Model-Driven Software Engineering

Model Transformations I

Dr. Jochen Küster (jku@zurich.ibm.com)
Contents

- Model Transformations in Model-Driven Software Engineering
- The Model Transformation Language ATL
- Developing Model Transformations using ATL
- Language Concepts of ATL
- Summary and References
Model Transformations in Model-Driven Software Engineering
Motivation for Model Transformations

- Model-Driven Software Engineering makes models key artifacts in the software engineering process
- When working with models, automation of recurring tasks can often be achieved by **model transformations**

- The MDA approach favors iterative refinement of models by **model transformations**
- Architecture-centric Model Driven Software Development **generates artifacts** (models, code) from models

Diagram:
- PIM
- PSM
- Domain Model
- Java Implementation
Applications of Model Transformations

- Generating more refined models from more abstract models
  - Example: Generating EJB classes from BusinessEntity classes

- Generating code from models
  - Example: Generating code from domain model in EMF

- Reverse engineering of abstract models from detailed models or code
  - Example: Generating class diagrams from Java code

- Mapping and synchronization of models describing a software system from different viewpoints
  - Example: Synchronization of class diagrams and sequence diagrams

- Model refactoring for improving the quality of models
  - Example: Refactoring of class diagrams by introducing patterns
What is a Model Transformation?

- **Models** can be defined in various modeling languages
  - Special case: Code can also be considered as a model

- A **model transformation definition** is specified in a dedicated model transformation language or in a programming language

- A **transformation engine** executes the model transformation definition and creates the target model

Source: [Czarnecki and Helsen]
In an Enterprise Application Integration scenario data models between two systems have to be exchanged.

The first system is developed at a government organization which is responsible for paying out child allowance.

The second system is maintained at another government organization, the registration office which keeps track of all persons.

To prevent fraud, the registration office should receive information about all persons registered at the child allowance office.

Unfortunately, the models used in each application differ and a model transformation must be developed.

The Family2Person transformation transforms Family models into Person models.
Transformation Example – Models and Meta-models

Source Meta-Model

Target Meta-Model

Child Allowance Metamodel

Registration Office Metamodel

Source Model

Target Model

<?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="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
</xmi:XMI>

<?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="Jim March"/>
  <Male fullName="Brandon March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Brenda March"/>
</xmi:XMI>
The **model transformation definition** consists of **transformation rules**

- Example given in the model transformation language **ATL**
Model Transformation Definition

- Expressed in a **model transformation language**
- Usually based on **model transformation rules**
- A rule captures the smallest unit of transformation
- Rules look different depending on the transformation language used

```plaintext
rule Member2Male {
    from
        s: Families!Member (not s.isFemale())
    to
        t: Persons!Male {
            fullName <- s.firstName + ' ' + s.familyName()
        }
}
```
Meta-Models in the Transformation Definition

- Meta-models define the **source** and **target** of the model transformation
- Elements of the meta-models are used in the model transformation definition for defining the model transformation

```plaintext
rule Member2Male {
  from
    s: Families!Member (not s.isFemale())
  to
    t: Persons!Male (
      fullName <- s.firstName + ' ' + s.familyName()
    )
}
```
Model Transformation Execution

- **Transformation engine** executes the model transformation definition
- **Reads** the source model and transforms it according to the model transformation definition to create a target model
- **Transformation engine** determines execution options
  - Source-target relationship
  - Rule scheduling
  - Rule organization
Selected Transformation Execution Options

- **Source-Target relationship**
  - Source-target execution, source model is read-only
  - In-place execution, source model is modified directly to create the target model

- **Rule scheduling**
  - Defines how different transformation rules scheduled
  - User defined or no user control

- **Rule organization**
  - Determines how rules can be organized
  - Rule can depend on other rules
Characteristics of Model Transformations

- **Unidirectional** model transformation
  - Only one direction of transformation is supported
  - A source model is transformed into a target model

- **Bidirectional** model transformation
  - Support for both directions of transformations
  - A source model is transformed into a target model
  - A target model is transformed into a source model
  - Can be used for model synchronization
  - Not easy to realize without sophisticated transformation language and execution environment
Characteristics of Model Transformations

- **Exogenous** model transformation
  - Source and target modeling language are not the same
  - Source model is translated into a target model
  - used when refining models into more detailed model

- **Endogenous** model transformation
  - Source and target modeling language are the same
  - Source model is transformed into a target model
  - used for refactoring of models
Overview of Model Transformation Languages

