2 <- s.a3.v2
    )
}
```

Structure simplification (needlessly more complex)

```
-- Source metamodel: MMA
class A1 { reference a2 : A2;
           reference a3 : A3;    }
class A2 { attribute v1 : String; }
class A3 { attribute v2 : String; }
```

```
-- Target metamodel: MMB
class B1 {
    attribute v1 : String;
    attribute v2 : String;
}
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from s1 : MMA!A1, s2 : MMA!A2,
    s3 : MMA!A3 (s1.a2 = s2 and s1.a3 = s3)
  to
    t : MMB!B1 (
      v1 <- s.a2.v1,
      v2 <- s.a3.v2
    )
}
```

Traceability: implicit resolution of *default* elements

```
-- Source metamodel: MMA
class A1 { reference a2 : A2;
           reference a3 : A3;    }
class A2 { attribute v1 : String; }
class A3 { attribute v2 : String; }
```

```
-- Target metamodel: MMB
class B1 { reference b2 : B2;
           reference b3 : B3;    }
class B2 { attribute v1 : String; }
class B3 { attribute v2 : String; }
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from
    s : MMA!A1
  to
    t : MMB!B1 (
      b2 <- s.a2,    -- HERE
      b3 <- s.a3     -- HERE
    )
}
```

```
rule A2toB2 {
  from s : MMA!A2
  to     t : MMB!B2 (
    v1 <- s.v1       )
}
rule A3toB3 {
  from s : MMA!A3
  to     t : MMB!B3 (
    v2 <- s.v2       )
}
```

Remark: same result, less modular

```
-- Source metamodel: MMA
class A1 { reference a2 : A2;
           reference a3 : A3;    }
class A2 { attribute v1 : String; }
class A3 { attribute v2 : String; }
```

```
-- Target metamodel: MMB
class B1 { reference b2 : B2;
           reference b3 : B3;    }
class B2 { attribute v1 : String; }
class B3 { attribute v2 : String; }
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from
    s : MMA!A1
  to
    t1 : MMB!B1 (
      b2 <- t2,
      b3 <- t3
    ),
    t2 : MMB!B2 (
      v1 <- s.a2.v1
    ),
    t3 : MMB!B3 (
      v2 <- s.a3.v2
    )
}
```

Traceability: resolveTemp for additional elements

```
-- Source metamodel: MMA
class A1 { reference a2 : A2; }
class A2 { attribute v1 : String;
           attribute v2 : String; }
```

```
-- Target metamodel: MMB
class B1 { reference b2 : B2;
           reference b3 : B3;    }
class B2 { attribute v1 : String; }
class B3 { attribute v2 : String; }
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from     s : MMA!A1
  to
    t : MMB!B1 (
      b2 <- s.a2,
      b3 <-
         thisModule.resolveTemp(s.a2, 't2')
    )
}
```

```
rule A2toB2andB3 {
  from
    s : MMA!A2
  to
    t1 : MMB!B2 (
      v1 <- s.v1
    ),
    t2 : MMB!B3 (
      v2 <- s.v2
    )
}
```

Traceability: resolveTemp even for first element

```
-- Source metamodel: MMA
class A1 { reference a2 : A2; }
class A2 { attribute v1 : String;
           attribute v2 : String; }
```

```
-- Target metamodel: MMB
class B1 { reference b2 : B2;
           reference b3 : B3;    }
class B2 { attribute v1 : String; }
class B3 { attribute v2 : String; }
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from s : MMA!A1
  to t : MMB!B1 (
       b2 <- -- possible but complex
          thisModule.resolveTemp(s.a2, 't1'),
       b3 <-
          thisModule.resolveTemp(s.a2, 't2')
    )
}
```

```
rule A2toB2andB3 {
  from
    s : MMA!A2
  to
    t1 : MMB!B2 (
      v1 <- s.v1
    ),
    t2 : MMB!B3 (
      v2 <- s.v2
    )
}
```

Structure creation revisited with resolveTemp

