INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Projektarbeit

Entwicklung eines in Eclipse integrierten Code Editors für die Programmierung in QVT-R

Thomas Klutsch

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Projektarbeit

Entwicklung eines in Eclipse integrierten Code Editors für die Programmierung in QVT-R

Thomas Klutsch

Aufgabensteller: Prof. Dr. Martin Wirsing

Betreuer: Dr. Philip Mayer

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 10. Februar 2013

.....

(Unterschrift des Kandidaten)

Zusammenfassung

Diese Arbeit beschreibt die Implementierung eines in die Entwicklungsumgebung Eclipse integrierten Editors zur Unterstützung des Entwurfs von Modelltransformationen in der QVT Relations Language, wobei – im Gegensatz zu anderen verfügbaren Tools – der Fokus auf der Separierung von Editorfunktionalität und semantischem Modell gegenüber der Ausführung der Transformationen liegt. Es werden alle grundlegenden Funktionen eines modernen Quelltext-Editors unterstützt: Die syntaktische Validierung des Quelltexts mit Anzeige von Fehlermeldungen, eine Unterstützung der Eingabe des Quelltexts (Code Assist/Code Completion), eine grafische Hervorhebung von Teilen des Codes (Syntax Highlighting) sowie eine Funktion zur Anpassung des eingegebenen Texts an das für QVT-R übliche Code-Muster (Quellcode-Formatierung). Für die Realisierung des Editors und der ihm zugrunde liegenden QVT-R Sprach-Spezifikation wurde das Sprach-Entwicklungs-Framework Xtext verwendet, welches neben der Unterstützung bei der Implementierung der genannten Funktionen auch ein Meta-Modell der modellierten Sprache bereitstellt.

Inhaltsverzeichnis

1 Einleitung

Einen wesentlichen Bestandteil des Prozesses innerhalb der modellgetriebenen Softwareentwicklung stellen die sogenannten Modelltransformationen [ITW13] dar. Dabei wird aus einem Quell-Modell mithilfe von auf Meta-Modellen basierenden Regeln ein Ziel-Modell erzeugt beziehungsweise so abgeändert, dass es mit dem Quell-Modell konsistent ist.

Eine der bekannteren Sprachen für die Modellierung solcher Modelltransformationen ist die Query/View/Transformation Relations Language (QVT-R) [Obj11a]. Wie aus dem Namen hervorgeht werden die Transformationen hierbei über eine Menge an Relationen zwischen Quell- und Ziel-Modell definiert. QVT-R ist eine deklarative Sprache und erlaubt auch die Abbildung bidirektionaler Transformationen; also Transformationen nicht nur von Quell- zu Ziel-Modell hin, sondern auch anders herum.

Die QVT Relations Language bildet die Grundlage dieser Arbeit, wobei hier auf Basis einer leicht angepassten (im Grunde genommen: vereinfachten) Form von QVT-R gearbeitet wird. Für diese soll nun ein passender Code Editor zur Unterstützung des Anwenders implementiert werden. Für die Realisierung dieses Editors wird das Sprach-Entwicklungs-Framework Xtext [Ecl13d] verwendet.

1.1 Motivation

Es existieren bereits einige Tools und Editoren, die QVT-R – mehr oder weniger vollständig – implementieren. So zum Beispiel ModelMorf (entwickelt von Tata Research Development and Design Centre) [Tat07], oder mediniQVT (ikv++ technologies) [ikv13]. Letzteres wird auch als Eclipse Plug-In angeboten; wobei auch die Eclipse Foundation selbst ein Projekt unter dem Namen MMT (Model To Model Transformation; ursprünglich: M2M [Ecl13c]) führt, welches als Teil des Eclipse Modeling Project eine QVT-R Implementierung enthalten soll.

Diese Produkte sind für den hier vorliegenden Anwendungsfall jedoch wenig geeignet. Zum einen werden sie teilweise nur kommerziell vertrieben, was nicht wünschenswert ist. Das größere Problem aber stellt die Tatsache dar, dass diese Tools erstens ihren Schwerpunkt auf die Transformation der Modelle setzen; während die Anforderung sich hier mehr auf die Abbildung von Modellrelationen, also das bloße "Model Checking" ohne Output eines transformierten Modells konzentriert. Zweitens ist bei diesen Tools rein sprachlich keine Individualisierung vorgesehen, womit sich die hier gewünschte Anpassung des QVT-R Sprach-Standards als entweder gar nicht möglich, oder nur mit großem Aufwand machbar herausstellen würde. Aus diesen Gründen kommen diese bereits verfügbaren Lösungen nicht als Kandidat für diese Arbeit in Frage.

Erwünscht ist hier vielmehr ein Tool, welches erstens weitestgehend unabhängig von Transformationen arbeitet, zweitens sich in Bezug auf die zugrunde liegende QVT-R Sprach-Spezifikation einfach individualisieren lässt, und drittens kostenfrei und als Open Source Software verfügbar sein soll.

1.2 Ansatz

Eine grundsätzlich immer vorhandene Option ist die Verwirklichung der Lösung "from scratch". In diesem Fall wäre das jedoch ein viel zu hoher Arbeitsaufwand. Die (Nach-)Implementierung der QVT-R Sprach-Spezifikation ist dabei nicht das größte Problem, sondern vielmehr die Implementierung eines entsprechenden Compilers sowie eines Code Editors mit all den Funktionen, die man heute von einem solchen erwarten würde.

An dieser Stelle kommt nun als weitaus angenehmere Alternative das Tool Xtext zum Einsatz. Xtext ist ein frei verfügbares Eclipse Plug-In, welches dem Anwender die Möglichkeit zur komfortablen Modellierung einer eigenen Sprache bietet, wobei Xtext im Zuge der Sprachkompilierung den benötigten Compiler und gewünschten Code Editor automatisch miterzeugt.

Neben dieser Hauptanforderung spricht für Xtext zudem die Tatsache, dass es Bestandteil des Eclipse Modeling Framework (EMF) [Ecl13b] ist. Da EMF häufig zur Definition von Meta-Modellen für Modelltransformationen verwendet wird, stellt Xtext ein einfach in die bereits bestehende Entwicklungsumgebung integrierbares Begleittool zur modellgetriebenen Softwareentwicklung dar.

1.3 Gliederung

Auf den folgenden Seiten wird nun zunächst ein genauerer Einblick in die QVT Relations Language sowie Xtext gegeben (Kapitel "Hintergrund"), um ein Grundverständnis für diese Technologien und somit auch die konkrete Arbeit zu schaffen.

Anschließend wird sich in Kapitel 3 dann dem konkreten Implementierungsvorgang von QVT-R in Xtext und dem zugehörigen Code Editor gewidmet.

Das vierte und letzte Kapitel gibt abschließend eine Zusammenfassung der Arbeit, sowie einige Aussichten darüber, was man an dem Tool in Zukunft noch erweitern und von ihm erwarten kann.

2 Hintergrund

Dieses Kapitel soll die beiden dieser Arbeit zugrunde liegenden Technologien – QVT-R und Xtext – genauer vorstellen, um das nötige Rahmenverständnis für die im nächsten Kapitel folgende Implementierung zu schaffen. Die Erläuterung der Architektur und Funktionsweise wird zum besseren Verständnis jeweils von einem kleinen Beispiel begleitet.

Es tauchen in diesem Zuge auch die Namen ein paar weiterer, implizierter Technologien auf, welche hier jedoch nebensächlich und daher nur der Vollständigkeit halber referenziert sind. Der Leser möge sich das entsprechende Hintergrundwissen zu diesen Technologien bei Bedarf selbstständig aneignen.

2.1 QVT-R

QVT steht für Query/View/Transformation und bezeichnet ein von der Object Management Group (OMG) entwickeltes Sprach-Set zur Modellierung von Modelltransformationen. Die QVT Spezifikation ist Teil der Meta Object Facility (MOF) [Obj13a], einer größeren Sammlung von Dokumenten der OMG zur formalen Definition von Modellen sowie deren Anwendung. QVT integriert den OCL-Standard (Object Constraint Language [Obj13b]), welcher einen allgemeinen, grundlegenden Rahmen für Modellierungssprachen festlegt und wiederum Teil der Unified Modeling Language (UML [Obj13d]) ist. QVT wurde in der Version 1.0 im Jahre 2008 veröffentlicht; die derzeit aktuelle Version ist 1.1 (erschienen 2011).

QVT besteht aus drei verschiedenen, jedoch miteinander verknüpften Sprachen: Der QVT Core Language, der QVT Operational Mappings Language, und der bereits genannten QVT Relations Language (QVT-R). Die Core Language und die Relations Language sind sich ähnelnde, deklarative Sprachen, wobei die Relations Language als eine Art benutzerfreundlichere Version der Core Language zu verstehen ist, die gewisse Schritte in der Ausführung einer Transformation implizit vornimmt, und mit der sich die Transformationen im Vergleich zur Core Language etwas leichter modellieren lassen. Die QVT Architektur enthält eine Schnittstelle zur Überleitung von Relations zu Core Language, bei der die erwähnten impliziten Schritte innerhalb der Relations Language zu den entsprechenden von der Core Language explizit geforderten Spezifikationen übertragen werden. Die Operational Mappings Language unterscheidet sich von Core und Relations Language dadurch, dass die Transformationen hier nicht deklarativ, sondern imperativ definiert werden. Auch hier ist eine Überleitung zur Core Language möglich; entweder direkt, oder über eine zwischengeschaltete Überleitung in die Relations Language.

Neben diesen drei Sprachen enthält die QVT Architektur noch einen weiteren Bestandteil: Die "Black Box", welche als Schnittstelle für die Integration externer, MOF-kompatibler Bibliotheken in QVT dient.

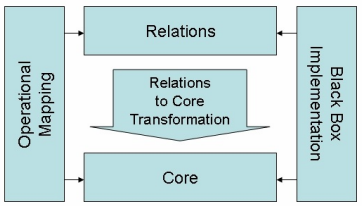

Abbildung 1: QVT Architektur
(Quelle: http://de.wikipedia.org/wiki/MOF_QVT)

Für diese Arbeit sind die neben der Relations Language existierenden Teile von QVT nicht von Interesse, daher wird im Weiteren lediglich der Einblick in die Relations Language vertieft.

2.1.1 Beispiel

Das folgende Beispiel einer QVT-R-Transformation basiert auf einem Beispiel aus der offiziellen Dokumentation [Obj11b], und modelliert eine Transformation zwischen UML Meta-Klassen und Tabellen einer relationalen Datenbank. Abbildungen 2 und 3 zeigen die der Transformation zugrunde liegenden Meta-Modelle als UML-Diagramm.

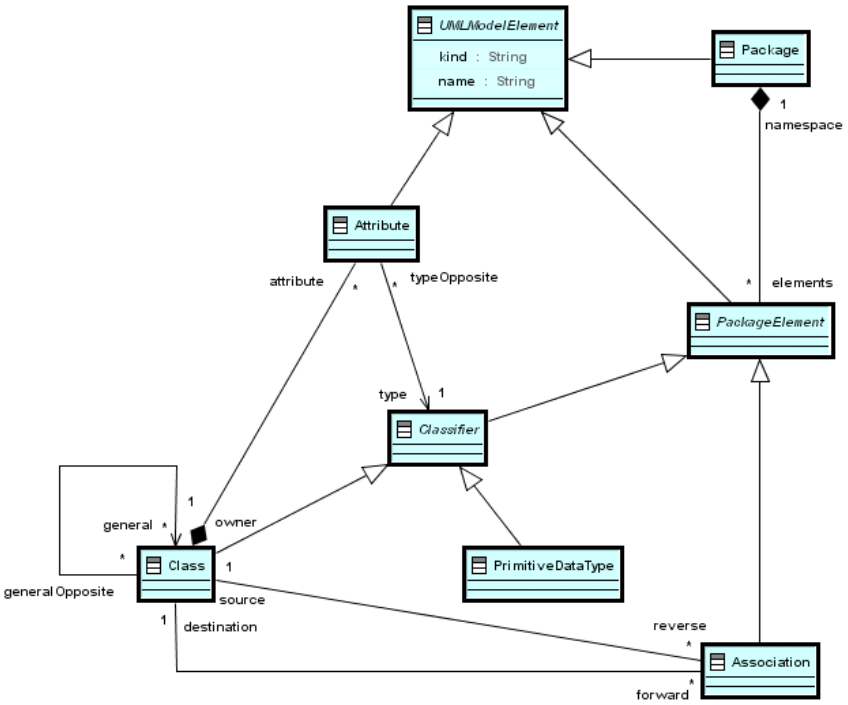

Abbildung 2: SimpleUML Meta-Modell
(Quelle: http://www.omg.org/spec/QVT/1.1/PDF/)

