

Eine praxisnahe Evaluierung von QVT am Beispiel von UML und XML

Konrad Voigt

HUMBOLDT-UNIVERSITÄT ZU BERLIN



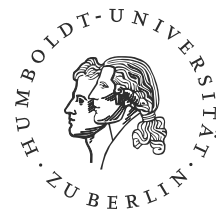
Eine praxisnahe Evaluierung von QVT am Beispiel von UML und XML

Konrad Voigt (konrad.voigt@gmail.com)

Lehrstuhl für Systemanalyse
Institut für Informatik
Humboldt-Universität zu Berlin

Berlin, März 2006

HUMBOLDT-UNIVERSITÄT ZU BERLIN



Inhaltsverzeichnis

1	Einleitung	1
1.1	Einführung	1
1.2	Motivation und Zielstellung	3
1.3	Entwicklung von QVT	4
2	QVT	5
2.1	Einführung	5
2.2	Architektur	7
2.3	Relationale Transformation	8
2.4	Operationale Transformation	14
2.5	Core	16
2.6	BlackBox	17
2.7	Verfügbare Werkzeuge	17
2.8	Zusammenfassung	18
3	Transformation von UML nach XML mit QVT	18
3.1	Einführung	18
3.2	Transformationsregeln	19
4	Verwandte Arbeiten zu QVT	26
5	Zusammenfassung und Schlussfolgerung	27

1 Einleitung

1.1 Einführung

Softwaresysteme und Softwarearchitekturen weisen in den letzten Jahren eine immer größere Komplexität auf, welche durch steigende Anforderungen und vielseitige Einsatzgebiete zu begründen ist. Ein Ansatz, um dies zu bewältigen, ist die Modellierung. Durch abstraktes Analysieren des Systems wird der Zugang erleichtert und der Austausch zwischen Anwender und Entwickler vereinfacht. Um eine effiziente Modellierung zu gewährleisten, wurden verschiedene Methoden und Ansätze entwickelt. Die wohl bekannteste stellt UML [UML, 2003] die Unified Modeling Language dar. Bei der Entwicklung dieser Sprache lag das Hauptaugenmerk auf einer verständlichen und austauschbaren Spezifikation von Modellen aus verschiedenen Anwendungsgebieten.

Die unter der Führung der OMG (Object Management Group) definierte MDA [MDA, 2003] (Model Driven Architecture) führt diesen Gedanken einen Schritt weiter und rückt die Modelle an sich ins Zentrum. Ein jedes Modell wird als Softwarekomponente aufgefasst und auch als solche behandelt. Das führt zu den Möglichkeiten der Komponierbarkeit wie auch Wiederverwendbarkeit. Die Idee der Model Driven Architecture beruht auf dem Prinzip der plattformunabhängigen Entwicklung und der Überführung der Spezifikation in ein oder mehrere plattformspezifische Modelle. Ein Softwaresystem wird durch ein formales Modell spezifiziert, aus dem durch automatisierte Transformationen die ausführbaren Softwarekomponenten generiert werden. Dazu wird zwischen plattformunabhängigen Modellen (PIM – Platform Independent Model) und plattformspezifischen Modellen (PSM – Platform Specific Model) unterschieden. Das PIM stellt das Modell dar und ist für die gesamte Entwurfsphase ausschlaggebend. Um jedoch die eigentliche Realisierung durchzuführen, müssen die PIMs in ein oder mehrere PSMs transformiert werden, so dass letztendlich die ausführbaren Softwarekomponenten generiert werden können. Hierbei kann ein einziger Schritt zur Überbrückung der Lücke zwischen PIM und PSM nicht ausreichen. Das bedeutet, es sind Zwischenmodelle und Repräsentationen notwendig, um modellnah zu transformieren.

Der Gedanke der MDA wird erst durch Transformationen konsequent umgesetzt, denn erst durch eine Modell-zu-Modell-Transformation ist eine hinreichende Modellüberführung möglich. Das führt zu einer Vielzahl spezifischer Transformationsdialekte zur Beschreibung des jeweiligen Zwischenschritts, welches wiederum eine erhöhte Komplexität und Aufwand zur Folge hat.

Durch den vom Lehrstuhl für Systemanalyse an der Humboldt-Universität zu Berlin geprägten Begriff der ULF-Ware [Fischer et al., 2005, 2004] wurde eine auf Metamodellen basierende Sprachumgebung definiert. Das Ziel von ULF ist die Vereinfachung von benötigten Metamodelldefinitionen und entsprechender Werkzeugentwicklungen und wird erreicht durch eine spürbar höhere Wiederverwendbarkeit von Konzepten und generischen Werkzeugen bzw. Metawerkzeugen. ULF-Ware selbst kann als Instanz der MDA verstanden werden, da es nicht nur die Definition von PIMs und PSMs ermöglicht und unterstützt, sondern auch deren Transformation bzw. Überführung. Man hat die Möglichkeit, den gesamten MDA-Prozess durchzuführen und letztlich auch die Codege-

nerierung zu nutzen. UML-Ware selbst arbeitet metamodellbasiert und vereint Werkzeuge für verschiedene Einsatzzwecke, wie Modellüberprüfung, Codegenerierung, Codeparasing und Modelltransformation. Der letzte Punkt legt die Benutzung einer formal basierten und standardisierten Sprache nahe, welche einen generischen Ansatz zur Transformation bereitstellt. Da es bisher keine Sprache gab, die diese Kriterien erfüllt, scheint QVT – eine metamodellbasierte, generische Transformationssprache – eine viel versprechende Möglichkeit zu sein.

Da neben der Syntax, der graphischen Bearbeitung, der Semantik und der Codegenerierung auch die Transformation eine Rolle spielt, ist hier ein Einsatzgebiet für QVT gegeben, um metamodellbasierte Sprachen beliebig zu transformieren und den generischen Einsatz von QVT in der Praxis zu erproben.

Zur Modelltransformation gibt es viele Ansätze und Techniken. Czarnecki und Helsen [Czarnecki u. Helsen, 2003] beschäftigen sich mit verschiedenen Konzepten und Möglichkeiten. Sie kommen zu dem Schluss, dass die Modelltransformation – als neues und junges Feld im Gegensatz zu den traditionellen Feldern der Programm-Transformation – noch keine befriedigende Lösung zur Transformation bereitstellt. Die verschiedenen Ansätze zur Modell-zu-Modell-Transformation wie z. B. direkte Manipulation, relationaler Ansatz und der graphenbasierter Ansatz sowie hybride Formen besitzen Vor- wie auch Nachteile. Jedoch ist für keinen von ihnen ein expliziter Standard für die Modelltransformation definiert.

Im Jahre 2003 startete die OMG einen Aufruf¹ als eine erste Initiative, die modellbasierte Transformation zu standardisieren. Das Ziel ist eine plattformunabhängige Modell Transformationssprache mit dem Namen QVT. Die sich daraus ergebende Überarbeitung zu einer ersten Fassung des Standards wie ihre nochmalige Revidierung resultieren in einem hybriden Transformationsansatz, der eine variable und angepasste Möglichkeit der Transformation ermöglicht. Durch einen langen und intensiven Diskussionsprozess ist die präfinale Version des Standards² – QVT – entstanden.

Im Rahmen dieser Studienarbeit setze ich mich mit den Konzepten und Strukturen von QVT auseinander und untersuche seinen praktischen Einsatz an einem praxisnahen Beispiel. QVT zerfällt in drei Sprachbestandteile (Transformationsdialekte), diese habe ich zur praktischen Umsetzung der Abbildung herangezogen. Die drei Sprachbestandteile werden von mir angewendet und abschließend bewertet. Nach einer Einführung in die Struktur der Konzepte von QVT und einer Erläuterung der sprachlichen Bestandteile werden die Abbildungsregeln und deren Umsetzung beschrieben. Abschließend diskutiere und bewerte ich den praktischen Einsatz von QVT und gehe auf Alternativen ein. Zuerst fasse ich die Konzepte und Strukturen von QVT im Abschnitt 2 zusammen, in einer kurzen Einführung erläutere ich das Grundkonzept von QVT und gehe dann näher auf seine Struktur ein, dann beschreibe ich die Untersprachen von QVT und stelle ihre Zusammenhänge dar. Im Anschluss beschreibe ich in Abschnitt 3 exemplarisch die praktische Abbildung von UML nach XML-Schema mit Hilfe der jeweiligen Sprachkon-

¹RFP – Request For Proposal

²Formal gesehen ist es eine Empfehlung (Recommendation), doch aufgrund der Quasistandard Natur benutze ich im Folgenden das Wort Standard anstelle von Empfehlung.

zepte. Nach der Betrachtung verwandter Arbeiten im Abschnitt 4 fasse ich am Ende in Abschnitt 5 die Arbeit zusammen, um dann meine Schlussfolgerung zu ziehen.

Diese Studienarbeit ist kein Vergleich existierender Ansätze oder eine Zusammenfassung von QVT, auch erwartet den Leser keine vollständige Abbildung aller UML-Konzepte. Vielmehr wird hier die praktische Anwendung von QVT gezeigt, sowie dessen Konzepte zusammenfassend erläutert.

1.2 Motivation und Zielstellung

Die vorliegende Version von QVT setzt sich als Ziel, eine allgemeine und flexible Transformationssprache zu definieren. Ziel meiner Arbeit ist es, die Konzepte von QVT zu erläutern, zusammenzufassen und allgemeine Probleme zu identifizieren. Das Hauptaugenmerk liegt auf dem Einsatz von QVT bei allgemeinen Transformationsproblemen und -mustern sowie der Beurteilung deren Lösungen mit Hilfe von QVT. So wird ersichtlich, welches Vorgehen mit Hilfe von QVT möglich und empfehlenswert ist, um häufig auftretende und allgemeinen Problemstellungen (bei Transformationen) zu lösen. Auch gilt es den Einsatz von QVT und existierenden Werkzeugen für den Einsatz im Kontext der ULF-Ware zu evaluieren.

Wie in der Einleitung erwähnt, definiert ULF-Ware eine generische Umgebung zur metamodellbasierten Transformation. Hierbei treten verschiedene Anwendungsszenarien auf: So sind neben der Modell-zu-Modell-Transformation, auch die Überführung von konkreter zur abstrakter Syntax, die Transformation der Semantik und die Modell-zu-Programm-Transformationen in ULF-Ware enthalten. Hier wäre eine generische und skalierbare Transformationssprache der Schlüssel zu einem einheitlichen und fundierten Transformieren. Im Rahmen meiner Studienarbeit werde ich überprüfen, inwiefern QVT sich theoretisch wie auch praktisch in ULF einsetzen und verwenden lässt.

Um QVT bewerten zu können, habe ich basierend auf meinen bisherigen Erfahrungen in der manuell implementierten Transformation [Böhme et al., 2005] von eODL[ITU-T Z.130, 2003] nach CIDL [CIDL, 2002] und dem Beitrag von Czarnecki und Helsen (siehe auch Kapitel 4) folgende Kriterien identifiziert. Eine Transformationssprache, die im Bereich der MDA eingesetzt wird, sollte die Abbildung wie auch Transformation von einer Menge von Modellen auf eine Menge von Modellen unterstützen, denn dies ist zentraler Punkt von MDA. Sie sollte Konzepte zur Verfügung stellen, die es ermöglichen, jede gewünschte Konstellation innerhalb eines Modells zu erfassen und abzubilden. Zur Nachverfolgbarkeit wie auch der inkrementellen Transformation sollte ein Modell definiert sein, welches dies unterstützt. Als Transformationssprache sollte sie eine formale Basis besitzen, um entsprechend formale Transformationen definieren zu können. Des Weiteren sollte sie skalierbar sein, um in unterschiedlichen Transformationsszenarien performant zu sein. Diese Kriterien legen den Rahmen für die Sprachdefinition an sich fest, doch um sie auch tatsächlich einzusetzen, sind Werkzeuge unabdingbar, welche erst den Mehrwert der Sprache erbringen. Daher gilt es auch, verfügbare Werkzeuge zu untersuchen und zu bewerten.

