ATL: ATLAS Transformation Language

MISO - Uniandes
“Models are first class entities”

“Transformations are models”

“Transformations are assets”
What is ATL?

- ATL is a model transformation language (MTL) developed at INRIA to answer the QVT Request For Proposal.
- It is specified both as a metamodel and as a textual concrete syntax.
- It is a hybrid of declarative and imperative.
What is ATL?

- In declarative style:
  - simple mappings can be expressed simply.

- Imperative constructs:
  - are provided to express more complex mappings
An ATL transformation program is composed of rules that define:

- how source model elements are matched and navigated
- how to create and initialize the elements of the target models.
ATL and Ecore
Input MM
Input M
Output MM

Input MM
Input M
Output MM

ATL Transformation rules

XLMI

ATL Engine

Output M'

XMI
module UML2JAVA;
create OUT : JAVA from IN : UML;
Structure of an ATL transformation

- An ATL module corresponds to a model to model transformation.

Elements:
- A header section:
  - defines attributes that are relative to the transformation module;
- An optional import section:
  - enables to import some existing ATL libraries
- A set of helpers:
  - can be viewed as an ATL equivalent to Java methods
- A set of rules:
  - defines the way target models are generated from source ones.
ATL modules

- They are the files that contain the transformations
- .atl extension
- Consist of:
  - Header (mandatory)
  - Import
  - Helpers
  - Rules
Import

- This section is optional
- Defines the ATL libraries to be imported
- Example:

  `uses strings;`
The target models declaration is introduced by the *create* keyword.

The source models are introduced either by the keyword *from* (in normal mode) or *refines* (in case of refining transformation).

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

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

output Metamodel

input Metamodel
Helpers

- They are **variables** or **functions** used to implement code that can be reused

```java
helper def: objectIdType : Relational!Type =
    Class!DataType.allInstances()->
    select(e | e.name = 'Integer')->first();
```

- Using a variable:

```java
type <- thisModule.objectIdType;
```
**Helpers**

- **Using a function:**

```java
helper context Book!Book def : getAuthors() : String =
    self.chapters->collect(e | e.author)
    ->asSet()->
    iterate(authorName; acc : String = '' |
        acc +
        if acc = ''
            then authorName
        else ' and ' + authorName
    endif);
```

- **Calling the function:**

```java
b.getAuthors()
```
Transformation Rules

- three different kinds of rules:
  - matched rules (declarative programming)
  - called rules (imperative programming).
  - Lazy rules (?)
**Matched Rules**

```plaintext
rule DataType2Type {
  from
dt : Class!DataType
  to
out : Relational!Type {
    name <- dt.name
  }
}
```
_matched Rules

rule DataType2Type {
  from
    dt : Class!DataType
  to
    out : Relational!Type {
      name <- dt.name
    }
}
Matched Rules

- We need:
  - To access any element of the input model
  - To create elements of the output model from information of the elements of the input model
- The access to the elements is done using The OCL: Object Constraint language
Matched Rules: Input

- The input pattern consists of:
  - the keyword `from`,
  - the declaration of an input variable
  - an OCL expression that returns the input element to be transformed

```
from
dt : Class!DataType
```
A filter is an OCL expression that restraints the elements of the input model to those that satisfy a set of constraints.

Example:
- Refer to the instances of the `Parameter` element in the UML Metamodel, where its `kind` is different from the value `#pdk_return`:

  ```ocl
  e : UML!Parameter (e.kind <> #pdk_return)
  ```
Matched Rules: Output

- The output pattern declares:
  - on which output elements of the output model, the input elements matching the input pattern will be transformed

- The implementation of the output pattern declares the details of the transformation

- It is possible to have more than one element in the output pattern
Matched Rules: Output (cont.)

