Model Transformation with Operational QVT

QVT Operational - M2M component

http://www.eclipse.org/m2m

Radomil Dvorak
QVT Operational component lead
Borland Software Corporation
Agenda

• Overview of QVT Operational language

• M2M/QVTO + tooling support

• Examples
  ♦ Simple illustrative Ecore2Emof
  ♦ MDD use-case within GMF project

• Q&A
Operational QVT

• Final Adopted Specification - ptc/07-07-07

• Why operational?

• Designed for transformations that have to build target models of a complex structure

• In cases when there is no direct correspondence between individual elements of the source and target models -> might be difficult to describe declaratively

• QVTo – imperative (procedural) language specifying explicit steps to execute in order to produce the result
Operational Transformation

• Defines the process of converting \(\{1..*\}\) source models into \(\{1..*\}\) target models.

• The most typical scenario - \(M_a\) conforming to metamodel \(M_{Ma}\) converted into a model \(M_b\) conforming to metamodel \(M_{Mb}\).

• If \(M_a = M_b\) -> in-place transformation

• The metamodels involved in the transformation are manifested in transformation signature.

\[
\text{transformation } M_{MaToMb}(\text{in } M_a : M_{Ma}, \text{out } M_b : M_{Mb});
\]

• Set of typed model parameters indicate the referred metamodels and provides a mechanism for inspecting actual model instances in runtime.
  
  • in | out | inout direction kind -> restrictions to object creation, changeability
Model type declaration

• Model type is the type of transformation model parameters

• Implicit - no model type is declared explicitly; the metamodels can be resolved by name -> the effect of implicit model type declaration, taking the name of referred metamodel.

• Explicit - a concrete syntax construct placed before transf. signature

modeltype MMA uses “http://qvtexample/mm/MMa”;

• The used metamodels are referred by uri identifying the metamodel package or by package name

• Model type identifier can be part of qualified type names to resolve ambiguities -> MMA::A
Model type declaration advanced

• Metamodel conformance kind can be specified
  
  ◦ effective - (default) structural match based; indicates a declaration time metamodel, the actual metamodel involved at runtime, typically different versions of logically the same metamodel with compatible changes -> flexibility, high applicability
  
  ◦ strict - model objects must be instance of the exact classes from the referred metamodels, required for XMI serialization

• Restricting conditions on metamodels accepted by transformations

   modeltype MMa uses “http://qvtexample/mm/MMa”

   where { self.objectsOfType(A)->notEmpty() };

• Allows for validation check on input models without executing the transformation, using self variable of model type instance (a model)
Model parameters

- A MOF extent is associated with every model parameter, provides model elements container

- Model elements queried or created in the scope of parameter associated extent

  \[ \text{Ma.objects}()[A]; \]
  \[ \text{Mb.objects}()[B]; \]

- Transformation is a class; a single instance instantiated by implicit constructor
  - the contents of \text{in} | \text{inout} parameters extents is initialized
  - \text{out} parameters created with empty model extent
  - model parameters mapped to attribute slots, accessible within transformation, \text{this} variable refers to transformation

\[ \text{transformation MMaToMMb} \]
\[ \text{(in Ma : MMa, out Mb: MMb);} \]
Transformation entry point

- main() – signature-less imperative operation, sequentially executes list of expressions - *body*

- First and last transformation operation executed
- Called automatically after transformation implicit instantiation
- Single `main` operation per transformation
- abstract transformations, designed for reuse and not direct execution – no entry operation defined

Typically, selects elements within `in` model parameter extents -> source objects to mapping calls

```java
transformation Ecore2EMOF{
    in.ecore : ECORE, out.emof : EMOF);
/*
* Maps all root.ecore to emof packages
*/
main()
{
    ecore.rootObjects()[EPackage]->map toPackage();
}
```
Mapping operation

- Maps \{1..*\} source model elements into \{1..*\} target elements
- Source and target types indicated by operation signature

QVT Operational

\[
\text{mapping } A::\text{AtoB()} : B;
\]

\[
\text{assert } (a.\text{resolve}() -> \text{notEmpty}());
\]

Relations

- creates trace instance
- relation holds after execution
Mapping operation definition


{ <mapping-body> }

Is that so complex to write a mapping?

mapping ( <contexttype>::)? <identifier>()? : <result-parameters>? ?

{ <mapping-body> }

The most frequent case -> let's start with that

```java
10 mapping EPackage::toPackage() : EMOF::Package {
11 name := self.name;
12 uri := self.nsURI;
13 ownedType := self.eClassifiers->map toTy;
14 nestedPackage := self.eSubpackages->map
15 }
```
Contextual mapping operation

- **logically** extends the context type -> model element as source of mapping calls

- physically owned by the transformation class

```plaintext
transformation MMaToMMb(
    in Ma : MMa, out Mb : MMb);
main() {
    var a := Ma.rootObjects![A];
    a.map AtoB();
}

mapping A::AtoB() : B {
    this.map AtoB(a);
}
```
**Contextual mapping operation environment**

mapping (<contexttype>::)?

<identifier>(<parameters>? : <result-parameters>)?

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

**Operation environment**

```plaintext
-- Contextual
mapping A::AtoB() : B {
}

-- Non-contextual
mapping AtoB(in a : A) : B {
}
```
Mapping operation with when clause

mapping A::AtoB() : B when { self.isValid() } 

Execution semantics dependent on invocation mode

- **standard**
- **strict**

- **guard** – selects model elements for mapping
  
  a.map AtoB(); -- *std call semantics*

- **pre-condition** – must be always satisfied
  
  a.xmap AtoB(); -- *strict call semantics*
Mapping operation body

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

1) New instances created assigned to un-initialized \texttt{out} parameters
2) \texttt{Trace} instance created $\rightarrow$ relation holds

updating \texttt{inout}, \texttt{out} instances using object or assignment expressions

final computations before exiting, typically additional mapping invocations, logging, assert

```plaintext
mapping A::AtoB() : B {
    init {
        var d := self.resolveone(D);
    }
    propOfB := self.propOfA;
    refToC := self.map AtoC();
    end {
        result.refToC.map modifyC(d);
    }
}
```
Mapping operation body – object instantiation

Implicit instantiation section - creates out parameters instances

\[
\text{result} := \text{new } B();
\]

\[
\text{-- no init section}
\]

\[
\text{mapping } \text{A::AtoB}() : B \{ \\
\text{name} := self.\text{name}; \\
\}
\]

Init section - may create out objects explicitly

\[
\text{if (result = null) then} \\
\text{result} := \text{new } B();
\]

\[
\text{mapping } \text{A::AtoB}() : B \{ \\
\text{init} \{ \\
\text{if (condition1) then} \{ \\
\text{result} := \text{object SubTypeOfB} \{ \}; \\
\text{endif;}
\}
\text{name} := self.\text{name};
\}
\]

\[
\text{result} := \text{new } B();
\]
Mapping operation body – object population

Modifications of instantiated **inout** | **out** objects

```plaintext
-- implicit population section
mapping A::AtoB() : B {
    name := self.name;
}

-- explicit population section
mapping A::AtoB() : B {
    population {
        object result : B {
            name := self.name;
        }
    }
}

mapping A::AtoBC() : b: B, c: C {
    object b: B {
        name := self.name;
    }
    object c: C {
        name := self.name;
    }
}

mapping A::AtoBC() : b: B, c: C {
    object b: B { name := self.name; }
    object c: C { name := self.name; }
}

Multiple results

expand as

may reduce
```
Inline instantiation

- Object expression – refers to the instantiated class, provides a body to initialize new instances

- Used for simple tasks where mappings are not desirable

- Instantiated objects not reachable by resolve call – no traces created

- Create or update semantics controlled by use of variable referring to created/updated objects

- Poor reusability level -> solved by constructors

```java
-- always new instance
object A {
}

var a := null;
-- (a = null) new instance set to a
object a : A {
  name := ‘Rich’;
}

-- (a <> null) -> update
object a : { -- type known already
  name := a.name + ‘ ‘ + ‘Gronback’;
}
```
Assignment expression

• Assignment of a right side value to the target property or variable on the left side

• Assignments semantics for targets of collection type
  - null values skipped from assignment
  - duplicates eliminated when assigning to Set, OrderedSet target types
  - Reset semantics
    elements := Sequence {}; -- set empty target collection

  - Additive semantics (collections only)
    - all left side (non-null) values added to the original contents
    elements += object Element {}; -- single element added
    -- adds 2 elements -> 3 elements in the target property
    elements += Sequence { object Element {}, object Element {} };
Mapping invocation semantics

```java
main() {
    var a: A := object SubA {};
    a.map AtoB();
}

mapping A::AtoB() : B {
}

mapping SubA::AtoB() : B {
}
```

1. Resolve mapping operation based on the actual context instance – virtual call.
2. Check when clause if not satisfied -> return null
3. Guard succeeded, a check for existing trace for the given sources, targets is performed.
4. If the relation holds -> result parameters fetched from traces and returned; otherwise body is executed
Resolving objects

- Supported by resolve expression family
- Based on trace inspection -> only mapping operation source, targets can be resolved

Execution semantics modifiers

- **Direction** – source to target or *inverse*
- **Specific mapping** – given mapping reference
- **Multiplicity** – resolve one or many
- **Filtering condition** – only matching object
- **Time** – resolve now or at deferred time

**Typical use-cases:**
- Updating objects resulting from executed mappings
- Checking whether a mapping already executed
- Realizing transformed model cross-referencing
Resolve examples

- Direction
  a. `resolve();` -- source -> target
  b. `invresolve();` -- target -> source

- Specific mapping
  a. `resolveIn(A::AtoB, B);`

- Multiplicity of result type
  a. `resolveone(B);` -- single Object
  a. `resolve(B);` -- Sequence(Object)

- Time
  -- resolve now
  a. `resolveone(B);`
  -- resolve at deferred time
  a. `late resolveone(B);`

- Filtering condition & result type
  a. `resolveone(name='Joe');` -- Object
  a. `resolve(A);` -- Sequence(A)
  a. `resolve(a : A | a.name <> null);`
Late resolve

Normal execution time

object A {
    refToB := findSource().late resolveone(B);
}

1. Assignment not executed
2. Evaluates the source object of late resolve call
3. Stores all data required for later execution

main() {
    ...

} // end of transformation

Executes deferred assignments in sequence as detected by normal execution
inout - Mapping operation

<qualifiers>? mapping <param-direction-kind>?
  (<contexttype>::)?<identifier>(<parameters>?) (: <result-parameters>)?
  <extensions>? <when>? <where>?

- param-direction-kind
  - direction of the contextual parameter (if available)
  - possible values (in | inout);
  - in - the default direction, not notated

Operation environment

mapping inout A::updateA() {
}

self : A -> inout contextual parameter - implicit

mapping inout A::updateA() : A {
}

self : A -> inout contextual parameter - implicit
result : A -> out parameter - implicit
Reuse by composition

```plaintext
transformation MMaToMMbExt(
    in Ma : MMa, out Mb : MMb)
access transformation MMaToMMb(in MMa, out MMb);

main() {
    var a2b : AtoB := new MMaToMMb(Ma, Mb);
    a2b.transform();

    Mb.objects()[B]->map processB();
}

mapping inout B::processB() {
    ...
}
```

Explicitly instantiated

```
MMaToMMb

MMaToMMb(MMa, MMb)
```
Reuse by extension

transformation MMaToMMbExt(in ma : MMA, out mb : MMb) extends transformation MMaToMMb(in MMA, out MMb);

mapping inout B::adjustB () {
   -- do it our way
}

overrides

 implicitly instantiated

MMaToMMb

MMaToMMb(MMa, MMb)

mapping AtoB() : B

mapping inout B::adjustB()
Mapping level reuse facility - *inherit*

```plaintext
mapping A::AtoB() : B {
    name := self.name;
}

mapping A::AtoSubB() : SubTypeOfB
inherits A::AtoB {
    init {
        var nullName := self.name = null;
        calls
        hasName := not nullName;
    }
}
```
Mapping level reuse facility - *merge*

1. **mapping** `A::toSuperB1() : SuperB1` {
   ```
   name := self.name;
   ```
} 

2. **mapping** `A::toSuperB2() : SuperB2` {
   ```
   hasName := self.name <> null;
   ```
} 

**mapping** `A::AtoB() : B` 

**merges** `A::toSuperB1, A::toSuperB2` 

```
end {
end {
``` 

calls 

```
}
```
Mapping level reuse facility - *disjunct*

- **mapping** `A::AtoNamedB() : B`  
  - when `{ self.name <> null }`  
  - `{ name := self.name; }`

- **mapping** `A::AtoNoNameB() : B`  
  - when `{ self.name = null }`  
  - `{ name := '<unknown>'; }`

- **mapping** `A::AtoB() : B`  
  - **disjuncts** `A::AtoNamedB, A::AtoNoNameB`  
  - `{}`

- Selects the first match by type and satisfied guard
- Returns *null* if no mapping can be selected
Contextual (intermediate) property

- Similar concept as contextual operation
- Owned by transformation class but logically extends the context type
- Exists only in the scope of defining module
- Manipulated as regular properties – read/write access

```java
property A::myExtraProp : String;

main() {
    object A {
        myExtraProp := 'a String';
    };
}
```
Intermediate classes

- Ordinary classes defined purely for the internal purpose of a transformation.

- Only in the scope of the defining transformation

- In case it’s referenced in traces, persistence must be ensured

- Typically used for additional structural working data associated with instances of existing classes, usually from (read-only) metamodels.

    intermediate class DataForA {
        extraProperty : String;
    }

    intermediate property A::extraData : DataForA;
Instantiation in specific model extents

- In simple cases – target model extents resolved automatically

- Multiple model parameters of `inout` or `out` direction kind of the same model type can be solved by explicit instruction

- Option for explicit indication of the target extent by referring to a model parameter

- However, model elements may move between model extents due to containment reference assignments

```plaintext
transformation MMaToMMb(
    in Ma : MMa, out Mb : MMb,
    out mbExt : MMb);

main() {
    object B@mbExt {
        name := 'John';
    }
}

mapping A::AtoB() : B@Mb {
}

mapping A::AtoBExt() : B@MbExt {
}
```
Imperative OCL constructs – OCL extension

- Loop support – **while**, **forEach** – (iterates over collection)

- Imperative iterators – powerful, concise
  - `Ma->objects()![A];` -- *selects single object of kind A*

- Execution control
  - `return` – usual semantics of exiting operation with a result value
  - `break`, `continue` - loops, iterators

- Variable initialization – scoped within block expressions

- Switch – avoids complex if else if ....

- Exceptions – try {...} catch {...} semantics
Black-boxing

Enables to escape the whole transformation/library or its parts that are difficult or impossible to implement in pure QVT.

**Black-box transformation**
contains only transformation signature and no implementation (entry point, mapping operations)

\[
\text{transformation} \text{ MMaToMMb(in Ma : MMa, out Mb : MMb);}
\]

**Black-box operation** – signature only operation, no body specified -> external

**mapping** A::AtoB() : B;

• Compliance points of transformation definition – indicated by the transformation writer
  • **QVT-Operational** - uses black-box operation
  • **QVT-Operational** - pure QVT language
Configuration properties

• **configuration** qualifier keyword used with module property declaration

```plaintext
transformation Diagram2GMFGen (in inMap : MAP, out genModel : GEN);
```

-- true indicates that RCP is targeted
```plaintext
configuration property rcp : Boolean;
```

• The initialization step - out of the QVT spec scope -> any external mechanism allowed
  • Launch configuration
  • property file

• The choice of implementation
Log expression

- Adds log record entry to the execution environment.
  - **message**  text
  - **element**  optional, model element associated with the log
  - **level**  optional, raw integer value – applicable for filtering

- May be conditional

```plaintext
abstract mapping EStructuralFeature::toProperty() : Property
  inherits ETypedElement::toTypedElement
  merges ETypedElement::toMultiplicity
{
  isDerived := self.derived;
  isReadOnly := not self.changeable;
  end {
    log('Transforming EReference', self.name)
    when self.oclIsKindOf(ERefERENCE);
  }
}
```
# Assertion

Asserts a condition and generates error message in case it does not hold.