```
-- Source metamodel: MMA
class A1 {
    attribute v1 : String;
    attribute v2 : String;
}
```

```
-- Target metamodel: MMB
class B1 { reference b2 : B2;
           reference b3 : B3;    }
class B2 { attribute v1 : String; }
class B3 { attribute v2 : String; }
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1andB2andB3 {
  from
    s : MMA!A1
  to
    t1 : MMB!B1 (-- possible but complex
      b2 <- thisModule.resolveTemp(s, 't2'),
      b3 <- thisModule.resolveTemp(s, 't3')
    ),
    t2 : MMB!B2 (
      v1 <- s.v1
    ),
    t3 : MMB!B3 (
      v2 <- s.v2
    )
}
```

Kinds of matched (declarative) rules

- We have only seen standard rules in default mode so far.

Kind of rule	Number of references to source pattern	Number of times the target pattern gets created	Kind of traceability link created
standard	0 1 n > 1	1 1 1	*default* or not (using keyword **nodefault**)
unique lazy	0 1 n > 1	0 1 1	Not *default*
lazy	0 1 n > 1	0 1 n	Not *default*

Syntax of the different kinds of matched rules

Kind of rule	Definition	Reference
standard **default**	**rule** R1 { **from** s : MMA!A1 **to** t : MMB!B1 }	<value of type MMA!A1> or <collection of MMA!A1>
standard **nodefault**	**nodefault rule** R1 { **from** s : MMA!A1 **to** t : MMB!B1 }	*not currently possible*
unique lazy	**unique lazy rule** R1 { **from** s : MMA!A1 **to** t : MMB!B1 }	thisModule.R1(<value of type MMA!A1>) *See (1)*
lazy	**lazy rule** R1 { **from** s : MMA!A1 **to** t : MMB!B1 }	thisModule.R1(<value of type MMA!A1>) *See (1)*

(1) For collections: aCollection->collect(e | thisModule.R1(e))

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 {
  -- ...
}
```

Copy class inheritance without rule inheritance

```
-- Source metamodel: MMA
class A1 { attribute v1 : String; }
class A2 extends A1 {
   attribute v2 : String;
}
```

```
-- Target metamodel: MMB
class B1 { attribute v1 : String; }
class B2 extends B1 {
   attribute v2 : String;
}
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from
    s : MMA!A1
  to
    t : MMB!B1 (
      v1 <- s.v1
    )
}
```

```
rule A2toB2 {
  from
    s : MMA!A2
  to
    t : MMB!B2 (
      v1 <- s.v1,
      v2 <- s.v2
    )
}
```

Copy class inheritance with rule inheritance

```
-- Source metamodel: MMA
class A1 { attribute v1 : String; }
class A2 extends A1 {
   attribute v2 : String;
}
```

```
-- Target metamodel: MMB
class B1 { attribute v1 : String; }
class B2 extends B1 {
   attribute v2 : String;
}
```

```
module MMAtoMMB;
create OUT : MMB from IN : MMA;
rule A1toB1 {
  from
    s : MMA!A1
  to
    t : MMB!B1 (
      v1 <- s.v1
    )
}
```

```
rule A2toB2 extends A1toB1 {
  from
    s : MMA!A2
  to
    t : MMB!B2 (
      v2 <- s.v2
    )
}
```

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;
```

Guidelines

- Make your transformation as complex as necessary but as simple as possible:
  - Prefer declarative over imperative: only use imperative for the part of a transformation that needs it if it even does.
  - Prefer simpler constructs over more complex ones:
    - Use standard rules when possible, otherwise use unique lazy rules, and use lazy rules only if necessary.
    - Only use resolveTemp if necessary.
    - Prefer iterators (e.g., select, collect) over iterate.

End of the presentation

- Thanks
  - Questions?
  - Comments?