- In recent years, many model transformation languages have been defined
- Different status and maturity, different support of the characteristics

- **Atlas Transformation Language (ATL)**
  - Developed and maintained by AtlanMod team
  - Tool support available as Eclipse plug-in

- **Query Views Transformation (QVT)**
  - OMG standard
  - Relational and operational part

- Many others: **VIATRA, GReAT, Kermeta, Epsilon, Tefkat, …**
Model Transformations using ATL
Atlas Transformation Language (ATL)

- ATL stands for **Atlas Transformation Language**
- ATL is a transformation language, toolkit and development environment for model-to-model transformations
- Current version is ATL 3.1.0
- Integrated into Eclipse platform
- Developed originally by the AtlanMod Team at INRIA, Nantes, France
**ATL Concept Overview**

- ATL is a **hybrid language** and contains imperative and declarative features.
- An **ATL transformation** is specified using a set of **ATL rules** and **ATL helpers**.
- **ATL rules** specify how elements of a source model are translated into elements of a target model.
- An **ATL program** transforms a set of source models into a set of target models – contains ATL rules and ATL helpers.

![ATL Concept Diagram](image-url)
ATL Rules – Syntax and Semantics

A **ATL rule** is composed of

- the **source pattern** and the **target patterns** (mandatory)
- the **local variables** and the **imperative** sections (optional)

The ATL rule specifies the way generated target model elements must be initialized from each matched source model element.

```atl
rule Member2Male {
  from
  s: Families!Member (not s.isFemale())
  to
  t: Persons!Male {
    fullName <- s.firstName + ' ' + s.familyName()
  }
}
```
The source pattern specifies a model element variable that corresponds to the type of source elements the rule has to match.

This type corresponds to an entity of a source metamodel of the transformation.

The rule will generate target elements for each source model element that conforms to the matching type.

In order to match only a subset of the source elements that conform to the matching type, an optional condition can be specified.

The rule will only generate target elements for the source model elements that both conform to the matching type and fulfill the condition.
The **target pattern** specifies the elements to be generated when the source pattern of the rule is matched.

The target pattern of a rule specifies a distinct target pattern element (here t) for each target model element the rule has to generate.

A target pattern element corresponds to a model element variable declaration associated with its corresponding set of initialization bindings.

The model element variable declaration has to correspond to an entity of the target metamodel of the transformation.
ATL Helpers

**Helper context** Families!Member **def:** isFemale() : Boolean =

if not self.familyMother.oclIsUndefined() then
  true
else
  if not self.familyDaughter.oclIsUndefined() then
    true
  else
    false
  endif
endif;

- Helpers are used for defining helper methods to be called in transformation rules
- Helpers are usually defined in the context of a model element and can then be called on such a model element in a transformation rule
ATL Matched Rules and Called Rules

- **Matched rules** are declarative part of ATL
- Guideline:
  - *Source model elements should only be matched by one matched rule*

- **Called rules** are imperative part of ATL
- Can be seen as particular type of helper
- No source pattern, optional “to” section for generating target model elements
An Example of a Called Rule

```java
rule NewMalePerson (fn: String) {
    to
    p : Persons!Male (fullName <- ' ')
    do {
        p.fullName <- ' '; 
    }
}
```

- **Called rule** for creating new male Persons model elements
- The imperative “do” part is executed after the “to” part
Structure of an ATL Program

- An **ATL program** is specified as a module
- Keyword **module** introduces the name of the module
- Keyword **create** declares the target models created
- Keyword **from** declares the source models
- Optional import section
- Helpers and rules follow the header

``` ATL
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: isFemale() : Boolean = if endif;

helper context Families!Member def: familyName() : String = if endif;

rule Member2Male {
}
```
ATL Module Execution Semantics

- **Module initialization phase**
  - Initialization of attributes defined in the context of the module
  - Also include attributes defined in the context of source model elements
  - Execution of an (optional) entry point rule after attribute initialization

- **Matching phase of source model elements**
  - Testing of matching conditions of declared matched rules, in case of matching, target model elements are allocated (not yet initialized)

- **Target model initialization phase**
  - Initialization of previously allocated target model elements, execution of bindings associated with target model elements

- **Execution of imperative code sections**
Developing Model Transformations using ATL
Developing ATL Model Transformations