- **severity level** - `warning` | `error` | `fatal`
  - `fatal` - throws exception and transformation execution terminates

- **log record** - optionally used with log expression

```java
transformation Ecore2EMOF(in.ecore : ECORE, out.enof : EMOF);

main() {
  assert fatal (ecore.objects()[ECORE::EPackage]->notEmpty())
    with log('Expecting at minimum 1 EPackage instance');

 .ecore.rootObjects()[ECORE::EPackage]->map toPackage();
}
```

<terminated> Ecore2EMOF(1) [Operational QVT Interpreter] In-process runner

ASSERT [fatal] failed at (Ecore2EMOF:0) : Expecting at minimum 1 EPackage instance
Terminating execution...
org.eclipse.m2m.qvt.oml.internal.ast.evaluator.QvtAssertionFailed: Fatal assertion failed
  at Ecore2EMOF.main(Ecore2EMOF.qvto:0)
  at Ecore2EMOF.<init>(Ecore2EMOF.qvto:7)
QVTO – where we are?

• Based on MDT OCL
  - reuses OCL metamodels
  - extends OCL parser
  - extends OCL evaluator

• So far, primary focus on concrete syntax, execution and reasonable tooling support
  - AST model with some differences from the spec – legacy reasons
  - concrete syntax – not complete, but major concepts supported

• Next steps
  - complete concrete syntax – executable (except parallel transf. etc)
  - standardize QVT AST -> XMI-Exportable
Editor support – syntax highlight, hovers, hyperlinks

```java
modeltype ECORE uses "http://www.eclipse.org/ecore"
modeltype EMOF uses "http://schema.omg.org/spec/emof/

transformation Ecore2EMOF (in ecore : ECore, 
main() {
    var ePackages := ecore.rootObjects() [ECORE;
ePackages->map toPackage();
}

mapping EPackage::toPackage() : EMOF::Package {
    name := self.name;
    uri := self.nsURI;
    ownedType := self.eClassifiers->map toTy;
    nestedPackage := self.eSubpackages->map
    nestedPackage : OrderedSet(Package) - eMOF::Package
}

mapping EClass::toClass () : EMOF::Class {
    name := self.name;
    isAbstract := self.abstract or self.int;
    superClass += self.eSuperTypes.late reso
```
Editor support - annotations, problem markers, outline

```csharp
model packages:

mappings:

- EPackage::toPackage(): EMOF::Package {
  name := x.name;
  uri := self.nsURI;
  ownedType := self.eClassifiers->map toType()->asorderedSet();
}

- EClass::toClass(): EMOF::Class {
  name := self.name;
  isAbstract := self.abstact or self.inAbstract;
}
```

- Variables
- Metamodel Explorer
- Outline

2 items

- Description
- Errors (2 items)
  - There is no property 'ownedType' in type 'emof::Package'
  - Unrecognized variable: (x.name)
Model Transformation with Operational QVT

Code completion

```java
mapping EClass::toClass() : EMOF::Class

    name := self.name;

    isAbstract := self.abstract or self.interface;

    superClass := self.eSuperTypes.late resolveIn (EClass::toClass, EMOF::Class)

    ownedAttribute += self.eAttributes; 
    ownedOperation += self.eOperations;

abstract mapping EClass::toDisjuncts

mapping EEnum::toEnum()

    ownedLiteral += self.eAllValues;

object EMOF::Enum

abstract mapping ECORE::NamedElement

abstract mapping EMOF::TypedElement
```
Debugging support
GMF generator model creation

```plaintext
modeltype MAP uses "http://www.eclipse.org/gmf/2006/mappings);
modeltype GEN uses "http://www.eclipse.org/gmf/2006/GenModel);

transformation Diagram2GMFGen(in inMap : MAP, out genModel : GEN);
```

Run QVTTo transformation