Model-Driven Software Engineering

Model Transformations II

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

- Model Transformations using QVT
- A Formalism for Model Transformations
- Important Properties of Model Transformations
- Model Transformations using Java
- Summary and References
Model Transformations using QVT
The QVT Standard

- **MOF 2.0 QVT** is an OMG standard for specifying model-to-model transformations
- Currently version 1.1
- Contains a **declarative** and an **operational part** for defining model transformations
- Q=Query, V=View, T=Transformation
- Based on MOF, defined on the basis of meta models
- Contains a graphical notation
Language Architecture of QVT

- Operational and Relations are mapped to the core language
- Core language can be considered as similar to Java bytecode
- Operational and relational part can be used together
**QVT Relations Example**

- A relation declares constraints that must hold between models.
- A relation is defined by two or more domains.
- A domain is a typed variable that can be matched in a model of a given type.
- In the example, two domains are declared.
- In the example, two patterns are declared, `p:Package` and `s:Schema`.
- The relation declares the constraint that both should have the same name.

```qvt
relation PackageToSchema {
    domain uml p:Package {name=pn}
    domain rdbms s:Schema {name=pn}
}
```

Source: [OMG QVT]
QVT Relations Example

- **When clause** specifies conditions under which the relation has to hold
- **Where clause** specifies the conditions which must hold for all participants in the relation

```
relation ClassToTable /* map each persistent class to a table */
{
  domain uml c:Class {
    namespace = p:Package {},
    kind='Persistent',
    name=cn
  }
  domain rdbms t:Table {
    schema = s:Schema {},
    name=cn,
    column = cl:Column {
      name=cn+'_tid', type='NUMBER'},
    primaryKey = k:PrimaryKey {name=cn+'__pk',
      column=cl}
  }
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

Source: [OMG QVT]
structuring relations in qvt

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {  
  top relation PackageToSchema {...}  
  top relation ClassToTable {...}  
  relation AttributeToColumn {...}  
}

- Relations may be structured using so-called top-level relations  
- Top-level relations are always required to hold  
- Non top-level relations are only required to hold if they are called  

Source: [OMG QVT]
QVT Relations – Execution Modes

- In QVT Relations, a transformation is specified as a **set of relations**
- A transformation between candidate models is called successful if the relations hold
- A transformation can be invoked in **different execution modes**
  - “Check-only” Mode
  - “Enforcement” Mode which modifies one of the models
- In the **Check-only** Mode, the transformation simply checks whether the relations hold
- In the **Enforcement** Mode, the transformation tries to modify one model in order to achieve consistency
  - “The execution of the transformation proceeds by first checking whether the relations hold, and for relations for which the check fails, attempting to make the relations hold by creating, deleting or modifying only the target model, thus enforcing the relationship”.
QVT Relations Pattern Matching

\[
\text{domain uml c:Class \{ namespace = p:Package \},}
\begin{align*}
\text{kind} &= \text{'Persistent'}, \\
\text{name} &= \text{cn}
\end{align*}
\]

- **Pattern matching** is used to bind/select a set of objects fulfilling the conditions defined in the object template expression.
- Some of the variables might already be bound (here p).
- The matching filters all of the objects of type Class in the "uml" model, any Class with its "kind" property not set to 'Persistent' is eliminated.
- Free variables (cn) are then bound depending on the value of the classes not filtered out.

Source: [OMG QVT]
Object Creation using Patterns in QVT

```java
domain rdbms t:Table {
    schema = s:Schema {},
    name=cn,
    column = cl:Column { name=cn+'_tid', type='NUMBER'},
    primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}
}
```

- **Object template expressions** are also used for object creation
  - If for a valid match in a source domain, there does not exist a valid match in the target domain

- A Table object is created with the values of the attributes set as specified (schema = s:Schema{},…)

- To avoid duplication of object creation, the concept of a key exists which is used to uniquely identify an object

Source: [OMG QVT]
Informal Semantics of Executing QVT Relations

“Check-only” Mode in the direction of target domain k:

- “For each valid binding of variables of the when clause and variables of domains other than the target domain k, that satisfy the when condition and source domain patterns and conditions, there must exist a valid binding of the remaining unbound variables of domain k that satisfies domain k's pattern and where condition.” (QVT standard, page 20)

- “Check-only” Mode allows to check whether a relation holds
Informal Semantics of Executing QVT Relations

“Enforcement” Mode of a relation in the direction of model k:

- For each valid binding of variables of the **when** clause and variables of domains other than the target domain k, that satisfy the **when** condition and source domain patterns and conditions, if there does not exist a valid binding of the remaining unbound variables of domain k that satisfies domain k's pattern and **where** condition, then create objects (or select and modify if they already exist) and assign properties as specified in domain k pattern.

- Whether an object is selected from the model or created afresh depends on whether the model already contains an object that matches the key property values, if any, specified in the object template.

- It is an error, if the template expression evaluation results in an assignment of a value to a property that clashes with another value set for the same property by another rule in the transformation execution, indicating an inconsistent specification. For primitive types, the values clash when they are different. An object assignment to a link of multiplicity “one” clashes if the object being assigned is different from the one that already exists.

- Also, for each valid binding of variables of domain k pattern that satisfies domain k condition, if there does not exist a valid binding of variables of the **when** clause and source domains that satisfies the **when** condition, source domain patterns and **where** condition, and at least one of the source domains is marked **checkonly** (or **enforce**, which entails check), then delete the objects bound to the variables of domain k when the following condition is satisfied: delete an object only if it is not required to exist by any other valid binding of the source domains as per the enforcement semantics (i.e. avoid delete followed by an immediate create). (QVT standard, page 20)

- “Enforcement” Mode allows to generate new model elements and change existing ones
Graphical Notation for QVT

relation UML2Rel {
    checkonly domain uml1 c:Class {name = n, attribute = a:Attribute{name = an}}
    checkonly domain r1 t:Table {name = n, column = col:Column{name = an}}
}

Source: [OMG QVT]
### Overview of Graphical Syntax (1)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td>A relation between models $m_1$ having MM1 as meta-model and $m_2$ having MM2 as meta-model. The label C/E indicates whether the domain in that direction is checkable or enforceable.</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td>An object template having type $C$ and referred to by the free variable $o$.</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td>An object template having type $C$ and a constraint that the property $a$ should take the value $val$. $val$ can be an arbitrary OCL expression.</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td>The domain in a relation.</td>
</tr>
</tbody>
</table>

Source: [OMG QVT]
Overview of Graphical Syntax (2)

- **o set**: An object template that matches a set of objects of type C.

- **{not}**: A template that matches only when there is no object of type C satisfying the constraints associated with it.

- **ocl expr.**: A constraint that may be attached to either a domain or an object template.

Source: [OMG QVT]
Benefits of QVT and Status of QVT Tool Support

- QVT aims at realizing bidirectional model transformations in a standardized transformation language
- Bidirectional model transformations are important for model-driven software engineering
  - If they are developed without a transformation engine, there is a lot of overhead
- Unfortunately, tool support for QVT is still limited
- Some tools implement support for QVT-Relations
  - MediniQVT, EclipseM2M (not released), ModelMorf
- Some tools implement support for QVT-Operational
  - Borland Together, SmartQVT
A Formalism for Model Transformations
Motivation for Formalizing Model Transformations

- Model transformations transform models where models can be considered as graphs
- Model transformations can be formalized using the concept of graph transformation
- Graph transformation is a formalism for model transformations for
  – abstracting from the implementation language (e.g. ATL, Java)
- Formalization can be used to study properties of model transformations
  – Confluence
  – Termination
Each meta model can be represented as an attributed type graph
Each model can be viewed as an attributed instance graph typed over the attributed type graph
A graph transformation rule \( p : L \rightarrow R \) is a structure preserving, partial mapping between two graphs
- \( L \) and \( R \) are two directed, typed and attributed graphs themselves
- Structure preserving, because vertices, edges and values may be preserved
- Partial, because vertices and edges may be added/deleted

Example: Loading of Container onto a Truck

[Slide from G. Kappel]
Graph transformation rule

- Structure preserving
  - All vertices and edges which are contained in the set $L \cap R$

- Example
Graph transformation rule

- **Adding**
  - All vertices and edges which are contained in the set \( R \setminus L \)

- **Example**

  ![Graph transformation example](image)

  [Slide from G. Kappel]
Graph transformation rule

- ** Deleting
  - All vertices and edges which are contained in the set \(L \setminus R\)

- ** Example**

[Slide from G. Kappel]
Graph transformation rule

- Calculating – Make use of attributes
  - Constants
  - Variables
  - Expressions (OCL & Co)

Example

```
1 : Container
weight = x

2 : Truck
load = 0

3 : Store
containers = y

4 : inFrontOf
```

```
1 : Container
weight = x

2 : Truck
load = x

3 : Store
containers = y - 1

4 : inFrontOf
```

[Slide from G. Kappel]
Graph transformation

- A graph transformation $t: G \rightarrow H$ is the result of the execution of a graph transformation rule $p: L \rightarrow R$ in the context of $G$
  - $t = (p, m)$ where $m: L \rightarrow G$ is an injective graph morphism (match)

[Slide from G. Kappel]
Graph transformation

- Prepare the transformation
  - Select rule \( p: L \rightarrow R \)
  - Select match \( m: L \rightarrow G \)

- Generate new graph \( H \) by
  - Deletion of \( L \setminus R \)
  - Addition of \( R \setminus L \)
Graph transformation example (1/2)

L

1 : Container
weight = x

2 : Truck
load = 0

3 : Store
containers = y

in

4:inFrontOf

R

1 : Container
weight = x

2 : Truck
load = x

3 : Store
containers = y-1

on

4:inFrontOf

m

G

1 : ShippingCompany
name = „LKW Maier“

1 : Truck
load = 0

2 : Truck
load = 100

1 : Container
weight = 10

1 : Store
name = „CompStore“
containers = 1

H

1 : ShippingCompany
name = „LKW Maier“

1 : Truck
load = 10

2 : Truck
load = 100

1 : Container
weight = 10

1 : Store
name = „CompStore“
containers = 0
Graph transformation example (2/2)

L:
- a: Container
  - weight = x
- b: Store
  - containers = y

G:
- 1: ShippingCompany
  - name = "LKW Maier"
- 1: Truck
  - load = 0
- 2: Truck
  - load = 100
- 1: Container
  - weight = 10
- 1: Store
  - name = "CompStore"
  - containers = 1

R:
- a: Container
  - weight = x
- c: Truck
  - load = x
- b: Store
  - containers = y - 1

H:
- 1: ShippingCompany
  - name = "LKW Maier"
- 1: Truck
  - load = 10
- 2: Truck
  - load = 100
- 1: Container
  - weight = 10
- 1: Store
  - name = "CompStore"
  - containers = 0
Modeling Transformations using Graph Transformation

- A model transformation can be formalized using graph transformation
- Each model transformation rule can be viewed as a graph transformation rule
- The theory of graph transformation can be applied to the domain of model transformation
- A graph transformation can be seen as a visual specification of the model transformation
- Implementation of a model transformation can be done using different implementation languages (ATL, QVT, Java)
Graph transformations vs. ATL

- Graph transformation

![Graph transformation diagram]

- ATL Program (Refining Mode)

```
rule Loading {
    from
        a : Container (a.in = b)
        b : Store
        c : Truck (c.inFrontOf = b AND c.load = 0)
    to
        _a : Container(weight <- a.weight, on <- _c)
        _b : Store(containers <- b.containers - 1)
        _c : Truck(load <- a.weight, inFrontOf <- _b)
}
```

[Slide from G. Kappel]
Important Properties of Model Transformations
Important Properties of Model Transformations

**Syntactic correctness:**
- The model transformation always produces a valid output model with regards to the syntax of the output language.
- The model transformation rules conform to their syntax.

**Semantic correctness:**
- The model transformation preserves the semantics of the source language and transforms a source model into a semantically equivalent target model.
- Only applicable for a special class of model transformations
Important Properties of Model Transformations

Confluence:

- If applied to a model, the model transformation always produces the same output model.
- Important if several transformation rules can be applied at the same time.

Termination:

- The model transformation terminates.
- Not so relevant if transformation languages such as ATL, QVT, Java are applied.
Example of a Syntactic Correctness Problem

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

- **Syntactic correctness comprises:**
  - Correctness with regards to the metamodel of the target language
  - Fulfillment of OCL constraints of the target language

- **ATL detects this at runtime (only partially at compile time)**
Example of a Confluence Problem of a Model Transformation

\[
\text{rule Member2Male2} \{
\text{from} \\
\quad s: \text{Families!Member} (\text{not} s.\text{isFemale}()) \\
\text{to} \\
\quad t: \text{Persons!Child} \\
\quad \quad \text{fullName} \leftarrow s.\text{firstName} + \ ' ' + s.\text{firstName}() \\
\}
\]

- Addition of another rule as above causes a confluence problem
- Sometimes these problems are very difficult to detect
Model Transformations using Java
Motivation for Model Transformations in Java

- In some occasions, model transformations cannot be designed and implemented using a model transformation language
  - Can be due to a customer requirement that does not allow open source or third party tools
  - Can be due to other requirements that require the explicit use of a programming language such as Java for implementation

- If model transformations are implemented in Java, the software engineer has to take care of model management that is obtained for granted in model transformation environments
  - Loading and saving of models
  - Deletion and creation of model elements

- Advanced concepts such as bidirectionality are difficult to obtain
Model Transformations in Java – Software Quality

- Model transformations in Java require the software engineer to ensure the quality of the transformation implementation
  - This is also the case if a model transformation language is used, but for Java this is even more important

- The architecture of the model transformation solution has to be designed by the software engineer

- Maintainability, reliability and functionality are very important requirements

- Typical measure known from software engineering can be applied to increase software quality
Problem:
- How do you structure the responsibilities of a model transformation or model transformation chain in such a way that it becomes easy to exchange transformations, change the order of executing transformations and reuse transformations on other models?

Solution:
- Model represents the model(s) to be transformed
- Transformation contains the transformation logic (transformation rules)
- Controller manages creation of the model(s) and calls transformations
Model-Transformation-Controller Pattern - Description

Forces:

- The rationale behind this pattern is to separate transformation, model and controller functionalities to increase maintainability, in particular changeability and testability.

- If the transformation logic is located inside one class this simplifies testing and developing the transformation and it enables transformation reuse. Concerning changeability, the transformation logic is not mixed with model-specific aspects that can make reuse impossible for other models. The model class itself encapsulates all data aspects of the model and provides the access to the model. The model class can either store all data relevant to the model or it can also act as a delegation point to the physical model.

- Separating the model from the transformation itself has several advantages: The transformation logic inside the transformation can be changed and the transformation can even be replaced completely. If the transformation class is abstract different implementations can be designed and easily exchanged. If the model class is abstract and part of an inheritance hierarchy then the transformation can be applied on a family of models that share the same properties.

- Separating model and transformation from the controller is important for several reasons: The design of controllers with different qualities of service becomes possible in this way. For example, one controller may include explicit error handling (see below) and another controller may not include error handling. In a model transformation chain, different controllers can be developed which differ in the order of calling transformations.
Summary and References
Summary of Lecture and References

- Model transformations represent a key ingredient for model-driven software engineering
- Model transformations can be defined in QVT or Java
- Formalization of model transformations using graph transformation in order to define properties such as confluence and termination

References:

Background: Attributed Graphs
Recap: Directed graph

- A directed graph $G$ consists of two disjoint sets
  - Vertices $V$ (Vertex)
  - Edges $E$ (Edge)

- Each edge has a source vertex $s$ and a target vertex $t$

- Summarized: $G = (V, E, s: E \rightarrow V, t: E \rightarrow V)$

- Example graph
  - $V = \{1, 2, 3\}$
  - $E = \{a, b, c\}$
  - $s(a) = 2$
  - $t(a) = 1$
  - $s(b) = 3$
  - $t(b) = 2$
  - $\ldots$

[Slide from G. Kappel]
Recap: Attributed Graphs

- An attributed graph $AG$ where only graph vertices can be attributed is a pair consisting of a directed unlabelled graph $G$ and a $\Sigma$-Algebra $A$ such that $|A| \subseteq GV$.

- The elements of $|A|$ represent potential attribute values which are regarded as special data vertices of the graph (besides the object vertices that model structural entities).

- An object vertex $v \in GV$ has an attribute value $a \in |A|$ if there is an edge from $v$ to $a$ in $AG$. 
Recap: Attributed Type and Instance Graph

- An attributed type graph $ATG$ is an attributed graph where $A$ is the final $\Sigma$-Algebra having $As = \{s\}$ for all $s \in S$.
- An attributed instance graph is an attributed graph with an additional typing morphism which specifies the type of all vertices of the graph.