Neuer QVT-Standard will Modelltransformationen erheblich vereinfachen

Model-to-Model

VON CHRISTOPH SCHMIDT-CASDORFF

Die modellgetriebene Softwareentwicklung (MDSD) setzt sich zunehmend durch. Allerdings gab es bisher keinen offiziellen Standard, der den Übergang von einem Modell in ein anderes regelt. Diese Lücke hat die OMG (Object Management Group) jetzt mit QVT (Query View Transformation) geschlossen.

In der MDSD sind Modelle der zentrale Dreh- und Angelpunkt (siehe Kasten „Metamodell – was ist das?“). Ausgehend von einem Modell werden über unterschiedliche Transformationen ausführbarer Java-Code, XML oder andere Artefakte erzeugt. Es macht den Charme dieses Ansatzes aus, dass die inhaltlich entscheidenden Informationen auf hohem Abstraktionsgrad (dem Ausgangsmodell) erfasst werden. Durch Transformationen können Modelle um technologische und plattformspezifische Aspekte erweitert werden.

Für den Übergang zwischen Modellen hat die OMG die Sprache QVT (Query View Transformation) entwickelt und in diesem Jahr den dazugehörigen Standard verabschiedet. QVT ist Teil des MDA-Konzepts (Model Driven Architecture [1]), welches den Beitrag der OMG zur MDSD darstellt.

Am häufigsten sind in derzeitigen MDSD-Anwendungen Transformationen anzutreffen, die ein Ausgangsmodell in Java-Code, XML oder andere Artefakte transformieren, die nicht auf Metamodellen basieren (*Model-to-Text*-Transformation oder M2T). Oft reicht eine einzige Transformation nicht aus, daher werden Modelle schrittweise verfeinert und um technologische oder plattformspezifische Informationen angereichert. Am Ende dieser Transformationsstrecke steht das Modell, auf dem die M2T-Transformation aufsetzt. Eine Transformation zur Verfeinerung oder Anreicherung von Modellen setzt auf einem Quellmodell auf und erzeugt ein Zielmodell, welches wiederum als Ausgangsmodell für wei-

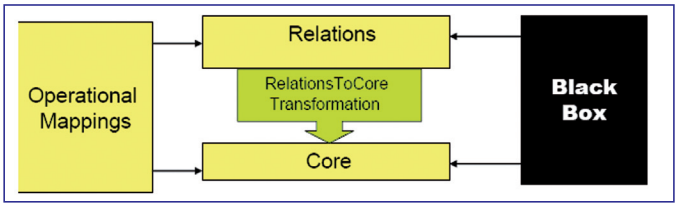

Abb. 1: Die von der Object Management Group definierte QVT-Architektur

tere Transformationen dienen kann. Solche Transformationen heißen *Model-to-Model*-Transformationen (M2M).

Architektur von QVT

Die QVT-Architektur gliedert sich in mehrere Bereiche (siehe QVT- Architektur). Der Bereich *Relation Language* bietet deklarative Sprachmittel (vergleichbar mit SQL), um so genannte Relationen zwischen Modellen zu definieren. Sie umfassen Konstrukte zum Musterabgleich zwischen Elementen des Quell- und Zielmodells, sowie Konstrukte, um Templates zur Erzeugung von Elementen des Zielmodells zu definieren. Diese so definierten Relationen werden in die so genannte *Core Language* überführt, ein weiterer Bereich der QVT-Spezifikation. Diese *Core Language* ist der *Relation Language* gleichzusetzen, ist jedoch einfacher und umfasst weniger Sprachelemente. In der QVT-Spezifikation wird als Metapher für das Verhältnis zwischen *Relation Language* und *Core Language* das Bild der *Java Virtual Machine* angeführt, in der die *Relation Language* die Rolle von Java und die *Core Language* die *Java Byte Code*s spielt.

Der Bereich *Operational Mapping*s stellt Sprachmittel zur Verfügung, mittels derer Transformationen mit imperativen Sprachkonstrukten (z.B. Java) definiert werden. Diese können einerseits Relationen ergänzen, wenn eine Transformation mit Mitteln der *Relation Language* nur unzureichend zu beschreiben ist, andererseits lassen sich auch Transformationen definieren, die nur auf *Operational Mapping*s basieren.

