UNIVERSITÄT KOBLENZ · LANDAU

Institut für Softwaretechnik,
Fachbereich 4

Universität
Koblenz-Landau

Diplomarbeit

GReTL- Entwurf und Implementierung eines operationalen Ansatzes für Modelltransformationen

Vorgelegt von:
Oliver Weichert
Mat.-Nr. 119820169
weichert@uni-koblenz.de

Studiengang:
Informatik

30.03.2009

Betreueung durch: Prof. Dr. Jürgen Ebert (Universität Koblenz-Landau)
Dr. Volker Riediger (Universität Koblenz-Landau)
Tassilo Horn (Universität Koblenz-Landau)

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Oliver Weichert

Danksagung

In erster Linie möchte ich mich bei meinen drei Betreuern, Prof. Dr. Jürgen Ebert, Dr. Volker Riediger und Tassilo Horn bedanken. Ihre Geduld und Ausdauer bei der Einarbeitung in dieses komplexe Themengebiet, bei der ich mich mit vielen mir bislang nicht vertrauten Konzepten beschäftigen musste, sowie ihre fortwährenden Anregungen, Hilfestellungen bei Fragen und konstruktive Kritik während der Erstellung dieser Arbeit waren eine große Unterstützung.

Darüber hinaus danke ich meinen Eltern, die mir durch Unterstützung jeglicher Art das Studium überhaupt erst ermöglicht haben.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Ein aktuelles Forschungsthema in der Softwaretechnik ist die modellgetriebene Softwareentwicklung. In diesem Zusammenhang hat die von der *Object Management Group* (OMG) entwickelte Model Driven Architecture (MDA) eine Vorreiterstellung eingenommen. Im Kern geht es darum, die Schwerpunkte der Softwareentwicklung von der Programmierung (dem manuellen Erstellen von Quellcode) stärker zur Modellierung hin zu verschieben. Generell geht es bei der Model Driven Architecture um *Modelltransformationen*.

Modelle sind dabei die zentralen Artefakte der MDA. Durch Verwendung verschiedener Modelle soll eine Trennung zwischen Spezifikation und Implementierung eines Systems erreicht werden. Die vier wichtigsten Modelle in diesem Zusammenhang sind das Computation Independent Model (CIM), das Platform Independent Model (PIM), das Platform Specific Model (PSM) und der Programmcode. Das CIM abstrahiert dabei von systeminternen Details und beschreibt ausschließlich die Systemumgebung. Dieses Modell findet bei der Anforderungsanalyse und Dokumentation Anwendung, um eine Grundlage für die Kommunikation zu schaffen. Dem CIM wird im MDA allerdings wenig Bedeutung zugemessen, da eine automatische Transformation zu einem anderen Modell nicht ohne weiteres Eingreifen der Entwickler möglich erscheint. Das PIM ist der eigentliche Ausgangspunkt des MDA-Prozesses. Es beschreibt die gesamten Aspekte eines Systems auf plattformunabhängige Weise. Dieses Modell wird in ein oder mehrere PSM transformiert, welche das PIM um Implementierungsdetails der jeweiligen Zielplattformen erweitern. Der Programmcode implementiert das PSM schließlich in einer (frei wählbaren) Programmiersprache.

Generell kann man bei den Transformationen zwischen zwei Transformationsvarianten unterscheiden. Zum einen gibt es die *Modell-zu-Modell-Transformation*, in der ein Modell in ein anderes überführt wird. Des weiteren kann ein Modell auch direkt in Quellcode oder Quellcodefragmente transformiert werden. Dies wird *Modell-zu-Code-Transformation* genannt. Geht man davon aus, dass die Syntax des Programmcodes jedoch auch als Modell betrachtet werden kann, so kann man diese Transformation eben-

falls als Modell-zu-Modell-Transformation einstufen.

Die Modelltransformation kann auf unterschiedliche Weise durchgeführt werden. Als erstes besteht die Möglichkeit der Transformation mit Hilfe von Algorithmen oder durch Transformationsregeln auf Modellebene. Die zweite Möglichkeit ist die Transformation von Metamodellen (Schematransformation), welche automatisch auch Transformationen auf Modellebene impliziert. In dieser Arbeit soll auf dem operationalen Ansatz, der in der Diplomarbeit von Florian Rheindorf vorgestellt wird, aufgebaut werden, welcher die letztgenannte Vorgehensweise verwendet.

Zur Transformation eines Modells sind verschiedene Sprachen notwendig, mindestens eine Sprache, die die Quell- und Zielmodelle definiert und eine Sprache, in der die Transformation definiert werden kann. In diesem Zusammenhang ist wichtig, dass insbesondere die UML eine Sprache darstellt. Es gibt bereits mehrere Sprachen, die zur Transformation eingesetzt werden wie zum Beispiel das von der OMG entwickelte *QVT* (query/view/transform) [Obj08a], die Atlas Transformation Language *(ATL)* [Atl08a] oder das im ReDSeeDS-Projekt eingesetzte *MOLA* [1]. Alle diese Sprachen besitzen jedoch gewisse Einschränkungen, die keine beliebigen Transformationen zulassen. Auf Grund dessen wurde mit *GReTL* in der Diplomarbeit von Florian Rheindorf ein von Prof. Dr. Jürgen Ebert an der Universität Koblenz-Landau entwickelter operationaler Ansatz vorgestellt, welcher diese Einschränkungen nicht mehr haben wird. Die dort aufgestellten Operationen stellen somit eine Transformationssprache dar.

Der operationale Ansatz der Transformation sollte *möglichst flexibel und allgemeingültig* sein und eine Transformation durch den Benutzer erlauben, ohne dass dieser explizit Transformationsregeln angeben muss. Dies wird erreicht durch die Zerlegung des Transformationsprozesses in einzelne *elementare Transformationsoperationen*. Diese elementaren Operationen lassen sich zu beliebigen Kombinationen zusammenfügen, so dass jede beliebige Transformation durchführbar ist, ohne sich dabei auf durch Regeln vordefinierte Transformationen beschränken zu müssen.

Der operationalen Ansatz in der Diplomarbeit von Florian Rheindorf [Rhe06] basiert auf dem *Meta-Meta-Modell von JGraLab* (Java Graph Laboratory). Das JGraLab ist eine API zur Manipulation von TGraphen, welche am IST entwickelt wurden.

TGraphen sind allgemeine Graphen, die über besondere Eigenschaften verfügen. Sie sind typisiert, attributiert, gerichtet und angeordnet. Zur Spezifikation von TGraphen kommt grUML (graphUML) zum Einsatz. Mit dieser aus UML Klassendiagrammen hergeleiteten Diagrammsprache können TGraphklassen definiert werden. Eine Klasse von TGraphen beinhaltet dabei alle TGraphen, die Instanzen des selben Schemas sind. Ein grUML-Diagramm visualisiert einen TGraphen, welcher selbst eine Instanz eines grUML-Schemas des grUML-Metaschemas, welches gleich dem JGraLab-Metaschema

ist, darstellt. Auf Grund der extensionalen Semantik der grUML-Diagramme lässt sich jedes Diagramm auch als TGraph darstellen. Eine Transformation eines grUML-Diagramms soll so automatisch auch eine Transformation des TGraphen nach sich ziehen.

Die Transformation der Modelle auf Schemaebene (M_{i+1}-Ebene, auf der die grUML-Schemata definiert werden) ist rein syntaktisch. Um eine Transformation auf Graphebene (M_i-Ebene, auf der die TGraphen bzw. die grUML-Diagramme als deren Visualisierungen existieren) durchzuführen, ist jedoch die Berücksichtigung der Semantik der verwendeten Schemata notwendig. Dazu wurden die aufgestellten Operationen um Parameter erweitert, welche die Semantik berücksichtigen. Diese Parameter wurden dabei formal (mathematisch) definiert.

Als Fortführung der Arbeit steht nun eine *Implementierung der definierten Operationen* an, damit diese produktiv eingesetzt werden können.

1.2 Ziele der Arbeit

Ziel der Diplomarbeit ist, die in der Arbeit von Florian Rheindorf definierten Operationen in Java zu realisieren. Die Umsetzung soll z.B. als Klassenbibliothek aufgebaut sein. Dabei sollen die elementaren Operationen so implementiert werden, dass sie sich zu Prozeduren zusammensetzen lassen, um die Wiederverwendbarkeit häufig verwendeter Transformationen zu gewährleisten. Bei der Implementierung soll auf vorhandene Klassenbibliotheken des IST (Institut für Softwaretechnik) zurückgegriffen werden. Dies sind insbesondere JGraLab, einer Bibliothek zur Erstellung und Manipulation von TGraphen, und dessen Erweiterung GReQL2, einer Anfragesprache für TGraphen. Der Einsatz von GReQL2 ist dabei gut geeignet, um die formal definierten Parameter der Operationen zu implementieren.

1.3 Aufbau der Arbeit

Nach einer Einführung in die *Model Driven Architecture (MDA)* in Kapitel 2, welche die Grundlage für die in dieser Arbeit verwendeten Konzepte bildet und einer kurzen Übersicht über verschiedene existierende *Transformationssprachen* in Kapitel 3 werden die Konzepte von GReTLin Kapitel 4 sowie die für die Implementierung von GReTL-verwendeten Bibliotheken in Kapitel 5 vorgestellt. Kapitel 6 stellt die Implementierung von GReTLaus Entwicklersicht dar, Kapitel 7 fokussiert die Anwendung und Implementierung eingener Transformationen durch den Benutzer. In Kapitel 8 wird ein um-

fassenderes Praxisbeispiel vorgestellt, in welchem GReTLverwendet werden kann. Kapitel 9 gibt einen Ausblick, wo GReTLeingesetzt werden könnte und wie eine weitere Entwicklung aussehen könnte. Anhang A gibt eine vollständige Übersicht über die Operationen, die im Kapitel 4 hergeleitet werden. Die Anhänge B und C sind eine Referenz der in einer Transformation verwendbaren Methoden/Operationen. Weitere Beispiele von Transformationen, die im Laufe der Entstehung von GReTLimplementiert wurden, sind in Anhang D erläutert. Die beiliegende CD enthält außer den Klassen von GReTL, den in dieser Arbeit vorgestellten Transformationen und den Testcases weitere Beispieltransformationen, die nicht in dieser Arbeit erläutert sind.

2 Model Driven Architecture

In diesem Kapitel werden die Grundlagen der *Model Driven Architecture* (MDA) vorgestellt, welche die Grundlage des in dieser Arbeit vorgestellten Konzeptes ist. Nach einer Einführung in die verwendeten *Meta-Architekturen* und die in dieser Arbeit verwendeten Konzepte und Begrifflichkeiten werden die Konzepte der MDA vorgestellt und auf die Grundlagen des darin verwendeten *Transformationsprozesses* dargestellt.

2.1 Meta-Architektur

Grundlage zur Durchführung von Modelltransformationen ist eine *Meta-Architektur*, da alle Operationen, die auf auf einer Ebene M_i durchgeführt werden sollen, auf der Ebene Mi_{+1} definiert werden müssen. Aus diesem Grund wird in diesem Abschnitt zuerst die von der OMG verwendete Meta-Architektur vorgestellt, auf der die meisten Transformationsansätze basieren, und im Anschluss daran die in dieser Arbeit verwendete, welche auf dem Meta-Meta-Modell von JGraLab basiert.

2.1.1 Meta-Architektur der OMG

Die Meta-Architektur der OMG besteht aus vier Ebenen (M0-M3). Diese werden in Abbildung 2.1 dargestellt. Generell beschreiben die Ebenen M1 bis M3 eine abstrakte Syntax, mit der die Instanzen der jeweils darunter liegenden Ebene (M_{i-1}) beschrieben werden, ohne Aussagen über die Semantik der Instanzen machen zu müssen. Genauer gesagt ist jedes Element auf einer Ebene Mi eine Instanz des zugehörigen Elements auf der Ebene M_{i+1}. Theoretisch sind beliebig viele Ebenen denkbar, da jedes Modell ein Meta-Modell besitzt; die Meta-Meta-Modelle auf der M3-Ebene sind in der Regel aber selbstbeschreibend, so dass keine weitere Ebene mehr notwendig ist.

Die grundlegende Ebene der Architektur ist die Meta-Meta-Ebene (M3). Auf ihr sind die Meta-Meta-Modelle angesiedelt. In den meisten Anwendungsfällen ist immer genau ein Meta-Meta-Modell dort definiert. Sie dient dazu, eine gemeinsame abstrakte Syntax

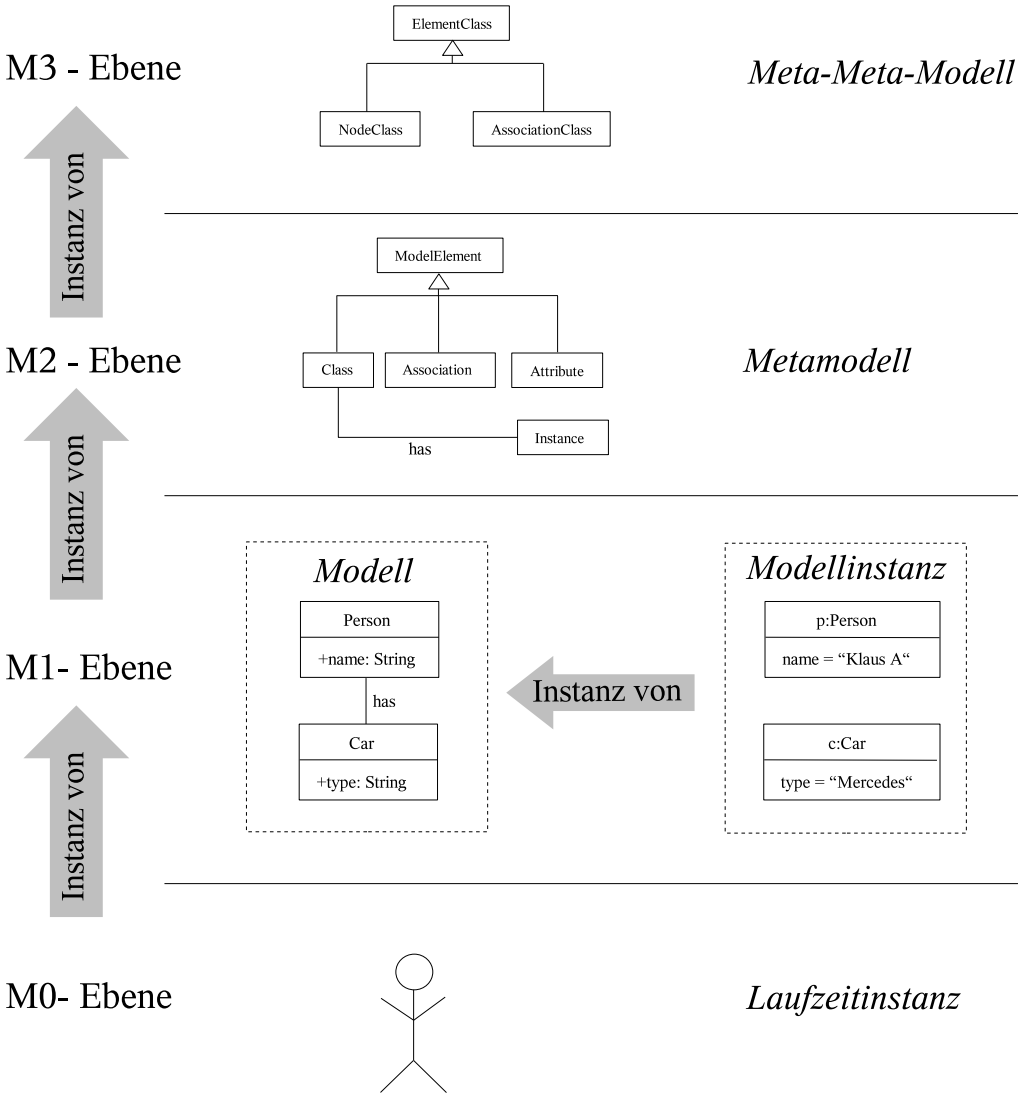

Abbildung 2.1: Meta-Architektur der OMG [Rhe06]

zur Beschreibung von beliebigen Meta-Modellen auf der Meta-Modell-Ebene (M2) zu definieren. Auf der M3-Ebene liegt das MOF 2.0 [Obj08a] der OMG, welches als Meta-Meta-Modell zur Definition der UML 2.0 verwendet wird.

Auf der nächsten Ebene liegt die Meta-Ebene (M2). Auf ihr werden die Meta-Modelle erstellt, welche Instanzen des zugehörigen Meta-Meta-Modells sind. Die Meta-Modelle beschreiben wiederum die Syntax, mit der die in der Modellebene (M1) liegenden Modelle beschrieben werden. Auf dieser Ebene ist zum Beispiel die UML 2.0 definiert.

Auf der Modell-Ebene (M1) werden die Modelle definiert. Seit Einführung der UML 2.0 werden auf dieser Ebene ebenfalls die ontologischen Modellinstanzen der Modelle angesiedelt. Diese Instanzbeziehung ist eine ontologische, wogegen die Laufzeitinstanzen der Modelle auf der Datenebene (M0) eine linguistische Instanzbeziehung zu den Modellen darstellt [Atk03].

2.1.2 Meta-Architektur dieser Arbeit

Da der operationale Ansatz, der dieser Arbeit zugrunde liegt, auf Graphen basiert, ist eine abgeänderte Meta-Architektur notwendig. Diese enthält ebenfalls die von der OMG vorgeschlagenen vier Ebenen M3 bis M0, auf denen allerdings andere Artefakte angeordnet sind (siehe Abbildung 2.2).

Auf der Meta-Meta-Modell-Ebene (M3) ist ebenfalls ein Meta-Meta-Modell positioniert, genauer gesagt das Meta-Meta-Modell von JGraLab, welches in Kapitel 2.1.5 genauer beschrieben wird.

Die M2-Ebene oder Schemaebene enthält ein Graphschema, welches eine Instanz des JGraLab-Meta-Meta-Modells ist und Elemente definiert, mit denen ein Graph auf der M1-Ebene beschrieben werden kann.

Die M1-Ebene enthält die Graphen an sich. Diese sind Instanzen des Graphschemas auf der Schemaebene. Die Instanzen eines Graphschemas, welches auf dem JGraLab-Meta-Meta-Modell basiert, sind immer TGraphen. Dies wird in Kapitel 2.1.3 näher beschrieben. Auf der M0-Ebene sind die Laufzeitinstanzen der Graphen angeordnet.

2.1.3 TGraphen

TGraphen sind *allgemeine Graphen*, die einige besondere Eigenschaften haben. Eine detaillierte Definition von TGraphen ist unter [Kah06] zu finden. Generell bestehen sie

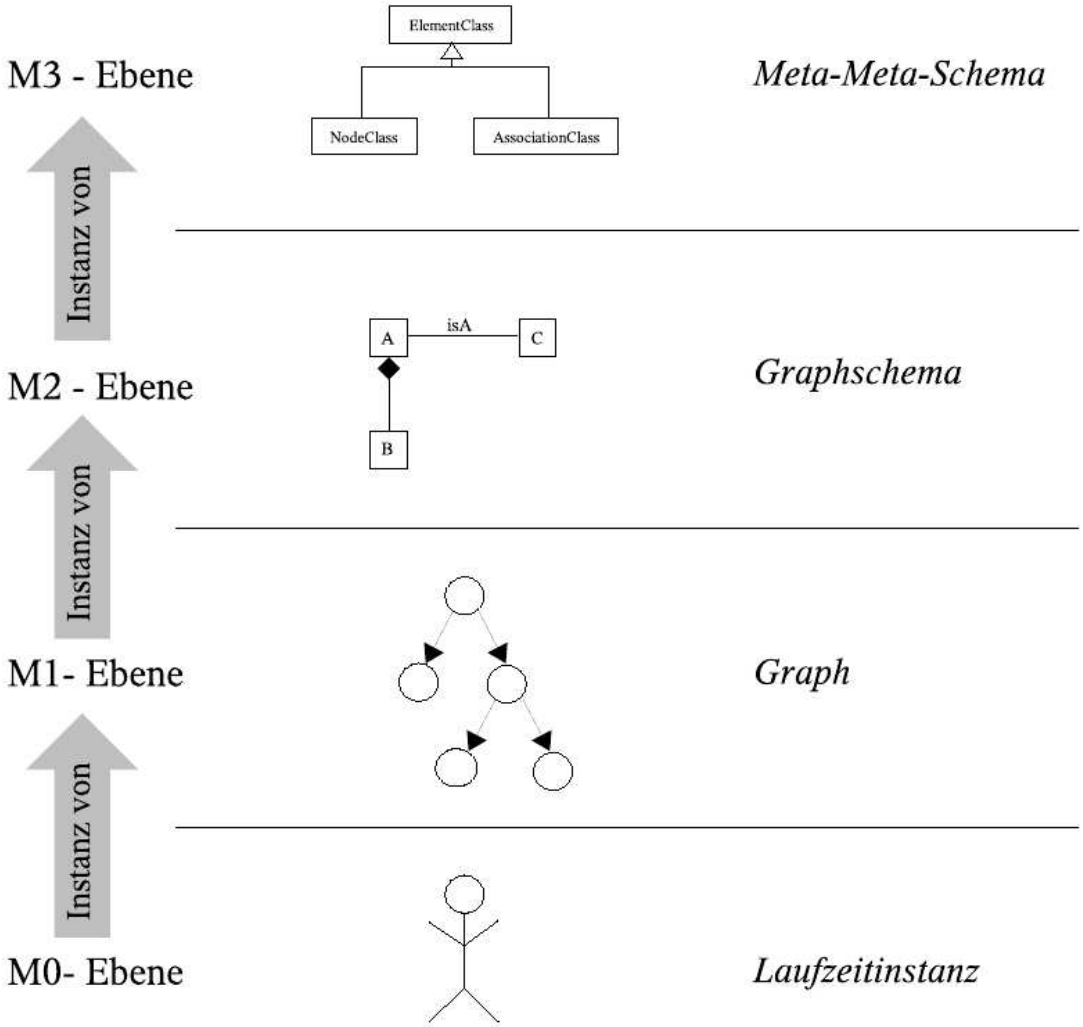

Abbildung 2.2: Meta-Architektur innerhalb der Diplomarbeit [Rhe06]

aus einer endlichen Menge von *Knoten* (Vertex), welche mit einer endlichen Menge von *Kanten* (Edge) verbunden sind. Darüber hinaus sind TGraphen

- typisiert: Die Knoten und Kanten eines TGraphen können einem bestimmten Typ zugeordnet werden. Die Typen werden durch das Graphschema auf M2-Ebene festgelegt. Für Typen kann über Spezialisierung im Graphschema eine Hierarchie erzeugt werden.
- attributiert: Den Knoten und Kanten eines TGraphen können Attribute mit Werten zugeordnet werden. Die möglichen Attribute werden wie bei den Typen im Graphschema definiert.
- gerichtet: Alle Kanten in einem TGraphen besitzen eine Richtung.
- angeordnet: Alle Kanten und Knoten sind in einer bestimmten Reihenfolge angeordnet.

Einer Kante sind genau zwei *Inzidenzen* zugeordnet, jeweils am Anfangs- und am Endknoten. Der Grad eines Knotens ist die Anzahl der Inzidenzen, mit denen der Knoten in Verbindung steht.

Ein TGraph ist eine Instanz eines grUML-Schemas, ein grUML-Diagramm eine Visualisierung eines TGraphen (siehe 2.1.4).

2.1.4 grUML

Die Graph UML (grUML) wurde am Institut für Softwaretechnik der Universität Koblenz-Landau entwickelt. Sie ist eine visuelle Sprache, die das Modellieren von Schemata für TGraphen (TGraphklassen) ermöglicht. grUML ist eine Teilmenge der UML und beschreibt

- Knotentypen durch Klassen und zugehörige Attribute. Die Klassen enthalten keine Methodendefinitionen.
- Kantentypen durch Assoziationen bzw. Aggregationen oder Kompositionen mit zugehörigen Attributen.
- Typspezialisierungen der Knoten und Kanten durch Spezialisierungen der Klassen und Assoziationen im Schema.
- Gradbedingungen der Knoten durch Multiplizitäten an den Assoziationsenden.

Für grUML existiert ein Konverter, der XMI-Dateien aus UML-Tools in das textuelle Format, welches JGraLab verwendet, umwandeln kann.

2.1.5 JGraLab Meta-Meta-Modell

Das JGraLab Meta-Meta-Modell (Abbildungen 2.3 und 2.4) beschreibt die Konzepte der Schemata, die für die Transformationen in dieser Arbeit verwendet werden. Es dient gleichzeitig als Meta-Meta-Model von grUML. Das JGraLab Meta-Meta-Modell wurde in der Diplomarbeit von Steffen Kable [Kah06] vorgestellt. Aus diesem Grund soll an dieser Stelle nur ein kurzer Überblick über die einzelnen Elemente gegeben werden.

Das Meta-Meta-Modell ist aufgeteilt in den *Attribut- und Graphenbereich* (Abbildung 2.3) und das *Wertebereichssystem* (Abbildung 2.4).

Der Attribut- und Graphenbereich beinhaltet die Klasse *Schema*. Ein Schema definiert eine *Graphklasse* (GraphClass) und kann beliebig viele *Wertebereiche* (Domains) und *Pakete* (Packages) enthalten. Ein Package beinhaltet beliebig viele *Graphelemente*, welche durch *Knotenklassen* (VertexClass) oder *Kantenklassen* (EdgeClass) spezialisiert werden. Des weiteren kann es beliebig viele Unterpakete beinhalten (*containsSubPackage*). Eine Kantenklasse lässt wiederum durch eine *Aggregationsklasse* (AggregationClass) oder *Kompositionsklasse* (CompositionClass) spezialisieren. Alle Graphelemente sowie die Graphklasse sind in der abstrakten Klasse *AttributedElementClass* zusammengefasst. Das Attribut *name* weist jedem Graphelement sowie der Graphklasse einen eindeutigen Namen zu. Über das Attribut *isAbstract* lässt zudem festlegen, ob das Graphelement oder die Graphklasse als Abstrakt definiert werden soll. Jede Graphklasse, Knotenklasse oder Kantenklasse kann beliebig viele *Attribute* (Attribute) besitzten, welche durch einen *Namen* (name) identifiziert werden und einem bestimmten *Wertebereich* (Domain) angehören.

Jede Knoten- oder Kantenklasse besitzt eine Assoziation zu sich selbst (*specializesVertexClass* bzw. *specializesEdgeClass*), womit sich eine Hierarchie aufbauen lässt. Eine Kantenklasse ist durch zwei Assoziationsklassen (*from* und *to*) mit der Kantenklasse verbunden. Mit den Attributen *min*, *max* und *roleName* werden jeweils die minimalen und maximalen Multiplizitäten und der Rollenname an jeweiligen Ende einer Kante auf Instanzebene festgelegt.

Im Wertebereich ist die abstrakte Klasse *Domain* definiert, von der alle konkreten Unterklassen *RecordDomain*, *EnumDomain*, *BooleanDomain*, *IntDomain*, *LongDomain*, *DoubleDomain*, *StringDomain*, *ObjectDomain*, *ListDomain* und *SetDomain* abgeleitet sind. Die Elemente von ListDomains sowie SetDomains setzen sich aus den Elementen einer Basisdomain ab, welche über *hasListElementDomain* bzw. *hasSetElementDomain* festgelegt wird. Eine RecordDomain setzt sich aus beliebigen Domains zusammen, was über die Aggregation *hasRecordDomainComponent* umgesetzt wird.

