Freie Universität Berlin

Diplomarbeit

Entwicklung eines visuellen modellgetriebenen Transformationsframeworks für QVT

Bearbeitet von:

Max Bureck

Betreut von:

Prof. Dr. Elfriede Fehr

Dipl.-Inform. Mohammad Al Saad

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.
Berlin, den 02.06.2009

Max Bureck

Zusammenfassung

In dieser Diplomarbeit wird die Entwicklung eines Frameworks beschrieben, welches das visuelle Definieren von Modelltransformationen in der Sprache Query/View/Transformation Relations (QVT/R) zwischen EMF Metamodellen erlaubt. Die Sprache ist Teil des Standards Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.0, der vom Industriekonsortium Object Management Group (OMG) beschlossen wurde und Teil der OMG Initiative Model Driven Architecture ist, die Technologien und Vorgehen für die Modellgetriebene Softwareentwicklung standardisiert. Die QVT/R-Sprachpezifikation umfasst die abstrakte sowie die konkrete textuelle und konkrete visuelle Syntax. Im Rahmen der Arbeit wurde die konkrete visuelle Syntax implementiert und eine Synchronisation mit der textuellen Syntax umgesetzt. Die Implementierung erlaubt darüber hinaus das Verknüpfen von Elementen der graphischen Syntax mit Trace-Daten, welche von QVT/R-Engines generiert wurden. Die Entwicklung selbst wurde mit Modellgetriebenen Technologien durchgeführt. Die technische Basis bilden das Eclipse Framework und dessen Unterprojekte EMF und GMF. Für die Synchronisation zwischen Modell und Text kam die Medini QVT Engine und das openArchitectureWare Framework zum Einsatz.

Inhaltsverzeichnis

1 Einführung und Motivation....7

1.1 Model Driven Software Development....7

1.2 Architekturzentrierte Modellgetriebene Softwareentwicklung....8

1.3 Model Driven Architecture....9

1.4 Query/View/Transformation....10

1.4.1 Überblick über QVT....10

1.4.2 QVT Relations....11

1.5 Motivation....24

2 Problemstellung und Ziel der Arbeit....25

3 Verwandte Arbeiten....26

4 Konzepte für die Realisierung....27

4.1 Modellvalidierung....27

4.1.1 Strukturelle Aspekte des Diagramms....27

4.1.2 Valide Metamodell Auswahl....29

4.1.3 Textueller Vadidator....30

4.2 Metamodell Registries....31

4.3 Verknüpfung mit Trace-Daten....32

5 Realisierung des Visual QVT/R Frameworks....33

5.1 Verwendete Technologien....33

5.1.1 Eclipse Framework....33

5.1.2 openArchitectureWare Framework....35

5.1.3 Medini QVT....36

5.2 Vorüberlegungen zur Implementierung....36

5.2.1 QVT Relations Metamodell....36

5.2.2 Synchronisierung zwischen textueller und graphischer Syntax....39

5.2.3 Konfiguration des Visual QVT/R Frameworks....39

5.2.4 Trace Datenzugriff und Editor....40

5.2.5 Wahl der Validierungs-Technologien....40

5.3 Architektur des Visual QVT/R Frameworks....41

5.3.1 Grobstruktur....42

5.3.2 Komplexe Abläufe....44

5.4 Technische Umsetzung....46
5.4.1 Metamodelle....46
5.4.2 Kernfunktionalität des Frameworks....49
5.4.3 Die Modell-zu-Text-Transformation....50
5.4.4 Medini spezifische Funktionalität....52
5.4.5 Anbindung an die EMF Registry....58
5.4.6 Die Visual QVT/R Nature....58
5.4.7 Visual QVT/R Modellvalidierung....59
5.4.8 Der graphische QVT/R-Editor....59
6 Funktionen des Frameworks....61
6.1 Modellierung von Transformationen....61
6.2 Konfiguration des Visual QVT/R Frameworks....64
6.3 Synchronisation zwischen texuteller und graphischer Syntax....65
6.4 Verknüpfung mit Trace-Daten....67
6.5 Anbindung an die openArchitectureWare Workflow Engine....69
7 Evaluierung....70
7.1 Stabilität der Implementierung....70
7.2 Performance der Implementierung....70
7.3 Fallbeispiel....73
8 Schlusswort....77
8.1 Erfahrungen....77
8.1.1 Erfahrungen mit den verwendeten Technologien....77
8.1.2 Erfahrungen mit dem QVT Standard....78
8.2 Erweiterungsmöglichkeiten des Frameworks....78
8.2.1 Weiterführende Konzepte....78
8.2.2 Interoperabilität....79
8.3 Fazit....79
9 Danksagungen....82
10 Anhang....82
A VQVT/R-Metamodell....82
A.I Überblick....82
A.II Named Elements....84
A.III Collection Content....84
A.IV OCL Expressions....84
B Beispieltransformation....85

C Abkürzungsverzeichnis.....89
D Abbildungsverzeichnis.....90
E Literaturverzeichnis.....92

1 Einführung und Motivation

Zunächst werden grundlegende Begriffe erläutert, bevor auf die Motivation und im nächsten Kapitel auf die sich daraus entwickelten Ziele der Diplomarbeit eingegangen wird.

1.1 Model Driven Software Development

Die Modellgetriebene Softwareentwicklung (MDSD: Model Driven Software Development) hat zum Ziel den Abstraktionsgrad und Automatisierungsgrad der Softwareentwicklung zu erhöhen und für eine bessere Wiederverwendbarkeit von Fachwissen und Quelltext zu sorgen. Zudem soll die Qualität gesteigert werden, indem Qualitätsanforderungen bereits früh formal festgelegt und geprüft werden können [Pet06].

Dazu werden formale Modelle erstellt, die bestimmte Aspekte einer Domäne festhalten und fachliches Wissen kapseln. In der traditionellen Softwareentwicklung ist dieses Fachwissen im Quelltext festgehalten und somit unter technischen Details der Programmiersprache versteckt. Modelle können mittels Transformationen zu Software übersetzt oder von anderen Anwendungen zur Laufzeit interpretiert werden. Um Modelle zu Software zu übersetzen werden Modell-zu -Modell (M2M: Model-To-Model) Transformationen, manchmal auch Modelltransformationen genannt, und Modell-zu-Text (M2T: Model-To-Text) Transformationen eingesetzt. Damit kann der Übersetztungsprozess stufenweise vorgenommen werden. Üblicherweise sinkt der Abstraktionsgrad mit den Transformationen und die entstehenden Artefakte werden um detailliertere Informationen angereichert. Oft sind die generierten Anteile um manuell geschriebene Teile zu ergänzen. Es gibt mehrere Vorgehensweisen um dies zu erreichen. Ist die Zielpattform einer Transformation eine objektorientierte Programmiersprache, kann die Ergänzung z.B. durch Klassen-Vererbung gelöst werden, ohne dabei generierte Dateien bearbeiten zu müssen. Unterstützt die Zielsprache aspektoriente Programmierung, so können manuelle Änderungen über Aspekte „eingewoben“ werden (siehe auch [Mosc05]). In einigen Fällen ist es jedoch unumgänglich, generierte Artefakte manuell zu ergänzen. Modell-zu-Text-Programmiersprachen führen daher das Konzept der geschützten Bereiche (Protected Regions) ein, damit manuelle Ergänzungen bei Wiederholung von Generator-Durchläufen erhalten bleiben. Bei vielen M2M-Transformationssprachen gestaltet sich das Beibehalten von manuellen Änderungen bei Retransformation allerdings schwierig. Ist keine Unterstützung der M2M-Sprache gegeben, so muss eine zusätzliche Transformation geschrieben werden, welche die alte Version des Ausgabemodells als Eingabemodell mit einbezieht und beizubehaltende Werte ausließt.

Um eine Klasse von Modellen formal zu beschreiben, werden Metamodelle erstellt; diese legen unter anderem fest, welche Modellelemente zur Verfügung stehen, welche Beziehungen unter den Modellelementen erlaubt sind und welche Attribute Modellelemente besitzen. Ist ein Modell nach den Vorgaben eines Metamodells aufgebaut, so ist das Modell ein Exemplar bzw. eine Instanz des Metamodells. Können einige Bedingungen für ein gültiges Modell nicht durch ein Metamodell ausgedrückt werden, so muss eventuell eine zusätzliche Sprache für Beschränkungen (Constraints) genutzt werden um auszudrücken, was gültige Modellinstanzen sind. Der Prozess des Überprüfens auf Gültigkeit eines Modells wird Modellvalidierung genannt. Bei der Beschreibung des Aufbaus von Modellen wird zwischen abstrakter und konkreter Syntax unterschieden. Die abstrakte Syntax beschreibt dabei den logischen Aufbau der Modelle durch das Metamodell, während die konkrete Syntax beschreibt, wie Modelle notiert werden sollen. Modelle können in einer konkreten textuellen oder konkreten graphischen Syntax notiert werden. Wird eine Sprache entworfen, die genau auf eine Domäne abgestimmt ist, so wird die Sprache auch domänenspezifische Sprache (DSL: Domain Specific Language) genannt [Piet07]. Dabei kann entweder eine konkrete Syntax entworfen werden, die auf eine Domäne zugeschnitten ist, oder die gesamte Sprache (abstrakte und konkrete Syntax) auf die Domäne abgestimmt werden. Im Gegensatz zu den DSLs stehen so genannte universelle Sprachen (General-Purpose Language), die für möglichst viele Zwecke geeignet sein sollen. Beispiel für letztgenannte Sprachen sind allgemeine Programmiersprachen wie Java, C# oder C++. Auf lange Sicht ist es sogar denkbar auch DSLs zu unterstützen, die aus Domänen stammen, welche keinen Bezug zur Informatik haben, so dass Software von Experten einer Domäne geschrieben bzw. modelliert werden kann. M2M- und M2T-Transformationssprachen sind selbst DSLs für die Domäne der Modellgetriebenen Softwareentwicklung. Ein generelles Vorgehen in der Modellgetriebenen Softwareentwicklung sieht vor, die Entwicklung von Generatoren- bzw. Infrastruktur-Projekten getrennt von der Entwicklung von Anwendungs- bzw. Kunden-Projekten zu betreiben. Dabei kann nach dem Prinzip einer „zweigleisig iterativen Entwicklung“ [Völt05] vorgegangen werden, hierbei laufen Infrastruktur- und Anwendungs-Projekt parallel und nach jeder Iteration in der Entwicklung des Anwendungs-Projektes kann es Rückmeldungen für die Entwicklung des Infrastruktur-Projektes geben.

1.2 Architekturzentrierte Modellgetriebene Softwareentwicklung

Bei der Architekturzentrierten Modellgetriebenen Softwareentwicklung (AC-MDSD: Architecture Centric MDSD) geht es primär darum, redundanten und dadurch fehleranfälligen Infrastruktur-Quelltext und Konfigurationsdateien, welche Software-Architekturen oder -Plattformen mit sich

bringen, zu generieren und damit den Entwickler zu entlasten [Völt05]. Dabei müssen die verwendeten Modelle nicht einmal unabhängig von einer Technologie oder Plattform sein, wie es ein Vorschlag der MDA vorsieht (siehe nächstes Kapitel). Die Modelle können im Gegenteil speziell auf eine Plattform zugeschnitten sein, um deren Konzepte möglichst prägnant zu formulieren.

1.3 Model Driven Architecture

Modellgetriebene Architektur (MDA: Model Driven Architecture) beschäftigt sich wie die AC-MDSD damit, Softwarearchitektur mittels Modellen zu definieren. Der Fokus der MDA liegt auf der Unabhängigkeit von Modellen zu einer bestimmten technischen Plattform, was Interoperabilität und Portabilität in der Softwareentwicklung verbessern soll. Es handelt sich also um MDSD für die Domäne der plattformunabhängigen Softwarearchitektur. Die MDA sieht sich mit Anhebung des Abstraktionsgrades und damit der Loslösung von plattformabhängigen Technologien in der Tradition des Compilerbaus und der Entwicklung der Programmiersprachen von Assembler bis zu aktuellen Sprachen der vierten Generation [Fran03]. Die Initiative mit dem Namen MDA [MDA] der Object Management Group (OMG) beschäftigt sich mit der Standardisierung von Vorgehensweisen, Sprachen und Technologien, welche für die MDA eingesetzt werden können. Der MDA-Guide [MDAG] führt die Begriffe des Geschäftsmodells (Computation Independent Model: CIM), plattformunabhängiges Modell (Platform Independent Model: PIM) und plattformspezifisches Modell (Platform Specific Model: PSM) ein. Ein Modell, welches noch unabhängig von Aspekten der Hard- und Software ist, wird CIM genannt. Es beschreibt ein Problem einer Domäne ohne Informationen über die spätere Implementierung des Systems zu enthalten. Das PIM beschreibt ein Softwaresystem, ist aber frei von Informationen einer festgelegten Plattform. Es kann in ein PSM überführt werden, wobei das Modell um Informationen angereichert wird, welche spezifisch für die gewählte Plattform sind. Dieser Vorgang kann entweder manuell erfolgen oder - wenn möglich - durch eine M2M-Transformation. Die Plattformunabhängigkeit ist dabei immer relativ zu einer Plattform zu betrachten. So kann ein PSM auch als ein PIM einer weiteren Transformation dienen.

Als PIM können UML-Modelle eingesetzt werden, die über den UML-Profilmechanismus um Plattform spezifische Informationen angereichert werden. Profile können aber auch Markierungen sein, welche eine Transformation zu einem Plattform spezifischen Modell steuern. Allerdings beschränkt sich die MDA nicht auf die UML als Modellierungssprache, schließlich hat die OMG mit MOF auch eine Technologie zur Definition eigener Modellierungssprachen durch Metamodelle standardisiert (worauf auch im MDA-Guide eingegangen wird).

Der überwiegende Teil der Technologien der MDA lässt sich auch für die allgemeinere MDSD ein-

setzen, weshalb die Abgrenzung zwischen den Begriffen MDA und MDSD mittlerweile an Deutlichkeit verloren hat. Zu den von der OMG entwickelten Standards für die MDA gehören:

- Die MetaObject Facility (MOF) zur Definition von Metamodellen [MOF]
- Die Unified Modeling Language (UML) zur Modellierung verschiedener (struktureller und dynamischer) Aspekte von Softwaresystemen [UML]
- Object Constraint Language (OCL) für die Definition von Einschränkungen in Modellen [OCL]
- Das XML Metadata Interchange (XMI) Format zum Austausch von Modelldaten [XMI]
- MOF Query/View/Transformation (QVT) für die Definition von M2M-Transformationen [QVT]
- Die MOF Model to Text Transformation Language (MOFM2T) zur Definition von M2T-Transformationen [MM2T]

Einen Überblick über die Konzepte der MDA gibt auch das Buch [Mell04] von Mellor et al.

1.4 Query/View/Transformation

Die M2M-Transformationssprache QVT ist, wie bereits erwähnt, eine M2M-Transformationssprache, die von der OMG standardisiert wurde.

1.4.1 Überblick über QVT

Der Standard Query/View/Transformation liegt seit dem 03.04.2008 in der finalen Version 1.0 vor. Die Sprache besteht aus drei Sprachteilen: Core (QVT/C), Operational Mappings (QVT/O) und Relations (QVT/R). Wie die drei Sprachen interagieren können, wird in Abbildung 1 dargestellt. Die Pfeile von „Operational Mappings“ und „Black Box“ zu „Relations“ und „Core“ besagen, dass Funktionen von QVT/O und BlackBox-Implemetierungen bereitgestellt werden können. QVT/O ist eine imperative Sprache, während QVT/C und QVT/R deklarative Sprachen sind. Alle drei Sprachen nutzen OCL für Modellabfragen bzw. erweitern OCL um weitere Ausdrücke. QVT/O ist eine recht komplexe Sprache, was sich dadurch ausdrückt, dass sie einen Großteil der Seiten im Standard für sich in Anspruch nimmt. Die Spezifikation von QVT/O erstreckt sich über 101 Seiten, während die Spezifikation von QVT/C mit 19 Seiten auskommt und QVT/R auf 32 Seiten beschrieben wird. Als weiteres Maß für die Komplexität kann die Anzahl der Schlüsselwörter herangezogen werden. Die Sprache QVT/O gibt eine nicht unerhebliche Menge von 116 Schlüsselworten vor. QVT/R defi-

niert hingegen nur 15 Schlüsselworte, QVT/C kommt sogar mit elf reservierten Wörtern aus. Zu allen drei Sprachen kommen noch die Schlüsselworte von OCL hinzu. QVT/C ist eine sich auf einige wesentliche Sprachelemente beschränkende Sprache. Sie ist gleich mächtig wie QVT/R, allerdings führt ihre Einfachheit dazu, dass semantisch äquivalente Transformationen in QVT/C wortreicher ausfallen. Transformationen, die in QVT/R geschrieben sind können automatisch nach QVT/C übersetzt werden. Eine Implementierung für QVT/R könnte also aus einer QVT/C Engine bestehen, die zur Ausführung die Transformation nach QVT/C übersetzt. In der QVT-Sprachspezifikation wird eine Analogie zur Java Virtual Machine (JVM) gezogen, die vom Java-Quelltext nach Java-Bytecode übersetzt, der dann von der JVM interpretiert werden kann (Seite 10, Kapitel 6.1.3). Neben den standardisierten Sprachen erlaubt die QVT-Spezifikation BlackBox-Operationen in fremden Sprachen zu definieren. In der QVT-Sprachspezifikation ist eine Transformation vom QVT/R-Metamodell zum QVT/C-Metamodell vorgegeben. Diese ist allerdings in QVT/R verfasst, so dass eine QVT/C Engine entweder eine manuelle Übersetzung dieser Transformation nach QVT/C bräuchte oder in einem Bootstrapping-Prozess müsste eine existierende QVT/R-Engine diese Transformation in eine QVT/C Transformation übersetzen.

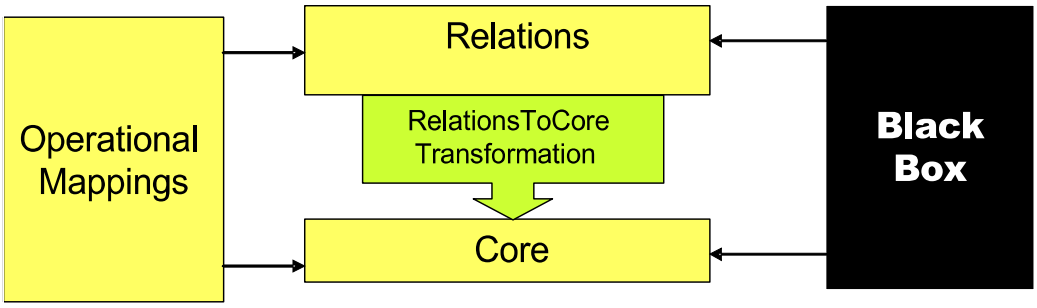


Abbildung 1: Zusammenhang der QVT-Sprachen (aus der QVT-Spezifikation)

Einen Überblick über QVT bietet auch der Artikel [Kur07].

1.4.2 QVT Relations

Zunächst wird eine kleine Beispiel-Transformation in konkreter textueller Syntax eingeführt. Diese wird zur Verdeutlichung der wichtigsten Konstrukte der Sprache QVT Relations genutzt. Danach wird im Detail auf die konkrete graphische Syntax von QVT/R eingegangen.

1.4.2.A Einführung in QVT/R

Das folgende Beispiel erhebt keinen Anspruch auf Praxistauglichkeit, sondern dient lediglich zur Anschauung. Die Transformation, mithilfe der die Einführung in QVT/R gegeben wird, wandelt

dem Metamodell „directory“ (siehe Abb. 2) entsprechende Modelle in (siehe Abb. 3) Modelle um, die dem Metamodell „container“ entsprechen.

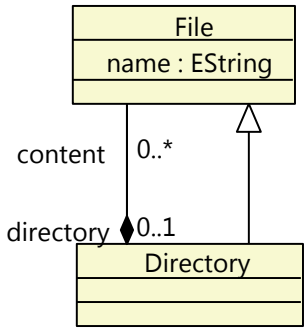


Abbildung 2: Metamodell „directory“

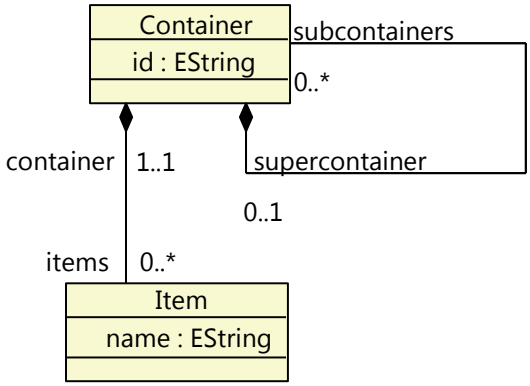


Abbildung 3: Metamodell „container“

```
1  transformation directory2container(di : directory, co : container) {
2
3    key container::Container { id , supercontainer };
4
5    key container::Item { name , container };
6
7    top relation RootDirectoryToContainer {
8
9          checkonly domain di d : directory::Directory { };
10
11         enforce domain co c : container::Container { };
12
13         when {
14               d.directory.oclIsUndefined();
15         }
16
17         where {
18               DirectoryToContainer(d, c);
19         }
20
21   }
```

```
22
23  relation DirectoryToContainer {
24
25        n : String;
26
27        checkonly domain di d : directory::Directory { };
28
29        enforce domain co c : container::Container {
30              id = n
31        };
32
33        where {
34              n = nameToId(d.name);
35              InnerDirectoryToSubcontainer(d, c);
36              FileToItem(d, c);
37        }
38
39  }
40
41  relation InnerDirectoryToSubcontainer {
42
43        checkonly domain di d : directory::Directory {
44              content = id : directory::Directory { }
45        };
46
47        enforce domain co c : container::Container {
48              subcontainers = sc : container::Container { }
49        };
50
51        where {
52              DirectoryToContainer(id, sc);
53        }
54
55  }
56
57  relation FileToItem {
58
59        n : String;
60
61        checkonly domain di d : directory::Directory {
62              content = f : directory::File {
63                    name = n
64              }{
65                    self.oclIsTypeOf(dir::File)
66              }
67        };
68
69        enforce domain co c : container::Container {
70              items = i : container::Item {
71                    name = n
72              }
73        };
74
75  }
76
77  query nameToId (name : String) : String {
78        name + '_id'
79  }
```

```
80
81  }
```

Quelltext 1: Einfache QVT/R-Transformation

Mit Hilfe der Sprache QVT/R werden Transformationen (Transformations) zwischen zwei oder mehr Modellen, die bestimmten Metamodellen entsprechen (Typed Models), beschrieben. Diese Modelle sind die Parameter der Transformation, im Beispiel sind dies `di` und `co`. Transformationen bestehen aus mehreren Relationen (Relations), die Beziehungen zwischen Metamodellelementen der an der Transformation beteiligten Modelle (Domains) definieren. Diese Beziehungen gelten in alle Richtungen. Im Folgenden wird diese Eigenschaft als Multidirektionalität bezeichnet. Zur Ausführung der Transformation muss angegeben werden, in Richtung welchen Modells die Transformation ausgeführt werden soll. Im Beispiel müsste zwischen Richtung `di` und `co` gewählt werden. Im Folgenden wird das Modell, in dessen Richtung transformiert werden soll, „Ausgabemodell“ oder „Zielmodell“ genannt. Die Modelle, in deren Richtung nicht transformiert wird, werden „Eingabemodelle“ genannt. Als Einstiegspunkte für die Transformation gelten die Relationen, die als „toplevel“ ausgezeichnet sind. In der konkreten textuellen Syntax steht das Schlüsselwort „top“ vor der Definition der Relation. Das Beispiel stellt lediglich eine toplevel Relation, nämlich `RootDirectoryToContainer`. Ob eine Relation überhaupt gelten muss, kann mittels eines „When-Block“ angegeben werden. Die Relation muss also nur dann erfüllt werden, wenn die Ausdrücke des Blocks zum Wert „wahr“ (true) ausgewertet werden. Für das Beispiel bedeutet dies, dass die Relation `RootDirectoryToContainer` nur dann ausgewertet wird, wenn das Directory-Modellelement, welches als Parameter übergeben wird, kein Elternverzeichnis hat (siehe Zeile 10); damit gilt diese Relation nur für das Wurzelverzeichnis. Eine Relation kann zwei oder mehr Domains beinhalten, diese stellen die Parameter für eine Relation dar. Für jede Domain wird angegeben, für welchen Transformationsparameter (also für welches Typed Model) sie gelten. Im Beispiel wäre das `di` oder `co`. Domains können mit dem Schlüsselwort „checkonly“ oder „enforce“ gekennzeichnet werden. Wird in Richtung einer Domain transformiert, die mit „enforce“ gekennzeichnet ist, so kann die ausführende Engine Modifikationen am Zielmodell vornehmen, um die Relation wahr werden zu lassen. Werden Domains als „checkonly“ ausgezeichnet, so wird, wenn die Transformation in die Richtung dieser Domäne ausgeführt wird, lediglich überprüft, ob die Relation gilt. Gilt sie nicht, so wird eine Warnung ausgegeben. Domänen sind entweder Variablen primitiven Typs (Primitive Domains) oder Vorlagen (Templates). Im Beispiel werden ausschließlich Templates, genau genommen Object Templates (siehe unten), verwendet. Es stehen Objekt-Vorlagen (Object Templates) und Sammlungs-Vorlagen (Collection Templates) zur Verfügung. Object Templates definieren Variablen für einen Typ von Modellelement und treffen eine Aussage über die Attribute

dieses Modellelements (in Property Template Items). Property Template Items geben mittels eines OCL-Ausdrucks an, welchen Wert ein Attribut eines Modellelementes annehmen kann, damit das Modellelement dem Template genügt. Diese OCL-Ausdrücke können selbst auch wieder Template Definitionen sein. Zeile 44 der Beispieltransformation zeigt solch ein geschachteltes Object Template. Collection Templates sind Vorlagen für OCL-Collections und geben an, welche Elemente in einer Collection vorhanden sein müssen, um dem Template zu genügen. Um die Elemente zu beschreiben, welche in der Collection auftreten müssen, können entweder Variablennamen, geschachtelte Templates oder der Variablenname „_“, welcher für ein beliebiges Element steht, angegeben werden. Zusätzlich, in der konkreten textuellen Syntax mittels „++“ abgetrennt, muss in jedem Collection Template der Name einer Collection-Varablen angegeben werden, welche alle restlichen Elemente aufnehmen kann. Für diese Variable kann selbst auch wieder die Variable „_“ verwendet werden.

```
1  is : Set(container::Item){
2     i : container::Item{ name = n }, i2 ++ _
3  }
```

Quelltext 2: Ein Collection Template mit Object Template und Variablennamen

Erfüllt ein Modellelement bzw. eine Collection die Bedingungen eines Templates, so ist dies ein Treffer (Match) für das Template und wird an die Template-Variable gebunden. Um die Menge der Matches für ein Template weiter einzuschränken, können für Templates zusätzliche OCL-Beschränkungen (Constraint) angegeben werden, die in der textuellen Syntax in geschweiften Klammern hinter der Template-Definition angeordnet sind. Im Quelltext 1 in Zeile 65 ist ein Beispiel für eine solche Beschränkung gegeben. Der Where-Block einer Relation gibt an, welche Zusammenhänge gelten müssen, um die Relation zu erfüllen. Dabei können auch andere Relationen mit Variablen aus der Relation aufgerufen werden. Zusätzlich können Zusammenhänge dadurch definiert werden, dass an unterschiedlichen Stellen einer Relation dieselbe Variable in Property Template Items genutzt wird. Die ausführende QVT-Engine muss - wenn nötig - Anpassungen am Ausgabemodell vornehmen, um die geforderten Zusammenhänge herzustellen. So kann ein Where-Block die Erzeugung von Template-Matches im Ausgabemodell bewirken. Es werden also in in der Domain, in deren Richtung transformiert wird, Modellelemente erzeugt, die an Template-Variablen gebunden werden, damit die Bedingungen der Relation erfüllt sind. Existieren mehrere Matches für ein Template, so müssen die Beziehungen, welche im Where-Block oder über das Benutzen von gleichen Variablen an unterschiedlichen Stellen aufgestellt wurden, für alle Matches gelten. Wenn es in einer Relation Matches für Templates einer „enforce“ Domain für das Ausgabemodell gibt, für die Beziehungen definiert sind, die nicht erfüllt sind, so müssen diese Modellelemente aus dem Zielmodell gelöscht

werden.

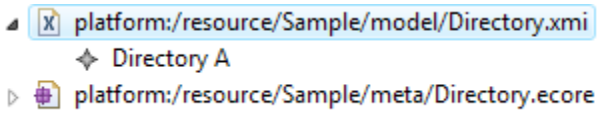


Abbildung 4: Belegung für Transformationsparameter `co`

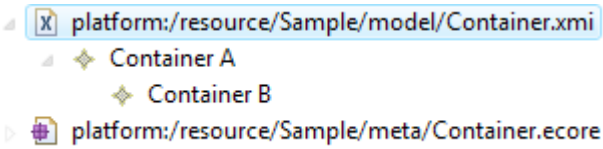


Abbildung 5: Belegung für Transformationsparameter `di`

Beispiel: Als Eingabe für die Transformation aus Quelltext 1 in Richtung `co` sei die Directory-Modelldatei gegeben, welche in Abbildung 4 gezeigt wird. Das Zielmodell wird aus der Modelldatei gelesen, welches in Abbildung 5 dargestellt ist.
Mit `InnerDirectoryToSubcontainer` besteht eine Regel, die Unterverzeichnisse auf Subcontainer abbildet. Da für Subcontainer B kein entsprechendes Verzeichnis vorhanden ist, wird er bei einer Transformation in Richtung Container gelöscht.
Wird keine Aussage (in Form einer Relation) über bestehende Modellelemente oder Attribute im Ausgabemodell getroffen, so bleiben diese Elemente erhalten. Es können auch Standardwerte (Default Values) angegeben werden, die gesetzt und nachträglich manuell geändert werden können, ohne bei einer Retransformation überschrieben zu werden. Um Modellelemente eindeutig zu identifizieren, können Attribute von Modellelementen als Schlüsselattribute festgelegt werden. Auf diese Weise wird verhindert, dass mehrere Elemente mit gleichen Werten in Schlüsselattributen im Zielmodell erzeugt werden. Sie können auch dazu führen, dass die QVT-Engine bestehende Elemente im Ausgabemodell anpasst. Dies ist dann der Fall, wenn bei einem Match nur die Schlüsselattribute übereinstimmen, aber weitere Beschränkungen nicht erfüllt sind. In Quelltext 1 sind zwei Definitionen für Schlüssel-Attribute angegeben (Zeilen 3 und 5). Um wiederkehrende OCL-Ausdrücke wiederverwendbar auszulagern, bietet der Standard die Möglichkeit, Abfragen (Queries) zu definieren. Queries sind nebeneffektfreie Funktionen mit beliebig vielen Eingabeparametern und einem Rückgabetyp. Ein simples Beispiel für eine Query ist in Zeile 77 des Beispiel-Quelltextes 1 zu sehen.

Für die Sprache QVT Relations wurde neben der konkreten textuellen auch eine konkrete graphische Syntax standardisiert. Es handelt sich bei der Syntax um eine Abwandlung von UML Objekt-

diagrammen. Allerdings wurde nur die graphische Notation übernommen, das QVT/R-Metamodell hat keine Abhängigkeit zur UML.

1.4.2.B Elemente der konkreten graphischen Syntax

Die folgenden Elemente werden in der QVT/R-Sprachspezifikation erwähnt und erläutert. Die graphische Syntax umfasst nur einen Teil der Elemente der abstrakten Syntax. In der folgenden Beschreibung wird auch auf Unklarheiten in den Beschreibungen der Syntaxelemente im Standard eingegangen.

1.4.2.B.i Relation

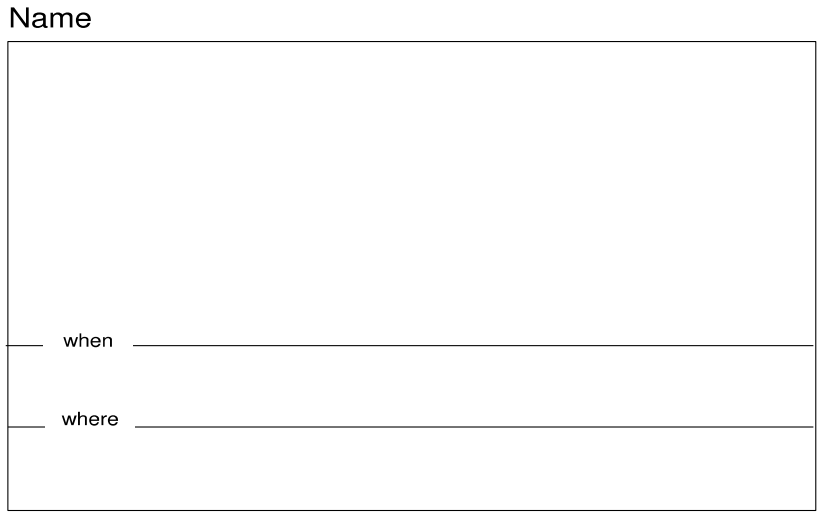


Abbildung 6: Graphische Syntax für Relations

In dem obersten Abteilung können die unten genannten Elemente platziert werden. When- und Where-Blocke dürfen ausgeblendet werden, wenn sie leer sind.

1.4.2.B.ii Relation Middle

Abbildung 7: Graphische Syntax für Relation Middle (aus der QVT-Spezifikation)

Dieses Syntaxelement hat keine Entsprechung in der abstrakten Syntax. Ein Exemplar dieses Elementes muss in der Mitte der oberen Abteilung der Relation vorhanden sein und wird mit allen

Domains der Relation verbunden. An den Verbindungspfeilen werden die Typed Models angegeben, für die die verbundene Domain gilt. Der Buchstabe „C“ wird angegeben, wenn die Domain „checkonly“ sein soll, der Buchstabe „E“, wenn die Domain mit „enforce“ gekennzeichnet werden soll. Diese Informationen an den Verbindungspfeilen sind im Metamodellelement der Domains gekapselt. In der Implementierung des visuellen Editors musste für dieses Element eine Entsprechung im Metamodell geschaffen werden.

1.4.2.B.iii Domain

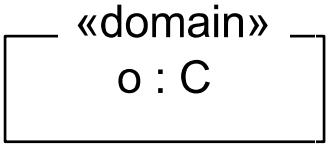


Abbildung 8: Graphische Syntax für Domains (aus der QVT-Spezifikation)

Der Standard macht in der graphischen Syntax keinen Unterschied zwischen einer Domain, die durch ein Object Template, durch ein Collection Template oder einer Primitive Domains beschrieben ist. In den Beispielen, welche im Standard gegeben werden, enthält diese Darstellung von Domains eine zusätzliche Abteilung für Property Template Items, so wie es bei der Darstellung für Object Templates der Fall ist (siehe unten). Allerdings ist anzumerken, dass Primitive Domains und Collection Domains keine Property Template Items enthalten können. Dies wird im Standard nicht explizit erwähnt, ist aber durch das Metamodell vorgegeben.

1.4.2.B.iv Object Template

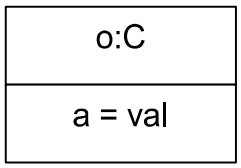


Abbildung 9: Graphische Syntax für Object Templates (aus der QVT-Spezifikation)

Die Abteilung, in der Property Template Items definiert werden, ist optional, bzw. darf ausgeblendet werden.

1.4.2.B.v Collection Template

Abbildung 10: Graphische Syntax für Collection Templates (aus der QVT-Spezifikation)

Auf Seite 45 des Standards wird dieses graphische Syntaxelement als Notation für Object Template festgelegt, das auf eine Menge von Modellelementen zutrifft. In einem Beispiel auf Seite 43 hingegen wird das Element benutzt wie ein Collection Template. Ansonsten wird dieses Syntaxelement an keiner weiteren Stelle im Standard erwähnt. Da eine Mehrfachbelegung des Object Templates und das Auslassen von Collection Templates wenig Sinn ergibt, wird im Folgenden davon ausgegangen, dass dieses Syntaxelemente für Collection Templates steht.

1.4.2.B.vi Not Template

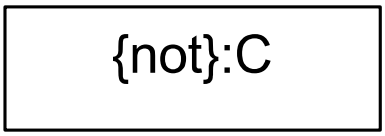


Abbildung 11: Graphische Syntax für Not Templates (aus der QVT-Spezifikation)

Not Templates sind in der abstrakten Syntax nicht vorhanden. Im Standard heißt es zu diesem Syntaxelement (Seite 44 und 46), dass ein Not Template die Nichtexistenz von bestimmten Objekten sicherstellt. Das Template soll nur dann zutreffen, wenn kein Objekt existiert, welches den Beschränkungen (Constraints) entspricht, die mit ihm verknüpft sind. Daraus ergibt sich, dass dieses Konstrukt bei Überführung in abstrakte oder konkrete textuelle Syntax auf OCL-Constaints abgebildet werden, die sicherstellen, dass das Attribut entweder den Wert OclUndefined hat (bei Einzelwerten) oder eine leere Collecion ist (bei Attributen mit einer Multiplizität höher als Eins). Ist das Attribut, für das das Template gilt eine Collection, so muss zusätzlich auf ein Property Template Item abgebildet werden. Der Standard definiert diese Abbildung allerdings nicht genau, sondern bietet nur ein Beispiel für eine Übersetzung für ein Not Template, welches für ein Attribut mit einer Multiplizität höher als Eins gilt. Zudem ist in dem Beispiel auch kein OCL-Contraint für das Not Template angegeben. Daher stellt sich die Frage, in welchem Kontext angefügte OCL-Constraints überhaupt gelten. Sie könnten entweder im Kontext der einzelnen Elemente des Attributs oder für das Collection-Objekt gelten, welches den gesamten Wert des Attributes darstellt. In der Implementierung zu diese Arbeit wird davon ausgegangen, dass ein OCL-Constraint für die einzelnen Elemente

des Attributs gilt.

1.4.2.B.vii OCL Constraint

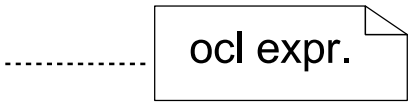


Abbildung 12: Graphische Syntax für OCL Constraints (aus der QVT-Spezifikation)

Constraints können an Templates, Domains und an die Verbindungen von Relation Middle zu Domain angehängt werden. Im letzten Fall gelten die Ausdrücke für die Domains.

1.4.2.C Vorteile von QVT/R

QVT ist die einzige von der OMG standardisierte M2M-Transformationssprache und hat damit alle Vorteile, die sich aus einem herstellerunabhängigen Standard ergeben.
Die Sprache QVT Relations erlaubt die multidirektionale Ausführung von Transformationen und ist damit geeignet für die Synchronisation von Modellen. Der Sprachstandard sieht auch einen „checkonly“ Modus vor (Kapittel 7.9, Seite 19), der das Überprüfen von Konsistenz zwischen mehreren Modellen erlaubt, ohne dabei Modelle zu verändern.
Ein weiterer Vorteil von QVT/R ist die inkrementelle Modifikation von Zielmodellen, so dass Modelle nicht immer vollständig generiert werden müssen. Dabei können auch manuelle Änderungen am Zielmodell erhalten bleiben. Wie bereits erwähnt, kann dazu der Mechanismus „Default Values“ genutzt werden. Vor allem bei einem iterativen Entwicklungsprozess sind hier Vorteile zu erwarten.
Die implizite Erstellung von Trace-Modellen, welche festhalten, wie an einer Transformation beteiligte Modellelemente miteinander in Relation stehen, ist eine zusätzliche Stärke der Sprache. Damit wird eine Rückverfolgung gewährleistet, mit der unter anderem festgestellt werden kann, welche Modellelemente aus welchen erzeugt wurden.

1.4.2.D Nachteile von QVT/R

QVT/R wird aufgrund seiner deklarativen Syntax und mathematisch anmutenden Semantik der Beziehungen vorgeworfen, wenig pragmatisch zu sein [Boh06]. Programmierer sind in der Regel imperative Programmiersprachen gewohnt. Die Unterstützung von Multidirektionalität führt dazu, dass Transformationen umständlicher und wortreicher notiert werden müssen. In der MDSD wird allerdings oft von abstrakteren zu konkreteren und detailreicheren Modellen transformiert. In diesen Szenarien spielt die Multidirektionalität oft keine große Rolle. Sie wird entweder nicht benötigt

oder ist gar nicht eindeutig definierbar. Hinzu kommt, dass QVT/R zwar die Möglichkeit bietet, multidirektionale Transformationen zu definieren, aber nicht jede Transformation ist automatisch multidirektional. Um eine Ausführung in Richtung aller Parameter zu erlauben, muss auf Sprachkonstrukte verzichtet werden, die sich nicht invertieren lassen [Stev07]. Anders als Triple Graph Grammar (TGG) [Kin07] unterstützt QVT/R auch keine vollständige Synchronisation zwischen unabhängig voneinander modifizierten Modellen [Gre07]. Das folgende Beispiel zeigt Probleme bei der Synchronisierung zweier unabhängig voneinander modifizierten Modelle mittels QVT/R. Dazu wird eine bidirektionale Variante der in Quelltext 1 eingeführten Transformation herangezogen.

Beispiel: Die Abbildungen 13 und 14 zeigen Änderungen an bestehenden Modellen, die über Transformationen synchron gehalten werden sollen.

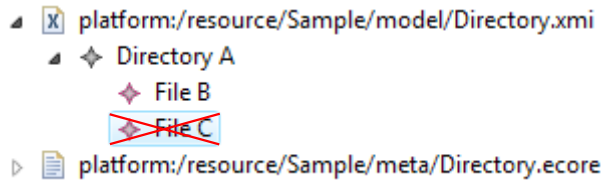


Abbildung 13: Änderung im Directory Modell

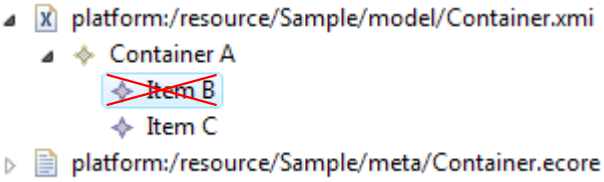


Abbildung 14: Änderung im Container Modell

Sollten die veränderten Modelle nun mittels QVT/R synchronisiert werden, würden je nach Transformationsrichtung entweder Item B und File B oder File C und Item C erhalten bleiben.
Es erscheint für einen echten Synchronisationsmechanismus allerdings sinnvoller, entweder File B und Item C wiederherzustellen oder Item B und File C zu löschen. Beide Möglichkeiten können nicht mit QVT/R umgesetzt werden, da sich Ausgangsmodelle bei QVT/R-Transformationen nicht ändern können.
Ein Nachteil, den beide deklarativen QVT Sprachen gegenüber der imperativen Sprache QVT/O haben, ist die fehlende Ausnahmebehandlung, so dass ein Fehler in einer Transformation immer zum Abbruch der Transformation führt.
Teile der QVT/R-Spezifikation sind noch immer unklar [IKVProb], weshalb die OMG auch bereits eine Task Force ins Leben gerufen hat, um diese Unklarheiten für die kommende Version 1.1 des

Standards auszuräumen [QVT11TF].
Das bisher einzige Buch [Nol09], welches sich ausführlich mit der Sprache QVT Relations beschäftigt, erscheint erst nach Fertigstellung dieser Diplomarbeit. Es existiert also bisher wenig Literatur zu QVT Relations, was den Einstieg in die Sprache erschwert. Um die Sprache zu erlernen, musste bisher immer der Standard gelesen werden. Ein englischsprachiges Buch ist bisher noch nicht erschienen.

1.4.2.E Vorhandene Engines

Es gibt vier Engines, welche QVT/R ausführen können. Es werden allerdings nicht von allen alle Sprachfeatures unterstützt.

1.4.2.E.i Medini QVT

Die QVT/R-Engine Medini QVT wird von der in Berlin ansässigen Firma ikv++ technologies ag entwickelt. Die Engine ist unter der Open Source Lizenz Eclipse Public License (EPL) veröffentlicht.

Der Medini QVT Engine fehlt noch die Unterstützung einer Reihe von Syntaxelementen der QVT/R-Spezifikation, u.a. Default Values, Collection Templates, OCL-Constraints für Templates bzw. Domains, BlackBox Operationen. Die Engine kann auch nicht mit dem Referenzieren von Template-Variablen innerhalb einer Domain umgehen. Die Variablen müssen mehrfach mit der gleichen Bezeichnung definiert werden.
Beispiel: Statt der standardkonformen Notation in Quelltext 3 muss in der Medini für die Ausführung die Schreibweise in Quelltext 4 gewählt werden. Der Unterschied ist jeweils in Zeile 9 zu sehen.

```
1  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
2      rdbmsName = cn,
3      rdbmsSchema = s : SimpleRDBMS::RdbmsSchema { },
4      rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
5        rdbmsName = cn + '_tid',
6        rdbmsType = 'NUMBER'
7      },
8      rdbmsKey = k : SimpleRDBMS::RdbmsKey {
9        rdbmsColumn = cl
10      }
11  };
```

Quelltext 3: Standardkonforme Notation

```
1  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
2      rdbmsName = cn,
3      rdbmsSchema = s : SimpleRDBMS::RdbmsSchema { },
4      rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
5        rdbmsName = cn + '_tid',
6        rdbmsType = 'NUMBER'
7      },
8      rdbmsKey = k : SimpleRDBMS::RdbmsKey {
9        rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn{}
10      }
11  };
```

Quelltext 4: Notation für Medini

Die Engine weicht auch an einer Stelle von dem im Standard vorgeschriebenen Verhalten ab. Sie führt nur dann Lösch-Operationen auf dem Zielmodell aus, wenn ein Element aus dem Ausgangsmodell gelöscht wird, für das im Trace einer vorherigen Transformation Modellelemente im Zielmodell verzeichnet sind. Diese Elemente im Zielmodell werden gelöscht. Das bedeutet, bei Retransformation mit der Medini QVT Engine bleiben alle manuell hinzugefügten Elemente im Zielmodell erhalten.

Neben der Engine bietet ikv++ technologies ag Eclipse Plug-ins für die QVT-Entwicklung, die unter dem Namen Medini Cockpit firmieren. Dazu gehört ein Editor für Quelltext mit Codevervollständigung für QVT-Syntax und Metamodell-Informationen. Die Medini Cockpit Plug-ins stehen im Gegensatz zur Medini QVT Engine nicht unter einer Open Source Lizenz, sondern sind lediglich kostenlos für nicht kommerzielle Zwecke nutzbar.

1.4.2.E.ii Eclipse QVTD

Das von der französischen Firma Obeo entwickelte Eclipse Projekt QVT Declarative [EQVTD] hat zum Ziel, die deklarativen Sprachen QVT/R und QVT/C des QVT-Standards zu implementieren. Die Implementierung steht unter der EPL und basiert auf der virtuellen Maschine der Transformationssprache ATL, welche ebenfalls unter dem Dach des Eclipse Projektes entwickelt wird. Eine erste Vorabversion, welche im Oktober des Jahres 2008 veröffentlicht wurde, stellte sich als recht fehleranfällig heraus. Zudem ist noch nicht dokumentiert, wie die API der Engine zu nutzen ist. Auch für Eclipse QVT Declarative werden Eclipse Plug-ins für die QVT Entwicklung erstellt. Diese stehen ebenfalls unter der EPL.

1.4.2.E.iii VMTS

Für das an der Universität von Budapest für Technologie und Ökonomie entwickelte Framework VMTS [VMTS] zum Erstellen von Metamodellen, Modellen und Tarnsformationen wird auf der Projektseite ein Plug-in vorgestellt. Dieses unterstützt welches QVT Relations sowohl in textueller

als auch in graphischer Syntax. Leider konnte nicht überprüft werden, wie weit die QVT/R-Kompatibilität reicht, da das Programm auf drei unterschiedlichen Betriebssystemen bereits beim Start abstürzte. Auch die Entwickler konnten nicht mitteilen, was den Fehler verursacht.

1.4.2.E.iv ModelMorph

Die QVT/R-Engine ModelMorph wurde von der Forschungsabteilung der in Indien ansässigen Firma Tata Consultancy Services entwickelt. Sie hat auch an der Entwicklung des QVT Standards mitgewirkt. Die Dokumentation von ModelMorph ist recht spärlich und gibt leider nicht an, welche Syntaxelemente unterstützt werden. Die Projektseite der Engine ist mittlerweile nicht mehr Online, das Programm kann nur noch über eine Übersichtsseite [TATA] bezogen werden. Die Engine steht nicht unter einer Open Source Lizenz, sondern unter einer eigenen Lizenz, die die Nutzung für nicht kommerzielle Zwecke genehmigt.

1.5 Motivation

Da graphische Modelle (bis zu einer gewissen Größe) oft intuitiv und leichter erfassbar sind als textuelle Repräsentation, bietet es sich an, auch Modelltransformationen in einer graphischen Art und Weise zu notieren. Da mit QVT/R bereits eine graphische Syntax für Modelltransformationen standardisiert ist, erscheint es sinnvoll, diesen Standard als Grundlage für eine Implementierung heranzuziehen.

2 Problemstellung und Ziel der Arbeit

Ziel der Arbeit war, die Entwicklung eines Frameworks, mit dem Transformationen visuell definiert werden können. Dies sollte in der von der OMG standardisierten graphischen Syntax von QVT Relations geschehen. Fehlerhafte Transformationen sollten im Editor bereits während der Modellierung aufgezeigt werden (Live Validation). Auf diese Weise sollen Fehler früh erkannt und einfach lokalisiert werden. Das Framework sollte aus der graphischen in die textuelle Syntax übersetzten können, um die Ausführung der Transformationen durch eine QVT/R-Engine zu ermöglichen. Dies sollte nach Möglichkeit automatisiert geschehen, so dass graphische und textuelle Syntax automatisch synchron gehalten werden. Ein weiteres Ziel war es einen Mechanismus zu implementieren, der von einer QVT/R-Engine gemeldete Fehler in die graphische Darstellung übernimmt. Da ein Vorteil von QVT/R die implizite Erstellung von Trace-Daten ist, sollte dies auch durch das Framework genutzt werden. Trace-Modelle sollten eingelesen werden können und einen geeigneten Zugriff über die Editor-Komponente bereitstellen. Das Framework sollte auf Basis des Eclipse Modeling Framework (EMF) entwickelt werden, da EMF eine populäre Implementierung von Essential MOF ist, auf dessen Basis bereits viele weitere Projekte und Implementierungen von OMG Standards beruhen [EUML2], [EBPN2], [EQVTD], [Acceleo]. Dem Nutzer sollte es möglich sein, Ecore Metamodelle anzugeben, dessen Informationen beim Erstellen von Transformationen genutzt werden können. Es sollten also nur Regeln erstellt werden können, die gültige Bezeichner aus den gewählten Metamodellen verwenden.

Da der Kern des Frameworks unabhängig von einer bestimmten QVT/R-Transformationsengine sein sollte, müssten alle Teile des Frameworks, die Abhängigkeiten zu einer Transformationsengine besitzen, austauschbar hinter Schnittstellen verborgen werden. Dies betrifft vor allem die Bereitstellung von Trace-Modellen und die Anzeige von Fehlern, welche eine QVT/R-Engine meldet.

Ein weiteres Ziel war, eine Machbarkeitsstudie (proof of concept) durchzuführen, in der Informationen aus QVT/R-Quelltext in das graphische Modell übernommen werden und damit eine Synchronisation zwischen graphischer und textueller Syntax zu erhalten (Roundtrip Fähigkeit).

Ziel war auch die Einbindung der ausführenden QVT-Engine die openArchitectureWare (oAW) [oAW] Workflow Engine, um eine Integration von QVT/R in die oAW Toolkette so einfach wie möglich gestalten zu können.

Bei Entwicklung des Frameworks sollte nach Möglichkeit selbst modellgetrieben verfahren werden.

3 Verwandte Arbeiten

Für diese Arbeit sind Modelltransformationssprachen interessant , die eine graphische Syntax bieten. Das sind z.B. Triple Graph Grammar unterstützende Sprachen. Hierzu gehören Fujuba [Fujuba] und MOFLON [MOFLON] [Kön05], welches auf Fujuba basiert. In seiner Dissertation stellt Alexander Königs die graphische Syntax der TGG Implementierung von MOFLON vor [Kön08], die er als Triple Graph Grammar basierte QVT Implementation bezeichnet. Die MOFLON Syntax implementiert etliche graphische QVT/R-Syntaxelemente, weicht aber teilweise stark von dieser ab. So können mehrere Relationen, die durch das Relation Middle Syntaxelement repräsentiert werden, mit ein und demselben Template verbunden werden, was in der standardisierten Syntax nicht möglich ist. Es gibt in MOFLON weder Collection Templates noch Not Templates. Ihr fehlen auch einige QVT/R-Konzepte wie Schlüsselattribute und wiederverwendbare Queries. Where-Blöcke werden nicht unterstützt, dafür hat MOFLON ein eigenes, weniger ausdrucksstarkes Konzept von Abhängigkeits-Verknüpfungen in der graphischen Syntax. Nach Angaben von Königs soll der bereits länger bekannte TGG Formalismus dafür eine präzisere Semantik als QVT/R haben. TGG ist zudem besser für Modellsynchronisation geeignet, wie in Kapitel 1.4.2.D beschrieben ist. MOFLON implementiert CMOF 2.0, bietet allerdings keine Unterstützung für Ecore Metamodelle.

Auch die Projekte AGG [AGG] und GreAT [GreAT] bieten die graphische Definition von Transformationsregeln. Die Syntax der graphischen Darstellung ist nicht standardisiert, sondern orientiert sich allenfalls an Grundsätzen wie TGG, so dass bei diesen Projekten jede Implementierung seine eigenen Wege geht. Die beiden Projekte VMTS [Leng06] und UMLX [UMLX] streben eine vollständige Unterstützung der konkreten graphischen Syntax von QVT/R an. Während UMLX nach Angaben der Entwickler auf einem Entwicklungsstand ist, in dem der Editor noch nicht sinnvoll genutzt werden kann [Wil09], ist VMTS nach Bekunden der Entwickler bereits einsatzbereit. Dies ließ sich allerdings, wie in Kapitel 1.4.2.E.iii bereits erwähnt, nicht nachvollziehen.

Kein Entwicklungswerkzeug, der in diesem Kapitel vorgestellten Sprachen, scheint eine Möglichkeit zu besitzen, aus den graphischen Transformationsbeschreibungen heraus auf Trace-Daten zugreifen zu können.

4 Konzepte für die Realisierung

In diesem Abschnitt werden allgemeine Ideen, Konzepte und Techniken erläutert, die in der Lösung zum Tragen kommen, aber auch unabhängig von der Implementierung in dieser Diplomarbeit gesehen werden können. Um die Konzepte zu verdeutlichen, werden Beispiele aus dem Visual QVT/R Framework (im Folgenden auch VQVT/R Framework genannt) gegeben, die für ein besseres Verständnis des jeweiligen Konzeptes sorgen sollen.

4.1 Modellvalidierung

Für die Validierung in graphischen Editoren für Modell-Transformationssprachen ließen sich drei grundsätzliche Arten der Validierung feststellen.

4.1.1 Strukturelle Aspekte des Diagramms

Fehlende Elemente oder fehlende Verknüpfungen in einem Diagramm stellen Fehler in der Struktur des Diagramms dar. Solche ungültigen Diagramme repräsentieren ungültige Modelle, weshalb hier Modellvalidierung angewandt werden kann, um solche strukturellen Fehler aufzudecken.

Beispiel 1: Im visuellen QVT/R-Editor müssen Relationen, die weniger als zwei Domänen haben, als fehlerhaft markiert werden. Quelltext 5 zeigt einen oAW Check Constraint, der diese Regel überprüft. Abbildung 15 zeigt die Auswirkung im Editor, wenn diese Regel nicht eingehalten wird.

```
context Relation ERROR "Relations must have at least two domains" :
      content.typeSelect(RelationDomain).size+
      content.typeSelect(PrimitiveDomain).size >= 2;
```

Quelltext 5: Constraint der auf das Vorhanden sein von zwei Domains prüft

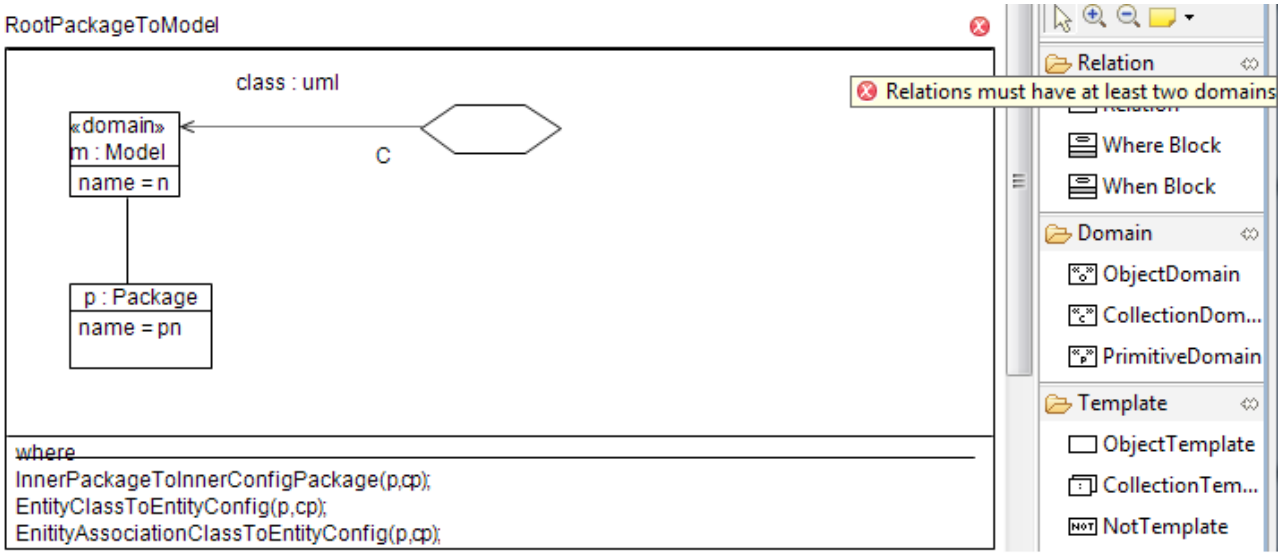


Abbildung 15: Auswirkung der Validierung bei nur einer Domäne

Beispiel 2: Um den Benutzer davor zu bewahren, ungültige Modelle zu erzeugen, werden einige OCL-Constraints eingeführt, die das Ziehen ungültiger Verbindungen unterbinden. Quelltext 6 zeigt einen Constraint, der im Kontext von Templates gilt und der verhindert, dass Constraints zu Templates anderer Relationen gezogen werden können.

```
1  self.relation = oppositeEnd.relation
```

Quelltext 6: Constraint, der prüft, ob das Ziel der Verbindung der gleichen Relation angehört

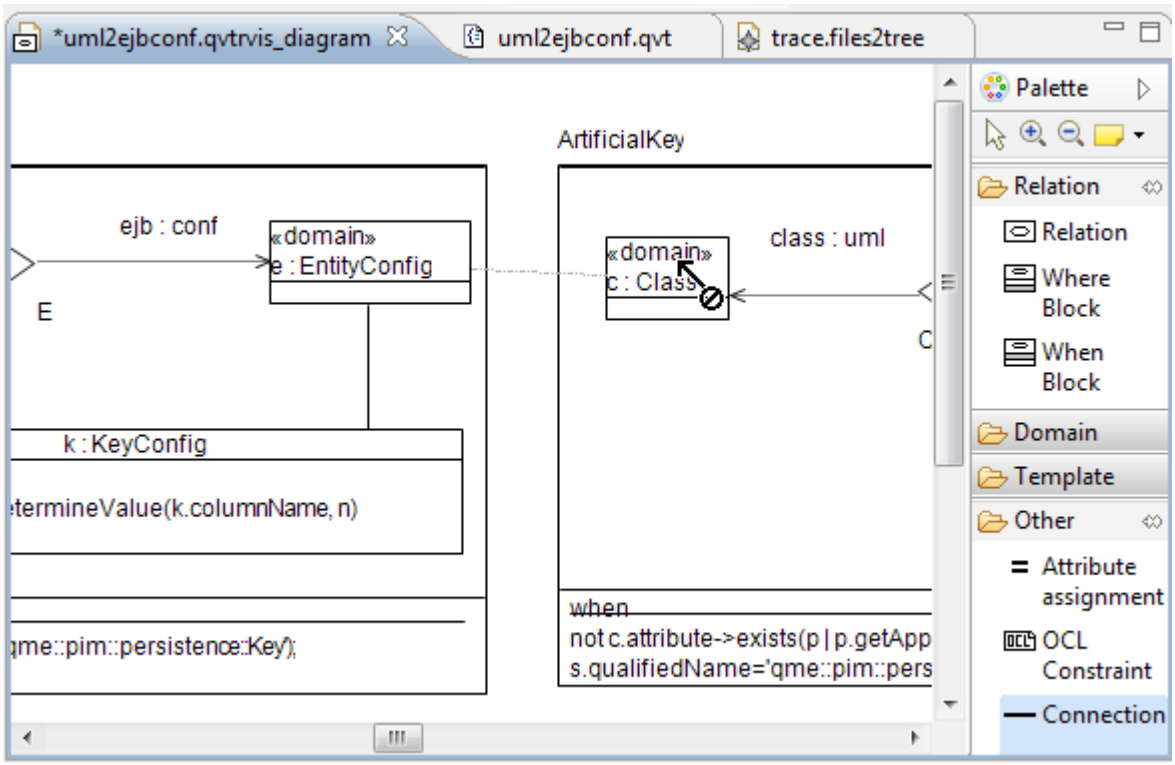


Abbildung 16: Auswirkung des Constraints, der falsche Verbindungen unterbindet

4.1.2 Valide Metamodell Auswahl

Sind an einer Stelle der Transformationsdefinition gewählte Metamodellelemente ungültig, sollte dies im Rahmen der Modellierung verhindert oder mittels Validierung aufgedeckt werden.

In der Lösung wurde ein Ansatz gewählt, bei der die Auswahl von Metamodellelementen bereits so weit eingeschränkt wird, dass nur noch eine gültige Auswahl getroffen werden kann. So muss mittels Constraints nur noch überprüft werden, ob ein Typ bereits gesetzt wurde oder nicht.

Beispiel: Domains muss eine Klasse aus dem Metamodell zugewiesen werden, was durch das Typed Model vorgegeben wird, für das die Domäne gilt. Abbildung 17 zeigt die Definition des OCL-Ausdruckes eines GMF Audit im Mapping Modell, welcher überprüft, ob der Domain ein Metamodellelement zugewiesen wurde. In Abbildung 18 zeigt die Darstellung im graphischen QVT/R-Editor bei Verletzung des Constraints.

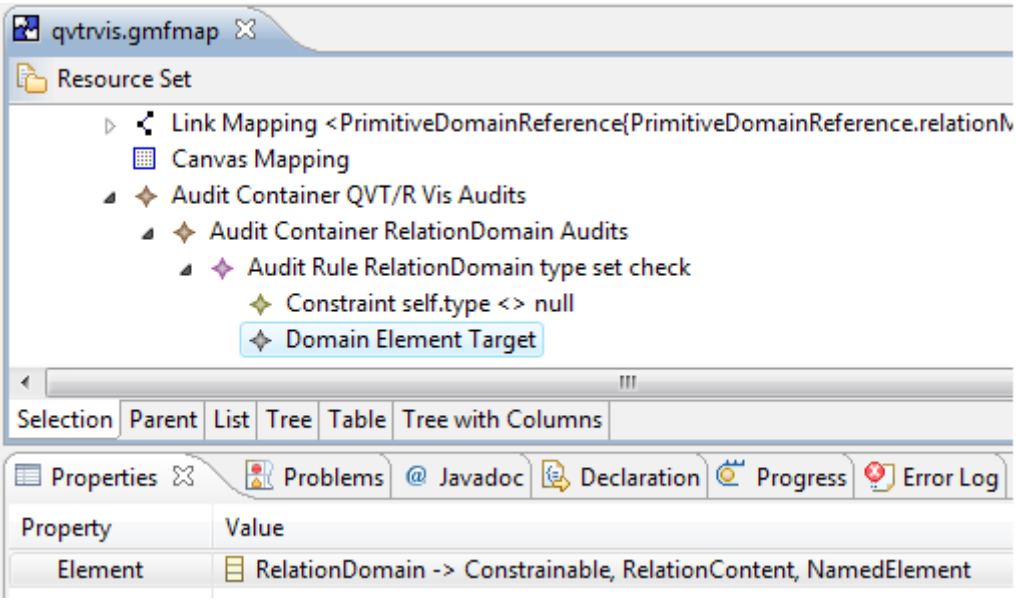


Abbildung 17: Definition der Validierung auf gesetzten Typ einer Domain

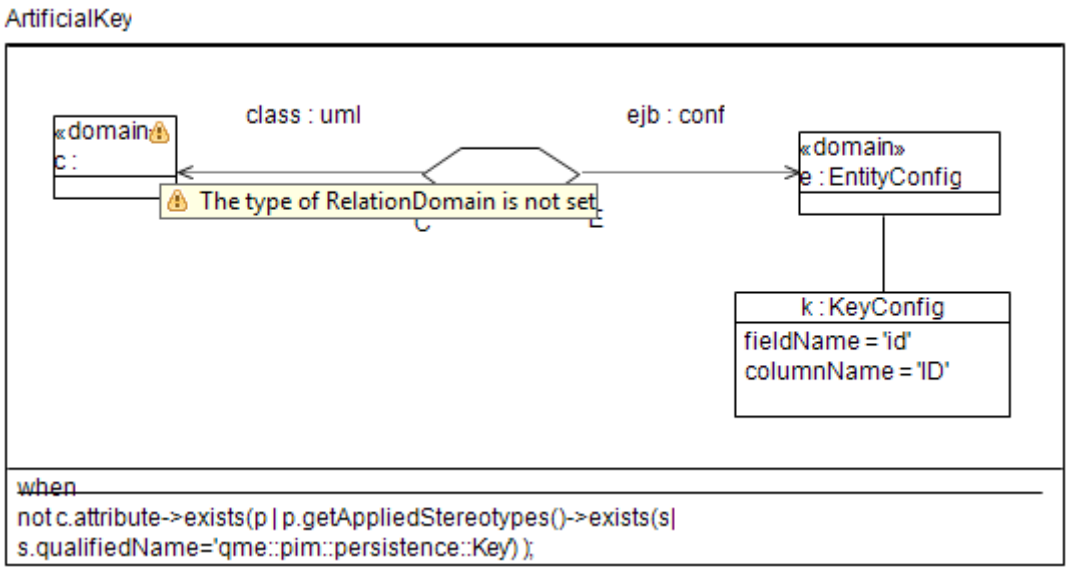


Abbildung 18: Auswirkung der Validierung auf gesetzten Typ von Domains

4.1.3 Textueller Vadidator

Im Metamodell graphischer Editoren sind im Allgemeinen nicht immer alle Aspekte einer Domäne in formaler Form gespeichert. So können die Modelle Zeichenketten für Befehle oder zusammengesetzte Ausdrücke enthalten, die sich nicht ohne weiteres im graphischen Editor validieren lassen. Daher lässt sich ein weiteres Konzept, nämlich das des textuellen Validators definieren. Dabei wird das Modell mittels M2T-Transformation in eine textuelle Repräsentation überführt. Ein Validator für die textuelle Syntax stellt fest, ob Fehler in der textuellen Repräsentation existieren. Im nächsten Schritt wird versucht, die Elemente, auf die der Fehler zutrifft, im Modell zu finden und dem graphischen Editor zu melden.

Wenn aus dem graphischen Modell mittels M2T-Transformation komplexere textuelle Repräsentationen erzeugt werden, kann es sinnvoll sein, Fehler im generierten Quelltext zurück auf das Modell zu projizieren. Dies ist in den Fällen sinnvoll, in denen Fehler im Modell vorhanden sein können, diese aber mittels Modellvalidierung nicht zu erkennen sind. Trotz der komplexeren textuellen Repräsentation muss aber eine eindeutige Abbildung von der Position des Fehlers in textueller Repräsentation auf das entsprechende Modellelement existieren.

Diese Art der Validierung ist vor allem dann interessant, wenn die M2T-Transformation schnell und am besten automatisiert nach dem Abspeichern des Modells erfolgen kann. Dann ist eine schnelle Anzeige der Fehler im Modell möglich, was zu schnellen Turn-Around Zeiten führt.

Beispiel 1: In When- und Where-Blöcken von Relationen werden OCL-Ausdrücke angegeben. Im Modell liegen die Ausdrücke nur in textueller Form. Die Überprüfung wird nicht im Modell,

sondern nach einer M2T-Transformation durch einen QVT/R-Validator durchgeführt, der die textuelle Repräsentation prüft. Das Ergebnis wird wieder im graphischen Editor angezeigt. Abbildung 19 zeigt das Ergebnis einer Validierung durch einen QVT/R-Validator im graphischen Editor. Moniert wird an dieser Stelle ein Tippfehler.