Mit *Black-Box*-Transformationen können Mengen von Elementen des Quell- und Zielmodells in Beziehung gesetzt werden, ohne diese im Rahmen der QVT genau zu beschreiben. Die Relation wird außerhalb der Sprachebene von QVT beschrieben und geht quasi an der QVT vorbei. Mit diesem Sprachmittel können externe Transformationen in QVT eingebunden werden. Im Spezifikationsumfang der OMG gibt es bereits eine Sprache, mit der auf Elemente von Modellen zugegriffen werden kann: die im Rahmen der UML bekannte OCL (Object Constraint Language). QVT setzt auf OCL auf, erweitert den Sprachumfang von OCL und formuliert eigene Sprachelemente zur Transformation von Modellen.

QVT und Eclipse

Neben der Spezifikation MOF der OMG hat sich mit EMF und dem ECORE eine zweite Ebene zur Metamodellierung entwickelt. ECORE ist annähernd kompatibel zu EMOF. Derzeit liegen im Umfeld von Eclipse nur Implementierungen des *QVT-Operational Mapping*s vor. Wir beschränken uns daher auf *QVT-Operational Mapping*, da wir Sie natürlich gerne in die Lage versetzen möchten, unsere Beispiele auszuprobieren, aber QVT auch in Ihren Projekten einzusetzen. Das von uns im Beispiel genutzte Werkzeug SmartQVT ist eine Open-Source-Implementierung dieses Standards (siehe Kasten „SmartQVT“). Es ist als Eclipse Plug-in implementiert und transformiert Metamodelle, die auf ECORE basieren. Unsere Beispiele finden Sie übrigens als Ressource auf der Magazin-CD einschließlich einer detaillierten Installationsbeschreibung.

Anfang 2007 wurde außerdem ein neues Eclipse-Projekt namens *M2M* als Subprojekt des *Eclipse Modelling Project*s aufgesetzt. Dieses Projekt beschäftigt sich mit *Model-to-Model*-Transformationen und dabei explizit mit QVT. Innerhalb dieses Projekts ist auch eine Komponente definiert, die sich mit deklarativer QVT

Metamodell – was ist das?

Metamodelle liefern die Sprachelemente, um Modelle zu definieren (Metasprache). So bedarf es z.B. eigener Sprachmittel, um ein UML-Modell zu beschreiben. Diese Sprachmittel, wie *Class*, *Attribute* usw., sind im UML-eigenen Metamodell hinterlegt. Dieses Spiel kann nun beliebig weit getrieben werden, denn das UML-Metamodell ist seinerseits ein Modell und benötigt wiederum Sprachmittel zur Beschreibung.

Die OMG hat zu diesem Zweck in ihrem Projekt MOF (Meta Object Facility) im Rahmen der MDA eine Hierarchie von Metamodellen und vor allem eine Metasprache EMOF (Essential MOF) zur Beschreibung aller Metamodelle festgelegt.

Das Eclipse Modelling Framework (EMF) ist eine Eclipse-spezifische Umsetzung der MOF-Konzepte. EMF basiert auf einer frühen Spezifikation der MOF. Die innerhalb von EMF entstandene Metasprache ECORE hat sich daher nicht parallel zu EMOF entwickelt. Obwohl sich beide in den aktuellen Spezifikationen MOF 2.0 und ECORE 2.3 stark angenähert haben, bestehen immer noch Unterschiede. Diese finden sich allerdings im Wesentlichen in der Benennung von Sprachelementen. Verkürzt lässt sich ECORE als Java-Implementierung von EMOF ansehen. In EMF 2.3 ist beispielsweise ein EMOF-Import/Export enthalten.

Im Kontext dieses Artikels werden nur Metamodelle betrachtet, die auf ECORE basieren. Zum Entwurf und zur Bearbeitung von Metamodellen dient in Eclipse EMF (aktuelle Version 2.3). Dort finden sich Editoren für ECORE, aber auch Generatoren, um das ECORE-Metamodell als Plug-in bereitzustellen. In Eclipse wird pro Metamodell ein eigenes Eclipse-Plug-in erstellt. Eine Einführung in EMF findet sich in [2].

