M2M-Transformation mit der QVT Relations Language

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 Relations Language 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 in die MDA

Die *Model Driven Architecture* (MDA) der Object Management Group (OMG) beschreibt einen Ansatz der Anwendungsentwicklung auf der Grundlage von modellbasierten Techniken [MDA03]. Auf der einen Seite sind dies Sprachen und Verfahren zur Modellierung von Sachverhalten einer betrachteten Domäne, auf der anderen Seite handelt es sich um Konzepte zur Transformation der Modelle bis hin zur Erzeugung von Software-Bausteinen. Die Modellierung eines betrachteten Ausschnittes der realen Welt, der Domäne, erfolgt unter Verwendung von formalen, metamodellbasierten Modellierungssprachen, zum Beispiel der Business *Process Modeling Notation* (BPMN) oder der *Unified Modeling Language* (UML).

Ein Anwendungsentwicklungsprozess im Allgemeinen wie auch Modellierung im Besonderen ist iterativ inkrementell, der Entwicklungsprozesses findet über mehrere Abstraktionsebenen statt, wobei die Modelle immer weiter formalisiert und konkretisiert werden bis hin zu einer maschineninterpretierbaren Form, zum Beispiel einem Code-Artefakt in einer beliebigen Programmiersprache. Rechnergestützte Transformation von Modellen aus einer Entwicklungsebene in eine andere – aus der rechnerunabhängigen Modellierungsebene (CIM) über die plattformunabhängige Modellierungsschicht (PIM) und die plattformspezifische Schicht (PSM) zum Implementierungsmodell (IM) – kann dabei von großem Wert sein [PM06].

Der Vorschlag der OMG zur Transformation von Modellen ist die QVT, die *Query View Transformation*, eine Spezifikation von Sprachen, mit denen Modelle einer formalen Modellierungssprache in Modelle anderer formaler Modellierungssprachen überführt werden können [QVT08]. Mit Veröffentlichung der Version 1.0 im April 2008 ist die QVT ein von der OMG herausgegebener Standard. Die QVT bietet zwei Klassen von Sprachen zur Modelltransformation an, zum einen deskriptive Sprachen, von denen als

Beispiel hier die *Relations Language* (QVT-R) näher beleuchtet werden soll, und zum anderen imperative Sprachen, ein Beispiel dafür ist die *Operational Mappings* (QVT-O).

2 Modelle und Metamodelle

QVT-Sprachen arbeiten auf Modellen, welche mit Hilfe von formalen Modellierungssprachen, sogenannten MOF-Sprachen, entwickelt worden sind [Nol07]. Formale Modellierungssprachen sind Sprachen, denen eine wohldefinierte Syntax zugrunde liegt. Nach der MOF-Spezifikation der OMG [MOF06] wird die Definition von Modellierungssprachen in ihrem syntaktischen Teil unter Verwendung der UML – konkret, UML-Klassendiagramme – vorgenommen. Die formale Spezifikation einer Modellierungssprache in einem Modell wird als Metamodellierung bezeichnet, das Modell der Modellierungssprache ist ein Metamodell. Alle OMG-Modellierungssprachen, so auch die BPMN, UML und SysML, sind formale Modellierungssprachen in diesem Sinne. Das Ergebnis einer Transformation auf dieser Grundlage ist jeweils wieder ein formales Modell, das mit einer formalen Modellierungssprache dargestellt und im Rahmen des inkrementellen Entwicklungsprozesses weiter bearbeitet werden kann.

3 Transformation mit QVT-R

An einem einfachen UML-Modell werden einige grundlegende Konzepte und Transformationstechniken mit der deskriptiven QVT-Sprache QVT-R demonstriert [Nol09]. Mit der Wahl von UML als Modellierungssprache ist das Metamodell festgelegt. Die Fachklassen, die im Rahmen der Transformation behandelt werden sollen, sind unter Verwendung eines speziellen UML-Profiles mit dem Stereotyp `<<entity>>` markiert. Die Möglichkeit der Einarbeitung von domänenspezifischen Belangen in die Modellierungssprache UML ist eine sehr interessante Technik, die das UML-Metamodell mit UML-Profilen anbietet [UML10]. Als Zielmodell soll ebenfalls ein UML-Klassendigramm erstellt werden, welches bereits einige Designaspekte wie Paketierungen, Schnittstellen und Muster aufzeigt. Als Entwicklungsplattform wird ein aus freien Komponenten bestehendes Produkt eingesetzt [Nol10], welches auf der Basis der Eclipse Modeling Workbench entwickelt worden ist [EMP10]. Zur Unterstützung der Transformation soll mediniQVT [Med10] dienen.

3.1 Transformationen, Relationen und Domänen

QVT-R-Transformationen haben einen Namen und eine Liste von Parametern, in der die Modellkandidaten angegeben werden. Die Modelle sind Instanzen von Modelltypen, diese referenzieren Metamodelle, die im Kontext bekannt sein müssen [Nol07]. In dem Eclipse-basierten mediniQVT müssen die Metamodelle entweder als ein Eclipse-Plugin vorliegen oder alternativ in Form des universellen Austauschformates XMI [XMI07]. Die erforderlichen Metamodelle werden im Rahmen der Eclipse-Präferenzen „QVT Metamodels“ spezifiziert.

Das folgende Beispiel überführt ein UML-Quellmodell in ein UML-Zielmodell, wobei alle Packages eines Quellmodells mit ihrem Namen in Packages eines Zielmodells transformiert werden. Der Kern einer QVT-R-Transformation ist die Relation. Eine Relation ist die Spezifikation einer gültigen Beziehung zwischen einem Element eines Quellmodells und einem Element eines Zielmodells. In dem Beispiel werden alle `Packages` des Quellmodells `source` gemäß den vorgegebenen, in diesem Fall sehr einfachen, Bedingungen geprüft und diejenigen, die die definierten Bedingungen erfüllen, werden als Packages im Zielmodell `target` angelegt:

```
transformation GI1_Transformation (source:uml, target:uml)
{
   top relation Packages
   {
      pckgName : String;
      checkonly domain source srcPkg : Package
      { name = pckgName };
      enforce   domain target dstPkg : Package
      { name = pckgName };
   }
}
```

Die Prüfungen der Bedingungen und die Zuweisungen werden nach den Prinzipien des *Pattern Matching* vollzogen. Diesem Zweck dienen die Domänenspezifikationen. Jede Domäne beschreibt ein Pattern, welches sich auf die Attribute des angegebenen Modellelementes bezieht. Im Fall von **`checkonly`**-Domänen erfolgt eine Prüfung und ein Abgleich der Werte der Elemente eines Quellmodells, im Fall **`enforce`** werden die Attribute des Prüfkandidaten mit den gegebenen Werten überschrieben und damit ein Objekt im Zielmodell erzeugt. Beim **`checkonly`** ist eine Besonderheit zu beachten, die hier ausgenutzt wird: `pckgName` ist eine selbstdefinierte freie Variable, diese besitzt initial keinen Wert, sie ist also `oclUndefined`. Beim *Pattern Matching* bezogen auf die Domäne „**`checkonly domain`** `source`“ wird der Wert von `name` an die freie Variable `pckgName` übergeben, womit die Bedingung erfüllt wird. Mit dieser Technik ist ein Initialisieren von Zielelementen unter Verwendung von Quellelementen möglich.

3.2 Top-level- und Non-Top-Level-Relationen

In der QVT-R wird unterschieden zwischen *top-level*- und *non-top-level*-Relationen. *top-level*-Relationen – gekennzeichnet mit dem Schlüsselwort **`top`** – sind solche, die bei der Ausführung einer Transformation grundsätzlich aktiviert und abgearbeitet werden, *non-top-level*-Relationen sind solche, die explizit aufgerufen werden müssen. In einer Transformation muss es mindestens eine *top-level*-Relation geben. Die *top-level*-Relation `Packages` würde also beim Start der Transformation alle Packages eines Modells nach einem vorgegebenen Muster untersuchen und die entsprechende Aktion im Zielmodell ausführen, vorausgesetzt, es gibt `Packages` im Quellmodell. Also sollte für jedes Element eines Quellmodells, welches „top“ in einem Diagramm vorkommen kann – also

zum Beispiel `Package`, `Class`, `Interface`, `DataType` – eine *top-level*-Relation definiert sein.