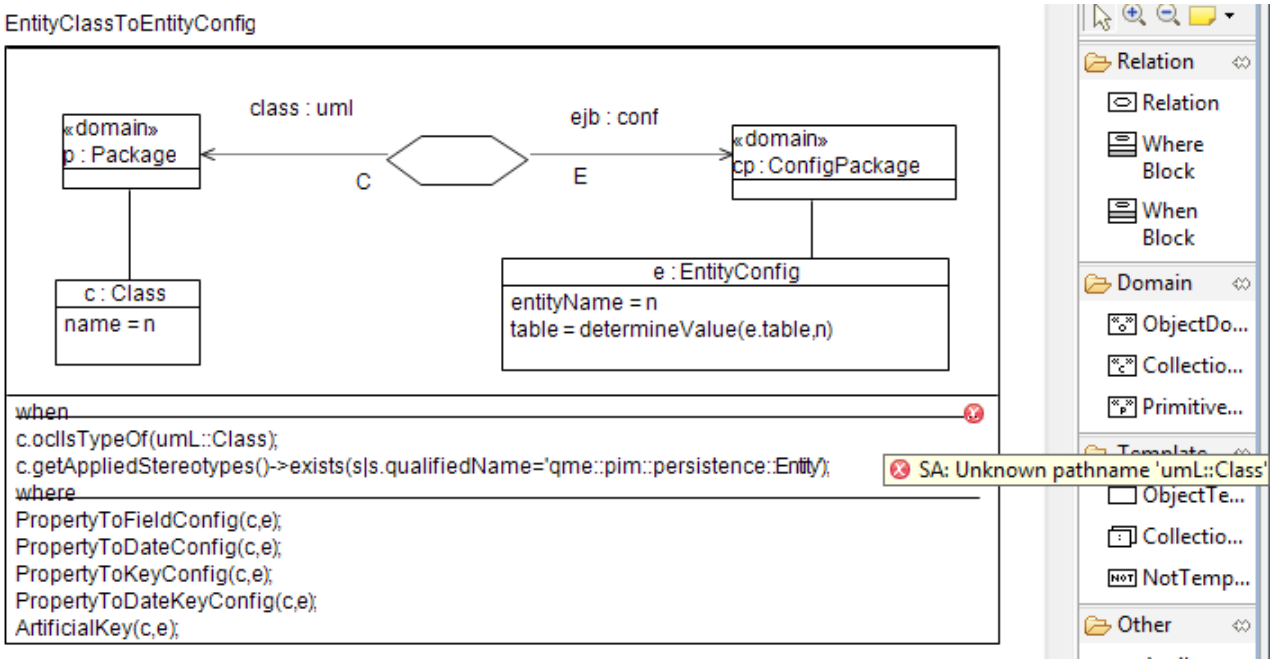


Abbildung 19: Tippfehler, der durch textuellen Validator aufgedeckt wurde

Beispiel 2: Ein anderer typischer Fehler ist das Erstellen von nicht-toplevel Transformationen, die dann vergessen und an keiner Stelle aufgerufen werden. Abbildung 20 zeigt, wie die entsprechende Fehlermeldung im graphischen Editor visualisiert wird.

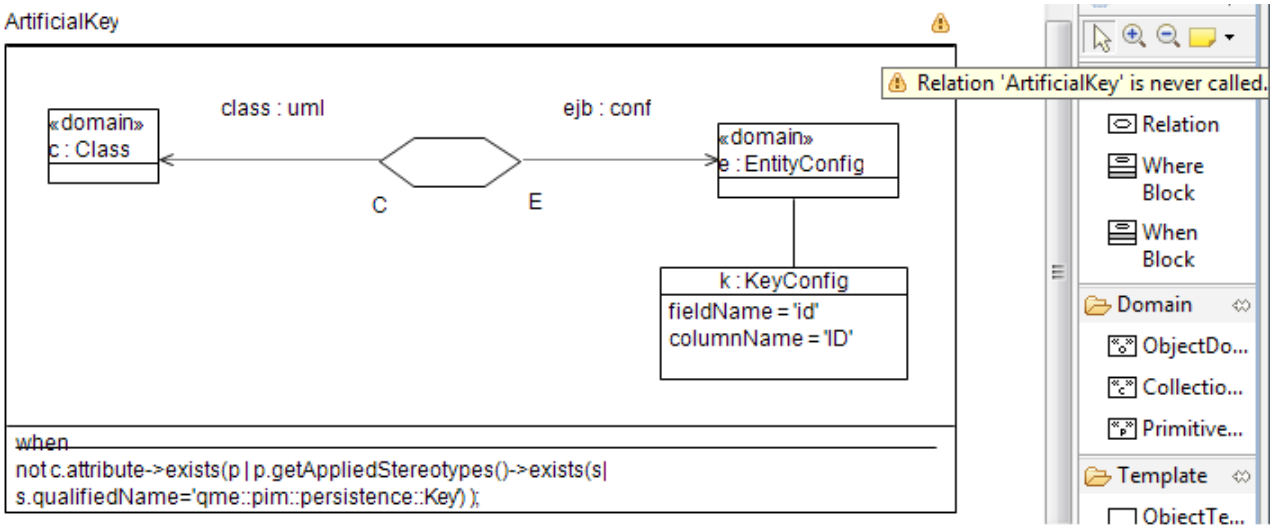


Abbildung 20: Nicht aufgerufene Relation

4.2 Metamodell Registries

Werden Metamodelle, die an der Transformation beteiligt sind, durch einen Bezeichner referenziert

(so wie es bei QVT/R der Fall ist), so muss es einen Abbildungsmechanismus geben, der von Bezeichner auf Metamodell-Pakete abbilden kann. Dieses Konzept wird hier Metamodell-Registry genannt. Für EMF existiert eine Package-Registry, die von URIs auf EPackage Elemente abbilden kann. Da aber durchaus mehrere Registries existieren können, die auch andere Schlüssel-Typen als URIs haben können, ist es sinnvoll eine zentrale Stelle zu schaffen, welche für den Nutzer transparent, mehrere vorhandenen Registries abfragen kann. Auf diese Weise können mehrere existierende Abbildungsmechanismen integriert werden. Im Falle der Implementierung zu dieser Arbeit bedeutet dies die Unterstützung von Abbildungsmechanismen, die durch existierende QVT/R-Engines bereitgestellt werden, so dass Abbildungen nur an einer Stelle definiert werden müssen, um für Engine und Editor genutzt werden zu können. Des Weiteren ist durchaus denkbar, über eine solche Registry-Implementierung andere Metamodell-Technologien zu unterstützen. Die Registry würde faktisch einen Adapter darstellen und könnte intern mittels einer M2M-Transformation implementiert werden, die von einer fremden Metamodell-Technologie zu Ecore-Modellen transformiert.

4.3 Verknüpfung mit Trace-Daten

Der QVT Standard sieht das automatische Erstellen von Trace-Modellen während der Transformation vor. Trace-Modelle sind ein wichtiges Konzept in M2M-Transformationen, denn in Transformationsketten lassen sich Fehler schwerer lokalisieren [Völt05]. Daher muss transparent gemacht werden, welche Regel für welche Ergebnisse verantwortlich ist, um die Ursache von fehlerhaften Transformationsergebnissen leichter feststellen zu können. Aus diesem Grund sollte der Zugriff auf Trace-Daten auch aus visuellen Editoren für Modelltransformationen möglich sein.

Im graphischen Editor, der im Rahmen dieser Diplomarbeit erstellt wurde, ist Zugriff auf Trace Daten über das Kontextmenü der Elemente der Relationen möglich. Zudem kann eine nach Relation gefilterte Ansicht auf Trace Daten aufgerufen werden. Im Kapitel 6.4 wird anschaulich erklärt, wie der Zugriff funktioniert.

5 Realisierung des Visual QVT/R Frameworks

Dieses Kapitel zeigt den Aufbau des Frameworks und beschreibt die technische Umsetzung im Einzelnen. Dabei werden alle relevanten Komponenten einzeln erörtert und es wird auf technische Probleme bei der Umsetzung eingegangen.

5.1 Verwendete Technologien

In den folgenden Unterkapiteln werden die für die Implementierung verwendeten Technologien vorgestellt und die Entscheidungen für die jeweiligen Technologien erläutert.

5.1.1 Eclipse Framework

Das Eclipse Framework bietet nicht nur eine ausgezeichnete Infrastruktur für die modulare Softwareentwicklung, sondern beherbergt im Eclipse Modeling Project auch eine hohe Anzahl an Projekten für die Modellgetriebene Softwareentwicklung.

5.1.1.A OSGi-Bundles / Eclipse Plug-ins

Die Implementierung von Visual QVT/R wurde auf Basis von Eclipse Technologien durchgeführt. Daher kommen die für Eclipse üblichen Konzepte zur Entwicklung von Software-Komponenten zum Tragen. Die Aufteilung in Komponenten wurde mittels des OSGi Bundles realisiert. Schnittstellen wurden über Eclipse Extension Point Schemas und Java-Interfaces definiert. Das führt zu einer losen Kopplung der Komponenten und erlaubt das Austauschen von Komponenten und Erweitern der Funktionalität des Visual QVT/R Frameworks. Ein großer Vorteil von OSGi ist die Versionierung von Komponenten, so dass die gleichen Komponenten in unterschiedlichen Versionen im gleichen Programm genutzt werden können.

5.1.1.B Eclipse-Features

Features sind eine Zusammenfassung von Plug-ins, die eine logische Einheit ergeben. Sie erleichtern die Installation des Frameworks. Für eine Menge von Features kann eine Update-Site erstellt werden. Eine Update-Site ist ein Repository, welches neben den Plug-ins und Features auch die Metadaten der vorhandenen Features und deren Plug-ins sowie deren Abhängigkeiten enthält. Dies vereinfacht die Installation, denn es reicht für die Installation eine URL anzugeben. Eventuelle Abhängigkeiten zu externen Features (welche nicht von der Update-Site bereitgestellt werden) werden bei der Installation aufgelöst und aus anderen Quellen nachgeladen. Im Falle von Visual

QVT/R existieren solche Abhängigkeiten unter anderem zu der GMF Laufzeitumgebung und zum openArchitectureWare Framework.

5.1.1.C Eclipse EMF

Das Eclipse Modeling Framework ist eine der zentralen Projekte des Eclipse Modeling Project. Es stellt eine Technologie zum Instantiieren von Modellen, die durch ein Metamodell beschrieben werden, sowie zum Laden, Abfragen, Modifizieren und Speichern von Modellen. EMF bietet das Serialisieren von Modellen in das XMI-Format, erlaubt aber auch das Speichern in andere Formate. Zur Definition von Metamodellen bietet EMF das Ecore-Metametamodell. Ecore ist isomorph zum OMG Standard EMOF und erlaubt transparent das Lesen und Schreiben von EMOF Metamodellen. EMF stellt auch eine mächtige Introspektions-API zur Verfügung, über die Meta-Informationen von Modellelementen abgerufen werden können. Mittels EMF lassen sich einfache Modell-Editoren generieren, in denen sich Modelle anhand einer Baum-Ansicht bearbeiten lassen. Die Implementierung zu dieser Diplomarbeit basiert auf EMF, denn Metamodelle und Editoren des Frameworks sind mittels bzw. aufbauend auf EMF realisiert.

5.1.1.D Eclipse GMF

Das Graphical Modeling Framework (GMF) ist ein Eclipse Projekt, das genutzt werden kann, um graphische Editoren für Ecore Metamodelle zu erstellen. Ein mit GMF erstellter Editor ermöglicht also das Erstellen von Modellen, die Instanzen dieses Metamodells sind. GMF besteht aus zwei Teilen, der eine Teil umfasst Runtime-Plug-ins, der andere Teil besteht aus einem Generator, der aus Modellen (sieh unten) Diagramm-Editoren generiert, die auf der GMF Runtime aufbauen. Neben dem Metamodell im Ecore-Format benötigt GMF ein Modell, in dem graphische Elemente definiert werden, das „Graphical Def Model“. In diesem Modell werden die Elemente der konkreten graphischen Syntax festgelegt, die im Editor zum Erstellen von Modellen genutzt werden. Um Falle von Visual QVT/R musste hier die graphische Syntax von QVT Relations definiert werden. Außerdem wird ein „Tooling Def Model“ erstellt, in dem einige Aspekte der Benutzeroberfläche beschrieben werden. Dazu gehört vor allem die Palette, über die neue Elemente erzeugt werden können. Das so genannte „Mapping Model“ verknüpft nun alle drei Modelle. Es kann z.B. angegeben werden, welches graphische Element für welches Metamodellelement genutzt werden soll und welcher Eintrag in der Palette dazu genutzt werden kann, um ein neues Exemplar des Metamodellelementes im Editor zu erstellen. In diesem Mapping Modell können auch zusätzlich OCL-Constaints für das Metamodell angegeben werden, welche der generierte Editor überprüfen kann. Diese Beschränkungen können entweder „live“, also sofort bei der Eingabe, oder „on demand“, also nach

manuellem Aufruf aus dem Menü heraus, überprüft werden. Das Mapping Model muss vor der Generierung der Quelltexte und Konfigurationsdateien in ein „Diagram Editor Gen Model“ überführt werden. In diesem Modell können letzte Feinheiten des generierten Editors angegeben werden. Dazu gehören z.B. Dateinamenserweiterungen, eigene Anpassungen am Generierungsprozess über zusätzliche Templates, ob die Modellvalidierung verfügbar sein soll und ob Modell und Diagramm-Daten in der gleichen Datei oder in getrennten Dateien gespeichert werden sollen. Das aus dem „Diagram Editor Gen Model“ erzeugte Projekt ist ein Eclipse Plug-in Projekt; dieses ist ohne manuelle Ergänzungen einsetzbar.

Da GMF derzeit das am weitesten entwickelte Werkzeug zum Erstellen von Diagramm-Modelleditoren auf Basis von Eclipse ist, fiel die Wahl für die Implementierung der Editor-Komponente auf GMF. Die Entwicklung des graphischen Editors mittels GMF,entspricht einem Vorgehen nach der AC-MDSD (siehe Kapitel 1.2), denn in GMF Modellen werden die wichtigsten Konzepte der Domäne von graphischen Editoren festgehalten. Das Generieren des Quelltextes befreit den Entwickler von der Aufgabe, Infrastruktur-Code zu schreiben, der durch ein hohes Maß an struktureller Redundanz gekennzeichnet ist. Damit ist die graphische Editorkomponente des Frameworks selbst mittels Modellgetriebener Technologien erstellt worden, was ein erklärtes Ziel der Arbeit war.

5.1.2 openArchitectureWare Framework

Das openArchitectureWare (oAW) Framework bietet eine Reihe von Technologien zur Modell-getriebenen Softwareentwicklung. Es stellt unter anderem die Modell-zu-Text-Sprache Xpand und die Untersprache Xtend bereit, welche für Modell-zu-Modell-Transformationen gebraucht werden kann. Um die Transformationen zu steuern und um Transformationsketten aufbauen zu können, enthält das Framework eine Workflow Engine. Die hier genannten Komponenten werden mittlerweile innerhalb des Eclipse Modeling Project weiterentwickelt. Xpand wird von Visual QVT/R benötigt, um das visuelle Modell in die konkrete textuelle Syntax zu transformieren. Xtend wird gebraucht, um das mittels des Medini Parsers eingelesene Modell in das visuelle Modell zu transformieren. Die Sprache Check bietet die Möglichkteit Validitätsprüfungen zu definieren. Es existiert eine einfache Anbindung von Check an EMF- und GMF-Editoren. Ein Eclipse Plug-in, welches ein Check Skript bereitstellen will, muss lediglich einen bestimmten Extension Point anbieten, der besagt, für welchen Ecore Namensraum das Skript gilt, und unter welchen Pfad das Skript im Plug-in zu finden ist. Check wird vom Visual QV/R genutzt, um die Validität des graphischen Modells zu überprüfen. Xpand, Xtend und Check haben als Basis eine gemeinsame, OCL ähnliche Abfragesprache.

5.1.3 Medini QVT

Die Engine für QVT Relations wurde bereits in Kapitel 1.4.2.E.i vorgestellt. Visual QVT/R nutzt den Parser der Engine zum Einlesen von QVT/R-Quelltexten, die dann zu visuellen Modellen transformiert werden können und zur Validierung und Formatierung der Ausgabe der Modell-zu-Text-Transformationen, die das visuelle QVT/R-Modell in die textuelle Syntax überführt. Um die bei der Validierung gefundenen Probleme im visuellen Editor darstellen zu können, waren einige Modifikationen an der Engine nötig. In Kapitel 5.4.4 wird genauer darauf eingegangen.

5.2 Vorüberlegungen zur Implementierung

In diesem Abschnitt werden Erwägungen aufgezeigt, die zu grundlegenden Entscheidungen für die Implementierung geführt haben.

5.2.1 QVT Relations Metamodell

Zunächst stellte sich die Frage nach dem zu benutzenden QVT/R-Metamodell. Auf den Webseiten der OMG steht ein QVT Relations Metamodell im Ecore-Format bereit. Eine Unterstützung dieses Modells hätte den Reiz gehabt, dass das Framework die Interoperabilitätsstufe „XMI Exportable“, entsprechend dem QVT Standard (Seite 1), erreicht hätte und damit ohne Umwege über die konkrete textuelle Syntax mit QVT Engines zusammenarbeiten könnte, die die Interoperabilitätsstufe „XMI Executable“ implementieren. Das Metamodell der OMG basiert allerdings auf dem EMOF Typsystem. Es werden also EMOF Metamodellelemente referenziert, zwischen denen transformiert werden soll. Die Implementierung des Frameworks sollte allerdings mit Ecore Metamodellen arbeiten. Da Ecore standardkonformes EMOF einlesen kann, ergibt sich für den Benutzer keine Einschränkung aus der internen Nutzung von EMF. Eine Alternative wäre das Metamodell des Eclipse QVT Declarative Projektes gewesen, das sowohl eine EMOF, als auch eine EMF basierte Version des standardisierten Metamodells bereitstellt. Allerdings stellte sich heraus, dass GMF mit den Strukturen des Metamodells nicht umgehen konnte. So existieren in der graphischen Syntax Elemente, die keine Entsprechung in der abstrakten Syntax haben (siehe 1.4.2.B.ii und 1.4.2.B.vi). Solche Elemente können nicht mittels GMF erzeugt werden. In der graphischen Syntax steht zudem das Domain Element für zwei Elemente in der abstrakten Syntax, nämlich das RelationDomain und das TemplateExp Metamodellelement, was sich ebenfalls nicht mit GMF umsetzten ließ. Ein weiteres Ausschlusskriterium war die Struktur des Metamodells. Um mittels GMF ein Element zu definieren, welches ein anderes Element enthält (das äußere Element bietet ein Compartment für Unterelemente), muss das Unterelement über eine Komposition vom Oberelement erreichbar sein.

Da die graphische Repräsentation einer Relation alle Templates enthält, die von den Domains referenziert werde, müssten also alle Templates über eine Komposition vom Metamodellelement der Relation aus erreichbar sein. Dies ist aber im standardisierten Metamodell nicht der Fall. Zudem hätte für den graphischer Editor eine sehr komplizierte Synchronisation zwischen Modell und Repräsentation im Editor implementiert werden müssen, da OCL-Ausdrücke im Modell als Modellelemente erscheinen, im Editor jedoch in textueller Repräsentation zu sehen und einzugeben sein müssen. Im Grunde hätte diese die feste Einbindung eines QVT-Parsers in den Editor bedeutet. Aus diesen technischen Gründen wurde ein eigenes Metamodell geschaffen, welches die Voraussetzung für ein sinnvolles Arbeiten mit GMF schuf. Es mussten zwar im eigenen Metamodell einige abgeleitete Attribute per Hand in den generierten Java Quelltext implementiert werden, um alle Teile der graphischen Syntax mit GMF korrekt zu unterstützen; doch der Aufwand hielt sich in Grenzen. Die Instanzen dieses Metamodells werden der Einfachheit halber in dieser Arbeit als visuelle Modelle bezeichnet, auch wenn die Modelle selbst keine Information über die Darstellung in Diagrammen in sich tragen. Da das Framework standardkonformen QVT/R-Quelltext erzeugt, erreicht es immerhin die im QVT Standard definierte Interoperabilitätsstufe „Syntax Exportable“. Abbildung 21 zeigt einen Ausschnitt aus dem eigens erstellten Metamodell und Abbildung 22 ein Modell in der graphischen Syntax, dessen Elemente entsprechend des Metamodells gefärbt wurde. Wie beschrieben müssen alle Elemente die sich in Compartments eines anderen Elementes befinden über Komposition mit dem äußeren Element verbunden sein. Das gesamte Metamodell ist in Anhang A zu finden.

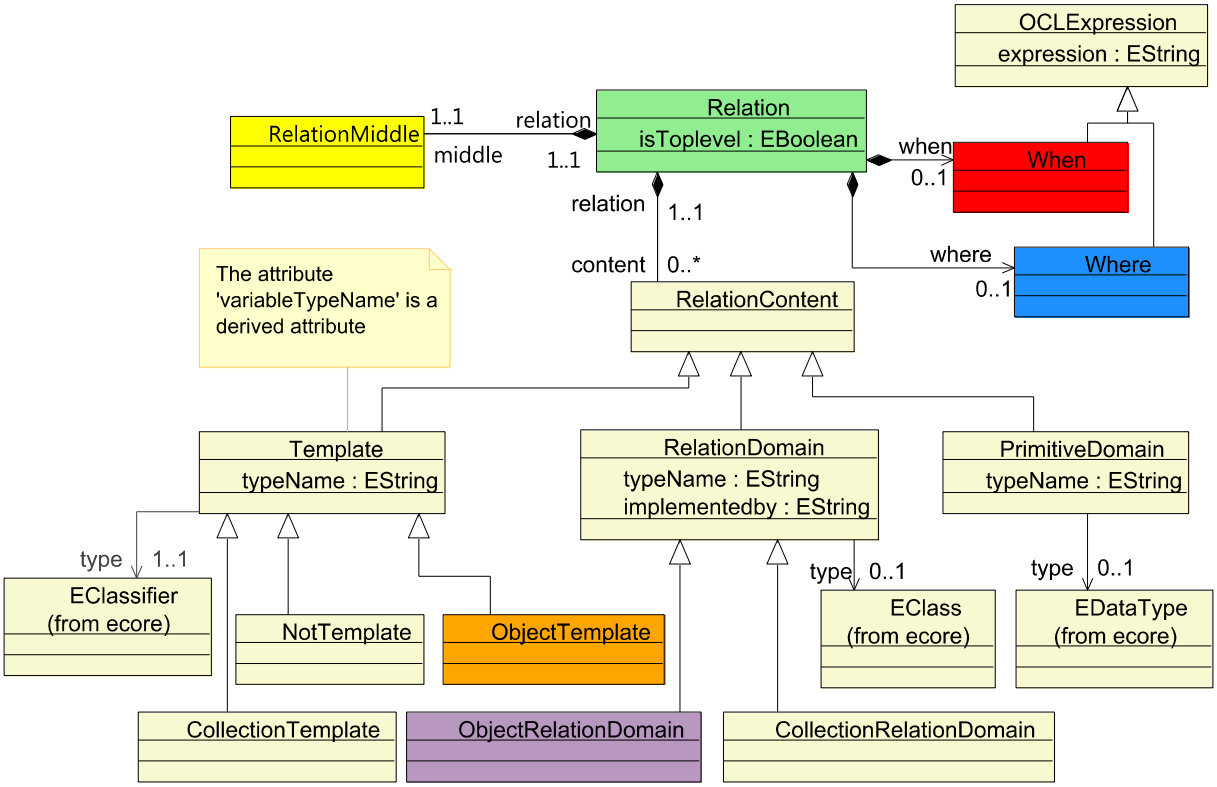


Abbildung 21: Metamodell Ausschnitt für Relation

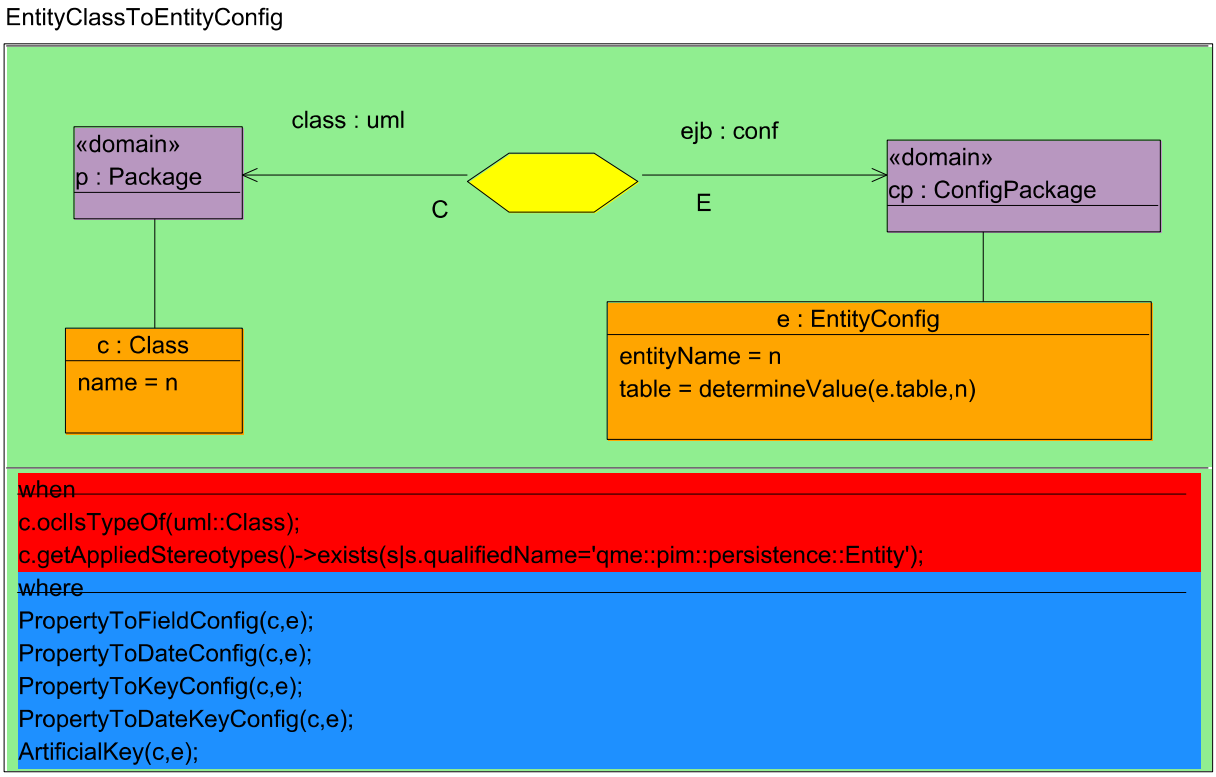


Abbildung 22: Beispiel Relation entsprechend dem Metamodell gefärbt

5.2.2 Synchronisierung zwischen textueller und graphischer Syntax

Für die Text-zu-Modell Transformation galt es, die Entscheidung zu treffen, entweder einen bestehenden QVT/R-Parser zu nutzen oder mittels einer Technologie wie Xtext, TCS oder EMFText einen eigenen Parser, auf Basis der im QVT Standard definierten konkreten textuellen Syntax, zu entwickeln. Der Aufwand einen eigenen Parser zu erstellen, erschien allerdings, trotz der Hilfe durch modellgetriebene Technologien, recht hoch. Zudem hätte es eine Abhängigkeit zu einer zusätzlichen Technologie bedeutet und da bereits Parser für QVT/R existierten, erschien die Notwendigkeit nicht gegeben, einen weiteren Parser zu schreiben. Zu Anfang der Entwicklung war Medini QVT die einzige funktionierende Open Source Engine, für die eine gute Eclipse Anbindung existiert. Daher fiel die Wahl auf die Nutzung von Medini QVT. Später in der Entwicklung traten allerdings einige Probleme mit Medini QVT auf, die eine Modifikation der Medini-Engine nötig machten. Diese Probleme sind in Kapitel 5.4.4 dokumentiert.

Grundsätzliche Probleme existieren bei der Synchronisation zwischen graphischer und textueller Syntax, denn das Not Template hat keine Entsprechung in textuellen Syntax (vgl. Kapitel 1.4.2.B.vi). Es handelt sich bei dem Not Template um ein abstrakteres Konstrukt, welches bei der Umwandlung in die konkrete textuelle Syntax in einigen Fällen in mehrere andere Sprachkonstrukte umgesetzt werden muss. Not Templates sind daher nicht roundtrip-geeignet. Abgesehen von diesem Problem war eine Abbildung von visuellem Modell auf textuelle Konstrukte mittels M2T-Transformation möglich. Bei der M2T-Transformationen werden keine geschützten Bereiche benötigt, da im graphischen Editor alle QVT/R-Sprachelemente unterstützt werden, auch diejenigen, die nicht in der konkreten visuellen Syntax auftreten. Damit muss kein Code in der textuellen Repräsentation ergänzt werden.

5.2.3 Konfiguration des Visual QVT/R Frameworks

Sind visuelle Modelle und textuelle Repräsentation synchron zu halten, so soll gespeichert werden, zwischen welchen Textdateien und Modellen zu transformieren ist. Zu diesem Zweck musste ein Konfigurationsmetamodell (siehe Kapitel 5.4.1.B) entworfen werden, dessen Instanzen pro Projekt angelegt werden sollen. Dieses hat jedoch transparent hinter der Kern API zu geschehen. Der Nutzer des API soll Konfigurationen für Dateien anfordern können, ohne sich um Konfigurationsdateien kümmern zu müssen. Zusätzlich können in diesen Konfigurationsmodellen weitere Abbildungen von Metamodell-Namen auf Metamodelle angelegt werden. Die Abbildungen können immer unabhängig von Metamodel-Registries angelegt werden. Dies ermöglicht das Erstellen von QVT/R-Transformationen mit dem graphischen Editor, ohne dass dieser eine Abhängigkeit zu einer

QVT/R-Engine benötigt.

5.2.4 Trace Datenzugriff und Editor

Der QVT Standard beschreibt leider nicht eindeutig, wie die Metamodelle zu Transformations-Tracemodellen aufgebaut sein sollen. Einzig bei der Spezifikation der Transformation von QVT/R nach QVT/C wird auf die Struktur von Trace-Modellen eingegangen. Dies ist an dieser Stelle nötig, da in QVT/C Traces explizit erzeugt werden und daher Trace-Metamodelle referenziert werden müssen. Auf Seite 150 des Standards wird kurz auf die Generierung von Trace Klassen für die Nutzung in QVT/C erläutert. Dort wird vorgeschrieben, für jede Relation eine Trace Klasse zu erstellen, die für jede in den den Patterns der Domänen auftretende „object node“, ein Attribut enthalten soll. Dies ist allerdings eine sehr vage Beschreibung und sagt nichts über die Benennung von Klassen und Attributen aus. Zudem gilt diese Regel streng genommen nur für Trace Klassen, die in QVT/C Transformationen genutzt werden, welche aus QVT/R-Transformationen generiert wurden. Daher sind Trace Daten faktisch abhängig von der Implementierung der QVT/R-Engine. Der Trace-Datenzugriff und das Öffnen des Trace Editors muss hinter einer Schnittstelle verborgen werden, soll die Abhängigkeit von einer bestimmten Engine vermieden werden. Das mit dem Visual QVT/R Framework mitgelieferte Plug-in für Trace-Datenzugriff ist auf Trace-Modelle zugeschnitten, die von der Medini QVT Engine erstellt wurden. Auf die Implementierung wird in Kapitel 5.4.4.E genauer eingegangen. Ein eigener GMF Editor für Traces ist nicht möglich, da die Metamodelle dynamisch für jede einzelne Transformation erzeugt werden. Es wäre möglich für jede Transformation einen eigenen GMF Editor für Traces zu schreiben oder zu generieren, was allerdings einen sehr hohen technischen Aufwand bedeuten würde und unhandlich anzuwenden wäre.

5.2.5 Wahl der Validierungs-Technologien

Die Technologien GMF und openArchitectureWare, die für die Implementierung zur Verfügung stehen, bieten die Möglichkeit Constraints zum Zwecke der Modellvalidierung einzusetzen. Für die Überprüfung von strukturellen Aspekten wurde die Sprache Check des oAW Frameworks gewählt. Da sich mit seinem pragmatischen OCL-Dialekt komplexe Zusammenhänge prägnant definieren lassen, erschien die Sprache für diese Aufgabe am besten geeignet. Einzige Ausnahme bilden GMF-Audits, die das Setzen von fehlerhaften Referenzen bereits im Voraus unterbinden können. GMF-Audits werden mittels standardkonformer OCL definiert und im GMF Mapping Model hinterlegt. Mittels Check könnten nur bereits bestehende Verbindungen im Nachhinein als fehlerhaft markiert werden. Da die Regeln der Validierung mittels Check in einer eigenen Datei separat abgelegt wer-

den, ist bei Check dafür eine bessere Trennung der Aufgabengebiete gegeben. Für die Überprüfung auf gesetzte Referenzen der Transformationselemente zu Metamodellelementen wurde auf die Audit-Technologie des GMF-Framework zurückgegriffen, denn Check hatte keinen Zugriff auf die referenzierten Metamodellelemente. Eine sich aus der Entwicklung des eigenen QVT/R-Metamodells ergebende Schwierigkeit für den graphischen Editor ergab ist, dass OCL-Ausdrücke im Modell nur als Text hinterlegt sind und dass dadurch keine einfache Validierung der Ausdrücke möglich ist. Dies entspricht dem allgemeinen Konzept, welches in Kapitel 4.1.3 vorgestellt wurde. Um auch solche Probleme bei der Validierung zu melden, wurde ein einfaches Problem-Modell eingeführt, welches Probleme für Modellelemente im graphischen Modell beschreibt (siehe 5.4.1.C). Nach der M2T-Transformation können Code-Validatoren Probleme in ein solches Modell schreiben. Um die Modellvalidierung, die das Problem Modell berücksichtigt, in einem graphischen Editor nach einer Codevalidierung automatisch anzustoßen, kann ein solcher Editor den Extension Point `fuberlin.qvtr.codevalidator.listener` anbieten.

5.3 Architektur des Visual QVT/R Frameworks

Dieser Abschnitt geht auf die gewählte Struktur des Frameworks ein und erläutert die Funktionen, die die groben Funktionseinheiten bereitstellen. Die Architektur des Framework ist stark geprägt durch die Technologien, auf denen die Implementierung beruht. Die Modularisierung basiert auf den bereits vorgestellten Eclipse Technologien. Die Grobstruktur wird technisch durch Eclipse-Features realisiert, die einzelnen Komponenten werden durch Eclipse Plug-ins (also OSGi-Bundles) realisiert. Die technische Umsetzung der Komponenten und die daraus entstandene Problem werden in Kapitel 5.4 erörtert. Abbildung 24 zeigt, über welche Extension Points die Komponenten des Frameworks miteinander verknüpft sind. Dabei sind die Plug-ins, die Extension Points bereitstellen, auf der rechten Seite platziert und Plug-ins, welche diese Extension Points nutzen, auf der linken Seite angeordnet. Einzig die Komponente `fuberlin.qvtr.vis.diagram` nutzt einen Extension Point und bietet selbst einen an.