Zur praktischen Anwendung der Transformation habe ich das Beispiel der Transformation von UML [UML, 2003] nach XML-Schema [XML-Schema, 2000] gewählt. UML

*Bewertungs-
kriterien*

*UML und
XML*

definiert als Modellierungssprache eine graphische Repräsentation, um die verschiedenen Aspekte wie z. B. Verhalten und Strukturen eines Systems zu modellieren. Hier konzentriere ich mich auf die Klassendiagramme von UML, welche die statische Systemstruktur beschreiben, um sie auf XML-Dokument abzubilden. XML ist eine Dokumentenbeschreibungssprache, welche ein lesbares wie auch maschinenverarbeitbares Format definiert. Eine Spezialisierung dieses Konzept stellt XML-Schema dar, welches dazu dient, eine Menge von XML-Dokumenten zu beschreiben und zu definieren, indem man Restriktionen auf Elemente und deren Struktur spezifiziert. Im Gegensatz zu XMI [XMI, 2002](XML Metadata Interchange)³, als ein reines Austauschformat, ist XML-Schema eine Schablone zum Erstellen von beliebigen XML-Dokumenten⁴.

Die Wahl fiel auf beide Sprachen, da auf der einen Seite, UML wie auch XML weit verbreitete und bekannte Technologien darstellen und somit die Verständlichkeit erhöht ist. Auf der anderen Seite ist für beide Sprachen ein leicht verständliches Metamodell verfügbar, so dass eine Implementierung der Abbildung denkbar flüssig vonstatten geht. Im Kontext von ULF wird unter anderem UML als Ausgangssprache für Transformationen benutzt, um so z. B. Modelle abzubilden. Um diese zu speichern wird der XML-Dialekt XMI benutzt. Selbstverständlich müssen hierzu die Modellelemente entsprechend abgebildet werden, so dass die Abbildung von UML nach XML-Schema einem ähnlichen Prinzip folgt.

Die Abbildung von UML nach XML beruht zu großen Teilen auf den Arbeiten von Dr. Eckhardt Eckstein und Dr. Silke Eckstein, welche in ihrem Buch [Eckstein u. Eckstein, 2004] und der Vorlesung XML und Semantic Web die entsprechenden Abbildungsregeln beschreibt.

1.3 Entwicklung von QVT

Nach einer Initiative der OMG und dem folgenden Aufruf (RFP) gab es im Jahr 2003 von acht verschiedenen Gruppierungen Vorschläge. Wie auch der OMG-Webseite zu entnehmen waren es im Einzelnen: Adaptive Ltd., DSTC/IBM, Alcatel/Softimeam/TNI-Valiosys/Thales, Compuware Corporation/Sun Microsystems, Kennedy Carter, TCS welches Artisan Software, Kinetum, Kins College und die Universität von New York vertritt, Codagen Technologies Corporation und Interactive Objects Software GmbH/Project Technology.

In den darauf folgenden Jahren wurden diese Vorschläge intensiv geprüft und revidiert, insbesondere Gardner et al. diskutierten in ihrem Beitrag [Gardner et al., 2003] die Vor- und Nachteile der einzelnen Standpunkte und kamen zu dem Schluss, dass ein hybrides System den Anforderungen einer flexiblen Transformationssprache gerecht werden würde. Sie schlagen einen deklarativen Sprachbestandteil für die Definition von einfachen Transformationen und einen imperativen für komplexere vor.

Im August 2003 wurde die erste Version [QVT, 2002] als zu diskutierender Vorschlag

Erster Entwurf

³Ein von der OMG auf der Basis von XML-Schema definiertes Austauschformat von MOF-basierenden Metamodellen, wie daraus resultierender Modelle.

⁴Der Unterschied wird offensichtlich, wenn man bedenkt, dass XMI mit Hilfe von XML-Schema definiert ist.

veröffentlicht. Die dort definierten Konzepte wurde daraufhin verfeinert, so dass seit April 2005 die erste Version eines finalen Standards [QVT, 2005] OMG Mitgliedern zu Verfügung steht. Die erste Version des QVT Standards traf zuerst eine Unterscheidung der Transformation in Relationen und Abbildungen. Eine Relation wurde als Beziehung zwischen Modellen verstanden, deren Gültigkeit verifiziert werden kann. Mit dem Konzept der Verfeinerung erweiterte man die Relation um modellrelevante Aspekte und erhielt so eine Transformation. QVT wurde als rein relationale musterbasierte Sprache definiert, die eine textuelle und eine graphische Repräsentation besitzt. Dieser Vorschlag wurde positiv aufgenommen, wies jedoch Schwächen auf. So gab es nur lose Aussagen über die Implementierung. Auch war die Sprache mit ihrem nicht deklarativen Charakter an vielen Stellen nicht intuitiv. *Queries* wie auch *Views* wurden über OCL 2.0 [OCL, 2003] definiert und beschrieben.

Basierend auf dieser Arbeit wurde in der Überarbeitung vollständig auf die Klassifizierung von Transformation in Relation und Abbildung verzichtet. Zentraler Bestandteil ist die Transformation als solche, welche nun manipulierend oder überprüfend vorgenommen werden kann, spezifizierbar für jedes an der Transformation teilnehmende Modell. Auch wurde der Sprachumfang erweitert. Neben der aufgenommenen und leicht modifizierten relationalen Sprache wurde eine neue Sprache eingeführt. Die erste, eine deklarative an Programmiersprachen angelehnte Sprache, soll die Brücke zwischen der Implementierung und der Spezifikation schlagen und die Defizite der relationalen Sprache ausgleichen.

2 QVT

2.1 Einführung

QVT (Query View Transformation) ist ein kommender Standard einer metamodellbasierten Transformationsprache. Man definiert in QVT Transformationsregeln zwischen verschiedenen Sprachen auf der Basis ihrer Metamodelle, um dann entsprechende Instanzen davon zu transformieren. Die Konzepte Modell und Metamodelle beruhen auf dem von der OMG gegebenen Standard *Meta Object Facility* (MOF) [MOF, 2003], welcher sie definiert und spezifiziert. Während ein Modell eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems ist, beschreibt das Metamodelle eine Menge von Modellen und abstrahiert sie somit.

QVT basiert auf drei Konzepten, welche gleichwohl die von der OMG definierten Anforderungen an eine Transformationsprache darstellen. Erstens: Anfragen auf MOF-2.0 Modellen, das bedeutet, aus einem gegebenen Modell wird eine Menge von Modellelementen ausgewählt, welche dann zur Weiterverarbeitung bereitgestellt wird. Die resultierende Menge ist neues Modell und eine Teilmenge des ursprünglichen Modells, somit wirken sich Änderungen an dieser auch auf das ursprüngliche Modell aus. Im Kontext von UML wäre eine exemplarische Anfrage: „Wähle alle Klasse mit nur einem Attribut und dem Namen »B« aus.“ Ein Beispiel für eine Anfragesprache ist OCL 2.0 [OCL, 2003].

Query

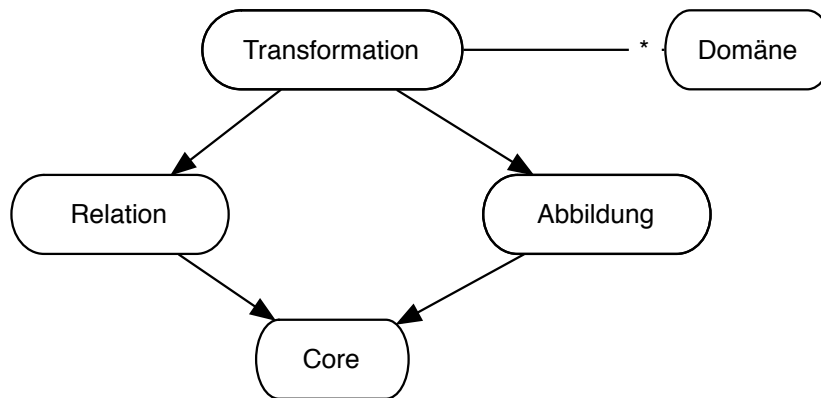


Abbildung 1: QVT Konzeptbeziehung

Zweitens: In den Anforderungen werden des Weiteren Sichten auf MOF-2.0-Metamodelle gefordert, das heißt ein *View* ist ein Modell, welches aus einem Ursprungsmodell abgeleitet wurde. Die Sicht ist dem Quellmodell fest zugeordnet, sollte also Änderungen in ihr vorgenommen werden, so wirkt sich das auch auf das Quellmodell aus. Das Metamodelle des durch die Sicht entstandenen Modells ist typischerweise unterschiedlich zum Metamodelle des Ursprungsmodells. Eine *query* ist eine eingeschränkte Art einer Sicht. Sichten werden aus Transformationen generiert.

Views

Als Drittes müssen Transformation von MOF-2.0-Modellen unterstützt werden, es wird ein Quellmodell in ein Zielmodell überführt. Hierbei können Ziel- und Quellmodell identisch sein.

Transformation

Der zentrale Oberbegriff von QVT ist das oben erläuterte Konzept der Transformation. Sie klassifiziert mögliche Arten der Modellüberführung. Im Kontext von QVT ist eine Transformation über einer Menge von Domänen definiert. Eine Domäne ist eine Menge von Modellelementen eines spezifischen Modells, die mit weiteren Bedingungsformulierungen eingeschränkt werden kann. Sie enthält somit die für die Transformation relevanten Konzepte, abgegrenzt gegen andere Domänen. Die Transformation zerfällt in die beiden Teilmengen Relation und Abbildung. Die Relation beschreibt und definiert Beziehungen zwischen Modellen und deren Elementen, bzw. Beziehungen zwischen Domänen, welche überprüft werden können. Die Abbildung beschreibt die Modellüberführung von einer Menge von Quellmodellen in ein Zielmodell. Sie ist eine operationale Überführung von Konzepten innerhalb der ihnen zugeordneten Domänen. Die von QVT definierte Basis zur Sprachtransformation und somit auch der Relation und der Abbildung ist die *Core*-Sprache. Sie definiert Konzepte zur Domänenspezifizierung und operationalen wie auch relationalen Transformation. Die Abbildung 1 zeigt graphisch das Verhältnis zwischen den Begriffen Transformation, welche aus ein oder mehreren Domänen bestehen, den beiden Unterbegriffen Abbildung und Relation, die wiederum beide auf der *Core*-Sprache beruhen, bzw. bei der Beschreibung zum Einsatz kommen.

Konzeptbeziehung

Mit Hilfe dieser Konzepte ist es möglich, in QVT Abbildungen und Transformationen

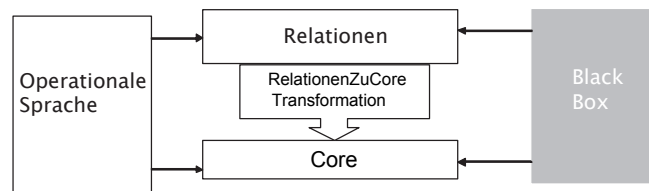


Abbildung 2: QVT Sprachbeziehungen

zu definieren. Ausgangspunkt einer jeden Transformation sind die jeweiligen Metamodelle der beteiligten Sprachen. Hierzu definiert man mit QVT Muster von Konzepten und deren Beziehungen aus den Metamodellen und setzt diese in Beziehung, um so die Transformation zu beschreiben. Weiterhin benötigt man die Anfragen (Queries), um die Menge der an den Mustern beteiligten Elemente zu beschreiben. Nun kann man jede Instanz der Metamodelle ineinander überführen oder überprüfend transformieren.