- ATL development environment integrated into Eclipse platform
- ATL transformation engine and ATL model management
- Support of EMF Ecore models and other formats
- Execution of ATL Transformations in an Eclipse launch configuration
ATL Development Environment in Eclipse
Defining a New ATL Configuration
The Console Output Messages about Execution

```plaintext
stack: OciUndfined
locals: self=Families2Persons : ASHModule, value='Kelly Sailor'
__resolve__:15 pop
stack:
locals: self=Families2Persons : ASHModule, value='Kelly Sailor'
__resolve__:16 load 1
stack: 'Kelly Sailor'
locals: self=Families2Persons : ASHModule, value='Kelly Sailor'
__resolve__:17 goto 10
stack: 'Kelly Sailor'
locals:
stack: OUT!<unnamed>, OUT!<unnamed>, 'Kelly Sailor'
locals: self=Families2Persons : ASHModule, link=TransientLink {rule = Member2Female, sourceElem
__applyMember2Female:19 set fullName
stack: OUT!<unnamed>
locals: self=Families2Persons : ASHModule, link=TransientLink {rule = Member2Female, sourceElem
__applyMember2Female:20 pop
stack:
locals:
stack:
locals: self=Families2Persons : ASHModule
__exec__:19 enditerate
stack:
locals:
stack:
locals:
Families2Persons executed in 0.61s.
```
Language Concepts of ATL
The ATL Transformation Language

- The ATL Language consists of three different kinds of ATL units
  - ATL transformation modules
  - ATL queries
  - ATL libraries

- Each unit can be composed of
  - ATL matched and called rules
  - ATL helpers
  - ATL attributes

- The ATL Language comprises data types and expressions
  - Based on the Object Constraint Language (OCL)
ATL Queries and ATL Libraries

- **ATL queries** compute primitive values from a set of source models
- ATL queries are often used for generating textual output

```query
PersonNb = Persons!Person.allInstances()
    ->size().toString().writeTo('result.txt');
```

- **ATL libraries** enable to define a set of helpers to be called from different ATL modules
- ATL modules may import ATL libraries and then use the helpers
ATL data types are close to OCL data types
Primitive Data Types

- ATL implements OCL primitive data types
- Boolean, Integer, Real, String data types are supported, together with appropriate operations

Examples:
- ‘test’.toUpperCase() evaluates to ‘TEST’
- 23 div 2 evaluates to 11
- true or false evaluates to true
Collection Data Types

- ATL implements OCL collection data types
  - Compare Lecture on Object Constraint Language

- Set, OrderedSet, Bag, Sequence

- Operations on collections such as
  - size(), includes(o: oclAny), excludes(o: oclAny), isEmpty(), notEmpty()

Examples:

- Set{1,2,4}->includes(1) evaluates to true
- OrderedSet{1,2,3}->first() evaluates to 1
Selected OclType and OclAny Operations in ATL

- Each ATL expression is defined in the context of an instance of a type
- The reserved keyword *self* refers to this instance

- `self.oclIsUndefined()`
  - returns a boolean stating whether self is undefined

- `self.oclIsKindOf(t: oclType)`
  - returns a boolean stating whether self is an instance of t or of one of its subtypes

- `self.oclIsTypeOf(t: oclType)`
  - returns a boolean stating whether self is an instance of t

- `self.allInstances()`
  - returns a set containing all currently existing instances of type self

- `self.allInstancesFrom(metamodel : String)`
  - returns a set containing all currently existing instances of type self in the model identified by metamodel
ATL Example with ATL Expressions

module ClassToRelation;
create OUT : Relational from IN : Class;

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

rule Class2Table {
    from
    c : Class!Class
to
    out : Relational!Table {
        name <- c.name,
        col <- Sequence {key} -> union(c.attr->select(e | not e.multiValued)),
        key <- Set {key} ),
        key : Relational!Column (name <- 'objectId', type <- thisModule.objectIdType )
    }
}
Class Metamodel and Relational Metamodel
Creating References to other Model Elements

- Three cases:
  1. assigning to a reference a target model element generated by the current rule
  2. assigning to a reference the default target model element of another rule
  3. assigning to a reference a non-default target model element of another rule

- In this example:
  - “col” reference in Table is assigned the value of t1

```plaintext
module SimpleClassToTable;
create OUT : MM1 from IN : MM;

rule Class2Table {
  from
c: MM!Class
to
t1: MM1!Column (
    name <- c.name + '_Column'),
t2: MM1!Table (
    name <- c.name,
    col <- t1,
    key <- t1
  )
}
```

Reference to table object
Summary and References
Summary of Lecture and References

- Model transformations represent a key ingredient for model-driven software engineering
- Model transformations can have different characteristics
- Model transformations can be defined in ATL in the Eclipse environment

References: