MDA and Model Transformation in Practice

Jon Oldevik, SINTEF
jon.oldevik at sintef.no
Atlas Transformation Language (ATL) Tutorial

Jon Oldevik, SINTEF
Introduction

- The Atlas Transformation Language (ATL) is a hybrid language (a mix of declarative and imperative constructions) designed to express model transformations as described by the MDA™ approach.
- It is not QVT, but similar and with the corresponding functionality.
- A transformation model in ATL is expressed as a set of transformation rules.
- The recommended style of programming is declarative.
- OCL is used to expression constraints on rules
  - Guards (constraints) on the entry point for a rule
- Different kinds of M3/M2 (meta)metamodel technology supported: Netbeans MDR and EMF Ecore
  - => Can use either EMF or MDR metamodels as input and output.
- Can also be used to produce textual output.
A Simple Example: Two metamodels

The Simple JavaClass Metamodel

The Simple Class Metamodel
A Simple Example – A transformation

module SM2JM;
create OUT:JavaModel from IN:ClassModel;

rule Model2Package {
  from inputModel : ClassModel!Model
  to javaPackage : JavaModel!Package
  {
    name <- inputModel.name,
    classes <- inputModel.modelElements
  }
}

rule ClassToClass {
  from inputClass : ClassModel!Class
  to javaClass : JavaModel!JavaClass
  {
    name <- inputClass.name,
    description <- 'Java class ' + inputClass.name,
    variables <- inputClass.attributes
  }
}
Simple example cont.

```plaintext
rule Attribute2Variable {
  from clAttribute : ClassModel!Attribute
  to javaAttribute : JavaModel!Variable
  (name <- clAttribute.name,
   type <- clAttribute.type.name,
   visibility <- 'private'
  )
}

helper context ClassModel!Class def : getJavaClassName() : String =
  self.name + 'Impl'
```

---

SINTEF ICT
ATL language structure

- **ATL File**
  - Ends with '.atl'

- **ATL module**
  - Header
  - Import
  - Rules
  - Helpers

```java
module SM2JM;
create OUT:JavaModel from IN:ClassModel;
```

```java
import strings;
```

```java
rule Model2Package {
  from inputModel : ClassModel!Model
  to javaPackage : JavaModel!Package
  {
    ... 
  }
}
```

```java
helper context CM!Class def : helperName() : String = ...
... some OCL expression
```
The module header

Module name declaration \( \text{[module SM2JM;]} \)
- The module name must match the file name (case not important)
- Module file: BookToPublication.atl
- This is compiled to ATL assembler code (ATL Stack Machine), file name BookToPublication.asm, which can be executed.

The keyword create introduces the target models declaration. \( \text{[create OUT : JavaModel]} \)

The keywords from introduces the source models declaration. \( \text{[from IN : ClassModel]} \)

- The names \( \text{IN, OUT, Publication, Book} \) must be specified in a launch configuration for execution to succeed.
An ATL module can import definitions specified on other modules. This is done with the keyword ‘uses’ followed by the library name. There can be several imports.

*Example:* `uses strings;`
- Imports the external string library.
- Do not declare imports that are not necessary.
- The `string.atl` library contains many useful functions for Strings but is not necessary for the simple Book to Publication example.
- The location of used libraries must be specified in the launch configuration for the execution…
Transformation Rules

- Rules describe the transformation from a source model to a target model by relating metamodels.
- Each rule contains a unique name. It is introduced by the keyword rule that is followed by the rule’s name. Its implementation is surrounded by curly brackets.
- The source pattern (from part) is used to declare which element type of the source model has to be transformed.
  - The source pattern consists of the keyword from, an in-variable declaration and optionally a filter.
  - A filter is an OCL expression restricting the rule to elements of the source model that satisfy certain constraints.
- The target pattern (the to part) is used to declare which element(s) of the target model the source pattern has to be transformed to.
Transformation rules - Guards

```
rule ClassToClass {
from inputClass : ClassModel!Class (inputClass.attributes->size() > 2)
to javaClass : JavaModel!JavaClass
    (name <- inputClass.getJavaClassName(),
     description <- 'Java class ' + inputClass.name,
     variables <- inputClass.attributes)
```

Guards brukes er constraints som kontrollerer om en regel skal kjøres eller ikke => Fungerer som pre-conditions for regler.

Bruker OCL-uttrykk.

Typisk en sjekk på egenskaper hos inputelementet.
Transformation Rules – Multiple output elements

```csharp
rule ClassToClass {
  from inputClass : ClassModel!Class
to javaClass : JavaModel!JavaClass
  (name <- inputClass.getJavaClassName(),
   description <- 'Java class ' + inputClass.name,
   variables <- inputClass.attributes,
   methods <- javaClassConstructor
  ),
  javaClassConstructor : JavaModel!Method
  (name <- inputClass.name,
   body <- ' // Default constructor'
  ),
  secondClass : JavaModel!JavaClass
  (name <- inputClass.name + '_Another',
   description <- 'Another class ...' + inputClass.name
   );
}
Helpers

- Helpers can be used to define (global) variables and functions. Helper functions are OCL expressions. They can call each other (recursion is possible) or they can be called from within rules. In general, they serve to define repetitive pieces of code in one place.

- A helper function has the following structure:
  - It starts with the keyword `helper` and ends with a semicolon. In between is an OCL expression.
  - The function name is introduced with `def:` and is followed by brackets.
  - The return type is between a colon and an equals sign that serves as starting point for the function implementation.
  - The function finishes with a semicolon.
The notion of the term context is similar to OCL and it may be compared to an input parameter of a method.

A context variable is specified with the help of the ATL path expression \texttt{metamodel!element} and is accessible via the self variable. If no context is specified, the module itself is taken as context.

\begin{verbatim}
helper context ClassModel!Class def: checkAttributeCount() : Boolean = self.attributes->size() > 2
;

helper context ClassModel!Attribute def: isFAttribute() : Boolean = self.name.substring(1,1).toLowerCase() = 'f'
;
\end{verbatim}
module SM2JM2;
create OUT:JavaModel from IN:ClassModel;

helper def: model : ClassModel!Model = ClassModel!Model.allInstances()->asSequence()->first();

helper def: javaClassPrefix : String = 'Impl';

self.name + thisModule.javaClassPrefix

Defining a global helper

Using a global helper
References

- Model Driven Architecture (MDA), http://www.omg.org/mda
- GMT Homepage, http://www.eclipse.org/gmt/
- ATL Binaries, http://dev.eclipse.org/viewcvs/indextech.cgi/%7Echeckout%7E/gmt-home/download/atl/installBinaries.html
Model to Text tool – “MOFScript”

Model to Text Transformation
Meeting WP5 requirements
Jon Oldevik, SINTEF
MOFScript-tool overview

- Eclipse plugin
- Lexical Editor for MOFScript
  - Syntax might be changed according to new requirements + OMG Model 2
  - Text process
  - Tailored for generating code from models
- Procedural language
  - Rules are functions which are called explicitly.
  - Might be integrated with a rule matching mechanism
  - Few control structures – simple language
- Execution Engine in Eclipse
  - Interpreted execution
  - Generation of output files based on model input
- Supports EMF metamodels and models
Language structure

- Textmodule
  - `textmodule UML2Java (in uml:uml2)`

- Entrypoint rule
  - `uml.Model::main () { self.ownedMember->forEach(p:uml.Package) { p.mapPackage() } }`

- Rules
  - `uml.Package::mapPackage () { self.ownedMember->forEach(c:uml.Class) c.mapClass() }`
Language structure

- **Properties**

  ```
  property packageName = "org.mypackage"
  var myVariable
  ```

- **Files**

  ```
  file (c.name + ".java")
  <% package %> c.ownerPackage.name <%; %>
  println ("public class" + c.name);
  ```

- **Iterators**

  ```
  c.elements->forEach(e:db:DBTable) {
    // statements.
  }
  ```

- **Conditional statements**

  ```
  if (c.name != "Car") {
    // statements
  }
  ```

- **Escaped output**

  ```
  <%
  public class %> c.name <% extends Serializable {
  %>
  ```
Logical expressions

- Logical expressions
  - Used in iterators (forEach) and if statements
  - If statements

<table>
<thead>
<tr>
<th>Expression = LogicalExpression</th>
<th>ComparisonExpression</th>
<th>ValueExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogicalExpression = (LogicalExpression)</td>
<td>not Expression</td>
<td>Expression and Expression</td>
</tr>
<tr>
<td>ComparisonExpression = ValueExpression {...=&lt;&gt;!=...} ValueExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValueExpression: &lt;literal&gt;</td>
<td>literal + ValueExpression</td>
<td></td>
</tr>
</tbody>
</table>

```javascript
self.ownedAttribute->forEach(p : uml.Property | p.association != null){
  ..
}

if (self.name = "Car" or self.name = "Person") {
}
```
Print statements

- **print | println**
  - Prints output to file
    - Uses "last" define file statement.
    - Or standard output (`stdout.println("…")`)

- **Escaped output**
  - Same behaviour as print statements
    `<% escaped output %> c.name `<% more escaped output %>`
Built-in operations

■ String operations
  ■ equals(), equalsIgnoreCase(), size(), substring(), substringBefore(), toLowerCase(), toUpperCase(), firstToLowerCase, firstToUpperCase(), indexOf(), endsWith(), startsWith(), trim(), normalizeSpace(), replace()
  ■ used like this: "myString"->toLowerCase(), c.name->size(), c.name->endsWith("Fa")

■ Collection operations
  ■ size()
Example – UML 2 Java

textmodule UML2Java (in uml:uml2)

property rootdir = "c:/tmp2/"
property package_name = "org.sintef.no"
property package_dir = "org/sintef/no/"
property ext = ".java"
property author = "Jon Oldevik"

uml.Model::main () {
    self.ownedMember->forEach(p:uml.Package) {
        p.mapPackage()
    }
}

uml.Package::mapPackage () {
    self.ownedMember->forEach(c:uml.Class)
        c.mapClass()

    self.ownedMember->forEach(p:uml.Package) {
        p.mapPackage()
    }
}
Example cont

```java
uml.Class::mapClass() : void {
    file (rootdir + package_dir + self.name + ext)
    self.classPackage()
    self.standardClassImport ()
    self.standardClassHeaderComment ()

    <%
    public class %> self.name <% extends Serializable { %>
        self.classConstructor()
        <%
        /*
        * Attributes
        */%>
        self.ownedAttribute->forEach(p : uml.Property | p.association = null)
        {
            p.classPrivateAttribute()
        }
    <%}%>
}

uml.Class::classPackage() {
    print ("package " +  package_name + ";")
}
...