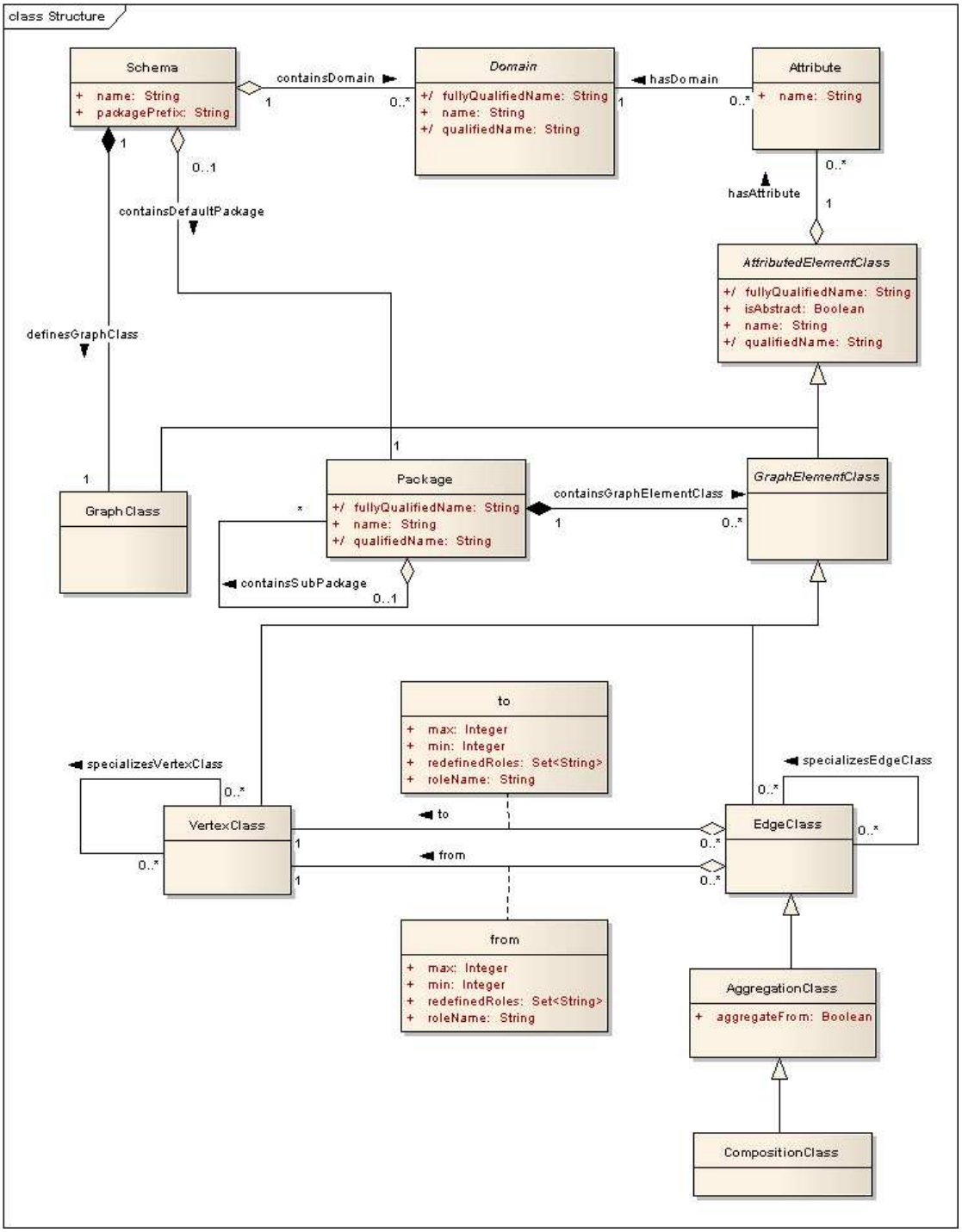

Abbildung 2.3: JGraLab Meta-Meta-Schema Attribut- und Graphenbereich

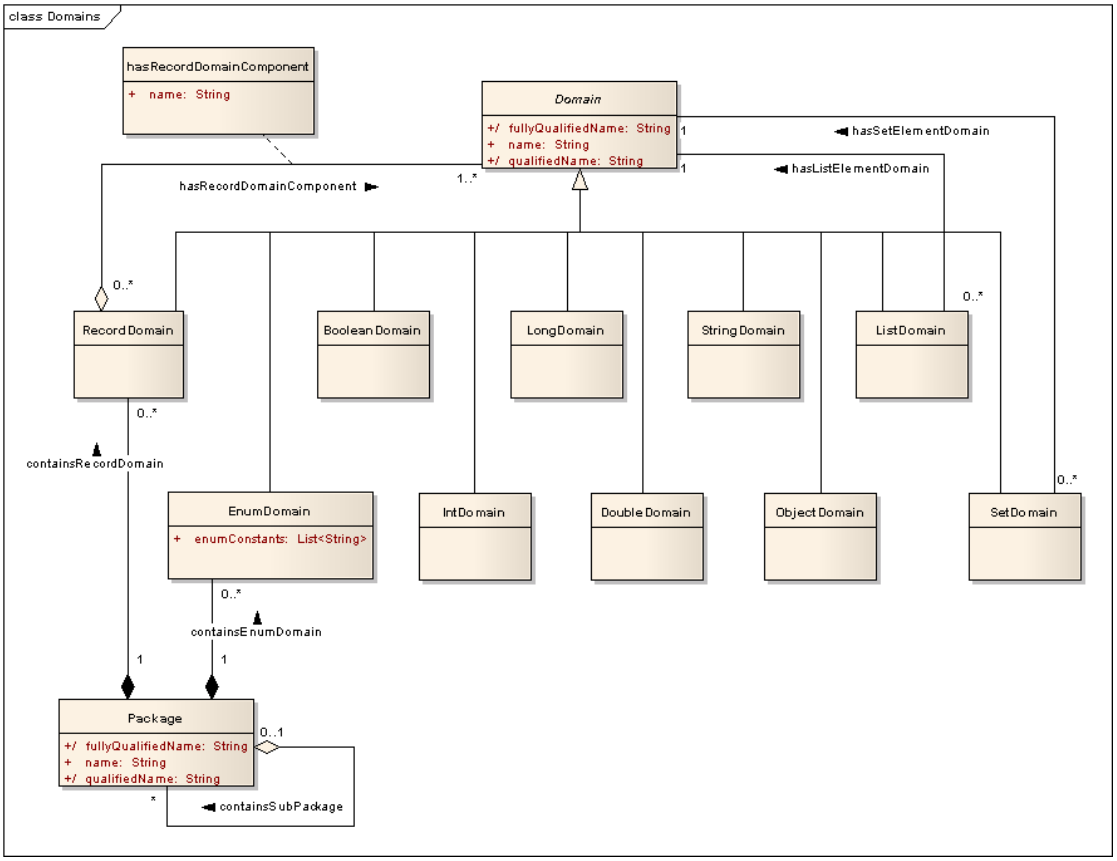

Abbildung 2.4: JGraLab Meta-Meta-Schema Wertebereich

2.2 Grundlagen der Model Driven Architecture

In diesem Abschnitt werden die Grundlagen der Model Driven Architecture erläutert. Zu diesem Zweck werden zuerst die verwendeten Modelle beschrieben, im Anschluss daran der MDA-Prozess. Zuletzt wird ein Überblick über die bei der MDA verwendeten Transformationen gegeben.

2.2.1 MDA-Modelle

Ziel der MDA ist es, eine komplette *Trennung zwischen der Spezifikation eines Systems und der Implementierung* herzustellen. Dazu existieren verschiedene Modelle, die durch Transformation ineinander überführt werden sollen (siehe Abschnitt 2.2.2). Diese Modelle sind im einzelnen

- das Computation Independent Model (CIM). Mit ihm werden die Anforderungen an ein System festgelegt, die unabhängig von dem Computersystem oder der Programmiersprache, in der das Programm später implementiert wird, existieren. Zu diesen Anforderungen zählen unter anderem die Geschäftsprozesse, die im System abzubildenden Objekte und andere Anforderungen, die nicht auf Systeminterna eingehen. Das CIM stellt eine Art Black Box dar, die aus einer externen Sichtweise das System und seine Umgebung beschreibt. Aktuell ist es nicht ohne weiteres möglich, textuelle Anforderungen zu analysieren und automatisch zu transformieren, dies ist ein aktuelles Forschungsthema zum Beispiel im ReDSeeDS-Projekt [1].
- das Platform Independent Model (PIM). Das PIM stellt das gesamte System dar, ohne auf plattformspezifische Details einzugehen. Damit ist es eine komplette Spezifikation des zu implementierenden Systems, welches sich auch dann nicht mehr ändert, wenn das System auf einer anderen Zielplattform implementiert werden soll.
- das Platform Specific Model (PSM). Es erweitert das PIM um Details zu einer bestimmten Zielplattform. Da es mehrere Zielplattformen für ein System geben kann, existieren je nach Bedarf auch mehrere PSM zu einem PIM.
- der Programmcode, der das PSM für eine gewählte Zielplattform in einer Programmiersprache implementiert.

2.2.2 MDA-Prozess

Am Anfang des MDA-Prozesses wird ein CIM des zu implementierenden Systems erstellt. Da es zur Zeit noch keine standardisierten Verfahren gibt, Anforderungen so zu definieren, dass sie automatisch in ein PIM transformiert werden können, wird in der Praxis meist direkt das PIM modelliert. Dieses PIM wird anschließend in ein oder mehrere PSM transformiert. Abschließend werden diese PSM in Quellcode transformiert. Ein grundlegendes Ziel der MDA ist es, diese Transformationen automatisch mit Tool-Unterstützung durchzuführen und nicht manuell durch die Entwickler.

Abbildung 2.5: Transformationsprozess

Um eine Transformation automatisch durchführen zu können, bedarf es zum einen Transformations-Tools, die die Transformationen durchführen, sowie Transformationssprachen, die die Transformationen beschreiben. Einige Beispiele für Transformationssprachen werden in Kapitel 3 vorgestellt. Der Transformationsprozess an sich wird im nächsten Abschnitt genauer erläutert.

2.3 Transformationsprozess

Beim Transformationsprozess wird wie in Abbildung 2.6 ein Quellmodell (M_a) in ein Zielmodell (M_b) transformiert. Für die Transformation wird ein Transformationsmodell verwendet (M_t), welches die Transformation beschreibt. Alle drei Modell besitzen ein eigenes Metamodell, auf welchem sie basieren. Diese drei Metamodelle basieren auf einem gemeinsamen Meta-Meta-Modell (MMM). Die einzelnen Modelle können beliebiger Form sein. Die verwendeten Modelle können zum Beispiel durch Graphen (dies wird in dieser Arbeit verwendet) oder auch Quellcode, UML-Diagramme, etc. repräsentiert werden. In dieser Arbeit wird ein Ansatz zur Implementierung einer Transformationssprache (MM_t) vorgestellt, welcher nicht die Modelle, sondern die Metamodelle transformiert; die Modelle selbst sollen bei dieser Transformation automatisch mitgeführt werden können (dazu mehr in Kapitel 4).

Generell kann man drei Arten der Modelltransformation unterscheiden:

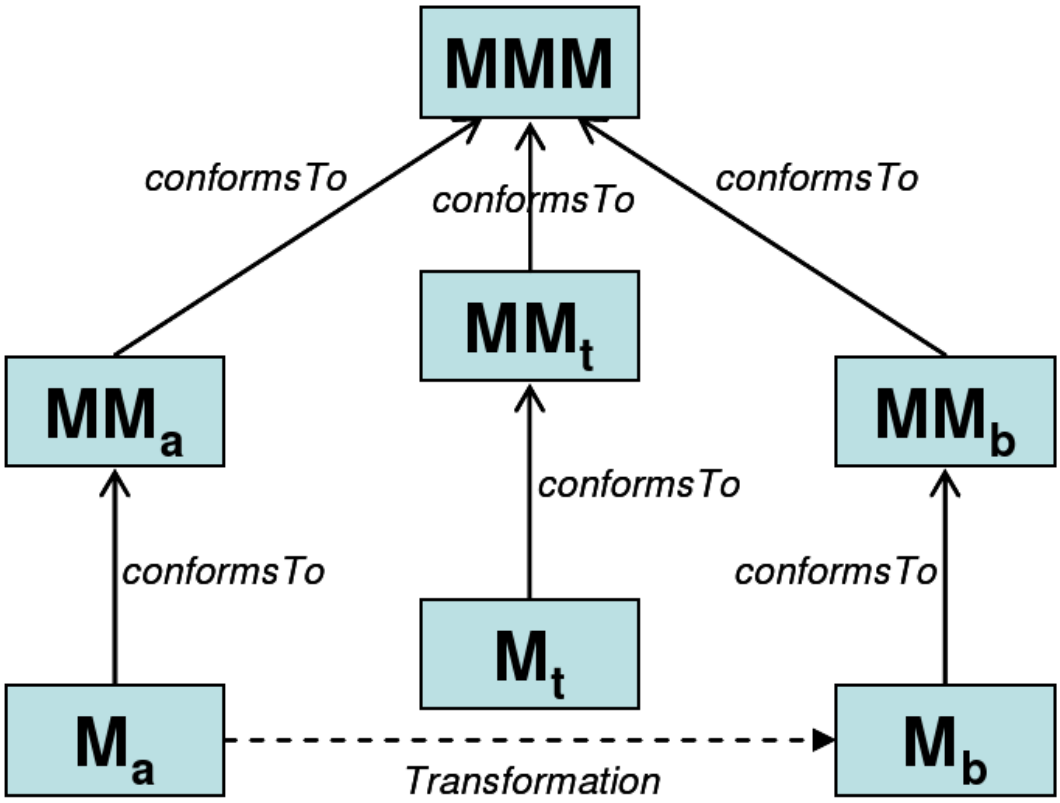

Abbildung 2.6: Modelltransformation

- Modell-zu-Modell-Transformation:
  Bei der Modell-zu-Modell-Transformation ist das Zielmodell ungleich dem Quellmodell. Die Meta-Modelle beider Modelle können (müssen aber nicht) unterschiedlich sein. Das Quellmodell wird bei dieser Transformationsart nicht geändert. Zum Beispiel könnte als Quellmodell der Graph einer vollständigen Straßenkarte sein, aus der ein Zielmodell erstellt werden soll, in dem ausschließlich Autobahnen durch ein UML-Objektdiagramm dargestellt werden. Innerhalb von dieser Arbeit wird ausschließlich diese Art der Transformation verwendet.
- Modellmodifikation:
  Bei der Modellmodifikation wird kein neues Zielmodell erstellt, sondern das Quellmodell im Laufe der Transformation modifiziert. Diese Art der Transformation kann verwendet werden, um einem bestehenden Modell neue Elemente hinzuzufügen oder Elemente zu verändern. Um das Beispiel der Straßenkarte aufzugreifen, könnte zum Beispiel zu jeder definierten Straße der Straßenname hinzugefügt werden. Bei der Modellmodifikation sind die Metamodelle von Quell- und Zielmodell immer identisch.
- Modell-zu-Code-Transformation:
  Die Modell-zu-Code-Transformation wird verwendet, um bestehende Modelle in eine textuelle Form zu transformieren. An häufigsten wird diese Art der Transformation von so genannten Codegeneratoren verwendet, welche das Modell eines Computerprogrammes in Quellcode übersetzt. Die Frage, ob diese Transformation als eigenständige Transformationsart bezeichnet werden kann, soll an dieser Stelle offen bleiben. Da eine Sprache zusammen mit ihrer Grammatik ein Metamodell darstellt, ein in ihr verfasster Text ein entsprechendes Modell, kann die Modell-zu-Code-Transformation auch als Sonderfall der Modell-zu-Modell-Transformation angesehen werden.

3 Transformationssprachen

In diesem Kapitel soll eine Übersicht über die bekanntesten Transformationssprachen gegeben werden. Dies sind im Speziellen QVT, ATL und MOLA.

3.1 QVT

MOF QVT (Query View Transformations) ist eine textuelle Transformationssprache, die eine graphische Repräsentation besitzt. Sie wurde von der OMG entwickelt und basiert auf der Meta Object Facility (MOF). Sie besteht aus deklarativen Teilen (Relations und Core) sowie imperativen Teilen (den Operational Mappings und der Black Box).

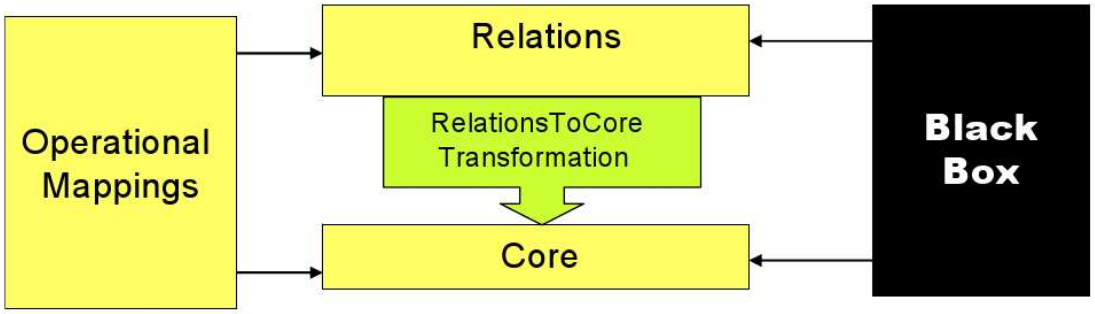

Abbildung 3.1: Beziehungen der QVT-Architektur

Die Architektur von QVT besteht aus vier Teilen:

- dem *„Relations“-Modell* mit einer zugehörigen Sprache, die dem Anwender ein leichtes Erstellen von Transformationen (die Spezifikation nennt dies in diesem Zusammenhang aber nicht Transformation, sondern „Beziehungen zwischen MOF-Modellen“ [Obj08b]) ermöglichen sollen. Die Sprache beinhaltet Elemente zur Deklaration von Eingabepattern und Templates für die Ausgabe in das Zielmodell. Sie besteht aus einer Kombination von Wörtern der englischen Sprache und

prädikatenlogischen Ausdrücken erster Ordnung. Anweisungen in dieser Sprache wird umgewandelt in das „Core“-Modell.

- dem *„Core“-Modell* mit einer zugehörigen Sprache. Diese Sprache ist genauso mächtig wie die entsprechende Sprache des Relations-Modells, besitzt aber eine wesentlich kompaktere Syntax, welche nur sehr schwer lesbar ist.
- den *„Operational Mappings“*, eine Sprache zur imperativen Definition von Transformationen. Sie ist genauso mächtig wie die Sprache des Relations-Modells. Sie kann entweder zur vollständigen Definition einer Transformation verwendet werden, oder zur Vervollständigung einer Transformation des Relations-Modells
- der *„Black Box“*, die die Ausführung von Transformationen zulässt, die von Drittanbietern erstellt werden und über eine vorgegebene Schnittstelle angesprochen werden. Dies ermöglicht, Transformationen auch in anderen als der von der Spezifikation vorgegebenen Sprachen zu entwickeln. Des weiteren ist es auf diese Art möglich, Bibliotheken für immer wiederkehrende Transformationen anzulegen.

Neben der „Relations“-Sprache existiert auch eine graphische Repräsentation der Sprachelemente. Dabei gibt es wiederum zwei Möglichkeiten Diagramme anzugeben; entweder als Standard-UML2-Diagramme oder mit einem proprietären Format. Die OMG schlägt in der Spezifikation vor, UML2-Objektdiagramme zu verwenden, die um einige Symbole und Notationen erweitert sind, um Transformationen darstellen zu können.

3.2 ATL

Wie auch QVT ist *ATL (Atlas Transformation Language)* eine textuelle Transformationssprache, die von ATLAS INRIA & LINA im Rahmen einer Ausschreibung der Object Management Group (OMG) eingereicht wurde. ATL ist als Erweiterung der Eclipse-Plattform implementiert und kann unter [Ecl08a] heruntergeladen werden. Wichtig ist, dass sie nicht auf Grundlage der fertigen QVT-Spezifikation [Obj08b], sondern auf Basis des Request for Proposals (RFP) entwickelt wurde und deshalb keine Implementierung von QVT darstellt [Wik08b].

Innerhalb der Eclipse-Erweiterung stehen *Editoren* zur Verfügung, mit denen die Metamodelle und die Modelle in einer textbasierten Version erstellt werden können. Diese werden im Editor mit KM3 (Kernel MetaMetaModel) [Gmt08a], einer Sprache zur textuellen Bescheibung von Metamodellen, welche der Java-Notation von Paketen und Klassen nachempfunden ist, erstellt und in ein von der ATL Virtual Machine lesbares Format umgewandelt [Atl08a]. Des weiteren stehen in der IDE ein *Konfigurations-Tool* zur Verfügung, mit dem Transformationen bearbeitet und gestartet werden können, und

ein *Debugger*, mit dem der Quellcode analysiert werden kann. Im Backend arbeiten ein *Compiler*, der die Eingabedaten in Bytecode umsetzt, sowie eine *virtuelle Maschine*, die die Transformationen durchführt [Atl08b]. Zusätzlich stehen eine Reihe von Methoden in einer Bibliothek zur Verfügung, die in einer Transformation verwendet werden können.

Eine ATL-Transformation besteht immer aus *vier Abschnitten*. Der erste Abschnitt ist der *Kopfbereich*, in dem festgelegt wird wie die Transformation heißt sowie welche Meta-Modelle als Quell- und Zielmodelle verwendet werden sollen. Der zweite Bereich gibt an, welche *ATL-Bibliotheken* importiert werden sollen; dieser Bereich ist optional. Im dritten Bereich können so genannte *Helper* definiert werden, die innerhalb einer Transformation wie eine Methode aufgerufen werden können; dieser Bereich ist ebenfalls optional. Im vierten Bereich werden die *Transformationsregeln* festgelegt.

Die Transformationsregeln in ATL gliedern sich in zwei Sorten auf: die *matched rules*, welche deklarativ arbeiten und mit Hilfe eines Pattern Matching auf die Elemente eines Modells angewandt werden, und die *called rules*, welche imperativ arbeiten und zusätzliche Eingabeparameter besitzen. Called rules müssen dabei immer von matched rules aufgerufen werden. Sie werden unter anderen dafür benötigt, Elemente im Zielmodell zu generieren, die nicht im Quellmodell vorhanden sind.

3.3 MOLA

MOLA [Kal05] ist eine graphische Transformationssprache, die an der Universität von Lettland entwickelt wurde. Ihre Hauptvorteile, so die Entwickler, liegen in einer vollständigen Definition graphischer Pattern, die mit aus der prozeduralen Programmierung übernommenen Kontrollstrukturen verknüpft werden.

In MOLA werden das *Ausgangs- und Zielmetamodell* in einem Klassendiagramm beschrieben, in welchem spezielle Assoziationen die Beziehungen der Klassen in den beiden Modellen definiert; Paketstrukturen können dabei zur Trennung der beiden Metamodelle eingesetzt werden. Zusätzlich erlaubt ist die Definition von temporären Klassen und Assoziationen, die nur für die Transformation benötigt werden. Die Sprache der Klassendiagramme ist eine “leicht eingeschränkte Version von EMOF (Essential Meta Object Facility)” [Kal05] der OMG.

Die *Transformation* an sich wird mit MOLA-Diagrammen beschrieben. Diese Diagramme verbinden einzelne (graphisch dargestellte) Anweisungen mit als Pfeile dargestellte Kontrollflusskanten und erzeugen so eine Ausführungssequenz. Zu diesen Anweisungen gehören Schleifen, Verzweigungen, Unterprogrammaufrufe und diverse Symbole

für Startpunkte, Endpunkte oder Übergabewerte für Unterprogramme.

Für MOLA existiert ein Satz von Tools, die das Arbeiten mit MOLA unterstützen. Dazu gehört eine *Arbeitsumgebung* zur Erstellung und Bearbeitung der Diagramme (TDE, Transformation Definition Environment) und einer *Umgebung für die Ausführung der Transformationen* (TEE, Transformation Execution Environment). Beide Umgebungen greifen dabei auf ein *gemeinsames Repository* zur Datenspeicherung zurück, welches mit einer relationalen Datenbank implementiert ist. Die TDE ist eine *Sammlung von graphischen Editoren*, mit der sowohl die Metamodelle als auch die MOLA-Diagramme bearbeitet werden können, sowie einem *Compiler*, der die Diagramme in ein Format übersetzt, welches im Repository gespeichert werden kann. Die TEE enthält einen Interpreter (MOLA Virtual Machine), der die MOLA-Diagramme in SQL-Anweisungen übersetzt und diese auf im Repository gespeicherte Modellinstanzen anwendet, sowie einen *Modell-Editor* zum Bearbeiten der Modellinstanzen.

4 GReTL

GReTL ist ein operationaler Ansatz zur Modelltransformation, der im Gegensatz zu den in Kapitel 3 vorgestellten Sprachen nicht auf dem MOF aufbaut, sondern auf dem Meta-Meta-Schema von JGraLab (siehe 2.1.5).

Grundidee des operationalen Ansatzes ist, dem Benutzer eine Transformation *durch Angabe von operationalen semantischen Ausdrücken zu elementaren programmierbaren Transformationsoperationen* zu ermöglichen. Dabei reicht bei der Schematransformation die Angabe der Transformationsoperationen an sich, bei der Graphtransformation ist zusätzlich die Angabe von semantischen Ausdrücken als Operationsparameter erforderlich, welche in Kapitel 4.2 beschrieben werden. Der Transformationsprozess soll ausschließlich durch die Reihenfolge der Ausführung einzelner elementarer Transformationsoperationen bestimmt werden. Diese Vorgehensweise erhebt den Anspruch, jede denkbare Transformation durchzuführen zu können.

4.1 Transformation auf Schemaebene

Zur Transformation auf Schemaebene wird eine Menge von Transformationsoperationen benötigt. Diese Operationen werden mit Hilfe eines in [Rhe06] beschriebenen Regelwerkes für das JGraLab-Meta-Meta-Modell aufgestellt, welches in Kapitel 4.1.1 beschrieben wird. Die gesamte Menge der elementaren Operationen soll vollständig sein, um jede mögliche Transformation ausführen zu können. Durch die Verwendung des JGraLab-Meta-Meta-Schemas können nicht nur die Schemata und deren Instanzen, sondern auch die entsprechenden TGraphen bzw. deren grUML-Repräsentation transformiert werden.

Die Operationen werden in einem an Java angelehnten Pseudocode dargestellt. Zur Unterscheidung der Typen von Knoten und Kanten werden diese mit der Typangabe als Subskript versehen. Beispiele für Signaturen von Operationen sind in der folgenden Auflistung dargestellt:

- Zum Anlegen eines Knotens vom Typ Diplomarbeit:

`Vertex`$_{\texttt{Diplomarbeit}}$ `createVertex(VertexClass Diplomarbeit)`
Die Operation gibt einen Knoten des Typs Diplomarbeit zurück und erwartet zum Knoten gehörende Knotenklasse als Eingabeparameter.

- Zum Löschen einer Kante:
  `void deleteEdge(Edge e)`
  Die Operation erwartet den zu löschenden Knoten als Eingabeparameter und gibt nichts zurück
- Zum Setzen eines Attributs eines Knotens oder einer Kante:
  `void setAttribute(AttributedElement e, Attribute a, Value v)`
  Sie erwartet das Element, dessen Attribut gesetzt werden soll, das Attribut selbst und den Wert, den es bekommen soll, als Eingabeparameter und gibt nichts zurück

4.1.1 Regeln zur Generierung der Transformationsoperationen

Die Transformationsoperationen werden auf Basis von Regeln erstellt, welche in folgenden dargestellt sind. Vor den Regeln ist jeweils die Nummer der Regel angegeben, welche auch im Anhang A verwendet wird, in dem die vollständige Menge an Operationen für das JGraLab-Meta-Meta-Schema angegeben sind.

Für jede Klasse C mit n Attributen $attr_i$ (i=1, ..., n) müssen anhand der folgende Regeln fünf Operationen erzeugt werden:

- A1: Erzeugen eines Knotens:
  $Vertex_c$ createC($Domain_1$ $attr_1$, ..., $Domain_n$ $attr_n$)
- A2: Löschen eines Knotens:
  void deleteC($Vertex_c$ aC)
- A3: Für jedes Attribut einer Klasse eine Methode zum Lesen des zugehörigen Wertes:
  Domain $getAttr_i$($Vertex_c$ aC)
- A4: Für jedes Attribut einer Klasse eine Methode zum Setzen des Wertes:
  void $setAttr_i$($Vertex_c$ ac, $Domain_i$ $attr_i$)
- A5: Kopieren eines Knotens:
  $Vertec_c$ copyC($Vertex_c$ aC)

Für jede Assoziation (bzw. Aggregation oder Komposition) zwischen zwei Klassen Alpha und Omega mit den Attributen $attr_i$ (i=1, ..., n) müssen anhand der folgende Regeln sechs Operationen erzeugt werden:

- B1: Erzeugen einer Beziehung:
  $Edge_A$ createA($Vertex_{Alpha}$ a, $Vertex_{Omega}$ o, $Domain_1$ $attr_1$, ..., $Domain_n$ $attr_n$)
- B2: Löschen einer Beziehung:
  void deleteA($Edge_A$ aA)
- B3: Für jedes Attribut einer Beziehung eine Methode zum Lesen des zugehörigen Wertes:
  Domain $getAttr_i$($Edge_A$ aA)
- B4: Für jedes Attribut einer Beziehung eine Methode zum Setzen des Wertes:
  void $setAttr_i$($Edge_A$ aA, $Domain_i$ $attr_i$)
- B5: Kopieren einer Beziehung:
  $Edge_A$ copyA($Edge_A$ aA)
- B6.1: Umlegen des Alpha-Endes einer Beziehung:
  void moveAAlpha($Edge_A$ aA, $Vertex_{Alpha}$ a)
- B6.2: Umlegen des Omega-Endes einer Beziehung:
  void moveAOmega($Edge_A$ aA, $Vertex_{Omega}$ a)

4.1.2 Beispiel: Herleitung von Operationen

Um das Erstellen der Operationen beispielhaft darzustellen, werden im folgenden die Operationen für die inzwischen neu zum JGraLab-Meta-Meta-Schema hinzugekommenen Packages durchgeführt.

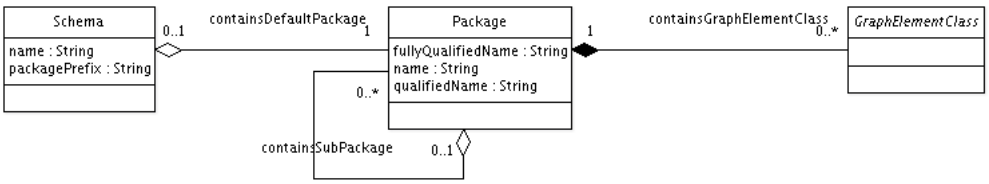

Abbildung 4.1: Ausschnitt Packages im JGraLab Meta-Meta-Schema

Die Klasse Package besitzt eine contains-Beziehung zu sich selbst (containsSubPackage), mit der Pakete hierarchisch geordnet werden können. Ein Package enthält eine beliebige Anzahl von GraphElementClass-Klassen (containsGraphElementClass). Eine RecordDomain ist einem Package zugeordnet (containsRecordDomain), genauso wie einer EnumDomain (containsEnumDomain). Einem Schema wird ein Default-Package zugeordnet (containsDefaultPackage). Dies führt zu folgenden Operationen (hier nur

die Klasse Package, die Aggregation containsSubPackage und die Komposition containsRecordDomain, eine vollständige Liste siehe Anhang A):

Package (Klasse):

A1	$\text{Vertex}_{\text{VertexClass}}$ createPackage(String name)
A2	**void** deletePackage($\text{Vertex}_{\text{Package}}$ v)
A3	**String** getName($\text{Vertex}_{\text{Package}}$ v)
A4	**void** setName($\text{Vertex}_{\text{Package}}$ v, String name)
A5	$\text{Vertex}_{\text{VertexClass}}$ copyPackage($\text{Vertex}_{\text{Package}}$ v)

Bei der Klasse Package kommen alle Regeln zur Herleitung von Operationen bei Klassen vor. Ein Package kann erzeugt (A1), gelöscht (A2) oder kopiert (A5) werden. Sein Attribut *name* kann gelesen (A3) oder gesetzt (A4) werden.

containsSubPackage (Aggregation von Package zu sich selbst):

B1	$\text{Edge}_{\text{containsSubPackage}}$ createContainsSubPackage($\text{Vertex}_{\text{Package}}$ alpha, $\text{Vertex}_{\text{Package}}$ omega)
B2	**void** deleteContainsSubPackage($\text{Edge}_{\text{containsSubPackage}}$ e)
B5	$\text{Edge}_{\text{containsSubPackage}}$ copyContainsSubPackage($\text{Edge}_{\text{containsSubPackage}}$ e)
B6.1	**void** moveContainsSubPackageAlpha($\text{Edge}_{\text{containsSubPackage}}$ e, $\text{Vertex}_{\text{Package}}$ v)
B6.2	**void** moveContainsSubPackageOmega($\text{Edge}_{\text{containsSubPackage}}$ e, $\text{Vertex}_{\text{Package}}$ v)

Bei den Operationen zur Aggregation containsSubPackage kommen die Regeln für Beziehungen zur Anwendung. Die Regel B3 und B4 werden nicht verwendet, da die Aggregation keine zugeordneten Attribute besitzt. Sie kann ebenfalls erzeugt (B1), gelöscht (B2) oder kopiert (B5) werden. Das Verschieben der Alpha- (B6.1) und Omega-Enden (B6.2) ist ebenfalls möglich.

containsRecordDomain (Komposition von Package zu RecordDomain):

B1	$\text{Edge}_{\text{containsRecordDomain}}$ createContainsRecordDomain($\text{Vertex}_{\text{Package}}$ alpha, $\text{Vertex}_{\text{RecordDomain}}$ omega)
B2	**void** deleteContainsRecordDomain($\text{Edge}_{\text{containsRecordDomain}}$ e)
B5	$\text{Edge}_{\text{containsRecordDomain}}$ copyContainsRecordDomain($\text{Edge}_{\text{containsRecordDomain}}$ e)
B6.1	**void** moveContainsRecordDomainAlpha($\text{Edge}_{\text{containsRecordDomain}}$ e, $\text{Vertex}_{\text{Package}}$ v)
B6.2	**void** moveContainsRecordDomainOmega($\text{Edge}_{\text{containsRecordDomain}}$ e, $\text{Vertex}_{\text{RecordDomain}}$ v)

Die Komposition containsRecordDomain ist analog zur Aggregation containsSubPackage aufgebaut.

4.2 Transformation auf Graphebene

Zusätzlich zur Transformation auf Schemaebene soll eine Transformation auf Graphebene ermöglicht werden, bei alle zum Schema gehörenden Instanzen ebenfalls transformiert werden. So kann sichergestellt werden, dass die Instanzbeziehung der Graphen zu den Schemata auch nach der Transformation gilt. Auf Graphebene ist es zusätzlich notwendig die Semantik der verwendeten Schemata zu berücksichtigen, um eine vollständige und korrekte Transformation zu gewährleisten. Es wird der "Instanz von"-Beziehung der M1-Ebene zur M2-Ebene eine Bedeutung zugewiesen.

Da durch die Definition eines Schemas nur syntaktische Aussagen über ein Modell getroffen werden können, muss die semantische Bedeutung der einzelnen Elemente einer zugehörigen Instanz durch zusätzliche Angaben ergänzt werden. Die Semantik der Graphsyntax innerhalb dieser Arbeit ist *extensional* definiert: Die Semantik eines Schemas auf M_{i+1} ist die Menge seiner Instanzen auf M_i. Durch diese Definition reicht es, die Menge der möglichen Instanzen eines Schemas formal zu definieren, um dessen Semantik festzulegen.

In der JGraLab-Meta-Architektur ist es notwendig, die *Teilsemantiken* von Graphklassen (GraphClass), Knotenklassen (VertexClass), Kantenklassen (EdgeClass), Attributen (Attribute) und Wertebereichen (Domains) genauer zu beschreiben. Die Semantik der Graphklassen, Knotenklassen und Wertebereiche ist eigenständig, bei Kantenklassen und Attributen müssen zusätzlich die ein- und ausgehenden Kanten mit ihren Endknoten berücksichtigt werden. Bei Kantenklassen sind dies die from- und to-Kanten mit den zugehörigen Knoten, bei Attributen die hasAttribute-Kante mit zugehörigem Ausgangsknoten und die ausgehende hasDomain-Kante mit zugehörigem Domain-Knoten.

Zur Definition der Teilsemantiken werden die Universen *GRAPH*, *VERTEX*, *EDGE* und *VALUE* verwendet. Diese enthalten jeweils eine unendliche Menge von Graphen, Knoten, Kanten und Werten. Mit deren Hilfe können die folgenden Teilsemantiken definiert werden:

- **GraphClass:**
  Die Graphklasse G auf M2-Ebene beschreibt die Menge von erzeugten Graphen $\overline{G}$ vom Typ G auf M1-Ebene. $\overline{G}$ ist eine Teilmenge des Universums *GRAPH*.
- **VertexClass:**
  Die Knotenklasse V auf M2-Ebene beschreibt die Menge von erzeugten Knoten $\overline{V}$ vom Typ V auf M1-Ebene. $\overline{V}$ ist eine Teilmenge des Universums *VERTEX*.
- **EdgeClass:**
  Die Kantenklasse E auf M2-Ebene beschreibt die Menge von erzeugten Kanten $\overline{E}$

Element	M2	M1
GraphClass	G	$\overline{G} \subseteq GRAPH$
VertexClass	V	$\overline{V} \subseteq VERTEX$
EdgeClass	$\mathrm{E}\frac{\text{from } 1}{\text{to } 1}\frac{\mathrm{V}}{\mathrm{W}}$	$\overline{E} \subseteq EDGE, \alpha : \overline{E} \rightarrow \overline{V}, \omega : \overline{E} \rightarrow \overline{W}$
Attribute	$\mathrm{A}\frac{\text{hasAttribute } 1}{\text{hasDomain } 1}\frac{\mathrm{X}}{\mathrm{T}}$	$\overline{A} : \overline{X} \rightarrow \overline{T}$
Domain	T	$\overline{T} \subseteq VALUE$

Tabelle 4.1: Teilsemantiken des JGralab Meta-Meta-Schemas

vom Typ E auf M1-Ebene. $\overline{E}$ ist eine Teilmenge des Universums *EDGE*. Zusätzlich beschreiben die Abbildungen α bzw. ω die from- und to-Beziehungen mit den jeweiligen Knoten. Die Abbildung α ordnet dabei jeder Kante aus der Menge $\overline{E}$ einen Anfangsknoten aus der Menge $\overline{V}$ zu, ω einen Endknoten aus der Menge $\overline{W}$.

- **Domain:**
  Die Menge T beschreibt eine Menge von konkreten Werten $\overline{T}$. $\overline{T}$ ist eine Teilmenge des Universums *VALUE*.
- **Attribute:**
  Die Abbildung $\overline{A}$ ordnet jedem Element $\overline{X}$ einen konkreten Wert $\overline{T}$ zu.

Zusätzlich müssen bei der Transformation auf M1-Ebene die Generalisierungsbeziehungen auf der M2-Ebene berücksichtigt werden. Diese haben keine konkreten Instanzen auf M1 und erfordern daher eine andere Behandlung als die bisher vorgestellten Elemente. Bei der Graphtransformation muss daher die Vererbung von Attributen und Beziehungen Berücksichtigung finden. Zu diesem Zweck wird das Superskript * eingeführt, welches kennzeichnet dass geerbte Attribute und Beziehungen berücksichtigt werden. Die Teilsemantiken von EdgeClass und Attribute müssen wie in Tabelle 4.2 angepasst werden.

Eine Operation kann dabei auch mehrere semantische Ausdrücke enthalten, abhängig davon, wie viele Ausdrücke zur genauen Beschreibung der Mengen benötigt werden. Die semantischen Ausdrücke werden den Operationen in der Pseudocodedarstellung als zusätzliche Parameter getrennt durch das Symbol ▶ übergeben.

Ein Aufruf einer Operation createAElement mit den ursprünglichen Parametern p_1, ..., p_n und den semantischen Ausdrücken s_1, ..., s_n hat dadurch die Form:

Element	M2	M1
EdgeClass	$\mathrm{E}\frac{\text{from}\ 1\ \mathrm{V}}{\text{to}\ 1\ \mathrm{W}}$	$\overline{E} \subseteq EDGE, \alpha^* : \overline{E} \rightarrow \overline{V}^*, \omega^* : \overline{E} \rightarrow \overline{W}^*$
Attribute	$\mathrm{A}\frac{\text{hasAttribute}\ 1\ \mathrm{X}}{\text{hasDomain}\ 1\ \mathrm{T}}$	$\overline{A}^* : \overline{X}^* \rightarrow \overline{T}$

Tabelle 4.2: Teilsemantiken des JGralab Meta-Meta-Schemas unter Berücksichtigung der Vererbung

createAElement(p_1, ..., p_n ► s_1, ..., s_n)

Bei Ausführung einer create-Operation wird auf M1-Ebene eine zusätzliche Funktion benötigt, die den Elementen im Zielgraphen die zugehörigen Elemente des Quellgraphen zuordnet. Zu diesem Zweck wurde in [Rhe06] die Abbildung $\zeta_{Element} : \overline{Element} \twoheadrightarrow M$ eingeführt. Sie stellt eine bijektive Abbildung dar, die jedem Element der Menge der Instanzen von *Element* im Zielgraphen genau einem Element der Vorbildmenge M aus dem Quellgraphen zuordnet. Durch das Subscript wird gekennzeichnet, zu welchem Schemaelement sie gehört. Sie wird beim Aufruf einer create-Operation erzeugt und ermöglicht. Durch sie und die zugehörige Umkehrfunktion $\zeta^{-1}_{Element}$ ist eine lückenlose Navigation zwischen allen Elementen im Quell- und Zielgraph möglich. An späterer Stelle wird die Schreibweise von $\zeta_{Element}$ bzw. $\zeta^{-1}_{Element}$ in eine textuelle Form umgewandelt und durch arch*Element* (engl. archtype = Urbild) bzw. img*Element* (engl. image = Bild) ersetzt, da sich die mathematische Schreibweise nicht in ASCII-Form darstellen lässt und statt dessen *GReQL2* zur Formulierung der Ausdrücke verwendet werden wird. In diesem Kapitel wird auf Grund der mathematischen Schreibweise der folgenden Beispiele allerdings noch auf die ASCII-Notation verzichtet.

4.2.1 Beispiel: Transformation eines einfachen Beispiels

Abbildung 4.2 beschreibt das Ausgangsschema, Abbildung und Zielschema des Beispiels. Das Ausgangsschema besteht aus zwei Knotenklassen vom Typ A und B, welche mit einer Kante vom Typ E verbunden sind. Die Knotenklasse B verfügt über ein Attribut X vom Typ String. Das Zielschema enthält die Knotenklassen AA, BB und EE. AA und EE sind mit einer Kante vom Typ CC verbunden, EE und BB mit einer Kante vom Typ DD. Die Klasse BB verfügt über ein Attribut Y vom Typ String.

Bei einer Transformation sollen alle Knoten vom Typ A in Knoten des Typs AA umgewandelt werden. Alle Knoten vom Typ B werden zu Knoten vom Typ BB. Aus den

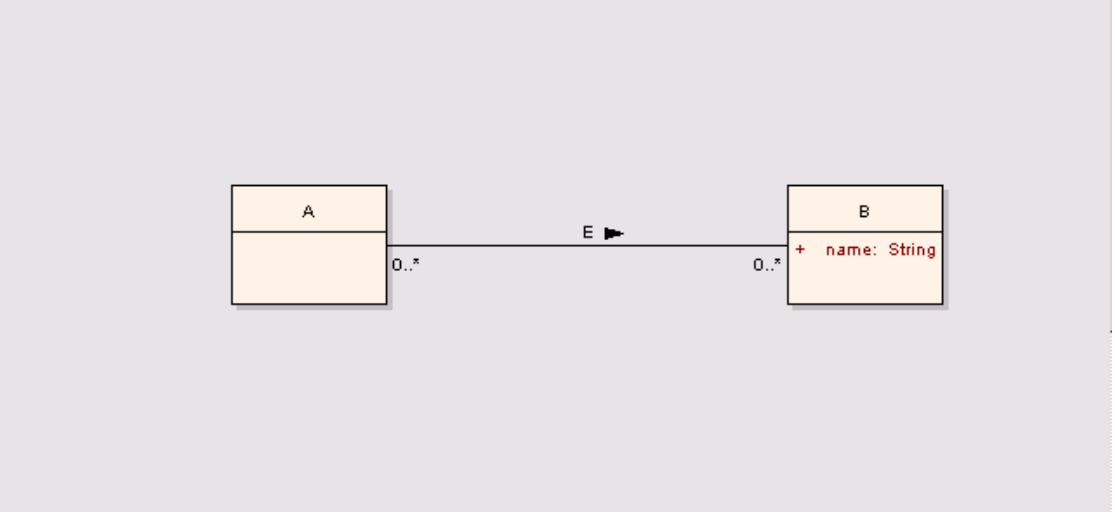

Abbildung 4.2: Beispiel: Quellschema Beispieltransformation

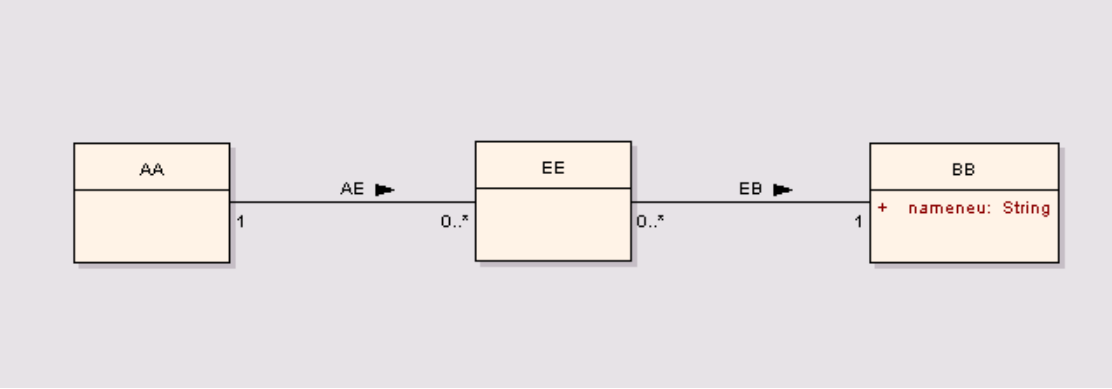

Abbildung 4.3: Beispiel: Zielschema Beispieltransformation

Kanten vom Typ E soll der Knotentyp EE entstehen. Für die Kanten CC und DD werden neue Kanten eingeführt, die den Inzidenzen der Kante E aus dem Ausgangsmodell entsprechen.

Durch die folgenden Operationsaufrufe wird die Transformation durchgeführt:

/* M2: Erzeugen einer neuen Knotenklasse “AA”
M1: “Kopieren”, A wird zur Vorbildmenge */
$\text{Vertex}_{\text{VertexClass}}$ A’ = createVertexClass(“AA”, false ▶ $\overline{A}$);

/* M2: Erzeugen einer neuen Knotenklasse “BB”
M1: “Kopieren”, B wird zur Vorbildmenge */
$\text{Vertex}_{\text{VertexClass}}$ B’ = createVertexClass(“BB”, false ▶ $\overline{B}$);

/* M2: Erzeugen einer neuen Knotenklasse “EE”
M1: “Kopieren”, E wird zur Vorbildmenge
$\text{Vertex}_{\text{VertexClass}}$ E’ = createVertexClass(“EE”, false ▶ $\overline{E}$);

/* M2: Erzegen einer neuen Kantenklasse “AE”
M1: “EE” wird zur Vorbildmenge der neuen Kanten, das Alpha-Ende ist jeweils der “AA”-Knoten, der über die ζ-Abbildungen von AE und AA erreicht wird, das Omega-Ende jeweils der über die ζ-Abbildungen von AE und EE erreicht wird */
$\text{Edge}_{\text{EdgeClass}}$ C’ = createEdgeClass(“AE”, false, A’, [], E’, [] ▶ $\overline{E}$, $\lambda\, c : \overline{EE} \bullet \zeta_{AA}^{-1}(\alpha(\zeta_{AE}(c)))$, $\lambda\, c : \overline{EE} \bullet \zeta_{EE}^{-1}\zeta_{AE}(c)$);

/* M2: Erzeugen einer neuen Kantenklasse “EB”
M1: Analog zu AB , das Alpha-Ende wird über die ζ-Abbildungen von EB und EE erreicht, das Omega-Ende über EB und BB.
$\text{Edge}_{\text{EdgeClass}}$ D’ = createEdgeClass(“EB”, false, E’, [], B’, [] ▶ $\overline{E}$, $\lambda\, c : \overline{EE} \bullet \zeta_{EE}^{-1}\zeta_{EB}(c)$, $\lambda\, c : \overline{EE} \bullet \zeta_{BB}^{-1}(\omega(\zeta_{EB}(c)))$);

/* M2: Erzeugen einer String-Domain für das Attribut
M1: Keine Angabe nötig */
$\text{Vertex}_{\text{StringDomain}}$ S’ = createStringDomain();

/* M2: Erzeugen der Attribute von BB
M1: “Kopieren”, die Werte werden über die ζ-Abbildung von BB und den Namen des Attributes im Quellschema erreicht.

$\text{Vertex}_{\text{Attribute}}$ Y' = createAttribute("Y", B', S' ► $\lambda\, v \bullet \zeta_{BB}.X$)

Bei der Modelltransformation eines Beispielgraphen in Abbildung 4.4 werden die Elemente im Zielgraphen so gebildet, wie es in der Transformation definiert wurde. In der Abbildung wurden die erzeugten ζ-Abbildungen nochmals dargestellt, die bei der Transformation verwendet wurden, wobei zur besseren Übersicht nur eine ζ-Abbildung pro Schemaelement dargestellt ist.

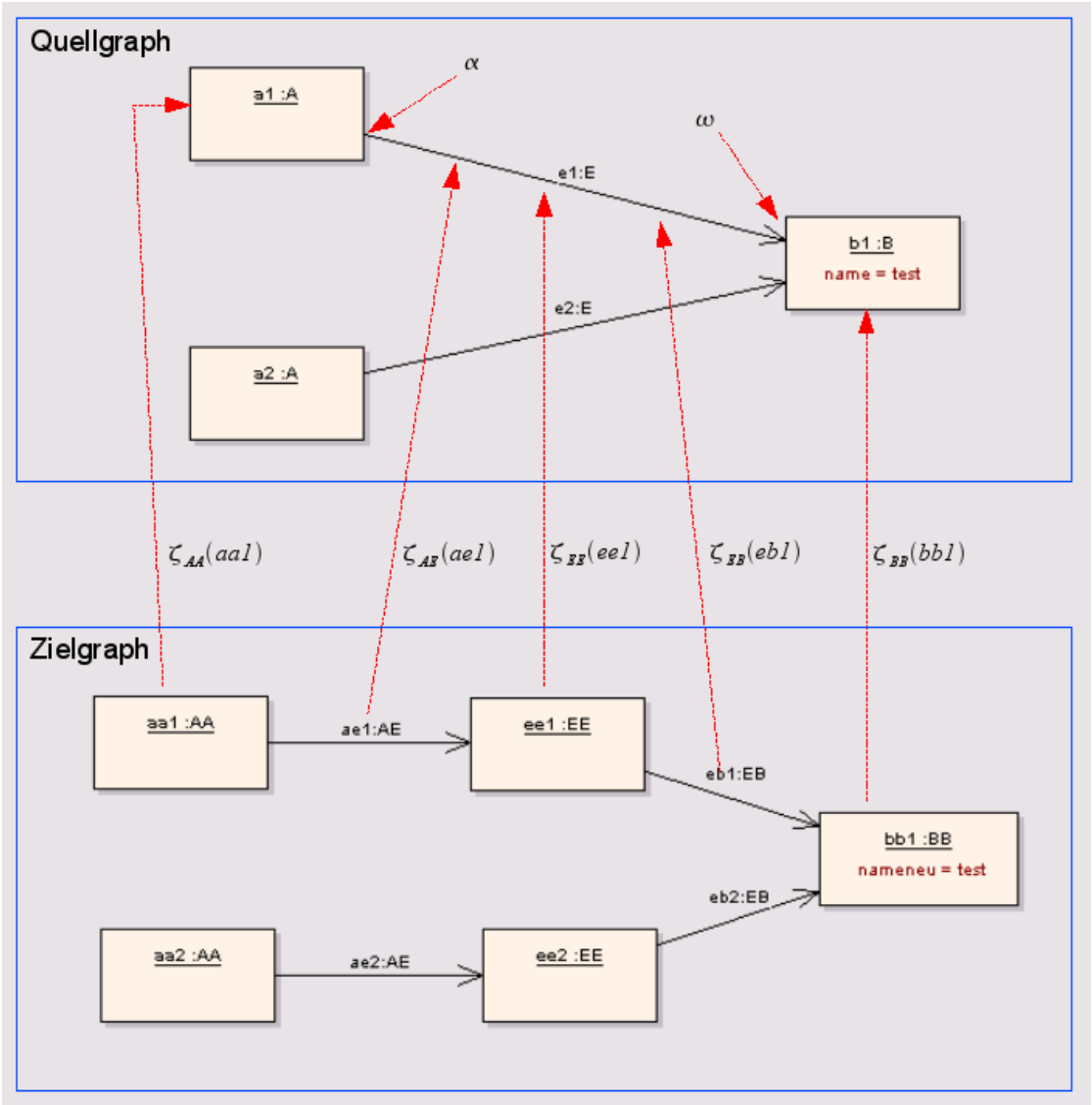

Abbildung 4.4: Beispiel: Quell- und Zielgraph

5 Für GReTLverwendete Bibliotheken

Dieses Kapitel beschreibt die Bibliotheken bzw. Komponenten, die für die Implementierung von GReTLverwendet werden. Dies ist zum einen JGraLab, eine Bibliothek zur Generierung von Schemata und Graphen. Des weiteren wird GReQL2 verwendet, eine Anfragesprache für TGraphen. Beide werden in diesem Kaptiel grundlegend beschrieben, so dass die in den folgenden Kapiteln aufgeführten Beispiele nachvollzogen werden können.

5.1 JGraLab

JGraLab [Kah06] ist eine in Java implementierte Klassenbibliothek, welche die Verwendung von TGraphen als Datenstruktur ermöglicht. Sie bietet eine API an, mit der Graphen sowie die darunter liegenden Schemata manipuliert werden können sowie Methoden zum Speichern und Laden von Schemata und Graphen. Es dient als Grundlage für GReTL, da es alle

JGraLab arbeitet in zwei Stufen, die auch in GReTLübernommen werden müssen. Im ersten Schritt wird ein Schema erstellt, in welchen alle Knoten- und Kantenklassen sowie die Wertebereiche angelegt werden. Das Schema wird anschießend kompiliert. Nach der Kompilierung werden die Graphelemente erzeugt.

GReTLkapselt die Zugriffe auf JGraLab. Dies ist notwendig, da die einzelnen Sprachelemente von GReTLteilweise eine andere Syntax als die entsprechenden Methodenaufrufe von JGraLab. Des weiteren müssen die Operationen von GReTLsowohl die Schemaelemente als auch die Graphelemente mit ein und dem selben Methodenaufruf generieren. Für eine detaillierte Beschreibung des Aufbaus von JGraLab und dessen genaue Funktionsweise sei auf die Diplomarbeit von Steffen Kahle [Kah06] verwiesen.

5.2 GReQL2

GReQL2 [Mar06] ist eine an der Universität Koblenz-Landau entwickelte Anfragesprache für TGraphen. Da die Transformation auf Graphebene Anfragen an die in einem Graphen vorhandenen Elemente (Knoten und Kanten) benötigt, und GReQL2 eine maschinenlesbare textuelle Repräsentation besitzt, ist sie ideal geeignet bei einer Implementierung der Transformationssprache die in [Rhe06] verwendeten formalen Ausdrücke zu ersetzen.

In diesem Kapitel sollen die Sprachelemente von GReQL2 beschrieben werden, die zum Verständnis der Beispiele in den folgenden Kapiteln notwendig sind. Für eine detailliertere Beschreibung der Anfragesprache sei auf die Diplomarbeit von Katrin Marchewka [Mar06] verwiesen, die eine vollständige Sprachbeschreibung enthält.

Generell werden Anfragen an GReQL gestellt, indem die gewünschte Anfrage zusammen mit dem abzufragenden Graphen an eine Instanz des GReQL-Auswerters (*evaluator*) übergeben wird, die Auswertung gestartet wird und anschließend die Ergebnisse abgefragt werden. Dieses wird in Listing 5.1 noch einmal dargestellt. In der ersten Zeile wird der Auswerter instantiiert, der auszuwertende Ausdruck in textueller Form wird als String, der abzufragende Graph als Instanz eines Graphen übergeben. In der zweiten Zeile wird die Auswertung des übergebenen Ausdruckes auf dem Graphen gestartet. In der dritten Zeile wird das Auswertungsergebnis abgefragt.

Listing 5.1: Beispiel: Aufruf von GReQL

```
1  GreqlEvaluator evaluator = new GreqlEvaluator(expression, graph);
2  evaluator.startEvaluation();
3  JValue result = evaluator.getEvaluationResult();
```

Im folgenden werden einige Ausschnitte der GReQL2-Syntax beschrieben, die für das Verständnic der Beispiele notwendig sind.

Knotenmengen werden durch `V{Typbezeichner}`, Kantenmengen durch `E{Typbezeichner}` repräsentiert. `Typbezeichner` steht dabei eine Knoten- bzw. Kantenklasse. Generell enthalten sie alle Knoten bzw. Kanten des jeweiligen Typs sowie deren Untertypen. Mehrere Typbezeichner können innerhalb der geschweiften Klammern durch Kommata getrennt angegeben werden. Die Knoten- und Kantenmengen können durch Angabe von Typbeschreibungen eingeschränkt werden. Folgt dem Typbezeichner ein Ausrufezeichen (“!”), werden die Untertypen nicht bei der Bestimmung der Menge berücksichtigt. Geht dem Typbezeichner ein Zirkumflex (“^“) voraus, so wird die angegebene Typklasse von der Menge ausgeschlossen. Die Typbeschreibungen können auch kombiniert verwendet werden.

Innerhalb dieser Arbeit werden nur einfache Anfragen (*SimpleQuery*) von GReQL verwendet. Eine solche Anfrage beginnt mit dem Schlüsselwort `from`, welchem eine Liste von Deklarationen folgt. Innerhalb der Deklarationsliste werden Variablen deklariert, über die bei der Auswertung des Ausdruckes iteriert wird. Eine Deklaration von `v:V{Person}` iteriert bei der Auswertung über alle Instanzen der Knotenklasse `Person` sowie deren Untertypen. Optional folgt auf die Deklarationsliste durch Angabe von `with` ein Prädikat, dem die deklarierten Variablen entsprechen müssen. Zum Beispiel kann durch Angabe von `with v.firstname = "Klaus"` die Variable `v` auf die Instanzen eingeschränk werden, deren Attribut `firstname` gleich "Klaus" ist. Anschließend wird eine Report-Klausel angegeben, die eine Liste von Ausdrücken enthält, die in die Ergebnismenge aufgenommen werden sollen. Die Arten der Report-Klausel werden nach dem nächsten Beispiel genauer beschrieben. Eine einfache Anfrage endet immer mit dem Schlüsselwort `end`. Das folgende Beispiel zeigt noch einmal zusammenfassend eine Anfrage, die alle Knoten der Knotenklasse "Angestellter" zurückliefert, deren Attribut "Vorname" gleich "Klaus" ist:

Listing 5.2: Beispiel: GReQL-Ausdruck

```
1  from v:V{Angestellter}
2  with v.firstname = "Klaus"
3  report v
4  end
```

Die Report-Klausel beschreibt, wie die Ergebnismenge aufgebaut sein soll. Möglich ist die Angabe von "report" für eine einfache Menge, "reportBag" für eine Multimenge, "reportSet" für eine Menge von Werten (kein Wert kommt doppelt vor) oder "reportMap" für Abbildungen. Die Ergebnismenge kann auch weiter beschrieben werden, indem der Ausdruck hinter ihr die Ergebnisse noch weiter beschreibt. Sollen im Listing 5.2 zum Beispiel nicht die Knoten, sondern die Werte des Attributs "lastname" zurückgeliefert werden, würde die dritte Zeile durch "`report v.lastname`" geändert werden.

Das Beispiel zeigt gleichzeitig, wie auf die Attribute der Elemente zugegriffen werden kann. Diese werden wie im Beispiellisting in Zeile 2 über den Punkt-Operator (".") angesprochen.

Des weiteren können Funktionen auf die Elemente angewendet werden. Eine Funktionsanwendung wird angegeben, indem der Funktionsname angegeben als Präfix des Ausdruckes angegeben wird und der Ausdruck in Klammern notiert wird. Eine Anwendung dafür ist beispielsweise, von einer Kante den Anfangs- oder Endknoten zu bestimmen. Sei `e` eine Menge von Kanten, so wird mit `startVertex(e)` der Anfangsknoten der Kante bestimmt, mit `endVertex(e)` der Endknoten.

Im Verlauf dieser Arbeit wurde GReQL (und JGraLab) um die Möglichkeit erwei-

tert, injektive Abbildungen (Maps) verarbeiten zu können. Die Abbildungen werden für GReTLz.B. benötigt, um einer Menge von Kanten ihre jeweiligen Start- bzw. Endknoten zuordnen zu können, was später in dieser Arbeit beim Erzeugen von Instanzen von Kantenklassen noch dargestellt wird. In GReQL werden diese über das schon erwähnte "reportMap" zurückgeliefert, welches im Gegensatz zu den anderen Report-Klauseln zwei Werte erwartet, dies sind jeweils das Urbild und der zugehörige Wert. Die Abfrage einer Abbildung aller Knoten einer Knotenklasse auf deren Startknoten würde zum Beispiel wir folgt ausgedrückt:

```
from e:E{Kantenklasse} reportMap e, startVertex(e) end
```

6 Implementierung der GReTL-Klassen

In diesem Kapitel werden die Konzepte zur Implementierung von Transformationen in Java dargestellt. Einschränkend zu den bisher in Kapitel 4 erzeugten elementaren Operationen, welche im Anhang A vollständig aufgeführt sind, werden bei der Implementierung von GReTL nur Operationen eingesetzt, mit denen das Zielschema komplett neu aufgebaut wird, da JGraLab bisher keine Möglichkeiten zur Veränderung bestehender Schemata besitzt.

Im ersten Abschnitt wird auf die Implementierung der GReTL-Klassen in Bezug auf die Schematransformation eingegangen sowie der Gesamtüberblick über alle bestehenden Klassen gegeben, woraufhin im zweiten Abschnitt ein Überblick über die implementierten elementaren Transformationsoperationen gegeben wird. Anschließend wird auf die Besonderheiten bei der Implementierung der Graphtransformation eingegangen. Zum Abschluss werden die Details behandelt, die auf Grund von Spezialisierung bzw. Generalisierung berücksichtigt werden müssen. Die Implementierungen der Transformationen an sich aus Benutzersicht wird im nächsten Kapitel beschrieben.

6.1 Implementierung der Schematransformation

Ziel der Implementierung ist es ein neues Zielschema aufzubauen. Mit Hilfe des aufgebauten Zielschemas und durch die Verwendung von semantischen Ausdrücken soll ein neuer Zielgraph aufgebaut werden.

Der Benutzer soll in der Lage sein, selbst Transformationen zu definieren und deren Ausführung zu starten. Dies erfordert eine Unterteilung in die Definition der elementaren Transformationsoperationen, die Definition der auszuführenden Transformation und die Ausführung der Transformation an sich. Die elementaren Transformationsoperationen sowie die Funktionalität zur Ausführung der Transformation werden durch GReTL bereitgestellt und in diesem Kapitel beschrieben. Die bestehenden Klassen stellen ein Framework dar, welches die Definition beliebiger Transformationen zulassen soll. Die

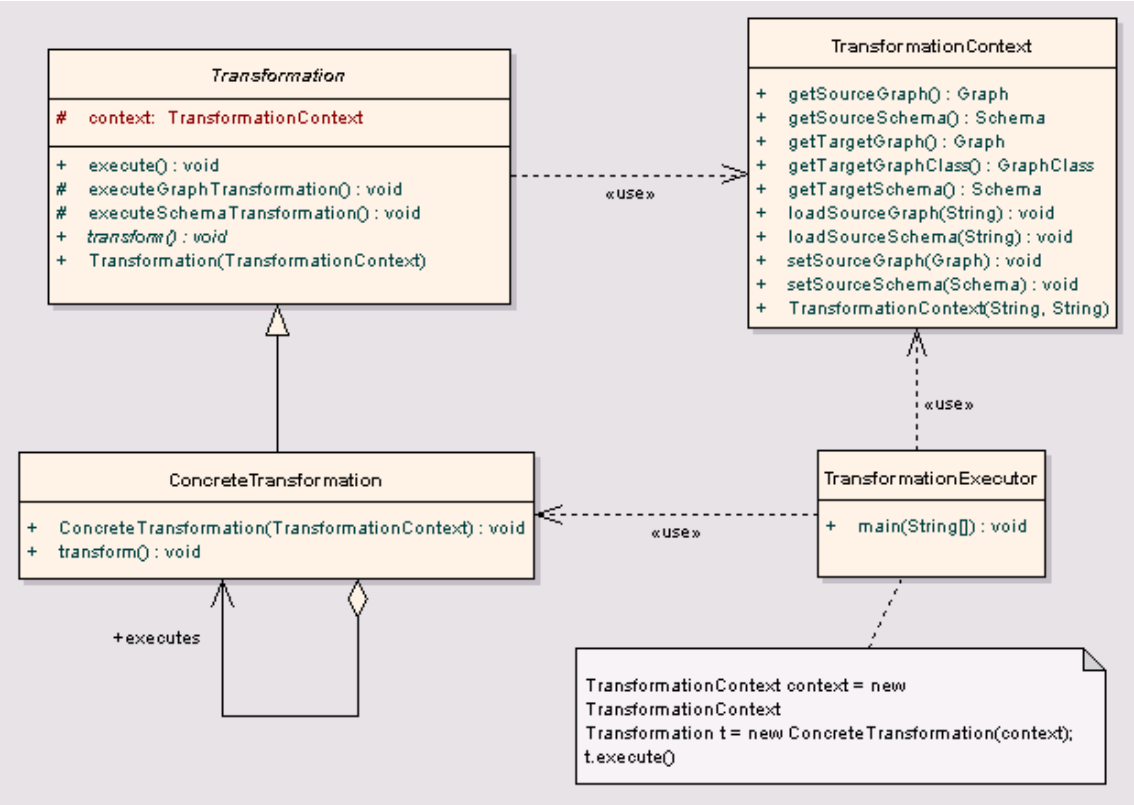

Abbildung 6.1: Vereinfachter Aufbau der Klassen von GReTL

Definition der auszuführenden Transformation durch den Benutzer wird im nächsten Kapitel erläutert.

Abbildung 6.1 zeigt den Aufbau von GReTL in einem vereinfachten Klassendiagramm. Die einzelnen Klassen werden im folgenden beschrieben.

Da alle Transformationen auf der gleichen Menge an elementaren Transformationsoperationen basieren, ist es sinnvoll, diese in einer gemeinsamen, abstrakten Klasse zu definieren, der `Transformation`-Klasse. Alle durch einen Benutzer definierten Transformationen werden von dieser Basisklasse abgeleitet und haben dadurch direkten Zugriff auf die elementaren Transformationsperationen. Die `Transformation`-Klasse definiert eine abstrakte Methode `transform()`, in welcher der Benutzer in seiner Klasse die Transformation an sich definiert (siehe Kapitel 7). Die zur Verfügung stehenden elementaren Transformationsoperationen sind in Anhang B noch einmal detailliert beschrieben, Abschnitt 6.2 gibt einen Überblick über ihren Aufbau.

In der Regel soll eine Transformation Daten aus einer Datenquelle abrufen können.

Diese Datenquelle ist ein Quellschema und ein dazu gehörender Quellgraph. Das Quellschema soll zu einem Zielschema transformiert werden, ein dazu passender Quellgraph analog dazu in einen Zielgraphen überführt werden. Diese zwei Schemata und zwei Graphen stellen zusammen den Kontext der Transformation dar. Da der Kontext innerhalb einer Transformation und eventuell darin aufgerufener weiterer Transformationen gleich ist, wird dieser in einer eigenen Kontextklasse `TransformationContext` abgelegt. Das Kontext-Objekt wird einer Transformation als erster Parameter im Konstruktor übergeben. Es enthält Getter und Setter für die einzelnen Schemata und Graphen. Zusätzlich enthält die Klasse Methoden, mit denen Schemata oder Graphen aus Dateien geladen oder gespeichert werden können. Eine genaue Beschreibung der verfügbaren Methodenaufrufe ist in Anhang C aufgeführt, die genaue Verwendung wird im Abschnitt 7.2 erläutert.

Die Ausführung einer Transformation erfolgt in zwei Schritten. Dies ist technisch notwendig, da ein Graph eine Instanz eines Schemas ist und daher der Zielgraph erst dann instantiiert werden kann, wenn das Zielschema vollständig aufgebaut ist. Aus diesem Grund wird die `transform()`-Methode zwei mal ausgeführt. Im ersten Durchlauf sollen die Operationen ausschließlich das Zielschema aufbauen (Schematransformation), im zweiten Durchlauf ausschließlich den Zielgraph (Graphtransformation). Damit dieser zweifache Aufruf nicht in jeder Transformation vom Benutzer neu implementiert werden muss, enthält die `Transformation`-Klasse eine Methode `execute()`, welche die Ausführung der beiden aufeinander folgenden Transformationsschritte automatisch durchführt. Zusätzlich setzt die Methode bei jedem Durchlauf ein Flag im Kontext-Objekt, mit dessen Hilfe die aufgerufenen elementaren Operationen feststellen können, ob sie auf dem Zielschema oder dem Zielgraph arbeiten sollen. Darüber hinaus sorgt die `execute()`-Methode dafür, dass sowohl das Zielschema als auch der Zielgraph zum richtigen Zeitpunkt instantiiert werden. Die Sequenz der Transformationsausführung ist in Abbildung 6.2 noch einmal kurz dargestellt.

An dieser Stelle soll schon einmal gezeigt werden, wie die Ausführung einer Transformation aus Benutzersicht aussieht, da dies auch dem Verständnis des gesamten Systems dient. In Listing 6.1 ist ein Beispiel eines Transformationsaufrufs dargestellt, bei dem das Kontext-Objekt vor dem Aufruf gesetzt wird:

Listing 6.1: Beispiel: Aufruf einer Transformation (Pseudocode)

```
1  class TransformationExecutor {
2     public static void main(String[] args) {
3        TransformationContext context = new TransformationContext("targetSchemaName",
4          "targetGraphClassName");
5        context.loadSourceGraph("filename");
6        Transformation t = new ConcreteTransformation(context);
7        t.execute();
8        Schema targetSchema = context.getTargetSchema();
9        Graph targetGraph = context.getTargetGraph();
10    }
```

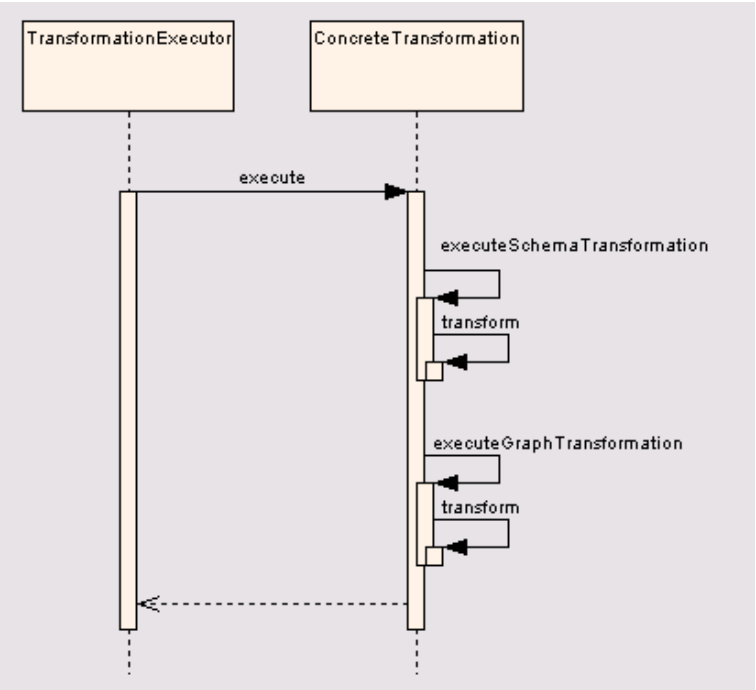

Abbildung 6.2: Interner Ablauf eines Transformationsaufrufes

```
11  }
```

Ein Aufruf einer Transformation erfolgt also mit Hilfe der folgenden Schritte, wie in Listing 6.1 dargestellt:

- Instantiieren des Kontext-Objektes, der Name des Zielschemas und der Name der Graphklasse des Zielschemas werden übergeben (Zeile 3)
- Initialisierung des Kontext-Objektes mit Quellschema und Quellgraph (Zeile 5). Die Methode zum Laden des Quellgraphen lädt dabei automatisch das erforderliche Quellschema mit, es ist ebenfalls in der Datei enthalten.
- Instantiierung des gewünschten Transformationsobjektes, der Kontext wird im Konstruktor übergeben (Zeile 6).
- Aufruf der Transformation durch Aufruf der `execute()`-Methode des Transformationsobjektes (Zeile 7).
- Abfrage des erzeugten Zielschemas (Zeile 8) sowie des erzeugten Zielgraphen (Zeile 9).

6.2 Beschreibung der elementaren Transformationsoperationen

Die elementaren Transformationsoperationen, welche in Kapitel 4 hergeleitet wurden, sind innerhalb der `Transformation`-Klasse implementiert. Im Rahmen dieser Arbeit soll das Zielschema neu aufgebaut, aber nicht verändert werden, aus diesem Grund sind nur die jeweiligen create-Operationen implementiert worden. Anhang B enthält eine vollständige Übersicht über die implementierten Methoden und eine detaillierte Beschreibung ihrer Parameter.

Durch die aktuelle Implementierung von JGraLab ergeben sich einige Unterschiede und Besonderheiten für die Implementierung der entsprechenden Operationen entgegen den Operationssignaturen, die bei der Analyse des JGraLab Meta-Meta-Modells in Kapitel 4 aufgestellt wurden:

- In JGraLab existieren keine getrennten Methodennamen zum Erzeugen einer Spezialisierung einer Knotenklasse oder einer Kantenklasse. Aus diesem Grund werden die Operationen `createSpecializesVertexClass()` und `createSpecializesEdgeClass()` analog zur Namensgebung in JGraLab zu `addSuperClass()` umbenannt.
- Innerhalb von JGraLab existiert pro Schema jeweils nur eine Domain der Typen String, Long, Int, Double und Boolean, welche automatisch angelegt werden. Die zugehörigen Operationen `createStringDomain()`, `createLongDomain()`, `createIntDomain()`, `createDoubleDomain()` und `createBooleanDomain()` geben diese Domains zurück, ohne eine neue Instanz zu erzeugen. Die Übergabe des Domainnamens kann bei diesen Methoden entfallen, da sie bei diesen Domains in JGraLab fest vorgegeben sind.
- JGraLab automatisiert das Anlegen von bestimmten Schema-Elementen bzw. Schema-Assoziationen. Da nur ein Schema verwaltet werden soll, kann die Operation `createSchema()` entfallen. Ebenso können die Operation `createGraphClass()` sowie `createContainsGraphClass()` entfallen, da nur eine Graphenklasse im Zielschema angelegt wird und diese automatisch dem Schema zugeordnet wird. Des weiteren enthält jedes Schema von sich aus ein DefaultPackage, so dass die Operation `createContainsDefaultPackage()` nicht benötigt wird. Beim Anlegen einer neuen Domain der Typen Enum, List, Record oder Set ordnet JGraLab diese automatisch dem richtigen Package im Schema zu, so dass die manuelle Zuordnung mit `createContainsDomain()` entfallen kann.

Der Grundaufbau einer elementaren Transformationsoperation ist in Listing 6.2 beispielhaft in Pseudocode dargestellt. Jede Operation enthält eine Fallunterscheidung (Zeile 2), anhand derer sie entweder das Zielschema oder den Zielgraph aufbaut. Der Methode werden immer alle definierten Parameter übergeben, so dass es beim Aufruf keine

Rolle spielt, ob gerade das Schema oder der Graph transformiert wird. Diese Vorgehensweise erlaubt es, dass die Methoden keine unterschiedlichen Signaturen für die beiden Transformationsschritte benötigen und die Definition einer Transformation in der `transform()`-Methode für den Benutzer erleichtert wird. Da im Verlauf einer Transformation die erzeugte Elementklasse (Knotenklasse, Kantenklasse, Domain, usw.) noch benötigt werden könnte, sind alle Methoden so implementiert, dass sie immer das zu bearbeitende Schemaelement (Zeile 5 bzw. 8) zurückgeben, unabhängig davon in welchem Transformationsschritt sie sich gerade befinden.

Listing 6.2: Aufbau einer elementaren Transformationsoperation (Pseudocode)

```
1  public elementType elementaryOperation(String elementName) {
2     if (context.isExecutingGraphTransformation) {
3        element = context.getTargetSchema().getElement(elementName);
4        // Operation auf dem Zielgraph ausfuehren
5        return element;
6     } else {
7        // Operation auf dem Zielschema ausfuehren, element erzeugen
8        element = context.getTargetSchema().createElement(elementName);
9        return element;
10    }
11 }
```

Die Parameter der Operationen enthalten auch die semantischen Ausdrücke. Diese semantischen Ausdrücke werden für die Graphtransformation benötigt. Die Anzahl der zusätzlichen Parameter für semantische Ausdrücke ist durch die Signatur der Operationen vorgegeben, wie sie in der Analyse erhoben wurden. Die Operation zum Anlegen von Knotenklassen (`createVertexClass()`) benötigt zum Beispiel nur einen zusätzlichen Parameter, der die zu erzeugenden Instanzen beschreibt, die Operation zum Anlegen einer Kantenklasse (`createEdgeClass()`) darüber hinaus noch die Angabe der Anfangs- und Endknoten ihrer jeweiligen Instanzen. Viele Operationen erfordern keinen zusätzlichen Parameter, da sie gar nicht auf Graphebene arbeiten. Dies sind alle Operationen zum Erzeugen von Packages, Domains, oder Spezialisierungen. Die genauen Parameter der jeweiligen Operationen werden im nächsten Abschnitt beschrieben, in dem die Besonderheiten bei der Implementierung der Graphtransformation betrachtet werden.

Insgesamt sind die folgenden Operationen in der `Transformation`-Klasse vorhanden, an dieser Stelle soll aber nur ein Überblick über die Operationen und die vorhandenen Parameter gegeben werden, eine detaillierte Beschreibung sämtlicher Operationen mit Beschreibung aller Parameter ist in Anhang B aufgeführt.

Zum Erzeugen von Knotenklassen wird die Methode `createVertexClass()` verwendet. Dieser muss übergeben werden, welchen Namen die neue Knotenklasse haben soll und ob sie als abstrakte Klasse definiert werden soll. Die Methode erwartet einen semantischen Ausdruck, welcher die Urbildmenge für die zu erzeugenden Instanzen beschreibt.

Kantenklassen können auf unterschiedliche Weise erzeugt werden, je nachdem um welche Art einer Kante es sich handelt. Zu diesem Zweck stehen die Methoden `createAggregationClass()` für Aggregationen, `createCompositionClass()` für Kompositionen und `createEdgeClass` für Assoziationen zur Verfügung. Diese Methoden sind jeweils mit mehreren unterschiedlichen Signaturen implementiert, die analog zu den in JGraLab vorhandenen Methoden ermöglichen, einige Parameter wie z.B. die Rollennamen der Assoziationen offen zu lassen. Der Name der neuen Kantenklasse sowie die Angabe, ob die neue Kantenklasse abstrakt sein soll müssen übergeben werden. Die Methoden erwartet drei semantische Ausdrücke als zusätzliche Parameter, welche die Urbildmenge der zu erzeugenden Instanzen beschreibt, sowie zwei Abbildungen zur Berechnung der jeweiligen Anfangs- und Endknoten einer neuen Kante.

Attribute werden mit Hilfe der Methode `createAttribute()` erzeugt, welche den Namen des Attributes, die dazu gehörende Elementklasse und die Domain erwartet. Diese Methode ist ebenfalls mit zwei verschiedenen Signaturen implementiert, so dass die Angabe des Domaintyps entweder als Objekt oder als String (für einfache Domains wie zum Beispiel String oder Integer) erfolgen kann. Die Methode erwartet einen semantischen Ausdruck, welcher die Werte beschreibt, die den Attributen zugewiesen werden sollen.

Spezialisierungen bzw. Generalisierungen von Knoten- oder Kantenklassen werden durch Aufruf der Methode `addSuperClass()` erzeugt, die entweder zwei Knotenklassen oder zwei Kantenklassen erwartet. Wichtig ist, dass alle Attribute und Assoziationen, die in einer Oberklasse definiert werden, automatisch den Kindklassen zur Verfügung stehen. Im Gegensatz zu allen anderen Methoden liefert diese Operation kein Element zurück. Diese Methode erwartet keinen semantischen Ausdruck.

Packages bzw. Subpackages werden mit den Methoden `createPackage()` bzw. `createSubPackage()` erzeugt. Sie erwarten jeweils den Namen des zu erzeugenden Package bzw. zusätzlich den Namen des übergeordneten Packages. Alle Packages, die nicht mittels `createSubPackage()` erzeugt werden, besitzen automatisch ein gemeinsames übergeordnetes Package, das Default-Package. Beide Methoden erwarten keinen semantischen Ausdruck.

Einfache Domains werden mit Hilfe der Methoden `createBooleanDomain()`, `createDoubleDomain()`, `createIntegerDomain()`, `createLongDomain()` bzw. `createStringDomain()` erzeugt, die keine Parameter erwarten. Diese Domains können nur ein mal pro Schema vorkommen und geben immer diese Instanz zurück. Komplexe Domains werden durch `createEnumDomain()`, `createListDomain()`, `createRecordDomain()` oder `createSetDomain()` erzeugt, die die Übergabe des Namens und einer Menge von Elementen der zu erzeugenden Domain erwarten. Alle Domainspezifischen Methoden erwarten keine semantischen Ausdrücke.

6.3 Implementierung der Graphtransformation

Bei der Transformation auf Graphebene wird der Zielgraph aufgebaut. Die für jedes Schemaelement anzulegenden Instanzen (bzw. bei Kanten auch die Abbildungen auf die jeweiligen Start- und Endknoten) müssen zu diesem Zweck explizit angegeben werden. Die Angabe der Instanzmengen kann auf zwei verschiedene Arten erfolgen. Die erste Möglichkeit ist, die Urbildmengen mit Hilfe von JValues direkt zu übergeben. JValue ist eine zu GReQL2 gehörende Container-Klasse, die verschiedenste Datentypen speichern kann und die auch vom GReQL-Evaluator zurückgegeben wird. Die zweite Möglichkeit ist die Angabe von semantischen Ausdrücken als Strings, welche die Urbildmengen der anzulegenden Instanzen mit Hilfe von GReQL2-Ausdrücken beschreiben. Der genaue Sprachaufbau von GReQL2 ist in [Mar06] genau erläutert und wird hier nicht weiter vertieft. Eine Einführung in GReQL2, welche die grundlegenden Sprachelemente, die zum Verständnis der Beispiele in dieser Arbeit benötigt werden, ist in Kapitel 5.2 zu finden. Die Semantik der einzelnen Elemente ist in Kapitel 4.2 bereits beschrieben worden.

Die zwei verschiedenen Arten, die zu erzeugenden Instanzen anzugeben, sorgen bei der Implementierung der `Transformation`-Klasse dafür, dass jede elementare Transformationsoperation, deren Signatur semantische Ausdrücke enthält, zwei mal vorhanden ist. Die erste Variante akzeptiert JValues als Eingabeparameter und führt die Operation durch. Die zweite Variante akzeptiert GReQL2-Ausdrücke als Parameter. Diese werden mit GReQL2 ausgewertet und die erste Variante wird mit den bei der Auswertung entstehenden JValues aufgerufen.

Es gibt drei Arten von Schemaelementen, für die im Zielgraph Instanzen angelegt werden können: Knotenklassen (VertexClass), Kantenklassen (EdgeClass) und Attribute (Attribute). Zusätzlich können auf Schemaebene Vererbungsbeziehungen angelegt werden, deren Auswirkungen auf die Graphtransformation im nächsten Unterkapitel genauer beschrieben werden. Für jede Art von Schemaelement müssen zur Beschreibung der anzulegenden Instanzen unterschiedliche semantische Ausdrücke angegeben werden, welche in diesem Kapitel genauer beschrieben werden. Eine Ausnahme stellen die Wertebereiche (Domains) dar, für die keine semantischen Ausdrücke angegeben werden. Deren Instanzen sind die Werte, die den Attributen zugeordnet werden. Die Menge der möglichen Instanzen wird dabei bereits auf Schemaebene festgelegt. Die Instanzen an sich werden erst beim Anlegen der Attribute erzeugt und nicht schon beim Anlegen des Wertebereiches.

Für jede im Zielschema angelegte Knoten- und Kantenklasse werden die in Kapitel 4.2 eingeführten bijektiven Abbildungen von der Bildmenge in die Urbildmenge erzeugt, welche jeder erzeugten Instanz im Zielgraphen (Bild, img) ihr erzeugendes Element

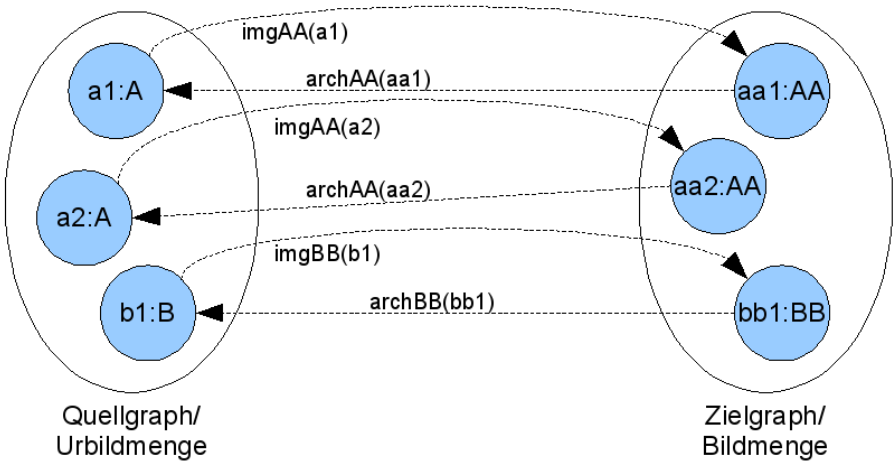

Abbildung 6.3: arch- und img-Abbildungen

(Urbild, arch) zuordnet. Das Anlegen der Abbildung geschieht automatisch ohne Zutun des Benutzers. Diese Abbildungen stehen dem Benutzer innerhalb der semantischen Ausdrücke wieder zur Verfügung. Auf diese kann er mit Hilfe von archXXX() für die Urbild-Abbildung bzw. imgXXX() für die Bild-Abbildung zugreifen. XXX steht dabei für den Namen der Knoten- oder Kantenklasse im Zielschema, auf die sich die Abbildung bezieht.

Die Abbildungen werden in der Klasse `TransformationContext` mit Hilfe einer zweistufigen JValueMap gespeichert. Diese speichert in der ersten Ebene die Zuordnung des Abbildungsnamens (also imgXXX bzw. archXXX) auf die eigentliche Abbildung, welche wiederum aus einer JValueMap besteht. Diese ordnet im Falle von archXXX den erzeugten Knoten- bzw. Kanteninstanzen des Zielgraphen ihr jeweiliges Urbild (also Elemente einer Menge oder Instanzen aus dem Quellgraphen) zuordnet (siehe auch Abbildung 6.3). imgXXX ist jeweils die Umkehrfunktion von arch und ordnet den Elementen der Urbildmenge die Instanzen im Zielgraph (Bildmenge) zu, die aus ihnen entstanden sind.

Da die Abbildungen nicht nur für die interne Zuordnung verwendet werden können, sondern auch für die Berechnung von Mengen durch GReQL-Ausdrücke, müssen diese innerhalb des GReQL-Evaluators für den Benutzer verfügbar sein. Zu diesem Zweck werden alle Abbildungen für den Benutzer transparent an den GReQL-Evaluator übergeben und in die Abfrage eingebaut, so dass dieser sich bei der Angabe der semantischen Ausdrücke nicht manuell um deren Integration kümmern muss.

Anlegen von Knotenklassen

Beim Anlegen einer Knotenklasse muss eine Menge angegeben werden, die als Urbildmenge zur Erzeugung der Instanzen im Zielgraphen dient. Für deren Beschreibung ist genau ein semantischer Ausdruck notwendig, mit dem entweder eine Urbildmenge direkt in einem JValueSet übergeben wird (siehe Beispiel 1) oder der die Urbildmenge mit Hilfe eines GReQL2-Ausdrucks aus dem Quellgraphen abfragt (siehe Beispiel 2). Während der Graphtransformation werden automatisch die Abbildungen angelegt, welche den neu erzeugten Knoten im Zielgraphen ihr Urbild zuordnet. Die Abbildung erhält den Namen der erzeugten Knotenklasse.

Beispiel 1: Anlegen einer nicht abstrakten Knotenklasse “A” mit drei Instanzen. Als Urbildmenge wird eine einfache Menge übergeben, welche die natürlichen Zahlen eins bis drei enthält:

Listing 6.3: Anlegen von Knotenklassen mit einer einfachen Menge

```
1  JValueSet a = new JValueSet();
2  a.add(new JValue(1));
3  a.add(new JValue(2));
4  a.add(new JValue(3));
5  createVertexClass("A", false, a);
```

Der Operationsaufruf in Listing 6.3 erzeugt während der Schematransformation im Zielschema eine Knotenklasse “A”. Während der Graphtransformation werden drei Instanzen dieser Klasse erzeugt, für jedes Element der Urbildmenge genau eins. Diesen Instanzen wird in der Zeta-Abbildung als Urbild jeweils das erzeugende Element der Menge a=$\{1,2,3\}$ zugeordnet.

Beispiel 2: Anlegen einer nicht abstrakten Knotenklasse “A”. Die Urbildmenge für die zu erzeugenden Instanzen wird aus dem Quellgraphen abgefragt. Dort sollen alle Instanzen der Knotenklasse “B” in die Menge einbezogen werden:

Listing 6.4: Anlegen von Knotenklassen durch Abfrage des Quellgraphen

```
1  createVertexClass("A", false, "V{B}");
```

Dieser Operationsaufruf erzeugt wie der vorherige im Zielschema eine Knotenklasse “A”. Hier wird als Urbildmenge die Menge der Instanzen der Knotenklasse “B” im Quellgraphen verwendet. Die Anzahl der erzeugten Instanzen im Zielgraphen hängt von der Anzahl der Instanzen der abgefragten Knoten des Quellgraphen ab. In der Abbildung wird jeder neu erzeugten Instanz der Knotenklasse “A” wieder ihr erzeugendes Element zugeordnet, also eine Instanzen der Knotenklasse “B” im Quellgraphen.

Anlegen von Kantenklassen

Beim Anlegen einer Kantenklasse müssen zusätzlich zur Urbildmenge der zu erzeugenden Kanten noch zwei Funktionen angegeben werden, welche für jede zu erzeugende Instanz die jeweiligen Urbilder der Anfangs- und Endknoten berechnen. Aus diesem Grund benötigt die Beschreibung drei semantische Ausdrücke. Die Angabe der Urbildmenge erfolgt analog zur Vorgehensweise bei Knotenklassen als erster Ausdruck. Bei der Angabe der beiden Funktionen muss jeweils ein Ausdruck verwendet werden, der ausgehend von der Urbildmenge der neu zu erzeugenden Kanten diesen die Urbilder ihre jeweiligen Startknoten (zweiter Ausdruck) bzw. Endknoten (dritter Ausdruck) zuordnet.

An dieser Stelle ist auch ein wichtiger Unterschied zur Beschreibung der semantischen Ausdrücke für die Funktionen zur Bestimmung der Start- und Endknoten aus der Diplomarbeit von Florian Rheindorf [Rhe06]. Florian Rheindorf verwendet für die Beschreibung der beiden Funktionen grundsätzlich die Bildmenge, sprich eine Abbildung vom Zielgraphen in den Zielgraphen. Ohne Einschränkungen ist es jedoch möglich ausschließlich Funktionen auf den Urbildmengen anzugeben, also zum Beispiel eine Abbildung vom Quellgraphen in den Quellgraphen, da über die gespeicherten Abbildungen die jeweiligen Bildmengen im Zielgraphen berechnet werden können. Welche der Abbildungen dafür benutzt werden muss, geht aus den Namen der Knotenklassen am jeweiligen Kantenende hervor. Genau diese Eigenschaft wird in dieser Arbeit verwendet, da die entstehenden Funktionen wesentlich kompakter sind.

Beispiel: Aus dem Quellgraphen soll eine nicht abstrakte Kantenklasse “E” kopiert werden. Sowohl im Quell- als auch im Zielschema soll das from-Ende der Kantenklasse auf die Knotenklasse “A” zeigen, das to-Ende auf die Knotenklasse “B”. Aus Gründen der Übersichtlichkeit sind keine Multiplizitäten oder Rollennamen angegeben.

```
1  createEdgeClass("E", "A", "B", false,
2     "E{E}",
3     "from e:E{E} reportMap e,startVertex(e) end",
4     "from e:E{E} reportMap e,endVertex(e) end");
```

Eine Angabe der semantischen Ausdrücke mit einem JValueSet für die Urbildmenge und zweier JValueMaps für die Abbildungen sind ebenfalls möglich.

Anlegen von Attributen

Beim Anlegen von Attributen muss ein semantischer Ausdruck angegeben werden. Dieser Ausdruck ist eine Funktion, welche dem Urbild jeder Instanz einer Knoten- oder Kantenklasse im Zielgraphen genau einen zugehörigen Attributwert zuordnet. Aus die-

sem Grund muss die Urbildmenge, aus der die Instanzen der erzeugten Knoten- bzw. Kantenklasse, für die das Attribut angelegt wird, als Urbildmenge der Funktion eingebunden werden. Der Ausdruck muss eine Abbildung (Map) zurückliefern.

Beispiel: Die Knotenklasse “A” ist aus dem Quellschema in das Zielschema kopiert worden, auf Graphebene wurden alle Instanzen kopiert. Der Knotenklasse “A” im Zielgraphen wird ein Attribut “name” hinzugefügt (ebenfalls kopiert). Die Wert des Attributes “name” aller Instanzen der Knotenklasse sollen von den Instanzen der Knotenklasse “A” im Quellgraphen abgefragt werden.

Listing 6.5: Anlegen von Attributen durch Abfrage des Quellgraphen

```
1  createAttribute("A", "name", "from v:V{A} reportMap v, v.name");
```

In diesem Beispiel wird eine Abbildung von der Urbildmenge der Instanzen der Knotenklasse “A” auf die Attributwerte des Attributs “name” der jeweiligen Elementes der Urbildmenge erzeugt.

Ebenso ist die der Aufruf der Methode mit Angabe einer JValueMap möglich.

6.4 Auswirkungen der Vererbung auf die Implementierung der GReTL-Klassen

Im Rahmen der Transformationen können *Spezialisierungen bzw. Generalisierungen* für Knotenklassen und Kantenklassen erzeugt werden. Diese sorgen dafür, dass Attribute, die in einer Oberklasse erzeugt worden sind, an die Unterklassen vererbt werden. Während der Schematransformation wird diese Zuordnung durch JGraLab automatisch beim Anlegen der Beziehung vorgenommen, so dass GRaTL hier nichts weiter beachten muss und die Angabe der Unter- und Oberklasse beim Aufruf von `addSuperClass(subclass, superclass)` ausreichend ist.

Während der Graphtransformation müssen beim Hinzufügen einer Vererbungsbeziehung *die img- und arch-Abbildungen sämtlicher Oberklassen verändert werde*n. Dabei müssen die jeweiligen Abbildungen der Unterklasse zu den Abbildungen der Oberklasse (und allen deren Oberklassen) hinzugefügt werden. Dieses Hinzufügen erledigt die `Transformation`-Klasse automatisch. Nur so ist es möglich, beim Anlegen eines Attributes in einer Oberklasse dieses auch den Instanzen sämtlicher Unterklassen hinzuzufügen, da es zur Zeit keine nicht belegten Attribute geben soll. Auch dieses Erweitern der Abbildungen benötigt keine weiteren Angaben durch den Benutzer. Beim Hinzufügen einer Abbildung zu der einer Oberklasse kann allerdings die Situation entstehen, dass Knoten

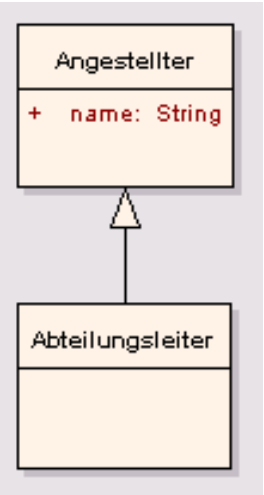

Abbildung 6.4: Beispiel einer Vererbungsbeziehung auf Schemaebene

oder Kanten durch das selbe Urbild erzeugt worden sind oder dass das selbe Graphelement in der Bildmenge liegt. Da von den Abbildungen gefordert ist, dass diese bijektiv sein sollen, wird innerhalb dieser Arbeit angenommen, dass die zu vereinigenden Mengen disjunkt sind. Diese Annahme gilt jedoch nur vorübergehend und nur für diese Arbeit, da diese Situation in der Praxis vorkommen kann (und wahrscheinlich auch wird). Sollte das Vereinigen der Abbildungsmengen dennoch dennoch die Bijektivität gefährden, so gibt die `Transformation`-Klasse eine Fehlermeldung aus.

Den Attributen von Knoten- bzw. Kanten werden während der Graphtransformation Werte zugewiesen. Dabei können allerdings *zwei verschiedene Situationen* entstehen, die einem Beispiel verdeutlicht werden sollen. Abbildung 6.4 zeigt ein Klassendiagramm mit zwei Knotenklassen, einem Attribut und einer Vererbungsbeziehung. Im Zielschema soll eine Klasse “Angestellter” erzeugt werden, welche ein Attribut “name” enthält. Des weiteren soll eine Klasse “Abteilungsleiter” erzeugt werden, welche von “Angestellter” abgeleitet ist und damit das Attribut “name” erbt. Das Quellschema soll analog zum Zielschema aufgebaut sein, hier haben die Knotenklassen noch ein vorangestelltes “Quell” im Namen. Der für das Beispiel zu erzeugende Zielgraph ist in Abbildung 6.5 dargestellt und enthält vier Knoten, jeweils zwei Instanzen der Knotenklasse “Angestellter” sowie zwei Instanzen der Knotenklasse “Abteilungsleiter”. Der Quellgraph enthält analog die vier Instanzen der jeweiligen “Quell”-Klassen. In diesem Beispiel wurden keine Kanten verwendet.

Bei beiden Situationen, die auftreten können, werden zuerst die beiden Klassen “Angestellter” und “Abteilungsleiter” angelegt. Im Anschluss daran kann entweder zuerst das Attribut angelegt werden oder zuerst die Generalisierung, was zu folgenden beiden

Abbildung 6.5: Beispiel einer Vererbungsbeziehung auf Graphebene

Listings führt, die Ausschnitte der `transformation()`-Methode darstellen:

Listing 6.6: Anlegen von Attributen durch Abfrage des Quellgraphen

```
1  VertexClass angestellter = createVertexClass("Angestellter", false,
2     "V{QuellAngestellter!}");
3  VertexClass abteilungsleiter = createVertexClass("Angestellter", false,
4     "V{QuellAbteilungsleiter}");
5  addSuperClass(abteilungsleiter, angestellter);
6  createAttribute("name", angestellter, "String",
7     "from v:V{QuellAngestellter} reportMap v, v.name end");
```

Listing 6.7: Anlegen von Attributen durch Abfrage des Quellgraphen

```
1  VertexClass angestellter = createVertexClass("Angestellter", false,
2     "V{QuellAngestellter!}");
3  VertexClass abteilungsleiter = createVertexClass("Angestellter", false,
4     "V{QuellAbteilungsleiter}");
5  createAttribute("name", angestellter,
6     "String", "from v:V{QuellAngestellter} reportMap v, v.name");
7  addSuperClass(abteilungsleiter, angestellter);
```

Während der Schematransformation spielt es keine Rolle, welche Reihenfolge verwendet wird, das erzeugte Schema bleibt in beiden Fällen gleich. Anders ist dies bei der Graphtransformation, bei der die semantischen Ausdrücke ausgewertet werden. In Listing 6.6, bei dem zuerst die Generalisierung und danach das Attribut angelegt wird, ist die Angabe des semantischen Ausdrucks beim Erzeugen des Attributes “name” problemlos möglich. Der in Zeile 7 angegebene Ausdruck liefert alle Instanzen von “QuellAngestellter” und dessen Unterklassen. Da die img-Abbildung der Knotenklasse “Angestellter” durch die vorangegangene Generalisierung auch schon alle Abbildungen von “Abteilungsleiter” im Zielgraphen besitzt, lassen sich allen Instanzen die richtigen Werte zuordnen.

Anders ist dies im Listing 6.7. Dort wird die Generalisierung erst nach Anlegen des Attributes erzeugt, so dass die img-Abbildung von “Angestellter” die “Abteilungsleiter” zu diesem Zeitpunkt noch nicht enthält. Obwohl die Menge VQuellAngestellter die “Abteilungsleiter” zwar beinhaltet, werden diesen jedoch die Werte nicht zugeordnet, da diesen Knoten im Zielgraphen keine Instanz zugeordnet werden kann. Durch das anschließende Hinzufügen der Generalisierung wird die Abbildung von “Angestellter” um

die Abbildung von “Abteilungsleiter” erweitert, jedoch ist die Zuordnung des Attributes “name” an dieser Stelle nicht mehr möglich. Das heißt, beim Anlegen der Generalisierung müssen die Attributwerte der Instanzen, die aufgrund des Fehlens der Abbildung nicht zugeordnet werden konnten, ergänzt werden.

Zur Zeit existieren *zwei Möglichkeiten*, mit der Ergänzung der Attributwerte umzugehen. Die erste Möglichkeit ist die Verwendung eines *vorausschauenden Anlegens der Abbildungen*, die zweite Möglichkeit *die Angabe eines semantischen Ausdruckes bei* `addSuperClass()`, welche nachträglich den Instanzen die fehlenden Attributwerte zuordnet. Beide Möglichkeiten sollen im folgenden beschrieben werden.

Der Modus zum *Vorausschauenden Anlegen von Abbildungen* macht sich die Tatsache zunutze, dass die Transformation in zwei Schritten abläuft. Während der Graphtransformation ist das Zielschema schon komplett aufgebaut und wird nicht mehr verändert. Für die Transformation heißt das, wenn ein Knoten oder eine Kante im Zielgraphen instantiiert wird, so existieren schon alle Attribute, da im Schema die Generalisierung schon erfolgt ist. Der Benutzer gibt beim Anlegen eines Attributes einen semantischen Ausdruck an, mit dem die Abbildung der Urbilder der Zielgraphelemente auf die zuzuweisenden Attributwerte bestimmt wird. Im Beispiel oben wurde gezeigt, dass die img-Abbildung, die die Zuordnung der Urbildelemente auf die Elemente im Zielgraph vornimmt, an dieser Stelle noch nicht für alle Instanzen existiert, falls eine Generalisierung erst später definiert wird. Diese Zuordnung kann jedoch auch früher erzeugt werden und nicht erst beim aufruf von `addSuperClass()`.

Beim Anlegen einer Knoten- oder Kantenklasse werden bei der Graphtransformation die img- und arch-Abbildungen für diese erzeugt, während die Instanzen angelegt werden. Beim späteren Aufruf von `addSuperClass()` werden diese Abbildungen zu den Abbildungen der übergeordneten Klassen hinzugefügt. Diese übergeordneten Klassen werden am Zielschema abgefragt. Da das Zielschema aber schon komplett aufgebaut ist und sich nicht mehr ändert, ist beim Anlegen der Abbildungen für eine Klasse schon bekannt, welche Abbildungen später erweitert werden müssen. Aus diesem Grund kann die Erweiterung der Abbildungen bereits erfolgen, wenn die Instanzen der Knoten- und Kantenklassen erzeugt werden. Für das Beispiel heißt dies, beide Listings ergeben auch den gleichen Zielgraph nach Abschluss der Graphtransformation.

Die Verwendung des vorausschauenden Anlegens der Abbildungen hat aber einen *Nachteil*, der deren Verwendung zumindestens für zukünftige Entwicklungen ausschließen könnte. Es ist geplant, JGraLab später so zu verändern, dass es simultan auf dem Schema und dem dazu gehörenden Graphen arbeiten kann. Diese Änderung hätte zur Folge, dass die Ausführung der Transformation in einem statt in zwei Schritten erfolgen kann. Das Vorausschauen macht es sich zum aktuellen Zeitpunkt zunutze, dass man zu Beginn der Graphtransformation schon weiß, welche Generalisierungen noch aufgebaut

werden, da diese am schon fertig aufgebauten Zielschema abgefragt werden können. Arbeitet man jetzt mit simultaner Erzeugung des Zielschemas und Zielgraphen, fällt dieser Vorteil jedoch weg und man muss sich einer anderen Methode bedienen. Diese basiert darauf, bei Anlegen einer Generalisierung die Attributwerte der Instanzen, welche noch nicht von der Abbildung erfasst waren, zu ergänzen, indem bei `addSuperClass()` ein zusätzlicher semantischer Ausdruck angegeben wird.

Der semantische Ausdruck, der zur *Ergänzung der fehlenden Attributwerte* angegeben werden kann, muss die Ergänzung bei mehreren Attributen auf einmal unterstützen. In dem oben gezeigten Beispiel existiert nur ein Attribut ("name"), welches bei der Instanz "X" ergänzt werden muss. In der Praxis werden dies aber mehrere Attribute sein. Aus diesem Grund muss der Ausdruck eine *zweistufige Abbildung* zurückliefern. In der ersten Stufe wird eine Abbildung vom Attributnamen auf die zweite Abbildung benötigt. In der zweiten Stufe eine Abbildung der Urbilder der zu ergänzenden Instanzen auf die jeweiligen Attributwerte.

Für das gezeigte Beispiel in Listing 6.7 musste also der semantische Ausdruck ergänzt werden, damit den Objekten "C" und "D" die Attributwerte von "name" zugeordnet werden können. Der benötigte Ausdruck ist in Listing 6.8 dargestellt. Dieser müsste die Zeilen 6-7 von Listing 6.7 ersetzen:

Listing 6.8: Reparatur der Attributwerte (Beispiel)

```
1  addSuperClass(abteilungsleiter, angestellter,
2     "reportMap name, from v:V{QuellAbteilungsleiter!} reportMap v, n.name end");
```

Alternativ kann an Stelle des semantischen Ausdrucks auch eine JValueMap im Format JValueMap(String, JValueMap(JValue, JValue)) übergeben werden.

Die beiden Modi können umgeschaltet werden, indem in der Klasse `TransformationContext` die Variable `miracleMode` umgestellt wird. Die Voreinstellung ist zur Zeit **`true`**, welches den ersten, also den vorausschauenden Modus aktiviert. Alternativ kann dem Konstruktor der `TransformationContext`-Klasse als dritter, optionaler Parameter der Modus mit übergeben werden. Dabei steht **`true`** für den vorausschauenden Modus, **`false`** für den ergänzenden Modus. Der Parameter lässt sich nur im Konstruktor setzen, da ein umschalten während einer Transformation nicht absehbare Ergebnisse liefern würde. Innerhalb der `transformation()`-Methode kann der Modus allersind mit Hilfe von `isMiracleMode()` abgefragt werden, falls eine Transformation sich je nach Modus anders verhalten soll.

7 Implementierung einer Transformation durch den Benutzer

In diesem Abschnitt wird die Implementierung einer Transformation durch den Benutzer beschrieben. Dazu wird zuerst der generelle Aufbau einer Transformationsklasse, die von der Klasse `Transformation` abgeleitet ist beschrieben. Im Anschluss daran wird die Verwendung des Kontext-Objektes genauer beschrieben. Zum Abschluss des Kapitels wird zusammenfassend noch ein kurzes, aber vollständiges Beispiel angegeben.

7.1 Genereller Aufbau einer Transformation

Um eine Transformation zu definieren erzeugt der Benutzer eine Klasse, die von der `Transformation`-Klasse abgeleitet ist. In dieser muss die Methode `transform()` implementiert werden, die alle auszuführenden Operationsaufrufe enthält. Zusätzlich ist die Implementierung eines oder mehrerer (falls es optionale Parameter gibt, benötigt man überladene Konstruktoren) Konstruktoren notwendig, in der der Kontext der Transformation sowie eventuelle Parameter an die Transformation übergeben werden. Listing 7.1 zeigt den Grundaufbau einer Transformationsklasse, wie sie vom Benutzer implementiert werden kann. Der in den Zeilen 2-4 gezeigte Konstruktor mit dem Aufruf des Konstruktors der Oberklasse (`super(context)`) muss immer vorhanden sein, damit das Kontext-Objekt übergeben wird.

Listing 7.1: Beispiel: Klassenrumpf einer Transformation

```
1 class ConcreteTransformation extends Transformation {
2    public ConcreteTransformation (TransformationContext context) {
3       super(context);
4    }
5    public void transform() {
6       // hier werden einzelne Transformationsoperationen aufgerufen...
7    }
8 }
```

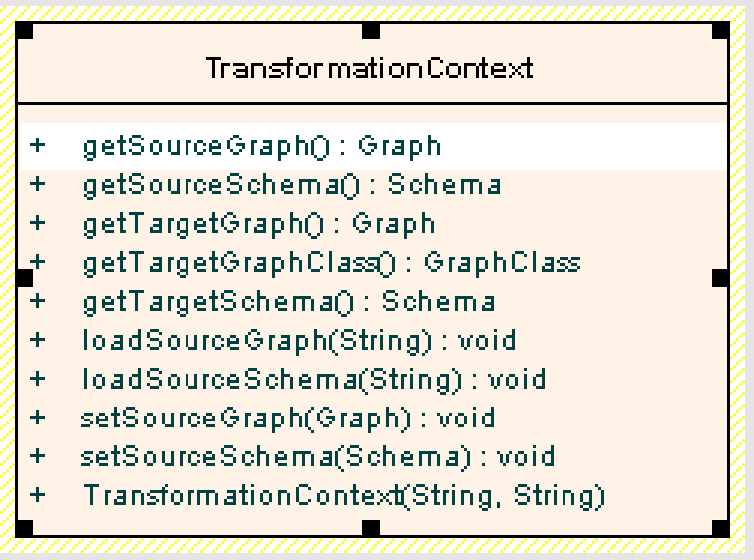

Abbildung 7.1: Aufbau TransformationContext-Klasse (Benutzersicht)

Innerhalb der Methode `transform()` wird die eigentliche Transformation definiert. Dabei können sämtliche elementaren Transformationsoperationen und andere Transformationsobjekte verwendet werden, die im Detail in Anhang B beschrieben werden.

7.2 Verwendung des Kontext-Objekts

Das Kontext-Objekt wird außerhalb der Transformation definiert und dieser beim Instantiieren übergeben. Es enthält alle Informationen, die während einer Transformation benötigt werden. Damit eine Transformation überhaupt ausgeführt werden kann, muss es zuvor mit dem Namen des Zielschemas sowie dem Namen der dazu zu erzeugenden Graphenklasse instantiiert und übergeben werden.

Innerhalb einer Transformation ist das Kontext-Objekt immer über die Variable `context` verfügbar. Es bietet eine Reihe von öffentlichen Methoden, die zur Abfrage verschiedener Informationen über die verwendeten Schemata und Graphen dienen sowie zur Abfrage, ob gerade die Schema- oder die Graphtransformation durchgeführt wird. Auf diese Informationen kann immer durch Aufruf von `context.methodenName()` zugegriffen werden. Die Methoden sind im Diagramm 7.1 aufgeführt und werden im folgenden genauer beschrieben.

Das Kontext-Objekt bietet verschiedene Möglichkeiten, das Quellschema und den Quell-

graph festzulegen. Zum einen können sie mit `loadSourceSchema(String filename)` bzw. `loadSourceGraph(String filename)` aus Quelldateien gelesen werden. Wird die Methode `loadSourceGraph(String filename)` verwendet, wird nicht nur der Graph aus der Datei geladen, sondern automatisch auch das dazu gehörende Schema (dieses ist in einer Datei mit einem Graphen immer mit gespeichert, da es für die Instantiierung des Graphen benötigt wird). Alternativ können sowohl das Quellschema als auch der Quellgraph mit Hilfe von Settern (`setSourceSchema(Schema schema)` bzw. `setSourceGraph(Graph graph)`) gesetzt werden, falls diese schon innerhalb des Programms verwendet wurden.

Zum Abfragen von Graphen oder Schemata sind die Methoden `getSourceGraph()`, welche den Quellgraphen liefert, `getSourceSchema()`, welche das Quellschema liefert, `getTargetGraph()`, welche den Zielgraphen liefert und `getTargetSchema()`, welche das Zielschema liefert, implementiert.

Zusätzlich verfügt die Kontext-Klasse über die Methode `isExecutingGraphTransformation()`, welche den boolschen Wert “true” zurückgibt, falls gerade die Graphtransformation durchgeführt wird bzw. “false”, falls gerade die Schematransformation durchgeführt wird. Die Verwendung dieser Methode ist innerhalb der `transformation()`-Methode sinnvoll, falls bestimmte Operationen nur in einem der beiden Transformationsschritte ausgeführt werden soll.

7.3 Instantiierung des Kontext-Objektes und Aufruf der Transformation

Die Transformation wird von einem steuernden Programm aus aufgerufen. Dieses instantiiert das Kontext-Objekt und übergibt ihm alle zur Ausführung der Transformation benötigten Informationen. Im Anschluss daran wird die Transformation instantiiert und durch Aufrufen der `execute()`-Methode ausgeführt. Nach der Transformation können das Zielschema und der Zielgraph vom Kontext-Objekt abgefragt werden. Dieser Zusammenhang ist in Listing 7.2 noch einmal kurz dargestellt und entspricht dem schon in Kapitel 6.1 vorgestellten Beispiel aus Listing 6.1.

Listing 7.2: Aufruf einer Transformation

```
1 class TransformationExecutor {
2    public static void main(String[] args) {
3       TransformationContext context = new TransformationContext("targetSchemaName",
4         "targetGraphClassName");
5       context.loadSourceGraph("filename");
6       Transformation t = new ConcreteTransformation(context);
7       t.execute();
8       Schema targetSchema = context.getTargetSchema();
9       Graph targetGraph = context.getTargetGraph();
```

```
10      }
11  }
```

7.4 Verwendung schon existierender Transformationen

Eine bestehende Transformation kann innerhalb einer anderen Transformation verwendet werden. Dazu muss diese instantiiert werden und danach deren `execute()`-Methode aufgerufen werden. Dies soll an folgendem Codefragment in Listing 7.3 verdeutlicht werden. In diesem Beispiel wird innerhalb einer zu erstellenden Transformation die schon existierende Transformation `CopyVertexClass` verwendet werden, welche auf Schemaebene eine Knotenklasse vom Quell- ins Zielschema kopiert und auf Graphebene jeweils die dazu gehörenden Instanzen vom Quell- in den Zielgraph. Der Name der zu kopierenden Knotenklasse wird dem Konstruktor von `CopyVertexClass` als zweiter Parameter übergeben.

Listing 7.3: Beispiel: Aufruf einer Transformation innerhalb einer anderen

```
1  public void transform() {
2     // Transformationsoperationen vor Aufruf von CopyVertexClass
3     Transformation copyVertexClass = new CopyVertexClass(context, "Diplomarbeit");
4     copyVertexClass.execute();
5     // Transformationsoperationen nach Aufruf von CopyVertexClass
6  }
```

7.5 Beispiel: Definition und Aufruf einer Transformation (Kopieren einer Knotenklasse)

Listing 7.4 zeigt beispielhaft eine vollständige Transformation. Die Transformation kopiert eine Knotenklasse aus dem Quellschema in das Zielschema. Zusätzlich ist ein Aufruf an die `copyAttributes`-Transformation im Beispiel vorhanden, welche später erläutert wird. Komplette Beispieltransformationen, die einzelne Knotenklassen, Kantenklassen oder ein komplettes Schema kopieren können, sind im Anhang D aufgeführt.

Listing 7.4: Beispieltransformation: Kopieren einer Knoten-Klasse

```
1  class CopyVertexClass extends Transformation {
2     protected String vertexClassName;
3     public void CopyVertexClass(TransformationContext context,
4        String vertexClassName) {
5        super(context)
```

```
6         this.vertexClassName = vertexClassName;
7      }
8      public void transform() {
9         Boolean isAbstract = context.getSourceSchema().getAttributedElementClass
10          (new QualifiedName(vertexClassName)).isAbstract();
11        String query = "V{"+vertexClassName+"}";
12        createVertexClass(vertexClassName, isAbstract, query);
13        Transformation copyAttr = new copyAttributes(context, vertexClassName);
14        copyAttr.execute();
15     }
16  }
```

Dabei wird in Zeile 2 der Parameter der Transformation (der Name der zu kopierenden Knotenklasse) definiert. Der Konstruktor dieser Klasse (Zeilen 3 bis 7) bekommt dazu sowohl den Kontext als auch den Namen der zu kopierenden Knotenklasse übergeben. Der Konstuktor ruft in Zeile 5 den Konstruktor der Superklasse (`Transformation`) auf, um diesem den Kontext zu übergeben. Die Methode `transform()` in den Zeilen 8 bis 15 definiert, was die Transformation machen soll. Dazu fragt sie in Zeile 9 aus dem Quellschema die Information ab, ob die zu kopierende Knotenklasse als abstrakt definiert wurde oder nicht. In Zeile 11 wird die Abfrage definiert, mit der im zweiten Transformationsschritt, der Graphtransformation, die Urbildmenge der zu erzeugenden Knoten aus dem Quellgraphen abgefragt wird. Abschließend wird in Zeile 12 bei der Schematransformation die neue Knotenklasse im Zielschema erzeugt bzw. bei der Graphtransformation die entsprechenden Knoten im Zielgraph.

Die Zeilen 13 und 14 zeigen den Aufruf einer Transformation innerhalb der aktuell laufenden. Zu diesem Zweck wird die aufzurufende Transformation in Zeile 11 instantiiert, der Parameter (in diesem Fall der Name der Knotenklasse, deren Attribute kopiert werden sollen) wird im Konstruktor übergeben. In Zeile 14 wird die "eingebettete" Transformation ausgeführt. Den Kontext, in dem sie laufen soll, wird automatisch in der `execute()`-Methode aus der `TransformationContext`-Klasse abgefragt.

Der Aufruf der Transformation erfolgt genau wie schon in Listing 7.2 beschrieben und wird aus diesem Grund nicht noch einmal beschrieben.

Listing 7.5 zeigt das Kopieren aller Attribute einer Klasse aus dem Quellschema in das Zielschema. Es wird angenommen, dass die Klasse in beiden Schemata den gleichen Namen besitzen. Als Parameter im Konstruktor erwartet sie neben dem Kontext-Objekt den Namen der GraphElement-Klasse aus dem Quellschema, deren Attribute kopiert werden sollen. Innerhalb von `transform()` bestimmt sie dazu zuerst die Elementenklasse im Quellschema (Zeilen 8 und 9). In Zeile 11 wird eine Schleife gestartet, die über alle eigenen (und keine geerbten) Attribute iteriert und diese einzelnd mit Hilfe der in Listing 7.6 aufgeführten Transformation `CopyAttribute` einzelnd kopiert.

Listing 7.5: Beispiel: Kopieren der Attribute einer Klasse

```
1  public class CopyAttributes extends Transformation {
2     protected String sourceElementName;
3     public CopyAttributes(TransformationContext context, String sourceElementName) {
4        super(context);
5        this.sourceElementName = sourceElementName;
6     }
7     public void transform() throws Exception {
8        AttributedElementClass ae = context.getSourceSchema().
9           getAttributedElementClass(new QualifiedName(sourceElementName));
10       // Nur eigene Attribute, keine geerbten kopieren!
11       for (Attribute attr : ae.getOwnAttributeList()) {
12          CopyAttribute copyAttribute = new CopyAttribute(context, ae,
13             attr.getName());
14          copyAttribute.execute();
15       }
16    }
17 }
```

Die Transformation `CopyAttribute`, welche in Listing 7.6 dargestellt ist, kopiert ein einziges Attribut einer Elementenklasse von Quellschema in das Zielschema. Es wird angenommen, dass die Klassen in beiden Schemata den gleichen Namen besitzen. Der Konstruktor in den Zeilen 4 bis 9 erwartet deshalb die Elementenklasse, die das zu kopierende Attribut enthält sowie den Namen des zu kopierenden Attributes. Die Transformation, wie sie hier dargestellt ist, funktioniert nur korrekt, wenn sie im vorausschauenden Modus (siehe Kapitel 6.4) ausgeführt wird. Aus diesem Grund wird zuerst in den Zeilen 11 und 12 geprüft, ob dieser Modus aktiviert ist. Eine Implementierung ohne den vorausschauenden Modus ist ebenfalls möglich, in diesem Fall müsste bei der Implementierung von Generalisierungen innerhalb einer anderen Transformation auf die Attribute rücksicht genommen werden. In den Zeilen 13 bis 15 wird zuerst das Attribut aus dem Quellschema abgefragt und die Elementenklasse, die das Attribut im Zielschema erhalten soll, bestimmt. In Zeile 17 wird eine Fallunterscheidung vorgenommen, da der semantische Ausdruck, der für die Graphtransformation benötigt wird, sich für Knoten- und Kantenklassen unterscheidet. Die Zeilen 18 bis 23 bzw. 25 bis 30 legen jeweils das Attribut in der Knoten- bzw. Kantenklasse im Zielschema an und generieren für die Graphtransformation einen semantischen Ausdruck, der zu der entsprechenden Elementenklasse passt.

Listing 7.6: Beispiel: Kopieren eines Attributes

```
1  public class CopyAttribute extends Transformation {
2     protected String attributeName;
3     protected AttributedElementClass sourceElement;
4     public CopyAttribute(TransformationContext context,
5        AttributedElementClass sourceElement, String attributeName) {
6        super(context);
7        this.sourceElement = sourceElement;
8        this.attributeName = attributeName;
9     }
10    public void transform() throws Exception {
11       if (!context.isMiracleMode()) throw new Exception("Fuer diese
```

```
12            Transformation wird der vorausschauende Modus benoetigt!");
13         Attribute sourceAttribute = sourceElement.getAttribute(attributeName);
14         AttributedElementClass target = context.getTargetSchema().
15            getAttributedElementClass(new QualifiedName(sourceElement.
16            getQualifiedName()));
17         if (target instanceof VertexClass) {
18            createAttribute(
19               attributeName,
20               target,
21               sourceAttribute.getDomain().getSimpleName(),
22               "from v:V{" + sourceElement.getQualifiedName() + "!}
23                  reportMap v, v." + attributeName + " end");
24         } else {
25            createAttribute(
26               attributeName,
27               target,
28               sourceAttribute.getDomain().getQualifiedName(),
29               "from e:E{" + sourceElement.getQualifiedName() + "!}
30                  reportMap e, e." + attributeName + " end");
31         }
32      }
33   }
```

8 Anwendungsbeispiel

In diesem Kapitel wird ein Anwendungsbeispiel dargestellt, welches die am Rahmen eines Papers zum Thema “Metamodel-driven Service Interoperability” [Ebe05] beschrieben Transformationen implementiert wurden. Das Paper befasst sich mit der Verwendung von Metamodellen zur Synchronisation und Austausch von Daten zwischen verschiedenen Werkzeugen beim Einsatz von (Web-)Services.

Ein *Service Provider* implementiert einen *Service* oder auch Dienst, welcher von einem *Service Requestor* verwendet wird. Ein Service besitzt ein Referenzschema, welches die Datenstruktur beim Austausch der Dienste zwischen Provider und Requestor festlegt. Um den Dienst nutzen zu können, muss der Requestor seine Eingabedaten für den Aufruf an dessen vorgegebene Datenstruktur anpassen. Das gleiche gilt auch für den Provider, welcher sich bei der Aufbereitung der Rückgabedaten an diese Struktur halten muss. Dies sorgt dafür, dass die Implementierung unterschiedlicher Service Provider möglich ist, die intern mit unterschiedlichen Datenstrukturen arbeiten, nach außen jedoch die gleiche Funktionalität anbieten. Der gleiche Zusammenhang gilt ebenfalls für die Service Requestor, deren innere Datenstruktur unabhängig vom jeweiligen Dienst wird.

Für einen Datenaustausch zwischen einem Provider und einem Requestor ist es deshalb notwendig, die jeweiligen intern verwendeten Schemata an das Referenzschema anzupassen, um die Daten synchronisieren zu können. In [Ebe05] wird vorgeschlagen, diese Synchronisation mit Hilfe der Metamodellierung und der Metamodell-basierten Modelltransformation zwischen den jeweiligen Requestor- bzw. Provider-Schema und dem Referenzschema zu realisieren. Um diese Transformationen durchführen zu können, sind zwei Schritte notwendig. Zuerst müssen die beteiligten Schemata definiert werden, wie sie im nächsten Abschnitt beschrieben werden. Im Anschluss daran muss die Datentransformation spezifiziert werden, indem die Metamodell-basierte Modelltransformation angegeben wird. Der Austausch der Daten an sich spielt innerhalb dieser Arbeit keine Rolle.

Ziel ist es, mit Hilfe der in dieser Arbeit erstellten Transformationsklasse zwei Transformationen zu implementieren, mit deren Hilfe zuerst das Datenschema des Service Requestors in das Referenzschema überführt wird und anschließend das Referenzsche-

ma in das Schema des Service Providers.

8.1 Beschreibung der beteiligten Schemata

Die verwendeten Schemata wurden aus [Ebe05] übernommen. Einzig die verwendeten Namen der Kantenklassen wurde auf Camelcase-Schreibweise umgestellt. In dem aufgeführten Beispiel wird der Austausch von Daten zur Visualisierung einer Dekompositionsschicht zwischen zwei Software-Reengineering-Werkzeugen dargestellt. Das *Bauhaus ReArchitecting Tool* ist ein Werkzeug, welches Software-Architekturen mit Hilfe von Flussgraphen repräsentiert. Es tritt als Service Requestor auf, dessen Daten von einem anderen Modellierungswerkzeug visualisiert werden sollen. Das Visualisierungstool soll ein Werkzeug sein, welches auf einer Variante des UML2 Metamodells arbeitet und als Service Provider eingebunden werden. Im Verwendeten Beispiel ist dies der *IBM Software Modeler*.

Abbildung 8.2 stellt das *Referenzschema* dar. Es stellt eine einfache Möglichkeit der Dekomposition von Software-Architekturen dar. Ein Service Requestor kann bei Verwendung des Schemas davon ausgehen, dass der Service Provider alle diese Schemaelemente auch visualisien kann. Eine Klasse (*Class*) kann Methoden (*Method*) und Attribute (*Attribute*) besitzen und selbst in einem Paket (*Package*) enthalten sein. Ein Paket kann neben beliebig vielen Klassen auch weitere Pakete enthalten.

Das *BauhausSchema* (Requestor-Schema) ist in Abbildung 8.1 dargestellt. Pakete (*Package*) deklarieren weitere Pakete oder abstrakte Typen *Type*. Diese Typen sind entweder Klassen (*Class*) oder Interfaces (*Interface*). Die Typen selbst setzen sich aus Methoden (*Method*) und Attributen (*Member*) zusammen. Darüber hinaus sind im Bauhausschema noch diverse weitere Beziehungen definiert, die sich allerdings nicht mit den Beziehungen im Referenzschema decken und somit unbeachtet bleiben.

Abbildung 8.3 beschreibt das *UMLSchema*. Dieses Schema basiert auf einem Ausschnitt des UML2 Metamodells, in welchem nur die in diesem Zusammenhang benötigten Schemaelemente übernommen wurden. Pakete (*Package*) und Klassen (*Class*) sind in einer abstrakten Superklasse *PackageableElement* zusammengefasst. Pakete können weitere PackageableElements enthalten. Eine Klasse setzt sich aus Methoden (*Operation*) und Attributen (*Property*) zusammen.

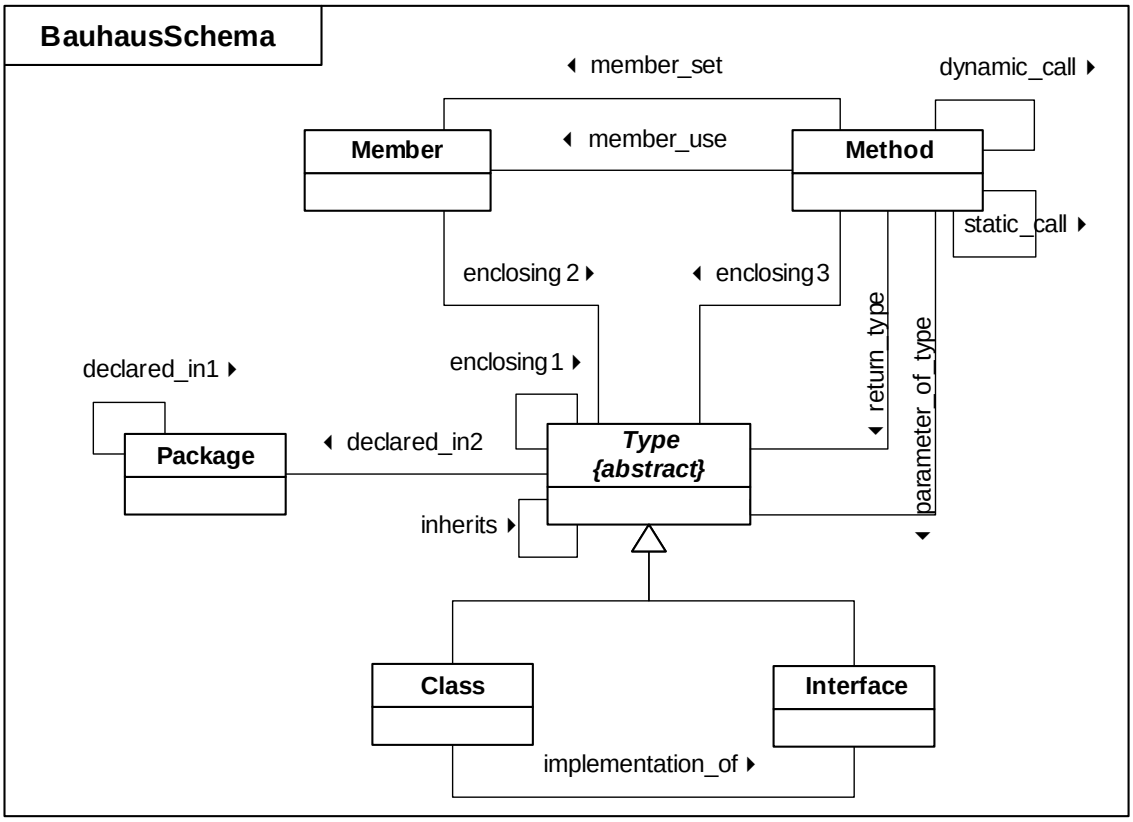

Abbildung 8.1: Bauhaus Graphschema

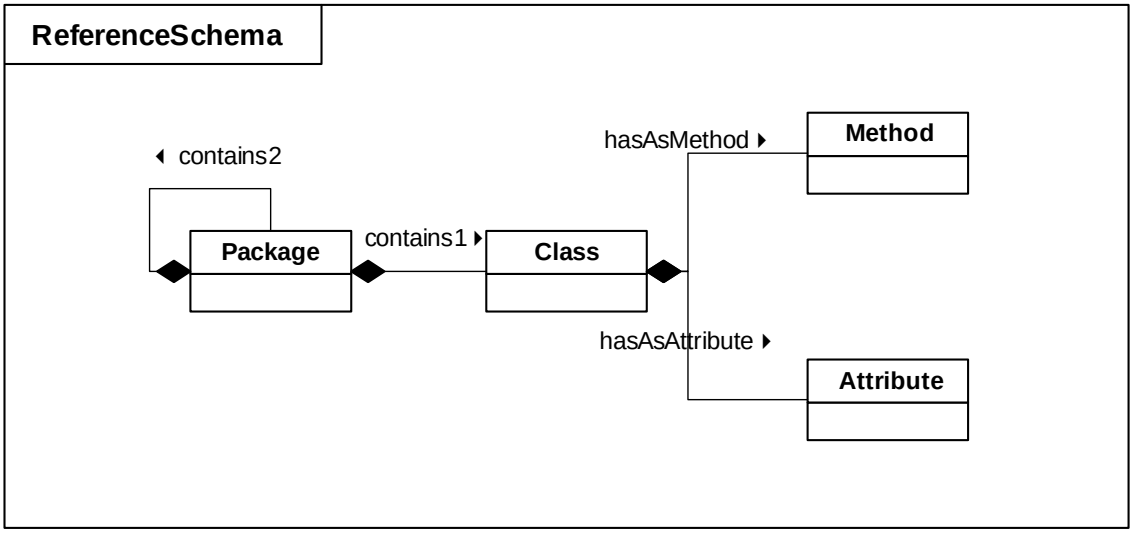

Abbildung 8.2: Referenz Graphschema

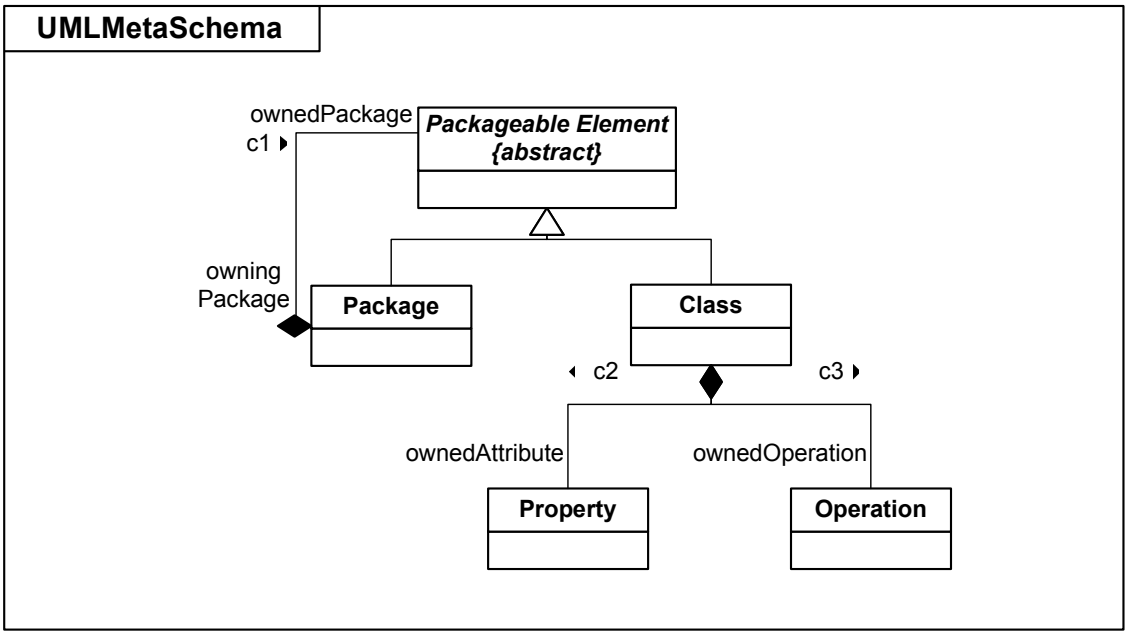

Abbildung 8.3: UML Graphschema

8.2 Transformation: Bauhaus- zu Referenzschema

Das folgende Listing zeigt die Transformation vom Bauhaus- zum Referenzschema. Im Anschluss an den Quelltext wird dieser erläutert, woraufhin ein Beispiel einer Transformation mit einem existierenden Quellgraphen dargestellt wird.

Listing 8.1: Beispiel: Transformation Bauhaus- zu Referenzschema

```
1  package de.uni_koblenz.jgralab.gretl;
2  import de.uni_koblenz.jgralab.schema.VertexClass;
3  public class BauhausToReference extends Transformation {
4     public BauhausToReference(TransformationContext context) {
5        super(context);
6     }
7     @Override
8     public void transform() throws Exception {
9        // Package kopieren
10       VertexClass pkg = createVertexClass("Package", false, "V{Package}");
11       // Class kopieren
12       VertexClass cls = createVertexClass("Class", false, "V{Class}");
13       // Method kopieren
14       VertexClass mth = createVertexClass("Method", false, "V{Method}");
15       // Umbenennen: Member->Attribute
16       VertexClass attr = createVertexClass("Attribute", false, "V{Member}");
17       // Neue Kompositionsklasse contains1 erzeugen...
18       this.createCompositionClass("Contains1", pkg, true, cls, false,
19          "E{DeclaredIn2}",
20          "from e:E{DeclaredIn2} reportMap e, endVertex(e) end",
```

```
21          "from e:E{DeclaredIn2} reportMap e, startVertex(e) end");
22       // Neue Kompositionsklasse contains2 erzeugen...
23       this.createCompositionClass("Contains2", pkg, true, pkg, false,
24          "E{DeclaredIn1}",
25          "from e:E{DeclaredIn1} reportMap e, startVertex(e) end",
26          "from e:E{DeclaredIn1} reportMap e, endVertex(e) end");
27       // Neue Kompositionsklasse hasAsMethod
28       this.createCompositionClass("HasAsMethod", cls, true, mth, false,
29          "from e1:E{Inherits}, e2:E{Enclosing2}, v1:V{Class}, v2:V{Method} with
30             v1 (--e1->)* <-e2-- v2 report tup(v1, e2) end",
31          "from e1:E{Inherits}, e2:E{Enclosing2}, v1:V{Class}, v2:V{Method} with
32             v1 (--e1->)* <-e2-- v2 reportMap tup(v1, v2), v1 end",
33          "from e1:E{Inherits}, e2:E{Enclosing2}, v1:V{Class}, v2:V{Method} with
34             v1 (--e1->)* <-e2-- v2 reportMap tup(v1, v2), v2 end");
35       // Neue Kompositionsklasse hasAsAttribute
36       this.createCompositionClass("HasAsAttribute", cls, true, attr, false,
37          "from e1:E{Inherits}, e2:E{Enclosing1}, v1:V{Class}, v2:V{Member} with
38             v1 (--e1->)* <-e2-- v2 report tup(v1, v2) end",
39          "from e1:E{Inherits}, e2:E{Enclosing1}, v1:V{Class}, v2:V{Member} with
40             v1 (--e1->)* <-e2-- v2 reportMap tup(v1, v2), v1 end",
41          "from e1:E{Inherits}, e2:E{Enclosing1}, v1:V{Class}, v2:V{Member} with
42             v1 (--e1->)* <-e2-- v2 reportMap tup(v1, v2), v2 end");
43    }
44 }
```

In den Zeilen 9 bis 14 werden zuerst die Knotenklassen “Package”, “Class” und “Method” aus dem Bauhausschema in das Zielschema kopiert. Die semantischen Ausdrücke kopieren auf Graphebene die entsprechenden Knoten aus dem Quellgraphen in den Zielgraphen.

Im Anschluss daran wird in der Zeile 16 die Knotenklasse “Member” des Quellschemas in das Zielschema kopiert und dabei zu “Attribute“ umbenannt. Der semantische Ausdruck kopiert wiederum auf Graphebene die entsprechenden Knoten.

Anschließend werden die Kompositionsklassen im Zielschema erzeugt. Dabei wird zuerst in den Zeilen 18 bis 21 die Kompositionsklasse ”Contains1“ erzeugt, welche der Kantenklasse ”DeclaredIn2“ des Bauhausschemas entspricht und eine Komposition von ”Package“ zu ”Class“ erstellt. ”Contains1“ ordnet den Paketen ihre zugehörigen Klassen zu. Die entsprechenden Kanten werden durch die Angabe der semantischen Ausdrücke auf Graphebene kopiert, wobei die Start- und Zielknoten der Kante jeweils vertauscht werden.

In den Zeilen 23 bis 26 wird analog zu ”Contains1“ die Kompositionsklasse ”Contains2“ erzeugt. Sie erzeugt eine Komposition von ”Package“ zu ”Package“ und entspricht der Kantenklasse ”DeclaredIn1“ des Quellschemas. Die zu erzeugenden Kanten im Zielgraphen werden aus den Instanzen der Kantenklasse ”DeclaredIn1“ aus dem Quellgraphen kopiert.

Die Kompositionsklasse ”HasAsMethod“ wird in den Zeilen 28 bis 34 erstellt. Diese erzeugt die Komposition von ”Class“ zu ”Method“ des Referenzschemas. Die seman-

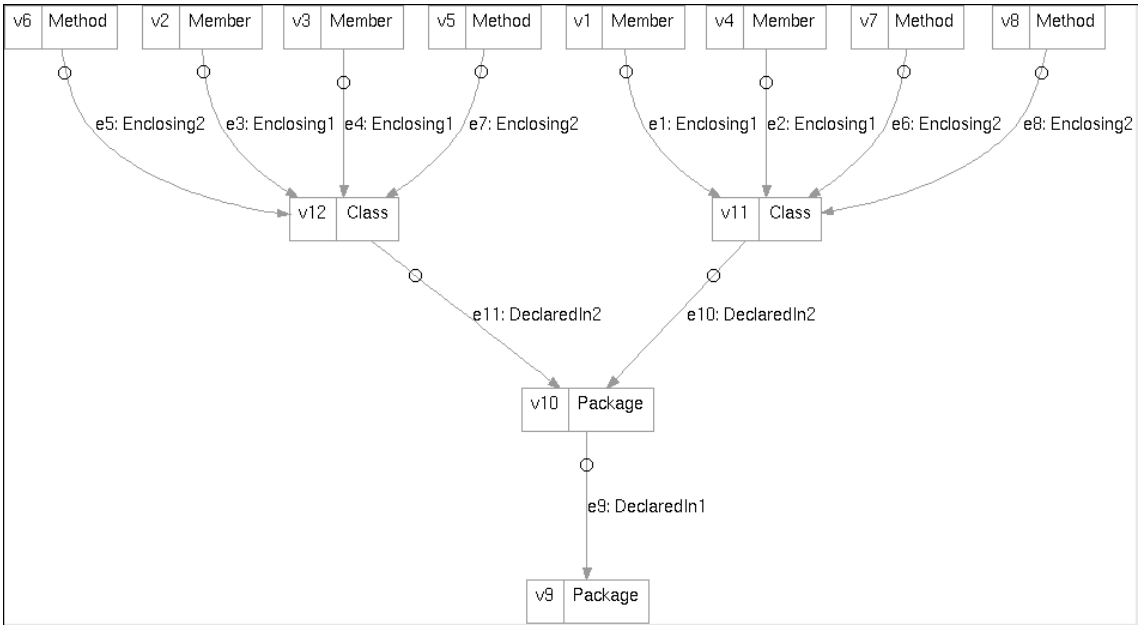

Abbildung 8.4: Bauhaus-Graph

tischen Ausdrücke bestimmen im Quellgraphen jeweils diejenigen Pfade, welche von ”Class“ zu ”Method“ führen und vom Typ ”Enclosing1“ sind und eventuell Kanten vom Typ ”Inherits“ enthalten. Wichtig bei den Semantischen Ausdrücken ist hierbei, das Urbild als Tupel von . Analog dazu wird in den Zeilen 36 bis 42 die Kantenklasse ”HasAsAttribute“ erzeugt, welche Pfade von der Kantenklasse ”Class“ zur Kantenklasse ”Member“ zur Bestimmung der zu erzeugenden Kanten im Zielgraphen bestimmt.

Abbildung 8.4 stellt einen Beispielgraphen im Bauhausschema dar, welcher manuell erstellt wurde und als Quellgraph dient. Abbildung 8.5 zeigt den erzeugten Graphen im Referenzschema, der mit Hilfe der Transformation aus Listing 8.1 erzeugt wurde. Zur Visualisierung wurden die entsprechenden Graph-Dateien mit Hilfe von tg2dot umgewandelt und mit dotty dargestellt. Der Quellgraph enthält zwei Packages (v9 und v10), die in den Zielgraphen kopiert werden (v1 und v2). Die ”DeclaredIn1“-Kante e10 des Quellgraphen wurde als ”Contains2“-Kante e3 ebenfalls in den Zielgraphen kopiert. Im Quellgraph sind zwei Klassen (v11 und v12) vorhanden, die über die Kanten e11 und e12 im Paket (Knoten v9) deklariert wurden. Diese wurden als v3 und v4 in den Zielgraph kopiert, zusammen mit den ”DeclaredIn2“-Kanten, die im Zielgraph als ”Contains1“-Kanten e1 und e2 erscheinen. Die Instanzen von ”Member“ und ”Method“ sind ebenfalls in den Zielgraphen kopiert worden und den entsprechenden Instanzen von ”Class“ zugeordnet worden.

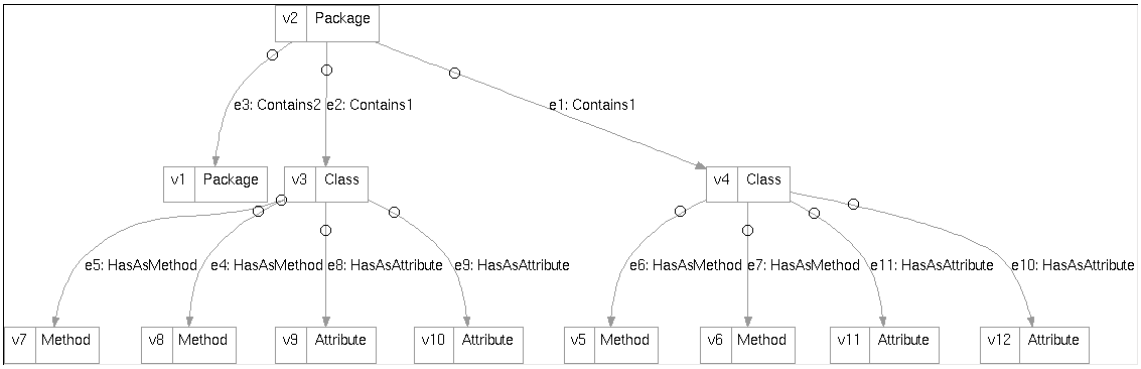

Abbildung 8.5: Reference-Graph

8.3 Transformation: Referenz- zu UMLSchema

In Listing 8.2 wird analog zum vorherigen Beispiel die Transformation vom Referenzschema zum UMLSchema definiert. Nachdem auch hier der Quellcode erläutert ist, wird der erzeugte Graph der ersten Transformation als Quellgraph zu einem Graphen transformiert, der dem UMLSchema entspricht.

Listing 8.2: Beispiel: Transformation Referenz- zu UMLSchema

```
1  package de.uni_koblenz.jgralab.gretl;
2  import de.uni_koblenz.jgralab.schema.VertexClass;
3  public class ReferenceToUML extends Transformation {
4     public ReferenceToUML(TransformationContext context) {
5        super(context);
6     }
7     @Override
8     public void transform() throws Exception {
9        // Abstrakte Knotenklasse PackageableElement erzeugen
10       VertexClass packageableElement = createVertexClass("PackageableElement", true, "");
11       // Class kopieren
12       VertexClass pkg = createVertexClass("Package", false, "V{Package}");
13       // Class kopieren
14       VertexClass cls = createVertexClass("Class", false, "V{Class}");
15       // Knotenklasse Operation erzeugen
16       VertexClass operation = createVertexClass("Operation", false, "V{Method}");
17       // Knotenklasse Property erzeugen
18       VertexClass property = createVertexClass("Property", false, "V{Attribute}");
19       // Generalisierungen von Package und Class zu PackageableElement erzeugen
20       addSuperClass(cls, packageableElement);
21       addSuperClass(pkg, packageableElement);
22       // Kompositionsklasse c1 erzeugen
23       createCompositionClass("Contains1", pkg, true, packageableElement, false,
24          "E{Contains1, Contains2}",
25          "from e:E{Contains1, Contains2} reportMap e, startVertex(e) end",
26          "from e:E{Contains1, Contains2} reportMap e, endVertex(e) end"
27       );
28       // Kompositionsklasse c2 erzeugen
29       createCompositionClass("Contains2", cls, true, property, false,
30          "E{HasAsAttribute}",
```

```
31            "from e:E{HasAsAttribute} reportMap e, startVertex(e) end",
32            "from e:E{HasAsAttribute} reportMap e, endVertex(e) end"
33        );
34        // Kompositionsklasse c3 erzeugen
35        createCompositionClass("Contains3", cls, true, operation, false,
36            "E{HasAsMethod}",
37            "from e:E{HasAsMethod} reportMap e, startVertex(e) end",
38            "from e:E{HasAsMethod} reportMap e, endVertex(e) end"
39        );
40    }
41 }
```

In Zeile 10 des Listings wird zuerst die abstrakte Klasse ”PackageableElement“ angelegt, welche die Oberklasse der Klassen ”Package“ , welche in Zeile 12 angelegt wird, und ”Class“, welche in Zeile 14 angelegt wird. Als Urbildmenge dieser beiden Knotenklassen wird mit den semantischen Ausdrücken die Knotenmenge von ”Package“ bzw. ”Class“ aus dem Quellgraphen abgefragt. In den Zeilen 20 und 21 werden die Generlisierungsbeziehung von ”Class“ und ”Package“ zu ”PackageableElement“ angelegt.

Die Knotenklasse ”Method“ des Quellschemas wird in Zeile 16 auf die Knotenklasse ”Operation“ im Zielschema kopiert. Die entsprechenden Instanzen werden durch den semantischen Ausdruck ebenfalls kopiert. Das Kopieren von ”Attribute“ des Quellschemas zu ”Property“ des Zielschemas erfolgt in Zeile 18 analog dazu.

Die Zielen 23 bis 27 erzeugen die Kompositionsklasse ”Contains1“ im Zielschema, welche einer Zusammenlegung von ”Contains1“ und ”Contains2“ des Quellschemas entspricht. Als Urbildmenge für die zu erzeugen Instanzen wird die Vereinigung der Instanzen von ”Contains1“ und ”Contains2“ des Quellgraphen bestimmt.

Zuletzt werden in den Zeilen 29 bis 33 bzw. 35 bis 39 die Kompositionsklassen ”Contains2“ und ”Contains3“ erzeugt, welche Kopien der Kompositionsklassen ”HasAsAttribute“ und ”HasAsMethod“ des Quellschemas sind. Die dazu gehörenden Instanzen werden ebenfalls vollständig kopiert.

Abbildung 8.6 zeigt den mit Hilfe der Transformation erzeugten Graphen, welcher aus dem Graph in Abbildung 8.5 erzeugt wurde. Die Abbildung zeigt deutlich, dass alle Elemente entsprechend kopiert worden sind. Einzig die Knoten- bzw. Kantennummern entsprechen nicht mehr denen des ursprünglichen Graphen (z.B. wurde v1 des Zielgraphen aus v2 des Quellgraphen erzeugt). Diese Nummerierung existiert allerdings nur intern in JGraLab und spielt für das Endergebnis keine Rolle.

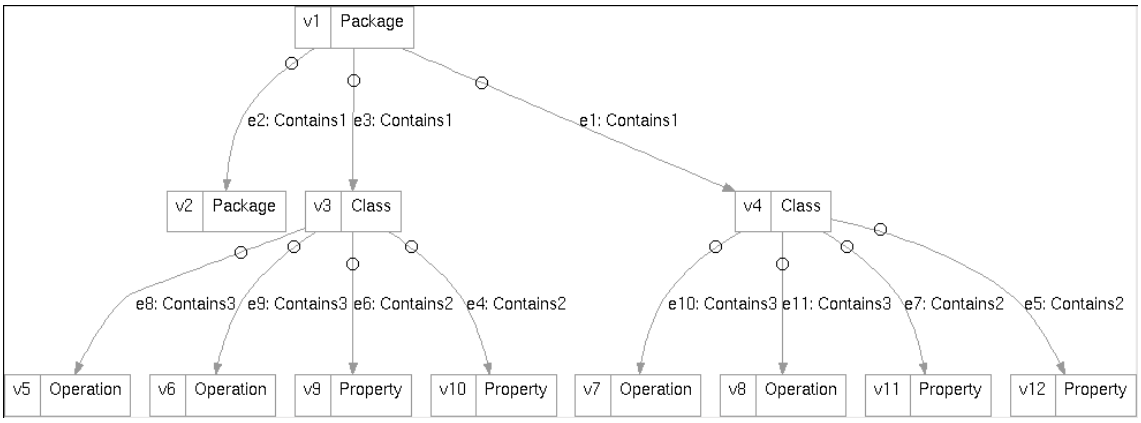

Abbildung 8.6: UML-Graph

9 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, den in [Rhe06] dargestellten operationalen Ansatz zur Modelltransformation mit Hilfe der am Institut für Softwaretechnik der Universität Koblenz-Landau vorhandenen Bibliotheken “JGraLab” und “GReQL” in Java zu implementieren. Die Implementierung sollte beweisen, dass der aufgezeigte Transformationsansatz in der Praxis umsetzbar ist. Dies wurde durch verschiedene Beispiele bewiesen.

Die geplante Verwendung in weiteren Projekten des IST wird für die Zukunft zeigen, ob sich weitere Transformationen umsetzten lassen oder wo die Grenzen des Ansatzes sind. Des weiteren ist denkbar, die Transformationen nicht mehr in zwei Schritten (Schematransformation vor Graphtransformation), sondern beide Schritte auf einmal ablaufen zu lassen. Dieser Schritt setzt jedoch voraus, dass JGraLab dies ebenfalls unterstützt.

A Operationsmenge zum JGraLab Meta-Meta-Schema

In den nachfolgenden Auflistungen werden alle Signaturen der elementaren Operationen basierend auf dem JGraLab Meta-Meta-Schema in alphabetischer Reihenfolge dargestellt. Die Signaturen von Attribute und EdgeClass wurden entsprechend den Anforderungen an die Semantik auf M1-Ebene angepasst, um die Transformation auf Modellebene zu ermöglichen.

A.1 Operationen zum Attribut- und Graphenbereich

AggregationClass (Klasse)

A1	$\text{Vertex}_{\text{AggregationClass}}$ createAggregationClass(String name, boolean isAbstract, boolean aggregateFrom, $\text{Vertex}_{\text{VertexClass}}$ from, List attributesFrom, $\text{Vertex}_{\text{VertexClass}}$ to, List attributesTo)
A2	void deleteAggregationClass($\text{Vertex}_{\text{AggregationClass}}$ v)
A3	boolean getAggregateFrom($\text{Vertex}_{\text{AggregationClass}}$ v)
A4	void setAggregateFrom($\text{Vertex}_{\text{AggregationClass}}$ v, boolean aggregateFrom)
A5	$\text{Vertex}_{\text{AggregationClass}}$ copyAggregationClass($\text{Vertex}_{\text{AggregationClass}}$ v)

Attribute (Klasse)

A1	$\text{Vertex}_{\text{Attribute}}$ createAttribute(String name, $\text{Vertex}_{\text{AttributedElementClass}}$ attributedElementClass, $\text{Vertex}_{\text{Domain}}$ domain)
A2	void deleteAttribute($\text{Vertex}_{\text{Attribute}}$ v)
A3	String getName($\text{Vertex}_{\text{Attribute}}$ v)
A4	void setName($\text{Vertex}_{\text{Attribute}}$ v, String name)
A5	$\text{Vertex}_{\text{Attribute}}$ copyAttribute($\text{Vertex}_{\text{Attribute}}$ v)

AttributedElementClass (abstrakte Klasse)

A1	*nicht bei abstrakten Klassen*
A2	void deleteAttributedElementClass($Vertex_{AttributedElementClass}$ v)
A3	String getName($Vertex_{AttributedElementClass}$ v) boolean getIsAbstract($Vertex_{AttributedElementClass}$ v)
A4	void setName($Vertex_{AttributedElementClass}$ v, String name) void setIsAbstract($Vertex_{AttributedElementClass}$ v, boolean isAbstract)
A5	$Vertex_{AttributedElementClass}$ copyAttributedElementClass ($Vertex_{AttributedElementClass}$ v)

CompositionClass (Klasse)

A1	$Vertex_{CompositionClass}$ createCompositionClass(String name, boolean isAbstract, boolean aggregateFrom, $Vertex_{VertexClass}$ from, List attributesFrom, $Vertex_{VertexClass}$ to, List attributesTo)
A2	void deleteCompositionClass($Vertex_{CompositionClass}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{CompositionClass}$ copyCompositionClass($Vertex_{CompositionClass}$ v)

containsDefaultPackage (Aggregation zwischen Schema und Package)

B1	$Edge_{containsDefaultPackage}$ createContainsDefaultPackage($Vertex_{Schema}$ alpha, $Vertex_{Package}$ omega)
B2	void deleteContainsDefaultPackage($Edge_{containsDefaultPackage}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsDefaultPackage}$ copyContainsDefaultPackage ($Edge_{containsDefaultPackage}$ e)
B6.1	void moveContainsAlpha($Edge_{containsDefaultPackage}$ e, $Vertex_{Schema}$ alpha)
B6.2	void moveContainsOmega($Edge_{containsDefaultPackage}$ e, $Vertex_{Package}$ omega)

containsDomain (Aggregation zwischen Schema und Domain)

B1	$Edge_{containsDomain}$ createContainsDomain($Vertex_{Schema}$ alpha, $Vertex_{Domain}$ omega)
B2	void deleteContainsDomain($Edge_{containsDomain}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsDomain}$ copyContainsComain($Edge_{containsDomain}$ e)
B6.1	void moveContainsDomainAlpha($Edge_{containsDomain}$ e, $Vertex_{Schema}$ v)
B6.2	void moveContainsDomainOmega($Edge_{containsDomain}$ e, $Vertex_{Domain}$ v)

containsGraphElementClass (Komposition zwischen Package und GraphElement-Class)

B1	$Edge_{containsGraphElementClass}$ createContainsGraphElementClass ($Vertex_{Package}$ alpha, $Vertex_{GraphElementClass}$ omega)
B2	void deleteContainsGraphElementClass($Edge_{containsGraphElementClass}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsGraphElementClass}$ copyContainsGraphElementClass ($Edge_{containsGraphElementClass}$ e)
B6.1	void moveContainsGraphElementClassAlpha ($Edge_{containsGraphElementClass}$ e, $Vertex_{Package}$ v)
B6.2	void moveContainsGraphElementClassOmega ($Edge_{containsGraphElementClass}$ e, $Vertex_{GraphElementClass}$ v)

containsSubPackage (Aggregation an Package)

B1	$Edge_{containsSubPackage}$ createContainsSubPackage($Vertex_{Package}$ alpha, $Vertex_{Package}$ omega)
B2	void deleteContainsSubPackage($Edge_{containsSubPackage}$)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsSubPackage}$ copyContainsSubPackage($Edge_{containsSubPackage}$ e)
B6.1	void moveContainsSubPackageAlpha($Edge_{containsSubPackage}$ e, $Vertex_{Package}$ alpha)
B6.2	void moveContainsSubPackageOmege($Edge_{containsSubPackage}$ e, $Vertex_{Package}$ omega)

definesGraphClass (Komposition zwischen Schema und GraphClass

B1	$Edge_{containsGraphClass}$ createContainsGraphClass($Vertex_{Schema}$ alpha, $Vertex_{GraphClass}$ omega)
B2	void deleteContainsGraphClass($Edge_{containsGraphClass}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsGraphClass}$ copyContainsGraphClass($Edge_{containsGraphClass}$ e)
B6.1	void moveContainsGraphClassAlpha($Edge_{containsGraphClass}$ e, $Vertex_{Schema}$ alpha)
B6.2	void moveContainsGraphClassOmega($Edge_{containsGraphClass}$ e, $Vertex_{GraphClass}$ omega)

Domain (abstrakte Klasse)

A1	*nicht bei abstrakten Klassen*
A2	void deleteEdgeClass($Vertex_{EdgeClass}$ v)
A3	String getName($Vertex_{Schema}$ v)
A4	void setName($Vertex_{Schema}$ v, String name)
A5	$Vertex_{EdgeClass}$ copyEdgeClass($Vertex_{EdgeClass}$ v)

EdgeClass (Klasse)

A1	$Vertex_{EdgeClass}$ createEdgeClass(String name, boolean isAbstract, $Vertex_{VertexClass}$ from, List attributesFrom, $Vertex_{VertexClass}$ to, List attributesTo)
A2	void deleteEdgeClass($Vertex_{EdgeClass}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{EdgeClass}$ copyEdgeClass($Vertex_{EdgeClass}$ v)

GraphClass (Klasse)

A1	$Vertex_{GraphClass}$ createGraphClass(String name, boolean isAbstract)
A2	void deleteGraphClass($Vertex_{GraphClass}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{GraphClass}$ copyGraphClass($Vertex_{GraphClass}$ v)

GraphElementClass (abstrakte Klasse)

A1	*nicht bei abstrakten Klassen*
A2	void deleteGraphElementClass($Vertex_{GraphElementClass}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{GraphElementClass}$ copyGraphElementClass($Vertex_{GraphElementClass}$ v)

Package (Klasse)

A1	$Vertex_{Package}$ createPackage(String name)
A2	void deletePackage($Vertex_{Package}$ v)
A3	String getName($Vertex_{Package}$ v)
A4	void setName($Verex_{Package}$ v, String name)
A5	void deletePackage($Vertex_{Package}$ v)

Schema (Klasse)

A1	$Vertex_{Schema}$ createSchema(String name, String packagePrefix)
A2	void deleteSchema($Vertex_{Schema}$ v)
A3	String getName($Vertex_{Schema}$ v) String getPackagePrefix($Vertex_{Schema}$ v)
A4	void setName($Vertex_{Schema}$ v, String name) void setPackagePrefix($Vertex_{Schema}$ v, String packagePrefix)
A5	$Vertex_{Schema}$ copySchema($Vertex_{Schema}$ v)

specializesEdgeClass (Assoziation an EdgeClass)

B1	$Edge_{specializesEdgeClass}$ createSpecializesEdgeClass($Vertex_{EdgeClass}$ alpha, $Vertex_{EdgeClass}$ omega)
B2	void deleteSpecializesEdgeClass($Edge_{specializesEdgeClass}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{specializesEdgeClass}$ copySpecializesEdgeClass($Edge_{specializesEdgeClass}$ e)
B6.1	void moveSpecializesEdgeClassAlpha($Edge_{specializesEdgeClass}$ e, $Vertex_{EdgeClass}$ alpha)
B6.2	void moveSpecializesEdgeClassOmega($Edge_{specializesEdgeClass}$ e, $Vertex_{EdgeClass}$ omega)

specializesVertexClass (Assoziation an VertexClass)

B1	$Edge_{specializesVertexClass}$ createSpecializesVertexClass($Vertex_{VertexClass}$ alpha, $Vertex_{VertexClass}$ omega)
B2	void deleteSpecializesVertexClass($Edge_{specializesVertexClass}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{specialiesVertexClass}$ copySpecializesVertexClass ($Edge_{specializesVertexClass}$ e)
B6.1	void moveSpecializesVertexClassAlpha($Edge_{specializesVertexClass}$ e, $Vertex_{VertexClass}$ alpha)
B6.2	void moveSpecializesVertexClassOmega($Edge_{specializesVertexClass}$ e, $Vertex_{VertexClass}$ omega)

VertexClass (Klasse)

A1	$Vertex_{VertexClass}$ createVertexClass(String name, boolean isAbstract)
A2	void deleteVertexClass($Vertex_{VertexClass}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{VertexClass}$ copyVertexClass($Vertex_{VertexClass}$ v)

A.2 Operationen zum Wertebereichsystem

BooleanDomain (Klasse)

A1	$Vertex_{BooleanDomain}$ createBooleanDomain(String name)
A2	void deleteBooleanDomain($Vertex_{BooleanDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{BooleanDomain}$ copyBooleanDomain($Vertex_{BooleanDomain}$ v)

containsEnumDomain (Komposition zwischen Package und EnumDomain)

B1	$Edge_{containsEnumDomain}$ createContainsEnumDomain($Vertex_{Package}$ alpha, $Vertex_{EnumDomain}$ omega)
B2	void deleteContainsEnumDomain($Edge_{containsEnumDomain}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsEnumDomain}$ copyContainsEnumDomain ($Edge_{containsEnumDomain}$ e)
B6.1	void moveContainsEnumDomainAlpha($Edge_{containsEnumDomain}$ e, $Vertex_{Package}$ alpha)
B6.2	void moveContainsEnumDomainOmega($Edge_{containsEnumDomain}$ e, $Vertex_{EnumDomain}$ omega)

containsRecordDomain (Komposition zwischen Package und RecordDomain)

B1	$Edge_{containsRecordComain}$ createContainsRecordDomain($Vertex_{Package}$ alpha, $Vertex_{RecordDomain}$ omega)
B2	void deleteContainsRecordDomain($Edge_{containsRecordDomain}$ e)
B3	*enthält keine Attribute*
B4	*enthält keine Attribute*
B5	$Edge_{containsRecordDomain}$ copyContainsRecordDomain ($Edge_{containsRecordDomain}$ e)
B6.1	void moveContainsRecordDomainAlpha($Edge_{containsRecordDomain}$ e, $Vertex_{Package}$ alpha)
B6.2	void moveContainsRecordDomainOmega($Edge_{containsRecordDomain}$ e, $Vertex_{RecordDomain}$ omega)

DoubleDomain (Klasse)

A1	$Vertex_{DoubleDomain}$ createDoubleDomain(String name)
A2	void deleteDoubleDomain($Vertex_{DoubleDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{DoubleDomain}$ copyDoubleDomain($Vertex_{DoubleDomain}$ v)

EnumDomain (Klasse)

A1	$Vertex_{EnumDomain}$ createEnumDomain(String name, List <String>enumConstants)
A2	void deleteEnumDomain($Vertex_{EnumDomain}$ v)
A3	String getName($Vertex_{EnumDomain}$ v) List <String>getEnumConstants($Vertex_{EnumDomain}$ v)
A4	void setName($Vertex_{EnumDomain}$ v, String name) void setEnumConstants($Vertex_{EnumDomain}$ v, List <String>enumConstants)
A5	$Vertex_{EnumDomain}$ copyEnumDomain($Vertex_{EnumDomain}$ v)

hasRecordDomainComponent (Aggregation zwischen RecordDomain und Domain)

B1	$Edge_{hasRecordDomainComponent}$ createHasRecordDomainComponent ($Vertex_{RecordDomain}$ alpha, $Vertex_{Domain}$ omega, String name)
B2	void deleteHasRecordDomainComponent ($Edge_{hasRecordDomainComponent}$ e)
B3	String getName($Edge_{hasRecordDomainComponent}$ e)
B4	void setName($Edge_{hasRecordDomainComponent}$ e, String name)
B5	$Edge_{hasRecordDomainComponent}$ copyHasRecordDomainComponent ($Edge_{hasRecordDomainComponent}$ e)
B6.1	void moveHasRecordDomainComponentAlpha ($Edge_{hasRecordDomainComponent}$ e, $Vertex_{RecordDomain}$ alpha)
B6.2	void moveHasRecordDomainComponentOmega ($Edge_{hasRecordDomainComponent}$ e, $Vertex_{Domain}$ omega)

IntDomain (Klasse)

A1	$Vertex_{IntDomain}$ createIntDomain(String name)
A2	void deleteIntDomain($Vertex_{IntDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{IntDomain}$ copyIntDomain($Vertex_{IntDomain}$ v)

ListDomain (Klasse)

A1	$Vertex_{ListDomain}$ createListDomain(String name, $Vertex_{Domain}$ domain)
A2	void deleteListDomain($Vertex_{ListDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{ListDomain}$ copyListDomain($Vertex_{ListDomain}$ v)

LongDomain (Klasse)

A1	$Vertex_{LongDomain}$ createLongDomain(String name)
A2	void deleteLongDomain($Vertex_{LongDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{LongDomain}$ copyLongDomain($Vertex_{LongDomain}$ v)

ObjectDomain (Klasse)

A1	$Vertex_{ObjectDomain}$ createObjectDomain(String name)
A2	void deleteObjectDomain($Vertex_{ObjectDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	void deleteObjectDomain($Vertex_{ObjectDomain}$ v)

RecordDomain (Klasse)

A1	$Vertex_{RecordDomain}$ createRecordDomain(String name)
A2	void deleteRecordDomain($Vertex_{RecordDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{RecordDomain}$ copyRecordDomain($Vertex_{RecordDomain}$ v)

SetDomain (Klasse)

A1	$Vertex_{SetDomain}$ createSetDomain(String name, $Vertex_{Domain}$ domain)
A2	void deleteSetDomain($Vertex_{SetDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{SetDomain}$ copySetDomain($Vertex_{SetDomain}$ v)

StringDomain (Klasse)

A1	$Vertex_{StringDomain}$ createStringDomain(String name)
A2	void deleteStringDomain($Vertex_{StringDomain}$ v)
A3	*enthält keine Attribute*
A4	*enthält keine Attribute*
A5	$Vertex_{StringDomain}$ copyStringDomain($Vertex_{StringDomain}$ v)

B Methoden der Transformations-Klasse

Im folgenden sind alle verfügbaren Methoden der Transformation-Klasse noch einmal in alphabetischer Reihenfolge mit allen detaillierten Erläuterungen der einzelnen Parameter aufgeführt. Im ersten Abschnitt werden dabei die Funktionen zum Operationsbereich aufgeführt. Die jeweiligen Parameter sind dabei nach der Übersicht aller Signaturen der jeweiligen Methode tabellarisch aufgeführt und erläutert. Im zweiten Abschnitt werden die jeweiligen Parameter der Funktionen zum Wertebereich aufgelistet.

B.1 Funktionen zum Operationsbereich

createAggregationClass

createAggregationClass erzeugt im Zielschema eine neue Aggregationsklasse. Als semantische Ausdrücke zur Generierung der Instanzen wird jeweils ein semantischer Ausdruck für die Urbildmenge erwartet, aus der die Instanzen erzeugt werden sollen sowie zwei Ausdrücke mit einer Abbildung von dieser Urbildmenge auf die Urbildmenge der Knoten am Anfang und Ende der Aggregation. Die neu erzeugte Aggregationsklasse wird zurückgegeben.

public final AggregationClass createAggregationClass(String name, VertexClass from, Boolean aggregateFrom, VertexClass to, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, Boolean aggregateFrom, VertexClass to, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, String fromRoleName, Boolean aggregateFrom, VertexClass to, String toRoleName,

Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, String fromRoleName, Boolean aggregateFrom, VertexClass to, String toRoleName, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, int fromMin, int fromMax, Boolean aggregateFrom, VertexClass to, int toMin, int toMax, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, int fromMin, int fromMax, Boolean aggregateFrom, VertexClass to, int toMin, int toMax, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, int fromMin, int fromMax, String fromRoleName, Boolean aggregateFrom, VertexClass to, int toMin, int toMax, String toRoleName, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final AggregationClass createAggregationClass(String name, VertexClass from, int fromMin, int fromMax, String fromRoleName, Boolean aggregateFrom, VertexClass to, int toMin, int toMax, String toRoleName, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

String name	Name der anzulegenden Aggregationsklasse
VertexClass from	Knotenklasse am Anfang der Aggregationsklasse
int fromMin	Minimale Kardinalität am Anfang der Aggregationsklasse
int fromMax	Maximale Kardinalität am Anfang der Aggregationsklasse
String fromRoleName	Name der Rolle am Anfang der Aggregationsklasse
Boolean aggregateFrom	Angabe, ob der Anfang der Aggregationsklasse am from- oder to-Ende liegen soll
VertexClass to	Knotenklasse am Ende der Aggregationsklasse
int toMin	Minimale Kardinalität am Ende der Aggregationsklasse
int toMax	Maximale Kardinalität am Ende der Aggregationsklasse
String toRoleName	Name der Rolle am Ende der Aggregationsklasse

Boolean isAbstract	Angabe, ob die Aggregationsklasse als abstrakt definiert werden soll
String semanticExpressionEdge	Semantischer GReQL-Ausdruck, welcher die Urbildmenge der zu erzeugenden Instanzen als Set liefert
JValueSet semanticExpressionEdge	Die Urbildmenge der zu erzeugenden Instanzen als Menge (JValueSet)
String semanticExpressionFrom	Semantischer GReQL-Ausdruck, welcher die Abbildung von der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Anfangsknoten liefert
JValueMap semanticExpressionFrom	Die Abbildung der Urbildmenge der zu Erzeugenden Instanzen auf die Urbildmenge der Anfangsknoten als JValueMap
String semanticExpressionTo	Semantischer GReQL-Ausdruck, welcher die Abbildung von der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Endknoten liefert
JValueMap semanticExpressionTo	Die Abbildung der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Endknoten als JValueMap

createAttribute

createAttribute erzeugt ein neues Attribut an einem Element des Zielschemas. Das Element kann eine Knotenklasse, eine Kantenklasse oder die Graphklasse im Zielschema sein. Das neu erzeute Attribut wird zurückgegeben.

public final Attribute createAttribute(String name, AttributedElementClass element, Domain domain, String semanticExpression)

public final Attribute createAttribute(String name, AttributedElementClass element, Domain domain, JValueMap semanticExpression)

public final Attribute createAttribute(String name, AttributedElementClass element, String domainName, String semanticExpression)

public final Attribute createAttribute(String name, AttributedElementClass element, String domainName, JValueMap semanticExpression)

String name	Name des anzulegenden Attributes
AttributedElement-Class element	Die Knoten- oder Kantenklasse bzw. Graphklasse, bei welcher das Attribut angelegt werden soll
Domain domain	Der Wertebereich, welcher die Werte die das Attribut annehmen kann definiert, als Domain-Objekt
String domain	Der Wertebereich, welcher die Werte die das Attribut annehmen kann definiert, für einfache Wertebereiche ist hier die Angabe des Namens der Domain möglich
String semantic-Expression	GReQL-Ausdruck, welcher eine Abbildung von der Urbildmenge der Instanzen der AttributedElementClass auf deren jeweilige Attributwerte generiert
JValueMap semanticExpression	Die Abbildung von der Urbildmenge der Instanzen der AttributedElementClass auf deren jeweilige Attributwerte als JValueMap

createCompositionClass

createAggregationClass erzeugt im Zielschema eine neue Kompositionsklasse. Als semantische Ausdrücke zur Generierung der Instanzen wird jeweils ein semantischer Ausdruck für die Urbildmenge erwartet, aus der die Instanzen erzeugt werden sollen sowie zwei Ausdrücke mit einer Abbildung von dieser Urbildmenge auf die Urbildmenge der Knoten am Anfang und Ende der Komposition. Die neu erzeugte Kompositionsklasse wird zurückgegeben.

public final CompositionClass createCompositionClass(String name, VertexClass from, boolean compositeFrom, VertexClass to, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from, boolean compositeFrom, VertexClass to, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from, String fromRoleName, boolean compositeFrom, VertexClass to, String toRoleName, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from,

String fromRoleName, boolean compositeFrom, VertexClass to, String toRoleName, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from, int fromMin, int fromMax, boolean compositeFrom, VertexClass to, int toMin, int toMax, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from, int fromMin, int fromMax, boolean compositeFrom, VertexClass to, int toMin, int toMax, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from, int fromMin, int fromMax, String fromRoleName, Boolean compositeFrom, VertexClass to, int toMin, int toMax, String toRoleName, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final CompositionClass createCompositionClass(String name, VertexClass from, int fromMin, int fromMax, String fromRoleName, Boolean compositeFrom, VertexClass to, int toMin, int toMax, String toRoleName, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

String name	Name der anzulegenden Kompositionsklasse
VertexClass from	Knotenklasse am Anfang der Kompositionsklasse
int fromMin	Minimale Kardinalität am Anfang der Kompositionsklasse
int fromMax	Maximale Kardinalität am Anfang der Kompositionsklasse
String fromRoleName	Name der Rolle am Anfang der Kompositionsklasse
Boolean compositeFrom	Angabe, ob der Anfang der Kompositionsklasse am from- oder to-Ende liegen soll
VertexClass to	Knotenklasse am Ende der Kompositionsklasse
int toMin	Minimale Kardinalität am Ende der Kompositionsklasse
int toMax	Maximale Kardinalität am Ende der Kompositionsklasse
String toRoleName	Name der Rolle am Ende der Kompositionsklasse
Boolean isAbstract	Angabe, ob die Kompositionsklasse als abstrakt definiert werden soll

String semanticExpressionEdge	Semantischer GReQL-Ausdruck, welcher die Urbildmenge der zu erzeugenden Instanzen als Set liefert
JValueSet semanticExpressionEdge	Die Urbildmenge der zu erzeugenden Instanzen als Menge (JValueSet)
String semanticExpressionFrom	Semantischer GReQL-Ausdruck, welcher die Abbildung von der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Anfangsknoten liefert
JValueMap semanticExpressionFrom	Die Abbildung der Urbildmenge der zu Erzeugenden Instanzen auf die Urbildmenge der Anfangsknoten als JValueMap
String semanticExpressionTo	Semantischer GReQL-Ausdruck, welcher die Abbildung von der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Endknoten liefert
JValueMap semanticExpressionTo	Die Abbildung der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Endknoten als JValueMap

createEdgeClass

createEdgeClass erzeugt im Zielschema eine neue Kantenklasse. Als semantische Ausdrücke zur Generierung der Instanzen wird jeweils ein semantischer Ausdruck für die Urbildmenge erwartet, aus der die Instanzen erzeugt werden sollen sowie zwei Ausdrücke mit einer Abbildung von dieser Urbildmenge auf die Urbildmenge der Knoten am Anfang und Ende der Kante. Die neu erzeugte Kantenklasse wird zurückgegeben.

public final EdgeClass createEdgeClass(String name, VertexClass from, VertexClass to, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, VertexClass to, Boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, String fromRoleName, VertexClass to, String toRoleName, Boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, String fromRoleName, VertexClass to, String toRoleName, boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, int fromMin, int fromMax, VertexClass to, int toMin, int toMax, boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, int fromMin, int fromMax, VertexClass to, int toMin, int toMax, boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, int fromMin, int fromMax, String fromRoleName, VertexClass to, int toMin, int toMax, String toRoleName, boolean isAbstract, String semanticExpressionEdge, String semanticExpressionFrom, String semanticExpressionTo)

public final EdgeClass createEdgeClass(String name, VertexClass from, int fromMin, int fromMax, String fromRoleName, VertexClass to, int toMin, int toMax, String toRoleName, boolean isAbstract, JValueSet semanticExpressionEdge, JValueMap semanticExpressionFrom, JValueMap semanticExpressionTo)

String name	Name der anzulegenden Kantenklasse
VertexClass from	Knotenklasse am Anfang der Kantenklasse
int fromMin	Minimale Kardinalität am Anfang der Kantenklasse
int fromMax	Maximale Kardinalität am Anfang der Kantenklasse
String fromRoleName	Name der Rolle am Anfang der Kantenklasse
VertexClass to	Knotenklasse am Ende der Kantenklasse
int toMin	Minimale Kardinalität am Ende der Kantenklasse
int toMax	Maximale Kardinalität am Ende der Kantenklasse
String toRoleName	Name der Rolle am Ende der Kantenklasse
Boolean isAbstract	Angabe, ob die Kantenklasse als abstrakt definiert werden soll
String semanticExpressionEdge	Semantischer GReQL-Ausdruck, welcher die Urbildmenge der zu erzeugenden Instanzen als Set liefert

JValueSet semanticExpressionEdge	Die Urbildmenge der zu erzeugenden Instanzen als Menge (JValueSet)
String semanticExpressionFrom	Semantischer GReQL-Ausdruck, welcher die Abbildung von der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Anfangsknoten liefert
JValueMap semanticExpressionFrom	Die Abbildung der Urbildmenge der zu Erzeugenden Instanzen auf die Urbildmenge der Anfangsknoten als JValueMap
String semanticExpressionTo	Semantischer GReQL-Ausdruck, welcher die Abbildung von der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Endknoten liefert
JValueMap semanticExpressionTo	Die Abbildung der Urbildmenge der zu erzeugenden Instanzen auf die Urbildmenge der Endknoten als JValueMap

addSuperClass

addSuperClass fügt einer Knoten- oder Kantenklasse eine Oberklasse hinzu. Die Methode hat keinen Rückgabewert.

public final void addSuperClass(EdgeClass subClass, EdgeClass superClass)

public final void addSuperClass(VertexClass subClass, VertexClass superClass)

EdgeClass subClass	Kantenklasse, der eine Oberklasse zugeordnet werden soll
EdgeClass superClass	Kantenklasse, der die Unterklasse subClass zugeordnet werden soll
VertexClass subClass	Knotenklasse, der eine Oberklasse zugeordnet werden soll
VertexClass superClass	Knotenklasse, der die Unterklasse subClass zugeordnet werden soll

createVertexClass

createVertexClass erzeugt im Zielschema eine neue Knotenklasse. Im Zielgraph werden die entsprechenden Instanzen angelegt. Die neu erzeugte Knotenklasse wird zurückgegeben.

public final VertexClass createVertexClass(String name, boolean isAbstract, String semanticExpression)

public final VertexClass createVertexClass(String name, boolean isAbstract, JValueSet semanticExpression)

String name	Der Name der zu erzeugenden Knotenklasse
boolean isAbstract	Angabe, ob die zu erzeugende Knotenklasse als abstrakt definiert sein soll
String semanticExpression	GReQL-Ausdruck, welcher die Urbildmenge der zu erzeugenden Instanzen als Menge (Set) liefert
JValueSet semanticExpression	Die Urbildmenge der zu erzeugenden Instanzen als Menge (JValueSet)

B.2 Funktionen zum Wertebereichssystem

createEnumDomain

Erzeugt eine neue Emum-Domain im Zielschema. Die neu erzeugte Domain wird zurückgegeben.

public final Domain createEnumDomain(String name, List<String> enumConstants)

String name	Der Name der zu erzeugenden Enum-Domain
List<String> enumConstants	Eine String-Liste, in der die möglichen Konstanten des Wertebereiches definiert sind.

createListDomain

Erzeugt eine neue List-Domain im Zielschema. Der Name der List-Domain wird anhand des Namens der übergebenen Base-Domain erzeugt. Die neu erzeugte Domain wird zurückgegeben.

public final Domain createListDomain(Domain baseDomain)

Domain baseDomain	Der Wertebereich, der als Wertebereich für die List-Domain dienen soll

createRecordDomain

Erzeugt eine neue Record-Domain im Zielschema. Werden keine recordComponents übergeben, wird eine leere Record-Domain erzeugt. Die neu erzeugte Domain wird zurückgegeben.

public final Domain createRecordDomain(String name)

public final Domain createRecordDomain(String name, Map<String, Domain> recordComponents)

String name	Der name der zu erzeugenden Record-Domain
Map<String, Domain> recordComponents	Eine Abbildung der Record-Domain-Elemente auf deren jeweilige Wertebereiche

createSetDomain

Erzeugt im Zielschema eine neue Set-Domain. Die neu erzeugte Domain wird zurückgegeben.

public final Domain createSetDomain(Domain baseDomain)

Domain baseDomain	Der Wertebereich, der als Wertebereich für die Set-Domain dienen soll

createMapDomain

Erzeugt eine neue Map-Domain im Zielschema. Die neu erzeugte Domain wird zurückgegeben.

public final Domain createMapDomain(Domain keyDomain, Domain valueDomain)

Domain keyDomain	Der Wertebereich, dem die Urbildelemente der Map-Domain entsprechen
Domain valueDomain	Der Wertebereich, dem die Bildelemente der Map-Domain entsprechen

createBooleanDomain

Gibt die Instanz der BooleanDomain des Zielschemas zurück. Die Methode erwartet keine Eingabeparameter.

public final Domain createBooleanDomain()

createIntegerDomain

Gibt die Instanz der IntegerDomain des Zielschemas zurück. Die Methode erwartet keine Eingabeparameter.

public final Domain createIntegerDomain()

createLongDomain

Gibt die Instanz der LongDomain des Zielschemas zurück. Die Methode erwartet keine Eingabeparameter.

public final Domain createLongDomain()

createStringDomain

Gibt die Instanz der StringDomain des Zielschemas zurück. Die Methode erwartet keine Eingabeparameter.

public final Domain createStringDomain()

createDoubleDomain

Gibt die Instanz der DoubleDomain des Zielschemas zurück. Die Methode erwartet keine Eingabeparameter.

public final Domain createDoubleDomain()

C Methoden der TransformationContext-Klasse

Im folgenden sind alle verfügbaren Methoden der TransformationContext-Klasse in alphabetischer Reihenfolge mit detaillierten Erläuterungen der einzelnen Parameter aufgeführt.

TransformationContext

Der Konstruktor des Kontext-Objektes.

public TransformationContext(String targetSchemaName, String targetGraphClassName)

public TransformationContext(String targetSchemaName, String targetGraphClassName, boolean miracleMode)

String targetSchemaName	Der QualifiedName der zu erzeugenden Zielschemas
String targetGraphClassName	Der Name der Graphklasse im Zielgraph
Boolean miracleMode	Gibt an, ob der Miracle-Mode (vorausschauende Modus bei Anlegen von Attributen) eingeschaltet sein soll (default: ein)

getSourceGraph

Gibt den Quellgraphen zurück.

public final Graph getSourceGraph()

getSourceSchema

Gibt das Quellschema zurück.

public final Schema getSourceSchema()

setSourceGraph

Setzt den Quellgraphen. Die Methode hat keinen Rückgabewert

public final void setSourceGraph(Graph sourceGraph)

Graph sourceGraph	Der Graph, der als Quellgraph verwendet werden soll

setSourceSchema

Setzt das Quellschema. Die Methode hat keinen Rückgabewert.

public final void setSourceSchema(Schema sourceSchema)

Graph sourceSchema	Das Schema, das als Quellschema verwendet werden soll

getTargetGraph

Gibt den erzeugten Zielgraphen zurück.

public final Graph getTargetGraph()

getTargetSchema

Gibt das erzeugte Zielschema zurück.

public final Schema getTargetSchema()

getTargetGraphClass

Gibt die erzeugte Zielgraphklasse zurück.

public final GraphClass getTargetGraphClass()

isExecutingGraphTransformation

Gibt true zurück, falls sich die Transformation gerade im zweiten Schritt (der Graphtransformation) befindet, false falls die Transformation sich gerade im ersten Schritt (der Schematransformation) befindet.

public final boolean isExecutingGraphTransformation()

isMiracleMode

Gibt true zurück, falls der vorausschauende Modus (MiracleMode) für die Erstellung der Attributwerte eingeschaltet ist, anderweitig false.

public boolean isMiracleMode()

loadSourceSchema

Lädt das Quellschema aus einer .tg-Datei. Die Methode hat keinen Rückgabewert.

public final void loadSourceSchema(String filename)

String filename	Name der .tg-Datei, aus der das Quellschema geladen werden soll

loadSourceGraph

Lädt den Quellgraphen aus einer angegebenen .tg-Datei. Das zu dem Graphen gehörende Quellschema wird automatisch aus der Datei mitgeladen. Die Methode hat keinen Rückgabewert.

public final void loadSourceGraph(String filename)

String filename	Name der .tg-Datei, aus der das Quellschema und der Quellgraph geladen werden soll

saveTargetSchema

Speichert das Zielschema in einer .tg-Datei. Die Methode hat keinen Rückgabewert.

public final void saveTargetSchema(String filename)

String filename	Name der .tg-Datei, in der das Zielschema gespeichert werden soll

saveTargetGraph

Speichert den Zielgraphen zusammen mit dem Zielschema in einer .tg-Datei. Die Methode hat keinen Rückgabewert.

public final void saveTargetGraph(String filename)

String filename	Name der .tg-Datei, in die der Zielgraph mit dem Zielschema gespeichert werden soll

D Beispiele für Transformationsobjekte

In diesem Kapitel sind verschiedene komplette Transformationen aufgeführt, die während der Implementierung entstanden sind. An Ihnen lässt sich die Funktionsweise der Transformationen sehr gut nachvollziehen, entsprechende Erläuterungen sind auch im Quelltext und in dem jedem Listing ergänzenden Beschreibungstext vorhanden.

D.1 Kopieren einer Knotenklasse

Das erste Beispiellisting zeigt das Kopieren einer Knotenklasse aus dem Quellschema in das Zielschema. Alle Instanten der Knotenklasse werden aus dem Quellgraphen in den Zielgraphen kopiert. Zusätzlich werden auch alle Generalisierungsbeziehungen sowie die Attribute kopiert.

Im Konstruktor der Transformationsklasse wird neben dem Kontext-Objekt, welches immer übergeben werden muss, der Name der zu kopierenden Knotenklasse als String übergeben. Der Konstuktor übergibt das Kontext-Objekt zur initialisierung an den Konstruktor der Oberklasse *Transformation*. Es ist fest voreingestellt (Zeile 19), dass immer sämtliche Attribute der Klasse mitkopiert werden sollen.

Innerhalb der Transformation (also der Methode `transformation()`) wird zuerst die Knotenklasse mit dem übergebenen Namen aus dem Quellschema abgefragt (Zeilen 23 und 24). In Zeile 25 wird überprüft, ob diese Knotenklasse als abstrakt definiert ist. In Zeile 26 wird der semantische Ausdruck für die Graphtransformation erzeugt, der alle Knoten dieser Knotenklasse (aber nicht deren Unterklassen) aus dem Quellgraphen abfragt. In Zeile 27 wird die die Knotenklasse mit den vorher bestimmten Parametern erzeugt.

In den Zeilen 28 bis 33 wird für jede direkte Oberklasse der zu kopierenden Klasse die Generalisierungsbeziehung auch im Zielschema angelegt. In den Zeilen 34 bis 37 werden die Attribute der Knotenklasse unter Verwendung der Transformation “CopyAttributes”, welche in Anhang D.3 beschrieben wird, kopiert.

Eine erweiterte Version dieser Klasse, bei der sich das Kopieren der Generalisierungsbeziehungen sowie der Attribute selektiv ein- bzw. ausschalten lässt, ist auf der dieser Arbeit beiliegenden CD zu finden.

```
1  package de.uni_koblenz.jgralab.gretl;
2
3  import de.uni_koblenz.jgralab.schema.AttributedElementClass;
4  import de.uni_koblenz.jgralab.schema.QualifiedName;
5  import de.uni_koblenz.jgralab.schema.VertexClass;
6
7  /**
8   *
9   * @author Oliver Weichert <weichert@uni-koblenz.de>, 2008, Diploma Thesis
10  */
11 public class CopyVertexClass extends Transformation {
12
13     protected String vertexName;
14     protected Boolean withAttributes = true;
15
16     public CopyVertexClass(TransformationContext context, String vertexName) {
17         super(context);
18         this.vertexName = vertexName;
19         withAttributes = true;
20     }
21
22     public void transform() throws Exception {
23         AttributedElementClass element = context.getSourceSchema().
24             getAttributedElementClass(new QualifiedName(vertexName));
25         Boolean isAbstract = element.isAbstract();
26         String query = "V{"+vertexName+"!}";
27         createVertexClass(vertexName, isAbstract, query);
28         for (AttributedElementClass superclass:element.getDirectSuperClasses()) {
29             addSuperClass((VertexClass) context.getTargetSchema().
30                 getAttributedElementClass(new QualifiedName(vertexName)), (VertexClass)
31                 context.getTargetSchema().getAttributedElementClass(new
32                 QualifiedName(superclass.getQualifiedName())));
33         }
34         if (withAttributes) {
35             CopyAttributes ca = new CopyAttributes(context, vertexName);
36             ca.execute();
37         }
38     }
39 }
```

D.2 Kopieren einer Kantenklasse

Dieses Beispiellisting zeigt das Kopieren einer Knotenklasse aus dem Quellschema in das Zielschema. Alle zugehörigen Instanzen werden ebenfalls aus dem Quellgraphen in den Zielgraphen kopiert. Die Generalisierungsbeziehungen sowie die Attribute werden wie schon im vorherigen Beispiel kopiert. Die Transformation geht davon aus, dass die beiden Knotenklasse an ihrem Anfang sowie Ende schon in das Zielschema kopiert worden sind.

Der Konstruktor ist analog zum Beispiel aus Anhang D.1 aufgebaut, hier wird zusätzlich zum obligatorischen Kontext-Objekt noch der Name der zu kopierenden Kantenklasse als String übergeben.

In der `transformation()`-Methode wird zuerst in den Zeilen 26 bis 31 aus dem Quellschema die Kantenklasse aus dem Quellschema abgefragt, bestimmt ob die Kantenklasse als abstrakt definiert ist sowie die Anfangs- und Endknotenklasse bestimmt. In den Zeilen 33 bis 48 erfolgt der eigentliche Kopiervorgang mit den entsprechenden Parametern. Dazu werden zusätzlich die jeweiligen Anfangs- und Endknotenklassen im Zielschema bestimmt, sowie die Attribute “min”, “max” und der Rollenname am jeweiligen Kantenende. In den Zielen 45 bis 47 werden die semantischen Ausdrücke für das Kopieren der Instanzen erzeugt. Der erste Ausdruck bestimmt alle Kanten vom Typ der Kantenklasse, die kopiert werden soll. Der zweite und dritte Ausdruck erzeugen die für Kanten benötigten Abbildungen von der Urbildmenge auf die jeweiligen Start- bzw. Endknoten der Urbildmenge.

Die Generalisierungsbeziehungen werden in den Zeilen 49 bis 53 analog zum Beispiel in Anhang D.1 erzeugt, ebenso die Attribute in den Zeilen 55 bis 58.

Eine erweiterte Version dieser Klasse, bei der sich das Kopieren der Generalisierungsbeziehungen sowie der Attribute selektiv ein- bzw. ausschalten lässt, ist auf der dieser Arbeit beiliegenden CD zu finden.

```
1  package de.uni_koblenz.jgralab.gretl;
2
3  import de.uni_koblenz.jgralab.schema.AttributedElementClass;
4  import de.uni_koblenz.jgralab.schema.QualifiedName;
5  import de.uni_koblenz.jgralab.schema.EdgeClass;
6  import de.uni_koblenz.jgralab.schema.VertexClass;
7
8  /**
9   * Copies an edge to the target schema
10  * copies also the needed vertices, if they dont already exist in the target schema
11  *
12  * @author Oliver Weichert <weichert@uni-koblenz.de>, 2008, Diploma Thesis
13  */
14 public class CopyEdgeClass extends Transformation {
15
16     protected String edgeName;
17     protected Boolean withAttributes;
18
19     public CopyEdgeClass(TransformationContext context, String edgeName) {
20         super(context);
21         this.edgeName = edgeName;
22         withAttributes = true;
23     }
24
25     public void transform() throws Exception {
26         Boolean isAbstract = context.getSourceSchema().
27             getAttributedElementClass(new QualifiedName(edgeName)).isAbstract();
28         EdgeClass sourceEdgeClass = (EdgeClass) context.getSourceSchema().
29             getAttributedElementClass(new QualifiedName(edgeName));
```

```
30          VertexClass edgeAlpha = sourceEdgeClass.getFrom();
31          VertexClass edgeOmega = sourceEdgeClass.getTo();
32          // Edge kopieren...
33          createEdgeClass(edgeName,
34              (VertexClass) context.getTargetSchema().
35                  getAttributedElementClass(edgeAlpha.getQName()),
36              sourceEdgeClass.getFromMin(),
37              sourceEdgeClass.getFromMax(),
38              sourceEdgeClass.getFromRolename(),
39              (VertexClass) context.getTargetSchema().
40                  getAttributedElementClass(edgeOmega.getQName()),
41              sourceEdgeClass.getToMin(),
42              sourceEdgeClass.getToMax(),
43              sourceEdgeClass.getToRolename(),
44              isAbstract,
45              "E{" + edgeName + "!}",
46              "from e:E{" + edgeName + "} reportMap e, startVertex(e) end",
47              "from e:E{" + edgeName + "} reportMap e, endVertex(e) end"
48              );
49          for (AttributedElementClass superclass:sourceEdgeClass.getDirectSuperClasses()) {
50              addSuperClass((EdgeClass) context.getTargetSchema().getAttributedElementClass(
51                  new QualifiedName(edgeName)), (EdgeClass) context.getTargetSchema().
52                  getAttributedElementClass(new QualifiedName(superclass.getQualifiedName())));
53          }
54          // Attribute kopieren
55          if (withAttributes) {
56              Transformation copyAttributes = new CopyAttributes(context, edgeName);
57              copyAttributes.execute();
58          }
59      }
60
61  }
```

D.3 Kopieren eines Attributes

Das in diesem Abschnitt aufgeführte Listing kopiert ein einzelnen Attribut eines Schemaelementes aus dem Quellschema in das Zielschema. Es wird dabei davon ausgegangen, dass das Schemaelement im Quell- sowie Zielschema den gleichen Namen besitzt, sowie dass der Wertebereich im Zielschema bereits existiert.

Der Konstruktor der Transformation erwartet neben dem Kontext-Objekt zwei weitere Parameter, nämlich das Element, dessen Attribut kopiert werden soll, als AttributedElementClass sowie den Namen des zu kopierenden Attributes als String.

Die Transformation prüft zuerst in den Zeilen 29 und 30, ob der vorausschauende Modus aktiviert wurde. Dieser wird benötigt, da die Transformation mit den beiden vorher beschriebenen Transformationen zusammenarbeiten soll, die das “ergänzen” fehlender Attributwerte wie es in Kapitel 6 beschrieben wurde, nicht beherrschen. Im Anschluss daran wird in Zeile 31 das zu kopierende Attribute aus dem Quellschema sowie in den

Zeilen 32 bis 34 das Element, welches das Attribut im Zielschema erhalten soll, abgefragt. In Zeile 35 wird eine Fallunterscheidung vorgenommen, da sich das Erzeugen des semantischen Ausdruckes zur Abfrage der Werte in Abhangigkeit davon, ob es sich um eine Knoten- oder Kantenklasse handelt, ändert. In den Zeilen 36 bis 39 bzw. 41 bis 45 wird das Attribut im Zielschema angelegt. Der Semantische Ausdruck bestimmt jeweils eine Abbildung der Elemente der Urbildmenge auf die jeweiligen Attributwerte.

```
1  package de.uni_koblenz.jgralab.gretl;
2
3  import de.uni_koblenz.jgralab.Attribute;
4  import de.uni_koblenz.jgralab.schema.AttributedElementClass;
5  import de.uni_koblenz.jgralab.schema.QualifiedName;
6  import de.uni_koblenz.jgralab.schema.VertexClass;
7
8  /**
9  * copies a single attribute
10 * Attribute has to be the same in the source and target schema!
11 *
12 * the element the attribute belongs to is automatically determined
13 *
14 * @author Oliver Weichert <weichert@uni-koblenz.de>, 2008, Diploma Thesis
15 */
16 public class CopyAttribute extends Transformation {
17
18     protected String attributeName;
19     protected AttributedElementClass sourceElement;
20
21     public CopyAttribute(TransformationContext context, AttributedElementClass
22         sourceElement, String attributeName) {
23         super(context);
24         this.sourceElement = sourceElement;
25         this.attributeName = attributeName;
26     }
27
28     public void transform() throws Exception {
29         if (!context.isMiracleMode()) throw new Exception("Fuer diese Transformation
30             wird der vorausschauende Modus benoetigt!");
31         Attribute sourceAttribute = sourceElement.getAttribute(attributeName);
32         AttributedElementClass target = context.getTargetSchema().
33             getAttributedElementClass(new QualifiedName(
34                 sourceElement.getQualifiedName()));
35         if (target instanceof VertexClass) {
36             createAttribute(attributeName, target,
37                 sourceAttribute.getDomain().getQualifiedName(),
38                  "from v:V{" + sourceElement.getQualifiedName() +
39                  "!} reportMap v, v." + attributeName + " end");
40         } else {
41             createAttribute(attributeName, target,
42                 sourceAttribute.getDomain().getQualifiedName(),
43                 "from e:E{" + sourceElement.getQualifiedName() + "!} reportMap e, e." +
44                 attributeName + " end");
45         }
46     }
47 }
```

D.4 Kopieren aller Attribute einer Knoten- oder Kantenklasse

Das folgende Listing kopiert alle Attribute einer Knoten- oder Kantenklasse aus dem Quellschema in das Zielschema. Dabei wird davon ausgegangen, dass die jeweilige Knoten- und Kantenklasse samt ihrer Instanzen schon kopiert worden sind und sowohl im Quell- als auch im Zielschema den selben Namen besitzen.

Der Konstruktor erwartet außer dem Kontext-Objekt den Namen der Knoten- oder Kantenklasse, deren Attribute kopiert werden sollen, als String. In diesem Beispiel ist voreingestellt, dass nur die eigenen Attribute kopiert werden und nicht zusätzlich diejenigen, die von Oberklassen geerbt worden sind. Dieses Verhalten lässt sich durch den Aufruf von `setCopyInheritedAttributes` umschalten.

Innerhalb der Transformation wird in Zeile 28 das Schemaelement, dessen Attribute kopiert werden sollen, aus dem Quellschema abgefragt. Es folgt eine Fallunterscheidung in Zeile 30, je nachdem ob auch die geerbten Attribute kopiert werden sollen oder nicht. Die jeweiligen Attribute werden in den Zeilen 31 bzw. 37 in einer Schleife durchlaufen und mit Hilfe der in Anhang D.3 definierten Transformation “copyAttribute” kopiert.

```
1  package de.uni_koblenz.jgralab.gretl;
2
3  import de.uni_koblenz.jgralab.Attribute;
4  import de.uni_koblenz.jgralab.schema.AttributedElementClass;
5  import de.uni_koblenz.jgralab.schema.QualifiedName;
6
7  /**
8  * copies all attributes of an element
9  *
10 * @author Oliver Weichert <weichert@uni-koblenz.de>, 2008, Diploma Thesis
11 */
12 public class CopyAttributes extends Transformation {
13
14     protected String sourceElementName;
15     protected Boolean copyInheritedAttributes;
16
17     public CopyAttributes(TransformationContext context, String sourceElementName) {
18         super(context);
19         this.sourceElementName = sourceElementName;
20         copyInheritedAttributes = false;
21     }
22
23     public void setCopyInheritedAttributes(Boolean copy) {
24         copyInheritedAttributes = copy;
25     }
26
27     public void transform() throws Exception {
28         AttributedElementClass ae = context.getSourceSchema().getAttributedElementClass(
29             new QualifiedName(sourceElementName));
30         if (copyInheritedAttributes) {
31             for (Attribute attr : ae.getAttributeList()) {
32                 CopyAttribute copyAttribute = new CopyAttribute(context, ae,
```

```
33                      attr.getName());
34                  copyAttribute.execute();
35              }
36          } else {
37              for (Attribute attr : ae.getOwnAttributeList()) {
38                  CopyAttribute copyAttribute = new CopyAttribute(context, ae,
39                      attr.getName());
40                  copyAttribute.execute();
41              }
42          }
43      }
44  }
```

D.5 Kopieren eines kompletten Schemas

Die in diesem Abschnitt aufgeführte Transformation kopiert ein komplettes Quellschema in das Zielschema. Aus diesem Grund erwartet der Konstruktor in den Zeilen 10 bis 12 ausschließlich das obligatorische Kontext-Objekt als Parameter. Die Transformation selbst ruft zuerst in den Zeilen 16 und 17 die Transformation “copyDomains” auf, welche alle im Quellschema definierten (komplexen) Domains in das Zielschema kopiert. Diese Transformation ist auf der beiliegenden CD zu finden und hier nicht gesondert aufgelistet. Im Anschluss daran werden in den Zeilen 19 und 20 alle vorhandenen Knotenklassen des Quellschemas bestimmt, die in den Zeilen 21 bis 27 in einer Schleife in das Zielschema kopiert werden. Dabei wird in Zeile 22 zusätzlich geprüft, ob es sich um eine JGraLab-eigene Knotenklasse handelt, welche nicht kopiert wird. In den Zeilen 23 bis 25 wird zum Kopieren die in Anhang D.1 definierte Transformation “copyVertexClass” für jede zu kopierende Knotenklasse aufgerufen. Das Kopieren der Kantenklassen in den Zeilen 29 bis 37 erfolgt analog zum Kopieren der Knotenklassen, weswegen eine weitere Erläuterung nicht notwendig ist.

```
1  package de.uni_koblenz.jgralab.gretl;
2
3  import java.util.List;
4
5  import de.uni_koblenz.jgralab.schema.EdgeClass;
6  import de.uni_koblenz.jgralab.schema.VertexClass;
7
8  public class CopySchema extends Transformation{
9
10     public CopySchema(TransformationContext context) {
11         super(context);
12     }
13
14     public void transform() throws Exception {
15         // Zuerst alle Domains kopieren...
16         Transformation copyDomains = new CopyDomains(context);
17         copyDomains.execute();
18         // dann alle vertices...
19         List<VertexClass> vertexList = context.getSourceSchema().
```

```
20              getVertexClassesInTopologicalOrder();
21          for (VertexClass v:vertexList) {
22              if (!v.isInternal()) {
23                  Transformation copyVertex = new CopyVertexClass(context,
24                      v.getQualifiedName());
25                  copyVertex.execute();
26              }
27          }
28          // dann alle edges...
29          List<EdgeClass> edgeList = context.getSourceSchema().
30              getEdgeClassesInTopologicalOrder();
31          for (EdgeClass e:edgeList) {
32              if (!e.isInternal()) {
33                  Transformation copyEdge = new CopyEdgeClass(context,
34                      e.getQualifiedName());
35                  copyEdge.execute();
36              }
37          }
38      }
39  }
```

Literaturverzeichnis

[Atk03] Atkinson, Colin; Kühne, Thomas: *Model-Driven-Development: A Metamodeling Foundation*. IEEE Software 20 (5), S. 36-41, 2001.

[Atl08a] Atlas group LINA & INRIA: ATL: ATL Starter's Guide version 0.1: Website: eclipse Foundation. http://www.eclipse.org/m2m/atl/doc. letzter Zugriff: 21.05.2008

[Atl08b] Atlas group LINA & INRIA: ATL: ATL Poster: Website: eclipse Foundation. http://www.eclipse.org/m2m/atl/doc. Letzter Zugriff: 21.05.2008

[Ebe05] Ebert, Jürgen; Winter, Andreas: *Using Metamodels in Service Interoperability*. Paper, Universität Koblenz-Landau, 2005.

[Ecl08a] Atlas group LINA & INRIA: *ATL: ATL Download Page*. Website: eclipse Foundation. http://www.eclipse.org/modeling/m2m/downloads/index.php?project=atl, letzter Zugriff: 21.05.2008.

[Gam04] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 31st Printing, 2004.

[Gmt08a] fehlt noch!

[Kah06] Kahle, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*. Diplomarbeit, Fachbereich Informatik, Institut für Softwaretechnik, Universität Koblenz-Landau, Juni 2006.

[Kal05] A. Kalnins, E. Clems, A. Sostaks: *Model Transformation Approach Based on MOLA*. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005). Montego Bay, Jamaica, October 2-7, 2005, S. 25.

[Mar06] Marchewka, Katrin: *Entwurf und Definition der Graphenanfragesprache GReQL2*. Diplomarbeit, Fachbereich Informatik, Institut für Softwaretechnik, Universität Koblenz-Landau, September 2006.

[Obj08a] Meta Object Facility (MOF) 2.0 Query/View/Transform Specification: Webseite: OMG. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2008, letzter Zugriff: 22.05.2008.

[Obj08b] Meta Object Facility (MOF) 2.0 Query/View/Transform Specification: Webseite: OMG. http://www.omg.org/cgi-bin/doc?ptc/2007-07-07, 2007, zuletzt besucht: 22.05.2008.

[1] [Red08] Requirements-Driven Software Development System. Webseite: http://www.redseeds.eu/. letzter Zugriff: 21.05.2008

[Rhe06] Rheindorf, Florian: *Herleitung eines operationalen Ansatzes zur Modelltransformation im Kontext modellgetriebener Softwareentwicklung*. Diplomarbeit, Fachbereich Informatik, Institut für Softwaretechnik, Universität Koblenz-Landau, 2006.

[Wik08b] Wikipedia: MOF QVT. Webseite: http://de.wikipedia.org/wiki/QVT. letzter Zugriff: 22.05.2008

Abbildungsverzeichnis

Tabellenverzeichnis