2.2 Architektur

QVT ist eine hybride Transformationssprache, bestehend aus einem deklarativen und einem imperativen Anteil. Das bedeutet, zum einen beschreibt man eine Menge von Beziehungen zwischen Mustern (deklarativ) und zum anderen operational eine Menge von sequentiellen Anweisungen zur Transformation (imperativ). Die Definition von Beziehungen zwischen Modellen erfolgt musterbasiert, entsprechend der zur Transformation verwendeten Metamodelle.

QVT selbst wird über Metamodelle definiert. Da es in drei Sprachbestandteile zerfällt, wird für jeden ein Metamodell angegeben. Jedes von ihnen basiert auf OCL 2.0 [OCL, 2003], der Anfragesprache von UML, und erweitert es um die jeweils benötigten Konzepte. Die drei Bestandteile sind die relationale, die operationale und die Core-Sprache.

Abbildung 2 zeigt den relationalen und Core-Bestandteil in der Mitte, sie entsprechen dem deklarativen Anteil von QVT. Jeweils daneben sind die zwei Möglichkeiten, den imperativen Aspekt einfließen zu lassen. Die operationale Sprache bietet auf der Basis von OCL 2.0 eine standardisierte Möglichkeit die Implementation der Transformation zu beschreiben, während die »Black Box« die größtmögliche Flexibilität erbringt, indem es dem Nutzer ermöglicht wird, beliebige Formen der Spezifikation anzuwenden – sofern eine vordefinierte Signatur eingehalten wird. Im Folgenden werde ich kurz auf die Sprachbestandteile eingehen und sie zusammenfassend erläutern. Im Anschluss wird jeder von ihnen in einem eigenen Abschnitt detailliert beschrieben. Auch weise ich auf fehlende Informationen oder Widersprüche im Standard hin.

Die relationale Sprache von QVT ist das Grundgerüst zur Definition von Transformationen. Basierend auf dem OCL-2.0-Metamodell wurden Erweiterungen vorgenommen, um die für Transformationen benötigten Konzepte einzuführen. So wurde z. B. das Konzept des Musters über Modellelemente eingeführt, wie auch das Konzept der Operati-

Sprachbestandteile von QVT

Die relationale Sprache

on verfeinert. Die relationale Sprache ermöglicht es, multidirektionale Transformationen zwischen Modellen (Metamodellinstanzen) durchzuführen. Hierzu spezifiziert man Muster und definiert eine Beziehung zwischen ihnen. Ein Muster (Domäne) ist eine Menge von Modellkonzepten und deren Assoziationen untereinander sowieso ihren Attributen. Diese werden entsprechend des jeweiligen Metamodells angegeben und definiert. Setzt man nun die Muster in Beziehung zueinander, ist es möglich, zwischen den Metamodellen bzw. Modellen zu transformieren und ihre Gültigkeit oder Überführbarkeit zu überprüfen.

Implizit wird während des Vorgangs der Transformation für jede Relation eine Instanzmenge von *Trace* Objekten angelegt, welche Referenzen auf die Quell- und Zielmodellelemente enthalten. Mit Hilfe dieser Objekte wird eine Historie und Protokollierung der Transformation ermöglicht, sowie die Möglichkeit für eine inkrementelle Transformation eröffnet.

Trace-Modell

Die *Core*-Sprache stellt den Kern von QVT dar. Sie ist im Vergleich zur relationalen kleiner und kompakter – mit dem Nachteil, dass sie weniger Konstrukte zur Verfügung stellt und so in komplizierteren Ausdrücken arbeitet. Sie unterstützt ebenfalls die Definition von Mustern über einer Menge von Elementen sowie Bedingungen auf diesen Mustern. Die Ausdrucksmächtigkeit ist gleich der der relationalen Sprache, doch aufgrund des geringen Umfangs und der Überschaubarkeit der *Core*-Sprache ist es wesentlich einfacher ihre Semantik zu definieren, bedingt durch die minimale Menge an Konzepten, um die OCL erweitert wurde. Ebenfalls ist hier ein *Trace* Modell definiert, äquivalent zur relationalen Sprache. Die benötigten Objekte werden jedoch nicht implizit erzeugt, sondern müssen explizit definiert werden.

Core

Der imperative Sprachbestandteil von QVT enthält eine operationale Sprache, mit welcher es möglich ist, Implementation zu korrespondierenden Relationen zu definieren. Wie die vorhergehenden Sprachbestandteile basiert es ebenso auf OCL 2.0 und erweitert es um Konzepte, wodurch eine prozedurale Weise der Definition möglich ist. Diese ist stark an imperative Programmiersprachen angelehnt, um so den Einstieg wie auch die Verständlichkeit zu erleichtern.

Die operationale Sprache

Die allgemeine Vorgehensweise wird wie folgt beschrieben: Zuerst definiert man mit Hilfe der relationalen Sprache Muster und deren Beziehungen (bezogen auf das jeweilige Metamodell), dann ist es optional möglich, die Implementierung bzw. Ausführung der Regeln mit Hilfe der operationalen Sprache genauer zu definieren. Dadurch wird dem Entwickler die Möglichkeit gegeben, in den verschiedenen Phasen der Softwareentwicklung QVT einzusetzen. Identifiziert man während der Entwurfsphase Beziehungen zwischen den verschiedenen Repräsentationen, so kann man sie mit Hilfe der relationalen Sprache festhalten, um dann in der Implementierungsphase die operationalen Aspekte zu definieren.

2.3 Relationale Transformation

Die relationale Sprache ist der Ausgangspunkt für jede Transformationsdefinition. QVT stellt mit ihr eine Sprache zur Verfügung, die es ermöglicht, Beziehungen (Relationen) zwischen Modellinstanzen auf Basis ihrer Metamodelle zu spezifizieren und zu

überprüfen. Voraussetzung ist, dass die Metamodelle MOF-Instanzen sind. Durch Angabe von Mustern auf Basis dieser Metamodelle werden Domänen definiert, die auf einer beliebigen Menge von MOF-konformen Metamodellen basieren.

Es wird zwischen zwei Arten der Notation der Relationen unterschieden: zum einen die textuelle, zum anderen die graphische. Letztere ist allerdings eher als Möglichkeit zur Präsentation als denn als äquivalente Sprache zu sehen. In der vorliegenden Version von QVT [QVT, 2005] wird behauptet, dass eine Äquivalenz gilt, doch wird keine Abbildung zwischen den beiden Formen beschrieben. Auch fehlen verschiedene Konzepte der textuellen Variante in der graphischen (z. B. Importieren und Angeben von Quellmetamodellen, Definition einer Transformation sowie der Transformationsrichtung). Dennoch erleichtert die graphische Notation in Form von Hybrid-Klassen-Objekt-Diagrammen das Verständnis der Transformationsregeln, sie kann aber nicht als vollständiger Ersatz betrachtet werden.

2.3.1 Relationale textuelle Transformation

Die textuelle Variante zur Spezifikation von Transformationsrelationen ist durch ein Metamodell und eine abstrakte Grammatik definiert. Sie dient zur Spezifizierung von Relationen zwischen Modellen. Basis einer jeden Regel sind die sogenannten Domänen, abgegrenzte einem Metamodell zugeordnete Gültigkeitsbereiche mit einer Menge von Variablen und Mustern. Ein Muster sind Objektschablonen, welche Modellelemente mit ihren Attributen und Assoziationen beschreiben. Zwischen den Domänen werden implizite, multidirektionale Beziehungen vereinbart. Zusätzlich kann jede Domäne mit einer Vor- und Nachbedingung weiter eingeschränkt werden. Das gilt ebenfalls für die gesamte Relation. Die Definition einer Transformation setzt sich aus einer Menge von Relationen zusammen, welche sich untereinander benutzen können, sowie einer Transformationsrelation, die der Ausgangspunkt der Transformation ist. Die Transformation gilt als erfolgreich, wenn jede Relation erfüllt wurde, dabei kann eine Transformation überprüfend oder auch manipulierend vorgenommen werden.

Transformation

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS)
```

Beispiel 1: Transformationsregel in QVT

Im Beispiel 1 ist die Definition einer Transformation zwischen UML und einem relationalen Datenbankschema gegeben. Mit dem Schlüsselwort `transformation` wird diese Relation als Transformation definiert. Der Name ist `umlRdbms`, die zwei Parameter entsprechen den Instanzen der benannten Metamodelle `SimpleUML` und `SimpleRDBMS`.

Um eine Transformation zu definieren, gibt man eine Menge von Relationen an, aus denen sich die Transformation zusammensetzt. Eine Relation besteht aus zwei oder mehr Domänen über den beteiligten Metamodellen. Diese bestehen aus einem Domänenmuster, welches eine Menge von Objektschablonen und Bedingungen darstellt. Modellelemente, welche an diese Schablonen gebunden werden, müssen die gegebenen

Relation

```

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS)
relation PackageToSchema
{
  domain uml p:Package {name=packageName}
  domain rdbms s:Schema {name=packageName}
}

```

Beispiel 2: Einfache Relation in QVT

Bedingungen erfüllen, so dass das Muster erfüllt ist.

Das Beispiel 2 zeigt eine einfache Relation zwischen zwei Domänen, die erste (uml) beinhaltet ein Muster, welches die Variable `p` und `packageName` definiert, die zweite (xml) definiert die Variable `s` und wiederverwendet `packageName`. Zur Transformation werden nun die Variablen an entsprechende Elementinstanzen gebunden, `p` an eine Instanz des Metamodellelements `Package` und `s` an eine Instanz von `Schema`. Zusätzlich muss `p` ein Attribut `name` enthalten welches an `packageName` gebunden wird. Weiterhin muss zwischen den Domänen gelten, dass die Schemainstanz ebenfalls ein Attribut `name` enthält, welches den gleichen Wert aufweisen muss wie der Paketname.

Es gibt die Möglichkeit, jede Relation mit dem Schlüsselwort `top` zu qualifizieren, da unter Umständen der Benutzer die Gültigkeit von Relationen nur dann gesichert wissen will, wenn eine andere erfüllt ist bzw. eine Hierarchie von Relationen spezifizieren möchte. Stellt man sich die Menge der Relationen als Baumstruktur vor, so ist jede `top`-Relation eine Wurzel und jede nicht mit `top` versehene Relation ein Kind der jeweiligen Relation, in deren `where`-Klausel ihre Gültigkeit gefordert wird. Jede nicht `top`-Relation muss also nur implizit gelten, wenn sie direkt oder transitiv in der `where`-Klausel einer Relation gelten muss.

*top-
Relationen*

```

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
  top relation PackageToSchema {...}
  top relation ClassToTable {...}
  relation AttributeToColumn {...}
}

```

Beispiel 3: Top-Relation in QVT

Das Beispiel 3 zeigt drei Relationen, während `PackageToSchema` und `ClassToTable` explizit gelten müssen, muss `AttributeToColumn` nur dann gelten, wenn es von einer der beiden vorhergehenden Relationen im `where`-Teil gefordert wird.

Neben der Angabe von Mustern in jeder Domäne bietet die relationale Sprache von QVT die Möglichkeit zur Angabe von zwei Prädikaten, die `where`-Klausel und die `when`-Klausel. Die `when`-Klausel spezifiziert die Bedingung unter der die Relation erfüllt ist, während die `where`-Klausel Bedingungen über alle an der Relation beteiligten Modell-

*when- und
where-Klausel*

```

relation ClassToTable
{
  domain uml class:Class {
    namespace = package:Package {},
    kind='Persistent',
    name=cn
  }
  domain rdbms table:Table {
    schema = schema:Schema {},
    name=cn,
    column = cl:Column {
      name=cn+'_tid',
      type='NUMBER' },
  }
  when {
    PackageToSchema(package, schema);
  }
  where {
    AttributeToColumn(class, table);
  }
}

```

Beispiel 4: When und Where Klauseln in QVT

elemente definiert, welche von ihnen erfüllt sein muss. Beide Klauseln können OCL-Ausdrücke enthalten und somit auch weitere Einschränkungen, QVT definiert die Semantik der Klauseln wie folgt:

Für jede gültige Belegung der Variablen in der when-Klausel und den Variablen aller von der Zieldomäne k verschiedenen Domänen, welche die when-Bedingung und Quelldomänenmuster sowie -bedingungen erfüllt, muss eine gültige Belegung der restlichen Variablen existieren, so dass die Muster und where-Bedingung der Domäne k erfüllt sind.

In Beispiel 4 sind eine where-Klausel und eine when-Klausel über die gesamte Relation angegeben. Nimmt man nun an, dass eine Transformation von *UML* nach *RDBMS* erfolgt, so müsste bei einer Erfüllung des UML-Musters gleichzeitig auch die when-Klausel erfüllt sein, also die Relation *PackageToSchema*. Das führt zu dem Seiteneffekt, dass die freien Variablen bei der Überprüfung an Instanzen der Modellelemente gemäß der Relation *PackageToSchema* gebunden werden und nur diese bei der Musterüberprüfung der Domäne *uml* herangezogen werden. Sind diese beiden Bedingungen erfüllt, muss nun die where-Klausel gelten und die Domäne *rdbms* überprüft bzw. manipuliert werden – entsprechend der dort angegebenen Muster.

Die tatsächliche Transformation in QVT erfolgt aus einer Menge von Quellmodellen in ein Zielmodell. Bei der Angabe der Relationen und der entsprechenden Domänen ist es möglich, für die Ausführung der Transformation die Art für das Zielmodell anzugeben. QVT unterscheidet hier zwischen *enforce* und *checkonly*. Bei der Angabe

checkonly
und *enforce*
Domänen

des Ersten wird bei nicht erfülltem Muster im Zielmodell eine Manipulation vorgenommen, das heißt, es werden Modellelemente gelöscht oder bei Bedarf erzeugt. Ist dagegen `checkonly` angegeben, so wird nur eine Überprüfung vorgenommen und bei einem Fehlschlag die Relation als nicht erfüllt betrachtet.

```

relation PackageToSchema
{
  checkonly domain uml p:Package {name=pn}
  enforce domain rdbms s:Schema {name=pn}
}

```

Beispiel 5: Enforce und checkonly in QVT

Das Beispiel 5 zeigt wiederum die zwei Domänen *uml* und *rdbms*. Führt man nun eine Transformation von einer gegebenen UML-Instanz nach einer RDBMS-Instanz aus, wird zuerst die Variable `p` an eine Instanz des Modellelements `Package` gebunden und die Variable `pn` mit dem Wert des Attributs `name` von `p` belegt. Im Zielmodell *rdbms* wird nun überprüft, ob es eine Instanz `s` von `Schema` gibt, deren Attribut `name` mit dem Wert von `pn` belegt ist. Sollte dies nicht der Fall sein, wird eine neue Instanz von `Schema` erzeugt und der Name entsprechend gesetzt, da *rdbms* mit `enforce` qualifiziert ist. Würde die Transformation in die andere Richtung ausgeführt werden, würde nur eine Überprüfung stattfinden, da hier `checkonly` gesetzt ist.

Die relationale Sprache ermöglicht also eine musterbasierte Angabe von Relationen zwischen definierten Domänen. Mit Hilfe der `when`- und `where`-Klauseln, welche OCL-Ausdrücke enthalten können, ist es möglich, Vor- und Nachbedingungen anzugeben. Hinsichtlich der Ausführung der Transformation ist es möglich Empfehlungen der Behandlung des Zielmodells anzugeben, so dass eine Überprüfung oder Manipulation vorgenommen wird. Die gesamte Transformation setzt sich aus einer Menge von Relationen zusammen und kann als ein gesamter logischer Ausdruck interpretiert werden, der seine Wertbelegungen aus den entsprechenden Modellinstanzen gewinnt.

2.3.2 Relationale graphische Transformation

QVT definiert eine weitere Repräsentationsmöglichkeit zur Spezifizierung der Relationen, die graphische Form. Entgegen der textuellen bietet sie eine anschauliche und intuitive Möglichkeit Relationen zu definieren. Eine Relation setzt zwei oder mehrere Muster in Beziehung, wobei ein Muster aus Objekten, Links und Werten besteht. Daher bieten sich UML-Objektdiagramme als Basis für die graphische Notation an, wobei ein neues Symbol eingeführt wird, um die Relation selbst darzustellen. QVT definiert die Darstellung der Domänen allein durch die Anordnung zum Transformationssymbol, so dass eine Anordnung zu treffen ist, die hinreichend eindeutig ist. Die einzelnen Domänen werden durch eine gerichtete Linie verbunden, an derer man den Buchstaben C oder E angibt, um `checkonly` und `enforce` zu definieren. Des Weiteren gibt man am Symbol relativ zu den Verbindungslinien die entsprechenden Modellinstanzen bzw. deren Metamodelle

*Graphische
Konventionen*

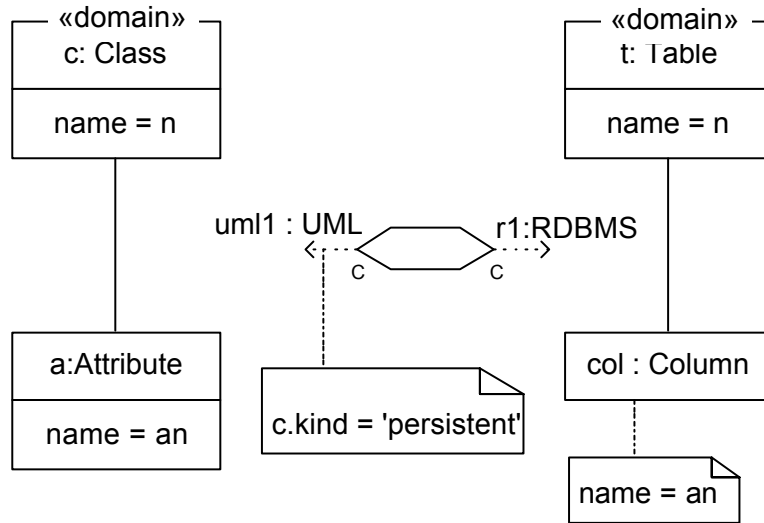


Abbildung 3: Einfache graphische Transformation

an. Abbildung 3 zeigt die graphische Notation der Relation aus Beispiel 4. Jedes Objekt stellt eine Instanz des entsprechenden Metamodellelements dar, mit den jeweiligen Attributen und Assoziationen. Der Stereotype *domain* hebt nochmals die entsprechenden Domänen hervor, die beiden Bedingungssteile (*when*- und *where*-Klausel) werden separat angegeben – entweder als UML-Kommentar oder über querverlaufende Unterteilungen. Die Semantik ist entsprechend der textuellen Sprache definiert und basiert daher auf dem *Core*- Sprachbestandteil.

Zur Vereinfachung wird ein Mittel zur Angabe der Negation definiert, das Schlüsselwort *{not}*. Damit ist es zum Beispiel möglich zu definieren, dass ein Element keinen Assoziationspartner haben darf oder ein bestimmtes Attribut nicht belegt ist. Abbildung 4 zeigt ein Muster bestehend aus einer Klasse, die nicht über einen Link mit einer Attribut-Instanz verbunden sein darf, spezifiziert wird dies über *{not}*. Das Muster trifft dann und nur dann zu, wenn eine Klasseninstanz existiert, zu der es kein entsprechendes Attribut gibt.

Der von QVT vermittelte Eindruck, die graphische Notation stelle eine äquivalente Form zur Textuellen dar, stellt sich nicht ein. Das knappe Kapitel innerhalb von QVT widmet sich in aller Kürze offensichtlichen relevanten Aspekten der Definition von Relationen, doch werden einige wichtige Details außer Acht gelassen. So findet man nirgends eine Möglichkeit zur Definition der Transformation selbst. Auch bleibt völlig offen, wie eine Anordnung der Domänen auszusehen hat. Man sucht vergebens nach der Möglichkeit zur Angabe von Parametern für entsprechende Relationen oder dem Einbin-

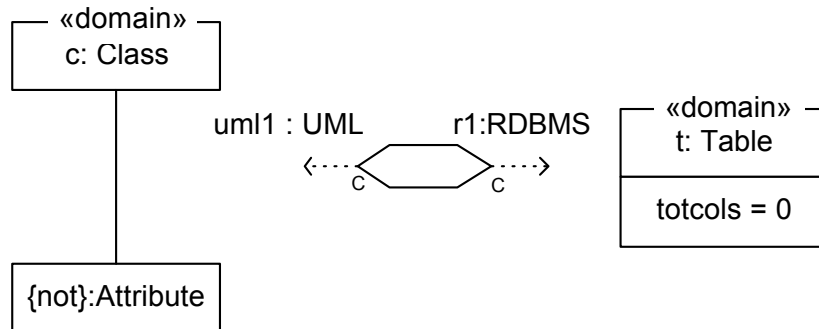


Abbildung 4: Relationale graphische Negation

den von Metamodellen. Es verdichtet sich der Eindruck, dass es sich bei der graphischen Notation nicht um eine äquivalente Sprache handelt, sondern eher um eine Ergänzung zur konkreten textuellen Syntax. Gedacht für einen intuitiven und einfachen Zugang und zur Präsentation entsprechender Relationen, nicht jedoch für eine fundierte Spezifikation.

2.4 Operationale Transformation

Die operationale Transformationssprache in QVT dient dazu die Ausführungssemantik einer Transformation zu definieren. Sie hat einen imperativen Charakter und basiert ebenso wie der relationale Sprachbestandteil auf OCL 2.0. Es ist mit ihr möglich, für jede Relation eine Regel anzugeben, die die enthaltene Transformation näher beschreibt. Im Gegensatz zur logikbasierten relationalen Sprache beschreibt man in der operationalen eine Folge von Anweisungen, die von einem Interpreter ausgeführt werden. Sie basiert auf einer abstrakten Syntax und einem Metamodell, des Weiteren bietet sie eine Standardbibliothek mit verschiedenen Funktionalitäten und Operationen zur Transformation an. Ihr Charakter ist nahe dem objektorientierter Programmiersprachen wie *Java* oder *C++*, so dass mit Kenntnis einer der Sprachen der Einstieg erleichtert wird. Die operationale Transformation benutzt das gleiche *Trace*-Modell wie die relationale Sprache.

Der Ausgangspunkt einer Transformation ist die sogenannte operationale Transformation, eine mit dem Schlüsselwort `transformation` versehene Einheit. Sie besteht – ähnlich einer Funktion aus Programmiersprachen – aus einem Namen, einer Signatur und einem Körper in geschweiften Klammern. Die operationale Transformation definiert des Weiteren eine Einstiegsoperation `main()`, von der aus die Transformation ausgeführt

Transformation

wird. Mit Hilfe der operationalen Transformation spezifiziert man eine unidirektionale Transformation von einer Menge von Quellmetamodellen nach einem Zielmetamodell.

```

transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
  main() {
    uml.objectsOfType(UML::Package)->map packageToSchema();
  }
  ...
}

```

Beispiel 6: Die operationale Transformation

Das Beispiel 6 zeigt eine einfache operationale Transformation. Sie transformiert von einer UML-Instanz in eine RDBMS-Instanz. Die Transformationsrichtung wird über die Schlüsselwörter `in` und `out` definiert. Die Metamodellinstanzen werden entsprechend benannt und in der `main()`-Funktion der Einstieg in die Transformation angegeben. Hierzu wird an der UML-Instanz, welche als Objekt aufzufassen ist, die OCL-Funktion `objectsOfType` gerufen, welche eine Menge von Instanzen des Metamodellkonzepts `Package` zurückgibt. QVT definiert den `»-><<`-Operator zur Handhabung von Objektinstanzmengen als abkürzende Schreibweise. An jedem dieser Objekte wird mittels `map` der Aufruf der Abbildungsoperation `packageToSchema()` angegeben. Eine Abbildungsoperation in QVT gehört zu einer entsprechenden Relation und definiert zu einer gegebenen Menge von Quellelementen die Erzeugung von Zielelementen. Sie besteht aus drei Teilen: Einer Vorbedingung (`when`-Klausel), einem Operationskörper und einer Nachbedingung (`where`-Klausel). Die beiden Bedingungen entstammen der korrespondierenden Relation, welche der Eigentümer der Klauseln ist. Sollte die Vorbedingung fehlschlagen, so wird der Operationskörper nicht ausgeführt und der Rückgabewert auf `null` gesetzt. Ansonsten wird der Körper ausgeführt und im Anschluss die Nachbedingung überprüft.

```

mapping UML::Package::packageToSchema() : result:UML::Schema
  when { self.name.startingWith() <> "-" }
  {
    population {
      object result:UML::Schema {
        result.name := self.name;
        table := self.ownedElement->map class2table();
      }
    }
  }
}

```

Beispiel 7: Operationale Abbildung von Paketen

In dem Beispiel 7 ist die Operation zum Abbilden von UML-Paketen auf RDBMS-Schema angegeben. Die Signatur der Operation setzt sich aus dem Schlüsselwort `mapping`, dem Namen der Operation `packageToSchema`, dem Kontextparameter vom Typ

Mapping

Package und dem Rückgabewert vom Typ *Schema* zusammen. Die Abbildungsoperation kann also an jeder Instanz eines Pakets aus UML gerufen werden. Dieses kann dann über die vordefinierte Variable *self* benutzt werden. In der Vorbedingung wird zuerst geprüft, ob der Name des entsprechenden Pakets nicht mit einem $\gg\ll$ beginnt. Sollte dies erfolgreich sein, so wird der Körper ausgeführt. Das bedeutet, dass eine Instanz eines Schemas angelegt und dessen Name auf den Paketnamen gesetzt wird. Für jede in dem Paket enthaltene Klasse wird nun die Abbildungsoperation `class2table()` gerufen und der Rückgabewert der Variablen `table` zugewiesen, welche dann wiederum weiter verwendet werden kann. Nach erfolgreicher Beendigung des Körpers wird die entsprechend erzeugte Schemainstanz zurückgegeben, ansonsten *null*.

Um denn häufigen Fall des Erzeugens eines Elements zu vereinfachen, gibt es in QVT eine entsprechend abgekürzte Schreibweise, welche implizit den Populationsabschnitt wie auch die Objekterzeugung beinhaltet.

```

mapping Package::packageToSchema() : Schema
  when { self.name.startingWith() <> "-"}
{
  name := self.name;
  table := self.ownedElement->map class2table();
}

```

Beispiel 8: Abgekürzte operationale Abbildung von Paketen

Beispiel 8 zeigt das gekürzte Beispiel 7. Es wurde auf die `result` Variable verzichtet, ebenso entfällt die Qualifizierung der Metamodellkonzepte, welche nur bei Namenskollisionen notwendig ist.

Neben den vorgestellten Konzepten bietet die operationale Sprache noch weitere Bestandteile zur Transformationsdefinition an. Es ist möglich, zusammengesetzte Transformationen zu definieren, Bibliotheken zu verwenden, Konstruktoren für oft benutzte Erzeugungen zu spezifizieren und Hilfsfunktionen zu definieren. Für Transformationen, welche in mehreren Schritten ausgeführt werden (Mehrphasstransformationen), bietet QVT Mittel zum Auflösen der Objektreferenzen auf Basis des *Trace*-Modells, so dass auf bereits erzeugte Modellelemente zugegriffen werden kann. Auch kann man Operationen disjunkt zu einer Menge anderer Operationen definieren, so dass auf Basis der Vorbedingung die Auswahl erfolgt.

*Weitere
Konzepte*

Die operationale Sprache bietet ein Verfahren zur Verfeinerung der Relationen, um so den eigentlichen Transformationsprozess näher zu beschreiben. Man definiert für jede Relation eine Abbildungsoperation, welche im Stil objektorientiert Programmiersprachen die Erzeugung der Elemente definiert. Die so definierte Transformation ist gerichtet und erfolgt von einer Menge von Quellmodellen nach einem Zielmodell.

2.5 Core

Die Basis der relationalen Sprache von QVT bildet die *Core*-Sprache. Sie folgt wie die *Core*

Core

relationale Sprache dem Prinzip der musterbasierten Regeldefinition. Ihre Ausdrucksmächtigkeit ist äquivalent zu der relationalen Sprache, doch ist sie an sich einfacher. Das bedeutet, sie besitzt weniger Konstrukte als die relationale Sprache. Das führt zu einer aufwändigeren Beschreibung der Beziehung zwischen den Modellelementen. Gleichzeitig ist die Definition der Semantik der *Core*-Sprache einfacher als die der relationalen – bedingt durch die reduzierte Anzahl von Konzepten. QVT definiert eine Abbildung der relationalen Sprache auf die *Core*-Sprache mit Hilfe der relationalen, so dass ihre Semantik über *Core* definiert werden kann. Gleichzeitig kann die *Core*-Sprache als Basis für Implementierungen genommen werden, es muss nur jede relationale Regel abgebildet werden.

In der relationalen Sprache werden für jeden Transformationsvorgang die sogenannten *Trace*-Objekte implizit erzeugt. Ihre Klassenstruktur ist ebenso vorgegeben. In der *Core*-Sprache dagegen muss jedes *Trace*-Objekt explizit erzeugt werden, ebenso seine Verbindungen zwischen den Quellobjekten und dem Zielobjekt. Das führt zwar zu einem Mehraufwand bei der Definition der Regeln, erlaubt aber auch eine flexible Spezifizierung der Struktur der *Trace*-Objekte. Es muss jedes *Trace*-Objekt explizit als Muster definiert werden.

Trace-Modell

Gleich der relationalen Sprache besteht eine Transformation aus einer Menge von Regeln, Abbildungen genannt, welche musterbasiert Beziehungen zwischen Modellelementen definieren. Ergänzend muss für jede Abbildung nicht nur gelten, dass sie die jeweiligen Modelle erfüllt, sondern auch dem angegebenen *Trace*-Modell genügt. Analog zur relationalen Sprache kann eine Transformation auf zwei Arten durchgeführt werden, überprüfend oder erzwingend. Jede Abbildung besteht wie eine Relation aus einer Menge von Domänen, welche eine Menge von Mustern beinhalten. Zusätzlich muss in der *Core*-Sprache für jedes Domänenpaar ein Muster zur Transition der Domänen wie auch der Erzeugung der *Trace*-Objekte angegeben werden.

2.6 BlackBox

Die *BlackBox* ist neben der operationalen Sprache eine weitere Möglichkeit die Ausführung von Transformationen näher zu beschreiben. Sie bietet die größten Freiheiten für den Entwickler und ermöglicht die Integration bestehender Transformationen, Implementierungen und Bibliotheken.

Man definiert in der relationalen Sprache entsprechende Transformationsregeln und verweist über das Schlüsselwort `implemented by` auf die *BlackBox* Implementierung der jeweiligen Regel. Die einzige Restriktion ist durch die Signatur der Implementierungsregel gegeben, welche wiederum durch die relationale Regel definiert ist. So ist es möglich jeden Ansatz bzw. jede Implementierungssprache in QVT einzubinden, sofern die spezifizierte Signatur eingehalten wird.

individuelle Implementierungen

2.7 Verfügbare Werkzeuge

Die einzige verfügbare QVT-Implementierung stammt aus dem Jahr 2003 und wurde vom Lip6 [MODFact, 2003] bereit gestellt. Sie stellt ein *proof-of-concept* des damaligen

ModFact

QVT-Stands dar. Selbst zu diesem Zeitpunkt war sie an kleinen Beispielen einsetzbar und bedingt lauffähig. Zum jetzigen Zeitpunkt entspricht sie kaum noch der Definition von QVT und ist nicht für den Produktiveinsatz geeignet. Leider ließen sich keine anderen Implementierungen von QVT auffinden, so dass aktuell davon ausgegangen werden muss, dass kein Werkzeug mit dedizierter QVT Unterstützung existiert. Das Kriterium einer Werkzeugunterstützung ist somit leider nicht erfüllt und verwehrt damit einen tatsächlichen praktischen Einsatz von QVT.

Ein weiterer Ansatz ist das von unter IBM entwickelte *Model Transformation Framework* (MTF) [Griffin, 2004] ein praktischer Java-basierter Ansatz zur Transformation zwischen EMF-Modellen [EMF, 2005]. Es besitzt eine textuelle Syntax zur Beschreibung von bidirektionalen Transformationen, ebenso gibt es eine persistente Aufzeichnung des Transformationsprozesses. Dieser Ansatz orientiert sich an der ersten Version der vorgeschlagenen Sprache von QVT. Er basiert auf der Angabe von Relationen zwischen zwei Modellelementen, über welche eine Bedingung in Java formuliert werden kann. MTF bietet keine OCL-Unterstützung. Es gibt eine gepflegte Implementierung von MTF, welche auf einer Eclipse-Integration [Eclipse, 2005] beruht.

Model Transformation Framework

An der Technischen Universität Berlin wurde M2T, ein graphbasierter Ansatz, entwickelt. M2T basiert auf OCL 2.0 und erlaubt unidirektionale musterbasierte Transformationen, indem man eine linke Seite zu einer rechten in Beziehung setzt. Zur Verfolgung der Transformation und der Möglichkeit der inkrementellen Transformation können in M2T *Traces* explizit definiert werden.

M2T

2.8 Zusammenfassung

Zusammenfassend bietet QVT durch die relationale und die operationale Sprache die Möglichkeit zum Definieren einer Transformation wie auch einer Abbildung. Die relationale Sprache bietet ein verständliches Konzept graphisch wie auch textuell zur Musterdefinition und dem Assoziieren von Modellelementen. Auch wird ein Trace-Modell definiert und unterstützt, sowie die Abbildung und auch Transformation von einer Menge von Modellen in eine Menge von Modellen. Durch die Basis OCL ist auch QVT eine formal basierte Sprache. Auf dieser Basis scheint QVT für den Einsatz in ULF zur Transformationsspezifikation geeignet. Im folgenden Abschnitt wird seine praktische Relevanz untersucht und an gebräuchlichen Transformationsmustern aufgezeigt.

3 Transformation von UML nach XML mit QVT

3.1 Einführung

Dieses Kapitel beschreibt den praktischen Einsatz von QVT am Beispiel der Transformation von einer Teilmenge von UML 1.5 (Klassendiagramme) nach XML. Hierzu beschreibe ich an Hand von gebräuchlichen Transformationsmustern den Einsatz von QVT und die Umsetzung der Transformationen sowie eventuell auftretende Schwierigkeiten.

Im Laufe der Modellierung und des Entwurfs kommen verschiedene Techniken und

UML

Technologien zum Einsatz, unter anderem auch UML und XML. UML ist eine domänenunabhängige Sprache zur Modellierung der verschiedenen Aspekte eines Systems mit Hilfe einer graphischen Notation, deren Ziel Verständlichkeit und Lesbarkeit ist. UML stellt hier neun Diagrammartentypen zur Verfügung, die in den verschiedenen Phasen des Softwareentwurfs zum Einsatz kommen. In dieser Arbeit werden nur die Klassendiagramme betrachtet, welche die statische Struktur eines Systems mit Hilfe von Objektklassen und deren Beziehungen beschreiben.

XML ist eine reine strukturbeschreibende Sprache, welche eine Dokumentstruktur definiert. XML-Schema als eine Instanz von XML, definiert die Syntax und Semantik zur Beschreibung von XML-Dokumenten. Im Gegensatz zu DTDs bietet sie jedoch mehr Mittel zur Differenzierung und ist selbst eine XML-Dokumentinstanz. XML-Schema stellt ein Schlüssel-Verweis-Konzept zur Verfügung, besitzt mehr vordefinierte Datentypen als DTDs, sowie eine Datentyphierarchie.

XML

Modelliert man nun sein Softwaresystem in UML oder liegen bestehende Diagramme vor, kann man durch die Abbildung nach XML-Schema diese nun automatisch transformieren⁵, als XML-Dokumente speichern und weiter benutzen. So verlangen viele Applikationen XML-Dokumente als Eingabeformat, welche nunmehr durch die Transformation generiert werden können. Auch ist ein Datenaustausch auf XML Basis möglich und ebenfalls eine domänenunabhängige Modellierung mit UML, welche dann weiterführend als XML-Dokument benutzt werden kann.

Die Abbildung beruht in großen Teilen auf dem Vorgehen von Dr. Eckstein & Dr. Eckstein. Sie beschreiben ihren Ansatz ausführlich in dem Buch „XML und Datenmodellierung“.

Die Form der Abbildungsregeln für verbreitete Transformationsmuster stellt sich wie folgt dar. Zuerst wird die Regel verbal beschrieben und folgend ihre Umsetzung in der jeweiligen Sprache (relational graphisch, relational textuell und operational) definiert. Dabei wird zuerst die relationale graphische Variante erklärt, dann der textuellen gegenübergestellt und an relevanten Stellen ebenso die operationale Variante mit einbezogen.

Konventionen

3.2 Transformationsregeln

In diesem Abschnitt beschäftige ich mich mit der graphischen Repräsentation der relationalen Transformation. Hierzu definiere ich die Abbildungsregeln von UML nach XML-Schema in entsprechenden Transformationsdiagrammen und gebe eine Erläuterung dazu an. Es wird nicht jede Abbildungsregel ausführlich erklärt oder genannt: Ziel ist es, den Einsatz von QVT an relevanten Punkten bzw. gebräuchlichen Transformationsmuster zu demonstrieren und Schwierigkeiten an den entsprechenden Punkten aufzuzeigen.

Anmerkung: Aufgrund fehlender Werkzeugunterstützung habe ich zur Spezifizierung ein herkömmliches UML Werkzeug (*Visual Paradigm* [Paradigm, 2005]) herangezogen. Dabei sind natürlich nicht alle von QVT definierten Symbole enthalten, so dass ich mit Ersatzkonstruktionen arbeiten musste. Bedingte Anweisungen, also das where- und

⁵Voraussetzung ist natürlich eine QVT Implementierung.

```

transformation uml2xml (uml1:UML, xml1:XML) {
    uml2xmlPackage;
    uml2xmlNoPackage;
}

```

Beispiel 9: Die Transformationsregel

```

transformation uml2xml(in uml:UML, out xml:XML) {
    main() {
        uml.objectsOfType(Package)->map uml2xmlPackage();
    }
}

```

Beispiel 10: Die operationale Transformationsregel

when-Konstrukte werden von mir als Kommentarfeld dargestellt. Zur Übersichtlichkeit sind der linke und rechte Teil einer jeden Regel in ein Paketsymbol eingebettet. Um eine Relation als Einstiegs- und rufbare Regel zu markieren, habe ich als *stereotype*⁶ für diese Regel <<top>> verwendet.

Ausgangspunkt einer jeden Transformation ist die Transformationsregel selbst, da es keine entsprechende Vereinbarung in der graphischen Notation gibt, gehe ich an dieser Stelle nur auf die textuelle Variante ein.

Im Beispiel 9 wird die Transformation `uml2xml` definiert. Als Parameter werden ihr eine UML-Instanz (`uml1`) und eine XML-Instanz (`xml1`) übergeben. Im Körper der Transformation wird vereinbart, dass die zwei Relationen gelten müssen. Die beiden Relationen müssen erfüllt sein, damit die Transformation erfolgreich ist. Diese Regeln sind als folgende Verzweigungen der Transformation zu verstehen und bilden daher die Wurzeln separater Transformationsbäume, wenn man den Pfad der Transformation als Baumstruktur auffasst. Beispiel 10 zeigt die Definition in operationaler Sprache, es werden alle Pakete des Modells erfasst und die Abbildungsregel angewendet.

Ein weit verbreitetes Konzept ist das *container-content*-Muster. Es setzt ein Modellelement (Behälter) mit einer Menge von Modellelementen (Inhalt) in Beziehung. Die Menge ist in ihm enthalten. UML enthält zur Strukturierung und der Wiederverwendbarkeit das Konzept des Pakets. Da diese einen eigenen Namensraum eröffnen liegt es nahe, diese ebenfalls auf das Namensraumkonzept im XML-Schema abzubilden. Das bedeutet, für jedes Paket in UML wird ein neuer Namensraum in XML erzeugt. Der Inhalt des Pakets wird entsprechend der nachfolgenden Regeln behandelt. Jedes im Paket enthaltene Modellelement wird diesem zugewiesen. Die Anfänge bzw. Einstiegspunkte stellen die

container-
content
Pakete

⁶Ein *stereotype* erweitert ein bestehendes UML-Element und seine Semantik.

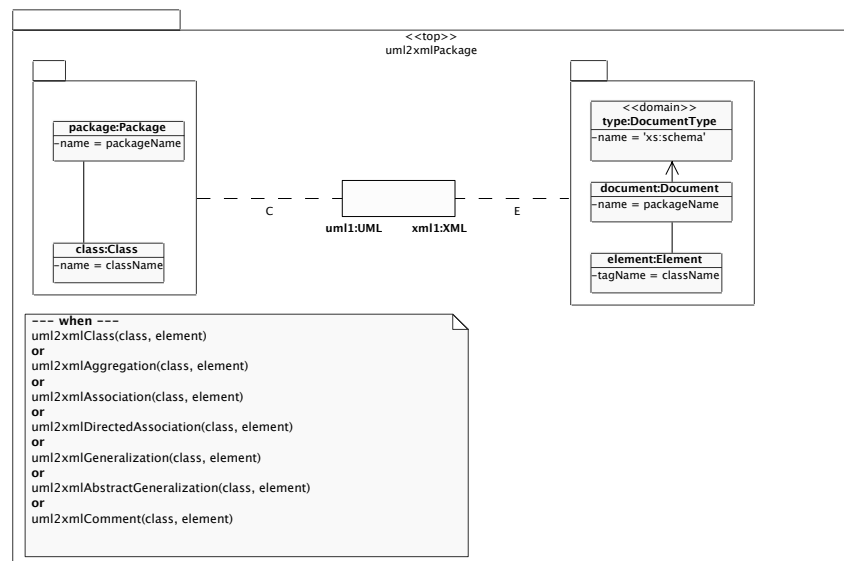


Abbildung 5: Transformation von UML Paketen

Paketstruktur von UML dar. Jedes Modellelement ist entweder in einem expliziten Paket enthalten oder im globalen impliziten Paket. Daher sind sie die mit *top* qualifiziert und einzigen explizit rufbaren Relationen. Alle anderen Relationen werden von der Paket Regel gerufen, welche damit alle Modellelemente erfasst und abbildet. Das bedeutet, der Behälter kann über die Benutzung der Konzepte *top* und einer Einschränkung über die *when*-Klausel erfasst und der Inhalt davon ausgehend abgebildet werden.

Abbildung 5 zeigt die Regel in graphischer Repräsentation. Sie trägt den Namen `uml2xmlPackage` und ist als Einstiegsrelation gekennzeichnet. Auf der linken Seite ist die betreffende Domäne UML, und alle Konzepte der linken Seite entspringen ihr. Das Muster spezifiziert, dass eine Instanz des Modellelements *Package* existiert, deren Attribut *name* an die Variable *packageName* gebunden wird. Weiterhin muss eine Instanz des Modellelements *Class* existieren, die dem Paket zugeordnet ist und deren Name ebenso an eine Variable gebunden wird. Da das Transformationssymbol auf seiner linken Seite mit einem *C* gekennzeichnet ist, würde eine Transformation von rechts nach links nur überprüfend vorgenommen werden. Dagegen kennzeichnet das *E* auf der rechten Seite, dass es sich um eine Abbildung von UML nach XML handelt. Damit nicht für jeden Regelaufruf neue Elemente erzeugt werden, übergebe ich nun die Referenzen auf die erzeugten Elemente an die nachfolgenden Regeln weiter. Im bedingten Abschnitt werden nunmehr alle Regeln aufgerufen, um mögliche Konstellationen abzubilden.

Da üblicherweise nicht jedes Element in einem Container enthalten ist, definiert man einen impliziten universellen Behälter. Dieser enthält alle behälterlosen Modellelemente sowie alle expliziten auf höchster Ebene stehenden Behälter. Am Beispiel von UML ist dies das implizite globale Paket.

*globaler
Container*

Die Regel für das implizite globale Paket ist analog zur Regel für das explizite Paket – mit dem Unterschied, dass der Name des erzeugten Dokuments auf *global_concepts* festgelegt wird. Sie ist identisch mit Abbildung 5, abgesehen von der Zuweisung des Dokumentnamens an eine statische Zeichenkette.

*implizites
globales Paket*

```

top relation uml2xmlPackage
{
  packageName, className : String;
  checkonly domain uml1 class:Class { name = className, namespace
    = package:Package
    { name = packageName}
  };
  enforce domain xml1 ns:NameSpace { name = packageName
    { child = element:Element
    { tagName = className}
  };
  where {
    uml2xmlClass(class, element);
    uml2xmlAggregation(class, element);
    uml2xmlAssociation(class, element);
    uml2xmlDirectedAssociation(class, element);
    uml2xmlTertiearAss(class, element);
    uml2xmlGeneralization(class, element);
    // und weitere Regeln, die aus Gruenden der
    // Uebersichtlichkeit ausgelassen wurden
  }
}

```

Beispiel 11: Abbildung von Paketen

Die textuelle Darstellung dieser Relation ist in Beispiel 11 zu sehen. Die beiden Domänen sind UML und XML. Sie wurden in der Transformationsregel 9 vereinbart und definiert. Die beiden lokalen Variablen `packageName` und `className` dienen zur Überführung der entsprechenden Namen in die jeweiligen Modellelemente. Als Parameter werden für jede Regel die Klassen- und Elementinstanzen weitergereicht, so dass sichergestellt ist, dass es sich tatsächlich um das gleiche Element und die dazugehörige Klasse handelt, die transformiert werden. Die Verknüpfung von Modellelementen erfolgt über ihre Assoziationen zueinander, genauer über die jeweiligen Rollennamen.

Ein weiteres Kernkonzept der metamodel basierten Transformation sind Instanzen von MOF-Klassen, welche Operationen wie auch Attribute enthalten können. Im Folgenden wird die Abbildung von Attributen auf entsprechende Attribute des Zielmodellelementes beschrieben. Das Konzept der Klasse in UML entspricht einer semantischen Einheit, die als Container und Namensraum fungiert. Klassen können Attribute wie auch Operationen enthalten, ebenso können sie an Assoziationen teilnehmen. Sie werden auf Elemente in XML abgebildet, denn Elemente weisen ein ähnliches Konzept auf. Sie sind ein ebenso strukturiertes Konzept, so dass für jede konkrete Klasse in UML ein Element angelegt wird. Die Attribute der Klasse werden in einer separaten Regel behandelt. Operationen werden nicht abgebildet, da erstens kein äquivalentes Konzept in UML existiert und sie zweitens lediglich Strukturinformationen darstellen.

Attribute

Nachfolgend ein Beispiel der textuellen Repräsentation der Definition von Abbildungsregeln in QVT. Es ist eine Abbildungsregel und damit Transformation, da für mindestens eine der beteiligten Domänen `enforce` gesetzt wurde (für XML-Schema).

```

relation uml2xmlClass(class, element) {
  checkonly domain uml1 class:Class {name = className, set {
    attribute = aUML:Attribute
    {name = attributeName, multiplicity = n..m} }
  };
  enforce domain xml1 element:Element { tagName = className,
    child = complT:Element
    { tagName = 'xs:complexType', child = seq:Element
      { tagName = 'xs:sequence', set {child = aXML:Element
        { tagName = attributeName, attribute = maxOccurs:Attr{
          name='maxOccurs', value = m}
          attribute = minOccurs:Attr {name='minOccurs', value =
            n }}}
      };
    when {
      uml2xmlAttribute(attribute);
    }
  }
}

```

Beispiel 12: Abbildung von UML Klassen

Es wird gemäß der Abbildungsregel für jede Klasse in UML ein Element in XML-Schema mit den Attributen der Klasse als Unterelemente seines komplexen Typs angelegt. Über die Formulierung der `when`-Klausel, wird eine Abbildung jedes einzelnen Attributes durch die Regel `uml2xmlAttribute` erzwungen. Die Ober- und Untergrenzen der Kardinalitäten werden an jeweils eine Variable gebunden, so dass sie in der zweiten Domäne wiederverwendet und zugewiesen werden können. Gegenüber der Klasse nimmt das Attribut die Rolle `attribute` ein. Diese wird einer Attributinstanz gleichgesetzt. Das bedeutet sollte eine Klasse existieren, so muss sie für dieses Muster auch mit einer Menge von Attributen assoziiert sein, welche wiederum einen Namen und eine Multiplizität besitzen. Beispiel 13 zeigt die operationale Definition der Regel.

```

mapping in uml::Class::uml2xmlClass() : inout xml::Element {
  object element:Element {
    element.name = self.name;
    object attribute:Attr {
      attribute.name = self.attribute.name;
      attribute.map uml2xmlAttributeType();
    }
    element.attribute = attribute;
  }
}

```

Beispiel 13: Operationale Abbildung von UML-Klassen

Eine Assoziation in UML stellt eine Beziehung zwischen zwei oder mehr Klassen dar. Sie trägt einen Namen und Kardinalitäten für die beteiligten Partner. Man unterscheidet zwischen gerichteten Assoziationen, also solchen bei denen man ausgehend von einer Klasse die beteiligten Partner ermitteln kann und den ungerichteten Assoziationen, bei denen man von jeder Klasse aus die Partner ermitteln kann. Daher werden auch beide Möglichkeiten in separaten Abbildungsregeln behandelt. Beide Abbildungen benutzen den key-keyref-Mechanismus von XML-Schema um die Referenzen zu erzeugen.

Assoziation

Für jede Klasse in UML, die mit einer anderen Klasse über eine gerichtete Assoziation verbunden ist, wird für die Klassen ein Element in XML mit demselben Namen erzeugt, sofern das nicht schon geschehen ist. Die Assoziation wird über einen Verweis von der nicht navigierbaren Klasse aus aufgelöst. Die Multiplizitäten werden auf die Kardinalitäten der Verweise abgebildet.

*gerichtete
Assoziation*

```

relation uml2xmlDirectedAssociation(class , element) {
  checkonly domain uml1 c:Class { name = cName, assName(c, c1:
    Class{name = c1Name})}
  enforce domain xml1 e:Element { tagName = className}
  enforce domain xml1 e1t:Element { tagName = className}
  when {
    let keyName=cName+'directedAssociation'+c1Name+'_'+
      assName
    and makeKeyKeyRef(c, c1, e, e1, keyName, keyName)
  }
}

```

Beispiel 14: Abbildung von gerichteten Assoziationen

Die Abbildung verläuft wie bei der gerichteten Assoziation mit dem Unterschied, dass hier für jeden der Partner ein Verweis auf die jeweiligen erzeugten Elemente generiert wird. Durch das Konzept der Generalisierung in UML ist es möglich, eine Strukturhierarchie wie auch eine semantische Beziehung zwischen Klassen aufzubauen. Das bedeutet, dass wenn eine Klasse mit einer oder mehreren Klassen in einer Generalisierungsrelation steht, sie alle Attribute, Operationen und Assoziationen erbt, sie diese also besitzt bzw. an ihnen teilnimmt. Da eine Basisklasse, das bedeutet, die Klasse von der geerbt wird, ebenfalls abstrakt (nicht instanzierbar) sein kann, unterscheidet die Abbildung zwischen einer konkreten (nicht abstrakten) und einer abstrakten Basisklasse. Eine konkrete Basisklasse in UML kann instanziiert werden, daher wird sie wie jede andere konkrete Klasse von den Regeln für Klassen behandelt. Zur Abbildung der Vererbungsrelation wird für jede abgeleitete Klasse ein Verweis auf das entsprechende Basisklassenelemente erzeugt.

*ungerichtete
Assoziation
Generalisierung**konkrete
Basisklasse*

```

relation uml2xmlGeneralization(class , element) {
  checkonly domain uml1 c1:Class { name = generalizedClassName ,
    parent = c:Class{name=generalizingClassName , isAbstract='
    false ' , }}
  enforce domain xml1 e:Element {name = generalizingClassName}
}

```