- Declares a variable and a sequence of assignment statements.

\[
\text{to}
\]
\[
\text{out} : \text{Relational!Type (}
\text{name} \leftarrow \text{dt.name}
\text{)}
\]

Building and instance of the element Type of the output model
The name attribute of the new element will have as value the
value of the name attribute of the element referenced by the variable dt
Called Rules

- They are a particular type of helpers.
- They have to be explicitly called to be executed.
- They can accept parameters.
- They can generate target model elements as matched rules do.
- A called rule has to be called from an imperative code section, either from a match rule or another called rule.
Called Rules

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

- Generates Person target model elements.
- Accepts two parameters that correspond to the name and the surname of the Person.
- The target pattern allocates a Person class each time the rule is called, and initializes the `name` attribute of the allocated model element.
- The imperative code section is executed after the initialization of the allocated element.
Tranformation examples

1. ATL to Problem
2. Ant to Maven
3. BibTeXML to DocBook
4. Book to Publication
5. **Class to Relational**
6. Geometrical transformations
   Grafcet to PetriNet
7. Java to Table
8. KM3 to DOT
9. KM3 to Metrics
10. KM3 to Problem
11. Make to Ant
12. Maven to Ant
13. Microsoft Office Excel Extractor
14. Microsoft Office Excel Injector
15. 
16. Microsoft Office Excel to Software
17. Quality Control
18. Monitor to Semaphore
19. PathExp to PetriNet
20. Software Quality Control to Bugzilla
21. Software Quality Control to Mantis Bug Tracker
22. Table to Microsoft Office Excel
23. UML to Amble
24. UML to Java
25. UML to MOF
26. UML Activity Diagram to MSProject
27. UMLDI to SVG
28. XSLT to XQuery
Figure 1 Simplified Java Metamodel
UML MM expressed using MOF XMI

```xml
<Model:Class xmi.id = 'a9' name = 'JavaElement' annotation = ''
  isRoot = 'false'
  isLeaf = 'false' isAbstract = 'true' visibility = 'public_vis'
  isSingleton = 'false'>
  <Model:Namespace.contents>
    <Model:Attribute xmi.id = 'a10' name = 'name' annotation = ''
      scope = 'instance_level'
      visibility = 'public_vis' isChangeable = 'true'
      isDerived = 'false'>
      <Model:StructuralFeature.multiplicity>
        <XMI.field>1</XMI.field>
        <XMI.field>1</XMI.field>
        <XMI.field>true</XMI.field>
        <XMI.field>true</XMI.field>
      </Model:StructuralFeature.multiplicity>
      <Model:TypedElement.type>
        <Model:PrimitiveType xmi.idref = 'a6'/>  
      </Model:TypedElement.type>
    </Model:Attribute>
  </Model:Namespace.contents>
</Model:Class>
```
Figure 2 Simplified UML Metamodel
UML2Java: Transformation rules

1. For each Package element of UML MM create an element Package of Java MM
   - The names have to be the same in the java case the name have be extended

```plaintext
rule P2P {
  from e: UML!Package (e.oclIsTypeOf (UML!Package))
  to out : JAVA!Package (name <- e.getExtendedName())
}
```
2. For each Operation element of UML MM create an Method element of Java MM
   names, types, modifiers have to be the same
rule O2M {  
  from e : UML!Operation  
  to out : JAVA!Method (  
    name <- e.name,  
    isStatic <- e.isStatic(),  
    isPublic <- e.isPublic(),  
    owner <- e.owner,  
    type <-  
      e.parameter->  
      select(x|x.kind= #pdk_return)->asSequence()->first().type,  
    parameters <-  
      e.parameter->select(x|x.kind<>  
        #pdk_return)->asSequence()  
    )  
}
Transformation Engine

- Implemented in Java
- Can use EMF or MDR as model repository
- Input/output models and metamodels are managed by the underlying model repository
- ATL has a proprietary repository to load the model transformations.
ATL Eclipse Plug-in

- There is a complete ATL perspective in Eclipse
- Also a perspective to Debug ATL transformations
- There are:
  - an ATL editor
  - an Outline view to review the current elements
ATL Eclipse Plug-in
Referencias