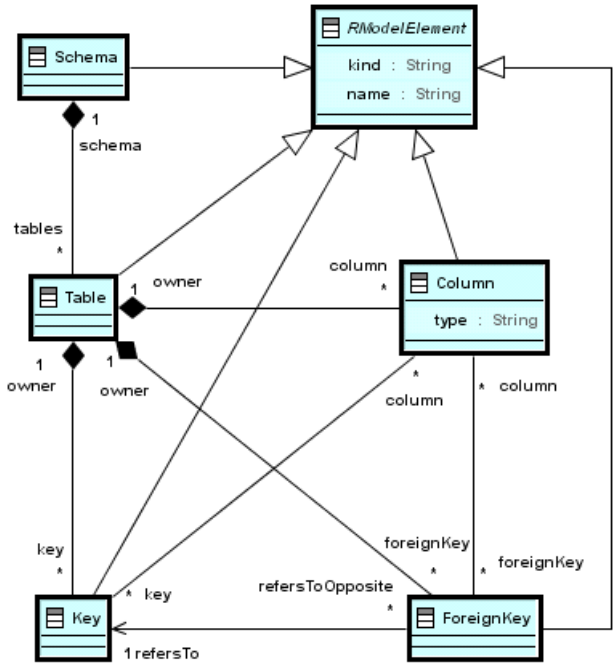

Abbildung 3: SimpleRDBMS Meta-Modell
(Quelle: http://www.omg.org/spec/QVT/1.1/PDF/)

Die auf diesen Meta-Modellen im Dokumentationsbeispiel definierte Transformation vollständig abzubilden würde mehrere Seiten füllen. Zur Verringerung der Komplexität stellt der nachfolgende Code daher eine stark vereinfachte Form der vollständigen Transformation dar, in der ein Großteil der Relationen weggelassen wurde. Es fehlen damit die Mappings zwischen den meisten der Klassen innerhalb der Meta-Modelle. Da dieses Beispiel nur zur Veranschaulichung dient und der praktische Sinn davon zweitrangig ist, und zudem die Logik hinter der Ausführung von Relationen immer die selbe ist, sollte diese Vereinfachung jedoch keinen Einfluss auf das Verständnis von Syntax und Semantik der Sprache haben:

```
1  transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS){
2
3     key Table (name, schema);
4     key Column (name, owner);
5     key Key (name, owner);
6
7     top relation PackageToSchema{
8        pn: String;
9        checkonly domain uml p:Package {name=pn};
10       enforce domain rdbms s:Schema {name=pn};
11    }
12
```

```
13     top relation ClassToTable{
14        cn: String;
15        checkonly domain uml c:Class{
16           namespace=p:Package {},
17           kind='Persistent',
18           name=cn
19        };
20        enforce domain rdbms t:Table{
21           schema=s:Schema {},
22           name=cn,
23           column=cl:Column {name=cn+'_tid', type='NUMBER'},
24           key=k:Key {name=cn+'_pk', column=cl}
25        };
26        when{
27           PackageToSchema(p,s);
28        }
29        where{
30           AttributeToColumn(c, t);
31        }
32     }
33
34     relation AttributeToColumn{
35        an, pn, cn, sqltype: String;
36        checkonly domain uml c:Class{
37           attribute=a:Attribute{
38              name=an,
39              type=p:PrimitiveDataType {name=pn}
40           }
41        };
42        enforce domain rdbms t:Table{
43           column=cl:Column {name=cn, type=sqltype}
44        };
45        where{
46           cn = an;
47           sqltype = PrimitiveTypeToSqlType(pn);
48        }
49     }
50
51     function PrimitiveTypeToSqlType(primitiveTpe:String)
52      :String{
53     if (primitiveType='INTEGER')
54        then 'NUMBER'
55        else if (primitiveType='BOOLEAN')
56           then 'BOOLEAN'
57           else 'VARCHAR'
58           endif
59        endif;
60     }
61  }
```

Die Signatur einer Transformation besteht also aus einem Bezeichner ("umlToRdbms") und einer Menge an Definitionen von Meta-Modellen, auf deren Instanzen die Transformation angewandt werden soll. Hier werden zwei Modelle erwartet: Eines aus dem SimpleUML Meta-Modell, referenziert über den Namen "uml", und eines aus dem SimpleRDBMS Meta-Modell ("rdbms").

Innerhalb der Transformation sind Relationen definiert, welche wiederum eine Menge von Domains enthalten. Die Domains definieren über sogenannte (Property-) Templates bestimmte Bedingungen für die Instanzen der in der Domain verwendeten Meta-Modell-Klassen, die alle erfüllt sein müssen, damit die Relation selbst erfüllt ist. Für die Modellierung der Bedingungen lassen sich Variablen verwenden, welche in der Relation oder in den Domains selbst deklariert sind und zur Angabe von Mappings zwischen Werten von Attributen verschiedener Meta-Modell-Klassen und deren Zwischenspeicherung zur Laufzeit genutzt werden.

Bei der Auswertung einer Relation muss auf die Ausführungsrichtung der Transformation, sowie auf die Art der Domains geachtet werden: Die Ausführungsrichtung einer Transformation gibt an, welches Modell das Quell- und welches das Ziel-Modell ist. Sie wird erst zur Laufzeit durch den Anwender bestimmt und spielt eine Rolle für die Art und Weise, in der die Domains der Relation evaluiert werden.

Mit dem Schlüsselwort "checkonly" versehene Domains bedeuten dabei für die Transformation, dass diese scheitert sofern die in der Domain gestellten Bedingungen nicht zutreffen und das in der Domain spezifizierte Modell das Ziel-Modell der Transformation ist. Bei "enforce" Domains dahingegen wird in solch einem Fall versucht, das Ziel-Modell so abzuändern, dass die Bedingungen in der Domain erfüllt sind.

Angenommen, die Transformation würde von SimpleUML hin zu SimpleRDBMS ausgeführt, das heißt SimpleUML ist das Quell-, und SimpleRDBMS das Ziel-Modell, dann wäre demnach beispielsweise die Relation "PackageToSchema" wie folgt zu verstehen: Für jede Instanz des Typs "Package" aus dem der Transformation übergebenen SimpleUML-Modell, welche einen bestimmten, in die Variable "pn" zwischengespeicherten Wert für das Attribut "name" hat, muss es auch eine Instanz vom Typ "Schema" aus dem SimpleRDBMS-Modell geben, deren Attribut "name" diesem Wert (von "pn") gleicht. Ist dies nicht der Fall, wird eine solche Schema-Instanz erzeugt ("enforce").

Hätte dahingegen die Transformation die entgegengesetzte Ausführungsrichtung, also von SimpleRDBMS hin zu SimpleUML, würde die Relation wie folgt evaluiert: Für jede Instanz des Typs "Schema" aus dem SimpleRDBMS-Modell, welche einen bestimmten, in die Variable "pn" zwischengespeicherten Wert für das Attribut "name" hat, muss es auch eine Instanz des Typs "Package" aus dem SimpleUML-Modell geben, deren Attribut "name" diesem Wert (von "pn") gleicht. Da die Domain des Ziel-Modells in diesem Fall als "checkonly" definiert ist, würde für den Fall, dass es kein passendes Package gibt, nicht versucht werden, eines zu erzeugen. Stattdessen würde die Transformation fehlschlagen und eine Inkonsistenz zwischen den Modellen melden.

Neben Meta-Modell-basierten Domains existieren außerdem "primitive" Domains, welche es ermöglichen auf primitiven (OCL-)Typen wie String oder Integer zu arbeiten. Solche Domains kommen in dem hier gezeigten Beispiel jedoch nicht vor.

Nicht nur Domains lassen sich unterscheiden, auch Relationen: Es existieren Top-Level-Relationen und Non-Top-Level-Relationen. Top-Level-Relationen sind mit dem Keyword "top" versehen und werden bei Ausführung der Transformation jeweils für jede im Quell-Modell existierende Instanz, für die eine Domain in der Relation definiert ist, automatisch ausgeführt. Non-Top-Level-Relationen dahingegen werden nur dann ausgeführt, wenn sie im Code explizit aufgerufen werden. Der Aufruf einer Relation geschieht innerhalb der "when"- oder "where"-Klausel einer anderen Relation.

Mit "when"/"where"-Klauseln lassen sich die Anforderungen an die Erfüllung der in der Relation durch die Domains gestellten Bedingungen an weitere Bedingungen knüpfen. Dabei muss eine Relation nur dann erfüllt sein, wenn alle in der when-Klausel aufgerufenen Relationen auch erfüllt sind. So gilt beispielsweise für die Relation "ClassToTable", dass die in ihr spezifizierten Konsistenzen zwischen Instanzen der Meta-Modell-Klassen "Class" und "Table" nur dann gelten müssen, wenn der Name des Packages der Class dem Namen des Schemas der Table gleicht. In der "where"-Klausel aufgerufene Relationen sind als Begleitbedingung für die Relation zu verstehen. In diesem Fall bedeutet das, dass alle durch die Relation "AttributeToColumn" spezifizierten Abhängigkeiten zwischen jeweiligem Table und Class gelten müssen, sofern die Relation zwischen Table und Class selbst erfüllt ist.

"when"- und "where"-Klauseln müssen nicht auf den Aufruf von Relationen beschränkt sein. Sie können jede beliebige OCL Expression enthalten, beispielsweise eine Zuweisung zu einer Variablen oder komplexere Kontrollstrukturen wie if-then-else-Statements.

Eine Transformation kann abgesehen von Relationen noch zwei weitere Dinge enthalten: "Keys" und "Functions".

Über einen Key lässt sich spezifizieren, wie zwei Instanzen des selben Meta-Modell-Typs beim Abgleich gegeneinander gewertet werden; ob sie also "matchen" oder nicht. Wie auch beispielsweise bei Datenbanken gibt man dabei diejenigen Attribute der dem Key zugehörigen Meta-Modell-Klasse an, über welche eine Instanz dieser Klasse eindeutig identifiziert werden soll. Dies ist im Falle von "enforce" Domains wichtig, um die Erzeugung von Duplikaten im Ziel-Modell zu vermeiden.

Functions schließlich sind als Hilfsfunktionen zu verstehen, die innerhalb von Relationen aufgerufen werden können. Sie bieten sich an, um wiederkehrenden und/oder komplexeren Code aus einer oder mehreren Relationen auslagern zu können. Zudem sind Functions Einhakpunkt für die QVT Black Box Implementierungen, über die eine Menge an externem Code in die Transformation eingekoppelt werden kann. In diesem Beispiel wird eine Function zur Überführung eines Datentypen aus dem UML Meta-Modell in den entsprechenden Datentypen einer relationalen Datenbank verwendet.

2.1.2 Spezifikation

Die folgenden Listings zeigen die textuelle Spezifikation der QVT-R Concrete Syntax aus der offiziellen Dokumentation [Obj11b], bestehend aus der Relations Textual Syntax Grammar:

```
1  <topLevel> ::= ('import' <unit> ';')* <transformation>*
2
3  <unit> ::= <identifier> ('.' <identifier>)*
4
5  <transformation> ::= 'transformation' <identifier>
6     '(' <modelDecl> (',' <modelDecl>)* ')'
7     ['extends' <identifier>]
8     '{' <keyDecl>* ( <relation> | <query> )*
9
10 <modelDecl> ::= <modelId> ':' ( <metaModelId |
11    '{' <metaModelId> (',' <metaModelId>* '}' )
12
13 <modelId> ::= <identifier>
14
15 <metaModelId> ::= <identifier>
16
17 <keyDecl> ::= 'key' <classId>
18    '{' <keyProperty (, <keyProperty>* '}' ';'
19
20 <classId> ::= <pathNameCS>
21
22 <keyProperty> ::= <identifier> |
23    'opposite' '(' <classId> '.' <identifier ')'
24
25 <relation> ::= ['top'] 'relation' <identifier>
26    ['overrides' <identifier>]
27    '{'
28    <varDeclaration>*
29    (<domain> | <primitiveTypeDomain>)+
30    [<when>] [<where>]
31    '}'
32
33 <varDeclaration> ::= <identifier> (, <identifier>)*
34    ':' <TypeCS> ';'
35
36 <domain> ::= [<checkEnforceQualifier>]
37    'domain' <modelId> <template>
38    ['implementedby' <OperationCallExpCS>]
39    ['default_values' '{' (assignmentExp)+ '}']
40    ';'
41
42 <primitiveTypeDomain> ::= 'primitive' 'domain'
43    <identifier> ':' <TypeCS> ';'
44
```

```
<checkEnforceQualifier> ::= 'checkonly' | 'enforce'

<template> ::= (<objectTemplate> | <collectionTemplate>)
   ['{' <OclExpressionCS> '}']

<objectTemplate> ::= [<identifier>] ':' <pathNameCS>
   '{' [<propertyTemplateList>] '}'

<propertyTemplateList> ::= <propertyTemplate>
   (',' <propertyTemplate>)*

<propertyTemplate> ::= <identifier> '=' <OclExpressionCS>
   | 'opposite' '(' <classId> '.' <identifier> ')'
     '=' <OclExpressionCS>

<collectionTemplate> ::= [<identifier>] ':'
   <CollectionTypeIdentifierCS> '(' <TypeCS> ')'
   '{' [<memberSelection>] '}'

<memberSelection> ::= (<identifier> | <template> | '_')
   (',' (<identifier> | <template> | '_'))*
   '++'
   (<identifier> | '_')

<assignmentExp> ::= <identifier> '=' <OclExpressionCS> ';'

<when> ::= 'when' '{' (<OclExpressionCS> ';')* '}'

<where> ::= 'where' '{' (<OclExpressionCS> ';')* '}'

<query> ::= 'query' <identifier>
   '(' [<paramDeclaration> (',' <paramDeclaration>)*] ')'
   ':' <TypeCS>
   (';' | '{' <OclExpressionCS> '}')

<paramDeclaration> ::= <identifier> ':' <TypeCS>
```

sowie der Expressions Syntax (OCL Extensions):

```
<OclExpressionCS> ::= <PropertyCallExpCS>
   | <VariableExpCS>
   | <LiteralExpCS>
   | <LetExpCS>
   | <IfExpCS>
   | '(' <OclExpressionCS> ')'
   | <template>
```

Da die OCL Extensions lediglich eine Schnittstelle zu OCL darstellen und nicht Teil der QVT-R Sprache selbst sind, wird an dieser Stelle nicht weiter auf deren Definition eingegangen. Sie sind der Concrete Syntax der OCL Dokumentation [Obj13c] zu entnehmen.

2.1.3 Fehler in der Spezifikation

Das dargelegte Beispiel entspricht nicht exakt dem aus der QVT Spezifikation, allerdings kommen die meisten der hier gezeigten Code-Zeilen so auch im vollständigen Beispiel aus der Spezifikation vor. Dieses Beispiel enthält semantische und syntaktische Fehler, von denen auch einige in den obigen Code übernommen wurden. Der Vollständigkeit halber und zur Aufklärung eventuell auftretender Mißverständnisse soll an dieser Stelle auf diese Fehler hingewiesen werden:

Im SimpleRDBMS Meta-Modell referenzieren die Meta-Klassen "Table" und "Column" jeweils eine Menge an Instanzen des Typs "Key". Der Bezeichner für diese Referenz lautet "key", welcher so auch im QVT-R-Code an zwei Stellen des vollständigen Beispiels aus der Dokumentation, und an einer Stelle im hier gezeigten Code (Zeile 24) verwendet wird. Da "key" aber laut Spezifikation der Concrete Syntax ein reserviertes Keyword ist (siehe "KeyDeclaration" im Listing zur Concrete Syntax), würde das Verwenden dieses Tokens im Code an jeder Stelle, die nicht eine KeyDeclaration darstellt, einen Syntax-Fehler hervorrufen. So, wie die Concrete Syntax spezifiziert ist, kann zur Modellierung einer Transformation kein Meta-Modell verwendet werden, deren Meta-Klassen in ihren Attributen – direkt oder referenziert – den Bezeichner "key" verwenden.

Des Weiteren wird im vollständigen Spezifikationsbeispiel innerhalb einer Relation namens "AssocToFKey" eine Domain über dem Typ "ForeignKey" defininert, welche ein Attribut des Typs "Schema" referenziert. Eine Referenz der zugrunde liegenden Meta-Klasse ForeignKey auf die Meta-Klasse Schema ist in dem zugehörigen UML-Diagramm allerdings nicht eingezeichnet.

Als Letztes enthält die in der Transformation definierte Function einen kleinen Flüchtigkeitsfehler: In ihrer Signatur wird ein Parameter namens "primitiveTpe" definiert; im Rumpf soll dieser Parameter aber unter dem (wohl auch korrekten) Bezeichner "primitiveType" verwendet werden.

Abgesehen vom Code-Beispiel enthält auch das Listing der Concrete Syntax Fehler: Statt des Typs "function" wird darin ein Typ namens "query" definiert, über den sich in der Spezifikation aber nichts weiter finden lässt, und der wohl eine frühere Bezeichnung für die jetzigen Functions war.

Zudem hält sich die Spezifikation nicht an die Definition der in ihr vorkommenden "LetExpCS". Dieser Typ ist von OCL übernommen, und dort folgendermaßen spezifiziert:

```
LetExpCS ::= 'let' VariableDeclarationCS LetExpSubCS
```

Das Keyword "let" kommt in der im Beispiel verwendeten Syntax jedoch nicht vor; stattdessen geschehen Zuweisungen so (Zeile 46 im obigen Code-Beispiel):

```
cn = an;
```

Schließlich sei auf zwei minimale Flüchtikgeitsfehler in den Definitionen der "keyDecl" und "varDecl" hingewiesen: Das Komma, welches zur Trennung der einzelnen Key Properties, beziehungsweise Identifier dient, ist Teil der konkreten Sprachsyntax und müsste demnach in einfache Anführungsstriche gefasst sein.

2.2 Xtext

Xtext ist ein kostenfreies, quelloffenes und als Eclipse Plug-In realisiertes Sprach-Entwicklungs-Framework der Eclipse Foundation [ecl13a]. Mit Xtext lassen sich allgemeine Programmier-, und insbesondere domänenspezifische Sprachen modellieren. Dabei muss sich der Anwender lediglich um die Spezifikation der gewünschten Sprachsyntax kümmern; mit wenigen Klicks generiert Xtext daraus dann einen passenden Compiler für die definierte Sprache, sowie auch einen Code Editor.

Dieser Editor unterstützt alle grundlegenden Funktionen, die man typischerweise von einem solchen erwarten würde: Die syntaktische Validierung des Quelltexts (Anzeige von Fehlermeldungen; auch bereits während der Eingabe), eine – ebenso on-the-fly arbeitende – Unterstützung beim Eintippen des Quelltexts (Code Assist/Code Completion), eine grafische Hervorhebung von Teilen des Quelltexts wie beispielsweise Schlüsselwörtern oder Literalen (Syntax Highlighting) sowie auch einer Funktion zur sauberen Formatierung des Geschriebenen.

In den meisten Fällen erfüllt die automatisch generierte Implementierung dieser Features aber dennoch nicht alle Anforderungen des Benutzers. Für diese Fälle bietet Xtext Schnittstellen in Form von mitgenerierten Java-Klassen, in denen der Programmierer durch Overriding der Default-Implementierung Anpassungen an der internen Arbeitsweise und der grafischen Darstellung des Editors vornehmen kann.

Xtext implementiert den generierten Code selbst wiederum als Eclipse Plug-In Projekt, welches in die Zielplattform eingebunden werden kann, und dort dann automatisch registriert, wenn eine Quellcode-Datei der spezifizierten Sprache erstellt wird – woraufhin es dem User den im Plug-In enthaltenen Code Editor anbietet.

2.2.1 Beispiel

Die Syntax der Xtext Meta-Sprache wird im Folgenden anhand eines Beispiels aufgezeigt. Dieses Beispiel basiert auf einem Beispiel aus der offiziellen Xtext Dokumentation [Ecl12], und ist der Einfachheit halber leicht gekürzt. Es stellt eine primitive domänenspezifische Sprache zur Modellierung von nicht weiter definierten Entitäten und ihren Attributen dar:

```
1  grammar org.example.domainmodel.DomainModel
2     with org.eclipse.xtext.common.Terminals
3
4  generate domainModel
5           "http://www.example.org/domainmodel/DomainModel"
6
7  Dominmodel:
8     (elements+=AbstractElement)*;
9
10 PackageDeclaration:
11    'package' name=QualifiedName
12    '{' (elements+=AbstractElement)* '}';
```

```
13
14 AbstractElement:
15    PackageDeclaration | Type;
16
17 QualifiedName:
18    ID ('.' ID)*;
19
20 Type:
21    DataType | Entity;
22
23 DataType:
24    'datatype' name=ID;
25
26 Entity:
27    'entity' name=ID
28    ('extends' superType=[Entity|QualifiedName])?
29    '{' (features+=Feature)* '}';
30
31 Feature:
32    name=ID ':' type=[Type|QualifiedName];
```

Dieser Code wird in die beim Anlegen eines Xtext Projekts erzeugte Datei mit der Endung "xtext" geschrieben.

Zunächst enthält diese Datei eine "grammar" und "generate" Anweisung, welche anhand des vom User spezifizierten Namens für die Sprache (hier: "org.example.domainmodel.DomainModel") automatisch erzeugt wird und nicht Teil der eigentlichen Sprache ist.

Danach beginnt die Definition der Sprache selbst, die durch eine Menge an "Rules" definiert ist. Die als erstes definierte Rule stellt die Wurzel der Sprache dar, von der ausgehend die gesamte Syntax und Semantik modelliert ist. In diesem Beispiel besteht die Sprache aus einer Menge an Elementen der Rule "AbstractElement", zur Laufzeit gespeichert in einer Variable namens "elements". Solch eine Variable bezeichnet man in Xtext auch als "Feature" einer Rule.

Dass es sich bei dem Feature "elements" um eine Menge, und nicht nur ein einzelnes Element handelt, wird durch das "+=" dargestellt (Ein einfaches Element wäre durch eine Zuweisung mittels "=" definiert). Der "*" gibt an, dass diese Menge zwischen einschließlich null und unendlich vielen Elementen enthalten kann.

Ein AbstractElement ist ein abstrakter Supertyp, und konkret entweder eine "PackageDeclaration" oder ein "Type", dargestellt durch den senkrechten Strich (OR).

Eine PackageDeclaration beginnt mit einem Keyword "package", gefolgt von einem "QualifiedName", welcher wiederum aus einem oder mehrerer, durch Komma getrennter "ID"s besteht. ID ist eine vordefinierte Xtext-"Terminal Rule" (auch genannt: "Token Rule" oder "Lexer Rule") und grundsätzlich in jeder Sprache durch die grammar-Answeisung importiert. ID ist in der importierten Datei wie folgt implementiert:

```
terminal ID : '^'?('a'..'z'|'A'..'Z'|'_')
              ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Bei einer ID handelt es sich also um einen Bezeichner, der vom Programmierer weitestgehend frei gewählt werden kann.

Terminal Rules sind grundsätzlich für das Parsen von Input zuständig. Neben der Nutzung der vordefinierten Terminal Rules lassen sich auch eigene spezifizieren. Die Syntax dafür entspricht der, wie sie eben für die ID gezeigt wurde.

Der andere konkrete Typ, den die Rule AbstractElement einnehmen kann, ist die Rule Type, welche wiederum in "DataType" und "Entity" unterschieden wird. Ein DataType ist definiert über das entsprechende Keyword, sowie eine ID, welche im Feature "name" gespeichert wird. Wie schon die Rule ID ist auch das Feature "name" in Xtext vordefiniert. Es hat insofern eine besondere Bedeutung, als dass es defaultmäßig für die Auflösung einer Referenz im späteren Programm herangezogen wird. Die Modellierung einer Referenz wird in der Rule "Entity" ersichtlich:

```
superType=[Entity|QualifiedName]
```

Die eckigen Klammern geben an, dass an dieser Stelle nicht die Definition einer neuen Entity oder eines QualifiedName erwartet wird, sondern eine Referenz auf eine bereits bestehenden Instanz. Der Anwender, der später in dieser Sprache programmiert, ist hier dann dazu aufgefordert, den Namen einer bereits definierten Instanz zu übergeben. Damit der von Xtext generierte Compiler diese Referenzierung korrekt auflöst, sollte vorher der vom Programmierer gewählte Name für die übergebene Instanz in dem Feature "name" abgespeichert sein. Zwar kann alternativ auch ein anderes, eigenes Feature für den Namen gewählt werden, allerdings muss in diesem Fall dann die Definition der Rule komplexer gestaltet werden, um eine Referenzierung möglich zu machen. Da dies weder im vorliegenden Beispiel noch in der später folgenden Implementierung von QVT-R nötig ist, wird darauf hier nicht genauer eingegangen.

Die Angabe einer soeben erwähnten Referenz für eine Entity ist hier nur optional, was durch das "?" modelliert ist.

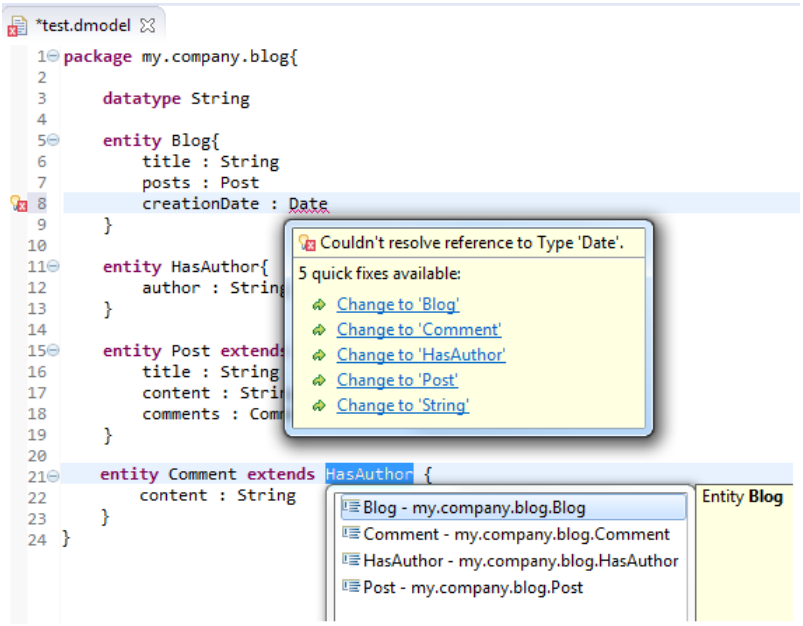

Abbildung 4: Beispiel eines von Xtext generierten Code Editors

Abbildung 4 zeigt einen Zusammenschnitt zweier verschiedener Zeitpunkte des Arbeitens mit dem Code Editor, wie Xtext ihn ohne weitere manuelle Anpassungen aus dem oben gelisteten Beispiel-Code generiert.

Wie zu sehen ist, werden Keywords optisch hervorgehoben. Syntax- und Semantik-Fehler werden in Echtzeit erkannt und unterringelt; ein Popup zeigt dabei eine genauere Fehlermeldung und bietet Vorschläge zur Beseitigung des Fehlers an.

Unten im Bild zu sehen ist das Content Assist Popup zur Code Completion.

Die Generierung des fertigen Plug-In aus der Sprach-Spezifikation geschieht durch einen Rechtsklick auf die xtext-Datei, gefolgt von “Run As -> Generate Xtext Artifacts”.

Abbildung 5 zeigt die daraus resultierende Projekt-Struktur, welche wie schon erwähnt ein Eclipse Plug-In Projekt darstellt.

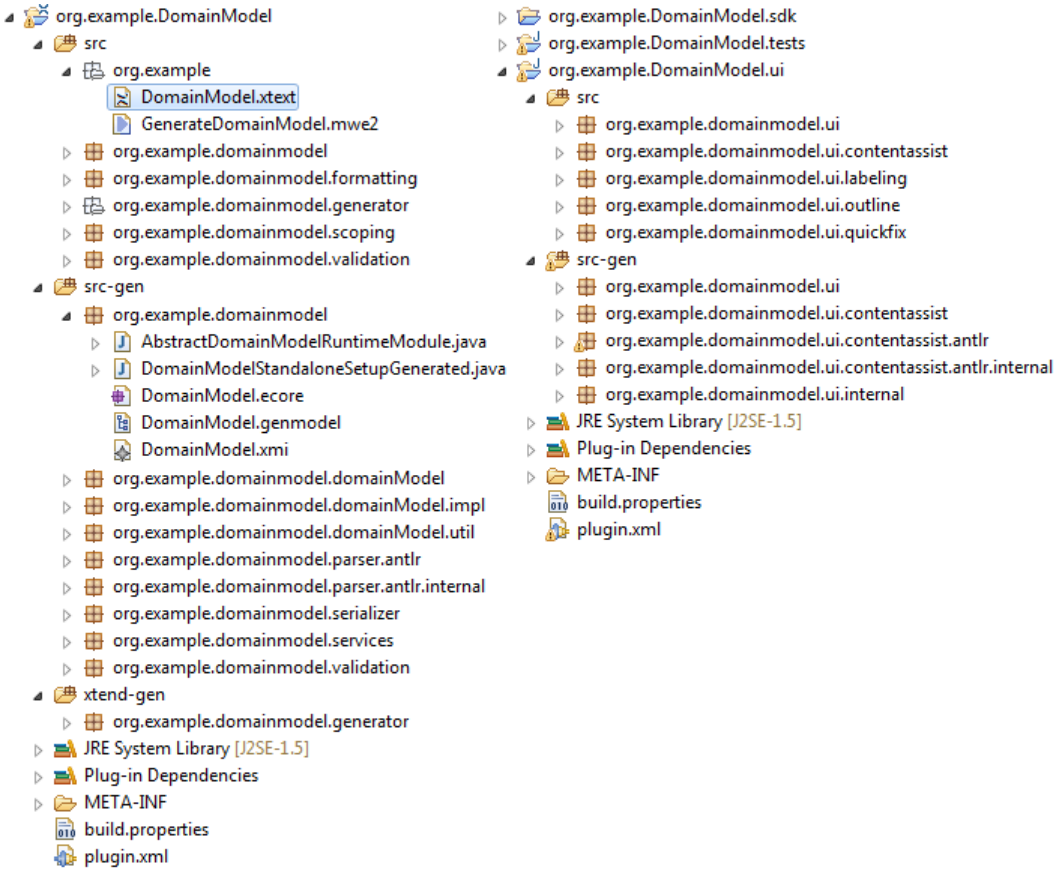

Abbildung 5: Xtext Projekt-Struktur nach der Code-Generierung

Das Haupt-Projekt enthält die Sprach-Spezifikation und die internen Mechanismen für den daraus generierten Compiler. Darin sei insbesondere auf das Package "org.example.domainmodel" aus dem Ordner "src-gen" hingewiesen: Dieses enthält das EMF Meta-Modell, in Form dessen Xtext die gesamte Sprach-Architektur realisiert. Die Generierung der Sprache erfolgt also durch eine Transformation der spezifizierten Rules und deren Features in ein Meta-Modell mit entsprechenden Meta-Klassen und dazugehörigen Attributen und Assoziationen.

Das .ui-Projekt enthält den generierten Code Editor. Die Projekte .sdk und .tests sind defaultmäßig leer und enthalten lediglich die grundlegende Konfiguration zum Zugriff auf die anderen Projekte.

Sofern das Plug-In in eine Eclipse-Instanz eingebunden ist, erscheint beim Anlegen einer Datei mit der beim Erstellen des Xtext-Projekts angegebenen Endung (im Beispiel: "dmodel") ein Dialog, der dem Anwender die Möglichkeit bietet, den generierten Code Editor für diese Sprache zu öffnen.

3 Implementierung

In diesem Kapitel werden die konkrete Implementierung der QVT-R Sprach-Spezifikation in Xtext, sowie die Anpassungen des von Xtext generierten Java-Codes für den Code Editor behandelt.

3.1 Sprach-Spezifikation

Die Modellierung der QVT-R Sprach-Spezifikation in der Datei QVT.xtext beginnt zunächst mit einer entsprechenden "grammar"- und "generate"-Klausel:

```
grammar de.lmu.ifi.pst.xtext.qvt.QVT
   with org.eclipse.xtext.common.Terminals
generate qVT "http://www.pst.ifi.lmu.de/"
```

Nachfolgend sind nun jeweils paarweise die einzelnen Bestandteile der QVT-R Concrete Syntax (oberhalb des Trennstrichs im eingerahmten Code) mit deren Implementierung in Xtext (unterer Teil) angegeben. Dabei werden die unter <OclExpressionCS> gelisteten Typen (ausgenommen <template>) und alle restlichen Typen getrennt voneinander betrachtet; beginnend mit Letzteren:

3.1.1 QVT-R – ausgenommen <OclExpressionCS>

```
<topLevel> ::= ('import' <unit> ';')* <transformation>*
-------------------------------------------------------------

TopLevel:
   (transformations+=Transformation)*;
```

Das Import-Feature ist nicht Teil der Anforderungen für diese Implementierung und wird deshalb in dieser weggelassen.

```
<transformation> ::= 'transformation' <identifier>
   '(' <modelDecl> (',' <modelDecl>)* ')'
   ['extends' <identifier>]
   '{' <keyDecl>* ( <relation> | <query> )*
-------------------------------------------------------------

Transformation:
   'transformation' name=ID
   '('
   modelDeclarations+=ModelDeclaration
   (','modelDeclarations+=ModelDeclaration)*
   ')'
   ('extends' superTransformation=[Transformation])?
   '{'
   (keyDeclarations+=KeyDeclaration)*
   (relations+=Relation | functions+=Function)*
   '}';
```

Der Typ <identifier> aus der Concrete Syntax entspricht einem beliebig wählbaren Bezeichner, der zur Identifikation einer entsprechenden Typ-Instanz verwendet wird. In Xtext existieren hierzu die vordefinierte Terminal Rule "ID" sowie das Feature "name".

Auch wenn dies nicht direkt aus der Concrete Syntax hervorgeht, so soll es sich bei dem nach dem Keyword "extends" spezifizierten <identifier> um eine Referenz auf eine andere Transformation handeln, was in der Xtext Implementierung entsprechend umgesetzt ist.

```
<modelDecl> ::= <modelId> ':' ( <metaModelId |
   '{' <metaModelId> (',' <metaModelId>* '}' )
-----------------------------------------------------------------

ModelDeclaration:
   name=ID ':' metaModelIds+=MetaModelId |
   ('{' metaModelIds+=MetaModelId
      (',' metaModelIds+=MetaModelId)*
   '}');
```

Der Typ <modelId> aus der Concrete Syntax ist nichts anderes als ein Wrapper für den Typ <identifier>:

```
<modelId> ::= <identifier>
```

Auf die Definition einer entsprechenden Xtext Rule wird daher verzichtet; statt dessen wird direkt das "name" Feature verwendet.

Die Definition des Typs <metaModelId> gleicht der von <modelId>. Für diesen Typ wird allerdings eine eigene Xtext Rule spezifiziert, da damit eine MetaModelId im späteren Code als eigener Datentyp existiert und somit explizit aufgelöst werden kann. Dies ist für die Code Validation nötig (siehe Abschnitt 3.2):

```
<metaModelId> ::= <identifier>
-----------------------------------------------------------------

MetaModelId:
   name=ID;
```

```
<keyDecl> ::= 'key' <classId>
   '{' <keyProperty (, <keyProperty>* '}' ';'
-----------------------------------------------------------------

KeyDeclaration:
   'key' classId=ClassId
   '{'
   keyProperties+=KeyProperty
   (',' keyProperties+=KeyProperty)*
   '}'
   ';';
```

```
<classId> ::= <pathNameCS>
-------------------------------------------------------------

ClassId: name=ID;
```

<pathNameCS> ist ein aus OCL übernommener Typ und dort im Endeffekt als String definiert. Zwar ist die Definition nicht deckungsgleich mit der von der Xtext Terminal Rule ID, was aber für diese Implementierung keine Rolle spielt. So wie für den Typ <metaModelId> wird auch für den Typ <classId> ein eigene Rule zur Unterscheidung dieses Typs von anderen Typen im generierten Java-Code benötigt.

```
<keyProperty> ::= <identifier> |
   'opposite' '(' <classId> '.' <identifier ')'
-------------------------------------------------------------

KeyProperty:
   PrimitiveKeyProperty | OppositeKeyProperty;

PrimitiveKeyProperty: name=ID;

OppositeKeyProperty:
   'opposite' '(' classId=ClassId '.' name=ID ')';
```

Zur Unterscheidung, ob es sich bei einer KeyProperty um eine direkte Property der von der KeyDeclaration implizierten Klasse, oder um eine von einer gegenseitigen Klasse referenzierte Property handelt, wird dieser Typ auf zwei verschiedene Rules ausgelagert, wobei die Rule "KeyProperty" einen abstrakten Supertyp dieser beiden Typen darstellt.

```
<relation> ::= ['top'] 'relation' <identifier>
   ['overrides' <identifier>]
   '{'
   <varDeclaration>*
   (<domain> | <primitiveTypeDomain>)+
   [<when>] [<where>]
   '}'
-------------------------------------------------------------

Relation:
   ('top')? 'relation' name=ID
   ('overrides' superRelation=[Relation])?
   '{'
   (variableDeclarations+=VariableDeclaration)*
   (domains+=Domain)+
   (whenStatement=WhenStatement)?
   (whereStatement=WhereStatement)?
   '}';
```

Ähnlich wie bei den Transformationen handelt es sich bei dem <identifier>, welcher dem Keyword “overrides” folgt, um eine Referenz auf eine andere Relation. Die Typen <domain> und <primitiveTypeDomain> sind über eine gemeinsame Rule “Domain” modelliert (siehe weiter unten).

