M2M-Transformation mit der QVT Operational Mappings

Siegfried Nolte

Beethovenstraße 57
22941 Bargteheide
siegfried@siegfried-nolte.de

Abstract: QVT ist ein Sprachkonzept der Object Management Group zur Transformation von formalen Modellen. In diesem Beitrag werden ausgehend von einem einfachen Fachklassenmodell die zentralen Konzepte und Techniken der Transformationssprache Operational Mappings vorgestellt, so dass abschließend eine vollständige Transformation eines UML-Modells der PIM-Ebene in eins der PSM-Ebene vorgenommen werden kann.

1 Einordnung der QVT-O in die MDA

Modellgetriebene Architekturentwicklung bedeutet das iterative Erstellen von Anwendungssystemen über ständig wechselnde Modellierungs- und Transformationsschritte [MDA03, PM06]. Mit Hilfe von formalen Modellierungssprachen, sogenannten MOF-Sprachen, werden Modelle erstellt, die in einem Transformationsschritt mit QVT-Sprachen in ein neues Modell überführt werden, welches wiederum mit Hilfe einer formalen Modellierungssprache dargestellt und bearbeitet werden kann. Auf diesem Weg wird als Ausgangsmodell einer Entwicklungsphase jeweils ein konsistentes Modell durch Transformation initial bereitgestellt.

MOF-Sprachen sind Modellierungssprachen, denen nach dem MOF-Ansatz der OMG [MOF06] ein formales Metamodell zugrunde liegt. Das Metamodell ist gewissermaßen eine Datenstruktur, die nach der MOF-Spezifikation in Form eines UML-Klassendiagramms definiert wird [Nol07]. Alle OMG-Modellierungssprachen – so auch die BPMN, UML, SysML – sind formale Modellierungssprachen, die in diesem Sinne mit UML-basierten Metamodellen spezifiziert worden sind. Eine weitere, etwas leichtgewichtigere Möglichkeit der Beschreibung von formalen Modellierungssprachen besteht aus der Einarbeitung von domänenspezifischen Belangen in die Modellierungssprache UML. Dies Mittel stellt das UML-Metamodell mit den UML-Profilen zur Verfügung [UML10]. Im Rahmen dieses Beitrags arbeiten wir exemplarisch mit einem UML-Profil, welches eine einfache Erweiterung des UML-Metamodells in Form des Stereotyps `<<entity>>` benutzt. Mit diesem Stereotyp, der mit der Metaklasse `Class` assoziiert ist, werden die Fachklassen eines UML-Modells markiert, die mit der Transformation in besonderer Weise behandelt werden sollen. Mit der Wahl des UML-Profils ist UML als Modellierungssprache und damit als Modelltyp für die Transformation festgelegt.

2 Transformation mit QVT-O

QVT *Operational Mappings* (QVT-O) ist eine Sprache zur Transformation von formalen Modellen, die der Klasse der imperativen Sprachen zuzuordnen ist [QVT08]. In QVT-O werden die Transformationen nicht, wie bei der *Relations Language*, deskriptiv über gültige Relationen zwischen den Elementen von Quell- und Zielmodellen beschrieben, sondern imperativ durch Mapping-Operationen. Zur Erläuterung der Sprachkonzepte der QVT-O und der Modelltransformation [Nol09] soll ein einfaches Fachklassendiagramm der PIM-Ebene dienen. Die Fachklassen, die im Rahmen der Transformation in besonderer Weise behandelt werden sollen, sind unter Verwendung eines einfachen UML-Profiles mit dem Stereotyp `<<entity>>` markiert. Als Zielmodell wird ebenfalls ein UML-Klassendigramm hergestellt.

2.1 Transformationen

Eine QVT-O-Transformation beschreibt die Überführung einer beliebigen Anzahl von Quellmodellen in eins oder mehrere Zielmodelle. Quellmodelle wie auch Zielmodelle müssen von bestimmten Modelltypen sein, die zu Beginn einer Transformation deklariert werden. Modelltypen referenzieren formale Metamodelle, die im Kontext bekannt sind. Im Fall des Eclipse-basierten Werkzeugs QVT-Operational [EMP10] werden Metamodelle als Eclipse-Plugins organisiert [Nol07].

```
modeltype UML uses 'http://www.eclipse.org/uml2/2.1.0/UML';
transformation GI1_Transform (in src :UML, out trgt :UML);
main(){ src.objects()[Package]->map transformPackage(); }
```

Die **main**-Methode ist die Mapping-Operation, die als Einstiegspunkt für die Transformation gilt. Es handelt sich dabei um eine Mapping-Operation, die die Modelle selbst, hier `src` und `trgt`, zum Gegenstand hat. Aus dem Grund ist es auch nicht erforderlich – und bei manchen QVT-O-Interpretern auch nicht möglich –, Eingangs- und Ausgangsargumente für die **main**-Methode anzugeben. Mit dem Zusatz **in** und **out** wird festgelegt, ob ein Modell Quelle oder Ziel ist. QVT-O basiert auf OCL [OCL10]. Einerseits ist QVT-O mit Hilfe der OCL entwickelt worden, andererseits steht die OCL in QVT-O vollständig zur Verfügung, soweit sie in dem jeweiligen Interpreter implementiert ist. Die Syntax der Anweisungen entspricht im Wesentlichen der OCL-Syntax. Die obige Anweisung „`objects()[Package]`“ ist ein sogenannter *Shorthand* für die QVT-Standardfunktion „`objectsOfType(Package)`“.

2.2 Objekterzeugung mit Mapping-Operationen und Inline Mapping

Die Transformation besteht aus einer Reihe von Mapping-Operationen, mit denen Schritt für Schritt die Überführung von Elementen des Quellmodells in Elemente des Zielmodells beschrieben wird:

```
mapping Package::transformPackage() : Package
when { self.name != oclIsUndefined(); }{ name:=self.name; }
```

In diesem Code-Abschnitt wird ein `Package` eines Quellmodells in ein `Package` eines Zielmodells überführt. Die in dem optionalen **when**-Prädikat angegebene Vorbedingung regelt, dass dies nur dann erfolgt, wenn das `Package` des Quellmodells als Objekt der Metaklasse „`NamedElement`“ auch einen definierten Wert besitzt. **self** ist eine QVT-Standardvariable, die den Gegenstand des Mappings selbst, hier das aktuelle `Package`, repräsentiert. In Mapping-Operationen stehen weitere Standardvariablen zur Verfügung, **result**, die das Zielobjekt einer Operation in seiner jeweiligen Ausprägung repräsentiert, und **this**, womit die Transformation selbst adressiert werden kann, um zum Beispiel Informationen über den Status abzufragen.

Bei der obigen Mapping-Operation handelt es sich um eine Kurzform. Ein vollständig spezifiziertes Mapping wird in frei gestaltbaren Blöcken organisiert, jeweils einem zur Initialisierung (**init**), zur Durchführung (**population**) und zur Nachbehandlung (**end**) der Operation. Dies kann bei komplexen Mappings durchaus Sinn machen. Die komplette Mapping-Anweisung mit allen optionalen syntaktischen Bestandteilen sieht folgendermaßen aus:

```
mapping Package::transformPackage() : Package
when  { self.name != oclIsUndefined(); }
{
   init       { var pckgName := self.name; }
   population { object result:Package { name:=pckgName; }}
   end        { /* Terminierende Anweisungen, Meldungen, */
                /* Statusabfragen */ }
}
```

Der Aufruf einer Mapping-Operation erfolgt mit einer **map**-Anweisung:

```
src.objects()[Package]->map transformPackage()
```

Auf jedes Objekt vom Typ `Package` eines Eingangsmodells `src` wird die Mapping-Operation `transformPackage` angewendet. Mit dem Mapping haben wir eine Technik der expliziten Objekterzeugung kennen gelernt. Eine weitere Möglichkeit, Objekte in Zielmodellen zu erzeugen, besteht darin, dies implizit mit Hilfe eines Inline-Mappings vorzunehmen. Die Technik des Inline-Mappings bietet sich insbesondere dann an, wenn die Objekte, die im Zielmodell neu angelegt werden sollen, im Quellmodell nicht in irgendeiner Weise existieren, zum Beispiel die `getter`- und `setter`-Methoden für den Zugriff auf private Attribute:

```
mapping Class::transformClasses() : Class
{
   name := self.name;
   ownedOperation+=self.ownedAttribute->object(p) Operation
   { name := 'get_' + p.name; };
```

```
   ownedOperation+=self.ownedAttribute->object(p) Operation
   { name := 'set_' + p.name; };
}
```

2.3 QVT-O und imperative Ausdrücke – eine Auswahl

Alle QVT-Sprachen basieren auf der Object Constraint Language [OCL10]. Dadurch können in beliebiger Weise QVT-Anweisungen mit OCL-Anweisungen kombiniert werden, was den QVT-Sprachen eine große Flexibilität verleiht. Eine der wertvollsten Anweisungen ist der **log**-Ausdruck, mit dem an beliebiger Stelle in einer Mapping-Operation eine Ausgabe auf die Konsole vorgenommen werden kann, um zum Beispiel aktuelle Ausprägungen von Variablen oder Objekten anzuzeigen.

```
log ( 'Do' ); log ( self.name );
log ( 'Inhalt der self-Variable ', self );
```