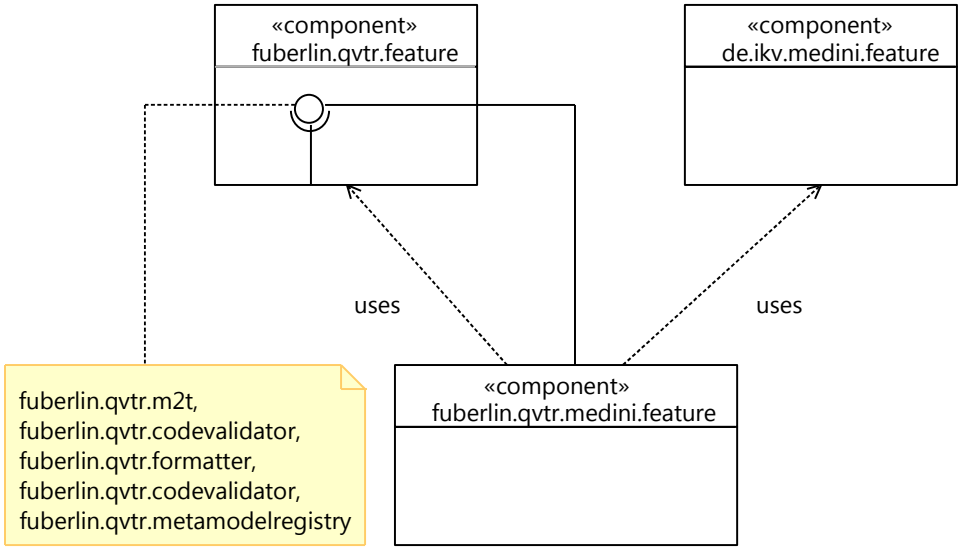


Abbildung 23: Abhängigkeiten zwischen den Features

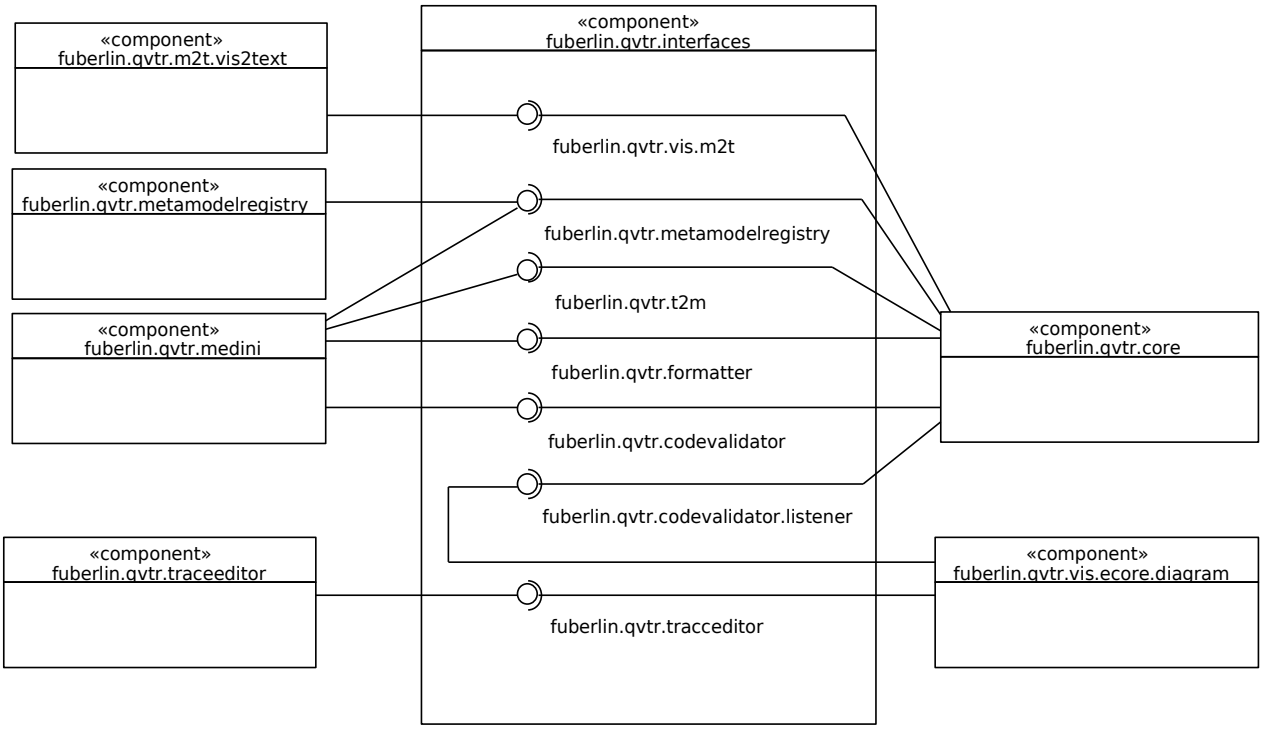


Abbildung 24: Lose Kopplung über Extension Points

5.3.1 Grobstruktur

In den Folgenden Abschnitten wird kurz auf die Eclipse-Features eingegangen, in die das Framework aufgeteilt ist. Implementierungsdetails können im Kapitel 5.4 nachgelesen werden.

5.3.1.A Visual QVT/R Basis Funktionalität

Das Feature „Visual QVT/R Feature“ mit der ID `fuberlin.qvtr.feature` enthält die Kern-Komponenten des Visual QVT/R Frameworks. Dazu gehört die Bereitstellung von Schnittstellenbeschreibungen für die lose Kopplung zwischen den Komponenten und Bereitstellung von einer konsolidierten Kern-API (Facade-Pattern) für den einfachen Zugriff auf die Basis-Funktionalität. Die Aufteilung des Frameworks ist über die genannten Komponenten-Schnittstellen geregelt. Damit lassen sich sowohl einzelne Komponenten des Frameworks austauschen als auch das Framework um zusätzliche Funktionalität erweitern. Auch die Anbindung der Kernfunktionen an den Eclipse-Workbench wird in diesem Feature geregelt. Zuletzt bietet dieses Feature auch den GMF-Diagramm Editor, mit dem sich die graphischen QVT/R-Modelle erstellen lassen.

5.3.1.B Medini QVT

Das „Medini QVT Feature“ (`de.ikv.medini.feature`) stellt die Plug-ins der Medini-Engine und die Medini Cockpit Plug-ins bereit. Da Medini QVT keine eigene Update-Site hat, werden die Plug-ins in diesem Feature mit dem Visual QVT/R Framework ausgeliefert.

5.3.1.C Funktionalität auf Basis von Medini QVT

Alle Komponenten des Visual QVT/R Frameworks, welche eine Abhängigkeit zur Medini-Engine oder einem der Medini Cockpit Plug-ins (und damit zum Medini QVT Feature) haben, sind Teil des „Visual QVT/R Medini Feature“ (`fuberlin.qvtr.medini.feature`). Da für einige Features des Visual QVT/R Framework die Medini QVT Engine modifiziert werden musste, wird die modifizierte Version der Engine ebenfalls Teil des Frameworks. Hier zahlt sich die Bildung von Komponenten mittels OSGi aus, denn die Medini Cockpit Plug-ins beinhalten die nicht modifizierte Version der Engine. Aufgrund der Möglichkeiten von OSGi zur Versionierung von exportierten und importierten Komponenten-Paketen können die beiden Versionen innerhalb der gleichen Eclipse Instanz ohne Probleme koexistieren. Dazu gehört die Bereitstellung einer Komponente für die Text-zu-Modell-Umwandlung um Transformationen von der texutellen in die graphische Syntax zu überführen. Auch für die Evaluation von QVT/R-Quelltexten und das Schreiben in eine Issues-Datei für die Auswertung durch einen Validator steht eine Komponente bereit. Da Trace-Modelle nicht eindeutig standardisiert sind, muss auch die Auswertung der (Medini spezifischen) Trace Dateien über einer Komponente im Medini QVT Feature bereitgestellt werden.

5.3.2 Komplexe Abläufe

Um einen Überblick über die Abläufe des Frameworks zu geben, wird in den folgenden Unterabschnitten der Ablauf Text-zu-Modell Transformation und der Modell-zu-Text-Transformation aus zwei Blickwinkeln betrachtet und näher erläutert. Dabei werden die Aktivitäten in logischer Reihenfolge notiert. Es wird keine Aussage darüber getroffen, welche Komponente welche Aktivität letztendlich anstößt.

5.3.2.A Transformation der textuellen Syntax in ein visuelles Modell

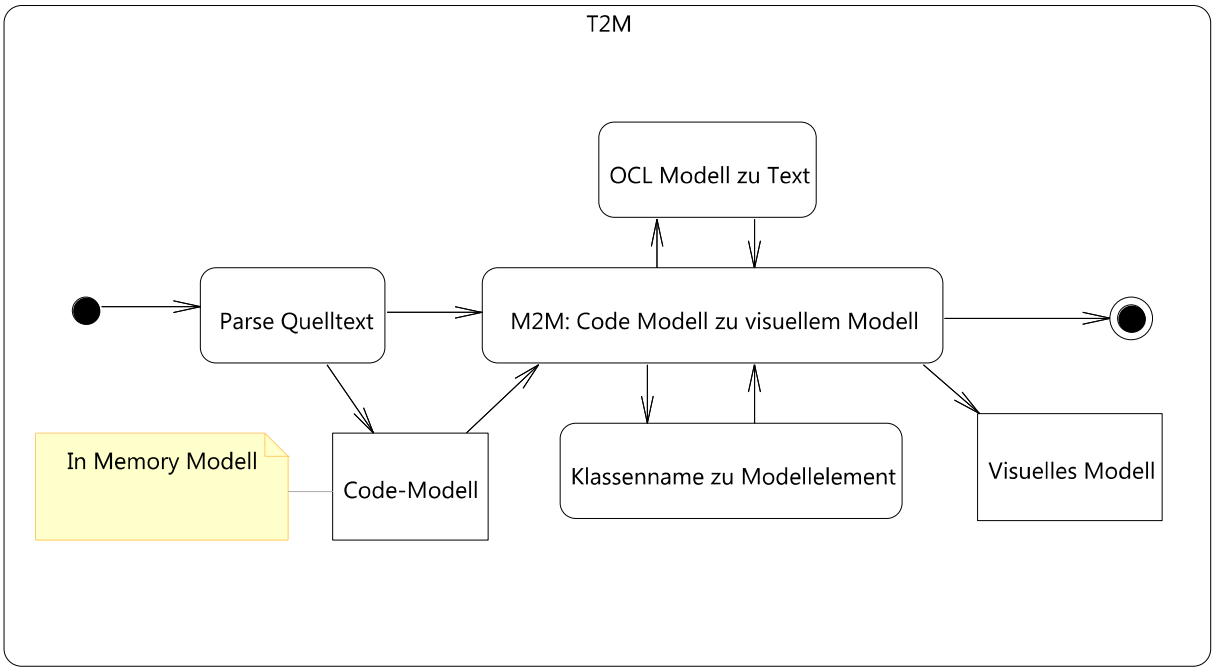


Abbildung 25: T2M Transformation im Detail

Abbildung 25 zeigt den Ablauf der Transformation von Quelltext zum visuellen Modell. Genauer zeigt die Abbildung den Ablauf der Implementierung der Transformation von konkreter textueller QVT/R-Syntax zum visuellen QVT/R-Modell. Das Framework erlaubt auch zusätzliche Transformationen beliebiger anderer Quelltexte zum visuellen QVT/R-Modell. Die Abläufe für andere Implementierungen dürften dem Ablauf dieser Implementierung ähneln.

Das Parsen des Quelltextes erzeugt ein formales Modell, das aber noch keine Referenzen auf die Metamodellelemente, zwischen denen transformiert werden soll, enthält. Im nächsten Schritt wird dieses Modell in ein visuelles Modell überführt. Da in diesem Modell die OCL-Ausdrücke in textueller Form hinterlegt sind, müssen die geparsten OCL-Ausdrücke wieder serialisiert werden. Um im visuellen Modell die Metamodellelemente referenzieren zu können, muss an Stellen, an denen im Code-Modell ein Metamodellbezeichner steht, in den Metamodell-Registries

nachgeschlagen werden, für welches Metamodellelement dieser Bezeichner steht.

5.3.2.B Überführung des graphischen Modells in die textuelle Repräsentation

Um ein graphisches Modell in die textuelle Repräsentation zu überführen, wird nicht nur die Modell-zu-Text-Transformation durchgeführt, sondern zusätzlich der generierte Code validiert und formatiert. Aufgrund der Modularität des Frameworks können weitere M2T-Transformatoren, Code-Validatoren und Code-Formatierer registriert werden.

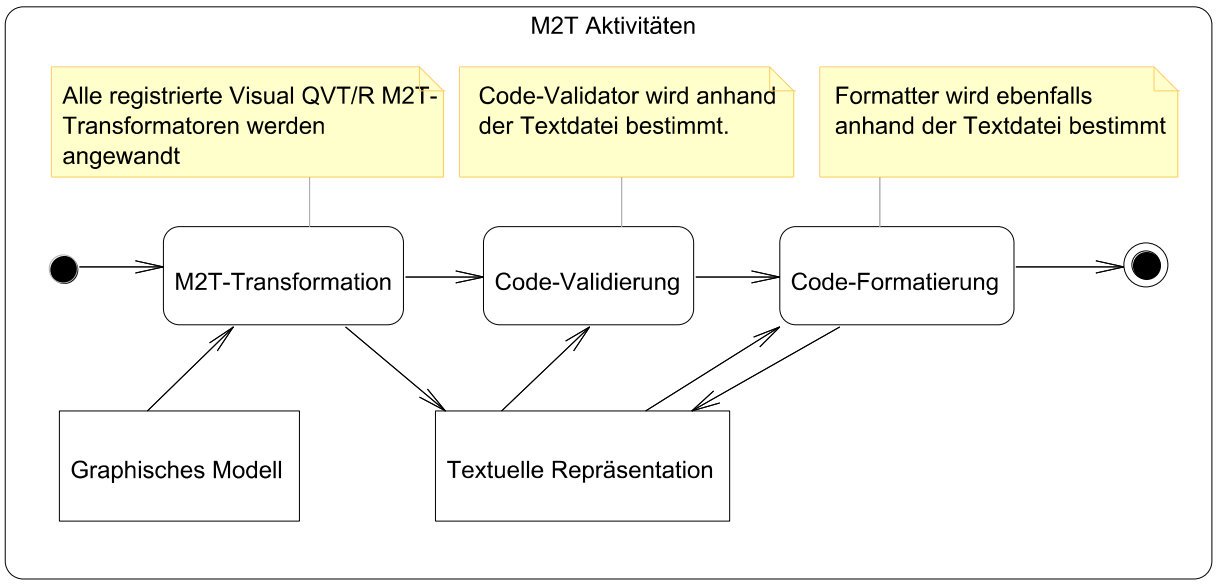


Abbildung 26: M2T Transformation im Detail

5.3.2.C Validierungsprozess im Überblick

Um alle Aktivitäten aufzuzeigen, die letztendlich zum Validierungsergebnis führen, wird an dieser Stelle noch einmal ein Überblick über die Abläufe des Validierungsprozess gegeben. Abbildung 27 zeigt den logischen Ablauf der Aktionen, die zum Ergebnis der Validierung führen. Ob die M2T-Transformation automatisch oder manuell gestartet wird, spielt für die logische Reihenfolge keine Rolle. Dies kann der Nutzer selbst festlegen, indem der entsprechenden Builder aktiviert oder nicht. Der Codevalidator versucht Fehler, die er im generierten Quelltext gefunden hat, im visuellen Modell zu lokalisieren. Die Fehlermeldungen werden dann jeweils mit einer Referenz auf das entsprechende Element im graphischen Modell in das Problemmodell gespeichert. Um die Probleme im graphischen Editor darzustellen, wird daraufhin eine Modellvalidierung angestoßen.

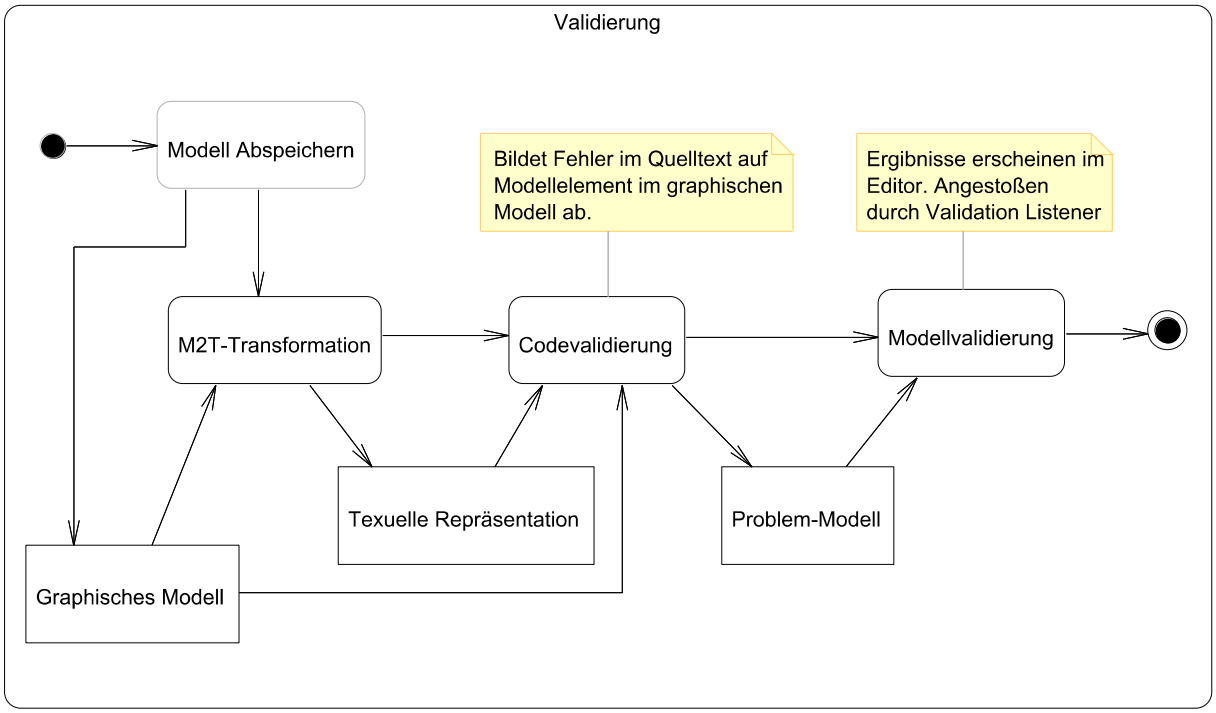


Abbildung 27: Aktivitäten zur Validierung

5.4 Technische Umsetzung

In diesem Abschnitt wird im Detail auf die Einzelheiten der technische Umsetzung der Komponenten, sowie auf die Hürden während der Entwicklung und deren Überwindung eingegangen.

5.4.1 Metamodelle

Im Folgenden werden die für das Visual QVT/R Framework erstellten Metamodelle vorgestellt.

5.4.1.A Das VQVT/R-Metamodell

Das QVT/R-Metamodell wurde für das visuelle erstellen von QVT/R-Transformationen entwickelt. Das Metamodell wurde von Grund auf neu entwickelt, um auf dessen Basis einen GMF Editor erstellen zu können. Die folgenden Unterschiede zum standardisierten Metamodell wurden ausgewählt, um einen Eindruck zu geben, wie weit die Unterschiede zwischen den Metamodellen reichen:

- OCL-Ausdrücke sind im graphischen Modell im Textformat gespeichert. Im standardisierten Metamodell sind die Ausdrücke formal als Modellelemente gespeichert.
- Mehrere sich referenzierende Klassen im standardisierten Metamodell, die in der graphischen Syntax allerdings als ein einziges Element auftreten, sind im Metamodell für die graphische Syntax als einzelne Klasse definiert. So referenzieren z.B. Domains im

standardisierten Metamodell Templates, welche Property Template Items beinhalten. In der graphischen Syntax hingegen enthalten Domains direkt Property Template Items.

- Es mussten Modellelemente für Relation Middle und Not Template eingeführt werden, da diese in der abstrakten QVT/R-Syntax nicht vorhanden sind.

Zusätzlich enthält das Metamodell einige abgeleitete Attribute, die für die Darstellung mittels GMF benötigt werden. So enthält das Modellelement das abgeleitete Attribut „modelName“, welches den Namen des referenzierten TypedModels (Transformationsparameters) ermittelt.

Um mittels Editoren, die das Editieren von VQVT/R-Modellen ermöglichen, valide Modelle zu erstellen, wurde das ursprünglich von EMF generierte Plug-in `fuberlin.qvtr.vis.ecore.edit` so erweitert, dass für die meisten Attribute bereits eine Auswahl von Werten zur Verfügung steht, die nur eine valide Belegung des Attributwertes erlauben.

Diagramme mit allen Elementen des Metamodells befinden sich in Anhang A.

5.4.1.B Das Metamodell für die Konfiguration

Das Metamodell hält fest, welche Dateien in textueller Syntax (CodeFile) und welche graphischen Modelle (QVTRVisModel) synchron gehalten werden sollen. Zusätzlich können Abbildungen von Namen auf Metamodell-Definitionen angegeben werden. Diese können auch hierarchisch definiert werden. Eine Abbildung, die für ein Verzeichnis definiert wurde, gilt demnach für alle enthaltenen Dateien. Verzeichnisinhalte können diese Abbildung mit eigenen Konfigurationen überschreiben.

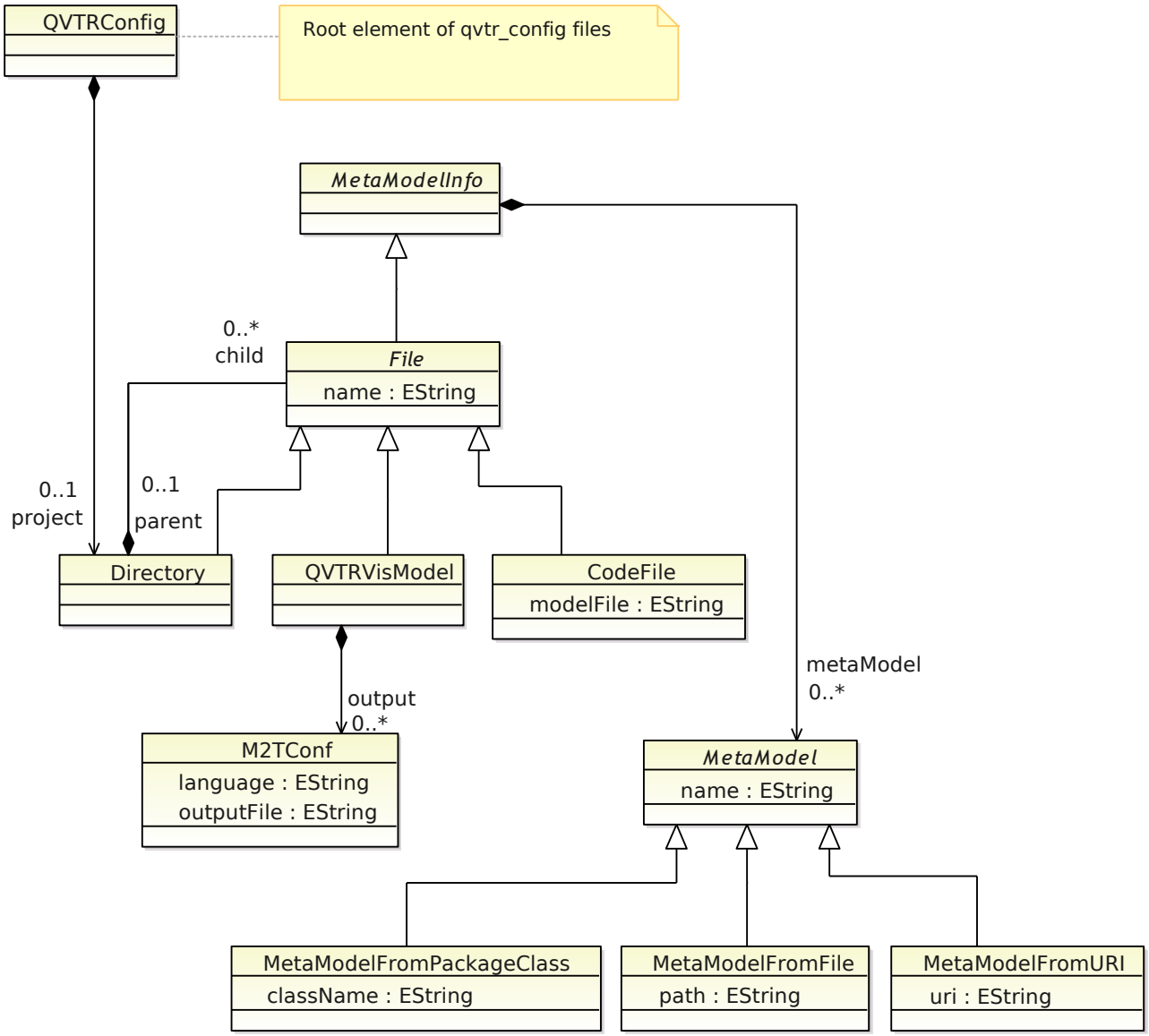


Abbildung 28: Konfigurations-Metamodell

5.4.1.C Das Problem Metamodell

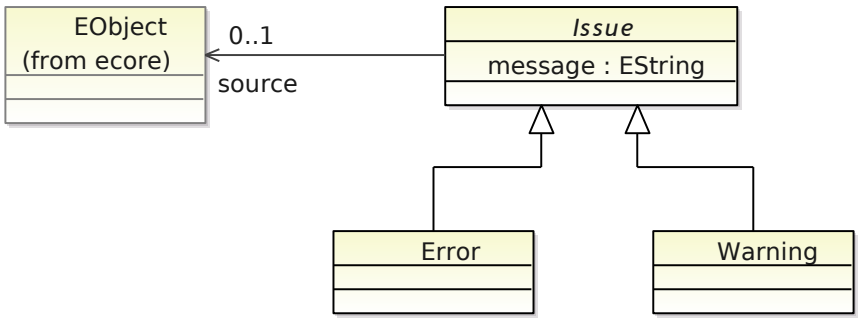


Abbildung 29: Problem Metamodell

Das Plug-in `fuberlin.qvtr.issues.ecore` beinhaltet ein Problem-Metamodell, welches Warnungen und Fehler definiert. Probleme können Modellelemente referenzieren, für die das Problem gilt.

Dieses Metamodell wird genutzt, um Fehler, die von einer QVT/R-Engine oder einem Code-Validator gemeldet werden, im graphischen Modell anzuzeigen.

5.4.2 Kernfunktionalität des Frameworks

Das Plug-in `fuberlin.qvtr.interfaces` ist eine der wichtigsten Komponenten des Frameworks. Es enthält die Extension Point Schemata und Java Schnittstellen, welche zusammen für eine lose Kopplung zwischen den Komponenten des Frameworks sorgen. Einige der Extension Point Schemata können auch von Plug-ins implementiert werden, um sich in das Framework einzuhängen und zusätzliche Funktionalität einzubringen. Zudem werden in dieser Komponente einige globale Bezeichner definiert, die in vielen Komponenten benötigt werden. Dazu gehört z.B. die Dateiendung für Dateien, die visuelle Modelle enthalten.

Die eigentliche Kern-Komponente des Visual QVT/R Frameworks stellt das Plug-in `fuberlin.qvr.core` dar. Über dieses Plug-in können die Funktionen zur Synchronisation zwischen visuellem Modell und textueller Syntax aufgerufen werden, ohne dass sich der Aufrufer darum kümmern müsste, Metamodell-Informationen bereitzustellen. Diese werden automatisch aus den Metamodell-Registires und Konfigurationsdateien ermittelt. Es ist zu beachten, dass eine M2T- oder T2M-Transformation nur dann ausgeführt wird, wenn die Quelldatei nicht aus einer Transformation entstanden ist und wenn die Quelldatei sich nach der Transformation nicht verändert hat. Dies unterbindet eine potentielle Endlosschleife von M2T- und T2M-Transformationen, wenn die Transformationen z.B. durch Builder automatisch gestartet werden. Dies führt dazu, dass Rücktransformationen von unveränderten Dateien nur nach manuellem Abspeichern der generierten Datei (um Eclipse eine Änderung an der Datei zu melden) durchgeführt werden können. Das Ermitteln von Metamodell-Informationen aus Metamodell-Registries lässt sich auch getrennt von der Synchronisation nutzen. Das Plug-in bietet auch die Möglichkeit Konfigurationsinformationen für QVT/R-Textdateien und visuelle QVT/R-Modellen abzurufen, welche ebenfalls bereits registrierte Metamodell-Informationen mit einbeziehen. Eine weitere Funktionalität, die das Plug-in bietet, ist das Öffnen von Editoren für Modellelemente. Dies wird vom graphischen Editor in Anspruch genommen, um einen Editor für Modellelemente, die aus Trace Daten ermittelt wurden, öffnen zu können.

Das Plug-in bietet eine konsolidiere Sicht auf die Grundfunktionen des Frameworks. Etliche der Funktionen des Frameworks, wie z.B. die Modell-zu-Text-Transformation und Metamodell Registries, werden über Extension Points angeboten, so dass es potentiell mehrere Implementierungen gibt. Davon bemerkt der Aufrufer der Funktionen allerdings nichts, da dieses Plug-in als Fassade

fungiert.

5.4.3 Die Modell-zu-Text-Transformation

Die Funktionalität ein visuelles Modell in eine textuelle Repräsentation zu überführen, wird über den Extension Point `fuberlin.qvtr.m2t` bereitgestellt. Die Standard-Implementierung dieses Extension Points wird vom Plug-in `fuberlin.qvtr.m2t.vis2text` angeboten. Dieses führt eine Transformation zu standardkonformen QVT/R-Quelltext durch, dadurch wird in einigen Fällen auch Quelltext erzeugt, der nicht kompatibel zu Medini QVT ist (Siehe Kapitel 1.4.2.E.i für die Abweichungen von Medini QVT vom QVT Standard). Diese Transformation soll unabhängig von der verwendeten QVT/R-Engine sein, daher werden keine Eigenheiten bestehender Engines berücksichtigt. Da mehrere M2T-Implementierungen möglich sind, können für bestimmte Engines angepasste Transformationen dennoch realisiert werden. Die mitgelieferte Transformation wird mittels oAW XPand durchgeführt. Quelltext 7 zeigt einen Ausschnitt aus dem dafür verwendeten Xpand Template. Dieser Ausschnitt zeigt die Regel, mit der Relationen in Text überführt werden.

```
2  «DEFINE relation FOR Relation»
3    «IF isToplevel»top «ENDIF»relation «name.qvtrEscape()»
4    «IF overrides!=null»overrides «overrides.name.qvtrEscape()»«ENDIF» {
5          «EXPAND variable FOREACH this.variable»
6          «EXPAND domain FOREACH this.content»
7          «IF when!=null»
8                when {
9                      «when.expression»
10               }
11         «ENDIF»
12         «IF where!=null»
13               where {
14                     «where.expression»
15               }
16         «ENDIF»
17   }
18 «ENDDEFINE»
```

Quelltext 7: Ausschnitt aus dem Xpand Template für die M2T-Transformation

Da das Not Templates, wie bereits unter 1.4.2.B.vi erwähnt, keine direkte Entsprechung in der konkreten Syntax haben, müssen stattdessen OCL-Constraints für Templates generiert werden. Diese Abbildung lässt sich nicht sinnvoll invertieren. Es handelt sich hier um einen Fall bei dem ein Modell mit reicher Semantik so transformiert werden muss, dass im Zielmodell eine komplexe Struktur entsteht. Dies ist ein typischer Fall für die MDSD, bei denen ein Roundtrip-Ansatz zumindest fragwürdig erscheint. Bei einer Rücktransformation kann nur erraten werden, an welcher Stelle ein Not Template eingesetzt wurde. Eine Unterscheidung, ob ein Konstrukt aus einem Not Template erzeugt wurde oder im graphischen Modell die Sprachkonstrukte gewählt wurden, die äquivalent zum Not

Template sind, kann bei der Rücktransformation nicht mehr ermittelt werden. Die einzige Möglichkeit bestünde darin, spezielle Kommentare als Markierungen zu erzeugen, die anzeigen, ob im visuellen Modell ein Not Template bestehen soll. Das setzt allerdings voraus, dass der für die T2M-Transformation genutzte QVT/R-Parser Kommentare beachtet. Da die T2M-Implementierung, die mit dem Visual QVT/R Framework mitgeliefert wird, auf dem Parser von Medini basiert und dadurch weder beim Parsen Kommentare beachtet, noch OCL-Beschränkungen für Templates unterstützt, ist es dem Framework in seiner derzeitigen Version unmöglich, eine solche Rücktransformation durchzuführen. Daher wurde auf eine Generierung von Kommentaren zur Erkennung von Not Templates verzichtet. Im folgenden Beispiel wird gezeigt, wie die Standard-M2T-Implementierung Not Templates übersetzt. In Abbildung 30 wird ein Beispiel für ein Not Template gegeben, wie es mit dem graphischen Editor modelliert werden kann; Quelltext 8 zeigt, in welche textuelle Repräsentation dieses übersetzt wird.

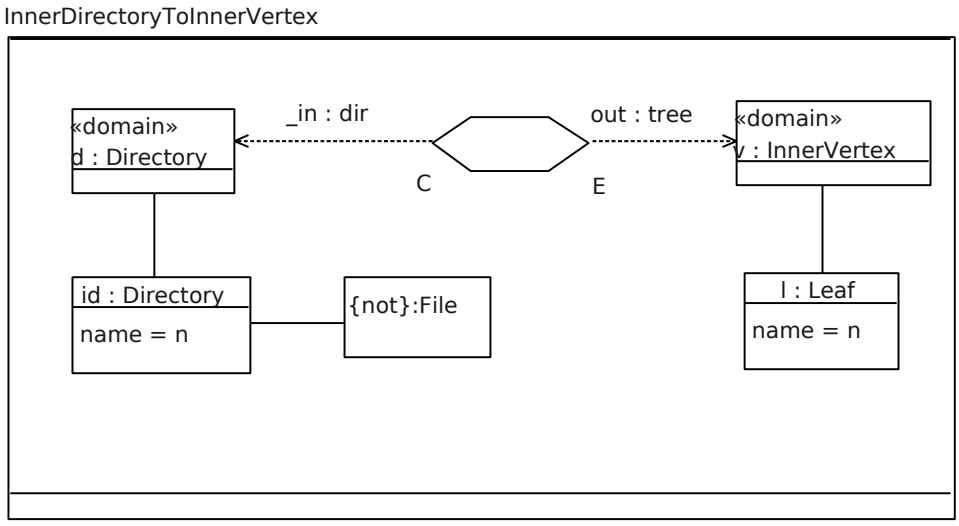


Abbildung 30: Beispiel für ein Not Template

```
1  relation InnerDirectoryToInnerVertex {
2
3    n : String;
4
5    checkonly domain _in d : dir::Directory {
6          content = id : dir::Directory {
7                name = n,
8                content = OrderedSet(dir::File){}
9          } {
10               content->size()=0
11         }
12   };
13
14   enforce domain out v : tree::InnerVertex {
15         children = l : tree::Leaf {
```

```
16                  name = n
17            }
18    };
19
20  }
```

Quelltext 8: Aus dem Not Template generierter QVT/R-Quelltext

5.4.4 Medini spezifische Funktionalität

Die meisten Funktionen des Visual QVT/R Frameworks, die von der Medini QVT Engine oder den Medini Cockpit Eclipse Plug-ins abhängen, sind in der Komponente `fuberlin.qvtr.medini` gebündelt. Die Funktionalität wird über Extension Points angeboten, wodurch diese Komponente durch eine andere Implementierung, die auf einem anderen Parser basiert, ausgetauscht werden kann.

5.4.4.A Die Text zu Modell Transformation

Eine Funktion ist die Umwandlung von textueller QVT/R-Transformationen in ein graphisches Modell. Dabei ist zu beachten, dass der Medini QVT nicht alle Sprachkonstrukte unterstützt (siehe 5.1.3). Einen groben Überblick über den Ablauf der T2M-Transformation wird in Kapitel 5.3.2.A gegeben. Der Parser gibt ein Modell zurück, dass einem konkreten Syntaxbaum entspricht; danach wird mittels Xtend eine M2M-Transformation zum graphischen Modell vorgenommen. Diese Transformation benötigt den Zugriff auf Metamodell-Registries und damit auf Java Methoden, da der QVT-Parser nicht die Abbildung von Metamodell-Bezeichnern auf Ecore-Elemente vornimmt. Dies ist auch der Grund, weshalb an dieser Stelle QVT/R-Transformationen mit der Medini-Engine anstatt der Xtend-Transformation zu aufwändig wären, denn die Medini-Engine unterstützt keine Black Box Operationen. Medini ermöglicht zwar den Aufruf von Operationen von Modellelementen, dies würde aber eine weitreichende Änderung am Syntax Modell benötigen. Die Operationen wären aus Sicht der Architektur auch nicht gut im Modell aufgehoben. Bei der Xtend Transformation müssen auch OCL-Ausdrücke, die als Modellelemente vorliegen wieder serialisiert werden, da sie im graphischen Modell nur als Zeichenketten abgelegt sind. Um die Eindeutigkeit der OCL-Ausdrücke zu gewährleisten, werden an einigen Stellen Ausdrücke geklammert. Dieses ließe sich bei einer erweiterten Implementierung durch eine Analyse der verwendeten OCL-Ausdrücke auf die notwendigen Fälle beschränken. Eine weitere Möglichkeit bestünde darin, durch Medini einen abstrakten Syntaxbaum zu erstellen, denn die Elemente dieses Baumes lassen sich mittels einer von Medini mitgelieferten Klasse serialisieren. Ein Problem, welches sich bei der Anwendung der Transformation offenbarte, war, dass das Überschreiben von bestehenden Modelldateien den ge-

öffneten graphischen Diagramm-Editor aus dem Tritt brachte, wenn er auf der Modelldatei arbeitete. Es kann passieren, dass graphische Elemente aus dem Diagramm entfernt werden.Tritt dieses Phänomen ein, sollte der Editor ohne abzuspeichern geschlossen werden. Nach einem erneuten öffnen des Editors erscheinen alle Elemente wieder dem Platz, an dem sie vor dem Auftreten des Problems zu finden waren. Da EMF standardmäßig eine Referenzierung von Modellelementen über eine Abfragesprache regelt und nicht über IDs, sind beim Überschreiben von Dateien des graphischen Modells nur dann Probleme zu erwarten, wenn sich die Struktur des Modells derartig ändert, dass die Reihenfolge von bisherigen Modellelementen in Listen verändert wird. Im ungünstigsten Fall muss die Modell-Datei, in der die Informationen über die Darstellung gespeichert ist („.qvtrvis_diagram“ Datei), gelöscht und neu angelegt werden.

5.4.4.B QVT/R-Quelltextvalidierung

Eine weitere Funktion des Plug-ins `fuberlin.qvr.core` ist die Validierung von generiertem QVT/R-Quelltext. Es wird dann versucht, die ermittelten Probleme im graphischen Modell zu lokalisieren. Die Fehlermeldung wird zusammen mit der Referenz auf das Element im graphischen Modell in einem Issues-Modell abgespeichert. Hier ergaben sich zwei Probleme.

Das eine bestand darin aufgrund der Fehlermeldungen die Position des Fehlers im visuellen Modell zu finden. Die vom Medini-Parser gelieferten Fehlerbeschreibungen, geben nur die Position des Fehlers im Quelltext an; intern enthalten die Beschreibungen allerdings Referenzen auf Modellelemente im konkreten Syntaxbaum, bei denen der Fehler festgestellt wurde. Die Engine wurde daher an dieser Stelle angepasst um, den Zugriff auf diese Elemente zu gewähren. Dazu genügte es, einen Getter für den Zugriff auf das Element hinzuzufügen. Dies allein war allerdings noch nicht ausreichend, um die Fehlerposition eindeutig im visuellen Modell zu bestimmen. Da die Elemente des konkreten Syntaxbaumes in den meisten Fällen keine Referenz auf ihr Elternelement im Baum besaßen, war es nicht möglich, den Kontext des Elementes im Syntaxbaum zu ermitteln. Um dies umzusetzen, musste das Metamodell des konkreten Syntaxbaumes so angepasst werden, dass die Referenzen durch Syntax-Modell-Kompositionen realisiert werden. Dadurch ist nun die Lokalisierung in den Fällen möglich, in denen die Fehlermeldung eine Referenz auf ein Element des konkreten Syntaxbaumes besitzt. Da OCL-Ausdrücke im Visual QVT/R Framework in textueller Form hinterlegt sind, lässt sich nicht immer die exakte Stelle des Fehlers im graphischen Modell zu markieren. Stattdessen werden so lange die nächsthöheren Elemente im Syntaxbaum durchsucht, bis ein Element gefunden wird, dass eine Entsprechung im visuellen Modell besitzt. Im ungünstigsten Fall gilt der Fehler für die gesamte Transformation.

Das zweite Problem der Validierung bestand in der Art der Speicherung von Fehlerbeschreibungen durch den Parser, der aus der unveränderten Medini-Engine aus den Medini Cockpit Plug-ins stammt. Diese Fehlerbeschreibungen werden in einer statischen Variable gespeichert. War nun der QVT/R-Texteditor, der Teil der Medini Cockpit Plug-ins ist, geöffnet und fand gleichzeitig eine Validierung durch das `fuberlin.qvtr.medini` Plug-in statt, so konnte der nebenläufige Zugriff beider Komponenten die Ergebnisse kompromittieren. Daher wurde die modifizierte Engine in ein eigenes Plug-in ausgelagert und den exportierten Klassen eine Versionsnummer gegeben. So können beide Versionen der Engine parallel existieren. Im Zusammenhang mit der Validierung ist auch eine Logger-Klasse entstanden, welche die Fehlermeldungen der Engine in einem Issues-Modell speichert. Diese Logger-Klasse kann auch von den Workflow-Komponenten der `fuberlin.qvtr.mwe.medini` genutzt werden.

5.4.4.C QVT/R-Quelltextformatierung

Eine weitere Funktionalität die ebenfalls vom `fuberlin.qvtr.medini` Plug-in über einen Extension Point bereitgestellt wird, ist die Möglichkeit QVT/R-Quelltext zu formatieren. Diese Funktionalität wird nach M2T-Transformationen automatisch durchgeführt, um den generierten Quelltext in eine lesbare Form zu bringen. Technisch wird dies dadurch realisiert, dass zunächst der Quelltext mittels Medini QVT eingelesen und in einen abstrakten Syntaxbaum überführt wird. Sollte beim Einlesen kein Fehler aufgetreten sein, wird das eingelesene Modell mithilfe einer Klasse aus der Medini-Engine wieder serialisiert und in die Quelltext Datei geschrieben, aus der der zu formatierende Quelltext eingelesen wurde. Beispiele für Fehler, die beim Einlesen gemeldet werden können, sind Syntaxfehler oder das Vorhandensein von Syntaxelementen, die von Medini nicht unterstützt werden.

5.4.4.D Medini Metamodel-Registry

Auch eine Metamodel-Registry wird von der `fuberlin.qvtr.medini` Komponente angeboten. Diese greift auf das Metamodell-Verzeichnis zu, welches von den Medini Cockpit Plug-ins bereitgestellt wird und vom Nutzer über den Eclipse-Einstellungsdialog befüllt werden kann. Auf diese Weise braucht ein Metamodell nur einmal registriert werden, um im graphischen Editor, im Medini QVT/R-Texteditor und beim Starten von QVT/R-Transformationen mit der Medini-Engine über Eclipse Run-Configurations genutzt zu werden. Da die Medini Plug-ins nicht Open Source sind, musste für diese Funktionalität ein wenig mit den Klassen der Medini Plug-ins experimentiert werden um eine Methode zu finden, die die registrierten Metamodelle zurückliefert. Eine Metamodel-

Registry bildet eine Abbildung von Namen zu Metamodell. Im Fall dieser Metamodel-Registry wird von den Namen der registrierten EPackages auf die Paket-Modellelemente abgebildet.

5.4.4.E Trace-Daten Zugriff und Trace Editor

Das mit dem Framework mitgelieferte Plug-in für den Zugriff des graphischen QVT/R-Editors auf Trace-Daten wurde auf Basis der von Medini erzeugten Trace-Metamodelle und diesen entsprechenden Modellen entworfen. Die Trace-Metamodelle, die von Medini erzeugt werden, enthalten für jede Relation der Transformation eine Klasse. Diese trägt den Namen der Relation und beinhaltet für jede Variable der Transformation (dazu gehören auch Template-Variablen) ein Attribut mit dem Namen der Variablen. Das Attribut besitzt den gleichen Typ wie die Variable, für die es steht. Im Trace-Modell wird jedes Mal ein Modellelement des Trace-Metamodellelementes für diese Relation erzeugt, wenn die Variablen dieser Relation an Werte (Modellelemente oder Werte primitiven Typs) gebunden werden, also die Relation angewandt wurde. Die Attribute dieses Modellelements können nun Referenzen auf die Modellelemente aus den Modellen, halten zwischen denen transformiert wurde. Um zu veranschaulichen, wie die Modellelemente des Trace-Modells aufgebaut sind, geben Abbildungen 31 und 32 ein Beispiel für eine Relation und das entsprechende Element im Trace-Metamodell.

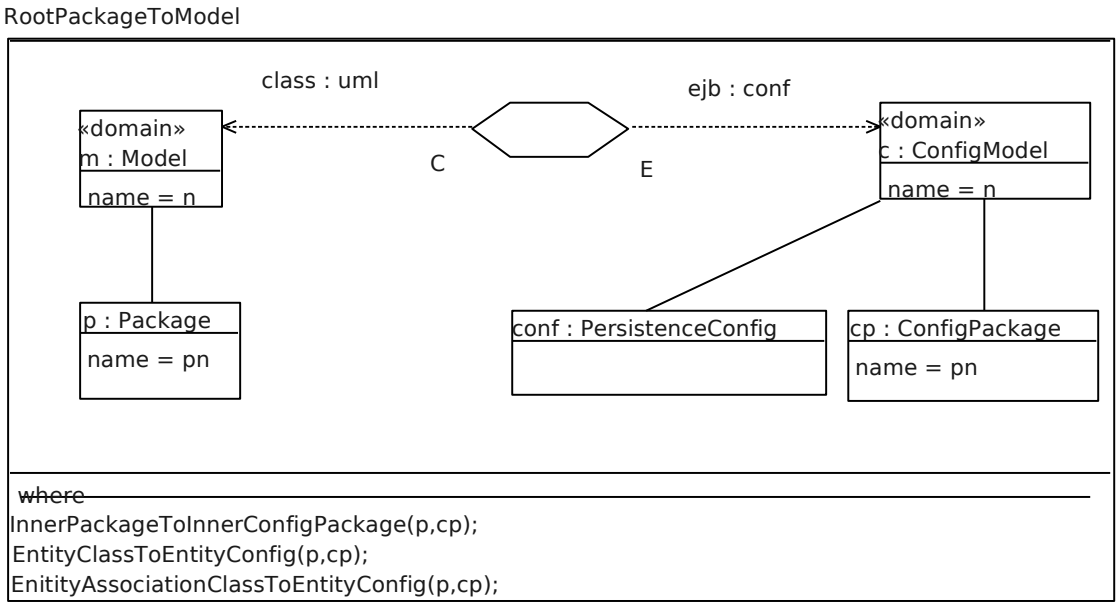


Abbildung 31: Beispiel Relation

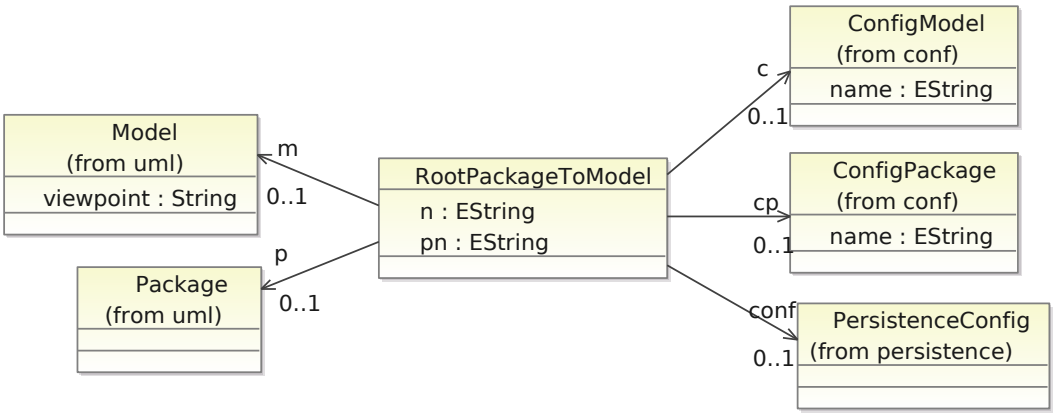


Abbildung 32: Ausschnitt aus Trace Metamodell

Das Plug-in ermöglicht die Abfrage von Modellelementen, die in einer Transformation als Wert an Template- oder Domain-Variablen gebunden wurden. Demnach wird das Trace-Modell nach Objekten durchsucht, dessen Metaklasse den Namen der Relation trägt, die das gefragte Template bzw. die gefragte Domain enthält. Dann werden die Werte der Attribute der Modellelemente ausgelesen, deren Attribut-Name dem Namen der Template-Variablen gleicht. Diese ausgelesenen Werte werden als Ergebnis der Anfrage zurückgegeben. Die Implementierung des Trace-Editors basiert auf dem Sample Reflective Ecore Model Editor. Das Plug-in erlaubt das gefilterte Öffnen des Editors, so dass nur Trace-Daten für eine bestimmte Relation angezeigt werden. Der Sample Reflective Ecore Model Editor benötigt im serialisierten Modell die Angabe einer Schema Location, also die Angabe des Speicherortes des Metamodells. Diese Angabe wird von der Medini-Engine leider nicht mit abgespeichert, weshalb eine weitere kleine Änderung an der Medini-Engine nötig war.

5.4.4.F Die modifizierte Medini-Engine

Da sowohl das Plug-in `fuberlin.qvtr.medini` als auch das Plug-in `fuberlin.qvtr.traceeditor` Modifikationen an der Medini-Engine erforderten, die es nötig machten eine Modifizierte Version der Medini-Engine bereitzustellen, die noch dazu nicht von den Medini Cockpit Plug-ins genutzt werden soll, wurde das Plug-in `fuberlin.qvtr.medini.engine` eingeführt. Es beinhaltet nichts weiter als die modifizierte Medini QVT Engine und das Plug-in exportiert ausgewählte Pakete der Engine. Den exportierten Paketen wurde eine Versionsnummer gegeben, so dass sie von den Paketen zu unterscheiden sind, welche die Medini Cockpit Plug-ins bereitstellen. Die Gründe für die Modifikation der Engine sind in Kapitel 5.4.4.B und 5.4.4.E dargelegt.

5.4.4.G Anbindung von Medini an die oAW Workflow Engine

Um eine Anbindung der Medini QVT Engine an die openArchitectureWare Workflow Engine zu gewährleistet, bietet das Plug-in `fuberlin.qvtr.mwe.medini` eine Reihe von Workflow-Komponenten. Um diese Komponenten nutzen zu können muss, der Nutzer das Plug-in als Abhängigkeit in die Manifest-Datei des oAW-Projektes eintragen, in dem die Workflow Komponenten genutzt werden sollen. Aufgrund von Problemen bei der Auflösung von Abhängigkeiten, muss zusätzlich eine Abhängigkeit zum Bundle `fuberlin.qvtr.medini.engine` eingetragen werden, welches die Klassen der QVT Engine enthält. Die Komponenten erlauben das Laden von Metamodell-Daten und Modell-Resourcen sowie das Starten von QVT/R-Transformationen. Es stellte sich heraus, dass es bei der Verwendung von Ecore-Metamodell Dateien eine Rolle spielt, ob Modell und Metamodell in das gleiche ResourceSet geladen wurden oder nicht. Daher wurde eine Möglichkeit eingebaut, explizit ResourceSets zu definieren, die genutzt werden können, um Metamodell und Modell mit demselben ResourceSet Exemplar zu laden. Die Workflow Komponenten des `fuberlin.qvtr.mwe.medini` Plug-in lassen sich losgelöst vom restlichen Visual QVT/R Framework nutzen. Nur die Abhängigkeit zum Bundle `fuberlin.qvtr.medini.engine`, welches die Medini QVT/R-Engine enthält, muss beachtet werden. Die Komponenten lassen sich auch außerhalb einer OSGi-Runtime ohne Einschränkung nutzen. In diesem Fall wird lediglich die Medini QVT/R-Engine im Klassenpfad benötigt.

Um die Fehlermeldungen der Engine im Editor zu visualisieren, kann die Logger-Klasse aus dem Plug-in `fuberlin.qvtr.medini` genutzt werden. Leider können Fehler, die zur Laufzeit der Engine auftreten, nicht auf VQVT/R-Modellelemente abgebildet werden, so dass diese Fehler im graphischen Editor immer im Kontext der gesamten Transformation angezeigt werden. Dies ist allerdings kein generelles Problem, sondern hat seine Gründe in der Art, wie die Medini-Engine Fehler zur Laufzeit meldet. Die Fehlerbeschreibungen beinhalten keine Referenz auf das Modellelement der Transformation, in dessen Kontext der Fehler aufgetreten ist. Dies wäre für eine genaue Lokalisierung des Laufzeitfehlers im visuellen Modell aber nötig. Damit die Logger-Klasse in einem Workflow verwendet werden kann, muss das `fuberlin.qvtr.medini` Plug-in ebenfalls in die Liste der Abhängigkeiten des oAW Projektes eingetragen werden. Da der Logger Zugriff auf den Eclipse-Workspace und auf das Visual QVT/R Framework benötigt, lassen sich Workflows, die den Logger verwenden, nur innerhalb einer Eclipse JVM nutzen. Um einen Workflow in der Java VM zu starten in der auch Eclipse läuft, bietet das Visual QVT/R Framework Einträge im Kontextmenü von Workflow-Dateien.

5.4.5 Anbindung an die EMF Registry

Die Komponente `fuberlin.qvtr.metamodelregistry.ecore` bietet eine Anbindung des Visual QVT/R Frameworks an die EMF Package-Regisry. Die Implementierung des `fuberlin.qvtr.metamodelregistry` Extension Points bietet eine Abbildung vom Paketnamen auf die EPackage Elemente, welche sich im EMF Paketverzeichnis befinden. Damit können im graphischen Editor ohne weitere Einstellungen sofort alle Metamodelle der EMF Package-Registry genutzt werden.

5.4.6 Die Visual QVT/R Nature

Das Plug-in mit der ID `fuberlin.qvtr.nature` bietet eine Anbindung des Eclipse Workbench an die Kernfunktionen des Visual QVT/R Framework. Das Plug-in bietet die Projekt-Nature „Visual QVT/R“, die einen M2T- und einen T2M-Builder enthält. Diese Builder sorgen automatisch für eine Synchronisation der visuellen Modelle und der konkreten textuellen Syntax. Stellt der Workbench eine Änderung an einer texutellen Datei oder einem visuellen Modell fest, so wird der entsprechende Builder gestartet. Die Nature kann über einen Menüeintrag zu einem Projekt hinzufügt und wieder entfernt werden. Beim Hinzufügen der Nature werden automatisch der M2T- und der T2M-Builder für das Projekt aktiviert. Es ist jedoch davon abzuraten, M2T- und T2M-Builder gleichzeitig zu nutzen, da keine komplette Roundtrip-Fähigkeit vorhanden ist. Die M2T- und T2M-Transformationen können beide verlustreich sein (siehe 5.4.3 und 5.4.4.A für Details). Daher erscheint es sinnvoller, die Transformationen entweder in textueller oder in graphischer Notation zu entwickeln. Wird in der graphischen Notation gearbeitet, kann der M2T-Builder eingeschaltet sein, um immer automatisch eine textuelle Version zum Ausführen mittels einer QVT-Engine bereit zu halten. Wird die Transformation textuell entworfen, so kann der T2M-Builder dazu genutzt werden das graphische Modell kontinuierlich zu erzeugen. Da die Diagramm-Datei für das visuelle Modell („.qvtrvis_diagram“ Datei) bei größeren Änderungen an der Transformation unbrauchbar werden kann, ist es sinnvoll Diagramm-Modelldateien nur dann zu erzeugen und aufzuarbeiten, wenn ein stabiler Zwischenstand erreicht wurde und die graphische Darstellung z.B. zu Dokumentations- oder Präsentationszwecken benötigt wird.

Auch ein Menü zum Öffnen des Konfigurationsmodells für eine visuelles Modell oder eine textuelle QVT/R-Datei wird von dem `fuberlin.qvtr.nature` Plug-in bereitgestellt.

Da openArchitectureWare-Workflows, die über die Run-Configuration der openArchitectureWare Eclipse Plug-ins gestartet werden, nicht in der Eclipse JVM laufen und es auch keine Option dafür gibt, haben Workflow-Komponenten keinen Zugriff auf den Eclipse Workspace. Da die Logger-

Klasse aus dem Plug-in `fuberlin.qvtr.medini` auf den Workspace zugreifen muss (siehe vorheriges Kapitel), wurde in Plug-in `fuberlin.qvtr.nature` eine Möglichkeit geschaffen, Workflows in der Eclipse JVM zu starten. Allerdings wird die Ausgabe Workflow-Meldungen in die Standardausgabe von Eclipse umgeleitet, so dass sie im Normalfall für den Benutzer unzugänglich sind.

5.4.7 Visual QVT/R Modellvalidierung

Bereits in den Vorüberlegungen zur Implementierung wurde erwähnt, welche Constraint-Technologien zum Tragen kommen (siehe Kapitel 5.2.5).

Aus den OCL-Constraints, die im GMF Mapping Modell definiert wurden, wurde Quelltext für den graphischen Editor (`fuberlin.qvtr.vis.ecore.diagram`) generiert, so dass die Einbindung in den graphischen Editor bereits gegeben ist und keine weiteren Eingriffe in den generierten Code oder in Konfigurationsdateien nötig waren.

Das Plug-In `fuberlin.qvtr.vis.check` beinhaltet die oAW Check Constraints, die über einen Extension Point EMF- und GMF-Editoren zur Verfügung stehen. Zusätzlich beinhaltet das Plug-in die Implementierung von zwei Constraints auf Basis des EMF-Validation Framework. Sie werden über Extension Points des Validation Frameworks angeboten und müssen (anders als bei den oAW Cheks) von den verwendeten Editoren explizit referenziert werden. Diese mittels Java implementierten Constraints geben Fehler und Warnungen aus, die sie in einem Problem-Modell finden, das dem zu prüfenden Modellelement zugeordnet ist. Ein Problem, welches sich aus der Nutzung der Constraints für das EMF Validation Framework ergab, war, dass bei jedem Aufruf eines der Constraints das Problem-Modell neu eingelesen werden musste, was zu starken Einbußen in der Performanz der Validierung führte. Um dieses Problem zu umgehen, wird das Problem Modell nun im gleichen ResourceSet geladen, wie das Modell das validiert wird. Dadurch wird das Problem-Modell im ResourceSet des Editors gehalten, der die Validierung anstößt und das Problem-Modell muss nicht immer wieder neu eingelesen werden.

5.4.8 Der graphische QVT/R-Editor

Der graphische Editor wurde auf Basis des VQVT/R-Metamodell (siehe 5.4.1.A) mithilfe von GMF entwickelt. Modell- und Darstellungsdaten werden in getrennten Dateien gespeichert, um die Synchronisation zwischen Modell und textueller Repräsentation zu vereinfachen. Der generierte Editor wurde um manuelle Anteile zur Anbindung an Trace-Daten und zum Starten der Modellvalidierung über eine Schnittstelle ergänzt.

Da die Metamodellelemente für Variablen, PrimitiveDomains, TypedModels, OCL-Queries, Collection-Inhalte und Schlüsseldefinitionen zwar in der abstrakten Syntax existieren, aber keine konkrete graphische Syntax im QVT Standard beschrieben ist, musste ein Weg gefunden werden, diese Elemente außerhalb des graphischen Editors bequem anzulegen und bearbeiten zu können. Um diese Funktionalität im graphischen Editor anzubieten, musste eine erweiterte Outline-View genutzt werden. Ein Patch für GMF, der für die Unterstützung dieser Outline-View sorgt, wurde im Rahmen der Diplomarbeit von Enrico Schnepel [Schne08] entwickelt und freundlicherweise vom Autor selbst nach GMF 2.1 portiert. Ein weiteres Problem ist, dass die graphische Syntax nicht angibt, in welcher Reihenfolge Parameter an Relationen übergeben werden müssen. Die graphische Position der Domains in der Relation macht keine Aussage über die Reihenfolge der Parameter. Dieses Problem ließ sich ebenfalls mit der Outline-View lösen. Die Reihenfolge der Parameter lässt sich nun über die Outline-View steuern. Die Parameter-Liste baut sich entsprechend der Reihenfolge in der Outline-View (von oben nach unten) auf. Die Reihenfolge kann nachträglich per Drag & Drop verändert werden.

Die Live-Validierung funktionierte in GMF Version 2.1 leider nicht wie geplant. Wünschenswert ist ein sofortiges Anzeigen der Dekoratoren an Modellelementen, für die Fehler oder Warnungen vorliegen. Die Dekoratoren werden allerdings nur nach einer „on demand“ Validierung des ganzen Modells angezeigt. OCL-Constraints für Live-Validierung, die fehlerhafte Zustände evaluieren, verhindern noch dazu Operationen, die zu diesen Zuständen führen, was nicht das erwünschte Verhalten ist. Live-Validierungen führen in den Standardeinstellungen auch dazu, dass Nutzer über auftretende Fehler und Warnungen mittels einer Popup-Meldung über das Verletzen des Constraints informiert. Da diese Popups relativ häufig auftreten, können die Meldungen nach kurzer Zeit als unangenehm empfunden werden. Es ist allerdings möglich, im Eclipse-Konfigurationsmenü einzustellen, dass die Meldungen in ein Konsolenfenster gelenkt werden und dieses nicht angezeigt wird, wenn eine neue Meldung eintrifft. Um die Problem-Dekoratoren trotzdem möglichst zeitnah zur Modellierung anzuzeigen, wurde der graphische Editor so modifiziert, dass eine vollständige Modellvalidierung durchgeführt wird, wenn das Modell abgespeichert wird. Da die Validierung des gesamten Modells recht aufwändig ist, erscheint es störend die Validierung in regelmäßigen zeitlichen Abständen durchführen zu lassen.

6 Funktionen des Frameworks

Dieser Abschnitt beschreibt die Features des Frameworks aus Sicht des Nutzers. Die beschriebenen Funktionen stehen zur Verfügung, wenn das Framework inklusive der von Medini QVT abhängigen Komponenten installiert wurde.

6.1 Modellierung von Transformationen

Mittels des QVT/R-Diagramm-Editors können Transformationen im kompletten QVT/R Sprachumfang modelliert werden. Es werden sowohl alle Elemente der graphischen Syntax unterstützt als auch solche Elemente der Sprache, für die keine graphischen Syntaxelemente standardisiert sind.

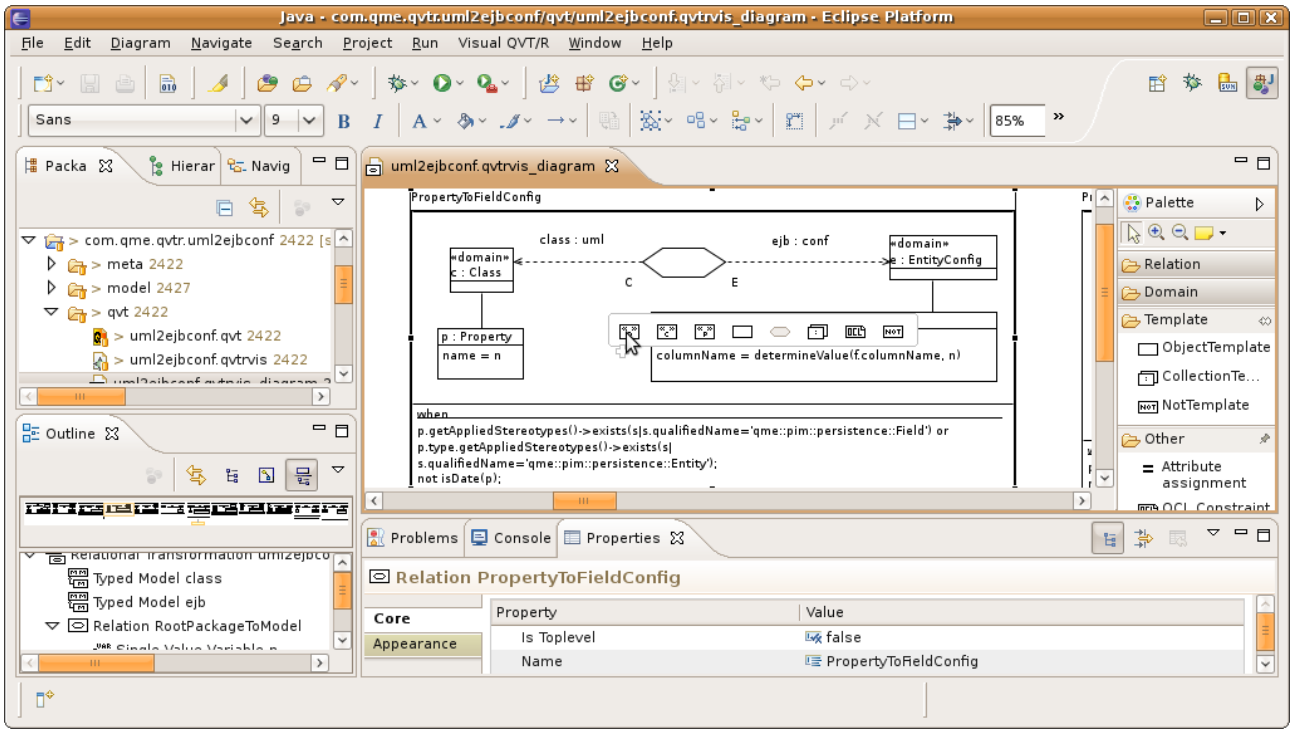


Abbildung 33: Visual QVT/R Editor

Nach Erstellen eines neuen VQVT/R-Modells mittels Wizzard in Eclipse bietet sich dem Nutzer eine Diagramm-Fläche, die für eine Transformation steht. Für diese Transformation sollte in der Properties-View ein Name eingegeben werden, daraufhin können in der Outline-View Parameter angegeben werden. Für die Auswahl des Typs eines Parameters werden die Metamodell-Repositories angesprochen. Hier werden auch die Metamodelle aufgelistet, die im Einstellungsdialog der Medini Cockpit Plug-ins registriert wurden. So kann der Nutzer erkennen, welche Metamodelle zur Auswahl stehen, und er kann keine ungültige Auswahl treffen. Beim Erstelluen von Relationen

können die angelegten Parameter nun ausgewählt werden, um zu definieren, aus welchem Modell die angelegten Domains stammen.

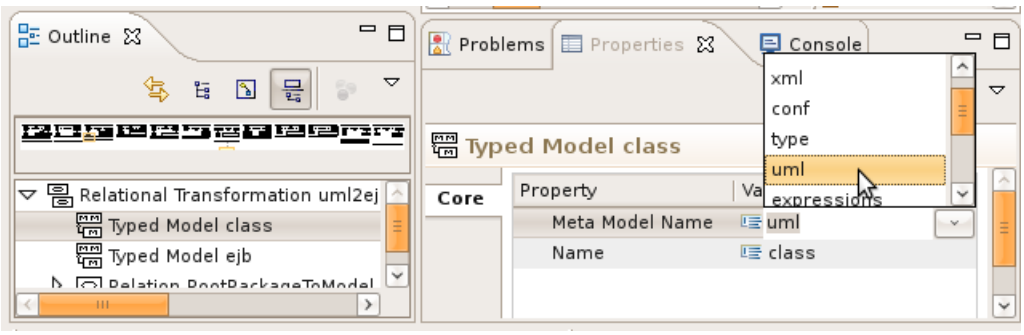


Abbildung 34: Auswahl des Metamodells

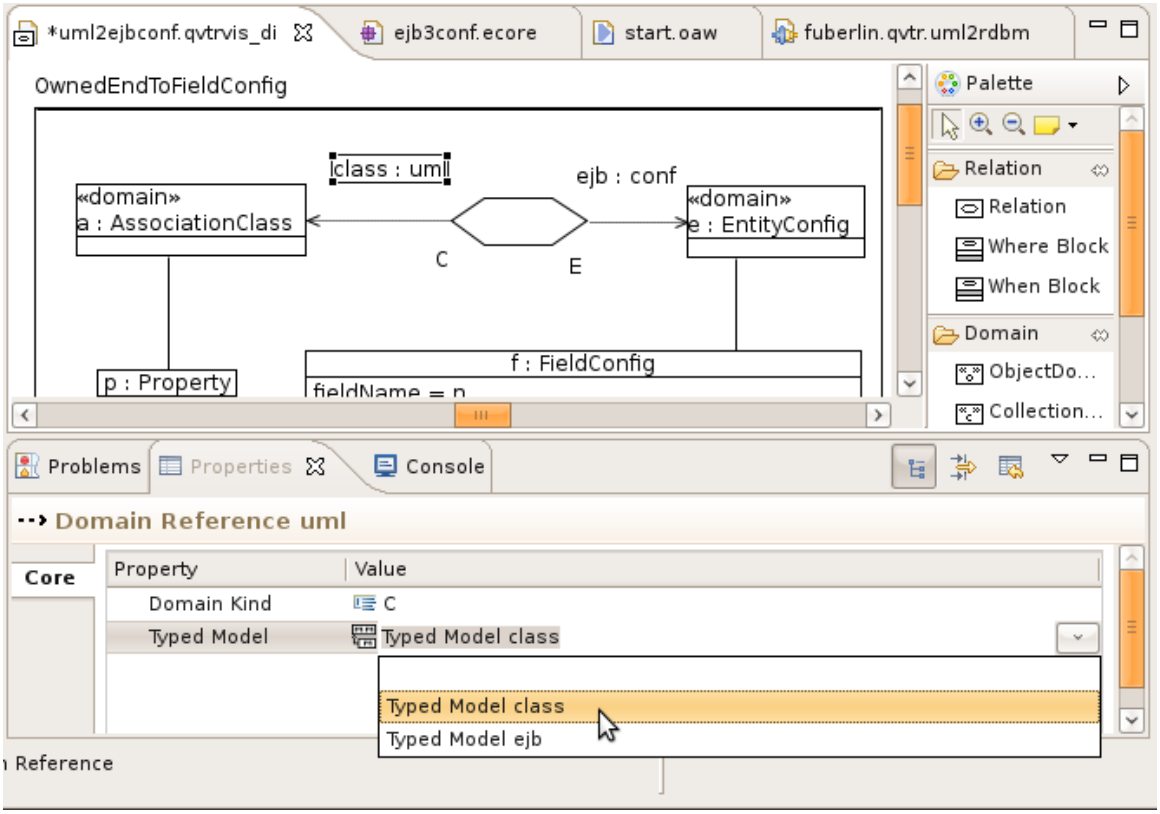


Abbildung 35: Auswahl aus den angelegten TypedModels

Wenn der Nutzer eine Eigenschaft eines Modellelementes setzten kann, wird ihm – wenn möglich – eine dem Kontext des Modellelemetes angepasste Auswahl geboten. Dieses Vorgehen wird im Editor durchgängig angewendet. So wird bei der Auswahl des Typs einer Domain die Auswahlliste der zur Verfügung stehenden Typen aufgrund des gewählten Parameters (TypedModels) eingeschränkt. In der Auswahlliste tauchen also nur Typen auf, die im Metamodell zu finden sind, dass durch den Parameter-Typ festgelegt ist. Ebenso wird die Liste von Attributen bei Property Template Items und Referenzen zwischen Templates eingeschränkt. Entsprechend der Attribut-Wahl

von Template-Referenzen schränkt sich dann auch die Typ-Auswahlliste der referenzierten Templates ein.