```
<varDeclaration> ::= <identifier> (, <identifier>)*
   ':' <TypeCS> ';'
-------------------------------------------------------------

VariableDeclaration:
   name+=VariableName (',' name+=VariableName)*
   ':' type=TypeCS ';';

VariableName:
   name=ID;
```

Die Auslagerung der Menge an <identifier> in die eigens definierte Rule ”VariableName” gewährleistet, dass die einzelnen Namen in der Sprache referenziert werden können, das heißt dass die Rule VariableName in der Xtext Modellierung in eckige Klammern gefasst werden kann. Rein syntaktisch wäre dies zwar auch für eine VariableDeclaration mit folgender Modellierung möglich:

```
VariableDeclaration:
   name+=ID (',' name+=ID)*
   ':' type=TypeCS ';';
```

Allerdings funktioniert die defaultmäßige Referenzierung einer Instanz über das Feature “name” nur dann korrekt, wenn es sich dabei um eine einfache Variable handelt. Hier würde “name” allerdings für eine Liste an Elementen (“*”) verwendet. In so einem Fall würde immer nur ein einziges Element der Liste referenziert werden können. Die obige Implementierung mit Auslagerung der einzelnen Namen in einen separaten Typ VariableName sorgt dafür, dass im späteren Programm-Code nicht nur ein, sondern alle Elemente der Menge korrekt referenziert werden können.

Der Typ <TypeCS> ist ein aus OCL übernommener Typ. Seine Definition ist komplexer als für den hier vorliegenden Anwendungsfall nötig; er ist deshalb auf folgende Definition vereinfacht:

```
TypeCS:
   name=ID;
```

```
<domain> ::= [<checkEnforceQualifier>]
   'domain' <modelId> <template>
   ['implementedby' <OperationCallExpCS>]
   ['default_values' '{' (assignmentExp)+ '}']
   ';'

<primitiveTypeDomain> ::= 'primitive' 'domain'
   <identifier> ':' <TypeCS> ';'
-----------------------------------------------------------------

Domain:
   PrimitiveTypeDomain | ComplexDomain;

PrimitiveTypeDomain:
   'primitive' 'domain' name=VariableName
   ':' type=TypeCS ';';

ComplexDomain:
   ('checkonly' | 'enforce')? 'domain'
   modelId=[ModelDeclaration]
   template=Template
   ('default_values'
    '{' (assignmentExpressions+=AssignmentExpression)+ '}'
   )?
   ';';
```

Die Typen <domain> und <primitiveTypeDomain> sind zunächst über eine einzige Rule "Domain" referenziert, die wiederum an die konkreten Rules "ComplexDomain" und "PrimitiveTypeDomain" weiterleitet.

Der <identifier> einer <primitiveTypeDomain> ist nicht über die Xtext Terminal Rule ID, sondern die Rule VariableName definiert. Für die Sprach-Semantik macht das insofern einen Unterschied, als dass an gewissen Stellen nur ein Wert vom Typ VariableName, nicht aber grundsätzlich jede ID referenzierbar sein soll.

Der <checkEnforceQualifier>, welcher in der Concrete Syntax wie folgt definiert ist:

```
<checkEnforceQualifier> ::= 'checkonly' | 'enforce'
```

ist direkt in die Definition der Rule ComplexDomain übernommen; eine eigene Rule für diese Keywords wird für die Implementierung nicht benötigt.

<modelId> stellt eine Referenz auf den Namen einer ModelDeclaration dar. Die "implementedby" Klausel ist nicht Teil der Anforderungen für diese Implementierung und somit weggelassen.

```
<template> ::= (<objectTemplate> | <collectionTemplate>)
   ['{' <OclExpressionCS> '}']

<objectTemplate> ::= [<identifier>] ':' <pathNameCS>
   '{' [<propertyTemplateList>] '}'

<collectionTemplate> ::= [<identifier>] ':'
   <CollectionTypeIdentifierCS> '(' <TypeCS> ')'
   '{' [<memberSelection>] '}'
-------------------------------------------------------------

Template:
   (name=VariableName)? ':' classId=ClassId
   '{'
   (propertyTemplateList=PropertyTemplateList)?
   '}'
   ('{' oclExpression=OclExpressionCS '}')?;
```

Auf eine Unterscheidung zwischen <objectTemplate> und <collectionTemplate> wird in der Implementierung verzichtet, da <collectionTemplate>, und die darin enthaltenen Typen <CollectionTypeIdentifierCS> und <memberSelection> darin nicht benötigt werden. Die Rule "Template" entspricht demnach dem Typ <template> unter der Annahme, dass dieser Typ wie folgt definiert ist:

```
<template> ::= <objectTemplate>
   ['{' <OclExpressionCS> '}']
```

Der im Typ <objectTemplate> definierte <identifier> ist in der Implementierung als Typ der Rule VariableName definiert, da eine Template im Programm später referenziert und in diesem Zuge von allgemeinen IDs unterschieden werden soll. Der Typ <pathNameCS> ist als Typ der Rule ClassId definiert (siehe Modellierung von <classId>).

```
<propertyTemplateList> ::= <propertyTemplate>
   (',' <propertyTemplate>)*
-------------------------------------------------------------

PropertyTemplateList:
   propertyTemplates+=PropertyTemplate
   (',' propertyTemplates+=PropertyTemplate)*;
```

```
<propertyTemplate> ::= <identifier> '=' <OclExpressionCS>
   | 'opposite' '(' <classId> '.' <identifier> ')'
     '=' <OclExpressionCS>
-----------------------------------------------------------

PropertyTemplate:
   PrimitivePropertyTemplate | OppositePropertyTemplate;

PrimitivePropertyTemplate:
   name=ID
   '=' oclExpression=OclExpressionCS;

OppositePropertyTemplate:
   'opposite'
   '(' classId=ClassId '.' name=ID ')'
   '=' oclExpression=OclExpressionCS;
```

Die Aufteilung des Typs <propertyTemplate> in zwei verschiedene Rules wurde für die leichtere Unterscheidung zwischen direkten Referenzierungen (PrimitivePropertyTemplate) und Referenzierung über eine gegenseitige Klasse (OppositePropertyTemplate) vorgenommen.

```
<assignmentExp> ::= <identifier> '=' <OclExpressionCS> ';'
-----------------------------------------------------------

AssignmentExpression:
   name=ID
   '=' oclExpression=OclExpressionCS ';';
```

```
<when> ::= 'when' '{' (<OclExpressionCS> ';')* '}'
-----------------------------------------------------------

WhenStatement:
   {WhenStatement}
   'when'
   '{'
   (oclExpressions+=OclExpressionCS ';')*
   '}';
```

```
<where> ::= 'where' '{' (<OclExpressionCS> ';')* '}'
-----------------------------------------------------------

WhereStatement:
   {WhereStatement}
   'where'
   '{'
   (oclExpressions+=OclExpressionCS ';')*
   '}';
```

“{WhenStatement}” und “{WhereStatement}” stellen eine spezielle Semantik für Xtext dar und bedeuten, dass bei Ausführung der jeweiligen Rule zur Laufzeit auf jeden Fall eine Instanz der Rule erzeugt werden soll.

Grundsätzlich ist dies nur dann der Fall, wenn innerhalb der Rule eine Zuweisung eines Wertes zu einem Feature geschieht. Da hier jedoch die einzige(n) Zuweisung(en):

```
(oclExpressions+=OclExpressionCS ';')*
```

nur optional sind (“*”), wäre solch eine Instanz-Erzeugung bei folgendem, durchaus gültigen Programm-Code:

```
where{} // oder: when{}
```

nicht gegeben. Xtext gibt bei der Definition einer solch gearteten Rule eine Warnung aus, und empfiehlt das Erzwingen einer Instanz-Erzeugung mittels des hier verwendeten Ausdrucks “{<Rule>}”.

```
<query> ::= 'query' <identifier>
   '(' [<paramDeclaration> (',' <paramDeclaration>)*] ')'
   ':' <TypeCS>
   (';' | '{' <OclExpressionCS> '}')
-------------------------------------------------------------

Function:
   'function' name=ID
   '('
   (parameters+=ParameterDeclaration
   (',' parameters+=ParameterDeclaration)*)?
   ')'
   ':' type=TypeCS
   (';' | '{' oclExpression=OclExpressionCS '}');
```

Der Typ <query> wird unter dem Namen “Function” implementiert, was der eigentlich korrekte Bezeichner für diesen Typ ist.

```
<paramDeclaration> ::= <identifier> ':' <TypeCS>
-------------------------------------------------------------

ParameterDeclaration:
   name=VariableName ':' type=TypeCS;
```

Sowie auch schon die Typen Template, PrimitiveTypeDomain und VariableDeclaration soll eine Instanz des Typs ParameterDeclaration eine im Programm referenzierbare Variable darstellen, weshalb ihr “name”-Feature über eine solche definiert ist.

3.1.2 <OclExpressionCS>

Die unter dem Supertyp <OclExpressionCS> definierten Typen sind, ausgenommen <template>, von OCL übernommen, und hier wie folgt implementiert:

```
<PropertyCallExpCS>
-----------------------------------------------------------------

CallExpCS:
   PropertyCallExpCS | CodeCallExp;

PropertyCallExpCS:
   name=[VariableName]
   '.' properties+=Property ('.' properties+=Property)*;

Property:
   name=ID;

CodeCallExp:
   name=[CallableCode]
   '('
   (parameters+=CodeCallExpParameter)?
   (',' parameters+=CodeCallExpParameter)*
   ')';

CallableCode:
   Relation | Function;

CodeCallExpParameter:
   name=[VariableName];
```

Der OCL-Typ "PropertyCallExpCS" impliziert auch den OCL-Typen "OperationCallExpCS". Letzterer definiert entweder einen herkömmlichen Aufruf (hier: einer Relation oder einer Funktion), oder den Aufruf einer Infix-Operation (im Beispiel von QVT-R vorkommend in der Werte-Konkatenation mittels des "+"-Symbols). Um die verschiedenen Szenarien unterscheiden zu können, ist zunächst eine Rule mit dem Supertyp "CallExpCS" implementiert, welche sich in "PropertyCallExpCS" und "CodeCallExp" aufteilt.

Die Rule PropertyCallExpCS modelliert zunächst die Referenz auf eine Variable, gefolgt von einer an objekt-orientierte Sprachen erinnernde Punkt-Notation zum Zugriff auf eine Property der über die Variable referenzierte Instanz.

Eine CodeCallExp erwartet die Referenz auf eine Relation oder Funktion, und definiert eine Liste an Parametern, die eine Menge im Programm defininerter Variablen enthält.

Damit fehlt noch die Implementierung des "+"-Operationsaufrufs. Diese könnte grundsätzlich über folgende Erweiterung der CallExpCS geschehen:

```
CallExpCS:
   PropertyCallExpCS | CodeCallExp | ConcatenateCallExp;

ConcatenateCallExp:
   expressions+=OclExpressionCS
   ('+' expressions+=OclExpressionCS)*;
```

Im Kontext der Gesamtheit des Typs OclExpressionCS ist dies jedoch nicht möglich. Xtext meldet in diesem Fall einen Kompilierfehler, da die Rule OclExpressionCS damit einen linksrekursiven Objektgraphen definiert. Dies liegt an der Tatsache, dass die ConcatenateCallExp ein Subtyp von OclExpressionCS ist, selbst jedoch wiederum mit einer solchen beginnt. Der Xtext Parser kann in so einem Fall nicht mehr eindeutig bestimmen, in welcher Rule er sich gerade befindet.

Deshalb ist die Implementierung der ConcatenateCallExp über den Umweg einer Auslagerung aus dem Typen OclExpressionCS realisiert:

```
ConcatenatedOclExpression:
   oclExpressionTerms+=OclExpressionTerm
   ('+' oclExpressionTerms+=OclExpressionTerm)*;

OclExpressionTerm:
   '(' OclExpressionCS ')'
   | Template
   | IfExpCS
   | VariableExpCS
   | CallExpCS
   | LiteralExpCS;
```

```
<VariableExpCS>
-----------------------------------------------------------------

VariableExpCS:
   name=[VariableName];
```

Eine VariableExpCS repräsentiert eine Referenz auf eine Variable.