Ein bedingter Ausdruck ist ein Ausdruck, in dem die Anweisungsteile in Abhängigkeit von einer Bedingung ausgeführt werden. QVT-O erlaubt als bedingte Ausdrücke **if**- und **switch** -Konstrukte:

```
if (self == oclIsUndefined())
then log ( 'Self hat keinen definierten Wert' );
else log ( 'Self.name = ' + self.name );
endif;

switch
{
   case (self == oclIsUndefined())
        { log ( 'Self hat keinen definierten Wert' ); }
   else { log ( 'Self.name = ' + self.name ); }
};
```

Schleifen und iterative Anweisungen werden entweder mit dem OCL-Iterationsausdruck **iterate** oder mit einem **while**- oder **for**-Ausdruck implementiert:

```
var s1:Integer := 0, s2:Integer := 0, s123 := Set{1,2,3};
s1 := s123->iterate(idx: Integer; i: Integer = 0 | i+idx);
s123->forEach (idx) { s2 := s2 + idx };
log ('Summe s1 = ', s1); log ('Summe s2 = ', s2);
```

Für das Sammeln von Objekten kann die **collect**-Anweisung hinzugezogen werden:

```
ownedAttribute := self.ownedAttribute->
   collect( p | object Property { name := p.name; } );
```

Für diese Konstruktion stellt die QVT-O-Spezifikation wieder einen sogenannten *Shorthand* zur Verfügung, der allerdings noch nicht von allen QVT-O-Interpretern unterstützt wird:

```
ownedAttribute :=
self.ownedAttribute->object (p) Property {name := p.name;};
```

2.4 Queries

Ein weiteres Hilfsmittel der QVT-O sind Queries. Dies sind Hilfsfunktionen, mit denen sich QVT- und OCL-Anweisungen bündeln und umfangreiche Skripte besser strukturieren lassen:

```
query getTypeName( srcType : Type) : String
{
   return if ( srcType.name = 'UnlimitedNatural' )
          then 'Double' else srcType.name endif;
}
```

Der Aufruf einer Query ist analog zu dem Aufruf einer Mapping-Operation, wobei die **map**-Anweisung weggelassen wird:

```
    name := getTypeName(self);
```

Queries in Kombination mit OCL-Anweisungen bieten recht vielfältige Möglichkeiten, mit denen sich das Leben eines MDA-Architekten doch sehr erleichtern lässt. Mit der folgenden Query zum Beispiel wird mit Hilfe der Standardfunktion `getAppliedStereotype()` geprüft, ob eine Klasse mit dem Stereotyp `<<entity>>` des UML-Profils `UMLEJB` markiert ist. Wenn ja, kann die Klasse im Rahmen der Transformation angemessen behandelt werden:

```
query Class::isStereotypedBy ( str : String ) : Boolean
{
   var st : Stereotype := null;
   st := self.getAppliedStereotype( 'UMLEJB::' + str );
   return if ( self.isStereotypeApplied(st) )
          then true else false endif;
}

mapping  Class::transformClasses () : Class
when { self.isStereotypedBy('entity') } { ... }
```

2.5 Weiterführende Konzepte

Häufig lassen sich nicht alle Transformationen und Operationen in einer Phase erledigen, so dass QVT-O spezielle Sprachmittel und Funktionen anbietet, damit bestimmte Trans-

formationsaktionen auch in nachfolgenden Phase vorgenommen werden können: Object Resolution und Intermediate Data. Alle Objekte, die im Rahmen einer Transformation angefasst werden, werden in einem *Trace* abgelegt. Mit Objekt-Resolution wird ein Ansatz beschrieben, die Objekte unter Verwendung des *Traces* zu adressieren. Dabei wird zwischen allgemeiner Resolution und spezieller Resolution unterschieden. Allgemeine Resolution gilt grundsätzlich für alle Objekte der gesamten Transformation; spezielle Resolution adressiert gezielt die Objekte einer bestimmten Mapping-Operation.

Allgemeine Resolutionsfunktionen sind:

- **`resolve()`**, welche zu einem Objekt des Quellmodells eine Liste aller Zielobjekte eines vorgegebenen Typs liefert, die aus dem Quellmodell generiert worden sind.
- **`resolveone()`** liefert das letzte Zielobjekt des gegebenen Typs, welches sich im Trace befindet.
- **`invresolve()`** wird angewendet auf ein Zielobjekt und liefert eine Liste aller Quellobjekte eines gegebenen Typs, die zur Erzeugung des Zielmodells benutzt wurden.
- **`invresolveone()`** liefert das letzte Quellobjekt eines gegebenen Typs.

Die speziellen Resolutionsfunktionen **`resolveIn()`**, **`resolveoneIn()`**, **`invresolveIn()`**, **`invresolveoneIn()`** sind analog definiert; bei ihnen wird jeweils noch die Mapping-Operation mit Namen angegeben, die bei der Selektion des oder der jeweiligen Elemente zusätzlich in die Betrachtung einbezogen werden soll.

Intermediate Data definiert ein Konzept, dem Metamodell im Rahmen einer Transformation dynamisch weitere Elemente hinzuzufügen, die nur während der Ausführungszeit der Transformation vorhanden sind und den Mapping-Operationen und Hilfsfunktionen zur Verfügung stehen. Es kann sich dabei um zusätzliche Metaklassen (**`intermediate class`**) oder um Attribute von vorhandenen Metaklassen handeln (**`intermediate property`**).

```
intermediate class Entity { name:String; kind:String; };
intermediate property Package::entities:Sequence (Entity);
```

In obigem Beispiel wird dem Metamodell eine weitere Klasse `Entity` hinzugefügt. Die Metaklasse Package erhält ein zusätzliches Attribut `entities`, welches eine Sammlung von Instanzen der Klasse `Entity` aufnehmen kann. Mit diesen Ergänzungen ist man in der Lage, Klassen im Modell, die zum Beispiel mittels eines Stereotyps als `<<entity>>` markiert sind, zu sammeln und vorübergehend einem Paket zuzuordnen, damit sie später bei der Erzeugung des Zielmodells angemessen eingebracht werden können. Bei der Vertiefung dieser Konzepte kann auf [Nol09] zurück gegriffen werden.

3 Zusammenfassung und Ausblick

QVT-O ist ein Bestandteil der *Eclipse Modeling Workbench* [EMP10] und steht damit als mittlerweile recht stabiles Produkt einem großen Kreis von Entwicklern und IT-Architekten zur Verfügung. Die für diesen Beitrag verwendete MDA-Entwicklungsplatt-

form ist in [Nol10] detailliert beschrieben. Die erforderlichen Metamodelle werden als Eclipse-Plugins erwartet, was jedoch mit Hilfe des Eclipse Modeling Frameworks und der UML Modeling Tools verhältnismäßig leicht zu bewerkstelligen ist [Nol07]. Das Metamodell der UML, so wie es hier verwendet wird, ist in einer spezifikationsnahen Ausprägung stets Bestandteil des Modeling Projektes. Die Repräsentation der Zielmodelle in einer grafischen Form ist jedoch nach wie vor eine Herausforderung, der sich die Anbieter von Modellierungswerkzeugen noch stellen müssen. Ohne dies ist es fraglich, ob das Angebot der Modelltransformation angenommen wird, da man mit mittlerweile sehr leistungsfähigen Model-nach-Text-Transformatoren auch schon in frühen Phasen der Entwicklung sinnvoll und effektiv tragfähige Ergebnisse produzieren kann. Trotzdem bleibt zu hoffen, dass Modelltransformationen über kurz oder lang mehr in die Kultur einer systematischen modellgetriebenen Architekturentwicklung einfließen werden. Dies hätte zweifelsohne wertvolle Auswirkungen auf die Qualität und Stabilität der zu entwickelnden Softwaresysteme zur Folge.

Literaturverzeichnis

[EMP10] Eclipse Modeling Project; http://www.eclipse.org/modeling/

[MDA03] MDA Guide, Version 1.0.1. www.omg.org/cgi-bin/doc?omg/03-06-01, 2003

[MOF06] MOF 2.0/Meta Object Facility Core, Version 2.0. http://www.omg.org/spec/MOF/, 2006

[Nol10] Nolte, S.: freeMDA - Eine freie Plattform für eine integrierte modellgetriebene Anwendungsentwicklung. EclipseMagazin - Heft 01, 2010

[Nol09] Nolte, S.: QVT – Operational Mappings. Springer-Verlag, 2009

[Nol08] Nolte, S.: Modelle und Transformationen – Eine pragmatische Betrachtung der MDA QVT. ObjektSpektrum - Heft 5, 8.2008

[Nol07] Nolte, S.: Modelle und Metamodelle im Eclipse Kontext. ObjektSpektrum, Heft 6, 2007

[OCL10] Object Constraint Language. http://www.omg.org/spec/OCL/2.2/, 2010

[PM06] Petrasch R, Meimberg O: Model Driven Architecture. dpunkt Verlag, 2006

[QVT08] MOF QVT - Version 1.0. http://www.omg.org/spec/QVT/1.0/, , 2008

[UML10] UML Superstructure and Infrastructure. http://www.omg.org/spec/UML/2.3/, 2010