SmartQVT

Das von uns im Beispiel genutzte Werkzeug SmartQVT ist eine Open-Source-Implementierung des *QVT-Operational Mappings* und wurde durch die France Telecom R&D entwickelt [3]. Es ist als Eclipse-Plug-in implementiert und transformiert Metamodelle, die auf ECORE basieren. Jede Transformation ist ein Eclise-Plug-in und besitzt eine separate *properties*-Datei. Dort wird insbesondere einem logischen Modellnamen die entsprechende URI des Metamodells zugeordnet. Dieser logische Modellname wird in der Transformation referenziert. Es fehlt allerdings in SmartQVT die aus dem Eclipse-Umfeld gewohnte Unterstützung für die Entwicklung, wie *Code Completion* oder gar ein *Debugger*.

(*Relation Language*) beschäftigt, so dass dort auch zugehörige Implementierungen zu erwarten sind.

QVT am Beispiel

Unser Beispiel ist der Spezifikation der QVT [4] entnommen. Implementierungen der Transformation aus [4] werden auch als Beispiel mit SmartQVT ausgeliefert. Die vorliegenden Listings basieren auf diesen Beispielen aus SmartQVT. SmartQVT liefert ebenfalls die Metamodelle für die Beispiele des Artikels.

Kenntnisse in *OCL* sind zum Verständnis unseres Beispiels nicht notwendig. Um eigene Transformationen zu entwickeln, sollten Sie aber über Grundkenntnisse verfügen. Eine gute und zu Beginn ausreichende Einführung in die wichtigsten Sprachelemente findet sich in [5, Kap.7].

Auf Basis eines vereinfachten UML-Metamodells (genannt *SimpleUML*) soll ein ebenfalls vereinfachtes RDBMS-Metamodell erstellt werden. In einem realen Szenario könnte ein solches RDBMS-Modell Ausgangsmodell für Generatoren für *DDL*-Skripte (*Data Definition Language*) sein. In den Abbildungen 2 und 3 finden sich die Klassendiagramme beider Metamodelle.

Eine persistente Klasse (Attribut *kind=='persistent'*) des *SimpleUML*-Modells wird in eine Tabelle, einen Primärschlüssel und eine identifizierende Spalte abgebildet. Ein identifizierendes Attribut einer persistenten Klasse (Attribut *kind=='primary'*) wird zu einem Objekt vom Typ *Key*, um das Primärschlüsselfeld zu beschreiben und Fremdschlüsselbeziehungen zu unterstützen.

Ein Attribut mit einem primitiven Datentyp wird zu einer Tabellenspalte. Attribute mit komplexem Datentyp (sprich: eine nicht persistente Klasse) werden in eine Menge von Spalten abgebildet, die der aufgefächerten Menge der Attribute des komplexen Datentyps entsprechen. Eine Assoziation zwischen zwei persistenten Klassen wird zu einer Fremdschlüsselbeziehung.

Im *SimpleUML*-Metamodell ist eine Klasse *PrimitiveDataType* definiert, welche primitive Datentypen repräsentiert. Um die Transformationen wesentlich zu

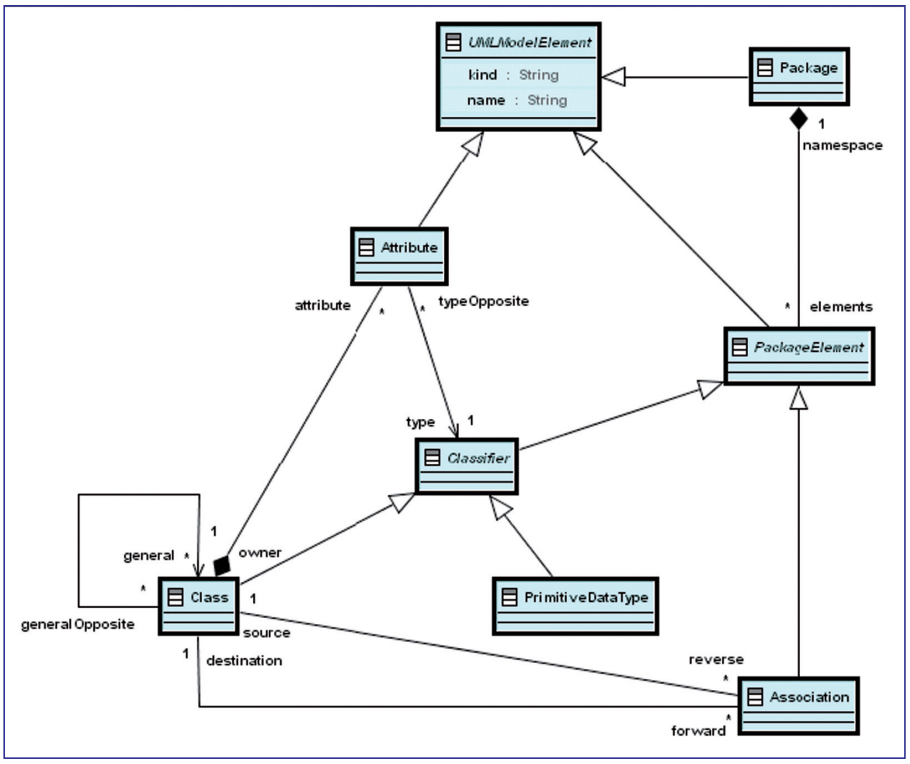

Abb. 2: Simple-UML-Metamodell

halten, sind nur die primitiven Datentypen *int* oder *string* zugelassen.

In einer ersten Transformation werden in Listing 1 Klassen in Tabellen und nichtkomplexe Attribute in Tabellenspalten überführt. Es werden weder komplexe Attribute noch Fremdschlüsselbeziehungen betrachtet. Die Lösung dieser Aufgabe heben wir uns für das weiterführende Beispiel auf.

Initialisierung der Transformation

In Zeile 3 des Listing 1 wird durch das Schlüsselwort *transformation* eine

Listing 1

```
01 -- Definition einer Transformation
02 -- Die referenzierten Metamodelle werden in
                    UML2RDBMS_simple.properties definiert
03 transformation iksgmbh_Uml2Rdb_simple
                    (in srcModel:UML,out dest:RDBMS);
04
05 -- Definition des Startpunkts der Transformation
06 -- Zuerst werden Tabellen aus Klassen erzeugt und
                                    anschliessend werden
07 -- Attribute der Klassen der zugehoerigen Tabellen
                                                ueberfuehrt
08 main() {
09   -- srcModel.objects()->select(oclIsKindOf
                    (UML::Class) )->map class2table();
10   srcModel.objects()[Class]->map class2table();
11 }
12
13 -- Aliases um Namenskonflikte zu vermeiden
14 tag "alias" RDBMS::Table::key_ = "key";
15
16 -- Klasse wird zu Tabelle mit einer Spalte pro
                                                primitivem Attribut
17 mapping Class::class2table () : t:Table
18 {
19   name := 't_' + self.name;
20   t.column := self.attribute->map attr2column();
21   t.key_  := object Key {
22        name := 'k_' + self.name;
23        column := t.column[kind='primary'];
24     };
25 }
26
27 mapping Attribute::attr2column () : Column
28   when {self.type.isKindOf(PrimitiveDataType); }
29 {
30   name := self.name;
31   kind := self.kind;
32   type := if self.type.name='int' then 'NUMBER' else
                                                'VARCHAR' endif;
33 }
```

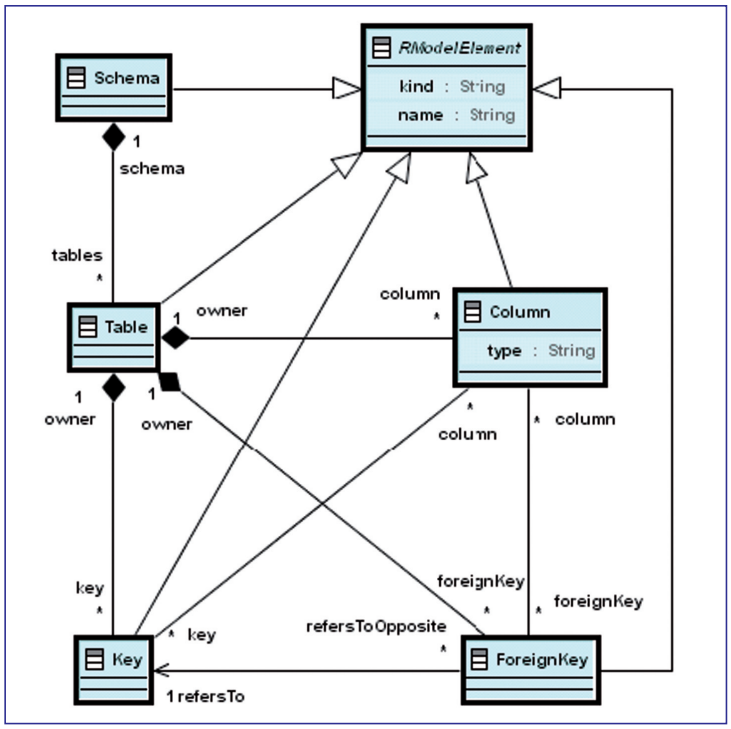

Abb. 3: Simple-RDBMS-Metamodell

Transformation deklariert und die entsprechenden Metamodelle angesprochen. Der Startpunkt der Transformation wird durch die *main*-Operation (Zeile 8) definiert. Dort wird eine erste Transformation (->*map*) angesprochen. Durch den Aufruf von *objects[Class]* wird die Menge aller Objekte des Typs *Class* aus dem Quellmo-dell ermittelt (*objects* ist eine Spracherweiterung der *OCL* durch *QVT*). Für alle Elemente dieser eingeschränkten Menge wird elementeweise *class2table()* aufgerufen. Die Elemente bezüglich derer die Transformation aufgerufen wird, werden als Kontext an die Operation übermittelt (siehe auch folgender Abschnitt „Kontextobjekte“).

Unsere Transformation soll aus einem Quellmodell ein Zielmodell erzeugen. Wo wird das Zielmodell aufgebaut, wo werden die neuen Elemente angelegt? In der Implementierung des *map*-Operators sind mehrere implizite Anweisungen von *QVT* versteckt, die wir im Weiteren kurz erläutern wollen. *map* iteriert über alle Elemente der Ergebnismenge von *objects()* und ruft für diese jeweils das *mapping* auf. Die Ergebnisse werden gesammelt und am Ende der Transformation als Gesamtergebnis bereitgestellt. Dieses Gesamtergebnis wird implizit im Zielmo-dell abgelegt. Die Ergebnisse mehrerer Aufrufe der *mapping*-Operation werden im Zielmodell aggregiert.

Listing 2

```
mapping X::mappingname (p1:P1, p2:P2) : r1:R1, r2:R2
  when { ... }
  where { ... }
{
  init { ... }
  population { ... }
  end { ... }
}
```

Beschreibung einer mapping-Operation

Eine *mapping*-Operation (oder auch kurz *mapping*) ist ein typisches QVT-Sprachelement, in dem sowohl deklarative à la SQL als auch imperative Sprachelemente genutzt werden. Sie sind die eigentlichen Arbeitstiere in einer QVT-Transformation. In Listing 2 finden Sie die (etwas verkürzte) Struktur einer *mapping*-Operation.

Durch *X* wird der so genannte Kontexttyp definiert. Der aus *OCL* bekannte Begriff des Kontextes definiert dasjenige Element des Quellmodells, für welches das *mapping* definiert wurde. Ebenfalls können dem *mapping* Parameter übergeben werden.

Die Ausgaben (und damit die eigentliche Transformation) der Transformation werden im *population*-Abschnitt beschrieben. Dieser ist damit das Herzstück einer Transformation. Dieser wird umrahmt von einem Initialsierungsabschnitt, in dem Berechnungen und Initialisierungen vor der Ausgabe durchgeführt werden und einem finalen Abschnitt, in dem Abschlussarbeiten erledigt werden. Zwischen dem *population*- und *init*-Abschnitt wird eine Phase impliziter Instanzierung ausgeführt, in der alle Ausgabeparameter erzeugt werden, die nach dem *init*-Abschnitt nicht initialisiert sind. Zusammengefasst werden im Ablauf eines *mappings* die einzelnen Phasen in folgender Reihenfolge durchlaufen: *init*, Instanzierung, *population*, *end*. In der Regel wird das Schlüsselwort *population* ausgelassen und stattdessen nur der Codeblock notiert. Mithilfe von *when* werden Bedingungen definiert, unter denen das *mapping* ausgeführt wird (*Guard Condition*). Unter *where* können Nachbedingungen definiert werden (*Postcondition*).

In einem *mapping* stehen die Schlüsselworte *this*, *self*, und *result* zur Verfügung. *this* repräsentiert die Instanz der Transformation und es kann auf Eigenschaften oder Operationen der Transformation zugegriffen werden. In einer kontextuellen Operation (*mapping/query*) repräsentiert *self* das Kontextobjekt. *result* bezieht sich auf die Resultatobjekte (Ausgabeobjekte). Ein *mapping* kann mehrere Resultate liefern.

class2table() in Listing 1 besteht nur aus einem *population*-Abschnitt. In der Instanzierungsphase wird ein Ausgabeobjekt *t* vom Typ *Table* angelegt, welches in *population* zur Verfügung steht. Da dieses *mapping* nur ein Ausgabeobjekt erzeugt, kann auf dessen Eigenschaften unqualifiziert zugegriffen werden. So können Attribute des Ausgabeobjekts unqualifiziert (Zeile 19) oder über den deklarierten Namen des Ausgabeobjekts (Zeile 20, 21) angesprochen werden.

In Zeile 20 ist beispielhaft die Belegung eines Attributs des Zielobjekts durch Aufruf eines weiteren *mappings* zu sehen. Die Zuweisung der Eigenschaft *key* des Zielobjekts (Zeile 21) zeigt zwei weitere Sprachelemente von QVT. Erstens ist die Originalbezeichnung *key* ein Schlüsselwort in QVT und daher wird ihm ein so genannter *alias key_* zugewiesen. Zweitens wird das Attribut *key_* nicht durch den Aufruf eines weiteren *mappings* erzeugt (was möglich wäre), sondern innerhalb des *mappings* mit dem Sprachkonstrukt *object expression* angelegt. Man bezeichnet diese Konstrukte auch als *inline mapping*.

Weiterführende Sprachelemente in QVT

Nachdem wir in Listing 1 eine einfache Transformation eines (vereinfachten) UML-Modells in ein Modell zur Beschreibung einer Datenbank vorgestellt haben, werden wir in einer weiterführenden Transformation komplexe Attribute und Fremdschlüsselbeziehungen behandeln. Die dazu notwendigen Sprachelemente von QVT werden schrittweise anhand des Listings 3 eingeführt.

Query

Eine Hilfsoperation (*helper*) führt eine oder mehrere Berechnungen auf Elementen des Quellmodells aus und liefert Ergebnisse. Während im Allgemeinen eine *helper*-Operation Seiteneffekte auf Kontextobjekt und/oder Parameter haben kann, sind *query*-Operationen spezielle *helper*, die keine Nebeneffekte erzeugen. Die Zeilen 19-20 in Listing 3 zeigen ein Beispiel für eine *query*-Operation.

Variablen

Mittels des Schlüsselwortes *var* lassen sich Variablen definieren. Werden sie im Rahmen einer Operation (z.B. *query*, *mapping*) definiert, so ist ihr Gültigkeitsbereich auf diese Operation beschränkt (s. *attr2LeafAttrs()*). Globale Variablen werden außerhalb von Operationen definiert.

Intermediate Data

Transformationen benötigen unter gewissen Umständen Zwischenklassen

Listing 3

```
01 --modeltype UML uses SimpleUml;
02 --modeltype RDBMS uses SimpleRdbms;
03 transformation iksgmbh_Uml2Rdb
                    (in srcModel:UML,out dest:RDBMS);
04
05 -- Aliases um Namenskonflikte zu vermeiden
06 tag "alias" RDBMS::Table::key_ = "key";
07
08 -- Zwischenstrukturen,um Attribute von komplexen
                                          Datentypen
09 -- zu referenzieren
10 intermediate class LeafAttribute {
11  name:String;
12  kind:String;
13  attr:UML::Attribute;
14 };
15 intermediate property UML::Class::leafAttributes :
                              Sequence(LeafAttribute);
16
17
18 -- Definition der genutzten Queries
19 query UML::Association::isPersistent() : Boolean =
20   (self.source.kind='persistent' and self.destination.
                                    kind='persistent');
21
22
23 -- Startpunkt der Transformation
24 -- Tabellen werden auf Basis von Klassen erzeugt und
                                          anschliessend
25 -- die Fremdschluesselbeziehungen hinzugefuegt, die
                                    durch ihre Assoziationen
26 -- impliziert werden
27
28 main() {
29  srcModel.objects()[Class]->map class2table();
                                          -- first pass
30  srcModel.objects()[Association]->map
                              asso2table(); -- second pass
31 }
32
33 --Klasse pro Tabellen und eine Spalte pro Attribute
34 mapping Class::class2table () : Table
35  when {self.kind='persistent';}
36 {
37  init {
38   -- Zwischenstrukturen werden initialisiert
39   self.leafAttributes := self.attribute
40         ->map attr2LeafAttrs("","")->flatten();
41  }
42
43  name := 't_'+ self.name;
44  column := self.leafAttributes->map
                          leafAttr2OrdinaryColumn("");
45  key_ := object Key {
46       name := 'k_'+ self.name; column :=
                        result.column[kind='primary'];
47     };
48 }
49
50 -- Mapping bewertet den Typ von Attributen und
                          erzeugt Zwischenstrukturen
51 mapping Attribute::attr2LeafAttrs
                    (in prefix:String,in pkind:String)
52   : Sequence(LeafAttribute) {
53  init {
54   var k := if pkind="" then self.kind else pkind endif;
55   result :=
56    if self.type.isKindOf(PrimitiveDataType)
57    then
58     -- erzeugt Sequence von LeafAttribute
59     Sequence {
60      object LeafAttribute
          {attr:=self;name:=prefix+self.name;kind:=k;}
61     }
62    else self.type.asType(Class).attribute
63     ->map attr2LeafAttrs(self.name+"_",k)->flatten()
64    endif;
65  }
66 }
67
68 -- Aus jedem Eintrag der Zwischenstruktur wird eine
                                          Spalte
69 mapping LeafAttribute::leafAttr2OrdinaryColumn
                          (in prefix:String): Column {
70  name := prefix+self.name;
71  kind := self.kind;
72  type := if self.attr.type.name='int' then 'NUMBER'
                          else ,VARCHAR' endif;
73 }
74
75 -- Hinzufuegen der Fremdschluesselbeziehungen zu
                                    einer Tabelle
76 mapping Association::asso2table() : Table
77  when {self.isPersistent();}
78 {
79  init {result := self.destination.resolveone(Table);}
80  foreignKey := self.map asso2ForeignKey();
81  column := result.foreignKey->column->flatten();
82 }
83
84 -- Aufbau der Fremdschluessel
85 mapping Association::asso2ForeignKey() : ForeignKey {
86  name := 'f_'+ self.name;
87  refersTo := self.source.resolveone(Table).key_;
88  column := self.source.leafAttributes[kind='primary']
89       ->map leafAttr2ForeignColumn
                          (self.source.name+'_');
90 }
91
92 -- Erzeugt einen Fremdschluessel aus einem
                                    LeafAttribute
93 -- Erbt von leaf2OrdinaryColumn und daher wird
                          leaf2OrdinaryColumn vor
94 -- dem eigenen population-Abschnitt gerufen.
95 mapping LeafAttribute::leafAttr2ForeignColumn
                          (in prefix:String) : Column
96 inherits leafAttr2OrdinaryColumn {
97  kind := "foreign";
98 }
```

oder - eigenschaften des Quellmodells. In unserem Beispiel heißen Attribute, die weder einen primitiven Typ haben noch auf persistente Klassen zeigen, komplex. Ist ein Attribut komplex, so werden alle Attribute des Zieltyps zu Spalten der Zieltabelle. Dazu werden in einer Zwischenstruktur *LeafAttribute* des Quellmodells, alle primitiven Attribute und alle Attribute des Zieltyps eines komplexen Attributs gesammelt. Auf Basis dieser Sammlung werden die Spalten aufgebaut. Es wird eine Zwischenklasse *LeafAttribute* definiert und an Elemente vom Typ *Class* gebunden. Durch diese wird das Quellmodell während der Transformation erweitert. *attr2LeafAttrs()* sammelt alle Attribute einer Klasse, welche zu Spalten der zugehörigen Tabelle werden sollen. Dazu wird in der Initialisierungsphase von *class2table()* die Zwischenstruktur *leafAttribute* initialisiert und steht somit in *attr2LeafAttrs()* zur Verfügung. Bei komplexen Attributen wird rekursiv *mapping* aufgerufen.

attr2LeafAttrs() demonstriert ebenfalls, dass in der Initialisierungsphase ohne *populations*-Abschnitt die vollständigen Ergebnisse eines *mappings* geliefert werden können.

Auflösung von Objektreferenzen

Während der Ausführung der Transformation werden zu jedem ausgeführten *mapping* das Kontextobjekt, die Parameter und das Ergebnis protokolliert (so genannte *trace*). Mit diesen Informationen kann ermittelt werden, welches Element des Zielmodells durch welches Element des Quellmodells erzeugt wurde und vice versa.

In *asso2ForeignKey()* wird zu einer Fremdschlüsselbeziehung in der Zieltabelle ein Element vom Typ *ForeignKey* erzeugt. Dazu sind diejenigen Tabellen zu finden, die durch die Quell- und Zielklasse der Assoziation erzeugt wurde. In der Initialisierungsphase von *asso2table()* wird mittels *resolveone()* die der Quellklasse zugeordnete Tabelle ermittelt und als Ausgabeobjekt definiert. In *asso2ForeignKey()* wird ermittelt, auf welches Element vom Typ *Key* der Zieltabelle sich der *ForeignKey* bezieht.

Ein spannender Einsatz des *trace* ist die verzögerte Auflösung von Referenzen (*late resolution*). Bei nicht verzögerter Auflösung referenziert ein Quellelement das durch dieses Quellelement erzeugte Zielelement. Bei verzögerter Auflösung referenziert das Quellelement dasjenige Zielelement, das in noch auszuführenden *mappings* erzeugt werden wird. Die Referenz wird nicht zum Zeitpunkt des Aufrufs, sondern am Ende der gesamten Transformation ausgewertet.

Konzepte zur Wiederverwendung

Auf der Ebene von *mappings* bietet QVT zwei Konzepte zur Wiederverwendung an: Vererbung (*inheritance*) und Verkettung (*merge*). Erbt ein *mapping*, so wird das vererbende *mapping* nach der Initialisierungsphase des erbenden *mappings* gerufen. *leafAttr2ForeignColumn()* zeigt ein Beispiel für Vererbung von *mappings*.

Ein *mapping* kann eine Liste von *mappings* deklarieren (*merge declaration*), die seine Verarbeitung vervollständigen. Die (*merge*-) Liste wird nach dem *end*-Abschnitt abgearbeitet.

Fazit

Mit QVT ist eine kompakte und leistungsstarke Sprache zur *M2M*-Transformation entstanden und stellt eine ernst zu nehmende Alternative zu Java-basierten Transformationen dar. Mit QVT eröffnen sich neue Möglichkeiten in einem MDSD-Entwicklungsprozess. Obwohl es in der Regel eine mühselige Aufgabe ist, sich mit OMG-Spezifikationen auseinander zu setzen, möchte ich Ihnen die Lektüre von [5] Kap. 8 zur Vertiefung empfehlen.

Mit SmartQVT steht außerdem eine Open-Source-Implementierung zur Verfügung, die aber noch Wünsche hinsichtlich des Bedienkomforts für die Entwicklung offen lässt. Es ist zu hoffen und zu erwarten, dass im Rahmen des Eclipse-M2M-Projekts leistungsstarke Implementierungen der QVT-Standards und moderne Entwicklungsumgebungen für QVT entstehen.

Sind Sie in OCL bewandert, erschließen sich die Konzepte von QVT schnell, und auch eigene komplexe Transformationen sind schon bald möglich. Der sinnvolle Einsatz von QVT setzt allerdings voraus, dass in Ihrem Umfeld ein MDSD-Prozess definiert ist, in dem QVT seinen Platz finden kann.

Christoph Schmidt-Casdorff ist als IT-Berater bei der iks Gesellschaft für Informations- und Kommunikationssysteme tätig. Er beschäftigt sich seit mehreren Jahren in großen Kundenprojekten mit dem Thema der modellgetriebenen Softwareentwicklung.
Kontakt: c.schmidt-casdorff@iks-gmbh.com

Links & Literatur

[1] Frankel, David:Model Driven Architecture, Wiley Publishing 2003

[2] dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html

[3] Meta Object Facility (MOF) 2.0 Core Specification: www.omg.org/docs/formal/06-01-01.pdf

[4] QVT 1.0 Final Specification: www.omg.org/docs/ptc/07-07-07.pdf

[5] OCL 2.0 Specification: www.omg.org/docs/formal/06-05-01.pdf

[6] SmartQVT Home Page: smartqvt.elibel.tm.fr

[7] Eclipse Modelling Project: M2M: www.eclipse.org/m2m/