QVT: Query, Views, Transformations

Rubby Casallas
Grupo de Construcción de Software
Uniandes

Basics

- Transformations are **essential** for the MDE
- A model transformation:
  - takes as input a model conforming to a given metamodel
  - produces as output another model conforming to a given metamodel.
**QVT Basics**

- An OMG Standard:
  - addresses the need for standardizing the way mappings are achieved between models whose languages are defined using MOF.
- Query/View/Transformation Specification
- Final Adopted: July 2007

---

**Q-V-T**

- **Query:**
  - on MOF models
  - requires to filter and select elements from a model
- **View:**
  - is a model that is derived from another model.
  - a mechanism for creating views
- **Transformation:**
  - standardize transformation rules
QVT specification

- It depends on:
  - MOF 2.0 Specification
  - OCL 2.0 Specification
- It has a hybrid declarative/imperative nature

---

QVT specification

- Three related transformation languages:
  - Core
  - Relations: has a graphical concrete syntax
  - Operational Mappings: is an imperative language that extends both QVT-Relations and QVT-Core
Architecture (declarative)

- Core:
  - supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models.
  - All trace classes are explicitly defined as MOF models.
  - trace instance creation and deletion is defined in the same way as the creation and deletion of any other object.
Architecture (declarative)

- Relations:
  - support complex object pattern matching and object template creation.
  - support traces between model elements involved in a transformation (they are created implicitly)

Operational Mappings Language

- OCL extensions with side effects that allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers.
Operational Mappings Language (cont.)

- Mappings Operations can be used to implement one or more Relations from a Relations specification when it is difficult to provide a purely declarative specification of how a Relation is to be populated.
- A transformation entirely written using Mapping Operations a called an operational transformation.

QVT Tools

- Smart QVT:
  - An open source model transformation tool implementing the MOF 2.0 QVT-Operational language
  - This tool is being developed by France Telecom R&D
  - Eclipse-Plugin
  - http://smartqvt.elibel.tm.fr/index.html
QVT-Operational language
(http://smartqvt.elibel.tm.fr/doc/index.html)

- Program structure
- Operator
- Basic types
- Variables
- Object creation
- Getting input model elements
- Most common control blocks
- QVT specific methods

EMOF
EMOF

Example: Book To Publication

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>1</td>
</tr>
<tr>
<td>Is Composite</td>
<td>true</td>
</tr>
<tr>
<td>Is Derived</td>
<td>false</td>
</tr>
<tr>
<td>Is ID</td>
<td>false</td>
</tr>
<tr>
<td>Is Ordered</td>
<td>true</td>
</tr>
<tr>
<td>Is Read Only</td>
<td>false</td>
</tr>
<tr>
<td>Is Unique</td>
<td>true</td>
</tr>
<tr>
<td>Lower</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>chapters</td>
</tr>
<tr>
<td>Opposite</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Class Chapter</td>
</tr>
<tr>
<td>Upper</td>
<td>1</td>
</tr>
</tbody>
</table>
transformation Book2Publication(in bookModel:BOOKS, out pubModel:PUB);

main() {
  bookModel.objects()[Book]->map book_to_publication();
}

mapping Book::book_to_publication () : Publication {
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}

Example: SimpleUML To Rdb
Example: Simpleuml To Rdb

simpleRDBms metamodel
Program structure

transformation Simpleuml_To_Rdb( in srcModel: UML, out dest: RDB);

main() {
  srcModel.objects()[Class]->map class2table(); -- first pass
  srcModel.objects()[Association]->map asso2table(); -- second pass
}

main entry point of the transformation
transformation Simpleuml_To_Rdb( in srcModel: UML, out dest: RDB);

main() {
    srcModel.objects()[Class]->map class2table(); -- first pass
    srcModel.objects()[Association]->map asso2table(); -- second pass
}

returns all objects in the input model which are conform to Class
shorthand for:

select(e | e.oclIsKindOf(UML::Class))
The Map operator

myClass.map Class2Column();

- applies a mapping operation named **Class2Column** on a **Class** element named myClass.

---

The Map operator

- **The "->" operator**
  - Each element of a collection can be acceded by the " -> " operator.
  - It can be seen like an iterate operator

- **myClass.attributes->map attribute2table();**

- **For each attribute the mapping named attribute2table is applied.**
Mapping

A mapping is an operation associating an element from a model with another element often from another model.

```cpp
mapping Class::class2table () : Table
when {self.kind='persistent';}
{  
  init { -- performs any needed initialization
    self.leafAttributes :=
      self.attribute->map attr2LeafAttrs("","")->flatten();
  }
}

-- population section for the table
name := 't_' + self.name;
column := self.leafAttributes->map leafAttr2OrdinaryColumn("")->asOrderedSet();
key := object Key { -- nested population section for a 'Key'
  name := 'k_' + self.name;
  column := result.column{kind='primary'};
};
```

Mapping Structure

```cpp
mapping <Type>::<mappingsName> (<kind> <paramName> : <paramType>) : <ReturnType>
when { <boolean-expression> }
where { <boolean-expression> }
{
  init {
    <init-instructions>
  }
  <main-instructions>
  end {
    <end-instructions>
  }
}
```
## Mapping Structure

```java
mapping <Type>::<mappingName> (<kind> <paramName> : <paramType>) : <ReturnType>

when { <boolean-expression> }

• The when clause can be used to define the guard of a mapping operation.
• A guard is a Boolean expression that can be seen as a pre-condition.
• When this guard is equal to the true value, the mapping is applied
• when the guard is true otherwise the mapping fails and returns null.

mapping Class::class2table () : Table
  when {
    self.kind='persistent';
  }
```

## Mapping Structure

```java
mapping <Type>::<mappingName> (<kind> <paramName> : <paramType>) : <ReturnType>

where { <boolean-expression> }

• The where clause can be used to define a post-condition for a mapping operation.
• It is composed of boolean expressions.
• If the expression is false, an exception is raised and the program is automatically interrupted.
```
Mapping Structure

\[
\text{mapping}\ \langle\text{Type}\rangle::\langle\text{mappingName}\rangle\ (\langle\text{kind}\rangle\ \langle\text{paramName}\rangle\ :\ \langle\text{paramType}\rangle)\ :\ \langle\text{ReturnType}\rangle
\]

\[
\text{when}\ \{\ \langle\text{boolean-expression}\rangle\ \}\n\]

\[
\text{where}\ \{\ \langle\text{boolean-expression}\rangle\ \}\n\]

\[
\begin{align*}
\text{init} & \{ \\
& \langle\text{init-instructions}\rangle \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{<main-instructions>}
\end{align*}
\]

\[
\begin{align*}
\text{end} & \{ \\
& \langle\text{end-instructions}\rangle \\
& \}
\end{align*}
\]

Mapping operation body
- variable assignments; keeps intermediate results
- uses query, mapping and resolve calls
- explicit \texttt{out} parameter assignment

---

Mapping Structure

\[
\text{mapping}\ \langle\text{Type}\rangle::\langle\text{mappingName}\rangle\ (\langle\text{paramName}\rangle)\ :\ \langle\text{ReturnType}\rangle
\]

\[
\text{init} \{ \\
\langle\text{init-instructions}\rangle \\
\}
\]

\[
\langle\text{main-instructions}\rangle
\]

\[
\text{end} \{ \\
\langle\text{end-instructions}\rangle \\
\}
\]

From: Model Transformation with Operational QVT | Long Talk, EclipseCon 2008 | © 2008 by Borland Software Corp. | Made available under the EPL v1.0
Mapping Structure: The init clause

- The init clause has for objective to execute some code before the initialisation of output elements

```plaintext
mapping Class::class2table () : Table
  when {self.kind='persistent';}
  {
    init { -- performs any needed intialization
      self.leafAttributes :=
        self.attribute->map attr2LeafAttrs("",""|)->flatten();
    }
    ...
  }
```

Mapping Structure: The end clause

- The end clause can be used to realise operations before the mapping return its return value.
Mapping operation body – object population

-- The attribute2column mapping can be used to -- transform an Attribute into a Column.

mapping Attribute::attribute2column(): Column {
    type := self.type.name;
    name := self.name;
    kind := self.kind;
}

Mapping operation body – object population (cont.)

mapping Class::class2table () : Table
    when {self.kind='persistent';}

    -- population section for the table
    name := 't_' + self.name;
    column := self.attributes->map attr2Column();
    key := object Key { -- nested population section for a 'Key'
        name := 'k_' + self.name;
        column := result.column[kind='primary'];
    };

• name is equal to the string "t_" to which is appended the name of the class.
• column corresponds to the result of the mapping attribute2column on class attributes.
• key _ is a new created element.
Mapping operation body – object population (cont.)

```java
mapping Class::class2table () : Table
when {self.kind='persistent';}

-- population section for the table
name := 't_' + self.name;
column := self.attributes->map attr2Column();
key_ := object Key { -- nested population section for a 'Key'
  name := 'k.' + self.name;
  column := result.column[kind='primary'];
};

-- the Table element currently produced by
-- the mapping

result.column contains all Column
elements
which have just been created by the
attribute2column mapping.
```

Mapping operation body – object population

```java
mapping Class::class2table () : Table
when {self.kind='persistent';}

init { -- performs any needed initialization
  self.leafAttributes :=
    self.attribute->map attr2LeafAttr("","")->flatten();
}

-- population section for the table
name := 't_' + self.name;
column := self.leafAttributes->map leafAttr2OrdinaryColumn("")->asOrderedSet();
key := object Key { -- nested population section for a 'Key'
  name := 'k.' + self.name;
  column := result.column[kind='primary'];
};
```
Operator

- Comments
  // It is a single line comment
  -- it is another single line comment
  /* It is a multi line comment */
- Basic operators
- QVT operators

Basic operators

- Sequentiality
  <instruction1> ; <instruction2>

- Affectation
  myVar := true;

- Equality
  myVar = true;
  myVar == true;
Basic operators (cont.)

- **Different**
  
  ```
  myVar != true; myVar <> true;
  ```

- **String concatenation**
  
  ```
  "a string" + " another string";
  ```

- **Adding in a list**
  
  ```
  myList += aElement;
  ```

---

QVT operators

- **The "." operator**
  - The "." operator can be applied on an object, to call attributes or operations of this object.
  - This meaning is the same as in Java language.

  ```
  myColumn.operationOnColumn();
  myColumn.type := "The type of the column";
  ```
Basic types

- Boolean type
- String type
- Collections

<table>
<thead>
<tr>
<th></th>
<th>Unique</th>
<th>Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>OrderedSet</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Sequence</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Bag</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Variables

- Typing:
  - QVT variables are typed explicitly during initialisation or implicitly at their first affectation with a value from a precise type.
- Declaration
  - Variables only exist in methods and always have a local visibility
Variables (declaration)

-- Typed and initialised variable
var myStringVar : String := "an explicitly";

-- Typed and initialised variable
var varNoInitializedAndExplicitelyTyped : Column;

-- Not typed but initialised variable
var anotherStringVar := "an implicitly ";

-- Not typed or initialised variable
var varNoInitializedAndNoTyped;

Predefined Variables

- **this**
  - The `this` variable represents the transformation containing this instruction. Its type is Transformation

- **self**
  - The `self` variable refers to the object on which the method is applied.

```plaintext
mapping Class::class2table() : Table {
    var myString := self.name;
    -- Other instructions
}
```
Predefined Variables

- **result**
  - The `result` variable exists only in a mapping operation type.
  - It represents the object returned by the mapping.
  - It is implicit at the left of an operator but it is necessary to use it at the right of an operator or in a block instruction.

```java
mapping Class::class2table() : Table {
  name := self.name;
  -- Some instructions
  var myString := result.name;
  -- Other instructions
}
```

Object creation

- Mapping operation creates an object when it is called
- "object" key-word:

```java
var myObject := object Attribute {
  name := "attribute name";
  kind := "attribute kind";
  type := null;
};

object myObject : Attribute { name := "attribute name"; kind := "attribute kind"; }
```
Getting input model elements

- `objectsOfTypes` method:
  - returns all element models which correspond of the given type.

```java
main() {
    srcModel.objectsOfTypes(Class)->map
    class2table();

    srcModel.objectsOfTypes(Association)->map
    asso2table();
}
```
Most common control blocks

- Conditional block

```pascal
var value1 := true;
var value2 := false;
var myString:String;
myString := if( value1) then
  "value1 is equal at true"
elif(value2) then
  "value2 is equal at true"
else null
endif;
```

Most common control blocks

- Collection operations: **The select operation**
  - filters collection elements
  - takes one or two parameters which have to be defined:
    - The first (optional) is the name of the collection element which is used by the select operation.
    - The second (mandatory) is a boolean expression filtering the elements.
Most common control blocks

- Collection operations: **The select operation**
  - Example

    ```
    myClass.attributes-> select(kind='persitent').map attr2table();
    myClass.attributes-> select(a:Attribute | a.kind='persitent').map attr2table();
    myClass.attributes[kind='persitent'].map attr2table();
    ```

Most common control blocks

- Collection operations: **The collect operation**
  - The *collect* operation returns a collection derived from another collection.

    ```
    collectionOfName := collectionOfClass->collect(name);
    ```

    returns a String collection from a Class collection because a Class has a name attribute
QVT specific methods

- query and helper:
  - a query can not cause side effects, i.e. it can not modify the system state.

```java
query Association::isPersistent() : Boolean =
(self.source.kind='persistent' and
 self.destination.kind='persistent');
```

```java
query Association::isPersistent() : Boolean {
 return (self.source.kind='persistent' and
 self.destination.kind='persistent')
}
```
QVT specific methods: helper

```java
helper Class::class2seqCol() : OrderedSet(Column) {
    var t := self.attributes->map Attribute2Column();
    return t;
}
```