```
<LiteralExpCS>
-------------------------------------------------------------

LiteralExpCS:
   StringLiteralExp | IntLiteralExp | FloatLiteralExp;

StringLiteralExp:
   name=STRING;

IntLiteralExp:
   name=INT;

FloatLiteralExp:
   name=FLOAT;
```

Der Typ <LiteralExpCS>, wie er in OCL implementiert ist, definiert eine größere Menge verschiedener (primitiver als auch komplexer) Datentypen. Für den hier vorliegenden Anwendungsfall sind die Typen "STRING", "INT", und "FLOAT" ausreichend, weshalb die Implementierung sich auf diese beschränkt.

"STRING" und "INT" sind in Xtext vordefinierte Terminal Rules, welche direkt verwendet werden können. "FLOAT" ist grundsätzlich nicht verfügbar; diese Terminal Rule ist hier explizit definiert worden:

```
terminal FLOAT returns ecore::EFloat:
   ('0'..'9')+ '.' ('0'..'9')+;
```

Damit der im Eclipse Modeling Framework definierte Ecore Typ "EFloat", zu dem das eingegebene Literal zur Laufzeit geparst werden soll, von Xtext aufgelöst werden kann, ist ein entsprechendes Import-Statement im Header der Datei nötig:

```
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

```
<IfExpCS>
-------------------------------------------------------------

IfExpCS:
   'if' ifOclExpression=OclExpressionCS
   'then' thenOclExpression=OclExpressionCS
   'else' elseOclExpression=OclExpressionCS
   'endif';
```

```
<LetExpCS>
-------------------------------------------------------------

ConcatenatedOclExpression:
   oclExpressionTerms+=OclExpressionTerm
   ('+' oclExpressionTerms+=OclExpressionTerm)*;
```

Der Typ <LetExpCS> ist wie im Abschnitt "Fehler in der Spezifikation" geschildert fälschlicherweise gelistet. Aus diesem Grund wurde für die Implementierung von Variablen-Zuweisungen eine eigene Defintion aufgestellt, die zur tatsächlich in der Spezifikation verwendeten Syntax und Semantik passt.

Dabei stellt sich zunächst die selbe Problematik, wie sie im Falle der Implementierung des "+"-Operatoraufrufs vorlag: Auch hier ist <LetExpCS> ein Subtyp von <OclExpressionCS> und beginnt selbst wiederum mit der Definition einer solchen.

Die Realisierung der Semnatik von <LetExpCS> erfolgt daher, analog zu der von der ConcatenatedOclExpression, über eine Auslagerung:

```
OclExpressionCS:
   concatenatedOclExpressions+=ConcatenatedOclExpression
   (
   '=' concatenatedOclExpressions+=ConcatenatedOclExpression
   )?;
```

Zur besseren Übersicht wird auf den nächsten Seiten noch einmal der Implementierungs-Code gelistet, wie er in seiner Gesamtheit in der Datei niedergeschrieben ist.

```
1  grammar de.lmu.ifi.pst.xtext.qvt.QVT
2     with org.eclipse.xtext.common.Terminals
3  generate qVT "http://www.pst.ifi.lmu.de/"
4
5  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
6
7  TopLevel:
8     (transformations+=Transformation)*;
9
10 Transformation:
11    'transformation' name=ID
12    '('
13    modelDeclarations+=ModelDeclaration
14    (','modelDeclarations+=ModelDeclaration)*
15    ')'
16    ('extends' superTransformation=[Transformation])?
17    '{'
18    (keyDeclarations+=KeyDeclaration)*
19    (relations+=Relation | functions+=Function)*
20    '}';
21
22 ModelDeclaration:
23    name=ID ':' metaModelIds+=MetaModelId |
24    ('{' metaModelIds+=MetaModelId
25       (',' metaModelIds+=MetaModelId)*
26    '}');
27
28 MetaModelId:
29    name=ID;
30
31 KeyDeclaration:
32    'key' classId=ClassId
33    '{'
34    keyProperties+=KeyProperty
35    (',' keyProperties+=KeyProperty)*
36    '}'
37    ';';
38
39 ClassId:
40    name=ID;
41
42 KeyProperty:
43    PrimitiveKeyProperty | OppositeKeyProperty;
44
45 PrimitiveKeyProperty:
46    name=ID;
47
48 OppositeKeyProperty:
49    'opposite' '(' classId=ClassId '.' name=ID ')';
50
```

```
51  Relation:
52     ('top')? 'relation' name=ID
53     ('overrides' superRelation=[Relation])?
54     '{'
55     (variableDeclarations+=VariableDeclaration)*
56     (domains+=Domain)+
57     (whenStatement=WhenStatement)?
58     (whereStatement=WhereStatement)?
59     '}';
60
61  VariableDeclaration:
62     name+=VariableName (',' name+=VariableName)*
63     ':' type=TypeCS ';';
64
65  VariableName:
66     name=ID;
67
68  TypeCS:
69     name=ID;
70
71  Domain:
72     PrimitiveTypeDomain | ComplexDomain;
73
74  PrimitiveTypeDomain:
75     'primitive' 'domain' name=VariableName
76     ':' type=TypeCS ';';
77
78  ComplexDomain:
79     ('checkonly' | 'enforce')? 'domain'
80     modelId=[ModelDeclaration]
81     template=Template
82     ('default_values'
83      '{' (assignmentExpressions+=AssignmentExpression)+ '}'
84     )?
85     ';';
86
87  Template:
88     (name=VariableName)? ':' classId=ClassId
89     '{'
90     (propertyTemplateList=PropertyTemplateList)?
91     '}'
92     ('{' oclExpression=OclExpressionCS '}')?;
93
94  PropertyTemplateList:
95     propertyTemplates+=PropertyTemplate
96     (',' propertyTemplates+=PropertyTemplate)*;
97
98  PropertyTemplate:
99     PrimitivePropertyTemplate | OppositePropertyTemplate;
100
```

```
101 PrimitivePropertyTemplate:
102    name=ID
103    '=' oclExpression=OclExpressionCS;
104
105 OppositePropertyTemplate:
106    'opposite'
107    '(' classId=ClassId '.' name=ID ')'
108    '=' oclExpression=OclExpressionCS;
109
110 AssignmentExpression:
111    name=ID
112    '=' oclExpression=OclExpressionCS ';';
113
114 WhenStatement:
115    {WhenStatement}
116    'when'
117    '{'
118    (oclExpressions+=OclExpressionCS ';')*
119    '}';
120
121 WhereStatement:
122    {WhereStatement}
123    'where'
124    '{'
125    (oclExpressions+=OclExpressionCS ';')*
126    '}';
127
128 Function:
129    'function' name=ID
130    '('
131    (parameters+=ParameterDeclaration
132    (',' parameters+=ParameterDeclaration)*)?
133    ')'
134    ':' type=TypeCS
135    (';' | '{' oclExpression=OclExpressionCS '}');
136
137 ParameterDeclaration:
138    name=VariableName ':' type=TypeCS;
139
140 OclExpressionCS:
141    concatenatedOclExpressions+=ConcatenatedOclExpression
142    (
143    '=' concatenatedOclExpressions+=ConcatenatedOclExpression
144    )?;
145
146 ConcatenatedOclExpression:
147    oclExpressionTerms+=OclExpressionTerm
148    ('+' oclExpressionTerms+=OclExpressionTerm)*;
149
150
```

```
151 OclExpressionTerm:
152    '(' OclExpressionCS ')'
153    | Template
154    | IfExpCS
155    | VariableExpCS
156    | CallExpCS
157    | LiteralExpCS;
158
159 CallExpCS:
160    PropertyCallExpCS | CodeCallExp;
161
162 PropertyCallExpCS:
163    name=[VariableName]
164    '.' properties+=Property ('.' properties+=Property)*;
165
166 Property:
167    name=ID;
168
169 CodeCallExp:
170    name=[CallableCode]
171    '('
172    (parameters+=CodeCallExpParameter)?
173    (',' parameters+=CodeCallExpParameter)*
174    ')';
175
176 CallableCode:
177    Relation | Function;
178
179 CodeCallExpParameter:
180    name=[VariableName];
181
182 VariableExpCS:
183    name=[VariableName];
184
185 LiteralExpCS:
186    StringLiteralExp | IntLiteralExp | FloatLiteralExp;
187
188 StringLiteralExp:
189    name=STRING;
190
191 IntLiteralExp:
192    name=INT;
193
194 FloatLiteralExp:
195    name=FLOAT;
196
197 terminal FLOAT returns ecore::EFloat:
198    ('0'..'9')+ '.' ('0'..'9')+;
```

3.2 Code Validation

Zwar arbeiten Compiler und Code Editor, wie sie Xtext aus der obigen Sprach-Spezifikation generiert, auf Ebene der Syntax bereits korrekt. Semantisch allerdings fehlen noch einige Konfigurationen.

So kann der Anwender beispielsweise für jede in der Sprach-Spezifikation definierte ID grundsätzlich einen beliebigen Bezeichner angeben; etwa für die Meta-Modelle, die für die Transformation verwendet werden sollen:

```
MetaModelId: name=ID;
```

Demnach wäre also folgender Code gültig:

```
transformation someFantasyTransformation(a:foo, b:bar){...}
```

Und das unabhängig davon, ob die Bezeichner "foo" und "bar" tatsächlich in der Umgebung vorhandene Meta-Modelle referenzieren oder nicht.

Der selbe Fall liegt vor bei der Referenzierung von Meta-Modell-Klassen oder deren Attribute:

```
ClassId: name=ID;
PrimitivePropertyTemplate: name=ID;
```

Rein syntaktisch hindert den Programmierer nichts an der Definition von:

```
checkonly domain d c:SomeNonExistingClass {
   someNonExistingProperty = ...
    ...
}
```

Es fehlt folglich an einer semantischen Prüfung dieser Bezeichner. Vorgenommen werden muss die Prüfung insgesamt bei folgenden Typen:

```
MetaModelId
ClassId
PrimitivePropertyTemplate
PrimitiveKeyProperty
OppositePropertyTemplate
OppositeKeyProperty
TypeCS
```

Neben der ID-Prüfung fehlt es außerdem an einer semantischen Evaluierung eines Relation- oder Function-Aufrufs. Rein der Sprach-Spezifikation zufolge verlangt der Aufruf einer Relation oder Function nach einer beliebigen Menge an Aufruf-Parametern:

```
CodeCallExp: name=[CallableCode]
   '('
   (parameters+=CodeCallExpParameter)?
   (',' parameters+=CodeCallExpParameter)*
   ')';
```

Natürlich sollten aber die übergebenen Parameter für eine Relation mit deren Domain-Definitionen, beziehungsweise für eine Function mit deren Parameter-Signatur übereinstimmen.

Xtext bietet für solche Anpassungen eine Schnittstelle in Form einer im Zuge der Generierung erstellten Klasse, welche dem Namensmuster "<Sprache>JavaValidator" folgt; in diesem Fall nennt sich die Klasse demnach "QVTJavaValidator". Sie leitet sich von Klasse "AbstractQVTJavaValidator" ab, welche die Implementierung von mittels einer "@Check"-Annotation versehenen Methoden ermöglicht. Die Signatur dieser Methoden muss dabei folgendem Muster entsprechen:

```
@Check
public void checkX(X anyName)
```

<X> steht dabei für den Namen einer in der Sprach-Spezifikation definierten Rule. Zur Laufzeit wird die Methode für jede definierte Instanz dieses Typs ausgeführt. Über den Parameter hat der Anwender dann Zugriff auf die einzelnen Features der Instanz.

Der konkreten Implementierungslogik dieser Methoden liegen grundsätzlich keinerlei Einschränkungen vor. Jedoch finden sich in der Regel folgende Code-Zeilen am Ende des Rumpfes wieder:

```
if (...) {
   error("some message", XPackage.Literals.Y);
}
```

Die Methode "error" ist auch vom Typ AbstractQVTJavaValidator geerbt und wird verwendet um auszudrücken, dass ein Fehler im Code Editor mit der entsprechenden Meldung "some message" angezeigt werden soll. Der zweite Parameter referenziert eine bei der Generierung der Sprache angelegte Konstante <Y>, welche ein bestimmtes Feature einer bestimmten Rule repräsentiert. Diese Konstanten liegen im Typ <X>Package.Literals, wobei <X> den Namen der Sprache darstellt. Durch die Angabe solch einer Konstante wird bestimmt, wo genau der entsprechende Fehler im Code Editor angezeigt werden soll.

Die error-Methode ist überladen und existiert noch in einigen weiteren Varianten, mit denen sich noch detailliertere Fehlermeldungen spezifizieren lassen.

Zudem existieren einige "warning"-Methoden, die wie der Name vermuten lässt keine Fehler-, sondern lediglich Warnmeldungen im Code Editor produzieren.