Wenn es Beziehungen zwischen den Elementen im Modell gibt, die es nötig machen, dass ein Element im Rahmen einer Objekterzeugung im Zielmodell bereits existiert, dann ist das in den Pattern-Definitionen durch ein **`when`**-Prädikat entsprechend zu formulieren. In folgendem Beispiel sind sowohl die Relationen `Packages` wie auch `Classes` *top-level*. Es gibt jedoch Klassen, die Komponenten von Paketen sind, in dem Fall müssen die entsprechenden Pakete bereits bearbeitet und im Zielmodell angelegt sein, bevor die Bearbeitung der Klassen erfolgen kann. Dies wird durch den expliziten Aufruf der Relation `Packages` in dem **`when`**-Prädikat der Relation `Classes` gewährleistet.

```
transformation GI2_TopLevel ( source:uml, target:uml )
{
   top relation Packages
   {
      pckgName : String;
      checkonly domain source srcPkg : Package
      { name = pckgName };
      enforce   domain target dstPkg : Package
      { name = pckgName };
   }
   top relation Classes
   {
      className : String;
      checkonly domain source srcCls : Class
      {
         name  = className,
         owner = srcPkg : Package {}
      };
      enforce   domain target dstCls : Class
      {
         name  = className,
         owner = dstPkg : Package {}
      };
      when { Packages  ( srcPkg, dstPkg ); }
   }
}
```

Wenn allerdings in dem Modell eine hierarchische Ordnung zwischen den Elementen existiert, wie das zum Beispiel bei UML-Modellen der Fall ist (mal abgesehen davon, dass im UML-Metamodell das Attribut `owner` ohnehin nur lesbar und nicht veränderbar ist und somit in der **`enforce`**-Domäne nicht verwendet werden kann), dann ist es oft einfacher, die Transformation explizit auszulösen durch Aufrufe von *non-top-level*-Relationen in den **`where`**-Klauseln.

```
transformation GI2_NonTopLevel ( source:uml, target:uml )
{
   top relation Package
   {
      checkonly domain source srcPkg : Package {...};
      enforce   domain target dstPkg : Package {...};
      where { Classes  (srcPkg, dstPkg ); }
   }
   relation Classes
   {
      checkonly domain source srcP : Package {...};
      enforce   domain target dstP : Package {...};
   }
}
```

3.3 when- und where-Prädikate

In den obigen Beispielen sind zwei unterschiedliche Bedingungstypen eingeführt worden, **when**- und **where**-Klauseln. **when**-Prädikate dienen als Vorbedingung für die Erzeugungskomponente einer Relation, die letztendlich erfüllt sein muss, damit der generative Teil überhaupt bearbeitet werden kann. Das **where**-Prädikat bewirkt, dass immer dann, wenn ein Zugriff auf ein Objekt oder eine Variable in der **enforce**-Domäne erfolgt, die in der **where**-Klausel beschriebenen Aktionen ausgeführt werden.

Zur Erinnerung: *top-level*-Relationen werden mit dem Start einer Transformation aktiviert und ausgeführt. Ein explizites Auslösen einer *top-level*-Relation kann nur über ein **when**-Prädikat vorgenommen werden. Der Aufruf von *non-top-level*-Relationen erfolgt ausschließlich in **where**-Klauseln. Angenommen, es gibt ein UML-Profil `UMLEJB`, in dem der Stereotyp `<<entity>>` definiert und mit der Metaklasse `Class` assoziiert ist. In dem folgenden Beispiel werden dann ausschließlich die Klassen behandelt, die mit dem Stereotyp `<<entity>>` markiert sind. `getAppliedStereotype()` und `isStereotypeApplied()` sind Standardfunktionen, die den Zugriff auf Stereotypen erlauben.

```
relation Classes
{
   className, className1 : String;
   st: Stereotype;
   checkonly domain source srcP : Package
   { packagedElement = srcCls : Class { name=className }};
   enforce   domain target dstP : Package
   { packagedElement = dstCls : Class { name=className1}};
   when { st = getAppliedStereotype('UMLEJB::entity');
          srcCls.isStereotypeApplied(st); }
   where{ className1 = className + '_DerivedFromEntity'; }
}
```

In der **where**-Klausel wird erzwungen, dass der Name der Zielklasse um einen konstanten String ergänzt wird.

3.4 Primitive Domänen und Queries

Eine Relation kann beliebig viele **checkonly**- oder **enforce**-Domänen besitzen. Bei dem Aufruf von Relationen in **when**- oder **where**-Klauseln müssen dann entsprechend der Domänentypen und Reihenfolge Aufrufparameter übergeben werden. In dem obigen Beispiel besitzt die Classes-Relation zwei Domänen, jeweils eine **checkonly** und eine **enforce**. Dementsprechend müssen beim Aufruf dieser Relation zwei Objekte vom Typ Package als Parameter mitgegeben werden. Neben den Domänen mit einer Check- und Generierungsfunktion erlaubt QVT-R, dass weitere auch neutrale, sogenannte **primitive**, Domänen definiert werden. In dem folgenden Beispiel wird damit der gerufenen Relation eine zusätzliche String-Variable übergeben, die zur Erzeugung des Namens des Zielpaketes herangezogen wird.

```
relation ClassesWithPrefix
{
   cName, cName1  : String;
   checkonly domain source pckg : Package
   { packagedElement = srcCls : Class { name = cName1 } };
   enforce domain target schm : Package
   { packagedElement = dstCls : Class { name = cName } };
   primitive domain prefix : String;
   where { cName = if prefix = '' then cName1
                   else prefix + '_' + cName1 endif; }
}
```

Aufruf:

```
ClassesWithPrefix (srcPckg, dstPckg, 'HelloWorld');
```

Der einzige Zweck für primitive Domänen ist die Option, der gerufenen Relation weitere Parameter zu übergeben. Ein ähnliches Ergebnis kann man mit der Verwendung von Hilfsfunktionen (**query**) erreichen.

```
query PrefixToClassname ( prefix : String, cName : String )
: String
{
   if prefix = '' then cName
   else prefix + '_' + cName endif
}
```

Aufruf:

```
   cName = PrefixToClassname('HelloWorld', cName1);
```

4 Zusammenfassung und Ausblick

Die QVT-R ist eine interessante, verhältnismäßig einfache Sprache, die allerdings als deskriptive Sprache und aufgrund des konsequenten *Pattern Matching*-Ansatzes eher etwas gewöhnungsbedürftig ist. Dieser Beitrag konnte lediglich einen kleinen Einblick geben in die Transformation von Modellen mit Hilfe der QVT-R. Für tiefer gehendes Interesse sei auf die angeführte Literatur verwiesen, insbesondere [Nol09]. Mittlerweile ist die Werkzeugsituation stabil und weitgehend ausgereift, so dass man sagen kann, der MDA-Ansatz der Transformationen von Modellen nach Modellen funktioniert. So mag es durchaus interessant sein, die modellgetriebene Entwicklung von Anwendungssystemen auch in der Praxis um den Aspekt der Transformation von Modellen zu ergänzen, und damit die Phasenübergänge eines iterativen Entwicklungsprozesses gerade in den frühen Phasen zu unterstützen.

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

[Med10] mediniQVT; http://projects.ikv.de/qvt

[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 – Relations Language. Springer-Verlag, 2009

[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

[XMI07] OMG, MOF 2.0/XMI Mapping, Version 2.1.1. http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm, 2007