```

enforce domain xml1 e1:Element {name = generalizedClassName}
when {
    makeKeyKeyref(c, c1, e, e1, generalizingClassName+
        '_general ',
        generalizingClassName + '_general ')
}

```

Beispiel 15: Abbildung der Generalisierung

Da eine abstrakte Basisklasse in UML nicht instanziiert ist, wird bei der Abbildung kein Element für diese Klasse in XML erzeugt. Stattdessen werden alle Eigenschaften und Assoziationen direkt in die erbende Klassen eingefügt.

*abstrakte
Basisklasse*

Die Aggregation und Komposition in UML ist eine Assoziation mit der die Teil-von-Beziehung ausgedrückt wird. Das bedeutet, dass eine Menge von Klasseninstanzen einer Klasse zugeordnet werden, welche aus diesen besteht. Der semantische Unterschied zwischen Aggregation und Komposition besteht darin, dass bei der Aggregation die aggregierende Klasse nicht ohne ihre Bestandteile existieren kann, bei der Komposition schon. Es gibt in XML kein vergleichbares Konzept, so dass Aggregation wie auch Komposition von einer Regel behandelt werden, welche das semantische Teil-von-Konzept erhält. Alle aggregierten Klassen werden auf Elemente abgebildet und als Unterelemente dem erzeugten Element der aggregierenden Klasse zugeordnet. Hierbei gelten natürlich die Regeln zur Abbildung der Klassen und ihre Attribute und Assoziationen.

*Aggregation
und
Komposition*

XML-Schema bietet zusammen mit XPath[XPath, 1999] (XML Path Language) einen *key-keyref*-Mechanismus, um innerhalb eines XML-Dokuments Verweise von einer Menge von Elementen zu einer Menge von Elementen zu definieren, basierend auf dem Schlüssel-Verweis-Prinzip. Hierbei ist XPath eine Sprache zur Adressierung von bestimmten Stellen innerhalb eines XML-Dokuments. Sie basiert auf einem Datenmodell, welches ein XML-Dokument als Baum auffasst und auf unterscheidbaren Knotentypen, die den jeweiligen XML-Konstrukten (z. B. Element, Attribut oder Kommentar) entsprechen.

*Verweise in
XML-Schema*

Diese Verweise innerhalb des Dokuments sind unidirektional und eindeutig. Es wird die Möglichkeit gegeben, den Gültigkeitsbereich des Schlüssels wie auch der Verweise zu spezifizieren, sowie deren Kardinalität. Hierzu definiert man in einem XML-Dokument ein *key*-Element und ein *keyref*-Element. Beide enthalten zwei Felder, das sogenannte *field* und den *selector*. Das *field* gibt das Attribut der entsprechenden Elemente an, von denen oder auf die verwiesen wird. Der *selector* hingegen spezifiziert über einen XPath-Ausdruck die entsprechenden Elemente für die der Schlüssel bzw. der Verweis gültig sein soll. Die Position innerhalb der Dokumenthierarchie der *key*- bzw. *keyref*-Elemente definiert den Gültigkeitsbereich.

4 Verwandte Arbeiten zu QVT

Im Jahr 2003 veröffentlichten Gardner et al. [Gardner et al., 2003] einen Beitrag, welcher die acht initialen Einreichungen zu QVT vergleicht, bewertet und eine Empfehlung für künftiges Vorgehen ausspricht. Zentraler Punkt ist die Notwendigkeit eines Standards zur Vereinheitlichung einer Definition von Modell-zu-Modell-Transformationen und den damit verbundenen Vorteilen eines einheitlichen Vokabulars. Es werden weitere Anforderungen an eine Sprache definiert wie Skalierbarkeit und Verständlichkeit. Des Weiteren sollte eine Sprache verschiedenen Transformationsszenarien genügen und somit uni- wie auch multidirektionale Transformationen ermöglichen. Sie kommen zu dem Schluss, dass eine hybride Transformationssprache die größtmögliche Flexibilität bietet. Zum einen sollte sie einen deklarativen Bestandteil für einfache und zum anderen eine imperativen für komplexere Transformationen beinhalten.

Gardner et al.

Neben Gardner et al., der die spezifischen Ansätze untersucht, haben Czarnecki und Helsen [Czarnecki u. Helsen, 2003] in ihrer Veröffentlichung den generischen Ansatz verfolgt und eine Klassifizierung von Transformationssprachen zur Modell-zu-Modell-Transformation angegeben. Sie unterscheiden zwischen direkter Manipulation, struktur-basierten Ansätzen, Graphentransformationen, relationalen Ansätzen und einer Mischform aus den genannten. Nach Czarnecki und Helsen stellt die direkte Manipulation, also die händische Implementierung, den offensichtlichen Ansatz dar. Da er jedoch auf eine spezifische Problemstellung und -domäne ausgerichtet ist, wenig generisches Potential bietet und die Wiederverwendbarkeit minimal ist, stellt dieser Ansatz langfristig keine Alternative dar. Die struktur-basierten Ansätze verfolgen eine Hierarchiebildung des Zielmodells und im Anschluss das Festlegen der Referenzen und Attribute der Modellelemente. Dieser Zwei-Phasen-Ansatz ist z. B. bei der Abbildung von UML-Modellen nach Datenbankschemata praktikabel. Inwiefern sich dies auf andere Domänen übertragen lässt, ist nach Meinung der Autoren ungewiss. Der graphenbasierte Ansatz bietet die größtmögliche Flexibilität und Ausdrucksmächtigkeit. Damit verbunden sind jedoch eine steigende Komplexität der Transformationsdefinition und eine erforderliche Erfahrung in der Regelbildung. Nach Czarnecki und Helsen stellt dies einen theoretisch fundierten Ansatz dar, bei dem die praktische Erfahrung noch evaluiert werden sollte. Relationale Ansätze scheinen den Mittelweg zwischen Komplexität und Flexibilität zu bilden, welches auch die Einreichungen für QVT belegen. Der hybride Ansatz ermöglicht den Einsatz verschiedener Techniken, abhängig vom Problemkontext und ist somit der wahrscheinlichste in der praktischen Anwendung, so Czarnecki und Helsen. Neben der Betrachtung der praktischen Relevanz von Transformationssprachen weisen die Autoren auf die Bedeutung der Performanz hin. Sie empfehlen die Entwicklung entsprechender Bewertungskriterien und Beispielprobleme, um auf deren Basis eine Aussage über die Sprache treffen zu können.

*Czarnecki
und Helsen*

5 Zusammenfassung und Schlussfolgerung

QVT ist eine metamodellbasierte, dreigeteilte Transformationssprache. Sie dient zur Definition einer multidirektionalen Transformation zwischen einer Menge von Modellen. Jedes Modell muss auf einem Metamodell basieren, welches selbst eine MOF-Instanz ist. Somit bietet QVT die Möglichkeit zur Modelltransformation zwischen beliebigen Modellinstanzen. QVT definiert Konzepte zur relationalen musterbasierten und operationalen Transformation. Die drei Bestandteile von QVT sind die relationale, die operationale und die *Core*-Sprache. Die relationale und *Core*-Sprache sind logikbasierte Sprachen, in denen die Transformation als eine Menge von Relationen definiert wird. Jede Relation setzt sich aus einer Menge von Mustern über Modellelementen der jeweiligen Metamodelle zusammen. Die operationale Sprache dient zur Spezifikation des Transformationsprozesses ebenso wie die nutzerdefinierte Variante, die *BlackBox*. Die drei von QVT definierten Sprachen basieren alle auf einer im Metamodell beschriebenen abstrakten Grammatik, welche OCL 2.0 erweitert.

Eine Transformation bzw. Abbildung wird in QVT mit Hilfe der relationalen Sprache definiert. Hierzu verwendet man die graphische oder textuelle Notation zur Definition der Relationen. Im Anschluss kann man mit der operationalen Sprache die Ausführung bzw. Implementierung der Transformation genauer spezifizieren. Neben dem Standardfall der operationalen Sprache gibt es auch die Möglichkeit, den nutzerdefinierten Ansatz die *BlackBox* zu verwenden. Hier ist es möglich, bestehende Transformationsimplementierungen und Bibliotheken wiederzuverwenden sowie andere Transformationsansätze zu integrieren. QVT definiert neben Konzepten zur Transformation ein *Trace*-Modell zur Protokollierung des Transformationsprozesses, das ermöglicht eine inkrementelle Transformation und eine Integration in den Entwicklungsprozess.

Nachdem ich die Entstehung von QVT umrissen und einige alternative Arbeiten, wie M2T oder MTF aufgezeigt habe, wurden die Architektur von QVT sowie ihre Komponenten beschrieben. Jede von ihnen habe ich zusammengefasst und die Konzepte exemplarisch erläutert. Nach der Beschreibung der Sprachbestandteile habe ich den praktischen Einsatz von QVT an der Abbildung von UML nach XML gezeigt und relevante Aspekte beschrieben und erläutert.

Von den eingangs im Abschnitt 1.2 definierten Kriterien erfüllt QVT den Großteil. So ist es eine formal auf OCL 2.0 definierte Sprache, die Konzepte zur Transformation wie auch Abbildung auf operationaler Basis bietet. Mit Hilfe von QVT ist es möglich Muster zu spezifizieren und so in den zu transformierenden Modellen zu navigieren. Des Weiteren wird auch die Möglichkeit geboten von einer Menge von Modellen in eine Menge von Modellen abzubilden. Während des praktischen Einsatzes gab es keinen Punkt an dem QVT ein Konzept fehlte, um die benötigten Informationen zu gewinnen oder darzustellen, so dass es in diesem Punkt als vollständig betrachtet werden kann. Aufgrund fehlender Implementierungen und daher Werkzeugunterstützung gibt es keine Möglichkeit QVT tatsächlich in der Praxis einzusetzen.

Im Laufe der Einarbeitung in QVT und seinem Einsatz an der Abbildung von UML nach XML kristallisierten sich zwei Aspekte heraus. Voraussetzung zur Arbeit mit QVT ist eine hinreichende Kenntnis von OCL, denn insbesondere die operationale Sprache

macht ausgiebig Gebrauch von OCL-Konstrukten. Des Weiteren ist es notwendig, Kenntnis von MOF wie auch der Modellelementkonzepte zu haben. Mit der operationalen und relationalen Sprache bietet QVT zwei gut zu handhabende Werkzeuge zur Transformationsdefinition.

Der Standardentwurf als Dokument weist an einigen Stellen Schwächen auf. So fehlt die exakte Definition von Begriffen bzw. ihrer Repräsentation. Beispiele sind teilweise fehlerhaft oder unvollständig. Durch die Einführung der *BlackBox* schlägt QVT den Bogen zu existierenden Lösungen und berücksichtigt sie somit. Insgesamt wird QVT damit eine mächtige und ausdrucksstarke Transformationssprache im generischen Einsatzfeld. Als ein kommender Standard der OMG bietet QVT eine standardisierte Methode zur Transformationsdefinition, welche sich in die Sprachfamilie von ULF eingliedern lässt, denn mit ihrer Hilfe ist es möglich, Transformationen zwischen beliebigen metamodellbasierten Sprachen zu definieren.

Das Konzept der Muster und ihrer Spezifikation sind leicht zugänglich und intuitiv einsetzbar, bedingt durch die Anlehnung an existierende Programmiersprachen. Auch im Einsatz an der Abbildung von UML nach XML traten keine Schwierigkeiten auf. Für jedes benötigte Konzept gab es ein Strukturelement in QVT. Alle Kriterien zum Einsatz von QVT als Sprache im Kontext von ULF-Ware sind erfüllt, abgesehen von einer Implementierung. Diese Schwäche macht es unmöglich, QVT produktiv einzusetzen, so dass es insbesondere für ULF-Ware zwar zur formalen Spezifikation von Abbildungen eingesetzt werden kann, jedoch nicht in der Entwicklungsumgebung selbst einsetzbar ist. Sollte in Zukunft eine entsprechende Implementierung verfügbar sein, ist eine Integration in ULF abzuwägen, da insbesondere Performanzaspekte evaluiert werden müssen.

Literatur

- [Böhme et al. 2005] BÖHME, Harald ; SCHÜTZE, Glenn ; VOIGT, Konrad: Component Development: MDA Based Transformation from eODL to CIDL. In: *SDL 2005: Model Driven*, Springer-Verlag GmbH, Juni 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26612–7, S. 68–84
- [CIDL 2002] CIDL: *CORBA Component Model, Version 3.0*. Object Management Group, 2002. – formal/2002-06-70
- [Czarnecki u. Helson 2003] CZARNECKI, Krzysztof ; HELSON, Simon: Classification of Model Transformation Approaches. In: *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003
- [Eckstein u. Eckstein 2004] ECKSTEIN, Dr. R. ; ECKSTEIN, Dr. S.: *XML und Datenmodellierung*. dpunkt.verlag GmbH, 2004
- [Eclipse 2005] ECLIPSE: *Eclipse SDK*. eclipse.org, 2005. – <http://www.eclipse.org>
- [EMF 2005] EMF: *Eclipse Modeling Framework*. eclipse.org, 2005. – <http://www.eclipse.org/emf/emf.php>
- [Fischer et al. 2005] FISCHER, Joachim ; KUNERT, Andreas ; PIEFEL, Michael ; SCHEIDGEN, Markus: ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages. In: *SDL 2005: Model Driven*, Springer-Verlag GmbH, Juni 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26612–7, S. 1–15
- [Fischer et al. 2004] FISCHER, Joachim ; PIEFEL, Michael ; SCHEIDGEN, Markus: A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In: *System Analysis and Modeling: 4th International SDL and MSC Workshop*, Springer-Verlag GmbH, Juni 2004 (Lecture Notes in Computer Science)
- [Gardner et al. 2003] GARDNER, Tracy ; GRIFFIN, Catherine ; KOEHLER, Jana ; HAUSER, Rainer: A review on OMG MOF 2.0 Query/View/Transformation Submissions and Recommendations towards the final Standard. In: *MetaModelling for MDA, Workshop*, 2003
- [Griffin 2004] GRIFFIN, Catherine: *Model Transformation Framework*. 2004. – <http://www.alphaworks.ibm.com/tech/mtf>
- [ITU-T Z.130 2003] ITU-T Z.130: *Extended Object Definition Language (eODL)*. International Telecommunication Union, 2003
- [MDA 2003] MDA: *Model Driven Architecture Guide, Version 1.0.1*. Object Management Group, 2003. – omg/03-06-01
- [MODFact 2003] MODFACT: *ModFact*. LIP6, 2003. – <http://modfact.lip6.fr/>

-
- [MOF 2003] MOF: *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group, 2003. – formal/06-01-01
- [OCL 2003] OCL: *UML 2.0 OCL Specification*. Object Management Group, 2003. – formal/06-05-01
- [Paradigm 2005] PARADIGM, Visual: *Visual Paradigm 5.0 Community Edition*. 2005. – <http://www.visual-paradigm.com>
- [QVT 2002] QVT: *Submission for MOF 2.0 Query/Views/Transformations RFP*. Object Management Group, 2002. – ad/2002-04-10, Request for Proposal
- [QVT 2005] QVT: *Revised submission for MOF 2.0 Query/View/Transformation RFP*. Object Management Group, 2005. – ad/2002-04-10 version 2.1, Request for Proposal
- [UML 2003] UML: *Unified Modeling Language, Version 1.5*. Object Management Group, 2003. – formal/2003-03-01
- [XMI 2002] XMI: *XML Metadata Interchange, Version 1.2*. Object Management Group, 2002. – formal/2002-01-01
- [XML-Schema 2000] XML-SCHEMA: *XML Schema Specification*. W3C, 2000. – <http://www.w3.org/XML/Schema>
- [XPath 1999] XPATH: *XML Path Language (XPath) Version 1.0*. W3C, 1999. – <http://www.w3.org/TR/xpath>

Abbildungsverzeichnis

1	QVT Konzeptbeziehung	6
2	QVT Sprachbeziehungen	7
3	Einfache graphische Transformation	13
4	Relationale graphische Negation	14
5	Transformation von UML Paketen	21

Beispielverzeichnis

1	Transformationsregel in QVT	9
2	Einfache Relation in QVT	10
3	Top-Relation in QVT	10
4	When und Where Klauseln in QVT	11
5	Enforce und checkonly in QVT	12
6	Die operationale Transformation	15
7	Operationale Abbildung von Paketen	15
8	Abgekürzte operationale Abbildung von Paketen	16
9	Die Transformationsregel	20
10	Die operationale Transformationsregel	20
11	Abbildung von Paketen	22
12	Abbildung von UML Klassen	23
13	Operationale Abbildung von UML-Klassen	23
14	Abbildung von gerichteten Assoziationen	24
15	Abbildung der Generalisierung	24