Als ein konkretes Beispiel soll die in dieser Arbeit vorgenommene Implementierung der Check-Methode für die Rule TypeCS dienen. Sie enthält Aufrufe einiger weiterer, eigens definierter Methoden, die hier nicht weiter behandelt werden sollen. Die grundlegende Funktionsweise sollte dennoch ersichtlich sein:

```
@Check
public void checkTypeCS(TypeCS type) {
   /* prepare list of valid types */
   List<String> validTypesNames = new ArrayList<String>();
```

```
    /* add predefined (hard-coded) types */
    validTypesNames.addAll(QVTUtil.getPredefinedTypes());

    /* add types from meta models */
    Transformation transformation =
       QVTUtil.getUnderlyingTransformation(type);
    List<String> classesNames =
       QVTUtil.getTransformationClassesNames(transformation);
    validTypesNames.addAll(classesNames);

    /* check if given type is one of the valid types */
    if (!validTypesNames.contains(type.getName())) {
       error("No such type found",
            QVTPackage.Literals.TYPE_CS__NAME);
    }
}
```

3.3 Code Assist

Analog zur Schnittstelle für die Anpassung der internen Validierung bietet Xtext auch eine Schnittstelle zur Anpassung des vom Editor bereitgestellten Code-Assistenten. Der standardmäßig generierte Code-Assistent ist statisch an die Default-Implementierung des generierten Validators geknüpft. Das bedeutet, dass solche Anpassungen an die interne Validierung, wie sie im vorigen Abschnitt gezeigt wurden, nicht automatisch auf den generierten Code-Assistenten übertragen werden. Stattdessen müssen auch hier die entsprechenden Anpassungen vorgenommen werden.

Dafür stellt Xtext eine Klasse namens "XProposalProvider" innerhalb des .ui-Projekts bereit, welche sich von einer außerdem generierten Klasse "AbstractXProposalProvider" ableitet (<X> entspricht dabei dem Namen der Sprache). Letztere implementiert eine Menge an Methoden mit der Signatur nach dem Muster:

```
public void completeRule_Feature
   (EObject model, Assignment assignment,
    ContentAssistContext context,
    ICompletionProposalAcceptor acceptor)
```

Dabei entspricht <Rule> dem Namen einer Rule aus der Sprach-Spezifikation, und <Feature> dem Namen eines Features dieser Rule. Eine solche Methode wird für jedes Feature jeder Rule aus der Sprach-Spezifikation generiert.

Diese Methoden können nun in der Kind-Klasse überschrieben werden, um das jeweilige Verhalten des Content Assist und der Code Completion unter Verwendung der übergebenen Parameter anzupassen. Im Einzelnen repräsentieren die Parameter:

`EObject model`

– die Instanz der Rule, die im Editor zum Zeitpunkt der Ausführung des Code-Assists und damit dieser Methode definiert wird.

`Assignment assignment`

– Informationen darüber, welchem Feature der betroffenen Rule das im Editor definierte Token zugewiesen wird.

`ContentAssistContext context`

– den gesamten Definitions-Kontext, in dem sich der Code-Assist befindet.

`ICompletionProposalAcceptor acceptor`

– eine Schnittstelle für die Definition der dem Anwender vorzuschlagenden Completion Proposals.

Der Interface-Typ ICompletionProposalAcceptor ist in diesem Zusammenhang von besonderem Interesse. Er definiert eine Methode:

```
void accept(ICompletionProposal proposal);
```

über deren Aufruf ein Completion Proposal in das Content Assist Popup des Editors eingebunden wird. Für die Erzeugung eines solchen Proposals kann entweder ganz klassisch der Konstruktor einer entsprechenden Implementierung verwendet werden, oder die der einfacheren Benutzung halber bereitgestellte, von der Vaterklasse geerbte Hilfsmethode:

```
ICompletionProposal createCompletionProposal
   (String proposal, ContentAssistContext contentAssistContext)
```

So ist beispielsweise in der Implementierung dieser Arbeit folgende Methode für den Code Assist für den Typen TypeCS implementiert:

```
@Override
public void completeTypeCS_Name
   (EObject model, Assignment assignment,
    ContentAssistContext context,
    ICompletionProposalAcceptor acceptor) {
   /* prepare list for suggested types */
   List<String> validTypes = new ArrayList<String>();

   /* suggest predefined (hard-coded) types */
   validTypes.addAll(QVTUtil.getPredefinedTypes());

   /* suggest types from meta models */
   Transformation transformation =
      QVTUtil.getUnderlyingTransformation(model);
   List<String> classesNames =
      QVTUtil.getTransformationClassesNames(transformation);
   validTypes.addAll(classesNames);

   /* add proposal for each valid type */
   for (String type : validTypes) {
      acceptor.accept(createCompletionProposal(type, context));
   }
}
```

Alternativ können analog dazu auch Methoden nach dem Muster:

```
public void complete_Rule
   (EObject model, RuleCall ruleCall,
    ContentAssistContext context,
    ICompletionProposalAcceptor acceptor)
```

implementiert werden (<Rule> entspricht dabei dem Namen einer konkreten Rule). Sie bieten sich für die Fälle an, in denen der Code-Assist nicht für ein ganz bestimmtes Feature einer Rule, sondern für den Kontext einer Rule in ihrer Gesamtheit konfiguriert werden soll.

Mit den eben gezeigten Completion Proposals sind die Möglichkeiten zur Anpassung des Code-Assistenten noch nicht ausgeschöpft. Es lassen sich noch diverse weitere Dinge konfigurieren, wie beispielsweise Quick Fixes oder eine dem Code Editor beiliegende Outline View. Da all dies nicht zu den Anforderungen dieser Arbeit zählt, wird darauf nicht weiter eingegangen.

3.4 Scoping

Die Funktionsweise des Default-Scoping der von Xtext generierten Sprach-Implementierung ist nicht mit allen Teilen der vorliegenden Sprach-Spezifikation kompatibel. So führt beispielsweise folgender Zugriff auf die Variable "cl" – vorkommend im QVT-R Dokumentations-Beispiel – zu einem Fehler:

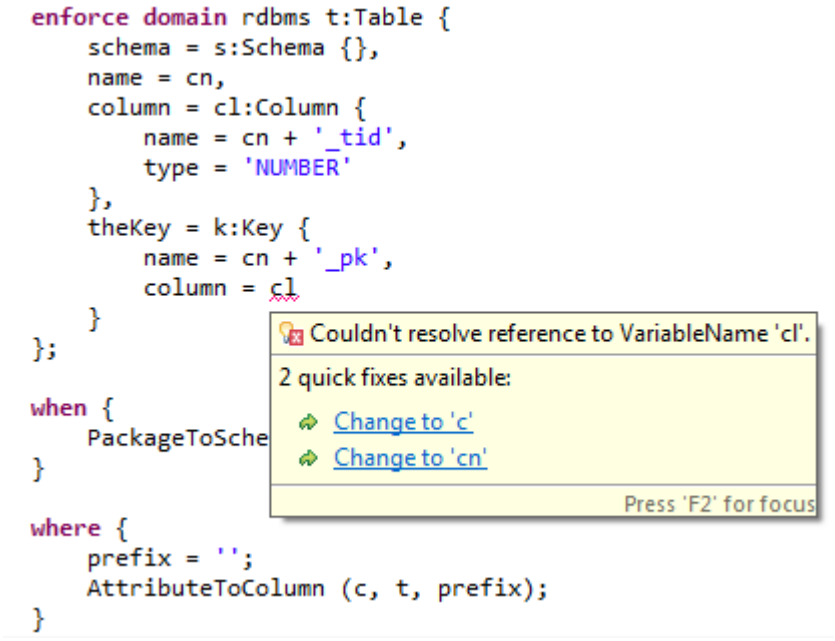

Abbildung 6: Problem beim Referenzieren von Variablen unter Verwendung des Default-Scoping

Ursache dieses Fehlers ist, dass die innerhalb einer Domain-Template definierten Variablen außerhalb der Template nur über ihren voll qualifizierten Namen aufgelöst werden können. Dieser würde für die Variable "cl" wie folgt lauten (Die im Ausschnitt gezeigte Domain befindet sich innerhalb einer Relation "ClassToTable", welche wiederum in der Transformation "umlToRdbms" definiert ist):

```
umlToRdbms.ClassToTable.column.cl
```

Dieser Bezeichner kann allerdings nicht angegeben werden, da in der Sprach-Spezifikation sowohl für das "name"-Feature als auch für die Referenzierung eines VariableName die Rule ID spezifiziert ist – welche das Symbol "." nicht enthalten kann.

Ein grundsätzlich wählbarer Ansatz zur Lösung des Problems bestünde darin, den Namen und die Referenzierung über eine andere Rule zu definieren, welche die Syntax eines voll qualifizierten Bezeichners erfüllt:

```
VariableName:
   name=FullyQualifiedName;

FullyQualifiedName:
   ID ('.' ID)*;
```

Da Referenzierungen in Xtext grundsätzlich über die Rule ID geschehen:

```
name=[Reference] // Kurzschreibweise für: name=[Reference|ID]
```

müssten diese zudem an den entsprechenden Stellen in der Sprach-Spezifikation angepasst werden – zum Beispiel in der im obigen Beispiel betroffenen Rule VariableExpCS:

```
VariableExpCS:
   name=[VariableName|FullyQualifiedName];
```

Diese Implementierung würde allerdings nicht mehr der offiziellen QVT-R-Syntax entsprechen, in der ein Zugriff auf die Variablen auch ohne voll qualifizierten Bezeichner möglich ist. Deshalb wurde hier ein anderer Ansatz gewählt; nämlich die Erweiterung des Scopes von Variablen auf die gesamte Relation oder Funktion, innerhalb der sie definiert sind. Damit ist die Angabe eines voll qualifizierten Bezeichners nicht mehr nötig.

Zur Anpassung der Funktionsweise des Scoping hat Xtext eine Klasse namens "QVTScopeProvider" generiert, in der sich Methoden mit folgender Signatur definieren lassen:

```
public IScope scope_Rule_feature(Rule var, EReference ref)
```

<Rule> muss dabei dem konkreten Namen einer in der Sprache spezifizierten Rule gleichen, und <feature> dem Bezeichner eines in dieser Rule definierten Feature. Der Parameter "var" enthält zur Laufzeit jeweils eine konkrete Instanz der angegebenen Rule, und der Parameter "ref" das Feature, über das die Referenzierung vorgenommen wird.

Zur Erzeugung einer passenden IScope-Instanz, die von der Methode zurückgeliefert werden muss, kann eine statische Hilfsmethode aus der Klasse "Scope" verwendet werden, die folgende Signatur hat:

```
public static IScope scopeFor
   (Iterable<? extends EObject> elements)
```

Sie nimmt eine beliebige Collection von Elementen des Typs EObject an und liefert einen Scope, der einen Zugriff auf alle Instanzen in der Collection erlaubt. Da EObject die Wurzelklasse aller EMF Meta-Klassen ist, kann diese Methode auf jede beliebige Instanz einer jeden Rule angewandt werden.

Im Folgenden wird zur besseren Veranschaulichung die konkrete Implementierung des QVTScopeProvider abgebildet. Die darin aufgerufene Methode:

```
QVTUtil.getNestedVariableNames
```

ist eine selbst erstellte Hilfsmethode, die ein beliebiges EObject-Element annimmt, es nach allen Vorkommnissen von VariableName-Definitionen untersucht, und diese dann in einer Liste zurückgibt.

```
public class QVTScopeProvider
   extends AbstractDeclarativeScopeProvider{

   public IScope scope_VariableExpCS_name
      (VariableExpCS var, EReference ref){
      List<VariableName> visibleVars =
         QVTUtil.getNestedVariableNames(QVTUtil
            .getUnderlyingRelationOrFunction(var));
      return Scopes.scopeFor(visibleVars);
   }

   public IScope scope_CodeCallExpParameter_name
      (CodeCallExpParameter param, EReference ref){
      List<VariableName> visibleVars =
         QVTUtil.getNestedVariableNames(QVTUtil
            .getUnderlyingRelationOrFunction(param));
      return Scopes.scopeFor(visibleVars);
   }
}
```

Die beiden Methoden erweitern den Scope von jeglichen Variablen demnach so, dass sie für alle Definitionen einer <VariableExpCS>, sowie für alle Definitionen eines <CodeCallExpParameter> innerhalb der jeweiligen Relation oder Funktion verfügbar sind.

So wie schon die Anpassungen am Validator nicht automatisch für den Code Assist übernommen wurden, mussten auch für die Änderungen am Scoping entsprechende Anpassungen im ProposalProvider vorgenommen werden, um die Variablen auch in den Popups anzuzeigen. Von der Abbildung der entsprechenden Implementierung wird an dieser Stelle abgesehen.

3.5 Code Formatter

Der defaultmäßig von Xtext generierte Code Formatter nennt sich "OneWhitespaceFormatter" und separiert jegliche Symbole jeweils durch ein Leerzeichen voneinander, wobei nach jeweils 80 Zeichen eine neue Zeile beginnt.

Dies ist für den hier vorliegenden Anwendungsfall nicht geeignet. Gewünscht ist eine Formatierung, die der aus den Code-Beispielen der QVT-R Dokumentation ähnelt. Das bedeutet: Separierung der einzelnen Transformationen, Relationen, Domains und Funktionen in voneinander getrennte Blöcke, sowie Einrückung der Elemente innerhalb dieser Blöcke und deren strukturierte Auflistung.