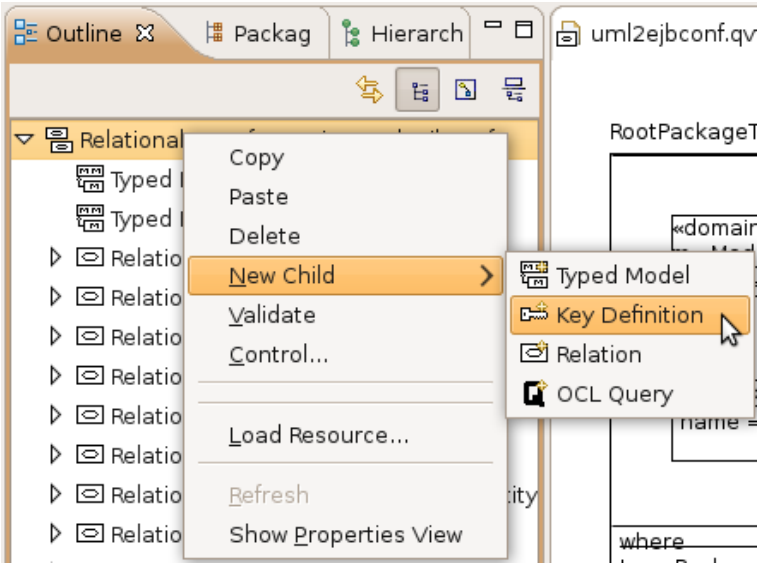


Abbildung 36: Anlegen eines neuen Schlüssels über die Outline-View

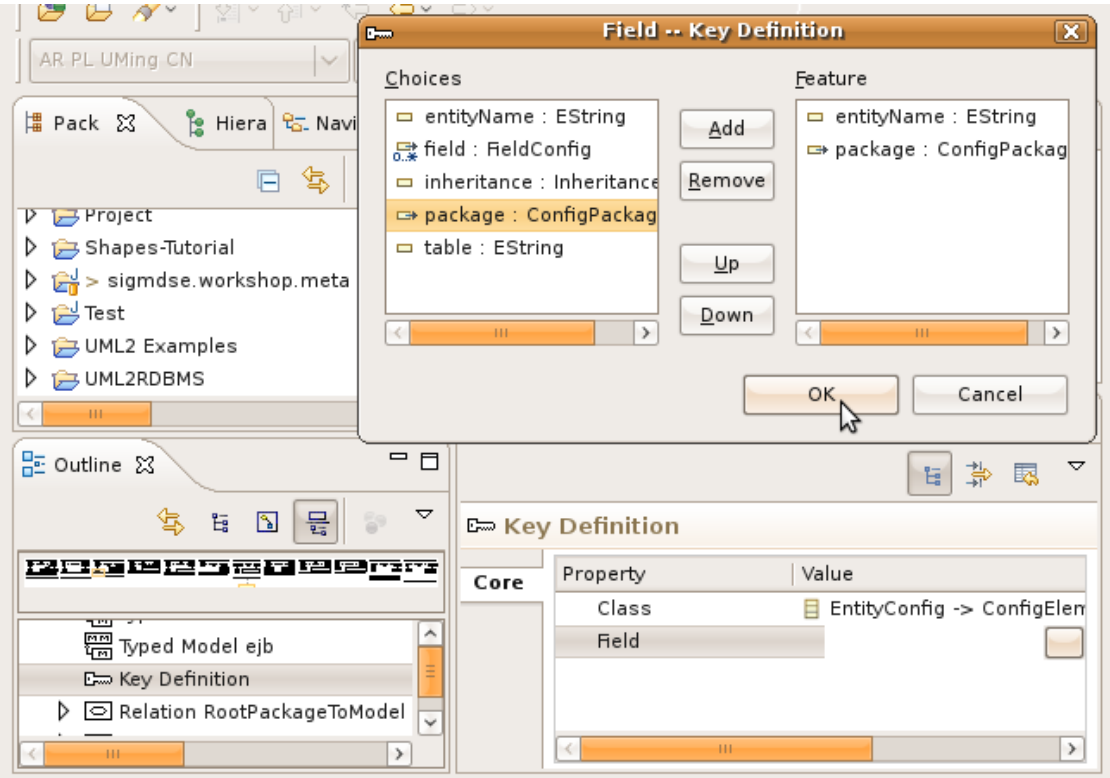


Abbildung 37: Auswahl von Schlüsselattributen

Um die Elemente der QVT Relations Sprache zu unterstützen, für die keine graphische Syntax standardisiert ist, können Transformationen in der detaillierteren Outline editiert werden. So können

z.B. Schlüsselattribute für Transformationen über die Outline angelegt und editiert werden.

Wichtig sind auch Variablen, die im Kontextmemü der Outline-View für Relationen angelegt werden können. OCL-Queries, die ebenfalls nicht in der graphischen Syntax auftauchen und somit ebenfalls über die Outline-View editiert werden, werden komplett in textueller Syntax angegeben.

6.2 Konfiguration des Visual QVT/R Frameworks

In der Konfiguration für das Visual QVT/R Framework wird festgelegt, welche Dateien mit der Synchronisation zwischen konkreter textueller Syntax und graphischen Modell synchron gehalten werden sollen. Die Konfiguration für Dateien der textuellen Syntax und für visuelle Modelle lässt sich über die gleichen Menüs öffnen, wie die M2T- und T2M-Transformationen zu starten sind (siehe nächstes Kapitel).

Da das Framework grundsätzlich die Generierung in unterschiedliche textuelle Sprachen erlaubt, beinhaltet die M2T-Konfiguration den Bezeichner der Sprache, in die generiert werden soll. Wird die Konfiguration für eine Datei angelegt, so wird automatisch eine M2T-Konfiguration für die Synchronisation mit einer QVT/R-Datei angelegt.

Das Visual QVT/R Framework wurde so entworfen, dass es im Prinzip unabhängig von QVT/R-Engines ist. Um diese Unabhängigkeit zu gewährleisten können in der Konfiguration für Modell- und Quelltext-Dateien auch Abbildungen von Namen auf Metamodelle definiert werden. Damit ist

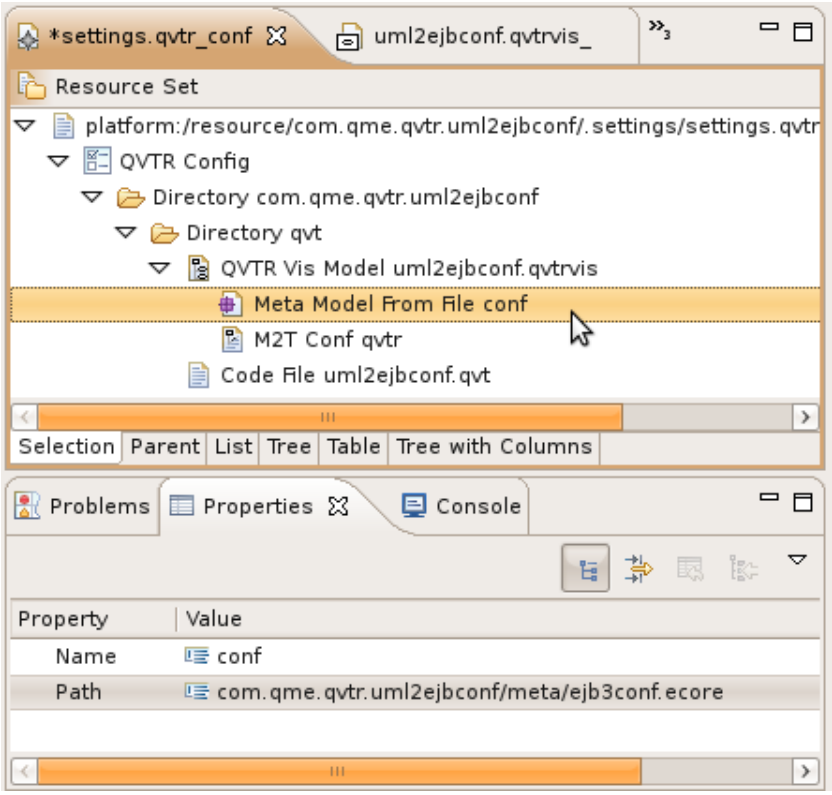


Abbildung 38: Angabe einer Abbildung eines Namens auf ein Metamodell in der Konfiguration

Visual QVT/R unabhängig allen Metamodell-Repositories, speziell den Repositories von QVT/R-Engines.

6.3 Synchronisation zwischen texuteller und graphischer Syntax

Das Framework bietet neben dem graphischen Editor die Möglichkeit, aus dem visuellen Modell die textuelle Repräsentation von Transformationen zu generieren und umgekehrt aus der textuellen Version ein visuelles Modell zu erstellen. Um die visuell erstellten Transformationen auszuführen, ist eine Überführung in die textuelle Repräsentation erforderlich.

Zum manuellen Starten der Synchronisierung wurden Kontextmenüs zu den Dateien im Eclipse Workspace hinzugefügt, die als Quelle für eine M2T- oder T2M-Transformation dienen können.

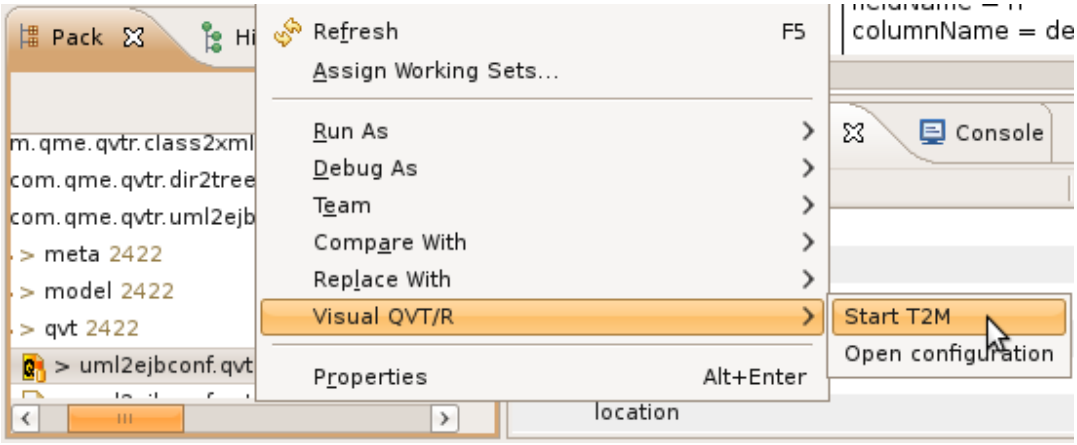


Abbildung 39: Start der T2M Transformation über das Kontextmenü

Alternativ kann aus dem „Visual QVT/R“ Menü des graphischen Editors heraus die M2T-Transformation für das gerade geöffnete Modell gestartet werden.

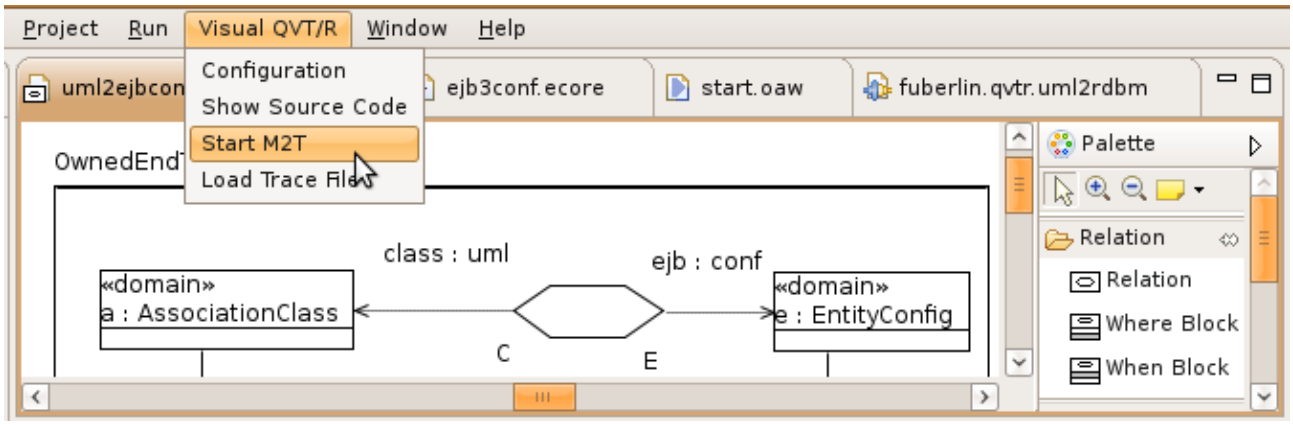


Abbildung 40: Start der M2T Transformation über das Hauptmenü

Um die Synchronisation zu automatisierten, wurde die Visual QVT/R Nature eingeführt, welche den Visual QVT/R M2T-Builder und den Visual QVT/R T2M-Builder zu einem Projekt hinzufügt.

Sie kann über das Kontextmenü eines Projektes dem Projekt hinzugefügt oder von ihm gelöscht werden. Es ist aufgrund der (teilweise Prinzip bedingten) Verluste während der Transformationen nicht empfehlenswert, während der Entwicklung sowohl im graphischen Editor als auch in der textuellen Syntax zu arbeiten. Stattdessen sollte in einer Syntax gearbeitet und die jeweils andere Syntax generiert, aber möglichst nicht bearbeitet werden.

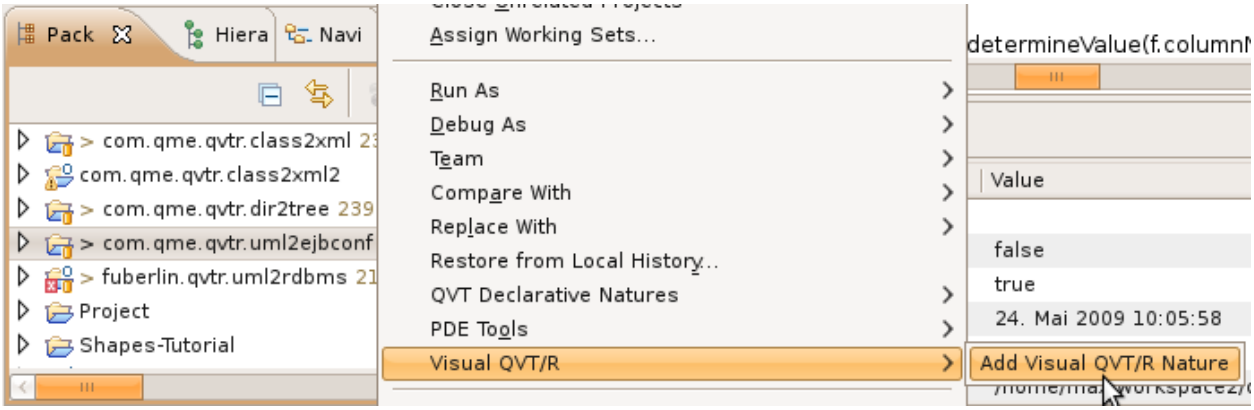


Abbildung 41: Hinzufügen der Visual QVT/R Nature zu einem Projekt

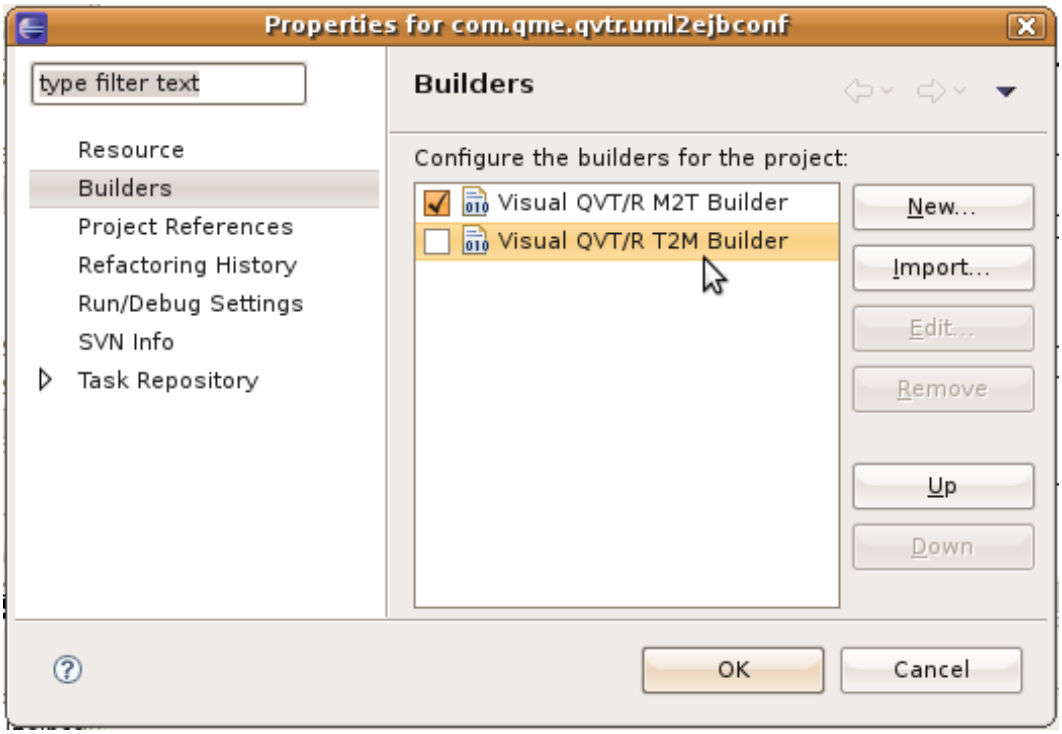


Abbildung 42: Builder-Einstellungen in den Projekteigenschaften

Die M2T-Transformation startet zusätzlich einen passenden Codevalidator, dessen Ergebnisse in den graphischen Editor angezeigt werden, und einen Quelltextformatierer, um ein lesbares Ergebnis der M2T-Transformation zu erhalten.

Die T2M-Transformation erzeugt nur die Modelldatei, welche die Modelldaten enthält. Um Dia-

gramme zu erhalten, muss eine „.qvtrvis_diagram“ Datei erstellt werden. Dies kann über das Kontextmenü der Datei des visuellen Modells erfolgen.

6.4 Verknüpfung mit Trace-Daten

Die nach der Durchführung einer Transformation entstandenen Trace-Daten können über das „Visual QVT/R“ Menü des graphischen Editors geladen werden. Daraufhin erscheinen Trace-Icons an den Relationen im Editor. Ein Klick auf einen dieser Dekoratoren öffnet den Trace-Editor, der eine gefilterte Sicht auf die Trace-Daten bietet. Es werden nur noch die Trace-Daten angezeigt, die durch die Relation entstanden sind, dessen Icon angeklickt wurde.

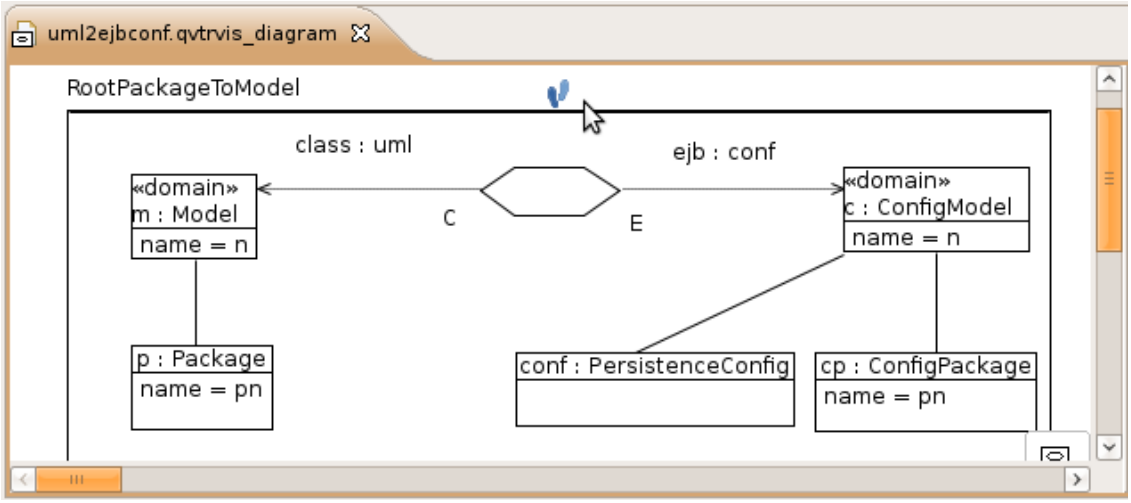


Abbildung 43: Trace Dekorator über einer Relation

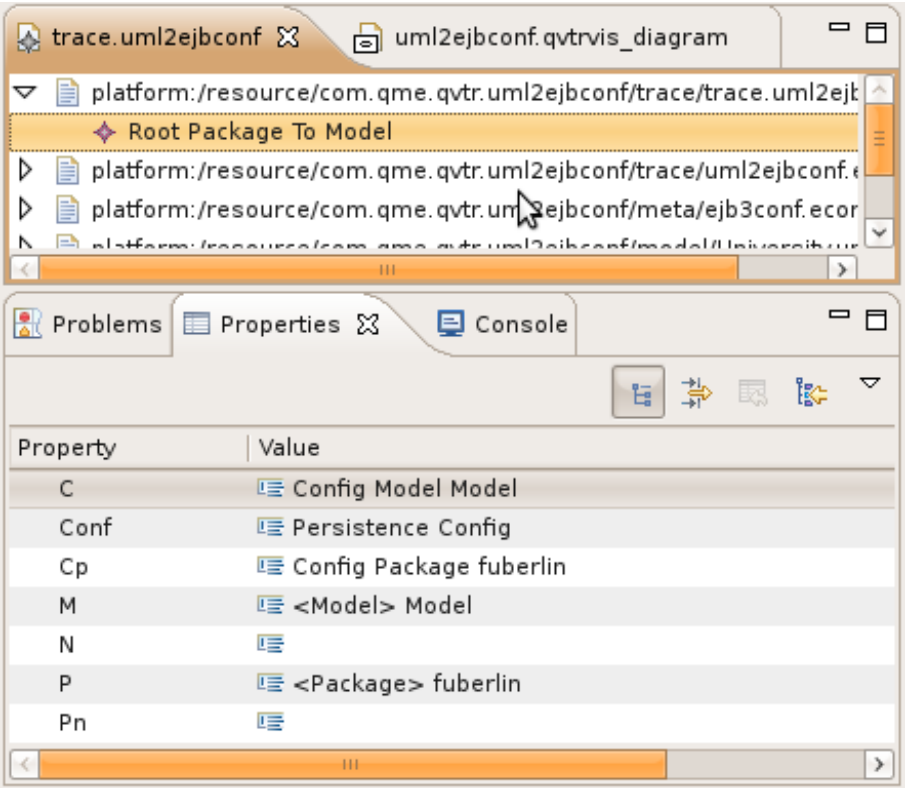


Abbildung 44: Gefilterter Trace-Editor nach Klick auf Dekorator

Des Weiteren erscheinen nun im graphischen QVT/R-Editor im Kontextmenü von Domains und Templates Menüeinträge für jeden Match, der in den Trace Daten aufgezeichnet ist. Nach dem Klick auf einen Menüeintrag, wird versucht einen entsprechenden Editor für das gewählte Modellelement zu öffnen.

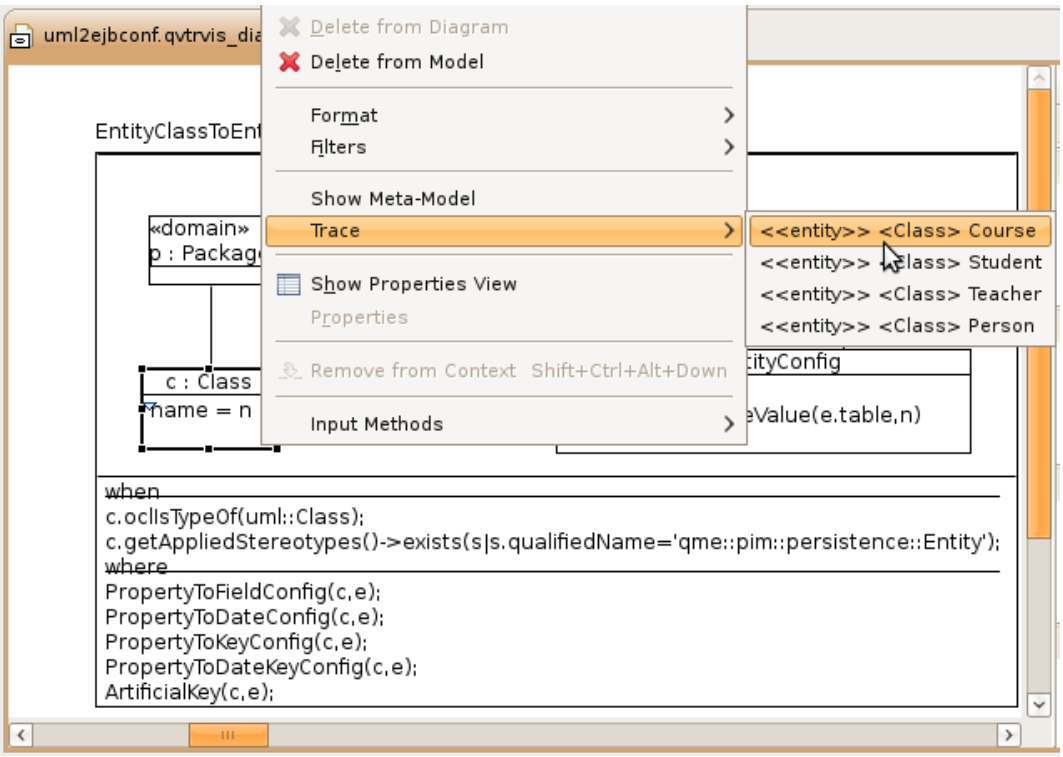


Abbildung 45: Tracedaten Zugriff aus dem Kontextmenü eines Templates

6.5 Anbindung an die openArchitectureWare Workflow Engine

Um QVT/R-Transformationen in eine Transformationskette einzubinden, wird eine Ablaufsteuerung benötigt. Die openArchitectureWare-Workflow Engine bietet eine Technologie, die solches leistet. Es wurden daher Komponenten zum Starten von Transformationen mittels der oAW-Workflow-Engine erstellt, die in oAW-Projekten genutzt werden können, indem das Plug-in `fuberlin.qvtr.mwe.medini` als Abhängigkeit eingetragen wird. Quelltext 9 zeigt einen Beispiel-Workflow, der eine QVT/R-Transformation anstößt.

```
1  <?xml version="1.0"?>
2  <workflow>
3          <component
4           class="fuberlin.qvtr.mwe.medini.component.ResourceSetComponent"
5                    resourceSetSlot="rs" />
6         <!-- Lade Metamodelle -->
7          <component
8             class="fuberlin.qvtr.mwe.medini.component.EcoreModelRegistry"
9                    resourceSet="rs"
10                    modelFile="./src/metamodel/SimpleUML.ecore"
11                    modelSlot="uml_mm" />
12          <component
13            class="fuberlin.qvtr.mwe.medini.component.EcoreModelRegistry"
14                    resourceSet="rs"
15                    modelFile="./src/metamodel/SimpleRDBMS.ecore"
16                    modelSlot="rdbms_mm" />
17        <!-- Lade Modelle -->
18          <component
19             class="fuberlin.qvtr.mwe.medini.component.EmfResourceReader"
20                    metaModel="uml_mm"
21                    modelFile="./src/qvt/source.xmi"
22                    modelSlot="uml_mdl" />
23          <component
24             class="fuberlin.qvtr.mwe.medini.component.EmfResourceReader"
25                    metaModel="rdbms_mm"
26                    modelFile="./src/qvt/target.xmi"
27                    modelSlot="rdbms_mdl" />
28        <!-- Starte Transformation -->
29          <component
30         class="fuberlin.qvtr.mwe.medini.component.MediniMweEmfComponent"
31                    transformation="uml2rdbms"
32                    direction="rdbms"
33                    firstSetOfModels="uml_mdl"
34                    secondSetOfModels="rdbms_mdl"
35                    traceDir="./src/trace"
36                    traceResourceSet="rs"
37                    qvtScript="./src/qvt/uml2rdbms.qvtr"
38                    loggerClass=
39                        "fuberlin.qvtr.medini.logger.IssuesModelLogger">
40                 <metaModel slot="uml_mm" />
41                 <metaModel slot="rdbms_mm" />
42          </component>
43  </workflow>
```

Quelltext 9: Beispiel-Workflow für das Starten einer QVT/R-Transformation

7 Evaluierung

In diesem Abschnitt wird auf qualitative Aspekte der Implementierung eingegangen und ein Fallbeispiel vorgestellt, mit dem die Nutzbarkeit des Frameworks in einer praktischen Anwendung unter Beweis gestellt wird.

7.1 Stabilität der Implementierung

Die Stabilität der Implementierung des Visual QVT/R Frameworks hängt insgesamt von der Stabilität der verwendeten Plattformen und Frameworks ab. Probleme gibt es z.B. mit dem Aufruf der Validierung in Klassen, die von GMF generiert wurden. Wird die Validierung von außerhalb des Threads aufgerufen, in dem der Diagramm Editor läuft, ist es möglich, dass eine NullPointer-Exception gemeldet wird. Dies scheint aufzutreten, wenn während der Validierung das Menü zum Erstellen von Elementen auf der Zeichenfläche angezeigt werden soll. Dieses Menü erscheint automatisch, wenn der Mauszeiger für einige Zeit über dem graphischen Editor verweilt. Eine Anfrage auf der GMF-Newsgroup zu diesem Problem blieb unbeantwortet. Ein weiteres Problem ist das Durchführen einer T2M-Transformation während der GMF Editor das zu überschreibende Modell geöffnet hat. In diesem Fall kann der Editor in fehlerhafte Zustände geraten, in denen existierende Modellelemente vom Diagramm entfernt werden. Wird der Editor, ohne zu speichern, geschlossen und wieder geöffnet, sind die Elemente wieder vorhanden. Kommt es durch die T2M-Transformation zu großen Änderungen im visuellen Modell oder zu Änderungen, die die Reihenfolge von Elementen verändern, dann kann das graphische Modell komplett ungültig werden. Das visuelle Modell ließe sich nicht in allen Fällen ohne genaue Kenntnisse über die Änderungen des Nutzers in der textuellen Repräsentation sinnvoll automatisch aktualisieren.

7.2 Performance der Implementierung

Die T2M- und M2T-Transformationen stellen aufwändige Operationen dar, die durch Verzögerungen den Arbeitsfluss des Nutzers des Frameworks unterbrechen können. Vor allem dürfte hier die Dauer der M2T-Transformation von Bedeutung sein, denn zum einen werden im graphischen Editor erst alle Fehlermeldungen angezeigt, wenn die M2T-Transformation und die anschließenden Validierungen abgeschlossen sind, und zum anderen wird die textuelle Syntax für die Ausführung der Transformationen benötigt. In der Entwicklung von Transformationen können häufige Änderungen in schneller Folge auftreten. Um die Transformationen in einem solchen Arbeitsfluss möglichst ohne Verzögerungen starten zu können muss, die M2T-Transformation entsprechend

schnell vonstatten gehen. Daher wurden einige Messungen durchgeführt, um zu sehen, wie lange diese Unterbrechungen ungefähr andauern. Als Grundlage der Messung diente die relativ komplexe Transformation zwischen Klassenmodell und Datenbankschema, welche vom QVT Standard vorgegeben wird. Eine Version der Transformation für die Medini-Engine liegt der Medini-Installation bei und wird an dieser Stelle genutzt. Die Tests wurden auf drei unterschiedlichen Rechnern jeweils unter Ubuntu Linux durchgeführt. Der schwächste für den Test verwendete Rechner, war ein Netbook, welches mit einem Intel Atom Prozessor N270 mit 1,6 GHz Taktfrequenz und 1 GB Hauptspeicher ausgerüstet war. Dieser Rechner wird im Folgenden als „Rechner A“ bezeichnet. Als Rechner mittlerer Leistungsfähigkeit stand ein Desktoprechner, bezeichnet als „Rechner B“, mit AMD Athlon 64 Prozessor 3200+ mit einer Taktfrequenz von 2,2 GHz und 1,8 GB Hauptspeicher bereit. Als dritter Rechner wurde ein Desktoprechner mit einem Intel Core2 Quad Q9450 Prozessor, der mit einer Taktrate von 2,67 GHz arbeitet und über vier GB RAM verfügt, verwendet. Dieser ist „Rechner C“. Auf allen drei Rechnern wurde ein Eclipse Ganymede SR1 in der Version für Java Entwicklung genutzt, das um die nötigen Plug-ins für das Visual QVT/R Framework ergänzt wurde.

Die Laufzeiten der T2M-Transformation werden maßgeblich durch die lange Zeit für das Nachschlagen von Typen in den Metamodel Registries bestimmt. Zwar wurde das Nachschlagen in der Xtend-Transformation mittels Cached Extensions realisiert, einmal gefundene Abbildungen von Typ-Bezeichner auf Typ zur Wiederverwendung werden auf diese Weise zwischengespeichert, aber dennoch reicht das einmalige Nachschlagen bereits für eine deutliche Verzögerung. Da die T2M-Transformation bei fehlerhaften Quelltext abgebrochen wird, wird dieser Fall für die T2M-Transformation nicht gesondert betrachtet.

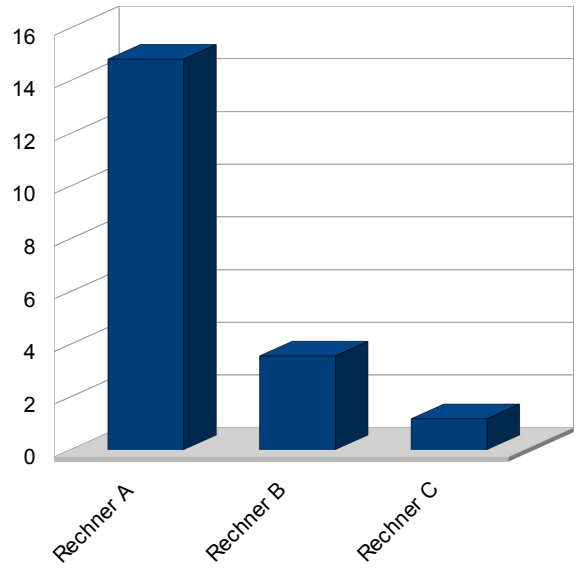


Abbildung 46: Dauer der T2M-Transformation in Sekunden

Die Zeit für die M2T-Transformation beinhaltet neben der eigentlichen Modell-zu-Text-Transformation auch die Dauer der Quelltextformatierung, der Quelltextvalidierung und der anschießenden Modellvalidierung. Da die Lokalisierung von Fehlern im graphischen Modell und die Anzeige der Fehler über die Modellvalidierung Auswirkungen auf die Laufzeit haben, wurden zwei Messungen für die M2T-Transformation durchgeführt. Eine ohne Fehler und eine, in der drei Fehler in das graphische Modell eingebaut wurden, die der textuelle Validator meldete und durch die Modellvalidierung im graphischen Editor angezeigt wurden.

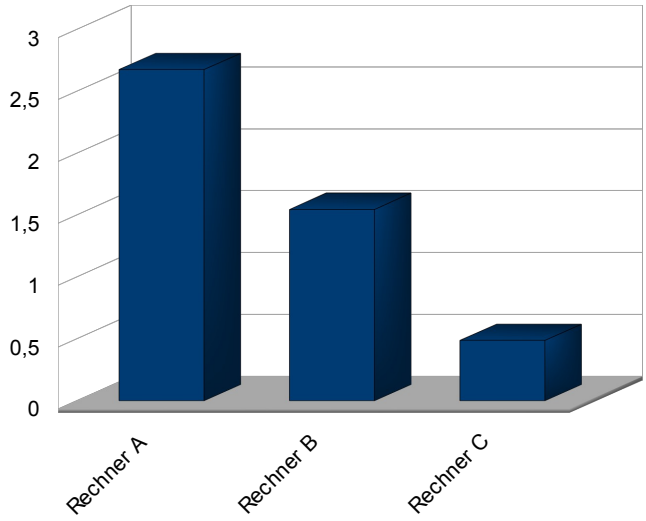


Abbildung 47: Dauer der M2T-Transformation in Sekunden

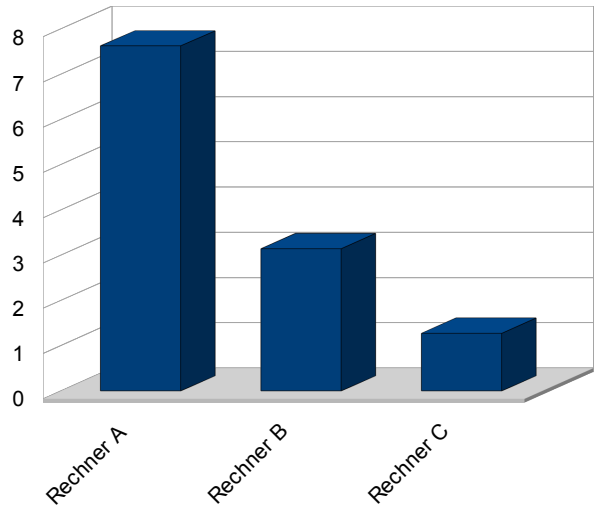


Abbildung 48 : Dauer der M2T-Transformation in Sekunden bei drei Fehlern

Wie deutlich zu erkennen ist, haben auftretende Fehler eine erhebliche Auswirkung auf die Dauer der M2T-Transformation. So kann ein fehlerfreies Modell schnell als solches erkannt werden,

müssen aber Fehler aus der textuellen Repräsentation im graphischen Editor dargestellt werden, so ist auf etwas schwächeren Rechnern mit einer Verzögerung bis zur Darstellung der Fehler zu rechnen.

Anzumerken ist, dass die M2T- und T2M-Aktivitäten mittels Eclipse-Jobs im Hintergrund ausgeführt werden und nicht die gesamte Entwicklungsumgebung für die Zeit der Transformationen blockieren. So wird der Nutzer zwar durch eine insgesamt langsamere Bedienung, eine verzögerte Darstellung von Fehlermeldungen und eine verzögerte Verfügbarkeit von textueller bzw. graphischer Syntax beeinträchtigt, kann aber weiterhin mit der Oberfläche arbeiten.

Für die T2M-Transformation muss erwähnt werden, dass das Nachschlagen von Metamodellelementen in Repositories derzeit nicht besonders performant implementiert ist. Es existieren daher noch einige Möglichkeiten, die Performance des Nachschlagens weiter zu verbessern. So kann eine intelligentere Auswahl der Repositories getroffen werden, in denen nach Modellelementen gesucht werden soll. Es könnte auch gespeichert werden, in welchen Metamodell-Paketen schon einmal gesucht wurde um bei weiteren Versuchen direkt auf die Pakete zuzugreifen, anstatt noch einmal alle Metamodell-Repositories zu durchsuchen. In Zukunft könnte das derzeit in Entstehung befindliche Projekt EMF Index [EMFI] bei dem Auffinden von Metamodellelementen helfen.

7.3 Fallbeispiel

Um die Praxistauglichkeit des graphischen Editors zu prüfen, wurden im Laufe der Entwicklung einige Beispieltransformationen entwickelt und auf Modelle angewandt. In diesem Abschnitt wird ein Beispiel vorgestellt, das eine praktische Relevanz hat, da es sich um eine Reimplementierung einer Transformation handelt, die ursprünglich in ATL [ATL] geschrieben wurde. Dieses wird bei der qme Software GmbH als Teil eines Frameworks zur Generierung von Grundgerüsten von EJB3-Anwendungen genutzt wird. Die Transformation bildet von UML-Klassenmodellen, die mit einem eigenem Profil markiert wurden, auf EJB3-Konfigurationsmodelle ab. Das Metamodell für die EJB3-Konfigurationsmodelle definiert EJB3 spezifische Informationen, die genutzt werden, um die UML-Modelle um zusätzliche Informationen zu ergänzen. Diese Konfigurationsmodelle können unabhängig vom UML Modell modifiziert und gegebenenfalls komplett ausgetauscht werden. Zu den Konfigurationsinformationen gehören Persistenzinformationen, wie Bezeichner im Datenbankschema und Datumsformate sowie Serviceinformationen, wie die Information, ob eine Klasse „Stateless“ oder „Stateful“ ist und ob Methoden auch entfernt aufgerufen werden können. Auf diese Weise liegen die EJB3 spezifischen Informationen lose gekoppelt in einem eigenen Modell. Um manuell gesetzte Informationen unter Retransformation beibehalten zu können, wurde in der ur-

sprünglichen Version eine weitere ATL-Transformation geschrieben, welche das bestehende EJB3-Konfigurationsmodell als zusätzliches Eingabemodell nutzte. Bestehende Informationen wurden bei dieser Transformation aus dem bestehenden EJB3-Konfigurationsmodell gelesen und in das neue Ausgabemodell übernommen. Der Aufwand für das Erstellen der Transformation hatte sich damit fast verdoppelt.

Die Funktionalität der ATL-Transformation ließ sich vollständig mittels QVT/R umsetzen. Die graphische Repräsentation der entstandenen QVT/R-Transformation befindet sich in Anhang A. Die neue Transformation ist direkt für Retransformation geeignet und hat sogar einen Vorteil gegenüber den Transformationen, die mittels ATL geschrieben wurden: Ändern sich Klassennamen im UML-Modell, so wird bei der QVT/R-Transformation der Name der bestehenden Konfiguration für die Klasse aktualisiert. Bei der ursprünglichen Transformation mittels ATL war es nicht möglich herauszufinden, ob sich Namen von Klassen geändert hatten, da die qualifizierten Namen als Identifikation für die Modellelemente genutzt werden mussten. Änderten sich die Namen der Klassen, so wurde die bestehende Konfiguration für diese Klasse, inklusive manueller Änderungen, gelöscht und ein neues Konfigurationselement für die Klasse mit neuem Namen mit Standardwerten angelegt. Da die QVT/R-Transformation das Zielmodell nur anpasst und nicht jedes Mal von neuem komplett generiert, ist eine wesentlich bessere Unterstützung für eine iterative Weiterentwicklung von Ausgangsmodellen gegeben.

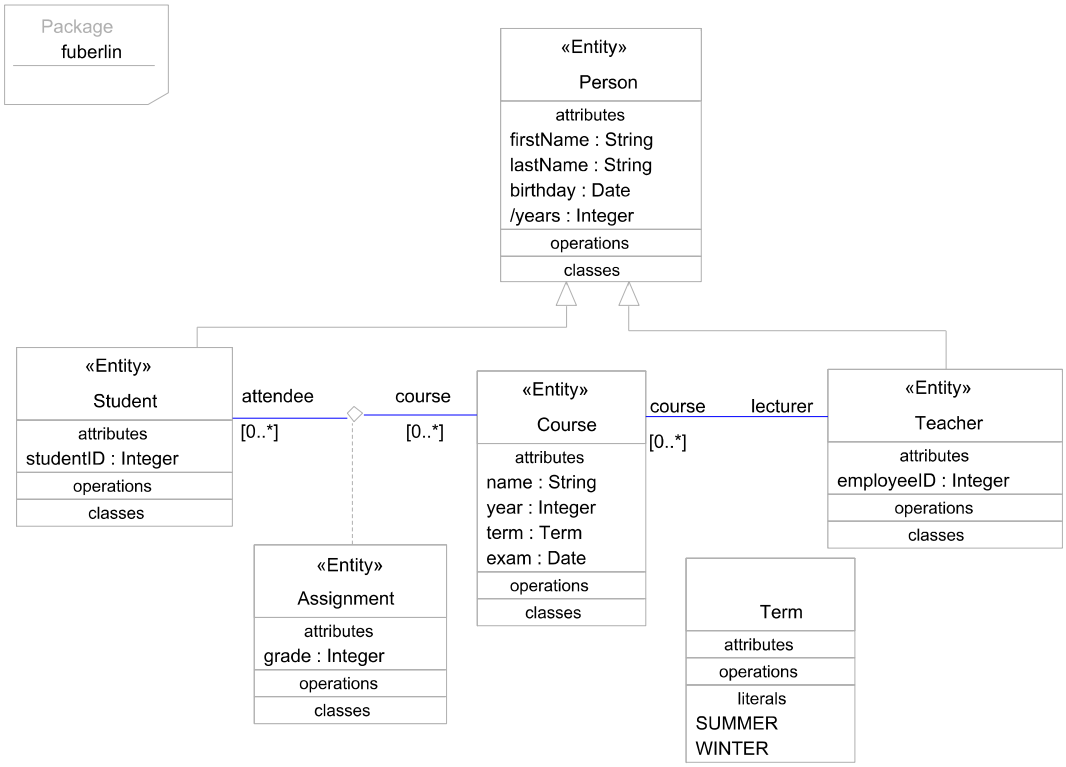


Abbildung 49: Beispiel Eingabemodell, Stereotypen an Attributen nicht sichtbar

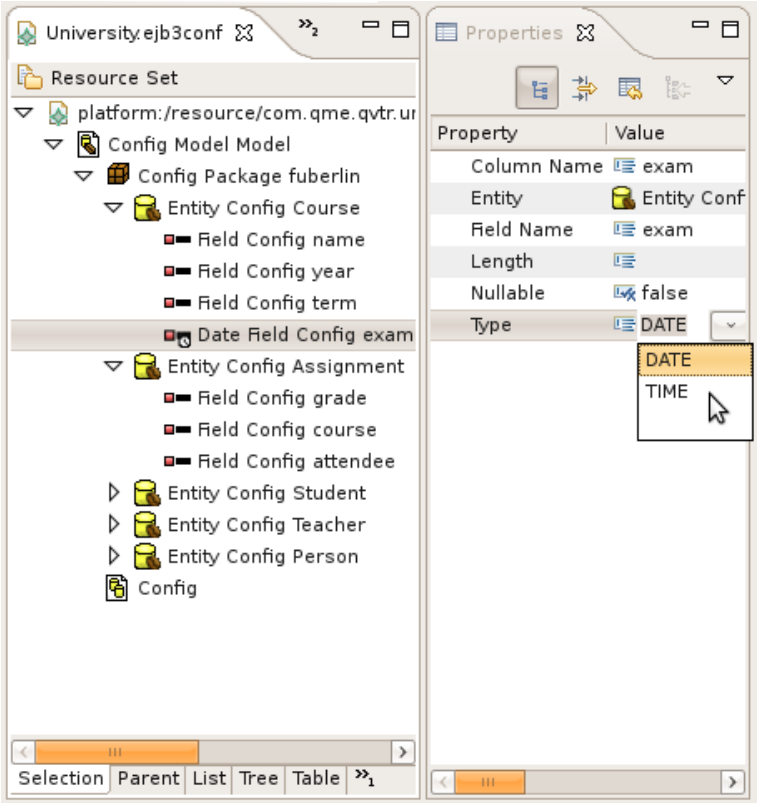


Abbildung 50: Beispiel Ausgabemodell

Die Funktion, Tracedaten in die Editorkomponente zu laden, hat geholfen Fehler aufzudecken, die während der Entwicklung entstanden. Die Relationen wurden nach und nach modelliert und immer nach dem Fertigstellen von Relationen, die eine bestimmte Funktionalität abdecken, wurde die Transformation ausgeführt und die Tracesdaten eingelesen. Dann wurde per Menü geprüft, ob die es die erwarteten Matches für die Domains und Templates auftragen. Auf diese Weise wurde z.B. aufgedeckt, dass in einem When-Block ein „not“ Operator fehlte, was dazu führte, dass es für eine Domain wesentlich mehr Matches gab, als erwartet.

Während der Modellierung stellte sich heraus, dass zwei Fehler besonders häufig beim Entwickeln von QVT/R-Transformationen auftraten. Zum einen war dies das Einführen von Relationen, ohne diese aufzurufen, zum anderen war es das Vergessen von Variablendeklarationen. Beide Fehler werden mittels des textuellen QVT/R-Validators ermittelt und im Editor angeprangert.

Anders als in Fällen, in denen die Modelle einen hohen Abstraktionsgrad von den generierten Artefakten haben oder eine große Menge von Infrastruktur Code generiert wird, der um manuelle Anteile ergänzt werden muss, wird mittels des Visual QVT/R Frameworks zwischen semantisch äquivalenten Repräsentationen transformiert. Es findet also keine Transformation zu einem technisch detaillierten Ziel statt. Daher lässt sich kein sinnvoller Vergleich zwischen herkömmlicher Entwicklung und Entwicklung mittels des graphischen Editors anstellen, bei dem z.B. überprüft

wird, wie viel Code manuell erstellt werden musste und wie viel Code generiert werden konnte. Aufgrund der semantischen Äquivalenz wird immer 100% des Quelltextes oder des Modells generiert, so dass keine manuellen Ergänzungen notwendig sind.

8 Schlusswort

In den Unterabschnitten dieses Kapitels wird ein Erfahrungsbericht über die Arbeit mit den genutzten Technologien und über die Anwendung des entwickelten Frameworks gegeben. Des Weiteren werden Wege gezeigt, das erstellte Framework weiter auszubauen und um Funktionalität zu ergänzen. Zuletzt wird zusammengefasst, was mit dieser Arbeit erreicht wurde.

8.1 Erfahrungen

Die gesammelten Erfahrungen mit Technologien und mit dem QVT-Standard werden in den folgenden Unterkapiteln beleuchtet.

8.1.1 Erfahrungen mit den verwendeten Technologien

Der Umgang mit GMF stellte sich, zumindest zu Anfang der Implementierung, als komplizierter heraus als erwartet. Aufgrund der relativ starren Regeln, die vorschreiben, wie graphische Syntaxelemente mit Metamodellelementen verknüpft werden, konnte das standardisierte Metamodell nicht als Basis für den Editor herangezogen werden. Infolgedessen musste ein eigenes Metamodell entwickelt werden, das die Voraussetzungen für das Erstellen eines Editors mit der Standardisierten Syntax mit GMF erfüllte. Die ersten Versuche, einen Editor für einige Syntaxelemente (die von den strukturellen Beschränkungen von GMF nicht betroffen waren) auf Basis des Metamodells des Eclipse QVT Declarative zu generieren, erzeugten lediglich Quelltexte, die über und über mit Syntaxfehlern behaftet waren. Auch nach dem Wechsel auf das eigene Metamodell gab es etliche Probleme mit GMF. Bei Regenerierung nach kleinen Änderungen am Metamodell wurde oft fehlerhafter Quelltext erzeugt. Häufig musste der komplette, bereits manuell erweiterte, Quelltext des Editors gelöscht und alle Quelltexte neu generiert werden. Die manuellen Änderungen mussten nachträglich wieder eingefügt werden. Ein iteratives Vorgehen beim Erstellen des Metamodells und des darauf aufbauenden Editors gestaltete sich daher recht mühselig.

Die Arbeit mit den Sprachen des openArchitectureWare Frameworks erwies sich dagegen wesentlich einfacher. Allerdings hatten die Editoren für Xpand und Xtend oft das Problem, Metamodell-Daten nicht zu finden, und markierten daher Code als fehlerhaft, obwohl es bei der Ausführung der Skripte zu keinem Fehler kam. Dadurch stand in den meisten Fällen keine Code-Vervollständigung zur Verfügung.

8.1.2 Erfahrungen mit dem QVT Standard

Die konkrete graphische Syntax von QVT Relations ist im Standard relativ kurz behandelt und wirft einige Fragen auf, welche die Abbildung der graphischen Syntaxelemente auf die abstrakte und textuelle Syntax betreffen. In Kapitel 1.4.2.B wird beschrieben, welche Interpretation bei unklaren Definitionen für die Implementierung des Visual QVT/R Framework gewählt wurde. Wünschenswert wäre eine formalere Definition der Abbildung von graphischer Syntax auf die abstrakte Syntax. In der derzeitigen Version des Standards ist die Abbildung teilweise durch Beispiele festgelegt, die keine vollständige Definition ergeben. Die existierenden Unklarheiten in der Spezifikation, welche die Semantik der Transformationen betreffen (siehe [IKVProb] und [QVT11TF]), spielten weder bei der Implementierung des Editors noch in der Entwicklung von Transformationen im Rahmen der Diplomarbeit eine Rolle.

8.2 Erweiterungsmöglichkeiten des Frameworks

Anknüpfend an diese Arbeit ergeben sich weitere Möglichkeiten zur Weiterentwicklung des Ansatzes und der Anbindung weiterer Konzepte und Technologien.

8.2.1 Weiterführende Konzepte

Eine Herausforderung wird sicherlich das Finden einer noch besseren und intuitiven Visualisierung von Trace-Daten sein. Um die Transformationen anschaulich zu machen, könnte z.B. ein Ansatz gewählt werden, der die Wiederverwendung von Modell-Editoren für die Parametermodelle beinhaltet. Wenn Modelleditor und Transformationseditor gleichzeitig geöffnet sind und die Trace-Datei geladen ist, könnte ein Klick auf ein Element im Modell-Editor die Elemente im Transformationsmodell markieren (z.B. dunkler einfärben), dessen Variablen laut Trace dieses Modellelement als Wert angenommen haben. Ein anderer Ansatz könnte darin bestehen, Trace Daten im dreidimensionalen Raum darzustellen. Visuellen Editoren für Eingabemodelle, QVT-Relationen und Ausgabemodelle könnten in Ebenen hintereinander stehen und die Trace-Daten könnten als Verbindungen (Linien) zwischen den Modellen visualisiert werden. Dies wäre eine Erweiterung des Konzeptes, welches von Pilgrim, Vanhooff, Schulz-Gerlach und Berbers [Pilg08] vorgestellt wurde. Da die Visualisierung der Transformationen eingebunden ist, können so Ursachen fehlerhafter Transformationen schnell erkannt werden, da offensichtlich ist welche Regel für welche Trace Verbindungen zwischen Ein- und Ausgabemodell verantwortlich ist.

Ein weitere Möglichkeit zur Weiterführung wäre die Entwicklung eines Debuggers, der zu einem bestimmten Zeitpunkt einer Transformation die genaue Belegung der Domains und Templates dar-

stellt. Auch diese Belegung ließe sich dreidimensional darstellen. Eine Möglichkeit, die sich aus dem Konzept eines Debuggers im graphischen Modell einer Transformation ergibt, wäre die Möglichkeit, Break-Point Conditions direkt an den (und im Kontext dieser) Domains bzw. Templates zu definieren. Die Conditions könnten z.B. graphisch wie OCL-Constraints an Modellelemente der Transformation gehängt werden.

8.2.2 Interoperabilität

Durch das modulare Konzept des Visual QVT/R Frameworks ergeben sich Möglichkeiten zum Austausch von Teilen oder Hinzufügen von neuen Teilen des Frameworks. So kann die Technologie zum Erstellen der textuellen Repräsentation aus dem graphischen Modell ausgetauscht werden. Es ist damit auch möglich, Plug-ins zu entwickeln, die das visuelle Modell in eine ganz andere textuellen Transformationssprache überführen. Da es grundsätzlich möglich ist, QVT/R nach QVT/C (nach Definition im QVT Standard) zu transformieren, wäre eine Übersetzung in diese Sprache theoretisch möglich. Eine Überführung in QVT/O [Rom08] oder TGG [Gre07] ist ebenfalls theoretisch möglich. Allerdings dürfte dies keine leichte Aufgabe für eine M2T-Komponente sein, da OCL-Ausdrücke im visuellen Modell nur in textueller Form und nicht als Modellelemente vorliegen. Auch der QVT/R-Parser bzw. die Technologie zum Überführen der textuellen Syntax in graphische Modelle ist austauschbar. Damit ließen sich Sprachen mit ähnlicher Konzeption wie QVT/R visualisieren. Sogar die Technologie des graphischen Editors ließe sich unter Beibehaltung des VQVT/R-Metamodells austauschen.

In der sich gerade in Arbeit befindlichen Diplomarbeit von Patrick Zwickel, TU Wien, mit dem Titel „A Graphical Debugger for QVT Relations“ trägt [QVTDebug], wird nach Angabe von Herrn Zwickel an einer Visualisierung der operationalen Semantik der Befehle hinter der deklarativen Syntax gearbeitet. Dabei soll ein „Fluss“ von Informationen von einem Quell- zu einem Ziel-Modell dargestellt werden. Die Visualisierung soll das Erkennen typischer Fehler bei der Erstellung von Transformationen mittels QVT Relations erleichtern. Der Ansatz wird in dem Artikel [Kru09] von Krusel et al. vorgestellt. Es ist denkbar, an diese Arbeit anknüpfend eine Anbindung des graphischen Editors an diese Implementierung zu erstellen.

8.3 Fazit

Um ein Fazit über das Erreichte abzugeben, wird an dieser Stelle überprüft, ob die in dieser Diplomarbeit in Kapitel 2 formulierten Ziele erfüllt werden konnten.

Wie gewünscht, wird die komplette konkrete graphische Syntax, wie sie im QVT/R Standard fest-

gelegt ist, implementiert und kann im Editor modelliert werden. QVT/R-Transformationen lassen sich mit Ausnahme des Import-Statements vollständig im Editor beschreiben, auch die Elemente der Sprache, für die keine graphische Syntax standardisiert ist, können modelliert werden. Der Editor wurde auf Basis von EMF entwickelt, so dass Ecore- und EMOF-Metamodelle unterstützt werden. Mittels GMF wurde die Editorkomponente mit modellgetriebenen Technologien entwickelt, was eine weitere Anforderung erfüllt. Der Editor schränkt bei der Modellierung die Editiermöglichkeiten in den meisten Fällen so ein, dass nur gültige Relationen modelliert werden können. Dies war eines der erklärten Ziele der Diplomarbeit. Auch der geforderte Zugriff auf Trace-Daten aus dem graphischen Editor heraus ist möglich. Das Ziel Live-Validation im Editor zu unterstützen, konnte aus technischen Gründen nicht vollständig umgesetzt werden. Die mittels GMF realisierten Live-Validierungen zeigen leider keine Problem-Dekoratoren an den Elementen im graphischen Editor. Um dieses Problem zu relativieren, wird die Validierung des kompletten Modells beim Speichern des Modells im graphischen Editor angestoßen.

Das Ziel, die graphische Syntax möglichst automatisiert in die textuellen Syntax zu übersetzen, wurde mit der M2T-Transformation und der Visual QVT/R Nature für Eclipse-Projekte erreicht. Die T2M-Transformation basiert auf dem Parser der Medini-Engine und unterstützt daher nur die Syntaxelemente, die auch Medini unterstützt. Daher ist kein vollständiger Roundtrip möglich. Ein grundsätzliches Problem für Roundtripfähigkeit stellt auch das Not Template dar. Generell würden nur zusätzliche Markierungen im generierten Quelltext einen Roundtrip ermöglichen. Werden allerdings Tansformationen für die Medini-Engine entwickelt, spielen die nicht unterstützten Syntaxelemente ohnehin keine Rolle. Da das M2T-Feature als Machbarkeitsstudie geplant war, kann an dieser Stelle eine grundsätzliche Machbarkeit attestiert werden.

Die Probleme der unvollständigen Unterstützung des QVT/R-Standards durch Medini sind als wenig schwerwiegend zu betrachten. In den Transformationen, die im Laufe der Diplomarbeit entwickelt wurden, spielten die von Medini nicht unterstützten Collection-Templates keine Rolle. Die fehlende Unterstützung von Constraints auf Templates bzw. Domains ließen sich durch OCL-Ausdrücke im When-Block umgehen. Als Ersatz für Default Values konnten Queries eingeführt werden, die zwischen bestehenden Wert und einem Standard-Wert auswählen können. Dies erforderte in einigen Fällen einen etwas höheren Aufwand in der Notation, war aber kein kritischer Faktor für das Erstellen der Transformationen. Auch die populäre Beispieltransformation, die von einem Klassen-Metamodell auf ein Metamodell für Datenbankschemta abbildet, lässt sich vollständig mittels Medini umsetzen. Es ist zu vermuten, dass das Collection Template kein essentielles Element für die Entwicklung von Modelltransformationen mittels QVT/R ist.

Um Fehler, die eine QVT/R-Engine meldet, in den graphischen Editor zu übernehmen, wurde das Problem-Metamodell eingeführt. Die mit dem Framework gelieferte Anbindung an die Medini-Engine bietet allerdings nur eine funktionierende Lokalisierung der Fehler für statische Fehler. Laufzeitfehler, die die Medini-Engine meldet werden immer im Kontext der gesamten Transformation angezeigt. Dies ist aber durch den Mechanismus bedingt, mit dem Medini Laufzeitfehler meldet, und ist keine grundsätzliche Beschränkung des Visual QVT/R Frameworks.

Die Anbindug der Medini QVT/R-Engine an die openArchitectureWare Workflow-Engine erfüllt ein weiteres Kriterium, welches als Ziel der Arbeit festgehalten wurde. Die Workflow-Komponenten lassen sich auch unabhängig vom Visual QVT/R Framework nutzen und bieten damit eine gute Anbindung von QVT/R an bestehende Projekte, deren Transformations-Abläufe mit der oAW Workflow-Engine gesteuert werden.

Die Architektur des Frameworks ist so gestaltet, dass es die Anforderung erfüllt, alle von einer QVT/R-Enginge abhängigen Teile austauschbar über Schnittstellen anzubieten. Dazu wurden Eclipse Extension Points und Java-Schnittstellen verwendet.

Damit sind die Ziele bis auf kleinere Abstriche erreicht und es steht eine funktionierende Implementierung der konkreten graphische Syntax von QVT/R bereit. Das Visual QVT/R Framework bietet außerdem Perspektiven für die Anbindung weiterer Technologien und Konzepte, um den Umgang mit Modelltransformationen weiter zu vereinfachen.

9 Danksagungen

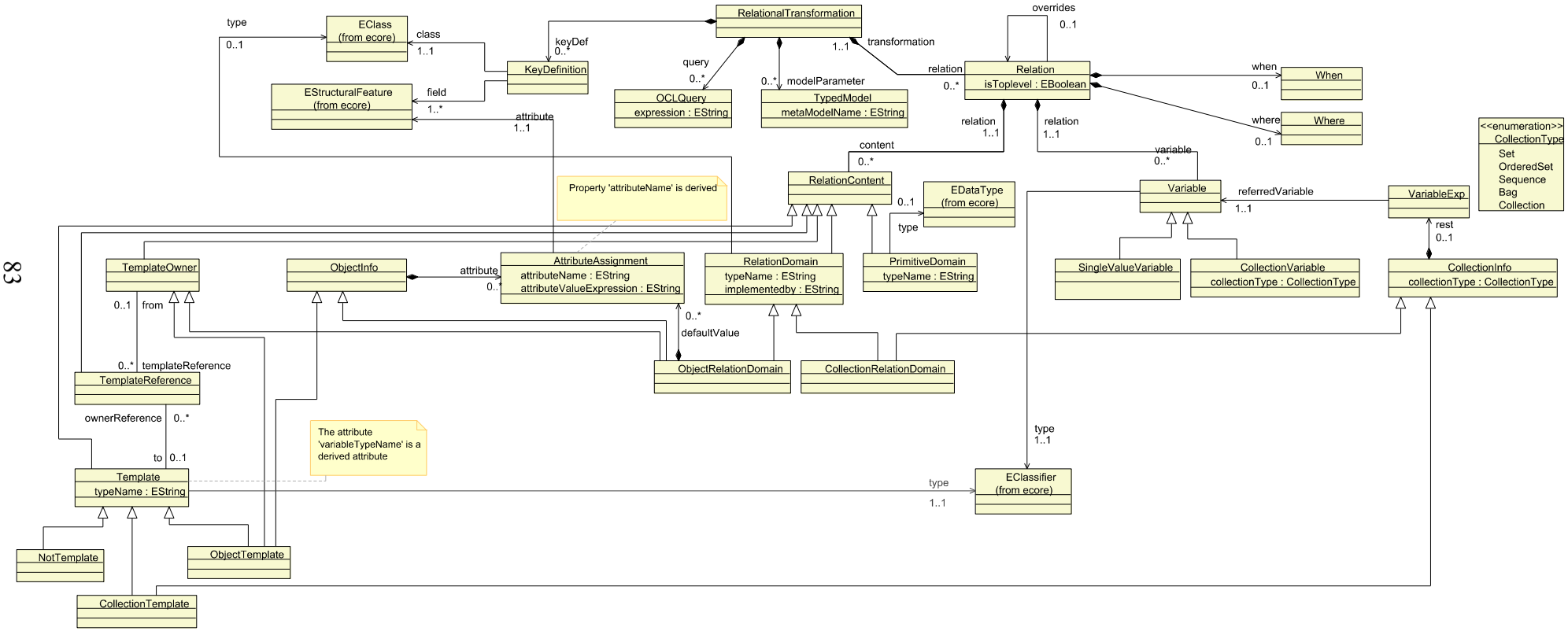
Ein Dank geht an meine Betreuer für die Unterstützung und Anregungen, die sie mir gegeben haben. Ich danke den Herren Roland Petrasch und Florian Fieber der qme Software GmbH dafür, dass sie mich für den Zeitraum der Diplomarbeit angestellt und mir die beratend beiseite gestanden haben. Auch Herrn Siegfried Nolte sei an dieser Stelle gedankt, der mir sein Buch „QVT Relations“ [Nol09] bereits vor Veröffentlichung zur Verfügung stellte; es wird erst nach Abgabe dieser Diplomarbeit erscheinen.

10 Anhang

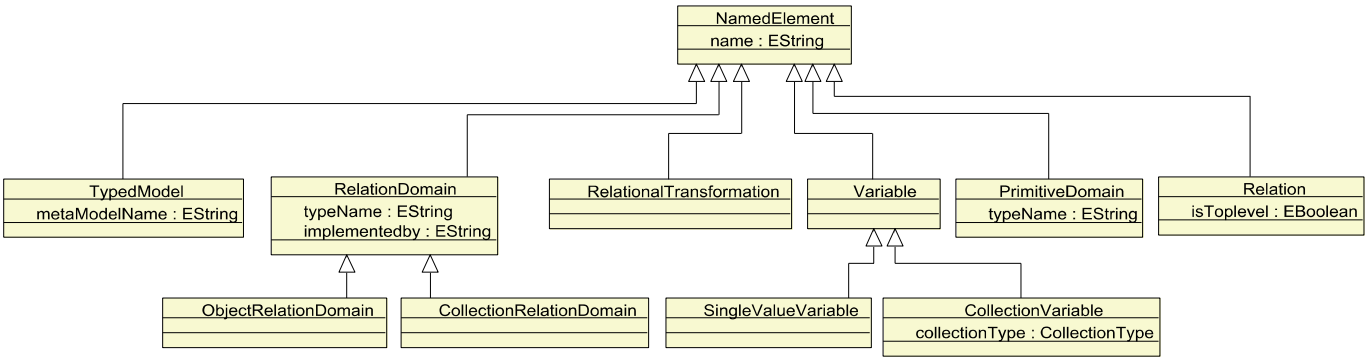
A VQVT/R-Metamodell

In den folgenden Unterabschnitten wird das Metamodell des Visual QVT/R Frameworks unter verschiedenen Gesichtspunkten gezeigt. Da das Metmodell sehr umfangreich ist, wurde auf ein Diagramm verzichtet, dass alle Elemente und Referenzen des Modells beinhaltet.

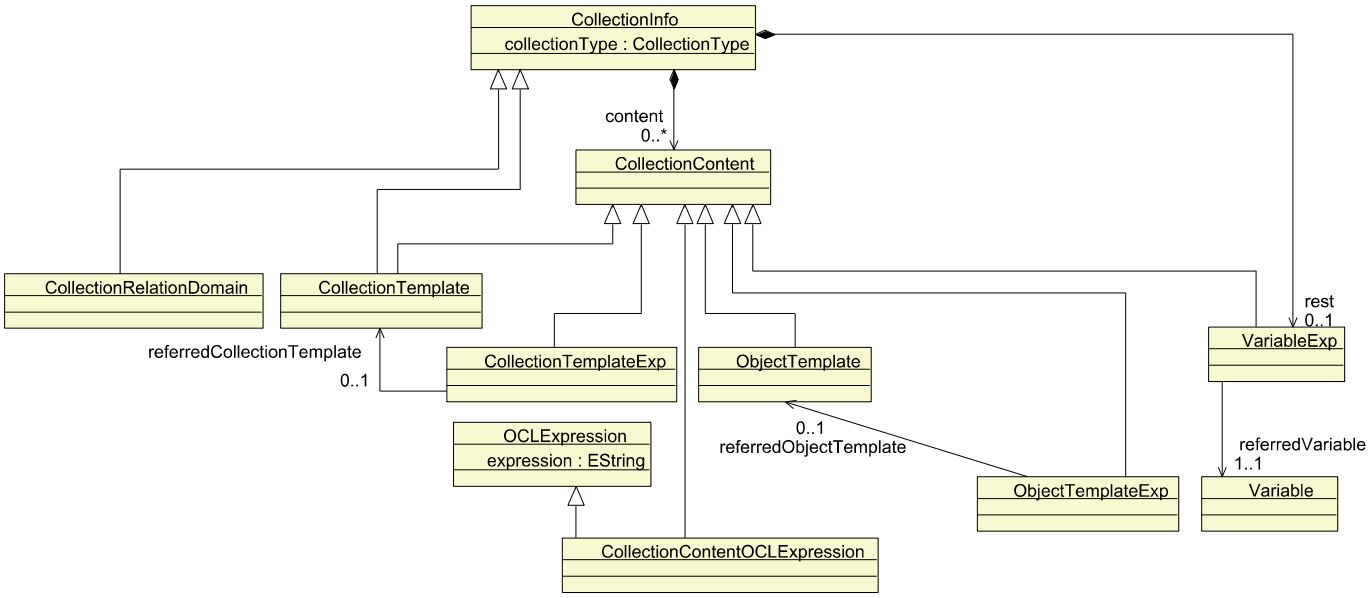
A.I Überblick


RelationalTransformation
EClass (from ecore)
KeyDefinition
EStructuralFeature (from ecore)
OCLQuery
expression : EString
TypedModel
metaModelName : EString
Relation
isToplevel : EBoolean
When
Where
RelationContent
EDataType (from ecore)
Variable
VariableExp
<<enumeration>>
CollectionType
Set
OrderedSet
Sequence
Bag
Collection
Property 'attributeName' is derived
AttributeAssignment
attributeName : EString
attributeValueExpression : EString
RelationDomain
typeName : EString
implementedby : EString
PrimitiveDomain
typeName : EString
SingleValueVariable
CollectionVariable
collectionType : CollectionType
CollectionInfo
collectionType : CollectionType
TemplateOwner
ObjectInfo
TemplateReference
ObjectRelationDomain
CollectionRelationDomain
The attribute 'variableTypeName' is a derived attribute
Template
typeName : EString
EClassifier (from ecore)
NotTemplate
ObjectTemplate
CollectionTemplate
type
class
keyDef
query
modelParameter
transformation
overrides
relation
when
where
field
attribute
content
variable
referredVariable
rest
defaultValue
from
templateReference
ownerReference
to

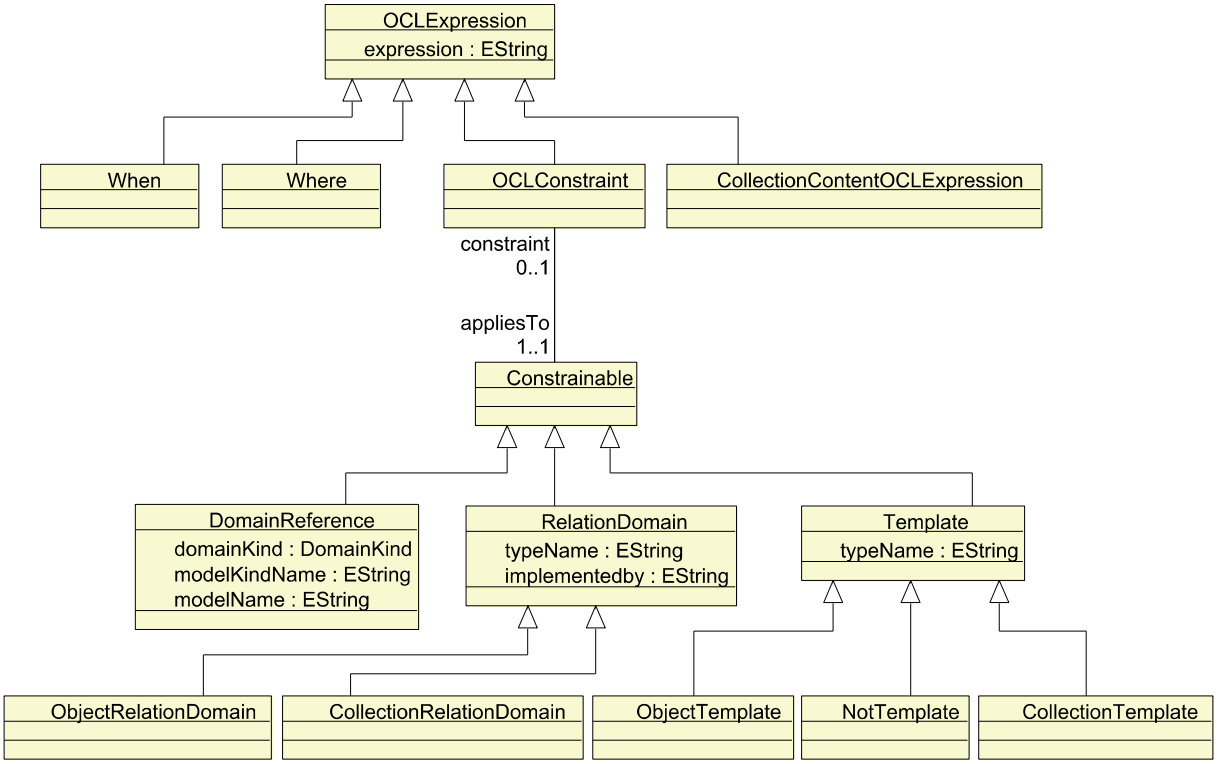
A.II Named Elements



A.III Collection Content

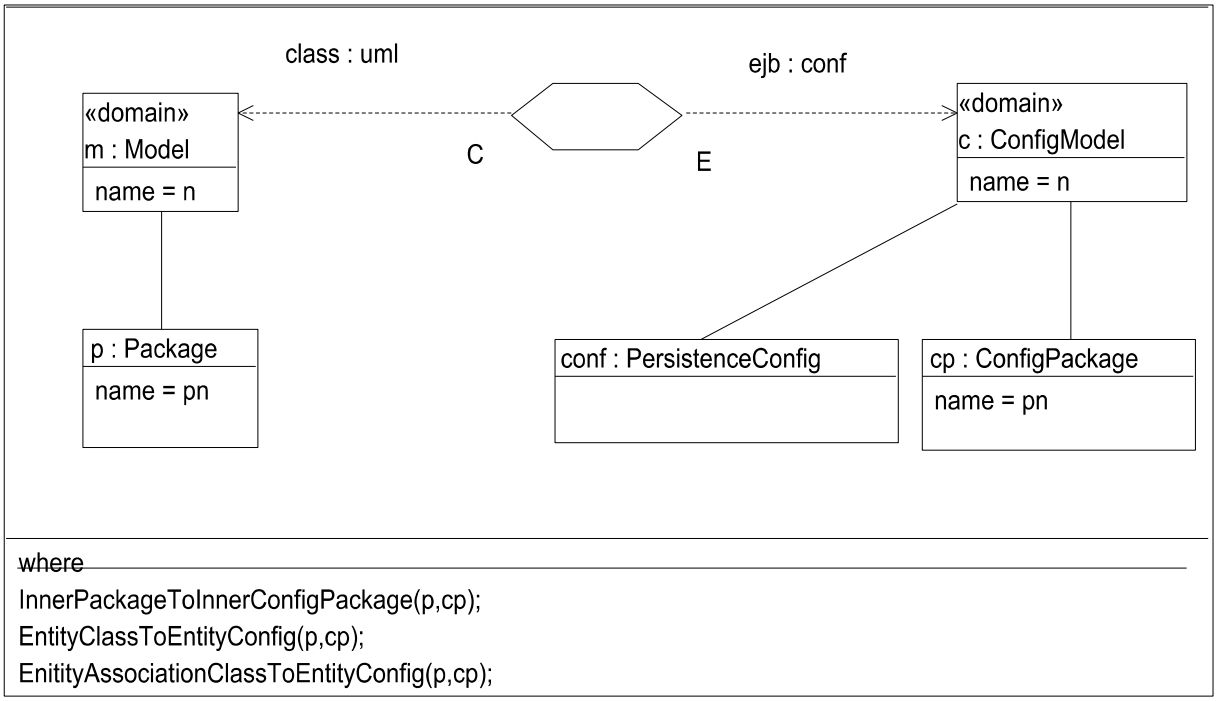


A.IV OCL Expressions

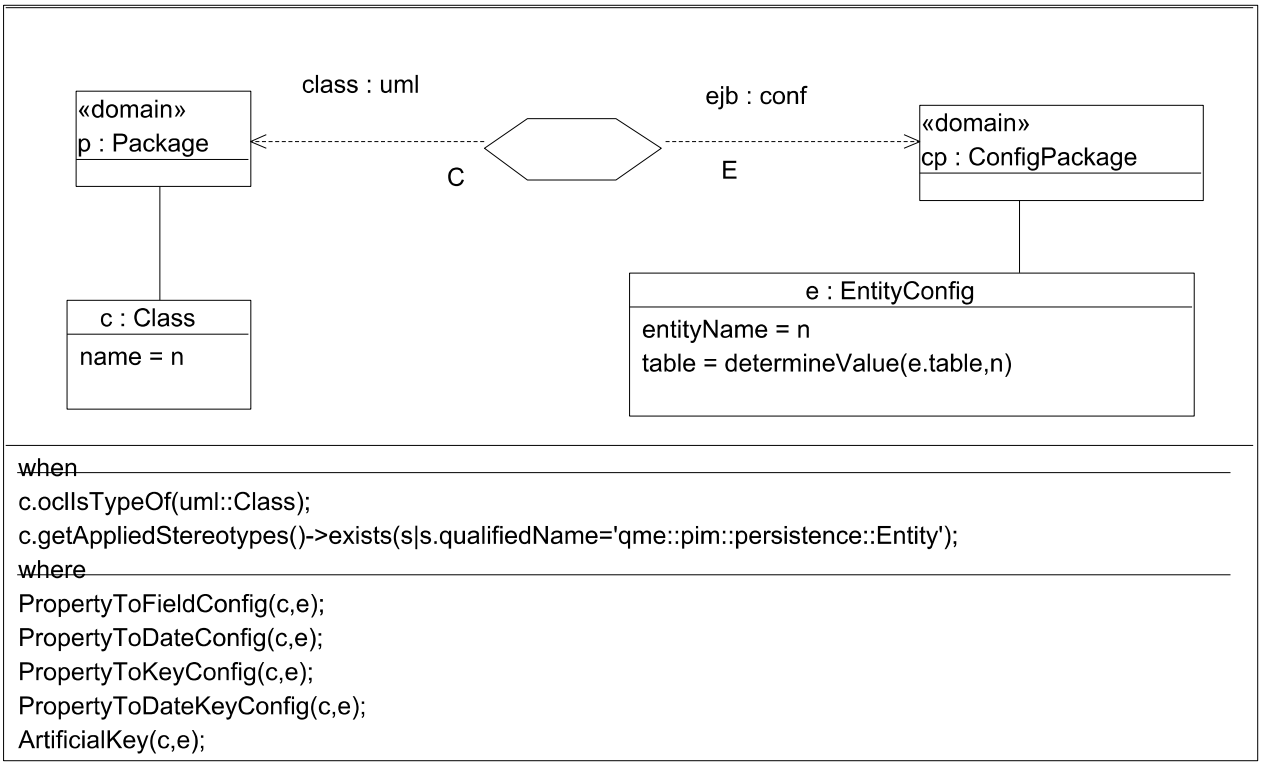


B Beispieltransformation

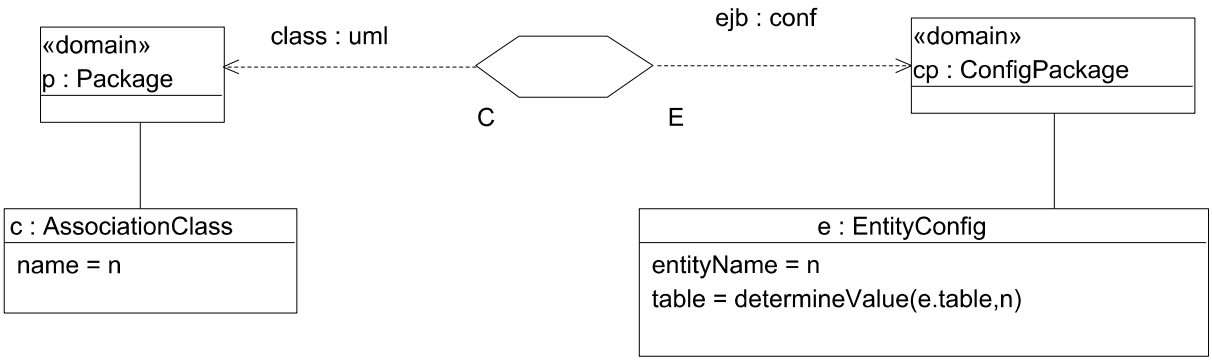
RootPackageToModel



EntityClassToEntityConfig



EnitityAssociationClassToEntityConfig



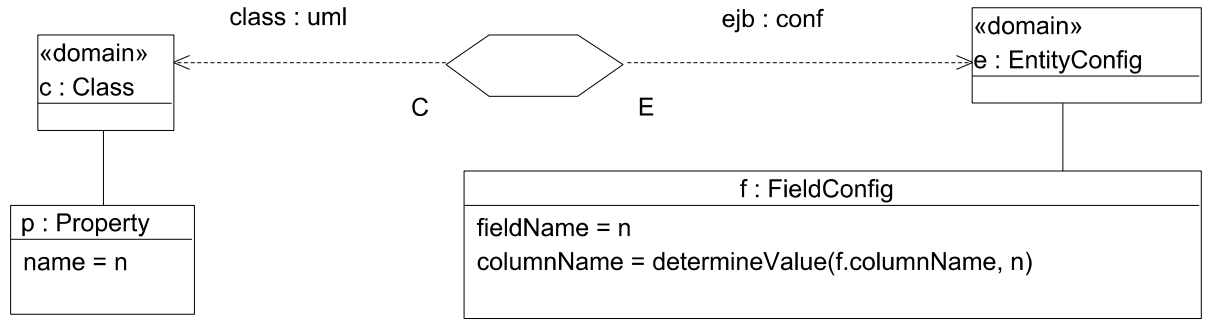
when

```
c.getAppliedStereotypes()->exists(s|s.qualifiedName='qme::pim::persistence::Entity');
```

where

```
PropertyToFieldConfig(c,e);
PropertyToDateConfig(c,e);
OwnedEndToFieldConfig(c,e);
PropertyToKeyConfig(c,e);
ArtificialKey(c,e);
```

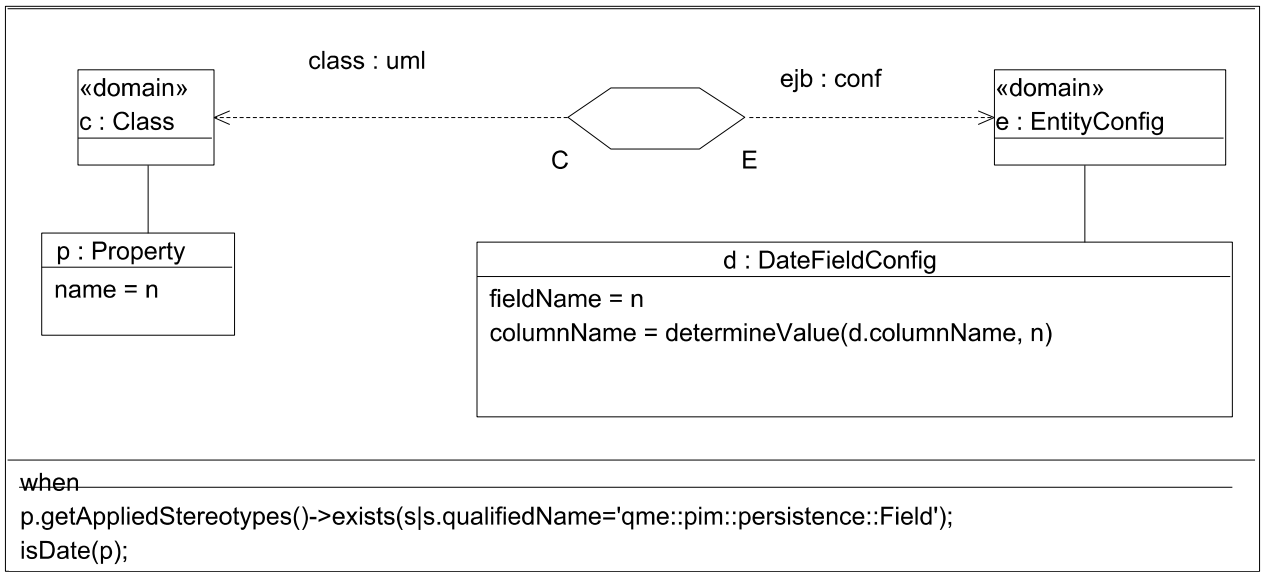
PropertyToFieldConfig



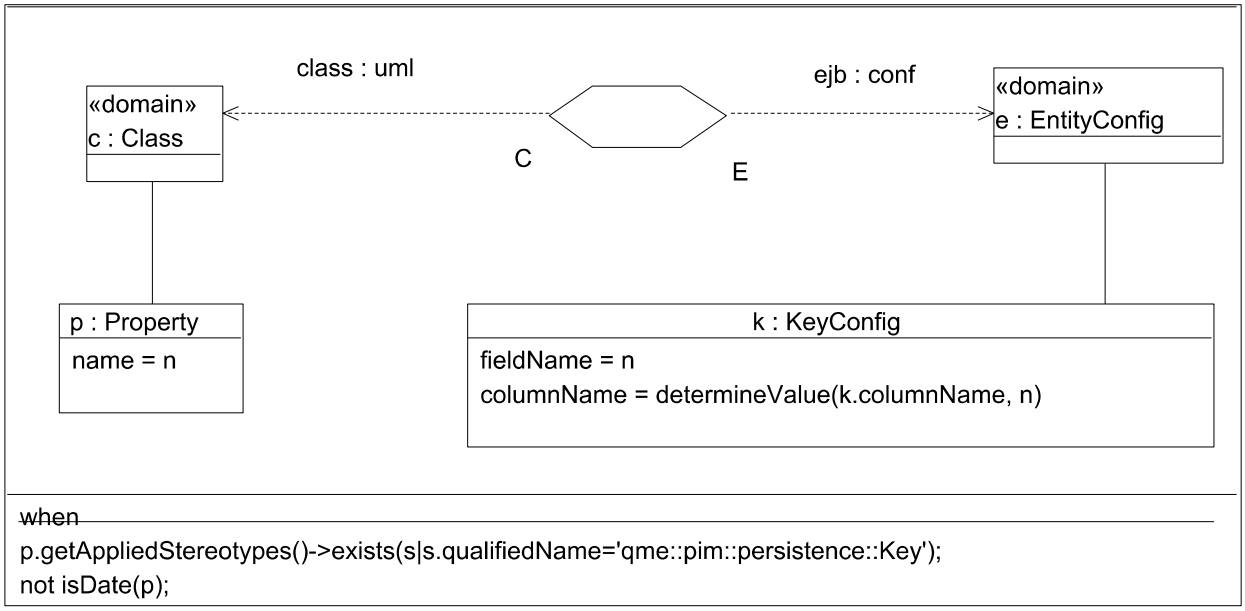
when

```
p.getAppliedStereotypes()->exists(s|s.qualifiedName='qme::pim::persistence::Field') or
p.type.getAppliedStereotypes()->exists(s|
s.qualifiedName='qme::pim::persistence::Entity');
not isDate(p);
```

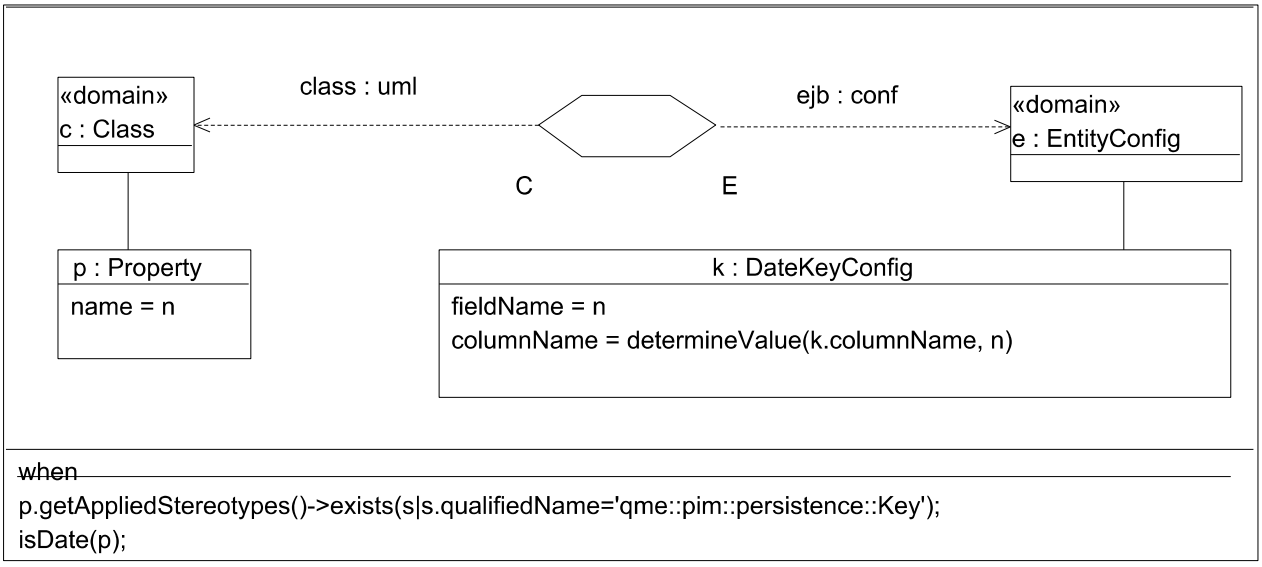
PropertyToDateConfig



PropertyToKeyConfig



PropertyToDateKeyConfig



```
1  query paramNames (o : uml::Operation) : OrderedSet(String) {
2    o.ownedParameter->collect(p : uml::Parameter | if
3          p.type.oclIsUndefined() then
4          'void'
5    else
6          p.type.qualifiedName
7    endif)->asOrderedSet()
8  }
9
10  query isDate (p : uml::Property) : Boolean {
11   p.type.oclIsTypeOf(uml::DataType) and p.type.name = 'Date'
12  }
13
14  query determineValue (value : String, defaultValue : String) : String {
15   if value.oclIsUndefined() then
16         defaultValue
17   else
18         value
19   endif
20  }
21
22  query allAttributesInclSuper (c : uml::Class) :
23   OrderedSet(uml::Property) {
24   if c.superClass->size() > 0 then
25         c.ownedAttribute->union(c.superClass->collect(s : uml::Class |
26          allAttributesInclSuper(s))->flatten()->asOrderedSet())
27   else
28         c.ownedAttribute
29   endif
30  }
```

C Abkürzungsverzeichnis

MDSD: Model Driven Software Development....7
M2M: Model-To-Model....7
M2T: Model-To-Text....7
DSL: Domain Specific Language....8
AC-MDSD: Architecture Centric MDSD....8
OMG: Object Management Group....9
MDA: Model Driven Architecture....9
OMG: Object Management Group....9
MOF: MetaObject Facility....10
UML: Unified Modeling Language....10
OCL: Object Constraint Language....10
XMI: XML Metadata Interchange....10
QVT: Query/View/Transformation....10
MOFM2T: MOF Model to Text Transformation Language....10
QVT/C: Query View Transformation Core....10
QVT/O: Query View Transformation Operational Mappings....10
QVT/R: Query View Transformation Relations....10
JVM: Java Virtual Machine....11
TGG: Triple Graph Grammar....21
EMF: Eclipse Modeling Framework....25
VQVT/R: Visual QVT/R....27
GMF: Graphical Modeling Framework....34
oAW: openArchitectureWare....35

D Abbildungsverzeichnis

Abbildung 1: Zusammenhang der QVT-Sprachen (aus der QVT-Spezifikation).....11
Abbildung 2: Metamodell „directory“.....12
Abbildung 3: Metamodell „container“.....12
Abbildung 4: Belegung für Transformationsparameter co.....16
Abbildung 5: Belegung für Transformationsparameter di.....16
Abbildung 6: Graphische Syntax für Relations.....17
Abbildung 7: Graphische Syntax für Relation Middle (aus der QVT-Spezifikation).....17
Abbildung 8: Graphische Syntax für Domains (aus der QVT-Spezifikation).....18
Abbildung 9: Graphische Syntax für Object Templates (aus der QVT-Spezifikation).....18
Abbildung 10: Graphische Syntax für Collection Templates (aus der QVT-Spezifikation).....19
Abbildung 11: Graphische Syntax für Not Templates (aus der QVT-Spezifikation).....19
Abbildung 12: Graphische Syntax für OCL Constraints (aus der QVT-Spezifikation).....20
Abbildung 13: Änderung im Directory Modell.....21
Abbildung 14: Änderung im Container Modell.....21
Abbildung 15: Auswirkung der Validierung bei nur einer Domäne.....28
Abbildung 16: Auswirkung des Constraints, der falsche Verbindungen unterbindet.....28
Abbildung 17: Definition der Validierung auf gesetzten Typ einer Domain.....29
Abbildung 18: Auswirkung der Validierung auf gesetzten Typ von Domains.....30
Abbildung 19: Tippfehler, der durch textuellen Validator aufgedeckt wurde.....31
Abbildung 20: Nicht aufgerufene Relation.....31
Abbildung 21: Metamodell Ausschnitt für Relation.....38
Abbildung 22: Beispiel Relation entsprechend dem Metamodell gefärbt.....38
Abbildung 23: Abhängigkeiten zwischen den Features.....42
Abbildung 24: Lose Kopplung über Extension Points.....42
Abbildung 25: T2M Transformation im Detail.....44
Abbildung 26: M2T Transformation im Detail.....45
Abbildung 27: Aktivitäten zur Validierung.....46
Abbildung 28: Konfigurations-Metamodell.....48
Abbildung 29: Problem Metamodell.....48
Abbildung 30: Beispiel für ein Not Template.....51
Abbildung 31: Beispiel Relation.....55
Abbildung 32: Ausschnitt aus Trace Metamodell.....56

Abbildung 33: Visual QVT/R Editor....61
Abbildung 34: Auswahl des Metamodells....62
Abbildung 35: Auswahl aus den angelegten TypedModels....62
Abbildung 36: Anlegen eines neuen Schlüssels über die Outline-View....63
Abbildung 37: Auswahl von Schlüsselattributen....63
Abbildung 38: Angabe einer Abbildung eines Namens auf ein Metamodell in der Konfiguration...64
Abbildung 39: Start der T2M Transformation über das Kontextmenü....65
Abbildung 40: Start der M2T Transformation über das Hauptmenü....65
Abbildung 41: Hinzufügen der Visual QVT/R Nature zu einem Projekt....66
Abbildung 42: Builder-Einstellungen in den Projekteigenschaften....66
Abbildung 43: Trace Dekorator über einer Relation....67
Abbildung 44: Gefilterter Trace-Editor nach Klick auf Dekorator....68
Abbildung 45: Tracedaten Zugriff aus dem Kontextmenü eines Templates....68
Abbildung 46: Dauer der T2M-Transformation in Sekunden....71
Abbildung 47: Dauer der M2T-Transformation in Sekunden....72
Abbildung 48 : Dauer der M2T-Transformation in Sekunden bei drei Fehlern....72
Abbildung 49: Beispiel Eingabemodell, Stereotypen an Attributen nicht sichtbar....74
Abbildung 50: Beispiel Ausgabemodell....75

E Literaturverzeichnis

Pet06: Roland Petrasch, Oliver Meimberg: *Model-Driven Architecture: Eine praxisorientierte Einführung in die MDA*. Dpunkt Verlag, 2006

Mosc05: Marco Mosconi: *Preserving the Separation of Aspects through all Abstraction Levels in Model-Driven Software Development*. In: *First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD*, 2005

Piet07: Georg Pietrek, Jens Trompeter, Benedikt Niehues, Thorsten Kamann, Boris Holzer, Michael Kloss, Karsten Thoms, Juan Carlos Flores Beltran, Steffen Mork : *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis*. entwickler.press, 2007

Völt05: Markus Völter, Thomas Stahl: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, 2005

Fran03: David S. Frankel: *Model Driven Architecture. Applying MDA to Enterprise Computing*. Wiley, 2003

MDA: OMG MDA, *http://www.omg.org/mda/* (Stand: 29.05.2009)

MDAG: MDA Guide, *http://www.omg.org/cgi-bin/doc?omg/03-06-01* (Stand: 29.05.2009)

MOF: Meta Object Facility, *http://www.omg.org/mof/* (Stand: 29.05.2009)

UML: Unified Modeling Language, *http://www.uml.org/* (Stand: 29.05.2009)

OCL: Object Constraint Language, *http://www.omg.org/spec/OCL/index.htm* (Stand: 29.05.2009)

XMI: XML Metadata Interchange, *http://www.omg.org/spec/XMI/index.htm/* (Stand: 29.05.2009)

QVT: Query/View/Transformation, *http://www.omg.org/spec/QVT/index.htm* (Stand: 29.05.2009)

MM2T: MOF Model to Text Transformation Language, *http://www.omg.org/spec/MOFM2T/index.htm* (Stand: 29.05.2009)

Mell04: Stephen J. Mellor, Kendall Scott, Dirk Weise: *MDA Distilled*. Addison-Wesley Longman, 2004

Kur07: Ivan Kurtev: *State of the Art of QVT: A Model Transformation Language Standard*. In: *Applications of Graph Transformations with Industrial Relevance*, Seiten 377–393. Springer, 2007

Boh06: Matthias Bohlen: *QVT und MultiI-Metamodell-Transformationen in MDA* In: *OBJEKTspektrum 2/2006*. 2006

Stev07: Perdita Stevens: *Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions*. In: *G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735*, Seiten 1–15. Springer, 2007

Kin07: Ekkart Kindler, Robert Wagner: *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*, 2007

Gre07: Joel Greenyer, Ekkart Kindler: *Reconciling TGGs with QVT.* In: *G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735*, Seiten 16–30. Springer, 2007

IKVProb: Probleme im QVT Standard, *http://projects.ikv.de/qvt/browser/mediniQVTEngine/trunk/mediniQVTInterpreter/examples/Shapes-Tutorial/qvt/OMGissues/OpenProblems.txt* (Stand: 29.05.2009)

QVT11TF: QVT 1.1 Task Force, *http://www.omg.org/issues/qvt-rtf.open.html* (Stand: 29.05.2009)

Nol09: Siegfried Nolte: *QVT - Relations Language: Modellierung mit der Query Views Transformation*. Springer, Berlin, 2009

EQVTD: Eclipse QVT Declarative, *http://www.eclipse.org/m2m/* (Stand: 29.05.2009)

VMTS: Visual Modeling and Transformation System, *http://www.vmts.aut.bme.hu/* (Stand: 29.05.2009)

TATA: ModelMorf, *http://www.tcs-trddc.com/solutions-software_rd.htm#ModelMorf* (Stand: 29.05.2009)

EUML2: Eclipse UML2 Project, *http://www.eclipse.org/modeling/mdt/?project=uml2* (Stand: 29.05.2009)

EBPN2: Eclipse BPMN2, *http://www.eclipse.org/modeling/mdt/?project=bpmn2* (Stand: 29.05.2009)

Acceleo: Eclipse Acceleo, *http://www.eclipse.org/modeling/m2t/?project=acceleo* (Stand: 29.05.2009)

oAW: openArcitectureWare, *http://www.openarchitectureware.org/* (Stand: 29.05.2009)

Fujuba: Fujaba Tool Suite, *http://wwwcs.uni-paderborn.de/cs/fujaba/* (Stand: 29.05.2009)

MOFLON: MOFLON meta-CASE Tool, *http://www.moflon.org/* (Stand: 29.05.2009)

Kön05: Alexander Königs: *Model Transformation with Triple Graph Grammars*. In: *L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713*, 2005

Kön08: Alexander Königs: *Model Integration and Transformation – A Triple Graph Grammar-based QVT Implementation*, 2008. TU Darmstadt

AGG: Attributed Graph Grammar, *http://tfs.cs.tu-berlin.de/agg/* (Stand: 29.05.2009)

GreAT: Graph Rewrite And Transformation, *http://www.escherinstitute.org/Plone/tools/suites/mic/great* (Stand: 29.05.2009)

Leng06: László Lengyel , Tihamér Levendovszky, Hassan Charaf: *Realizing QVT with Graph Rewriting-Based Model Transformation*. In: *ELECTRONIC COMMUNICATIONS OF THE EASST Volume 4, Proc. of 2nd InternationalWorkshop on Graph and ModelTransformation (GraMoT) 2006*. 2006

UMLX: Eclipse UMXL, *http://www.eclipse.org/gmt/umlx/* (Stand: 29.05.2009)

Wil09: Ed Willink über QVTD und UMLX, *http://www.eclipse.org/newsportal/article.php?id=3965&group=eclipse.modeling.m2m#3965* (Stand: 29.05.2009)

Schne08: Enrico Schnepel, *Entwicklung eines graphischen Editors zur Unterstützung eines modellgetriebenen Softwareentwicklungsprozesses*, 2008. Fachhochschule für Technik und Wirtschaft Berlin

EMFI: EMF Index, *http://www.eclipse.org/proposals/emf-index/* (Stand: 29.05.2009)

ATL: Eclipse ATL, *http://www.eclipse.org/m2m/atl/* (Stand: 29.05.2009)

Pilg08: Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, Yolande Berbers: *Constructing and Visualizing Transformation Chains*. In: *I. Schieferdecker, A. Hartman (Eds.): ECMDA-FA 2008, LNCS 5095*, Seiten 17–32. Springer, 2008

Rom08: Raphael Romeikat, Stephan Roser, Pascal Müllender, Bernhard Bauer: *Translation of QVT Relations into QVT Operational Mappings* . In: *A. Vallecillo, J. Gray, A. Pierantonio (Eds.): ICMT 2008, LNCS 5063*, Seiten 137-151. Springer,

2008

QVTDebug: A Graphical Debugger for QVT Relations, *http://www.big.tuwien.ac.at/teaching/theses/ma/102.html* (Stand: 29.05.2009)

Kru09: Angelika Kusel, Wieland Schwinger, Manuel Wimmer, Werner Retschitzegger: *Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy.* In: *Fourth IEEE International workshop UML and AADL*, 2009