The Art of Model Transformation with Operational QVT

Sergey Boyko
Radomil Dvorak
Alexander Igdalov

Borland Software Corporation
QVTO Key Concepts

Operational QVT (QVTO)

• operates with EMF models
• uses OCL for model navigation
• Main goal - model modification and transformation
• required an explicit and complete algorithm model-to-model mapping
QVTO structure

- **QVTOperational package** – general structuring elements and top-level constructions
- **ImperativeOCL package** – extension to OCL expressions and type system
- **Standard Library**
QVT Operational package

- Transformation declaration
- Imperative operations (mappings, helpers, queries, constructors)
- Intermediate data
- Object creation and update mechanism
- Trace resolution expressions
Operational Transformation 1

A simple transformation example

modeltype ECORE uses
  'http://www.eclipse.org/emf/2002/Ecore';
modeltype UML uses
  'http://www.eclipse.org/uml2/2.0.0/UML';

transformation Ecore2Uml(
  in inModel : ECORE, out outModel : UML);

main()
{
  inModel.rootObjects()[EPackage]->map
ePackage2Model();
}

mapping EPackage::ePackage2Model() : Model{
  name := self.name;
}
Operational Transformation 2

The content of the transformation definition may be placed within the transformation element:

```plaintext
modeltype ECORE uses 'http://www.eclipse.org/emf/2002/Ecore';
modeltype UML uses 'http://www.eclipse.org/uml2/2.0.0/UML';

transformation Ecore2Uml(in inModel : ECORE, out outModel : UML) {
    main() {
        inModel.rootObjects()[EPackage]->map ePackage2Model();
    }

    mapping EPackage::ePackage2Model() : Model {
        name := self.name;
    }
}```
Imperative Operations

- define an imperative body
- enriched signature

Types of QVTO imperative operations
  - Entry operation
  - Mappings
  - Helpers
  - Queries
  - Constructors
Entry Operation

An entry operation is the entry point for the execution of a transformation.

```java
main() {
    inModel.rootObjects()[EPackage]->map ePackage2Model();
}
```

Typically refers to model parameters and invokes top-level mappings.
Helpers and Queries

A *helper* is an operation that performs a computation on one or more source objects and provides a result. It is illegal to create or update object instances except for pre-defined types like sets, tuples, and for intermediate properties.

```csharp
helper EPackage::someHelper1() : Set(String) {
    if (self.name = 'A') then {
        return Set {'B'};
    } endif;
    return Set {self.name};
}
```

A *query* is a “read-only” helper which is not allowed to create or update any objects.

```csharp
query EPackage::getNameAtoB() : String {
    if (self.name = 'A') then {
        return 'B';
    } endif;
    return self.name;
}
```

```csharp
helper EPackage::someHelper2() : Set(String) = Set{self.name};
query EPackage::getName() : String = self.name;
```
Constructors

A constructor is an operation that defines how to create and populate the properties of an instance of a given class.

```java
constructor EClass::EClass(s : String, op : EOperation) {
    name := s;
    eOperations += op;
}
```

Calling the constructor:

```java
new EClass("AClass", new EOperation());
```
Mappings

A mapping between one or more source model elements into one or more target model elements.

```xml
<mapping> ::= <qualifier>* 'mapping' <param_direction>? (<context_type>::)? <identifier>
       '(' <param_list>? ')' '('<param_list>? <mapping_extension>* <when>?
       '{' <mapping_body> '}'
```

*Most typical case:*

```java
mapping <context_classifier>::<mapping_name> ( <parameters> ) : <return_type> {
  <mapping body>
}
```

```java
mapping ECORE::EPackage::ePackage2Package() : UML::Package {
  name := self.name;
}
```

ePackage.map ePackage2Package(); // calling a mapping for a single context
ePackages->map ePackage2Package(); // calling a mapping consequently for a collection of contexts
Mapping Parameters Direction Kind

Mapping parameter direction kind

- **in** – object passed for read-only access, the default direction
- **inout** – object passed for update, retains its value
- **out** – parameter receives new value (not necessarily newly created object)
Mappings – when clause

```cpp
mapping EPackage::ePackage2Package() : Package
    when {self.name <> null} {
        name := self.name;
    }
```

WHEN-clause contains a Boolean condition

```cpp
self.name = null
(condition not satisfied)
```

Invocation:
- in standard mode
  when-clause acts as a guard which filters input parameters

```cpp
a.map ePackage2Package();
```

- in strict mode
  when-clause acts as a pre-condition which must always hold

```cpp
a.xmap ePackage2Package();
```

- mapping not executed
- null returned
- mapping not executed
- exception thrown
Mapping Body – General Form

```
mapping EPackage::myMapping() : Package {
  init {
    var tmp := self.map otherMapping();
    if (self.name = 'AAA') then {
      result := object Package {};
    } endif;
  }
  population {
    object result : Package {
      name := self.name;
    }
  }
  end {
    assert (result.name <> null);
  }
}
```

**Predefined variables in mappings:**
- `self` – refers to the context
- `result` – refers to the result
Mapping Body
Population Keyword Omitted

Omitted population keyword is the most typical case!
Overriding Mappings

Simple overriding:

```plaintext
mapping ENamedElement::makeClass(): EClass {
    name := 'NE:' + self.name;
}
overrides

mapping EClassifier::makeClass(): EClass {
    name := 'CLASSIFIER:' + self.name;
}
overrides

mapping EClass::makeClass(): EClass {
    name := 'CLASS:' + self.name;
}
overrides

mapping EDataType::makeClass(): EClass {
    name := 'DT:' + self.name;
}
```
Abstract mapping `EClassifier::makeClassifier()`:

```plaintext
EClassifier {
    name := self.name + '1';
}
```

Mapping `EClass::makeClass(): EClass`:

```plaintext
inherits EClassifier::makeClassifier {
    init {
        var tmp := '2';
    }

    name := name + tmp;
}
```

Evaluation result:
`result.name = self.name + '12'`

**Execution flow:**

- init section (of `EClass::makeClass`)
- instantiation section (of `EClass::makeClass`)
- inherited mapping(s) (`EClassifier::makeClassifier`)
- mapping population and termination sections (of `EClass::makeClass`)

Implicit instantiation
Mapping Extension - merges

```
abstract mapping EClassifier::makeClassifier(): EClassifier {
  name := name + '1';
}

mapping EClass::makeClass(): EClass
  merges EClassifier::makeClassifier {
    init {
      var tmp := '2';
    }
    name := self.name + tmp;
  }
```

Evaluation result:
result.name = self.name + '21'

Execution flow:
- merging mapping (EClass::makeClass)
- merged mapping(s) (EClassifier::makeClassifier)
Mapping Extension - disjuncts

**Execution flow:**

- when-clauses of the disjuncted mappings are evaluated
  
  \(- \text{ of } \text{EClass::makeAClass} \)
  
  \(- \text{ of } \text{EClass::makeBClass} \)

2. If all when-clauses are not satisfied \textbf{null} is returned

3. Otherwise, the first mapping with a \textbf{true} when-clause is executed

**Evaluation results:**

- object \text{EClass} \{name := ‘CClass’\}.\text{map} \text{makeClass()} = \textbf{null};
- object \text{EClass} \{name := ‘AClass’\}.\text{map} \text{makeClass().name} = ‘AClassA’;
- object \text{EClass} \{name := ‘BClass’\}.\text{map} \text{makeClass().name} = ‘BClassB’;
Traceability Concept

- Trace contains information about mapped objects
- Trace consists of trace records
- A trace record is created when a mapping is executed
- Trace records keep reference to the executed mapping and the mapping parameter values
- A trace record is created after the implicit instantiation section of the mapping is finished

**Usage:**
- Prohibit duplicate execution with the same parameters
- Used in resolve expressions
- May be serialized after the transformation execution
A **resolve expression** is an expression that inspects trace records to retrieve source or target objects which participated in the previous mapping executions.

- **resolve** – resolves target objects for a given source object
- **inv (invresolve)** – resolves source objects for a given target object
- **One (resolveOne)** – finds the first matching object
- **In (resolveIn)** – inspects trace records for a given mapping only
- **late (late resolve)** – performs resolution and assignment to some model object property after the transformation execution

```cpp
mapping EPackage::myMapping() : Package {
    ePackage.resolve(Package)
}
```

$2^4 = 16$ combinations, e.g. **invresolveOne or late invresolveOneIn**
Resolve Expressions 2

```
mapping EClassifier::c2c() : EClass {
    name := 'mapped' + self.name;
}

// somewhere in the code
var orig := object EClass { name := 'original' };
var mapped := orig.map c2c();

// in some other place
var res1 := orig.resolve(EClass);
var res2 := resolveoneIn(EClassifier::c2c, t : EClass
    | t.name.startsWith('mapped' ));
var res3 := mapped.invresolveIn(EClassifier::c2c, EClass);
```

Resolve expressions are a useful instrument of retrieving trace information!
Object Expression

An *object expression* is an *inline* instantiation facility.

```
object x:X { … } // An explicit variable here
object Y { … } // No referred variable here
object x: { … } // the type of 'x' is skipped here when already known
```

If `x` exists then it is updated, otherwise created and updated

```
object EPackage {
    name := ‘pack’;
    nsURI := ‘http://myuri.org’;
    eClassifiers += object EClass {
        name := ‘clazz’;
    }
}
```
Model Extents

A *model extent* is a container for model objects. For each model parameter there is a model extent.

```plaintext
modeltype ECORE uses 'http://www.eclipse.org/emf/2002/Ecore';

transformation transf(in m: ECORE, out x: ECORE, out y: ECORE);

main() {
    var a := object EPackage@x {
        name := 'a'
    };

    var b := object EPackage@y {
        name := 'b'
    };
}

mapping EClass::toClass() : EClass@y {
    name := self.name;
}
```

Refer to model extents with @model_parameter_name
Intermediate Properties

An *intermediate property* is a property defined as an extension of the type referred by the *context*.

- typically defined as class extensions of model metaclasses
- created temporarily by a transformation
- not a part of the output
- used for intermediate calculations associated with the instances of the extended class

```plaintext
intermediate property EClass::intermProp : String;

main() {
    object EClass {
        name := 'original';
        intermProp := 'abc'
    };
}
```
Intermediate Classes

An *intermediate class* is a class created temporarily by a transformation to perform some needed calculation but which is not part of the expected output.

```java
intermediate class MyEPackage extends EPackage {
    myName : String;
}

mapping EClassifier::c2c() : EClass {
    object MyEPackage {
        name := 'name';
        myName := 'someThoughtfulName';
    }
}
```
ImperativeOCL package

- Assignments
- Variables
- Loops (while, forEach)
- Loop interrupt constructs (break, continue)
- Conditional execution workflow
- Convenient shorthand notation
- Mutable collections
Assignments

- Assignment to variables
- Assignment to properties (including complex nested constructions)

```plaintext
mapping EClassifier::c2c() : EClass {
    name := self.name;
}

mapping EPackage::p2p() : EPackage {
    name := nsPrefix := nsURI := 'aaa';
    eClassifiers += self.eClassifiers->map c2c();
    eClassifiers += object EClass {
        name := 'A'
    };
    eSuperPackage.eSuperPackage.eSubpackages->any(true).name := 'A';
}
```
Variables in QVTO

OCL variables in **let** expression:

```ocl
class Resource {

  String id;

  let id := '123'; // simple variable declaration and initialization

  let id := id + 'A'; // variable modification

  id := id + 'B'; // variable read access

}
```

QVTO extends OCL with variable initialization expressions and assignments to variables:

```qvt
var a : String := 'A'; // full notation
var b := 'B'; // type deduced from the initialization expression
var c : String; // default value assigned
```

```qvt
mapping EPackage::p2p() : EPackage {
  var tmp := 'A' + self.name; // variable declaration and initialization
  name := tmp; // variable read access
  tmp := tmp + 'B' // variable modification
  eClassifiers += self.eClassifiers->map c2c();
  eClassifiers += object EClass {name := tmp}; // another access
}
```
While Loop

OCL iterator expressions iterate through collections and cannot be interrupted by break, continue or return statements. They are rather specific, e.g.:

```java
collection->collect( v : Type | expression-with-v )
```

**While loop** is a Java-like imperative cycle that can be interrupted by break, continue and return.

```java
mapping EPackage::p2p() : EPackage {
    var i : Integer := 0;
    while (i < 10) {
        eClassifiers += object EClass {};
        i := i + 1;
    }
}

mapping EPackage::p2p() : EPackage {
    while (i := 0; i < 10) {
        eClassifiers += object EClass {};
        i := i + 1;
    }
}
```
ForEach Loop

*ForEach* evaluates some expression(s) for each element of a given collection.

```
var abc := Sequence {'a', 'b', 'a', 'b'};
var res : String := "";
abc->forEach(i) {
    res := res + i;
};
```

```
abc->forEach(i | i = 'b') { // forEach with a condition
    res := res + i;
};
```

**forOne** – equivalent to **forEach** with a **break** statement:

```
abc->forOne(i | i = 'b') {
    res := res + i;
};
```

```
abc->forEach(i | i = 'b') {
    res := res + i;
    break;
};
```
Loop Interruption
Break and Continue

**break** and **continue** – used within while, forEach loops and imperative iterators

```java
var i : Integer := 0;
while (i < 10) {
   if (i = 3) then {
      i := i + 2;
      continue;
   } endif;
   if (i = 8) then {
      break;
   } endif;
   object EClass {}; 
   i := i + 1;
};
```
**Operation Interruption - Return**

`return` – used to interrupt imperative operations

```cpp
query EPackage::getName() : String {
    if (self.name = 'A') then {
        return 'B';
    } endif;
    if (self.name = 'B') then {
        return 'C';
    } endif;
    return self.name;
}
```
Conditional Execution

- **If-expression**

```java
if <condition> then {
    <expressions>;
} else {
    <expressions>;
} endif;
```

- **Switch-expression**

```java
var a : Collection(SomeType)
:= coll->switch(i) {
    case (cond_with_i_1) { <exprs_with_i>}
    case (cond_with_i_2) <expr_with_i>;
    else <expression_with_i>;
};
```
Imperative Iterators

New in QVTO:
• xcollect
• xselect
• collectselect
• collectOne
• selectOne
• collectselectOne

Inherited from OCL:
• break
• continue
• return
• nulls skipped

```
var coll := Sequence { object EClass { name := 'a'},
                      object EDataType { name := 'b'},
                      object EClass { name := null}):

var c1 := coll->xcollect(i | i.name); // c1 = Sequence {'a', 'b'}
var c2 := coll->collectOne(i | i.name); // c2 = 'a'
```
Shorthand Notation

Convenient shorthand notation make code concise and effective:

- list->prop; // same as list->xcollect(i | i.prop)
- list[condition]; // same as list->xselect(i; condition)
- list->prop[startsWith("_")]; // same as list->collectselect(i;res= i.prop | // res.startsWith("_")) ;
- list->prop![startsWith("_")]]; // calling collectselectOne(i;res= i.prop | // res.startsWith("_")))

QVTO shorthand snippet:

```qvt
list->prop![startsWith("_")]);
```

The same in pure OCL:

```ocl
list->collect(prop)-> select(not oclIsUndefined() and startsWith("_"))-> first();
```
Mutable Collections

OCL collections – **Sequence**, **Bag**, **Set**, **OrderedSet** are immutable:

```text
Sequence {'a', 'A', 'b'} -> select(equalsIgnoreCase('a')); // creates a new sequence
```

*New in QVTO* – mutable collections:

- **List** – mutable sequence
- **Dict** – mutable hash table

```text
var dict : Dict(String, Integer) := Dict { 'key1' = 5 };
dict->put('key2', 10);
var i : Integer := dict->get('key1') * dict->get('key2'); // i = 50
```
Standard Library

- **Element operations**
  - `subobjects` // all immediate sub objects of an object (in terms of containment)
  - `deepclone`

- **Model operations**
  - `rootObjects`
  - `copy` // full copy of a model

- **String routines**
  - `startsWith`
  - `indexOf`

- **Mutable collection routines**
  - `List::add`
  - `Dict::get`

- **Transformation execution routines**
  - `transform`
Generating Documentation

- Create XHTML documentation for the simple UML model
  - Input: UML model with classes
  - Output: XHTML document

  M2M

- UML metamodel: exists
- XHTML metamodel: generate from XSD
Generating Documentation

• Use existing XSDs to generate EMF metamodels
  – Generate EMF metamodel from XSD, deploy
  – Write transformation against the new metamodel

• Example: create HTML documentation for the simple UML model
  – xsd from http://www.w3.org/2002/08/xhtml/xhtml1-strict.xsd
  – Minor changes to xhtml.ecore, direct used from workspace
  – Create XHTML document that lists classes with their descriptions
  – Add debug and constraint capabilities
  – Compose QVTO transformations
UML to XHTML Example

• Start with empty XHTML document
  – Typical document contains

  The **HTML** element representing document itself
  The document head
  The **HEAD** element
  The **TITLE** element
  The document body
  The **BODY** element

• Highlights:
  – Create document's skeleton in main()
  – XSD-backed model requires instance of DocumentRoot as a root element
  – Modify xhtml.ecore to allow access to text parts of mixed references
UML to XHTML Example

• List all classes containing by the model
  – In XHTML that list represents as follows

```xml
<div>
  <h2>Classes in simpleuml</h2>
  <table>
    <tr>
      <td>Association</td>
    </tr>
    <tr>
      <td>Class</td>
    </tr>
    ....
  </table>
</div>
```

  - UML classes is requested by (preserving lexical order on class names)

```cpp
query Package::allClasses() : OrderedSet(Class)
```

```cpp
mapping Class::class2row() : TrType
```
UML to XHTML Example

- Describe each UML class

<table>
<thead>
<tr>
<th>UML Class</th>
<th>mapping Class::class2div() : DivType</th>
</tr>
</thead>
<tbody>
<tr>
<td>-[*] generals</td>
<td>mapping Generalization::gen2listitem() : LiType</td>
</tr>
<tr>
<td>-[*] attributes</td>
<td>mapping Property::attr2listitem() : LiType</td>
</tr>
</tbody>
</table>

- Highlights:
  - evolve allClasses() query to mapping for achieving singleton behavior
  - filter empty generals/attributes sections by means of mapping guard ("when" clause)
UML to XHTML Example

- **Beautify XHTML document**
  - Add cross references

  `<a href="#simpleuml.Enumeration">Enumeration</a>`
  
  Reference to

  `<div id="simpleuml.DataType">class description</div>`

  - `href` id is created by means of recurrent `fullName()` query

- **Workflow examination**
  - console debug output (appearance controlled via configuration property)
  - mapping preconditions with asserting
Transformation composition

• Compose just created transformation with the previous one
  – Chain Ecore2Uml and Uml2Xhtml

    • Chain by means of QVTO transformation's instantiation
      – var t : Transformation := new Uml2Xhtml(Uml, Xhtml);
        var s : Status := t.transform();

    • Chain by means of Ant script (QVT Operational Help gives an example)
      – <qvto:transformation
          uri="transforms/Uml2Xhtml.qvto"
            >
            ...