Zur Implementierung eines Formatters mit eigenen Formatierungsregeln generiert Xtext eine Klasse, die sich in diesem Fall "QVTFormatter" nennt. Sie leitet sich vom Typ "AbstractDeclarativeFormatter" ab und überschreibt eine Methode namens "configureFormatting". Diese ist standardmäßig als No-Op implementiert, bietet aber die Schnittstelle zur Implementierung von eigenem Formatierungs-Code.

Dazu werden das der Methode übergebene Objekt vom Typ "FormattingConfig", sowie der Kontext über die generierte Sprach-Grammatik verwendet. Letzterer kann über eine auch geerbte Methode mit der Bezeichnung "getGrammarAccess" erlangt werden.

Auf die vollständige Auflistung der verfügbaren Methoden sowie deren Dokumentation wird an dieser Stelle aus Gründen des Umfangs verzichtet. Stattdessen soll der auf der nächsten Seite folgende Auszug aus der konkreten Implementierung für das Verständnis über die Herangehensweise an die Definition sprachspezifischer Formatierungsregeln hinreichend sein.

Abbildung 7 zeigt einen Beispiel-Code, wie er nach Anwendung der Formatierung (Tastenkürzel: CTRL+Shift+F) unter Verwendung des vollständig implementierten Formatters für diese Arbeit vom Code Editor angezeigt wird.

```
public class QVTFormatter extends AbstractDeclarativeFormatter {
   @Override
   protected void configureFormatting(FormattingConfig c) {

      QVTGrammarAccess f = (QVTGrammarAccess) getGrammarAccess();

      // ...

      /* space formatting for ':', '(', ')' and '.' */
      List<Keyword> leftRoundBrackets = f.findKeywords("(");
      for (Keyword leftRoundBracket : leftRoundBrackets) {
         c.setNoSpace().after(leftRoundBracket);
      }
      List<Keyword> rightRoundBrackets = f.findKeywords(")");
      for (Keyword rightRoundBracket : rightRoundBrackets) {
         c.setNoSpace().before(rightRoundBracket);
      }
      List<Keyword> colons = f.findKeywords(":");
      for (Keyword colon : colons) {
         c.setNoSpace().before(colon);
         c.setNoSpace().after(colon);
      }
      List<Keyword> points = f.findKeywords(".");
      for (Keyword point : points) {
         c.setNoSpace().before(point);
         c.setNoSpace().after(point);
      }

      /* if-then-else formatting */
      c.setLinewrap().after(
         f.getIfExpCSAccess().
         getIfOclExpressionOclExpressionCSParserRuleCall_1_0());
      c.setIndentationIncrement().after(
         f.getIfExpCSAccess().
         getIfOclExpressionOclExpressionCSParserRuleCall_1_0());
      c.setLinewrap().after(
         f.getIfExpCSAccess().
         getThenOclExpressionOclExpressionCSParserRuleCall_3_0());
      c.setLinewrap().after(
         f.getIfExpCSAccess().
         getElseOclExpressionOclExpressionCSParserRuleCall_5_0());
      c.setLinewrap().before(
         f.getIfExpCSAccess().getEndifKeyword_6());
      c.setIndentationDecrement().before(
         f.getIfExpCSAccess().getEndifKeyword_6());

      // ...
   }
}
```

```
    checkonly domain uml a:Association {
        namespace = p:Package {},
        name = an,
        source = sc:Class {
            kind = 'Persistent',
            name = scn
        },
        destination = dc:Class {
            kind = 'Persistent',
            name = dcn
        }
    };

    enforce domain rdbms fk:ForeignKey {
        schema = s:Schema {},
        name = fkn,
        owner = srcTbl,
        column = fc:Column {
            name = fcn,
            type = 'NUMBER',
            owner = srcTbl
        },
        refersTo = pKey
    };

    when {
        PackageToSchema (p, s);
        ClassToTable (sc, srcTbl);
        ClassToTable (dc, destTbl);
        pKey = destTbl.theKey;
    }

    where {
        fkn = scn + '_' + an + '_' + dcn;
        fcn = fkn + '_tid';
    }
}

function PrimitiveTypeToSqlType (primitiveType:String):String {
    if (primitiveType = 'INTEGER')
        then 'NUMBER'
        else if (primitiveType = 'BOOLEAN')
            then 'BOOLEAN'
            else 'VARCHAR'
        endif
    endif
}
```

Abbildung 7: Formatierung des implementierten Formatters

4 Zusammenfassung und Ausblick

Dieses Kapitel gibt einen abschließenden Überblick über das in dieser Arbeit implementierte Tool sowie einen Ausblick darüber, welche Erweiterungen an dem Tool für die Zukunft denkbar sind.

4.1 Zusammenfassung

Ziel dieser Arbeit war es, einen quelloffenen und kostenlosen Code Editor für die Programmierung in der Transformationssprache QVT-R zu implementieren, der sich im Gegensatz zu den sonst dafür erhältlichen Tools mehr auf die Abbildung von Relationen zwischen den Modellen fokussiert, anstatt auf deren Transformation. Zudem sollte die dem Editor zugrunde liegende QVT-R Sprach-Spezifikation nicht auf den offiziellen Sprach-Standard beschränkt sein, sondern eine Anpassung der Sprach-Syntax und -Semantik erlauben – was in der hier vorgenommenen Implementierung bereits der Fall ist, da einige Teile der Sprache weggelassen oder vereinfacht wurden.

Zur Implementierung der Sprach-Spezifikation und gleichzeitig auch der Realisierung des Code Editors wurde das Sprach-Entwicklungs-Framework Xtext verwendet. Dabei wurde zunächst die gewünschte QVT-R Syntax in der Xtext Meta-Sprache modelliert. Der auf Basis dieser Modellierung von Xtext erzeugte Code Editor arbeitete auf Ebene der Syntax damit zwar bereits korrekt, allerdings mussten noch einige semantische Anpassungen vorgenommen werden, da es erstens an einer Prüfung der vom Programmierer im Code referenzierten Meta-Modelle und deren Klassen und Attribute, sowie der aufgerufenen Relationen und Funktionen fehlte, zweitens der von Xtext generierte Scoping-Mechanismus nicht alle Anforderungen der Sprach-Spezifikation erfüllte.

Vorgenommen wurden diese Anpassungen über Erweiterungen bestimmter Java-Klassen, die Xtext bei der Sprach-Kompilierung zu diesem Zweck generiert. Die Anpassungen wurden dabei jeweils auf zwei Ebenen vorgenommen: Zum Einen auf Ebene der internen Sprach-Validierung, zum Anderen auf Ebene der vom Editor zur Verfügung gestellten Code Assist Popups.

Neben den semantischen Anpassungen wurde schließlich noch ein eigener Code Formatter implementiert, sodass der geschriebene Code vom Editor auf Tastendruck hin in eine gut leserliche, den Code-Beispielen aus der QVT-R Spezifikation nahekommende Form gebracht wird.

4.2 Ausblick

Es sind einige zukünftige Erweiterungen an dem implementierten Tool denkbar, die im Folgenden kurz angesprochen werden sollen.

Zunächst wurde die QVT-R Concrete Syntax in der hier vorgenommenen Implementierung nicht vollständig abgebildet. So wird im QVT-R Sprach-Standard zwischen zwei Typen von Templates unterschieden: <objectTemplate> und <collectionTemplate>. Letzterer wurde in der Implementierung vernachlässigt. Eine Collection Template erlaubt in QVT-R die Definition und Verwendung

von Mengen an Instanzen in (auch sortierten) Listen, Sets und anderen Arten von Collections. Für diesbezügliche Erweiterungen der Möglichkeiten für den Anwender könnte die Implementierung der Templates also noch um Collection Templates vervollständigt werden.

Auch der in QVT-R vorkommende Import-Mechanismus wurde nicht implementiert. Da der Anwender des Tools dadurch darauf beschränkt ist, all seinen Code in eine einzige Datei zu schreiben, und auf die Aufteilung von Code in mehrere logisch zusammenhängende Dateien verzichten muss, wäre eine zukünftige Implementierung dieser Option im Hinblick auf die Modellierung größerer Projekte sinnvoll.

Ein weiteres für den Anwender nützliches Feature würde die Möglichkeit zur Filterung der zur Laufzeit verfügbar gestellten Meta-Modelle darstellen. Die jetzige Implementierung erlaubt bei der Definition der Meta-Modelle für eine Transformation jegliche in die Umgebung der laufenden Eclipse Instanz eingebundenen Meta-Modelle – und listet diese auch entsprechend im Code Assist Popup. Da dies je nach Eclipse-Konfiguration eine unüberschaubar große Menge an Modellen sein kann, und außerdem die in der Eclipse Plattform selbst definierten Meta-Modelle wohl kaum für eine Transformation verwendet werden, wäre es wünschenswert, eine Möglichkeit zu haben, um nur eine selbst definierte Menge an konkreten Meta-Modellen für die Transformation zu berücksichtigen. Dies könnte beispielsweise in Form einer vom Anwender zu spezifizierenden Datei realisiert werden, in der die Bezeichner der gewünschten Meta-Modelle aufzulisten sind, und auf welche sich Code Validator und Editor dann beschränken.

Weiterhin sei die fehlende Durchführung einer vollständigen Typ-Prüfung erwähnt. Zwar wird ein Kompatibilitäts-Check zwischen Meta-Modellen und den darin enthaltenen Meta-Klassen und Properties durchgeführt; allerdings nur im Bezug auf die bloße Existenz dieser Modelle/Klassen/Properties, und nicht auf deren Typ. So kann beispielsweise eine Zuweisung eines Wertes zu einer Property vorgenommen werden, deren Typ gar nicht dem Typen des zugewiesenen Wertes entspricht. Dies gilt neben Zuweisungen auch für den Aufruf von Relationen und Funktionen. In diesem Fall wird lediglich die Anzahl der übergebenen Parameter geprüft, nicht aber ob deren Typen mit den spezifizierten Typen der Relation-Domains oder der Funktions-Parameter übereinstimmen.

Um solch eine "Type Safety" zu implementieren müsste zunächst jedoch eine weitere Anpassung vorgenommen werden, nämlich die Auflösung von Namenskonflikten. Xtext erlaubt grundsätzlich die Definition mehrerer gleichnamiger Instanzen, wodurch es nicht nur für den Anwender ein Problem wird festzustellen, nach welcher Instanz beim Referenzieren von überladenen Namen aufgelöst wird, sondern auch für den soeben angesprochenen Type Check.

Insgesamt ist es zu empfehlen, in zukünftigen Updates zunächst die Auflösung von Namenskonflikten, und dann die Implementierung einer vollständigen Type Safety umzusetzen – denn dies sind die beiden Dinge, welche die größten Einschränkungen in der jetzigen Implementierung darstellen.

Literatur

[Ecl12] Eclipse Foundation: *Xtext Dokumentation, Version 2.3.* 2012. – http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf

[ecl13a] *Eclipse Foundation.* 2013. – http://www.eclipse.org/

[Ecl13b] Eclipse Foundation: *EMF: The Eclipse Modeling Framework.* 2013. – http://eclipse.org/modeling/emf/

[Ecl13c] Eclipse Foundation: *Model To Model Transformation (MMT).* 2013. – http://wiki.eclipse.org/M2M

[Ecl13d] Eclipse Foundation: *Xtext.* 2013. – http://www.eclipse.org/Xtext/

[ikv13] ikv++ technologies: *medini QVT.* 2013. – http://projects.ikv.de/qvt

[ITW13] ITWissen - Das grosse Online-Lexikon für Informationstechnologie: *Begriffsklärung: Modelltransformation.* 2013. – http://www.itwissen.info/definition/lexikon/Modelltransformation-model-transformation.html

[Obj11a] Object Management Group (OMG): *Query/View/Transformation (QVT).* 2011. – http://www.omg.org/spec/QVT/1.1/

[Obj11b] Object Management Group (OMG): *QVT-R Dokumentation, Version 1.1.* 2011. – http://www.omg.org/spec/QVT/1.1/PDF/

[Obj13a] Object Management Group (OMG): *Meta Object Facility (MOF).* 2013. – http://www.omg.org/mof/

[Obj13b] Object Management Group (OMG): *Object Constraint Language (OCL).* 2013. – http://www.omg.org/spec/OCL/

[Obj13c] Object Management Group (OMG): *OCL Dokumentation, Version 2.3.1.* 2013. – http://www.omg.org/spec/OCL/2.3.1

[Obj13d] Object Management Group (OMG): *Unified Modeling Language (UML).* 2013. – http://www.uml.org/

[Tat07] Tata Research Development and Design Centre: *ModelMorf.* 2007. – http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm