Konzeptuelle Modellierung mit UML und OWL – Untersuchung der Gemeinsamkeiten und Unterschiede mit Hilfe von Modelltransformationen

Dipl.-Inf. Jesper Zedlitz

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2013

1. Gutachter:	Prof. Dr.-Ing. Norbert Luttenberger Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Prof. Dr. Bernhard Thalheim Christian-Albrechts-Universität zu Kiel

Datum der mündlichen Prüfung: 26. Juni 2013

Zusammenfassung

Heute wird für konzeptuelle Modellierung sowohl die Unified Modeling Language (UML) als auch auch die OWL 2 Web Ontology Language (OWL 2) verwendet. Beide Sprachen entstammen verschiedenen Technologieräumen und setzen unterschiedliche Schwerpunkte. In dieser Arbeit wird untersucht, ob und wie sich konzeptuelle Modelle, die in der einen Sprache geschrieben sind, in konzeptuelle Modelle, die in der anderen Sprache geschrieben sind, überführen lassen. Dadurch würden für ein Modell Verfahren und Software-Werkzeuge beider Technologieräume verfügbar.

Für die automatische Transformation wurde – anders als bei bisherigen Arbeiten – eine Herangehensweise gewählt, die von konkreter Syntax bzw. XML-Serialisierung abstrahiert und auf Ebene der Metamodelle von UML und OWL arbeitet. So lässt sich unabhängig von einzelnen Beispielmodellen zeigen, welche Modellelemente transformiert werden können und welche nicht.

Für eine Vielzahl von Modellierungskonzepten wird eine formale Beschreibung gegeben und untersucht, wie sich das jeweilige Konzept mit UML bzw. OWL repräsentieren lässt. In den Fällen, in denen die Semantiken der Repräsentationen in beiden Sprachen hinreichend ähnlich sind, um – unter Beibehaltung der Semantik – eine Transformation durchführen zu können, werden die Transformationen ausführlich sowohl als Freitext als auch formal in Form deklarativer QVT-R-Transformationsregeln beschrieben. Da sich diese Regeln nur auf Elemente der Metamodelle beziehen, sind die Transformationen unabhängig von einzelnen Modellen (Instanzen dieser Metamodelle).

Dadurch, dass die Transformationen in beide Richtungen vollständig und formal in QVT-R aufgeschrieben vorliegen, lassen sich die Überlegungen nachvollziehen und bei Bedarf schnell an einzelnen Beispielen testen. In dieser vollständigen und formalen Beschreibung unterscheidet sich diese Arbeit von anderen Arbeiten, die solche Beschreibungen nicht oder nur in Fragmenten enthalten.

Während sich in UML beschriebene Datenmodelle bis auf wenige Einschränkungen (abstrakte Typen, bestimmte Arten der Generalisierung, Erweiterung durch Stereotypen) relativ gut mit Ontologien darstellen lassen, ist die Transformation allgemeiner OWL-Ontologien in UML-Datenmodelle nicht immer möglich. So stellt z.B. die automatische Klassifizierung von Objekten eine Schwierigkeit dar. Aber selbst in diesen Fällen ist oft eine Transformation möglich, wie bei Kardinalitätsbeschränkungen, die innerhalb von Vererbungsbeziehungen auftreten. Ein Anwendungsbeispiel aus der Praxis der Deutschen Zentralbibliothek für Wirtschaftswissenschaften (ZBW) zeigt, wie gut sich die Transformationsregeln auf konkrete Modelle anwenden lassen und so ohne große Verluste einen Wechsel von einem Technologieraum zum anderen möglich machen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Für die konzeptuelle Modellierung von Informationssystemen wird heutzutage oftmals die Unified Modeling Language (UML) eingesetzt und auch von internationalen Standards empfohlen.[1] Die gute Verständlichkeit der grafischen Syntax hilft, mittels UML aufgeschriebene Modelle schnell zu verstehen. Auch Nicht-Informatiker können UML-Diagramme meist nach kurzer Einarbeitung lesen. Die große Auswahl an Software-Werkzeugen zur Arbeit mit UML-Modellen macht UML ebenfalls zu einer guten Wahl bei der Modellierung und einem Quasi-Standard bei der Software-Entwicklung.

Konzeptuelle Modellierung lässt sich ebenfalls mit Ontologien betreiben. Eine Definition des Begriffs "Ontologie" macht den engen Zusammenhang von Ontologie und konzeptuellem Modell deutlich: "*[. . .] an ontology is a specification of an abstract data model (the domain conceptualization) that is independent of its particular form.*"[2] Eine weit verbreitete Sprache zur Definition von Ontologien ist die Web Ontology Language (OWL) mit ihrer aktuellen Version OWL 2. Aufgrund der Tatsache, dass OWL vollständig mit formaler Logik hinterlegt ist, können mit Hilfe eines Reasoners logische Schlüsse über Modelle gezogen werden. Dabei ist das Reasoning nicht nur für die Beantwortung komplexer Anfragen, sondern auch für das Finden von Inkonsistenzen, sowohl in den Daten als auch im konzeptuellen Modell selbst, hilfreich.

Wie man sieht, hat jede der beiden Sprachen – UML und OWL – ihre Vorteile in bestimmten Anwendungsgebieten. Um in der Praxis von den Vorteilen und Software-Werkzeugen beider Sprachen profitieren zu können, ist es meist notwendig, den Modellierungsprozess für jede Sprache zu wiederholen. Jede Änderung in einem Modell muss ebenfalls in dem anderen, in anderer Sprache geschriebenen, Modell nachvollzogen werden. Um dieses Verfahren zu vereinfachen, wäre es hilfreich, eine Transformation von einem Modell, das in der einen Sprache formuliert wurde, in ein Modell, das in der anderen Sprache formuliert wurde, zur Verfügung zu haben.

In dieser Arbeit wird daher untersucht, ob und wie sich konzeptuelle Modelle, die in der einen Sprache geschrieben sind, in konzeptuelle Modelle, die in der anderen Sprache geschrieben sind, überführen lassen.

Dass eine Transformation möglich erscheint, lässt sich bereits an der Beobachtung fest-

[1] ISO: Norm ISO/TS 19103:2005 Geographic information – Conceptual schema language, Genf 2005, S. V: "*This Technical Specification identifies the combination of the Unified Modeling Language (UML) static structure diagram with its associated Object Constraint Language (OCL) and a set of basic type definitions as the conceptual schema language for specification of geographic information.*"

[2] Gruber, T. R.: Ontology, in: Liu, L./Özsu, M. T. (Hrsg.): Encyclopedia of Database Systems, Berlin/Heidelberg 2009.

machen, dass bei genauerem Hinsehen ein – dem "Subjekt, Prädikat, Objekt"-Muster der "Ontologie-Sprachen" RDF, RDF(S), OWL und OWL 2 ähnliches – "object-property"-Muster auch in vielen mit UML geschriebenen konzeptuellen Modellen zu entdecken ist. Ein gutes Beispiel für die konsequente Nutzung dieses "object-property"-Musters ist die Geography Mark-Up Language (GML).[3]

Der Frage nach dem "ob und wie" einer Transformation folgend, müssen konsequenter Weise auch weitere Fragestellungen bearbeitet werden: Lässt sich all das, was mit einem UML-Klassenmodell ausgedrückt werden kann, auch in OWL darstellen? Sollte dies nicht der Fall sein, muss zumindest Klarheit darüber bestehen, was sich nicht mit OWL ausdrücken lässt. Ebenso verhält es sich für den umgekehrten Fall: Gib es Sachverhalte, die sich mit OWL beschreiben lassen, die jedoch nicht mit UML-Klassendiagrammen darstellbar sind?

Statt eine Menge von Beispielen zu betrachten bzw. zu transformieren – was problematisch ist, da man nie genau weiß, ob mit den gewählten Beispielen alle relevanten Fälle abgedeckt sind – nähere ich mich der Beantwortung der Fragen, indem ich eine automatische Transformation von UML-Klassenmodellen und OWL-Modellen (bzw. -Ontologien) auf Basis der jeweiligen Metamodelle entwickle. Durch diesen systematischen Ansatz wird unabhängig von Beispielmodellen gezeigt, welche Modellelemente transformiert werden können und welche nicht.

Während sich in UML beschriebene Datenmodelle bis auf wenige Einschränkungen (u.a. abstrakte Typen, bestimmte Arten der Generalisierung, Erweiterung durch Stereotypen) relativ gut mit Ontologien darstellen lassen, ist die Transformation allgemeiner OWL-Ontologien in UML-Datenmodelle nicht immer möglich. So stellt z.B. die automatische Klassifizierung von Objekten eine Schwierigkeit dar. Aber selbst in diesen Fällen ist oft eine Transformation möglich, wie bei Kardinalitätsbeschränkungen, die innerhalb von Vererbungsbeziehungen auftreten. Ein Anwendungsbeispiel aus der Praxis der Deutschen Zentralbibliothek für Wirtschaftswissenschaften (ZBW) zeigt, wie gut sich die Transformationsregeln auf konkrete Modelle anwenden lassen und so ohne große Verluste einen Wechsel von einem Technologieraum zum anderen möglich machen.

Eine weitere Sprache zu Datenmodellierung, die sich besonders im Bereich der Kommunikation Maschine-zu-Maschine in Form XML-basierter Webservices zu einem de-facto Standard entwickelt hat, ist die XML Schema Definition Language (XSD). Für die Transformationen UML → XSD und XSD → UML kann jedoch auf die Spezifikation der Geography Mark-Up Language (GML) zurückgegriffen werden, die eine Beschreibung dieser Transformationen enthält. Der Vollständigkeit halber wird zu jedem in der Arbeit betrachteten Modellierungskonzept eine Ausdeutung für XSD gegeben: Es wird gezeigt, wie sich das Konzept in XSD darstellen lässt, eine formale Betrachtung (z.B. auf Metamodellebene) findet jedoch nicht statt.

[3] OGC: Geography Markup Language (GML) Encoding Standard 3.2.1, 2007 ⟨URL: `http://www.opengeospatial.org/standards/gml`⟩, S. 21: "*This encoding pattern is sometimes referred to as the object-property model and has been the basis of the GML encoding model [...]*"

Aufbau der Arbeit

Der restliche Teil der Arbeit ist folgendermaßen gegliedert: Zunächst werden in Kapitel 2 wichtige Begriffe erläutert, die für das Verständnis der nachfolgenden Kapitel notwendig sind. Da einige Begriffe mehrdeutig sind, wird an dieser Stelle auch die in dieser Arbeit verwendete Bedeutung festgelegt. Es folgt in Kapitel 3 die Vorstellung der Sprachen UML, OWL und QVT, bei der speziell auf Details eingegangen wird, die für die in dieser Arbeit behandelte Fragestellung relevant sind. Die grundlegende Idee der Transformation wird in Kapitel 4 vorgestellt und mit anderen existierenden Ansätzen verglichen. Hier findet sich auch eine Zusammenfassung zu "Metamodellierung". Kapitel 5 gibt einen Überblick über bisher veröffentlichte Arbeiten zum Verhältnis von UML und OWL sowie der Transformation zwischen beiden Sprachen. Grundlegende Unterschiede zwischen UML und OWL, die unmittelbar Auswirkung auf die Transformationen haben und sie in einigen Fällen unmöglich machen, werden in Kapitel 6 beleuchtet. Der eigentliche Kern der Arbeit findet sich in den Kapiteln 7 bis 11, wo detailliert auf die einzelnen Bestandteile statischer Modelle eingegangen wird und die Transformationen beschrieben werden. Es schließt sich in Kapitel 12 eine Bewertung der entwickelten Regeln sowie der durch die Transformation entstehenden Ontologien an. In Kapitel 13 schließlich wird ein Anwendungsfall im Zusammenhang mit der "Digitalen Reichsstatistik" der Deutschen Zentralbibliothek für Wirtschaftswissenschaften (ZBW) vorgestellt: die Modellierung historischer Importdaten mit UML und OWL. Kapitel 14 fasst die Ergebnisse der Arbeit zusammen und zeigt mögliche weitere Forschungsfragen auf.

Kapitel 2

Begriffe

In diesem Kapitel werden einige Begriffe aus den Bereichen Modellierung und Transformation erläutert. Dabei handelt es sich um Begriffe, die sowohl in der Literatur als auch in der Praxis oft verwendet werden – teilweise jedoch mit unterschiedlicher Bedeutung. Daher soll mit den Ausführungen in diesem Kapitel eine einheitliche Grundlage für den Rest der Arbeit geschaffen werden.

Gegliedert ist das Kapitel in vier Abschnitte: Zunächst werden Begriffe aus dem Bereich der Modellierung erläutert. Es folgt ein Abschnitt über zwei in der Modellierung auftretende Beziehungen, *IsA* und *InstanceOf*, deren Verwechslung oft zu Missverständnissen führt. Der dritte Abschnitt beschäftigt sich mit Begriffen aus dem Bereich Transformation. Im letzten Abschnitt wird der Begriff *Technologieraum* eingeführt und drei für diese Arbeit relevante Technologieräume erläutert.

2.1 Modellierung

2.1.1 Der Begriff "Modell"

Hesse und Mayr nennen drei bereits 1973 von H. Stachowiak aufgestellte Merkmale von Modellen:[4]

1. Abbildungsmerkmal – Jedes Modell steht für ein Original, wobei als "Original" nicht nur Gegenstände, sondern auch nicht-physische Dinge wie Zusammenhänge, Ideen oder Pläne dienen können. Auch ein Modell kann wiederum als Original für ein weiteres Modell dienen. Der Zusammenhang zwischen Original und Modell ist stets abhängig vom Nutzer bzw. Modellierer, und es kann für ein Original eine Vielzahl von Modellen geben. Ebenso kann ein Modell für eine Menge von Originalen stehen.

2. Reduktionsmerkmal – Da ein Modell nur einen Ausschnitt der Realwelt darstellt, weist es nicht alle Eigenschaften des Originals auf. Fast immer findet eine Reduktion statt, bei der Eigenschaften entfallen (präterierte Eigenschaften). Es ist jedoch auch möglich, dass bei der Modellierung weitere Eigenschaften hinzukommen (abundante Eigenschaften).

[4] Vgl. Hesse, W./Mayr, H.C.: Modellierung in der Softwaretechnik: eine Bestandsaufnahme, in: Informatik-Spektrum 31/5, Berlin/Heidelberg 2008, Seite 380 f.

3. Pragmatisches Merkmal – Eine Modellierung findet zweckgebunden statt und soll somit nur unter bestimmten Bedingungen und bezüglich spezieller Fragestellungen das Original ersetzen.

Modelle lassen sich anhand ihrer Eigenschaften in verschiedene Gruppe einteilen. Eine mögliche Klassifikation ist die in *statische Modelle* auf der einen und *dynamische Modelle* auf der anderen Seite.[5]

1. Bei statischen Modellen werden Zusammenhänge von Dingen und ihren Beziehungen betrachtet, wie sie zu einem bestimmten Zeitpunkt gelten. Hesse und Mayr führen einige Unterkategorien statischer Modelle auf: Gegenstands-, Struktur-, Entitäts- oder Klassenmodelle. Es bleibt aber "*auch bei den Klassenmodellen – wie sie z.B. bei UML gängig sind – die statische Betrachtungsweise bestimmend.*"[6]

2. Dynamische Modelle geben hingegen Auskunft über das Verhalten eines Systems und dessen Zustandsänderungen.

Üblicherweise werden auch die in einem statischen Modell beschriebenen Elemente irgendwann einer Änderung unterworfen sein, jedoch steht dies nicht im Fokus eines statischen Modells – vergleiche "Reduktionsmerkmal" oben.

Da bei der konzeptuellen Modellierung die statische Betrachtungsweise bestimmend ist und somit das Verhalten von Objekten üblicherweise keine Rolle spielt, **werden im Rahmen dieser Arbeit nur statische Modelle betrachtet**, Operationen bzw. Methoden also nicht berücksichtigt.

2.1.2 Maschineninterpretierbarkeit

In der Informatik versteht man unter *Maschineninterpretierbarkeit* allgemein, dass ein Text mit Hilfe eines Algorithmus gelesen und interpretiert werden kann. Ein solcher Text dient – im Gegensatz zu nicht-maschineninterpretierbaren Texten, die für die Kommunikation zwischen Menschen eingesetzt werden können – im weitesten Sinne dazu, das Verhalten eines Laufzeitsystems zu steuern. Damit dies möglich ist, muss ein maschineninterpretierbarer Text einer formalen Syntax und Struktur entsprechen. So ist gewährleistet, dass für den verarbeitenden Algorithmus keine Widersprüche oder Entscheidungsmöglichkeiten existieren.

Dass eine bestimmte Software ein Modell einlesen kann, ist für eine Maschineninterpretierbarkeit noch nicht ausreichend – wichtig ist die Beeinflussung des Laufzeitverhaltens.

2.1.3 Konzeptuelles Modell

Wie bereits beschrieben, beschränkt sich ein Modell auf einen Ausschnitt der Realwelt und wendet eine – in der Regel zweckgebundene – Sicht auf diesen Ausschnitt an. Bei einem

[5] Vgl. Hesse/Mayr: Modellierung, S. 382.
[6] Vgl. a. a. O.

konzeptuellen Modell wird eine Sicht gewählt, die die Realwelt als Menge von Objekten, Beziehungen zwischen diesen Objekten sowie *Konzepten* betrachtet.[7]

Ein *Konzept* ist dabei eine gedankliche Abstraktion, bei der man die für einen Anwendungszweck relevanten, gemeinsamen Eigenschaften einer Gruppe von Objekten bzw. Beziehungen identifiziert. Jedes Konzept besitzt eine Extension und eine Intension: Die Menge der möglichen Objekte bzw. Beziehungen, die zu diesem Konzept gehören können, bildet die Extension des Konzepts; die Intension des Konzepts besteht aus den gemeinsamen Eigenschaften aller zum Konzept gehörenden Objekte bzw. Beziehungen.[8]

Oft spricht man bei konzeptuellen Modellen auch von semantischen Modellen: "*Modelle [...] werden vielmehr immer dann ‚semantisch' genannt, wenn sie – wie die meisten konzeptuellen Modelle – möglichst viele innere Bezüge und Abhängigkeiten explizit machen [...].*"[9]

Es ist durchaus üblich, dass es für denselben Realweltabschnitt verschiedene konzeptuelle Modelle gibt. Diese können sich sowohl in der Menge der für die Fragestellung relevanten Informationen als auch in ihrer Struktur unterscheiden.

2.1.4 Konzeptuelles Schema

Wird für die Niederschrift eines konzeptuellen Modells eine formale Sprache verwendet, so wird von einem *konzeptuellen Schema* (engl. conceptual schema) gesprochen.[10] Ein anderer Begriff für die formale Notation eines konzeptuellen Modells ist *metaschema*.[11]

2.1.5 Taxonomie, Ontologie

Eine *Taxonomie* beschreibt den Zusammenhang zwischen Elementtypen (→ 7 Elementtypen) untereinander, der über *IsA*-Beziehungen (→ 2.2 IsA und InstanceOf) gebildet wird. Elementtypen und *IsA*-Beziehungen bilden eine Netzstruktur, die Taxonomie genannt wird.

Für den Begriff *Ontologie* existieren verschiedene Definitionen. Eine oft zitierte stammt von Gruber: "*[...] an ontology is a specification of an abstract data model (the domain conceptualization) that is independent of its particular form.*"[12] In der Regel beinhaltet einen Ontologie auch eine Taxonomie der in ihr auftretenden Elementtypen.

Wie Atkinson et al. ausführen, handelt es sich bei jeder Ontologie auch um ein Modell. Die umgekehrte Inklusion, dass ein Modell auch eine Ontologie darstelle, stimme oft – jedoch nicht immer.[13]

[7] Vgl. Olivé, A.: Conceptual Modeling of Information Systems, Berlin/Heidelberg/New York 2007, S. 11.
[8] Vgl. a. a. O., S. 12.
[9] Hesse/Mayr: Modellierung, S. 385.
[10] Vgl. ISO: Norm ISO 19101 Geographic information – Reference model, Genf 2002.
[11] Vgl. Olivé: Conceptual Modeling, S. 406.
[12] Gruber: Ontology.
[13] Vgl. Atkinson, C./Gutheil, M./Kiko, K.: On the relationship of ontologies and models, in: Proceedings of the 2nd Workshop on Meta-Modelling and Ontologies (WoMM06), Bonn 2006 ⟨URL: http://subs.emis.de/LNI/Proceedings/Proceedings96/GI-Proceedings-96-3.pdf⟩.

2.1.6 Begriffe "konzeptuell" und "konzeptionell"

In Arbeiten zu konzeptuellen Modellen findet sich gelegentlich auch der Begriff "konzeptionelles Modell". Dies ist jedoch ein meist falsch verwendetes Synonym, wie Hesse und Mayr anmerken:[14] Zwar leiten sich beide Adjektive vom selben englischen Wort "concept" ab, haben jedoch eine unterschiedliche Bedeutung:

- Ein konzeptuelles Modell ist "*ein Konzept aufweisendes*" Modell.
- Ein konzeptionelles Modell dagegen ist ein "*ein Konzept betreffendes*" Modell.

2.1.7 Sprache

Modelle werden mit Hilfe von Sprachen[15] beschrieben. Sprachen lassen sich nach mehreren Gesichtspunkten klassifizieren. So kann z.B. zwischen formalen und informellen Sprachen unterschieden werden:

- formal – Ein wichtiges Merkmal einer formalen Sprache ist ihre Maschineninterpretierbarkeit. Eine formale Sprache unterliegt genauen Regeln, so dass sie eindeutig und widerspruchsfrei interpretiert werden kann. Beispiele formaler Sprachen sind Programmiersprachen und Sprachen zur Datenmodellierung.
- informell – Zu den informellen Sprachen zählen natürliche Sprachen und Plansprachen wie Deutsch, Englisch und Esperanto. Sie unterliegen zwar auch syntaktischen Regeln, besitzen eine Semantik und sind im Fall von elektronischen Texten maschinenlesbar. Eine Maschineninterpretierbarkeit ist jedoch nicht gegeben.

Auch bei der Darstellung lassen sich zwei Gruppen von Sprachen unterscheiden:

- textuell – Textuelle Sprachen bestehen aus Zeichen, die in einer sequenziellen Folge gemäß ihrer Syntax zu Wörtern und Sätzen aneinandergereiht werden.
- visuell – Visuelle Sprachen verwenden grafische Symbole, die gemäß ihrer Syntax zu komplexeren Grafiken zusammengesetzt werden.

Aufgrund dieser Klassifikationen ergeben sich folgende vier Kombinationsmöglichkeit, zu denen Beispiele genannt werden:

	formal	**informell**
textuell	XML, OWL 2	Deutsch, Englisch, Esperanto
visuell	UML, Nassi-Shneiderman-Diagramme	ägyptische Hieroglyphen

[14] Vgl. Hesse/Mayr: Modellierung.

[15] Im Gebiet der modellgetriebenen Softwareentwicklung wird statt "Sprache" oft den Begriff "Metamodell" verwendet.

Eine (formale) Sprache wird durch folgende drei Teile definiert, die in den nächsten Abschnitten genauer erläutert werden:

1. konkrete Syntax
2. abstrakte Syntax
3. statische Semantik

Auch für informelle Sprachen kann es diese drei Teile geben, doch sind sie für informelle Sprachen oft nicht vollständig definiert. Der Zusammenhang der Begriffe ist in Abbildung 2.1 dargestellt.

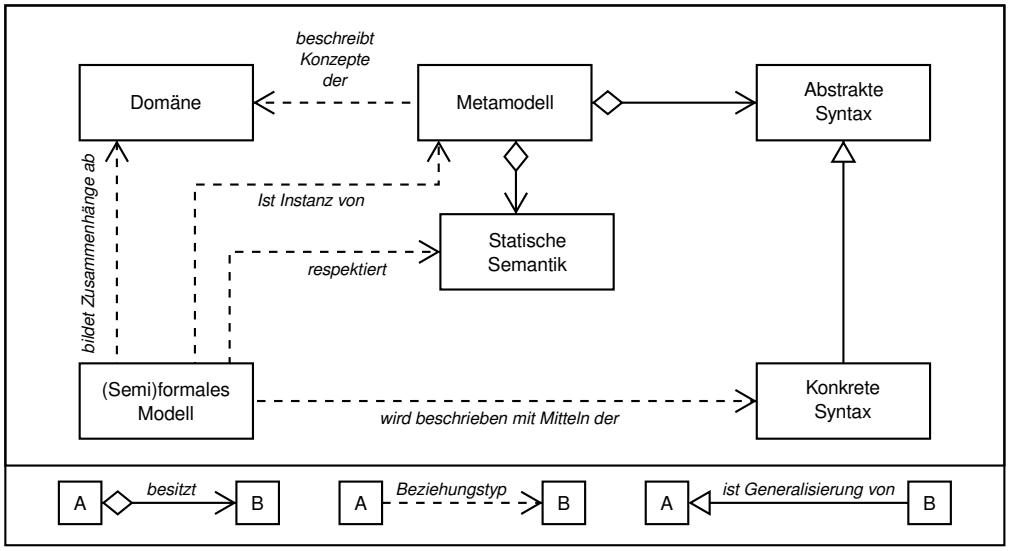


Abbildung 2.1. Zusammenhang der Begriffe "Modell", "Syntax" und "Semantik".[16]

2.1.8 Konkrete Syntax

Sowohl formale als auch informelle Sprachen haben eine konkrete Syntax. Sie legt Regeln fest, in welcher Weise bei einer menschenlesbaren Form Bestandteile der Sprache kombiniert werden dürfen.

▻ Bei **informellen, textuellen Sprachen** ist die Syntax eine Menge von Regeln, die beschreiben, wie aus Zeichen Wörter und aus diesen wiederum Sätze gebildet werden. Zunächst ist dabei die Bedeutung der gebildeten Sätze unwichtig.

[16] In Anlehnung an Pörnbacher, C.: Modellgetriebene Entwicklung der Steuerungssoftware automatisierter Fertigungssysteme, München 2011, S. 92.

- ▻ Bei **formalen, textuellen Sprachen** wird festgelegt, wie Modelle oder Programmtexte gebildet werden dürfen. Die Bedeutung der Modelle ist dabei wiederum zunächst unwichtig.[17]

- ▻ Auch **visuelle Sprachen** besitzen eine Syntax: Sie legt fest, welche Symbole es gibt und wie diese im Bild platziert werden dürfen.

Es ist durchaus üblich, dass sich die Syntaxen verschiedener Sprachen ähneln. Dies ist bei informellen Sprachen wie Deutsch oder Englisch ebenso der Fall wie bei formalen Sprachen, wo viele Programmiersprachen wie Java oder C# eine ähnliche Syntax verwenden. Es ist ebenfalls möglich, für die gleichen Informationen unterschiedliche Syntaxen zu definieren, wie es bei OWL 2 mit der Vielzahl von Syntaxen zum Aufschreiben von Ontologien der Fall ist (→ 3.2 OWL 2 Web Ontology Language (OWL 2)).

2.1.9 Abstrakte Syntax

Neben der konkreten Syntax, die menschenlesbar ist, gibt es (insbesondere bei formalen Sprachen) noch die abstrakte Syntax, unter der eine interne Darstellung der Strukturen, die in der konkreten Syntax definiert wurden, zu verstehen ist. Diese abstrahiert – wie es der Name schon sagt – von der konkreten Darstellung des Modells. Dabei werden alle für die Semantik einer Sprache nicht notwendigen Bestandteile (z.B. Leerzeichen, konkrete Schlüsselwörter) weggelassen.

2.1.10 Statische Semantik

Die Syntax einer Sprache gibt vor, welche elementaren Modellierungselemente es gibt und sie beschreibt, in welcher Weise Zeichen kombiniert werden dürfen, um gültige Wörter bzw. Modellierungselemente zu bilden. Im Gegensatz dazu beschreibt die statische Semantik, unter welchen Bedingungen es überhaupt sinnvoll ist, bestimmte durch die Sprache beschriebene Elemente miteinander zu verknüpfen.[18]

Angenommen, es gäbe (neben der Syntax) eine formale statische Semantik der deutschen Sprache, die die Aussagen “Eine Firma hat eine Person als Geschäftsführer.” und “Eine Firma ist keine Person.” enthalte. Dann wäre der Satz “Herr Meier ist Geschäftsführer von ACME.” sowohl syntaktisch als auch (statisch) semantisch korrekt. Der Satz “Herr Meier ist Geschäftsführer von Herrn Meier.” dagegen wäre zwar syntaktisch korrekt, im Sinne der genannten statischen Semantik jedoch nicht korrekt.

[17] Vgl. Tantau, Till: Syntax versus Semantik – Vorlesung zu Logik für Informatiker (WS 2006/07), Lübeck 2006.

[18] OMG: Unified Modeling Language, Infrastructure Version 2.4, 2011 ⟨URL: `http://www.omg.org/spec/UML/2.4/Infrastructure`⟩, S. 21: “*The static semantics of a language define how an instance of a construct should be connected to other instances to be meaningful, [...]*”

2.2 IsA und InstanceOf

Eine häufige Quelle von Verwirrung bei Modellierung ist der Unterschied zwischen *IsA* und *InstanceOf*.[19]

Mit *InstanceOf* wird die Beziehung eines Elements zu einem Elementtyp beschrieben. Wird ein Objekt also dadurch klassifiziert, indem beschrieben wird, dass es Instanz eines bestimmten Elementtyps ist, so wird eine solche *InstanceOf*-Beziehung hergestellt. Dagegen beschreibt eine *IsA*-Beziehung eine Generalisierung zwischen zwei Elementtypen.

Im üblichen Sprachgebrauch sind beide Beziehungen meist nicht zu unterscheiden. So heißt es sowohl "ACME ist eine Aktiengesellschaft." als auch "Eine Aktiengesellschaft ist eine juristische Person." Jedoch ist etwas formaler geschrieben mit "ist" im ersten Fall die Aussage

ACME *InstanceOf* Aktiengesellschaft

im zweiten Fall die Aussage

Aktiengesellschaft *IsA* juristische Person

gemeint.

Im Bereich der Metamodellierung wird diese Verwirrung zusätzlich verstärkt, da nun Elementtypen selbst wieder als Instanzen von (Meta-)Elementtypen auftreten. Hier kann mit "Eine Aktiengesellschaft ist eine Klasse." nun die formaler geschriebene Aussage

Aktiengesellschaft *InstanceOf* Klasse

gemeint sein.

Olivé nennt zwei Hinweise darauf, wie sich entscheiden lässt, welche Beziehung in einem konkreten Fall gemeint ist.[20] Ein wichtiges Merkmal ist die Tatsache, dass *IsA* transitiv ist, während *InstanceOf* es nicht ist. So lässt sich aus den Aussagen

Aktiengesellschaft *IsA* juristische Person
juristische Person *IsA* Person

folgern, dass auch gilt:

Aktiengesellschaft *IsA* Person

Bei *InstanceOf* ist dies nicht der Fall, aus den beiden obigen Aussagen

ACME *InstanceOf* Aktiengesellschaft
Aktiengesellschaft *InstanceOf* Klasse

lässt sich nicht folgern, dass "ACME" eine Klasse ist.

[19] Vgl. Olivé: Conceptual Modeling, S. 387.
[20] Vgl. a. a. O.

Ein weiteres Merkmal ist das Klassifikationslevel (→ 2.4.1 Model-Driven Architecture Technology Space) der beteiligten Elemente. Eine *IsA*-Beziehung kann nur zwischen Elementen mit demselben Klassifikationslevel bestehen. Außerdem muss das Klassifikationslevel größer als null sein.[21] Bei einer *InstanceOf*-Beziehung ist jedoch das Klassifikationslevel des auf der rechten Seite stehenden Elements um eins größer als das Klassifikationslevel des auf der linken Seite stehenden Elements.

2.3 Transformation

2.3.1 Begriff "Modelltransformation"

Mit Hilfe einer *Modelltransformation* kann aus einem vorhandenen Quellmodell ein neues Zielmodell (gesprochen wird dann von horizontaler oder Modell-zu-Modell-Transformation) oder andere Arten von Texten (gesprochen wird dann von vertikaler oder Modell-zu-Text-Transformation) erstellt werden. **Da für diese Arbeit nur Modell-zu-Modell-Transformationen von Interesse sind, wird auch nur auf diese eingegangen.** Quell- und Zielmodell können – müssen jedoch nicht – konform zum selben Metamodell sein. Eine Modelltransformation wird mit Hilfe von Transformationsregeln definiert, die sich auf die verwendeten Metamodelle beziehen. Die Definition der Transformationsregeln ist unabhängig von dem zu transformierenden Quellmodell, wobei dies jedoch konform zum Quell-Metamodell sein muss.

Die Transformation wird auf den Modellen selbst und nicht auf den Metamodellen ausgeführt. Ein Transformationswerkzeug liest das (zum Quell-Metamodell konforme) Quellmodell ein, führt die Transformationsregeln aus und erstellt so ein zum Ziel-Metamodell konformes Zielmodell.

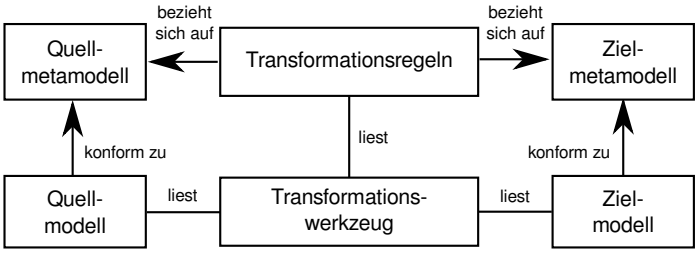


Abbildung 2.2. Modelltransformation – Übersicht

Im Gegensatz dazu findet bei einer *modellbasierten Transformation* keine Transformation von Modellen statt, sondern es wird auf der syntaktischen Ebene von Transferformaten gearbeitet und eine Formatumwandlung vorgenommen.[22]

[21] Vgl. Olivé: Conceptual Modeling, S. 388.

[22] Vgl. Eisenhut, C./Kutzner, T.: Vergleichende Untersuchungen zur Modellierung und Modelltransformation in der Region Bodensee im Kontext von INSPIRE, München 2010, S. 28.

2.3.2 Transformationssprache

Transformationsregeln beschreiben, wie Elemente eines konzeptuellen Schemas auf Elemente eines anderen Schemas abgebildet werden können. Um eine automatische Transformation zu ermöglichen, müssen Transformationsregeln maschinenlesbar und -interpretierbar notiert werden. Es ist also notwendig, sie in einer formalen Sprache zu notieren, einer *Transformationssprache*.

Es gibt eine große Vielzahl von Transformationssprachen; Czarnecki und Helsen etwa untersuchen 32 verschiedene Transformationssprachen.[23]

Eine prominente Rolle bei den Tranformationssprachen nimmt die als Sieger im Rahmen eines Wettbewerbs der Object Management Group (OMG) gewählte Transformationssprache Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) (→ 3.7 QVT) ein.

Zwei Arten von Transformationen sind zu unterscheiden: *unidirektionale* und *bidirektionale*. Bei unidirektionalen Transformationen ist eine Transformation nur in eine Richtung, von Quell- zu Zielmodell möglich. Ist mit derselben Transformation zusätzlich auch die umgekehrte Richtung von Ziel- zu Quellmodell möglich, wird dies als bidirektionale Transformation bezeichnet.

2.3.3 Bijektive Transformationen

Bidirektionale Modelltransformationen können bijektiv sein, müssen es aber nicht zwangsläufig. Eine Transformation zwischen zwei Metamodellen M und N, die durch die Relation R beschrieben wird, ist *bjektiv*, wenn für jedes Modell m, das zu M konform ist, genau ein Modell n exisitert, das zu N konform ist, sodass m und n in Relation R stehen und umgekehrt.[24]

2.4 Technologieraum

Der Begriff Technologieraum (engl. Technology Space (TS), manchmal auch "technological space") wurde von Kurtev et al. eingeführt, um Ansätze für komplexe Lösungen klassifizieren zu können.[25] Ein Technologieraum wird hier – etwas vereinfachend – als eine Menge von wissenbeschreibenden Modellierungssprachen und Transformationsmöglichkeiten verstanden. Im Folgenden werden die drei für diese Arbeit relevanten Technologieräume vorgestellt: der Model-Driven Architecture Technology Space (MDA TS), der Ontology Engineering Technology Space (Ontology TS) sowie der XML Technology Space (XML TS).

[23] Siehe Czarnecki, K./Helsen, S.: Feature-based Survey of Model Transformation Approaches, in: IBM Systems Journal 45/3, Riverton 2006 ⟨URL: http://gsd.uwaterloo.ca/sites/default/files/ibm06.pdf⟩.

[24] Vgl. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions, in: Engels, G. et al. (Hrsg.): Model Driven Engineering Languages and Systems. 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings. Berlin/Heidelberg 2007.

[25] Vgl. Kurtev, I./Bézivin, J./Aksit, M.: Technological spaces: An initial appraisal, in: International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, 30 Oct - 1 Nov 2002, Irvine, USA, 2002 ⟨URL: http://eprints.eemcs.utwente.nl/10206/01/0363TechnologicalSpaces.pdf⟩.

2.4.1 Model-Driven Architecture Technology Space

Bei der Model-Driven Architecture (MDA) wird eine geschichtete Architektur von Modellen verwendet, bei denen die Struktur eines Modells der einen Schicht durch ein Modell auf der nächsthöheren Schicht (Metamodell) beschrieben wird. Jedes Modell kann wiederum als Metamodell für eine weitere Schicht dienen. Dies lässt sich beliebig oft rekursiv fortsetzen. Mit Hilfe einer reflexiven Sprache (d.h. die Sprache wird verwendet, um sich selbst zu definieren) kann ein Ende dieser Modell-Metamodell-Schichten gebildet werden.[26] Um die Schichten trotz der wiederholt auftretenden Begriffe "Modell" und "Metamodell" identifizieren zu können, ist jeder Schicht ein *Klassifikationslevel* zugeordnet. Dabei ist das Klassifikationslevel eines Metamodells stets um eins größer als das seiner Instanzen.

Heutzutage steht der MDA TS eng in Verbindung mit den im Umfeld von UML entwickelten Technologien. Üblicherweise beschränkt sich die Metamodellierung in diesem Zusammenhang darauf, Modellierungs- sowie Programmiersprachen zu definieren.[27]

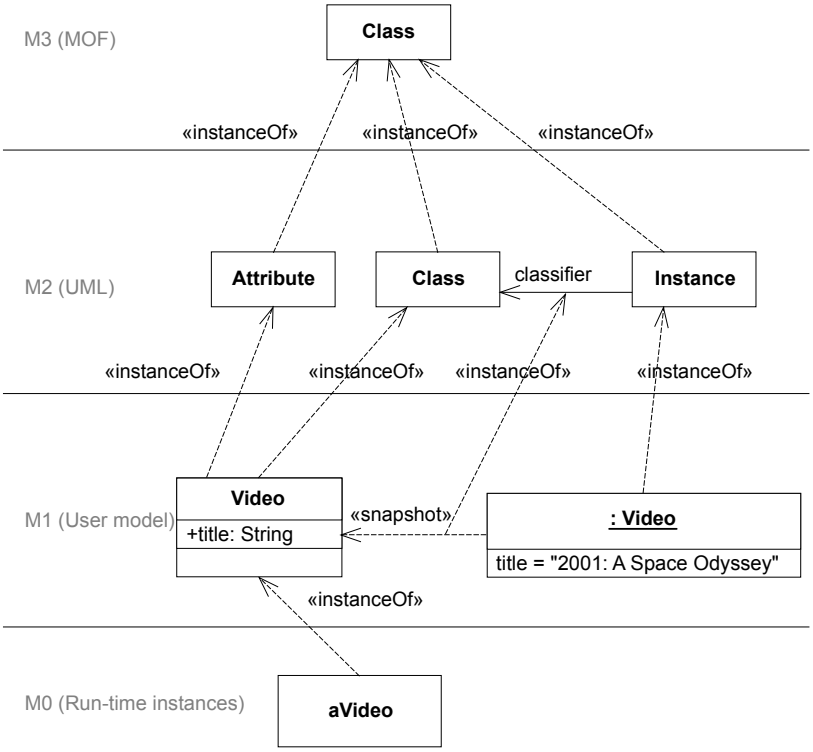


Abbildung 2.3. Ein Beispiel für eine vierschichtige Metamodell-Hierarchie ("An example of the four-layer metamodel hierarchy").[28]

[26] Vgl. OMG: UML Infrastructure, S. 17.

[27] Vgl. Parreiras, F.S./Staab, S./Winter, A.: On Marrying Ontological and Metamodeling Technical Spaces, in: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York 2007.

[28] OMG: UML Infrastructure, S. 20

Die UML-Spezifikation enthält die Beschreibung einer vierschichtigen Architektur:[29] Die oberste Schicht (M3) definiert das Meta-Object Facility (MOF) in reflexiver Weise. Dies wird als Meta-Metamodell bezeichnet.[30] Instanzen[31] des MOF werden für Sprachspezifikationen wie z.B. der UML oder dem OMG Common Warehouse Metamodel verwendet. Diese Sprachspezifikationen befinden sich auf der Schicht M2 und werden in dieser Architektur als Metamodelle bezeichnet. Auf der Schicht der Modelle (M1) befinden sich von Nutzern definierte Modelle, wie z.B. (UML-)Klassendiagramme (wie sie in dieser Arbeit betrachtet werden) oder (UML-)Zustandsautomaten. Auf der untersten Schicht M0 sind Laufzeit-Instanzen angesiedelt. Abbildung 2.3 zeigt ein Beispiel für diese vierschichtige Architektur.

Im MDA TS werden Modelle als Netzwerke von Objekten oder Graphen aufgefasst. Ihre abstrakte Syntax (→ 2.1.7 Sprache) wird üblicherweise mit (UML-)Klassendiagrammen beschrieben, die mit Hilfe einer Constraintsprache wie der Object Constraint Language (OCL) verfeinert werden können. Außerdem ist es bei UML möglich, die Semantik mittels so genannter UML Profile zu erweitern (→ 3.1.4 Semantikerweiterung durch Profile).

Zur Abfrage und Transformation von Modellen im MDA TS gibt es eine Reihe von Software-Werkzeugen, die auf den Objekt-Netzwerken oder Graphen arbeiten. Im Bereich von MOF spielt hier die von der OMG standardisierte QVT (→ 3.7 QVT) eine wichtige Rolle, bei der die Ansätze verschiedener Transformationssprachen zusammengeflossen sind.[32]

2.4.2 Ontology Engineering Technology Space

Ontologien haben ihre Wurzeln im Bereich der Wissensrepräsentation und Reasoning. In ihnen wird formal der Zusammenhang von Elementtypen und Beziehungstypen beschrieben. Ähnliches wird auch bei konzeptuellen Modellen der Software- und Datenmodellierung betrieben, jedoch werden Ontologien stets anders verwendet als konzeptuelle Modelle im Software- und Datenengineering.[33]

Im Bereich der auf Beschreibungslogiken (engl. Description Logics (DL)) basierenden Ontologien liegt der Schwerpunkt darauf, Zusammenhänge (insbesondere die von Elementtypen) so aufzuschreiben, dass mit möglichst wenig Angaben eine genaue Einordnung der Typen möglich ist.[34,35] Beschreibungslogiken ermöglichen es, Zusammenhänge aus formal aufgeschriebenen Axiomen abzuleiten. Bei einem solchen formalen Axiom kann es sich z.B. um die notwendigen und hinreichenden Bedingungen handeln, unter denen ein Objekt Instanz eines Elementtyps ist. Es ist somit – anders als im MDA TS – nicht unbedingt notwendig, für jedes

[29] Vgl. OMG: UML Infrastructure, S. 17 ff..

[30] Vgl. a. a. O., S. 17.

[31] Vgl. Atkinson, C./Kuhne, T.: Model-driven development: A Metamodeling Foundation, in: IEEE Software 20/5, Los Alamitos 2003.

[32] Einen breiten Überblick über Transformationssprachen bietet Czarnecki/Helsen: Feature-based Survey.

[33] Vgl. Parreiras/Staab/Winter: On Marrying Ontological and Metamodeling Technical Spaces, Abschnitt 2.1.

[34] Vgl. Baader, F. et al. (Hrsg.): The Description Logic Handbook: Theory, Implementation and Applications, Cambridge 2003.

[35] Vgl. Rector, A. et al.: OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns, in: Carbonell, J. G./Siekmann, J. (Hrsg.): Engineering Knowledge in the Age of the Semantic Web, Berlin/Heidelberg 2004.

Objekt *InstanceOf*-Beziehungen anzugeben. Bei dieser Art von Ontologien kommt meist die bei DL und First-Order-Logic gebräuchliche Open World Assumption (OWA) zum Einsatz.

Dagegen liegt bei den auf Logikprogrammierung basierenden Ontologien[36] der Schwerpunkt auf der formalen, integrierten Betrachtung ausdrucksstarker Klassen- und Regeldefinitionen. Die zugehörigen Ontologie-Sprachen sind üblicherweise turingvollständig, folgen aber der Closed World Assumption (CWA).[37]

Eine gewisse Sonderrolle im Ontology TS vertritt das Resource Description Framework (RDF) und die dazugehörige Schemasprache RDFS, da sie nicht sehr ausdrucksstark sind: Der Schwerpunkt liegt bei beiden Sprachen auf der Repräsentation und der Verknüpfung von Informationen im Internet; die Modellierung von Element- und Beziehungstypen spielt nur eine untergeordnete Rolle.

Eine (auf DL basierende) Ontologie besteht aus einer Vielzahl von Axiomen, die sich in T-Box und A-Box unterteilen lassen:[38]

Die T-Box (auch *Schema der Ontologie* und *terminologisches Wissen* genannt) beinhaltet Informationen über Elementtypen, Beziehungstypen und ihre Verbindungen untereinander. Diese Informationen gelten unabhängig von einer konkreten Situation. Folgendes Beispiel zeigt, wie aus drei Aussagen eine weitere geschlossen werden kann:

gegeben	jede *Firma* *beschäftigt* mindestens einen *Mitarbeiter*
gegeben	jede *GmbH* *beschäftigt* mindestens einen *Geschäftsführer*
gegeben	*Geschäftsführer* sind *Mitarbeiter*
geschlossen	jede *GmbH* ist eine *Firma*

[36] Vgl. Angele, J./Kifer, M./Lausen, G.: Ontologies in F-logic, in: Staat, S./Studer, R. (Hrsg.): Handbook on Ontologies, Berlin/Heidelberg 2009.

[37] Vgl. Parreiras/Staab/Winter: On Marrying Ontological and Metamodeling Technical Spaces, Abschnitt 2.1.2.

[38] Außerdem kann eine OWL-Ontologie Annotationen enthalten. Diese spielen aber für diese Arbeit keine Rolle, da sie keinen semantischen Gehalt haben.

Die A-Box (auch *faktisches Wissen* und *Assertions* genannt) hingegen enthält Angaben über einzelne Individuen, Beziehungen und ihre Zugehörigkeit zu in der T-Box definierten Elementtypen und Beziehungstypen. Auch zu faktischem Wissen sind Schlussfolgerungen möglich, wie diese Beispiel in Verbindung mit der im obigen Beispiel beschriebenen T-Box zeigt:

gegeben	*ACME* *beschäftigt MaxMuster*
gegeben	*MaxMuster* ist ein *Geschäftsführer*
geschlossen	*ACME* ist eine *GmbH*

2.4.3 XML Technology Space

Der XML Technology Space (XML TS) umfasst zahlreiche Sprachen, die als Gemeinsamkeit den vom World Wide Web Consortium (W3C) standardisierten Ideen von der Extensible Markup Language (XML) folgen. XML ist im Bereich der Repräsentation und der Kommunikation strukturierter und semi-strukturierter Daten weit verbreitet. Ein zentrales Konzept im XML TS ist das "Dokument". XML-Dokumente müssen grundsätzlich den in der XML-Spezifikation vorgegebenen syntaktischen Regeln folgen und werden dann als *well-formed* bezeichnet. Um über die Syntax von XML hinausgehende Vorgaben für die Struktur eines XML-Dokuments machen zu können, lassen sich Regeln in einem separaten Text aufschreiben. Dazu existieren diverse *Schema-Sprachen*, z.B. Document Type Definition (DTD), XML Schema Definition Language (XSD) und RelaxNG. Die größte Bedeutung unter ihnen hat XSD, dessen Struktur selbst in XSD aufgeschrieben und damit komplett im XML TS enthalten ist.

Im XML TS gibt es eine große Menge von Software-Werkzeugen zum Verarbeiten von XML-Dokumenten. Zur Transformation innerhalb des XML TS werden meist die Extensible Stylesheet Language Transformations (XSLT) verwendet, da so die gesamte Verarbeitung innerhalb des XML TS stattfindet – XSLT-Stylesheets selbst sind XML-Dokumente.

Kapitel 3

Technik

In diesem Kapitel werden einige im Rahmen der Arbeit verwendeten Sprachen kurz vorgestellt. Dabei wird speziell auf Details eingegangen, die für die in dieser Arbeit behandelte Fragestellung relevant sind. Es handelt sich jedoch nicht um eine umfassende Einführung in die jeweilige Sprache – hierzu wird auf die Sprachspezifikationen sowie weiterführende Literatur verwiesen.

3.1 Unified Modeling Language (UML)

3.1.1 Überblick

Bei der Unified Modeling Language (UML) handelt es sich um eine formelle, visuelle Sprache (→ 2.1.7 Sprache) aus dem MDA TS (→ 2.4.1 Model-Driven Architecture Technology Space). Die UML wurde 1997 von der Object Management Group (OMG) standardisiert und seitdem fortlaufend überarbeitet.

Neben der Spezifikation konzeptueller Schemata, den Klassendiagrammen, enthält die UML noch viele weitere Bestandteile.[39] **Diese Arbeit beschäftigt sich ausschließlich mit dem Teil von UML, der zur Spezifikation konzeptueller Schemata benötigt wird.**

3.1.2 Bestandteile von UML-Klassendiagrammen

Die Abbildung 3.1 zeigt ein Beispiel für ein UML-Klassendiagramm, in dem die wichtigsten Bestandteile hervorgehoben sind. Da die graphische Syntax der Bestandteile von UML detailliert in den Kapiteln 7 bis 11 vorgestellt wird, folgt hier nur ein sehr knapper Überblick.

Zwei Gruppen von Elementen sind zu erkennen: Rechtecke und Linien zwischen diesen Rechtecken. In Form von Rechtecken werden dargestellt:

Klassen — Eine Klasse fasst Objekte mit gleichen Eigenschaften zusammen (→ 7 Elementtypen). Zur Identifikation besitzen Klassen einen Namen, der im oberen Teil des Rechtecks angegeben ist. Mit Kursivschreibung werden Klassen als abstrakt (→ 7.4 Abstrakte Elementtypen) gekennzeichnet, d.h. diese Klassen haben keine direkten Instanzen.

Datentypen — Ein Datentyp (→ 8 Datentypen) fasst Werte mit gleichen Eigenschaften zusammen. Im Gegensatz zu Objekten haben Werte keine Identität. Am Schlüsselwort «data-

[39] Olivé: Conceptual Modeling, S. 407: *"The UML metaschema is very large because UML deals not only with conceptual schemas but also with other kinds of software model."*

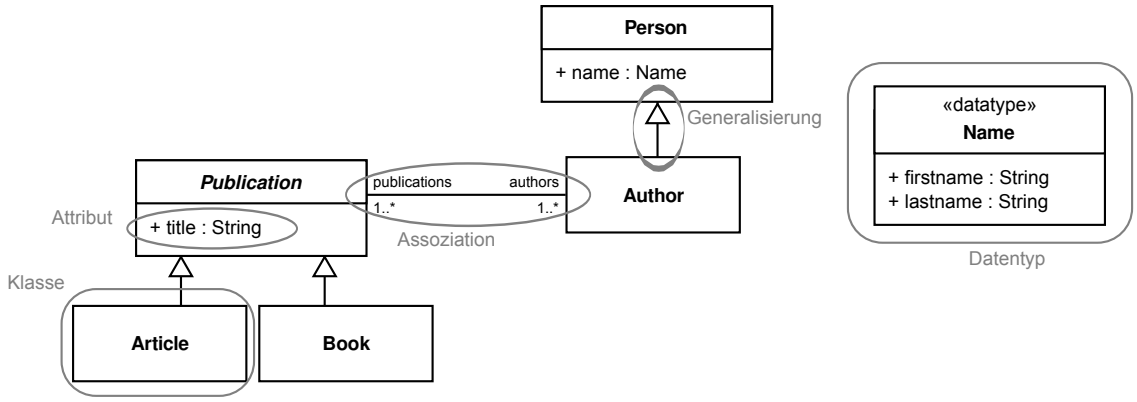


Abbildung 3.1. Beispiel für ein UML-Klassendiagramm.

type» am oberen Rand des Rechtecks ist zu erkennen, dass es sich um einen Datentypen und keine Klassen handelt.

Sowohl Klassen als auch Datentypen können **Attribute** enthalten. Hierbei handelt es sich um benannte Beziehungen zu einer anderen Klasse oder zu einem anderen Datentyp (→ 9 Beziehungstypen). Diese werden in den Bereich unterhalb des Klassen- bzw. Datentypnamens geschrieben. In einem weiteren Bereich des Klassen- bzw. Datentyp-Rechtecks können Operationen angegeben werden. Da sich diese Arbeit nur mit statischen Datenmodellen befasst, wird auf diese nicht weiter eingegangen.

Linien zwischen Rechtecken gibt es sowohl ohne als auch mit verschiedenen Pfeilspitzen. Als Linien bzw. Pfeile werden dargestellt:

Generalisierungen — Eine Generalisierung (→ 7.3 Vererbung), auch Vererbung genannt, wird mit Hilfe eines Pfeils mit weißem Dreieck als Spitze dargestellt. Bei einer solchen Vererbung übernimmt die erbende Klasse (Unterklasse) die Eigenschaften der vererbenden Klasse (Oberklasse). Der Pfeil zeigt von Richtung der Unterklasse auf die Oberklasse.

Assoziationen — Mit Hilfe einer Assoziation wird eine Beziehung zwischen zwei Klassen abgebildet (→ 9 Beziehungstypen). Sie wird als einfache Linie dargestellt. Wird die Linie mit einer einfachen Pfeilspitze versehen, so handelt es sich um eine gerichtete Beziehung, die nur von einer Klasse aus nutzbar ist. An die Enden einer Assoziation lassen sich Rollennamen sowie Kardinalitätsbeschränkungen (→ 9.4 Kardinalitätsbeschränkungen) schreiben.

3.1.3 Das UML-Metamodell

Das UML-Metamodell – auch konzeptuelles Modell der UML oder UML-Metaschema genannt – ist eine Instanz des MOF und besteht aus mehreren hundert Elementtypen.[41] Abbildung 3.2 zeigt die wichtigsten Elementtypen, die für konzeptuelle Modellierung benötigt werden.

[40] Nach Olivé: Conceptual Modeling, S. 407.
[41] Vgl. a. a. O.

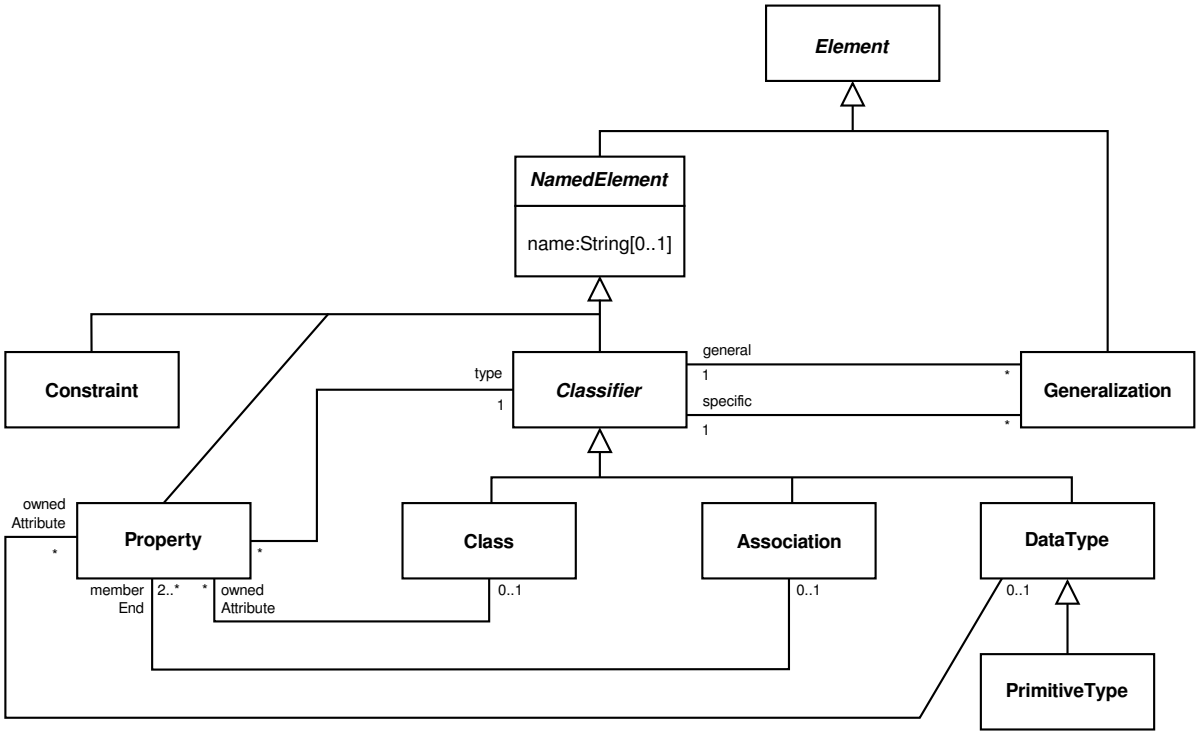


Abbildung 3.2. Vereinfachter Auszug aus dem UML-Metamodell.[40]

Die Elementtypen von UML werden jeweils in den Kapiteln 7 bis 11 im Detail vorgestellt, hier daher nur eine knappe Erläuterung der in Abbildung 3.2 gezeigten Elementtypen:

- ▻ *Association* – Die Instanzen dieses Meta-Elementtyps sind Assoziationen. (→ 9 Beziehungstypen)
- ▻ *Class* – Die Instanzen dieses Meta-Elementtyps sind Elementtypen. (→ 7 Elementtypen)
- ▻ *Classifier* – Dieser abstrakte Elementtyp fasst Elementtypen, Datentypen, Assoziationen und einige weitere Modellelemente zusammen. Es handelt sich bei ihnen um Klassifikationselemente, für die Vererbungsbeziehungen definiert werden können. Sie lassen sich als verallgemeinertes Klassenkonzept auffassen.[42]
- ▻ *Constraint* – Die Instanzen dieses Meta-Elementtyps sind Einschränkungen.
- ▻ *DataType* – Die Instanzen dieses Meta-Elementtyps sind Datentypen. (→ 8 Datentypen)
- ▻ *Element* – Mit Hilfe dieses abstrakten Elementtyps werden alle anderen Modellelemente zusammengefasst, unabhängig davon, ob sie einen Namen besitzen oder nicht.

[42] Vgl. Reussner, R./Hasselbring, W. (Hrsg.): Handbuch der Software-Architektur, Heidelberg, 2. Auflage 2009, S. 40f.

- *Generalization* – Instanzen dieses Meta-Elementtyps repräsentieren *IsA*-Beziehungen (→ 2.2 IsA und InstanceOf) zwischen zwei Elementtypen. (→ 9.3 Vererbung)
- *NamedElement* – Dieser abstrakte Elementtyp fasst diejenigen Modellelemente zusammen, die einen Namen besitzen. Wie in Abbildung 3.2 zu erkennen ist, besitzt eine *Generalization* keinen Namen. (→ 7.2 Namen)
- *PrimitiveType* – Hierbei handelt es sich um vordefinierte Datentypen wie z.B. Boolean oder String. (→ 8 Datentypen)
- *Property* – Instanzen dieses Meta-Elementtyps sind entweder klassenabhängige Attribute oder Teile einer *Association*. (→ 9 Beziehungstypen)

3.1.4 Semantikerweiterung durch Profile

Bei UML besteht die Möglichkeit, mit Hilfe eines sogenannten *Profils* ein Metamodell so anzupassen, dass es für ein spezielles Anwendungsgebiet besser geeignet ist. Ein Profil ist eine Instanz von *Profile* und damit ebenfalls ein *Package* (→ 11.1 Pakete). Mit Hilfe eines Profils lässt sich ein Metamodell – meist das UML-Metamodell – spezialisieren, indem z.B. striktere Einschränkungen als im Metamodell selbst formuliert werden.[43]

Bei der Definition und Verwendung eines Profils bleibt das UML-Metamodell selbst unverändert.[44] Dies ist im Zusammenhang mit dem "UML-Profil" der ISO19100 Normenfamilie wichtig und wird in Abschnitt 3.4 ausführlich diskutiert. Da das UML-Metamodell unverändert bleibt, wird das Sprachkonzept Profil auch "leichtgewichtiger Erweiterungsmechanismus"[45] genannt.

Im Wesentlichen besteht ein Profil aus einer Menge von definierten *Stereotypen*. Mit der Definition eines Stereotypen wird festgelegt, in welcher Weise ein Elementtyp des UML-Metamodells angepasst wird. Neben einem obligatorischen Namen verfügt ein Stereotyp über optionale Eigenschaftsdefinitionen (engl. *tag definitions*) und Einschränkungen (engl. *constraints*). Ein Stereotyp kann stets nur auf einen bestimmten (Meta-)Elementtyp angewendet werden.

Soll ein zuvor definiertes Profil genutzt werden, so muss es angewendet (engl. *applied*) werden. Dabei handelt es sich um eine spezielle Art des Paketimports (→ 11.1 Pakete). Wurde das Profil angewendet, können Modellelemente mit den dort definierten Stereotypen versehen werden. In der grafischen Syntax wird dazu der Name des Stereotypen in französischen Anführungszeichen an das Modellelement geschrieben – meist oberhalb seines Namens.

[43] OMG: UML Infrastructure, S. 177: "*1. A profile must provide mechanisms for specializing a reference metamodel (such as a set of UML packages) in such a way that the specialized semantics do not contradict the semantics of the reference metamodel. That is, profile constraints may typically define well-formedness rules that are more constraining (but consistent with) those specified by the reference metamodel.*"

[44] A. a. O., S. 178: "*8. [. . .] The reference metamodel is considered as a 'read only' model, that is extended without changes by profiles. It is therefore forbidden to insert new metaclasses in the UML metaclass hierarchy (i.e., new super-classes for standard UML metaclasses) or to modify the standard UML metaclass definitions (e.g., by adding meta-associations).*"

[45] Balzert, H.: Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering, Band 1, Heidelberg, 3. Auflage 2009, S. 103.

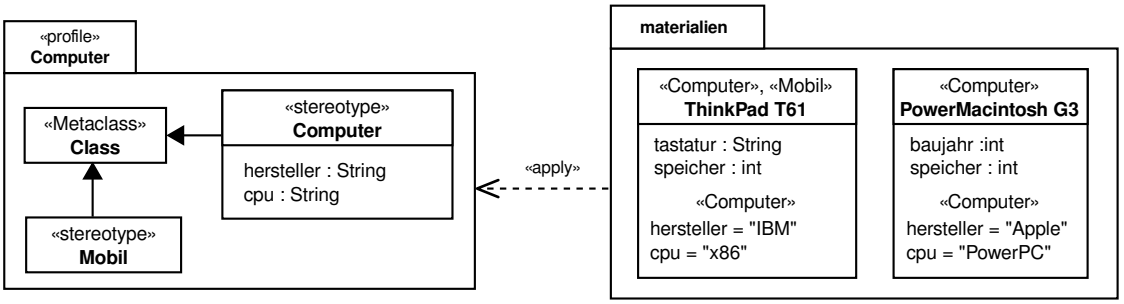


Abbildung 3.3. Beispiel für die Definition und Anwendung eines UML-Profils.

In Abbildung 3.3 wird auf der linken Seiten die Definition eines Profils mit zwei Stereotypen gezeigt. Die rechte Seite zeigt ein Paket, auf das dieses Profil angewendet wird. Beide Stereotypen können auf Klassen angewendet werden. Während der Stereotyp «Mobil» nur zur Kennzeichnung von Klassen (im Beispiel "ThinkPad T61") verwendet werden kann, sind dem Stereotyp «Computer» zusätzlich zwei tagged values "hersteller" und "cpu" zugeordnet. Klassen (nicht die Instanzen dieser Klassen), die mit dem Stereotyp «Computer» versehen sind, müssen entsprechende Werte aufweisen. Diese werden unterhalb der klassenabhängigen Attribute in den Attribut-Teil des Klassenrechtecks geschrieben. Um zu kennzeichnen, dass die beiden tagged values zu dem Stereotyp «Computer» gehören, wird der Stereotypnamen über die Wertzuweisungen geschrieben.

3.2 OWL 2 Web Ontology Language (OWL 2)

Im Folgenden ist mit der Abkürzung OWL immer die OWL 2 Web Ontology Language gemeint. Sollte es die Möglichkeit der Verwechslung zwischen der OWL Web Ontology Language und der OWL 2 Web Ontology Language geben, so werden zur Klarheit an der Stelle die Abkürzungen OWL 1 bzw. OWL 2 verwendet.

3.2.1 Überblick

Bei der OWL handelt es sich um eine formelle, textuelle Sprache (→ 2.1.7 Sprache) aus dem Ontology TS (→ 2.4.2 Ontology Engineering Technology Space). Die OWL folgt dem RDF-Sprachparadigma, bei dem Aussagen über Resourcen getroffen werden. Jede Resource ist über einen Bezeichner eindeutig identifiziert. Aussagen erfolgen in der Form: Subjekt–Prädikat–Objekt. Zentrale Konzepte von OWL sind Klasse, Individuum, Eigenschaft, Beziehung.

OWL bietet neben der Definition einer abstrakten Struktur für Ontologien[46] mehrere konkrete Syntaxen an, mit denen Ontologien serialisiert und ausgetauscht werden können. Darunter sind sowohl XML-basierte Syntaxen (RDF/XML und OWL/XML) als auch nicht-XML-basierte Syntaxen (functional-style, Manchester und Turtle Syntax). In dieser Arbeit wird

[46] Vgl. W3C: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, 2012 ⟨URL: http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/⟩.

als Syntax für OWL ausschließlich die functional-style Syntax verwendet, die der abstrakten Struktur sehr ähnlich ist. So lassen sich konkrete und abstrakte Syntax gedanklich leichter in Einklang bringen.

Um einen besserer Eindruck für den Zusammenhang zwischen konkreter und abstrakter Syntax von OWL zu bekommen, zeigen Listing 3.1 und Abbildung 3.4 die gleiche Ontologie in konkreter und abstrakter Syntax. Die Kapitel 7 bis 11 enthalten zahlreiche weitere Gegenüberstellungen von konkreter und abstrakter Syntax.

```
1 Prefix( :=<urn:Package#> )
2 Ontology( <urn:Package>
3   Declaration( Class( :Book ) )
4   Declaration( Class( :Person ) )
5
6   Declaration( ObjectProperty( :author ) )
7   ObjectPropertyDomain( :author :Book )
8   ObjectPropertyRange( :author :Person )
9 )
```

Listing 3.1. Beispiel für konkrete (functional-style) Syntax einer OWL-Ontologie.

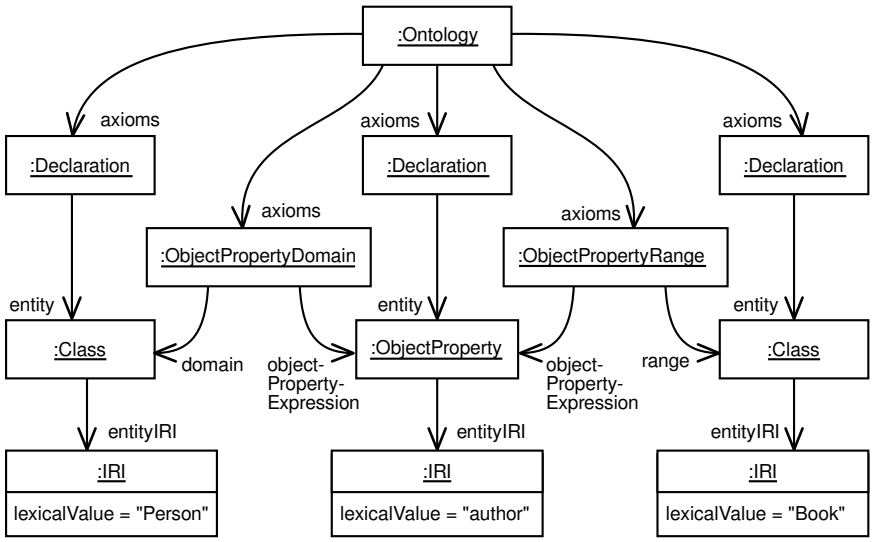


Abbildung 3.4. Beispiel für die abstrakte Syntax (Instanzen der OWL-Metaklassen) einer OWL-Ontologie als UML-Objektdiagramm geschrieben.

Seit OWL 2 ist die Definition der abstrakten Syntax, dort "structural specification" genannt, Teil der Spezifikation.[47] Diese verwendet zur Modellierung MOF: "*The structural specification is defined using the Unified Modeling Language (UML), and the notation used is compatible with the*

[47] Zuvor gab es bei OWL 1 Bestrebungen, eine solche abstrakte Syntax außerhalb der Spezifikation nachträglich zu definieren. Ein prominentes Modell ist das Ontology Definition Model (ODM), siehe OMG: Ontology Definition Metamodel, 2009 ⟨URL: `http://www.omg.org/spec/ODM/1.0/`⟩.

Meta-Object Facility (MOF).”[48]

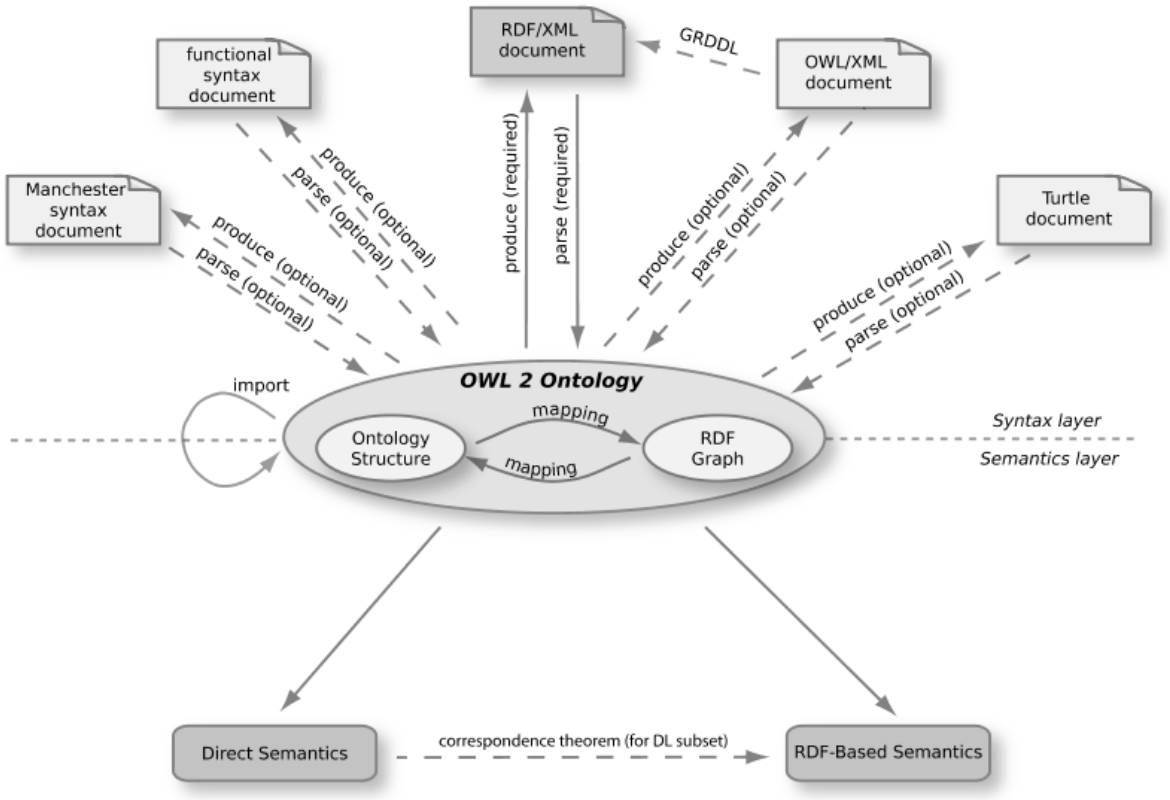


Abbildung 3.5. Die Struktur von OWL 2.[49]

Wie in Abbildung 3.5 zu sehen ist, existieren für die OWL 2 zwei Semantiken: die *RDF-Based Semantics*[50] sowie die modelltheoretische *Direct Semantics*[51], die kompatibel zur Beschreibungslogik $\mathcal{SROIQ}$ ist. Wird die Direct Semantics verwendet und werden die Regeln von OWL 2 DL eingehalten, so lassen sich aus der Literatur bekannte Techniken zum Reasoning einsetzen.[52] **Es werden für diese Arbeit die Direct Semantics und OWL 2 DL zugrunde gelegt**, um entsprechend mit den Ontologien maschinell arbeiten zu können.

Neben der abstrakten Struktur ist es auch möglich, OWL-Ontologien als RDF-Graph zu verstehen (rechte helle Ellipse in Abbildung 3.5) und anstelle der Direct Semantics eine RDF-basierte Semantik zu verwenden (Rechteck rechts unten in Abbildung 3.5). Auf beide

[48] W3C: OWL 2 Structural specification, Abschnitt 2.1.

[49] W3C: OWL 2 Web Ontology Language Document Overview, 2012 ⟨URL: `http://www.w3.org/TR/2012/REC-owl2-overview-20121211/`⟩, Abschnitt 2.

[50] Siehe W3C: OWL 2 Web Ontology Language RDF-Based Semantics, 2012 ⟨URL: `http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/`⟩.

[51] Siehe W3C: OWL 2 Web Ontology Language Direct Semantics, 2012 ⟨URL: `http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/`⟩.

[52] Vgl. W3C: OWL 2 Structural specification, Abstract.

Alternativen wird aber im Rahmen dieser Arbeit nicht weiter eingegangen.

3.2.2 Bestandteile von OWL-Ontologien

Wie eine jede (auf DL basierende) Ontologie lassen sich die Bestandteile einer OWL-Ontologie in A-Box und T-Box aufteilen (→ 2.4.2 Ontology Engineering Technology Space).

Bei OWL lassen sich die Bestandteile der T-Box, die das Schema einer Ontologie definiert, in zwei große Gruppen einteilen: Expressions und Axiome. Mit Hilfe der Expressions werden Elementtypen und Beziehungstypen beschrieben. Um Aussagen über die so beschriebenen Typen treffen zu können, werden Axiome verwendet. Axiome definieren

- Eigenschaften von Beziehungstypen sowie
- Beziehungen von Elementtypen und Beziehungstypen untereinander und miteinander.

Expressions

Es gibt drei Arten von Expressions, die sich als abstrakte Klassen im Metamodell (Abbildung 3.6) wiederfinden: *ClassExpression*, *ObjectPropertyExpression* und *DataPropertyExpression*.

ClassExpression — Eine *ClassExpression* stellt in OWL einen Elementtyp (Klasse) dar. Es existiert eine Vielzahl von *ClassExpression* (wie in Abbildung 3.6 zu sehen ist). Eine besondere Rolle spielen Instanzen von *Class*, bei denen es sich um einen Untertyp von *ClassExpression* handelt. Diese besitzen einen Namen und stellen somit benannte Elementtypen dar. Alle anderen *ClassExpression* lassen sich als anonyme Elementtypen verstehen, bei denen eine Bedingung angegeben ist, unter denen ein Individuum Instanz dieses Elementtyps ist. So beschreibt die *ClassExpression* `ObjectSomeValuesFrom( :arbeitetBei :Firma )` einen anonymen Elementtyp, dem genau die Individuen angehören, die eine oder mehrere Beziehungen vom Typ "arbeitetBei" zu einer Instanz von "Firma" besitzen.

ObjectPropertyExpression — Eine *ObjectPropertyExpression* stellt in OWL einen Beziehungstyp dar. Es gibt zwei Arten von *ObjectPropertyExpression*: benannte *ObjectProperty* sowie anonyme Inverse eines Beziehungstyps *InverseObjectProperty*. Ist ein inverser Beziehungstyp `ObjectInverseOf( :arbeitetBei )` definiert, so ist Voraussetzung für die Existenz dieser Beziehung zwischen zwei Individuen I_1 und I_2, dass zwischen I_2 und I_1 eine "arbeitetBei"-Beziehung besteht.

DataPropertyExpression — Es existiert genau eine Art von *DataPropertyExpression*, nämlich *DataProperty*. Mit ihr wird ein benannter Beziehungstyp zwischen einem Individuum und einem Wert beschrieben.

Axiome

Wie im OWL 2-Metamodell (Abbildung 3.6) zu erkennen ist, enthält eine Ontologie als direkte Bestandteile nur Axiome, keine Expressions. Erst innerhalb eines Axioms wird eine Aussage über eine oder mehrere Expressions getroffen. Es lassen sich drei große Gruppen von

Axiomen identifizieren, die als abstrakte Klassen im Metamodell (Abbildung 3.6) zu finden sind: *ClassAxiom*, *ObjectPropertyAxiom* und *DataPropertyAxiom*. Daneben gibt es noch weitere Axiome:

- *Declaration* zur Zuordnung von Namen zu Elementtypen, Beziehungstypen und Individuen
- *DatatypeDefinition* zur Definition von Datentypen
- *HasKey* zur Definition von Schlüsseln (→ 10 Beschränkungen).

ClassAxiom — Mit Hilfe eines *ClassAxiom* werden Beziehungen zwischen *ClassExpression* definiert. Dabei handelt es sich entweder um eine Vererbungsbeziehung *IsA* oder um die Disjunktheit zweier Elementtypen.

ObjectPropertyAxiom — Mit einem *ObjectPropertyAxiom* werden Aussagen über *ObjectProperty* gemacht. Drei Arten von Aussagen sind möglich:

- Beziehungen zwischen zwei Object Properties wie Vererbung, Invertierung und Disjunktheit
- Eigenschaften einer Object Property wie Symmetrie, Transitivität, Reflexivität, etc.
- Angabe von Definitionsbereich und Zielbereich der Object Property, also der Festlegung, von welchem Elementtyp die beteiligten Individuen sein müssen.

DataPropertyAxiom — Mit einem *DataPropertyAxiom* werden Aussagen über *DataProperty* gemacht. Wie auch bei einem *ObjectPropertyAxiom* sind die drei Arten von Aussagen möglich. Allerdings kann hier nur eine einzige Eigenschaft festgelegt werden, nämlich dass eine Data Property funktional ist.

Die Axiome der A-Box werden in OWL *Assertion* genannt. Sie beschäftigen sich mit Individuen – der Zuordnung von Individuen zu Elementtypen sowie Beziehungen zwischen Individuen – und spielen daher für die konzeptuelle Modellierung keine Rolle.

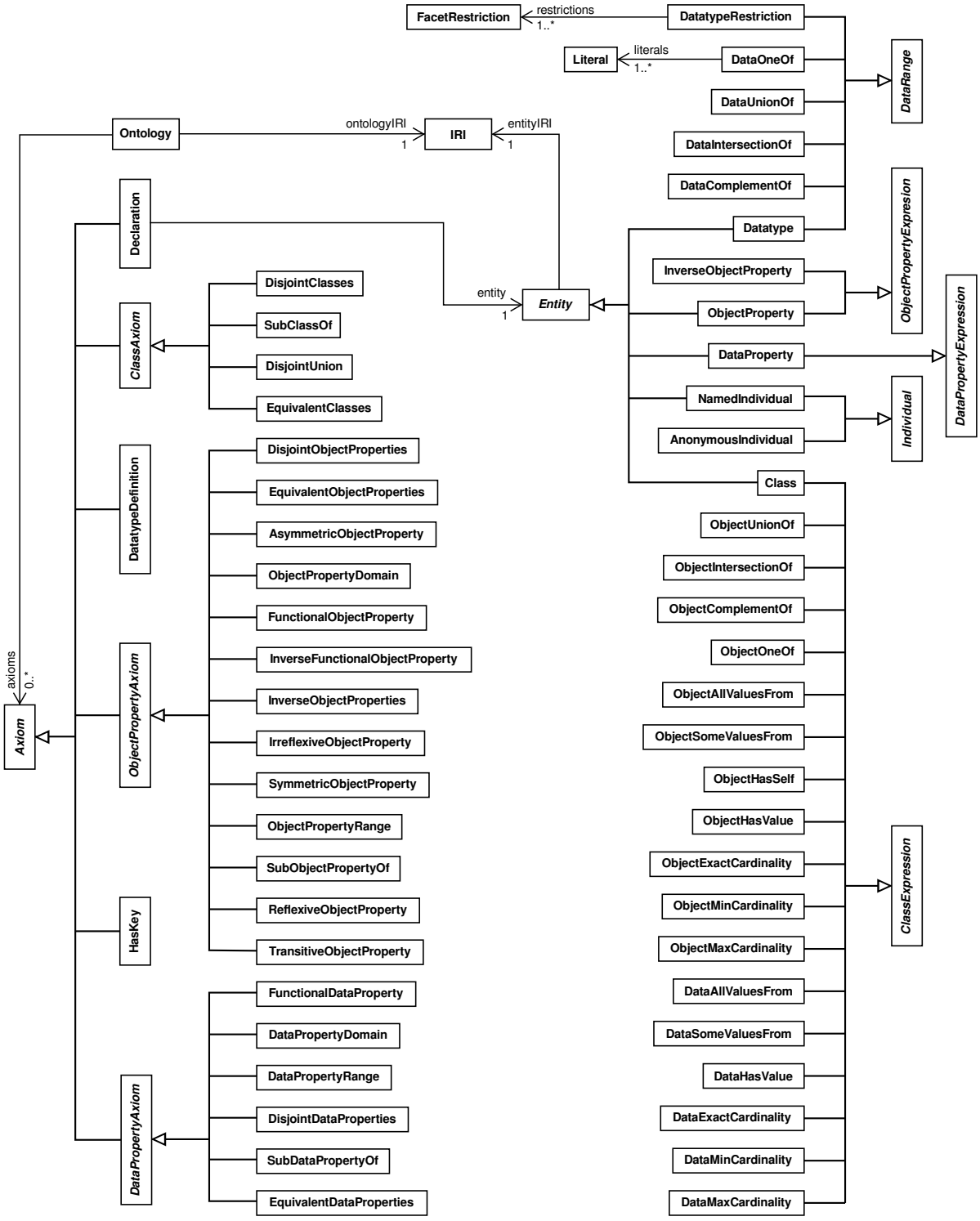


Abbildung 3.6. Die 69 Elementtypen des OWL 2-Metamodells, die für die Definition des Schemas einer Ontologie verwendet werden. Dargestellt sind Vererbungsbeziehungen, sowie wenige weitere Beziehungen zwischen Elementtypen.

3.2.3 Das OWL-Metamodell

Das OWL-Metamodell besteht aus insgesamt 86 Klassen. Von diesen stehen neun Klassen[53] in Zusammenhang mit Annotationen, die – wie bereits geschildert – im Rahmen dieser Arbeit keine Rolle spielen.

Von den zu den Elementen der A-Box gehörenden zehn Klassen[54] werden ebenfalls viele nicht benötigt, wenn nur die in einer Ontologie aufgeschriebenen konzeptuellen Modelle betrachtet werden sollen. Lediglich die zur Bildung abgeschlossener Klassen benötigten Elementtypen *NamedIndividual* und *AnonymousIndividual* werden zur Definition des Schemas der Ontologie benötigt.

Somit bleiben 69 Klassen des OWL-Metamodells, die bei der Definition des Schemas einer Ontologie zum Einsatz kommen. Von diesen Klassen sind wiederum zehn abstrakte Klassen. Abbildung 3.6 zeigt den Zusammenhang der Vererbungsbeziehungen zwischen diesen Klassen. Da eine Darstellung aller sonstiger Beziehungen zwischen den Elementtypen, bei denen es sich nicht um *IsA*-Beziehungen handelt, ein kaum zu lesendes Diagramm zur Folge gehabt hätte, sind nur einige ausgewählte andere Beziehungen eingetragen.

3.2.4 "Syntaktischer Zucker"

OWL enthält eine Vielzahl von Ausdrücken, die "syntaktischer Zucker" sind, d. h. es lässt sich eine semantisch äquivalente Aussage auch ohne diese zusätzlichen Ausdrücke aufschreiben, wenn auch meist auf recht komplizierte Weise. Für einen Menschen sind die Kurzschreibweisen in der Regel besser verständlich.

Bei der Transformation kann an manchen Stellen ausgenutzt werden, dass ein Ausdruck in der Langschreibweise eines anderen Ausdrucks enthalten ist. Ebenso erklärt die Langschreibweise in einigen Fällen, warum ein Ausdruck nicht zu transformieren ist.

Tabelle 3.1 zeigt eine Gegenüberstellung der Kurzschreibweise von OWL-Axiomen und ihrer ausführlichen Schreibweise.

3.2.5 Eine graphische Repräsentation für Ontologien

Es gibt in der Literatur eine Vielzahl von Ansätzen, wie Ontologien graphisch repräsentiert werden können. Katifori et al. geben einen Überblick über die bis 2007 veröffentlichten Ideen.[55] Schwerpunkt der Forschung scheint momentan eher der Umgang mit einer großen Menge von Individuen und weniger die Visualisierung der Struktur einer Ontologie zu

[53] Dies sind: Annotation, AnnotationAssertion, AnnotationAxiom, AnnotationProperty, AnnotationPropertyDomain, AnnotationPropertyRange, AnnotationSubject, AnnotationValue, SubAnnotationPropertyOf.

[54] Dies sind: AnonymousIndividual, Assertion, ClassAssertion, DataPropertyAssertion, DifferentIndividuals, NamedIndividual, NegativeDataPropertyAssertion, NegativeObjectPropertyAssertion, ObjectPropertyAssertion, SameIndividual.

[55] Katifori, A. et al.: Ontology Visualization Methods—A Survey, in: ACM Computing Surveys (CSUR) 39/4, New York 2007 ⟨URL: http://disi.unitn.it/~p2p/RelatedWork/Matching/a10-katifori.pdf⟩.

Tabelle 3.1. Kurzschreibweisen von OWL-Axiomen und ihre ausführliche Schreibweise

Abkürzung	**Äquivalente Schreibweise**
DataAllValuesFrom(DPE$_1$ ... DPE$_n$ DR)	DataMaxCardinality(0 DPE DataComplementOf(DR))
DataHasValue(DPE lt)	DataMinCardinality(1 DPE DataOneOf(lt))
DataPropertyDomain(DPE CE)	SubClassOf(DataMinCardinality(1 DPE rdfs:Literal) CE)
DataPropertyRange(DPE DR)	SubClassOf(owl:Thing DataMaxCardinality(0 DPE DataComplementOf(DR)))
DataSomeValuesFrom(DPE$_1$ ... DPE$_n$ DR)	DataMinCardinality(1 DPE DR)
DisjointClasses(CE$_1$ CE$_2$)	SubClassOf(CE$_1$ ObjectComplementOf(CE$_2$))
DisjointClasses(CE$_1$ ... CE$_n$)	paarweise DisjointClasses(CE$_i$ CE$_j$)
DisjointDataProperties(DPE$_1$ ... DPE$_n$)	paarweise DisjointDataProperties(DPE$_i$ DPE$_j$)
DisjointObjectProperties(OPE$_1$ ... OPE$_n$)	paarweise DisjointObjectProperties(OPE$_i$ OPE$_j$)
DisjointUnion(C CE$_1$ ... CE$_n$)	SubClassOf(C ObjectUnionOf(CE$_1$ ... CE$_n$)), SubClassOf(ObjectUnionOf(CE$_1$ ... CE$_n$) C), DisjointClasses(CE$_1$ ... CE$_n$)
EquivalentClasses(CE$_1$ CE$_2$)	SubClassOf(CE$_1$ CE$_2$), SubClassOf(CE$_2$ CE$_1$)
EquivalentDataProperties(DPE$_1$ ... DPE$_n$)	SubDataPropertyOf(DPE$_1$ DPE$_2$), SubDataPropertyOf(DPE$_2$ DPE$_1$)
EquivalentObjectProperties(OPE$_1$ OPE2)	SubObjectPropertyOf(OPE$_1$ OPE$_2$),SubObjectPropertyOf(OPE$_2$ OPE$_1$)
FunctionalDataProperty(DPE)	SubClassOf(owl:Thing DataMaxCardinality(1 DPE))
FunctionalObjectProperty(OPE)	SubClassOf(owl:Thing ObjectMaxCardinality(1 OPE))
InverseFunctionalObjectProperty(OPE)	SubClassOf(owl:Thing ObjectMaxCardinality(1 ObjectInverseOf(OPE)))
InverseObjectProperties(OPE$_1$ OPE$_2$)	SubObjectPropertyOf(OPE$_1$ ObjectInverseOf(OPE$_2$)), SubObjectPropertyOf(ObjectInverseOf(OPE$_2$) OPE$_1$)
IrreflexiveObjectProperty(OPE)	SubClassOf(ObjectHasSelf(OPE) owl:Nothing)
ObjectAllValuesFrom(OPE CE)	ObjectMaxCardinality(0 OPE ObjectComplementOf(CE))
ObjectHasValue(OPE a)	ObjectMinCardinality(1 OPE ObjectOneOf(a))
ObjectPropertyDomain(OPE CE)	SubClassOf(ObjectMinCardinality(1 OPE owl:Thing) CE)
ObjectPropertyRange(OPE CE)	SubClassOf(owl:Thing ObjectMaxCardinality(0 OPE ObjectComplementOf(CE)))
ObjectSomeValuesFrom(OPE CE)	ObjectMinCardinality(1 OPE CE)
ReflexiveObjectProperty(OPE)	SubClassOf(owl:Thing ObjectHasSelf(OPE))
SymmetricObjectProperty(OPE)	SubObjectPropertyOf(OPE ObjectInverseOf(OPE))
TransitiveObjectProperty(OPE)	SubObjectPropertyOf(ObjectPropertyChain(OPE OPE) OPE)

sein. Es gibt Ansätze, die Syntax von UML auch für Ontologien zu verwenden. Dazu zählen Arbeiten von Brockmans et al.[56,57], Djurić et al.[58] sowie Bārzdinš et al.[59] Im praktischen Gebrauch sind Visualisierungen, die mit einfachen grafischen Elementen auskommen und z.B. in den verschiedenen Ansichten des Protégé Ontologie-Editors zum Einsatz kommen.[60,61]

Für diese Arbeit ist es weniger wichtig, eine große Menge von Individuen einer Ontologie darzustellen, sondern vielmehr, die Struktur übersichtlich zu präsentieren. Daher soll folgende, an die obigen Arbeiten angelehnte, stark vereinfachte graphische Syntax für Ontologie verwendet werden.

Eine Klasse wird durch ein Oval dargestellt. In das Oval ist der Name der Klasse geschrieben.

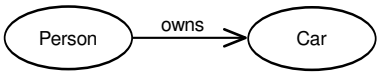
```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Person ) )

)
```

Ein Datentyp wird durch ein Rechteck dargestellt. In das Rechteck ist der Name der Klasse geschrieben.

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Datatype( :ISBN ) )

)
```

Eine Object Property wird durch einen gerichteten Pfeil von Definitionsbereich zu Zielbereich dargestellt. Der Pfeil verbindet zwei Ovale (=Klassen) miteinander. Die Beschriftung des Pfeils ist der Name der Object Property.

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Person ) )
  Declaration( Class( :Car ) )
  Declaration( ObjectProperty( :owns ) )
  ObjectPropertyDomain( :owns :Person )
  ObjectPropertyRange( :owns :Car )
)
```



[56] Brockmans, S. et al.: Visual Modeling of OWL DL Ontologies Using UML, in: McIlraith, Sheila A./ Plexousakis, Dimitris/Harmelen, Frank van (Hrsg.): The Semantic Web – ISWC 2004, Heidelberg 2004 ⟨URL: http://www.aifb.uni-karlsruhe.de/WBS/sbr/publications/iswc04_20sbr.pdf⟩.

[57] Brockmans, S. et al.: A Metamodel and UML Profile for Rule-extended OWL DL Ontologies, in: Cruz, I. et al. (Hrsg.): The Semantic Web – ISWC 2006, Heidelberg 2006.

[58] Djurić, D. et al.: A UML Profile for OWL Ontologies, in: Model Driven Architecture. European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers, Berlin/Heidelberg 2005 ⟨URL: http://www.springerlink.com/content/49yb6365gymtryfg/⟩.

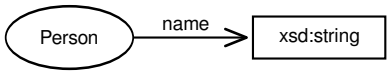
[59] Bārzdinš, Jānis et al.: UML Style Graphical Notation and Editor for OWL 2, in: Perspectives in Business Informatics Research. 9th International Conference, BIR 2010, Rostock Germany, September 29–October 1, 2010. Proceedings, Berlin/Heidelberg 2010.

[60] Milvich, M.: Ein Semantisches Web für die Universitätsbibliothek Heidelberg, Masterthesis Fachhochschule Karlsruhe, Karslruhe 2005.

[61] Flynn, J.: VisioOWL, 2006 ⟨URL: http://mysite.verizon.net/jflynn12/VisioOWL/VisioOWL.htm⟩ – Zugriff am 9. Februar 2012.

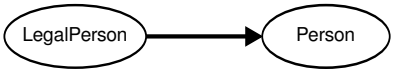
Eine Data Property wird ebenfalls durch einen gerichteten Pfeil von Definitionsbereich zu Zielbereich der Data Property dargestellt. Der Pfeil verbindet ein Oval (=Klasse) mit einem Rechteck (=Datentyp). Die Beschriftung des Pfeils ist der Name der Data Property.

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Person ) )
  Declaration( DataProperty( :name ) )
  DataPropertyDomain( :name :Person )
  DataPropertyRange( :name xsd:string )
)
```



Eine *IsA*-Beziehung zwischen zwei Klassen wird durch einen dickeren Pfeil mit dreieckiger Pfeilspitze symbolisiert. Der Pfeil zeigt von der speziellen zur generellen Klasse.

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Person ) )
  Declaration( Class( :LegalPerson ) )

  SubClassOf( :LegalPerson :Person )
)
```



3.3 XML Schema Definition Language (XSD)

3.3.1 Überblick

Die Extensible Markup Language (XML) ist eine formelle, textuelle Sprache (→ 2.1.7 Sprache) aus dem XML TS (→ 2.4.2 Ontology Engineering Technology Space). Sie folgt dem nach ihr benannten XML-Sprachparadigma, bei dem Elemente weitere Elemente als Kindelemente enthalten können und sich so eine baumförmige Struktur ergibt. Eine solche baumförmige Struktur wird als XML-Dokument bezeichnet. XML-Dokumente müssen grundsätzlich den in der XML-Spezifikation vorgegebenen syntaktischen Regeln folgen und werden dann als *well-formed* bezeichnet. Listing 3.2 gibt ein Beispiel für ein well-formed XML-Dokument an.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <Person id="Verne_Jules">
3     <name>Jules Verne</name>
4     <publications>
5         <Book>
6             <title>Vingt mille lieues sous les mers</title>
7         </Book>
8     </publications>
9 </Person>
```

Listing 3.2. Beispiel für ein well-formed XML-Dokument.

XML Information Set

Neben der bekannten konkreten Syntax von XML gibt es mit XML Information Set eine abstrakte Syntax für XML-Dokumente. Abbildung 3.7 zeigt ein Beispiel des zu Listing 3.2 gehörenden XML Information Set.

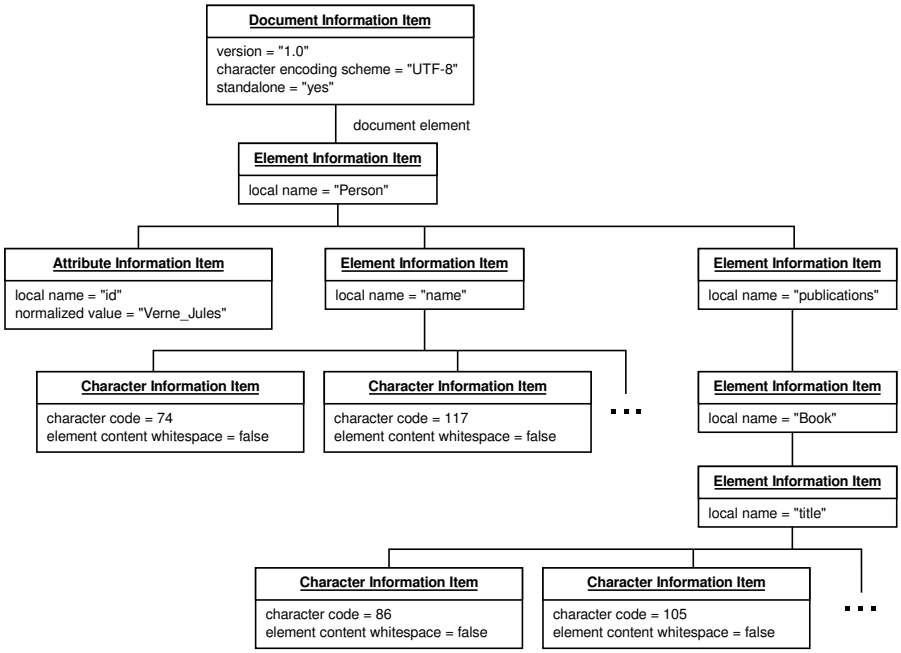


Abbildung 3.7. Auszug der Knoten aus dem Information Set für das Beispiel in Listing 3.2 in Form eines UML-Diagramms. Die Knoten sind als Objekte, ihre Eigenschaften als Attribute dargestellt.

3.3.2 XML Schema

Um über die Syntax von XML hinausgehende Vorgaben für die Struktur eines XML-Dokuments machen zu können, können Regeln in einem separaten Schema aufgeschrieben werden. Sind diese Regeln erfüllt, wird von einem *validen* XML-Dokument gesprochen. Zum Aufschreiben der Regeln kommt in den meisten Fällen die XSD zum Einsatz. Bei ihr handelt es sich ebenfalls um eine XML-basierte Sprache, die XML-Schemata sind also selbst wieder XML-Dokumente. Listing 3.3 zeigt ein Beispiel für ein XML-Schema, zu dem das XML-Dokument in Listing 3.2 valide ist.

XML Schema Component Model

So wie sich XML-Dokumente mit dem XML Information Set in einer abstrakten Weise darstellen lassen, ist dies auch für XML Schemata mit Hilfe des *XML Schema Component Model* (manchmal auch als XML Schema Abstract Data Model bezeichnet) möglich. Das in Abbildung 3.8 gezeigte Modell enthält neben dem *Schema*-Modellelement selbst 13 Modellelemente, die sich auf drei Gruppen aufteilen:[62]

- Haupt-Modellelemente mit optionalem Namen (*Complex Type Definition*, *Simple Type Definition*) sowie obligatorischem Namen (*Attribute Declaration*, *Element Declaration*)

[62] Vgl. W3C: XML Schema Part 1: Structures, 2004 ⟨URL: `http://www.w3.org/TR/xmlschema-1/`⟩, Abschnitt 2.2.

```
 1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 3   <xsd:element name="Person" type="Person" />
 4   <xsd:complexType name="Person">
 5     <xsd:sequence>
 6       <xsd:element name="name" type="xsd:string" />
 7       <xsd:element name="publications" type="Publications" />
 8     </xsd:sequence>
 9     <xsd:attribute name="id" type="xsd:ID" />
10   </xsd:complexType>
11   <xsd:complexType name="Publications">
12     <xsd:sequence>
13       <xsd:element name="Book" type="Book" maxOccurs="unbounded" />
14     </xsd:sequence>
15   </xsd:complexType>
16   <xsd:complexType name="Book">
17     <xsd:sequence>
18       <xsd:element name="title" type="xsd:string" />
19     </xsd:sequence>
20   </xsd:complexType>
21 </xsd:schema>
```

Listing 3.3. Beispiel für ein XML-Schema, zu dem das Dokument in Listing 3.2 valide ist.

- Neben-Modellelemente, die stets einen Namen besitzen müssen (*Attribute Group Definition*, *Identity-constraint Definition*, *Model Group Definition*, *Notation Declaration*)
- "Helfer"-Modellelemente, die nicht alleine Bestandteil eines Schemas sein können, sondern immer einem Haupt- oder Neben-Modellelement zugeordnet sind (*Annotation*, *Attribute Use*, *Model Group*, *Particle*, *Wildcard*).

[63] nach W3C: XML Schema Part 1, Anhang E

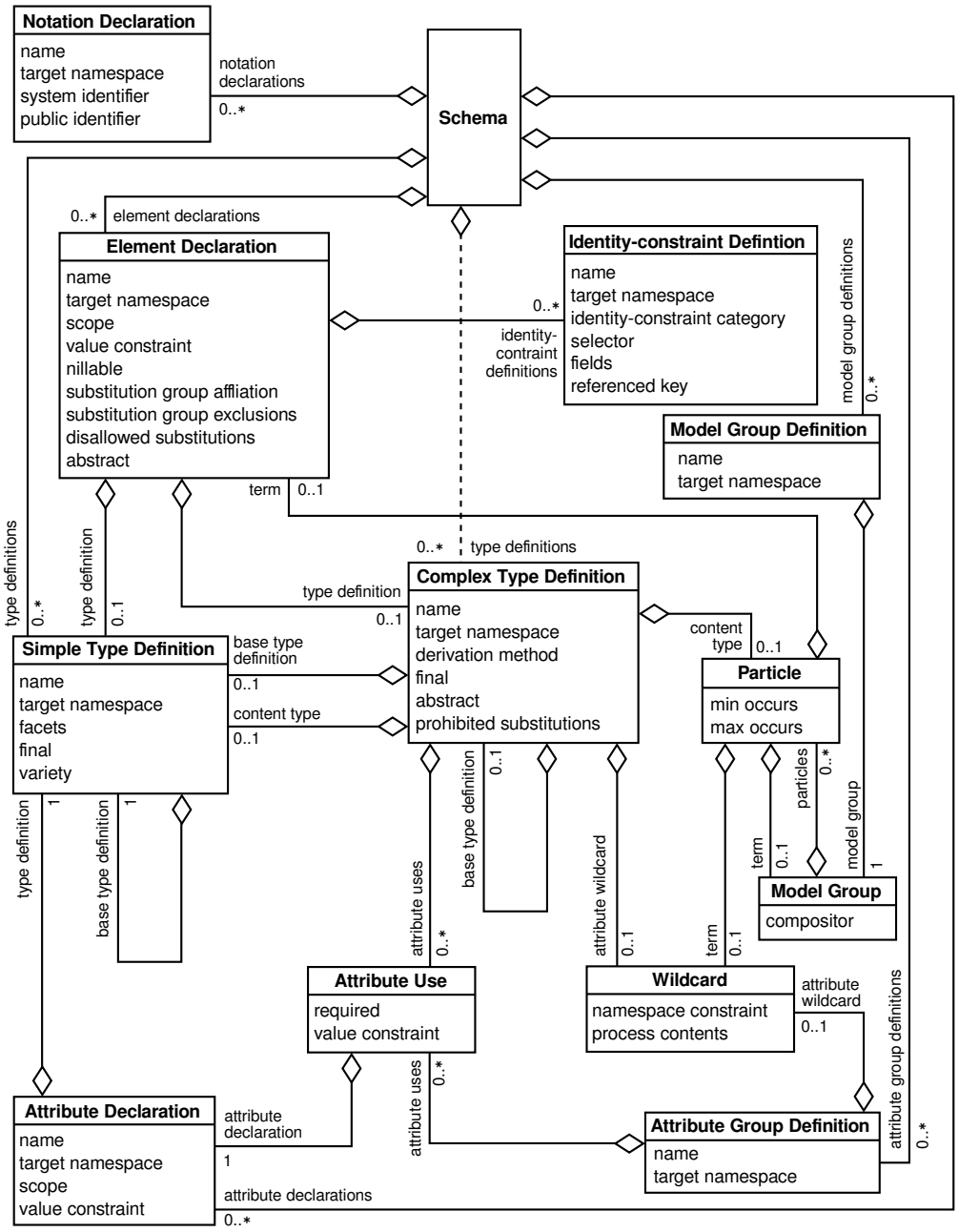


Abbildung 3.8. Das "XML Schema Component Model".[63]Nicht aufgeführt ist das *Annotation*-Modellelement, das an sehr vielen Stellen genutzt werden kann und das Diagramm unübersichtlich gemacht hätte.

3.4 Die ISO 19100 Normenfamilie

Ein gutes Beispiel für die Nutzung von UML als Sprache zur Definition eines konzeptuellen Schemas (→ 2.1.4 Konzeptuelles Schema) ist die ISO 19100 Normenfamilie einschließlich GML. Insbesondere im Bereich geographischer Informationssysteme spielen die nach diesen Vorgaben erstellten Modelle eine große Rolle. Daher soll im Rahmen dieser Arbeit auch auf einige dort definierte Spezialfälle eingegangen werden.

Die Norm ISO 19109 legt Regeln fest, mit deren Hilfe Anwendungsschemata (sogenannte "UML Application Schema") in einer einheitlichen Weise erstellt werden können.[64] Grundlage der Anwendungsschemata ist das General Feature Model (GFM). Das GFM definiert jedoch nur die Semantik des Metamodells, stellt aber keine konkrete Syntax zur Verfügung, mit der solche Schemata notiert werden können. In ISO 19103 wird UML als "conceptual schema language" ausgewählt.[65] Es werden Regeln für die Verwendung von UML aufgestellt und so ein UML-Profil definiert.

Die Einschränkungen, die in ISO 19103 gemacht werden, begrenzen die Menge der zu verwendenden UML-Modellelemente und ihre Verwendung. Es gibt auch einige Erweiterungen – besonders zu erwähnen sind hierbei die Stereotypen «CodeList» und «Union». Diese sind jedoch nicht unumstritten, wie weiter unten zu sehen ist. Die Norm ISO 19136 – GML greift die Einschränkungen auf und verstärkt sie zum Teil:

- Alle UML-Elemente haben die Sichtbarkeit "public".[66]
- Klassennamen innerhalb eines Modells müssen eindeutig sein.[67,68]
- Operationen werden ignoriert.[69].
- Klassen können entweder ein
  - "FeatureType", wenn sie mit dem Stereotyp «FeatureType» markiert sind, ein
  - "DataType", wenn sie mit dem Stereotyp «DataType» markiert sind, oder ein
  - "ObjectType", wenn sie nicht mit einem Stereotyp markiert sind

  sein.[70]
- Eine Generalisierung zwischen zwei Klassen ist nur dann zulässig, wenn beide Klassen "FeatureType", beide Klassen "ObjectType" oder beide Klassen "DataType" sind.[71]

[64] Vgl. ISO: Norm ISO 19109 Geographic information – Rules for application schema, Genf 2005.
[65] Vgl. ISO: Norm 19103.
[66] Vgl. OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.1.
[67] Vgl. ISO: Norm 19103, S. 31.
[68] Vgl. OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.2.
[69] Vgl. a. a. O.
[70] Vgl. a. a. O.
[71] Vgl. a. a. O.

- ▷ Eine Generalisierung zwischen Klassen darf nicht mit einem Stereotyp versehen sein.[72]
- ▷ Mehrfachvererbung ist nicht zulässig.[73,74]
  Während in ISO 19103 lediglich vom Einsatz von Mehrfachvererbung abgeraten wird (“*[. . .], the use of multiple inheritance should be minimized or avoided [. . .]*”[75]), wird sie im GML-Standard komplett ausgeschlossen: “*If a class is a specialization of another class, then this class shall have only one supertype (no support for multiple inheritance).*”[76]
- ▷ Jede Assoziation hat genau zwei Enden, die sich auf einen “FeatureType”, “ObjectType” oder “DataType” beziehen.[77,78]
- ▷ Mindestens ein Ende einer Assoziation muss benannt sein.[79]
- ▷ Assoziationen dürfen nicht mit Stereotypen versehen sein.[80]

Der GML-Standard spezifiziert ebenfalls eine XML-Kodierung für entsprechend eingeschränkte “UML Application Schemata”.[81] Mit Hilfe von in der GML-Spezifikation festgeschriebenen Regeln wird beschrieben, wie ein “UML Application Schema” in ein XML Schema umwandelt werden kann. Dabei werden Ideen des RDF aufgegriffen:[82]

- ▷ jede Klasse (bis auf Datentypen) wird über eine Kennung identifiziert,
- ▷ es wird eine Notation ähnlich zu Subject-Prädikat-Objekt verwendet, [83]
- ▷ es gibt eine gemeinsame Oberklasse `AbstractGMLType`.

Problemfall UML-Profil

Bei dem in der Norm ISO 19103 beschriebenen UML-Profil handelt es sich nicht um ein Profil im Sinne der UML-Profil-Definitionen der OMG. Das lässt sich darin begründen, dass das Profil zwei Stereotypen für Datentypen definiert, die auf Klassen angewendet werden. Wie Eisenhut und Kutzner feststellen, handelt es sich bei «CodeList» und «Union» nicht um semantikerhaltende Spezialisierungen. Vielmehr werden mit den beiden Stereotypen

[72] Vgl. OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.2.
[73] Vgl. ISO: Norm 19103, S. 9.
[74] Vgl. OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.2.
[75] ISO: Norm 19103, S. 9.
[76] OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.2.
[77] Vgl. ISO: Norm 19103, S. 28.
[78] Vgl. OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.3.
[79] Vgl. ISO: Norm 19103, S. 28.
[80] Vgl. OGC: GML Standard 3.2.1, Abschnitt E.2.1.1.3.
[81] A. a. O., S. 17: “*[. . .] conforming to both the constraints on such schemas and the rules for mapping them to GML application schemas specified in Annex E of this International Standard.*”
[82] A. a. O., S. 20: “*[. . .], GML follows RDF (W3C, 1999) terminology [. . .]*”
[83] A. a. O., S. 21: “*This encoding pattern is sometimes referred to as the object-property model and has been the basis of the GML encoding model [. . .]*”

markierte Klassen mit einer völlig neuen Semantik versehen – unter anderem werden sie als Datentypen verwendet. Das in der Norm ISO 19103 beschriebenen "UML-Profil" enthält diese beiden Stereotypen. Daher kann es kein gültiges UML-Profil sein, da es gegen die weiter oben im Abschnitt 3.1.4 zitierten Anforderungen aus der UML-Spezifikation verstößt. Somit ist auch jedes davon abgeleitete "UML-Profil" im engeren Sinn kein UML-Profil.[84]

Bei der Transformation der mit den fraglichen Stereotypen markierten Klassen spielt diese Beobachtung jedoch keine Rolle. **Im Rahmen dieser Arbeit wird an den Stellen, die sich mit Modellen gemäß der ISO 19100 Normenfamilie befassen, jeweils mit der neu definierten Semantik der so markierten Klassen gearbeitet.**

3.5 Meta-Object Facility (MOF)

3.5.1 Überblick

Die von der OMG standardisierte Meta-Object Facility (MOF) ist eine formale, visuelle Sprache (→ 2.1.7 Sprache) des MDA TS (→ 2.4.1 Model-Driven Architecture Technology Space). Es ist das wohl bekannteste Meta-Metamodell, das unter anderem für die Definition des UML-Metamodells verwendet wird. Das UML-Metamodell ist eine Instanz des MOF. Ebenfalls dient das MOF als Grundlage des XMI-Formats. Da auch das MOF selbst eine MOF-Instanz ist, handelt es sich beim MOF um ein sogenanntes ω-Metamodell – ein Metamodell, dessen Instanzen verschiedene Klassifikationslevel (→ 2.4.1 Model-Driven Architecture Technology Space) haben.

3.5.2 Das MOF Meta-Metamodell

Für MOF werden einige der Konzepte von UML wiederverwendet, so dass MOF insgesamt wesentlich weniger umfangreich als das UML-Metamodell ist. Abbildung 3.9 zeigt eine vereinfachte Version des MOF.

Instanzen des MOF-(Meta-Meta-)Elementtyps *Class* sind (Meta-)Elementtypen (→ 7 Elementtypen) in einem Metamodell. Diese besitzen einen Namen und können abstrakt (→ 7.4 Abstrakte Elementtypen) sein. Der in Abbildung 3.9 gezeigte Ausschnitt des MOF (der ebenfalls Instanz des MOF ist) enthält zwölf Instanzen von *Class*, von denen drei (Element, NamedElement und Classifier) abstrakt sind.

Instanzen der rekursiven Assoziation "subClass–superClass" sind *isA*-Beziehungen im Metamodell. Abbildung 3.9 enthält elf solcher Beziehungen, z.B. "NamedElement *isA* Element", "Class *isA* Classifier".

Bei Instanzen von *Association* handelt es sich im Metamodell um binäre Beziehungstypen (→ 9 Beziehungstypen), die ebenfalls benannt sein können. Abbildung 3.9 enthält folgende acht Beziehungstypen – als Linien ohne Pfeil oder Linien mit schwarzer Raute dargestellt:

[84] Vgl. Eisenhut/Kutzner: Vergleichende Untersuchungen zur Modellierun, Abschnitt 2.4.3.

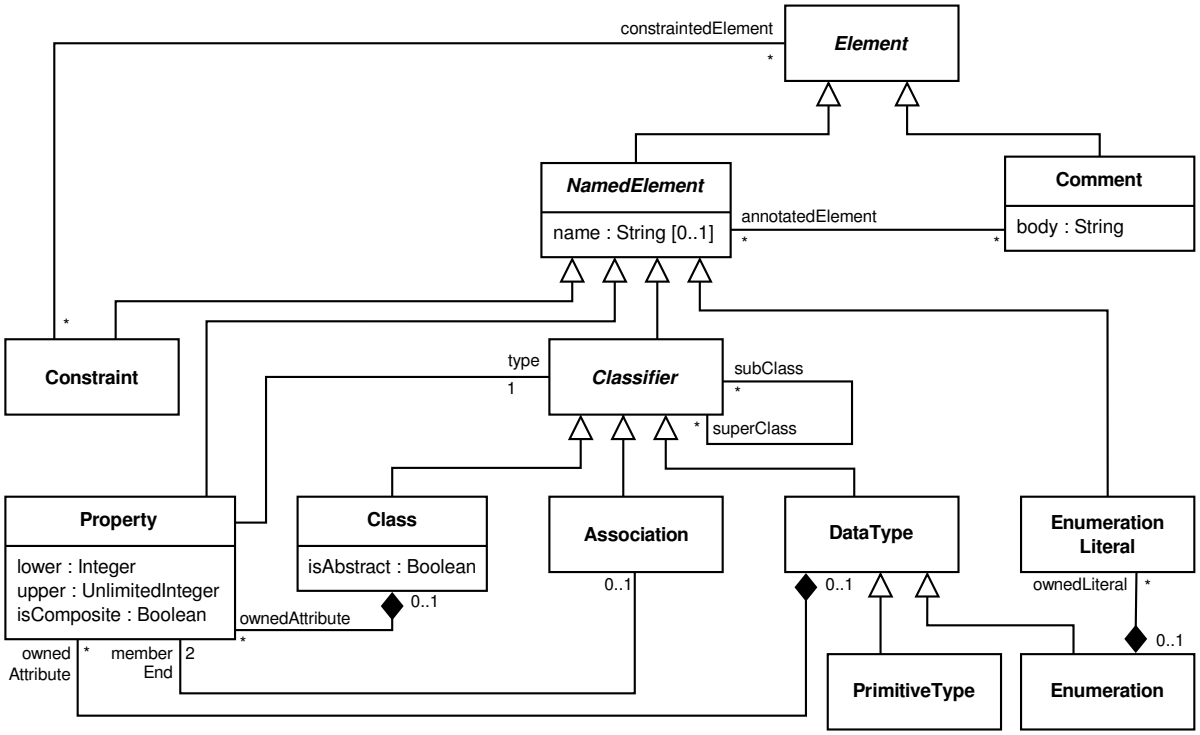


Abbildung 3.9. Vereinfachte Version des MOF.[85]

"constraintedElement–Constraint", "annotatedElement–Comment", "type–Property", "subClass–superClass", "ownedAttribute–Class", "ownedLiteral–Enumeration", "ownedAttribute–DataType" und "memberEnd–Association".[86]

Zu jedem Beziehungstypen im Metamodell gehören drei Instanzen des MOF: eine Instanz von *Association* und zwei Instanzen von *Property*. Die Namen der *Property*-Instanzen sind dabei die Bezeichnung der Rollen (→ 9 Beziehungstypen). Mit Hilfe der Werte der Attribute *lower* und *upper* können Kardinalitätsbeschränkungen (→ 9.4 Kardinalitätsbeschränkungen) angegeben werden. Ist der Wert des Attributs *isComposite* wahr, dann handelt es sich bei der Beziehung um eine Teil-Ganzes-Beziehungen (→ 9.6 Teil-Ganzes-Beziehungen). Dieser etwas komplizierte Zusammenhang wird in Abbildung 3.10 am Beispiel des Beziehungstyps "memberEnd–Association" aus dem MOF dargestellt.

Eine *Property* kann auch als *ownedAttribute* eines Elementtyps auftreten, ein Beispiel dafür in Abbildung 3.9 ist das klassenabhängige Attribut *isAbstract* des Elementtyps *Class* mit dem Namen "isAbstract".

Wie bei UML wird bei Datentypen zwischen primitiven Datentypen (*PrimitiveType*) und Aufzählungstypen (*Enumeration*) unterschieden. Aufzählungstypen bestehen aus mehreren *EnumerationLiterals*. In Abbildung 3.9 werden vier primitive Datentypen (String, Integer,

[85] Nach Olivé: Conceptual Modeling, S. 417.

[86] Angegeben ist – sofern vorhanden – der Name der *Property*, ansonsten der Name des als *type* verbundenen Elementtyps (*Classifier*).

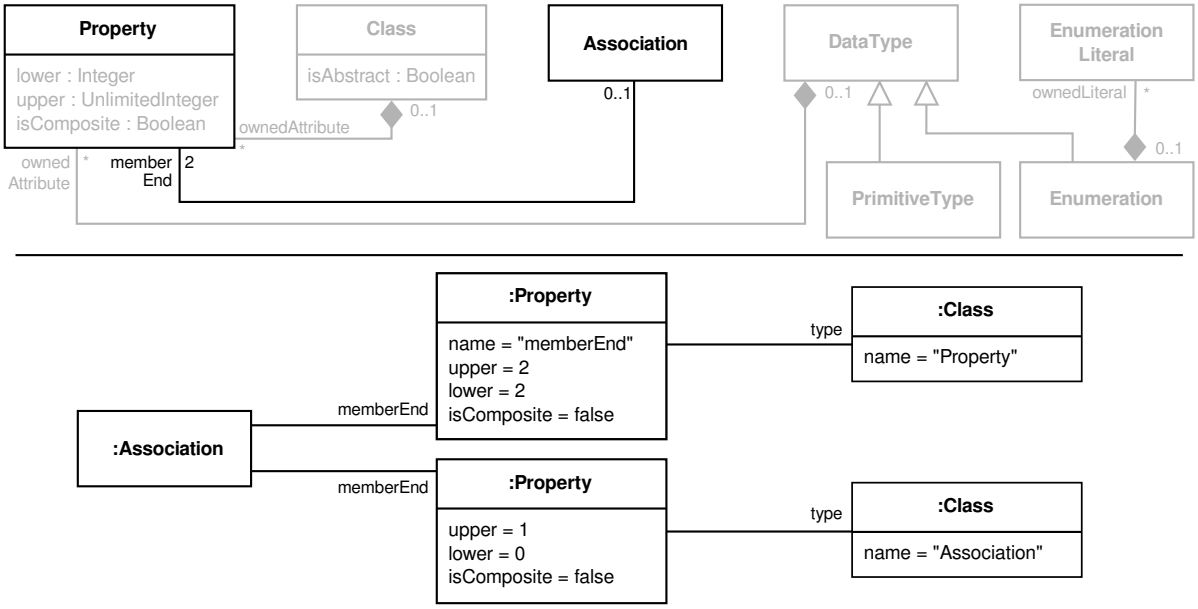


Abbildung 3.10. Beispiel für die Modellierung des Beziehungstyps "memberEnd–Association".

UnlimitedInteger und Boolean) verwendet, die jedoch nicht separat im Diagramm aufgeführt sind.

Instanzen von *Comment* lassen sich verwenden, um Instanzen von *NamedElement* mit Freitextkommentaren zu versehen. Der Text dieser Kommentare wird im Attribut *body* der jeweiligen *Comment*-Instanz abgelegt.

3.6 XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) ist ein Standard der OMG, der einen Austausch von Instanzen von MOF-Typen in Form von XML-Dokumenten beschreibt.[87] Dabei können mit Hilfe von XMI sowohl Modelle als auch Instanzen von Modellen kodiert werden. Eine komplette Darstellung von XMI würde den Rahmen dieser Arbeit sprengen, daher werden hier nur einige wichtige Grundlagen erläutert. Für detaillierte Informationen sei neben dem Standard selbst auf weiterführende Literatur verwiesen.[88]

XMI beschreibt, wie Instanzen und ihre Beziehungen als XML-Elemente und -Attribute repräsentiert werden. In vielen Fällen hat der Nutzer dabei jedoch Freiheiten, verschiedene gleichwertige Möglichkeiten der Repräsentation zu wählen. So kann es zu einem Modell mehrere unterschiedliche XML-Dokumente geben – später wird dies an einem Beispiel in den Listings 4.1 und 4.2 gezeigt. Einen Ausschnitt des zweiten Beispiels zeigt Listing 3.4:

[87] Siehe OMG: MOF 2 XMI Mapping Specification, 2001 ⟨URL: `http://www.omg.org/spec/XMI/`⟩.

[88] U.a. Grose, T.J./Doney, G.C./Brodsky, S.A.: Mastering XMI: Java Programming with XMI, XML, and UML, New York 2002.

```
1 <xmi:XMI xmi:version="2.1">
2
3   <uml:Package xmi:id="1" name="Library">
4       <packagedElement xmi:type="uml:Class" xmi:id="7" name="Author">
5           <generalization xmi:id="15" general="5" />
6       </packagedElement>
7       <packagedElement xmi:type="uml:Class" xmi:id="11" name="Book" generalization="17"/>
8   </uml:Package>
9
10  <uml:Generalization xmi:id="17" general="8" specific="11" />
11
12  <uml:Property xmi:id="9" name="title">
13      <type xmi:type="uml:PrimitiveType" href="UMLPrimitiveTypes.library.uml#String" />
14  </uml:Property>
15 </xmi:XMI>
```

Listing 3.4. Beispiel für ein XMI-Dokument – (in sich unvollständiger) Ausschnitt aus Listing 4.2.

Wurzelelement eines jeden XMI-Dokumente ist `xmi:XMI`. Das obligatorische XML-Attribut `xmi:version` gibt die verwendete Version des XMI-Standards an.

Jede Instanz eines Elementtyps wird durch ein XML-Element repräsentiert. Um Beziehungen zwischen Instanzen herstellen zu können, müssen die XML-Elemente identifiziert werden können. Dafür sieht XMI drei XML-Attribute vor. Das wichtigste davon ist das `xmi:id`-Attribut. Da sein Wert vom XSD-Datentyp `ID` ist, ist sichergestellt, dass jeder Wert nur einmal pro XML-Dokument vorkommt und somit jedes XML-Element innerhalb dieses Dokuments eindeutig referenziert werden kann.

Zur Übermittlung der Information, dass ein durch ein XML-Elements repräsentiertes Objekt *e* Instanz eines Elementtyps *E* ist, gibt es bei XMI mehrere Möglichkeiten:

- Der Name des Elementtyps *E* wird als Name des XML-Elements verwendet. In Listing 3.4 ist dies in den Zeilen 3, 10 und 12 zu sehen.
- Der Elementtyp ergibt sich durch Wissen des Metamodells und den Angaben einer Beziehung, in der das Objekt *e* steht. Zeile 5 zeigt ein solches Beispiel. Im Metamodell ist eindeutig, dass an dieser Stelle nur Instanzen eines Elementtyps auftreten können.
- Mit Hilfe des XML-Attributs `xmi:type` wird der Elementtyp explizit angegeben. Dies ist in den Zeilen 4, 7 und 13 zu sehen.

Beziehungen zwischen Instanzen können mit Hilfe von XML-Attributen dargestellt werden, deren Wert die schon erwähnten `xmi:id` referenzieren (Zeilen 5, 7 und 10). Als Name des XML-Attributs werden dabei die Rollennamen des Beziehungstyps (→ 9 Beziehungstypen) verwendet. Die durch die XML-Element in den Zeilen 7 und 10 repräsentierten Instanzen stehen in Beziehung, die Rollennamen sind `generalization` (Zeile 7) sowie `specific` (Zeile 10).

Eine alternative Repräsentation für Beziehungen ist in den Zeilen 4, 5 und 7 zu sehen: Das XML-Element, das das in Beziehung stehende Objekt repräsentiert, wird als Kindelement eingefügt. Als Name des XML-Elements wird der Name der Rolle verwendet, in der das Objekt auftritt.

Auch der Bezug auf Objekte, die in einem externen XML-Dokument aufgeschrieben wurden, ist möglich. In diesem Fall muss ein Kindelement für die Repräsentation der Beziehung verwenden werden. Bei diesem Kindelement gibt das `href`-Attribut an, wo die externe Repräsentation zu finden ist. Dies ist in Zeile 13 zu sehen, wo auf einen Datentyp in der UML-Bibliothek mit primitiven Datentypen Bezug genommen wird.

3.7 QVT

Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)[89] ist eine von der OMG entwickelte Sprache für Modell-zu-Modell-Transformationen im MDA TS (→ 2.4.1 Model-Driven Architecture Technology Space). Die Spezifikation der Sprache greift auf MOF und die Object Constraint Language (OCL) zurück. QVT-Transformationen können sowohl deklarativ (QVT Relations (QVT-R)) als auch imperativ (Operational QVT) beschrieben werden. QVT-R besitzt sowohl eine textuelle als auch eine visuelle konkrete Syntax (→ 2.1.7 Sprache).

Mit QVT lassen sich Instanzen von Meta-Modellen (die selbst wiederum Instanzen des MOF sein müssen) überprüfen sowie transformieren. Abbildung 3.11 gibt einen Überblick über die Artefakte, die bei einer QVT-Modelltransformation eine Rolle spielen.

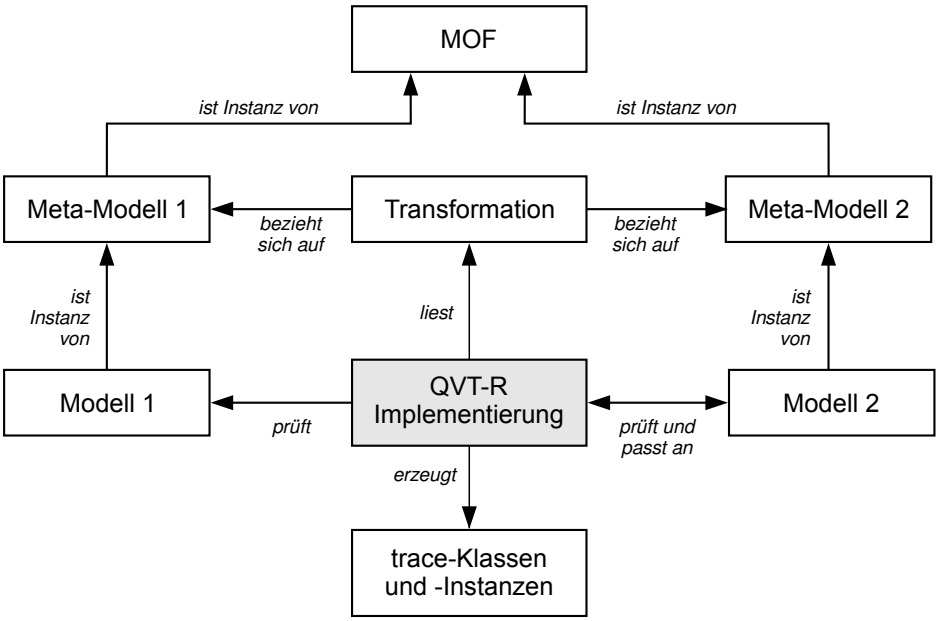


Abbildung 3.11. Übersicht über eine Modelltransformation mit QVT.

[89] Siehe OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0, 2008 ⟨URL: `http://www.omg.org/spec/QVT/1.0/PDF/`⟩.

Während der Ausführung einer QVT-R Transformation werden sogenannte *trace-Klassen* (engl. *trace class*) und deren Instanzen (engl. *trace instance*) geschrieben.[90] Mit Hilfe beider lässt sich später nachvollziehen, wie die Transformation abgelaufen ist. Auch in dieser Arbeit werden die trace-Klassen verwendet, um die entwickelten Transformationsregeln zu überprüfen.

Im Folgenden werden kurz die Semantik und die grafische Syntax der QVT-R erläutert. Für eine ausführlichere Einführung in QVT-R sei auf weiterführende Literatur verwiesen.[91]

Transformation – Mit Hilfe einer Transformation wird die Prüfung bzw. die Transformation von Modellen beschrieben. Neben einem Namen, der die Transformation identifiziert, enthält sie ein oder zwei Metamodelle als Parameter.[92] Ist nur ein Metamodell angegeben, so lässt sich mit Hilfe der Transformation ein Modell (=eine Instanz dieses Metamodells) auf Korrektheit überprüfen. Bei zwei Metamodellen lassen sich mit Hilfe der Transformation Umwandlungen zwischen Modellen (=Instanzen der Metamodelle) durchführen. Eine Transformation enthält Keys, Relations sowie Queries.

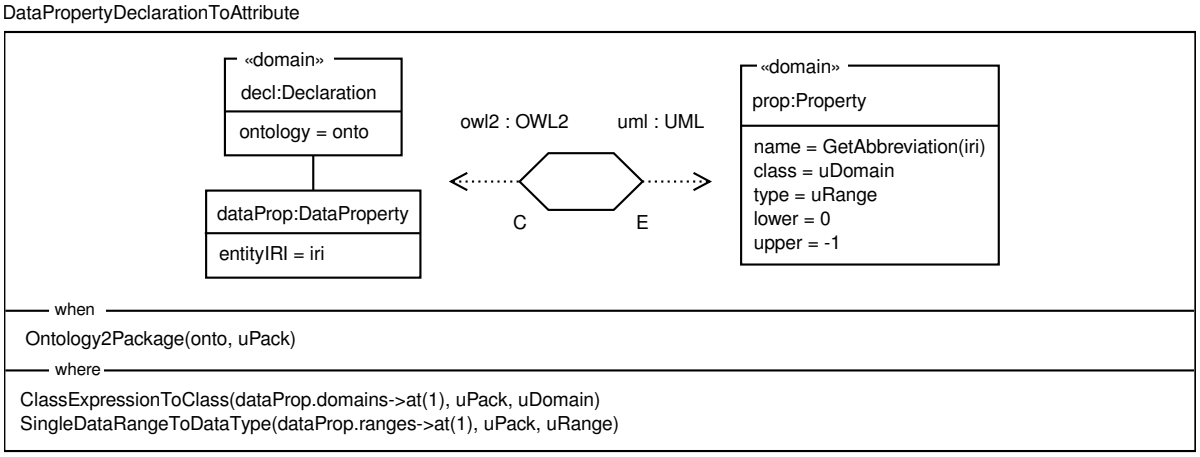


Abbildung 3.12. Beispiel für die grafische Repräsentation einer QVT-R-Relation.

Relation – Eine Relation (Schlüsselwort `relation`) beschreibt den Zusammenhang zwischen Elementen der verschiedenen Metamodelle. Dabei ist zwischen Top-Relationen (in der textuellen Syntax mit dem Schlüsselwort `top` gekennzeichnet) und anderen Relationen zu unterscheiden: Top-Relationen werden auf alle passenden Modellelemente angewendet, andere Relationen nur in dem Fall, dass sie von anderen Relationen aus referenziert werden. Jede Relation besitzt einen Namen und enthält Variablen, Domain(s), eine `when`- und/oder eine `where`-Klausel. Abbildung 3.12 zeigt ein Beispiel der grafischen Syntax für eine Relation.

[90] Vgl. OMG: QVT Specification 1.0, S. 5 und S. 9.

[91] U.a. Nolte, S.: QVT-Relations Language, Berlin/Heidelberg 2009.

[92] Theoretisch ist es auch möglich, mehr als zwei Metamodelle gleichzeitig zu verwenden. Dies spielt jedoch bei dieser Arbeit keine Rolle.

Domain – Mit Hilfe der Angabe einer Domain wird der Bezug zu einem Element eines Metamodells hergestellt. Das Metamodell, auf das sich eine Domain bezieht, wird in der grafischen Syntax oberhalb des Sechsecks angegeben. Domains können entweder "checkonly" oder "enforce" sein (Schlüsselwörter `checkonly` und `enforce` in der textuellen Syntax, Buchstaben C und E unterhalb des Sechsecks in der grafischen Syntax). Im Beispiel in Abbildung 3.12 bezieht sich die linke `checkonly`-Domain auf das OWL-Metamodell, die rechte `enforce`-Domain auf das UML-Metamodell. Bei einer `checkonly`-Domain, die sich sowohl auf das Quell- als auch auf das Zielmetamodell beziehen kann, wird geprüft, ob das angegebene Muster der Domain in dem Modell gefunden wird. Ist dies nicht der Fall, wird die Ausführung der gesamten Relation verhindert. Eine `enforce`-Domain kann sich nur auf das Ziel-Metamodell beziehen. Hier wird nicht nur geprüft, ob das angegebene Muster der Domain in dem Modell auftritt, sondern es werden ggf. noch fehlende Elemente ergänzt oder geändert. Die Kombination von `checkonly`- und `enforce`-Domains ermöglicht die Modell-zu-Modell-Transformation. QVT-R folgt dem "check before enforce"-Prinzip, d.h. es werden zunächst alle `checkonly`-Domains einer Relation überprüft, bevor Änderungen an einer `enforce`-Domain vorgenommen werden.

when-Klausel – Eine Relation kann eine `when`-Klausel beinhalten, die eine beliebige Anzahl von OCL-Ausdrücken umfassen kann. Durch eine `when`-Klausel lassen sich – zusätzlich zu der Überprüfung des Domain-Musters – Bedingungen festlegen, unter denen die Relation angewendet wird. Ein übliches Vorgehen ist es beispielsweise, dass die Anwendung der Relation an die Ausführung bestimmter `top`-Relationen geknüpft ist, um so sicherzustellen, dass Quell- und Ziel-Elemente jeweils Teil passender anderer Elemente sind. Die Relation wird nur dann angewendet, wenn alle OCL-Ausdrücke der `when`-Klausel zu "wahr" ausgewertet werden können.

where-Klausel – Eine Relation kann eine `where`-Klausel beinhalten, die eine beliebige Anzahl von OCL-Ausdrücken umfassen kann. Mit Hilfe der `where`-Klausel ist es möglich, Relationen einzubeziehen, die nicht als `top` markiert sind, sowie – z.B. mit Hilfe von Queries – Variablen zu belegen. Es ist notwendig, dass alle OCL-Ausdrücke einer `where`-Klausel zu "wahr" ausgewertet werden, andernfalls schlägt die Transformation mit einem Laufzeitfehler fehl. Eine `where`-Klausel kommt dann zur Anwendung, wenn Änderungen im Ziel-Modell vorgenommen werden müssen. Mit Hilfe der `where`-Klauseln lassen sich Relationen rekursiv verschachteln.

Primitive Domain – Zusätzlich zu den bereits beschriebenen Domains können in einer Relation auch primitive Domains (Schlüsselwort `primitive domain`) angegeben werden, die kein Domain-Muster besitzen und auch nicht an ein Metamodell gebunden sind. Sie werden verwendet, um der Relation Parameter zu übergeben. Eine Primitive Domain kann daher nicht in einer Top-Relation auftreten.

Query – Eine Query (Schlüsselwort `query`) ist eine in der OCL geschriebene Hilfsfunktion. Eine Query besitzt einen Namen, eine beliebige Anzahl getypter Parameter sowie einen Rückgabetyp. Queries können innerhalb von Domains sowie `when`- und `where`-Klauseln verwendet werden.

Key – In einer Transformation lassen sich Keys definieren. Ein Key (Schlüsselwort `key`) gibt an, mit Hilfe welcher Eigenschaften Modellelemente identifiziert werden können. Haben zwei Modellelemente identische Werte für ihre identifizierenden Eigenschaften, so werden sie als identisch angesehen und daher im Ziel-Modell nicht doppelt angelegt. Ein Beispiel für eine identifizierende Eigenschaft ist bei UML-Klassen die Kombination aus Paket und Name.

Variable – In einer Relation können (in der textuellen Syntax vor dem ersten Auftreten einer Domain) getypte Variablen deklariert werden. Ihnen kann durch die `checkonly`-Domain oder eine `where`-Klausel ein Wert zugewiesen werden. Dieser Wert kann dann in weiteren Klauseln oder der `enforce`-Domain verwendet werden.

Kapitel 4

Konzept/Idee

In diesem Kapitel sollen die grundlegenden Ideen einer Transformation konzeptueller Modelle auf Metamodell-Ebene mit Hilfe der deklarativen QVT-R vorgestellt werden. Zunächst wird begründet, warum das MOF es ermöglicht, zwischen den verschiedenen Technologieräumen (→ 2.4 Technologieraum) zu transformieren. Anschließend wird dargestellt, welche Vorteile eine Transformation auf Metamodell-Ebene gegenüber Transformationen auf syntaktischer Ebene hat. Im dritten Abschnitt wird die Wahl der Transformationssprache QVT-R motiviert. Abgeschlossen wird das Kapitel mit einer Übersicht über den gesamten Transformationsprozess einschließlich der Punkte, die beim Übergang von konkreter zu abstrakter Syntax und umgekehrt zu beachten sind.

4.1 MOF als Basis

Um zwischen OWL- und UML-Modellen transformieren können, ist es notwendig, ein "gemeinsames Drittes" zu finden. Bei diesem gemeinsamen Dritten kann es sich um ein gemeinsames Transferformat, d.h. eine gemeinsame Syntax, oder ein gemeinsames Metamodell handeln, das es gestattet, die Sprachelemente der einen Sprache syntaxunabhängig auf korrespondierende Sprachelemente der anderen Sprache abzubilden.

Betrachten wir nun zuerst die beiden Technologieräume MDA TS mit der Sprache UML und Ontology TS mit der Sprache OWL 1. Hier gibt es kein gemeinsames Metamodell, da das UML-Metamodell in MOF spezifiziert ist, OWL 1 jedoch durch Resource Description Framework Schema (RDFS) in einer funktionalen Architektur beschrieben wird: "*... RDFS, as a schema layer language, has a non-standard and non-fixed-layer metamodeling architecture, which makes some elements in the model have dual roles in the RDFS specification.*"[93] Aufgrund dieses Unterschiedes zwischen einer MOF-basierten Architektur auf der einen und einer funktionalen Architektur auf der anderen Seite, können Transformationen zwischen dem UML-Metamodell und dem RDF-Schema von OWL 1 nicht direkt definiert werden, sondern benötigen ein Transferformat. Die in Kapitel 5 vorgestellten Arbeiten machen es sich zunutze, dass sowohl für UML als auch für RDFS bzw. OWL XML-Serialisierungen existieren.

Ein zentraler Unterschied zwischen OWL 2 und der Vorgängerversion ist es, dass OWL 2 ein MOF-kompatibles Metamodell besitzt: "*The structural specification is defined using the Unified*

[93] Pan, J.Z./Horrocks, I.: Metamodeling architecture of web ontology languages, in: Cruz, I.F. et al. (Hrsg.): The Emerging Semantic Web: selected papers from the first Semantic Web Working Symposium, Amsterdam 2002.

Modeling Language (UML), and the notation used is compatible with the Meta-Object Facility (MOF)."[94] Damit können OWL 2-Ontologien nicht nur in Form serialisierter XML-Dokumente, sondern auch als auf MOF basierende Modelle verarbeitet werden. Es können daher alle Werkzeuge, die im Rahmen von MOF für Modelltransformation zur Verfügung stehen und für UML-Modelle sowieso schon genutzt werden konnten, auch für Ontologien verwendet werden.

Da mit dem XML Schema Component Model auch für XML Schema ein MOF-kompatibles Metamodell bereitsteht, eignet sich der in dieser Arbeit vorgestellt Ansatz auch für Transformationen zwischen UML und XSD. Damit könnten auch diese Transformationen mit den gleichen, hier vorgestellten, Analyseverfahren untersucht werden.

4.2 Verarbeitung auf syntaktischer Ebene?

Da OWL-Ontologien in einer konkreten Syntax geschrieben werden können, und sich MOF-basierte Modelle mit Hilfe von XMI (→ 3.6 XML Metadata Interchange (XMI)) serialisieren lassen, ist es denkbar, eine modellbasierte Transformation (→ 2.3.1 Begriff "Modelltransformation") auf syntaktischer Ebene durchzuführen. Eine solche Transformation kann z.B. mit XSLT durchgeführt werden, wenn die Transferformate XML-basiert sind.

Im folgenden Abschnitt soll ein solches Vorgehen skizziert werden. Anschließend werden einige Nachteile einer Verarbeitung auf syntaktischer Ebene dargestellt. Als Beispiel soll das in der unteren Hälfte der Abbildung 4.1 gezeigte UML-Klassendiagramm des User-Modells auf Schicht M1 dienen. Bei den dort gezeigten Modellelementen handelt es sich um Instanzen des im oberen Bereich der Abbildung gezeigten (vereinfachten) UML-Metamodells auf Schicht M2.

Um neben der visuellen Syntax von Abbildung 4.1 eine Darstellung in konkreter textuellen Syntax zu haben, zeigt Listing 4.1 eine XMI-Repräsentation des User-Modells.

Für eine Verarbeitung eines solchen Modells mit Hilfe der XMI-Serialisierung müssten grob skizziert z.B. folgende Regeln definiert werden:

- Für jedes XML-Element mit dem Namen `uml:Class` erzeuge ein passendes Element im Zieldokument. Aus dem Attribut `name` generiere einen passenden Namen. Suche `uml:Property`-Elemente, die eine `xmi:id` haben, die im Attribut `ownedAttributes` enthalten ist, und verarbeite sie geeignet.

- Für jedes XML-Element mit dem Namen `uml:DataType` erzeuge ein passendes Element im Zieldokument. Aus dem Attribut `name` generiere einen passenden Namen. Suche `uml:Property`-Elemente, die eine `xmi:id` haben, die im Attribut `ownedAttributes` enthalten ist, und verarbeite sie geeignet.

- Für jedes XML-Element mit dem Namen `uml:Association` erzeuge ein Element im Zieldokument. Aus dem Attribut `name` generiere einen passenden Namen. Suche `uml:Property`-

[94] W3C: OWL 2 Structural specification, Abschnitt 2.1.

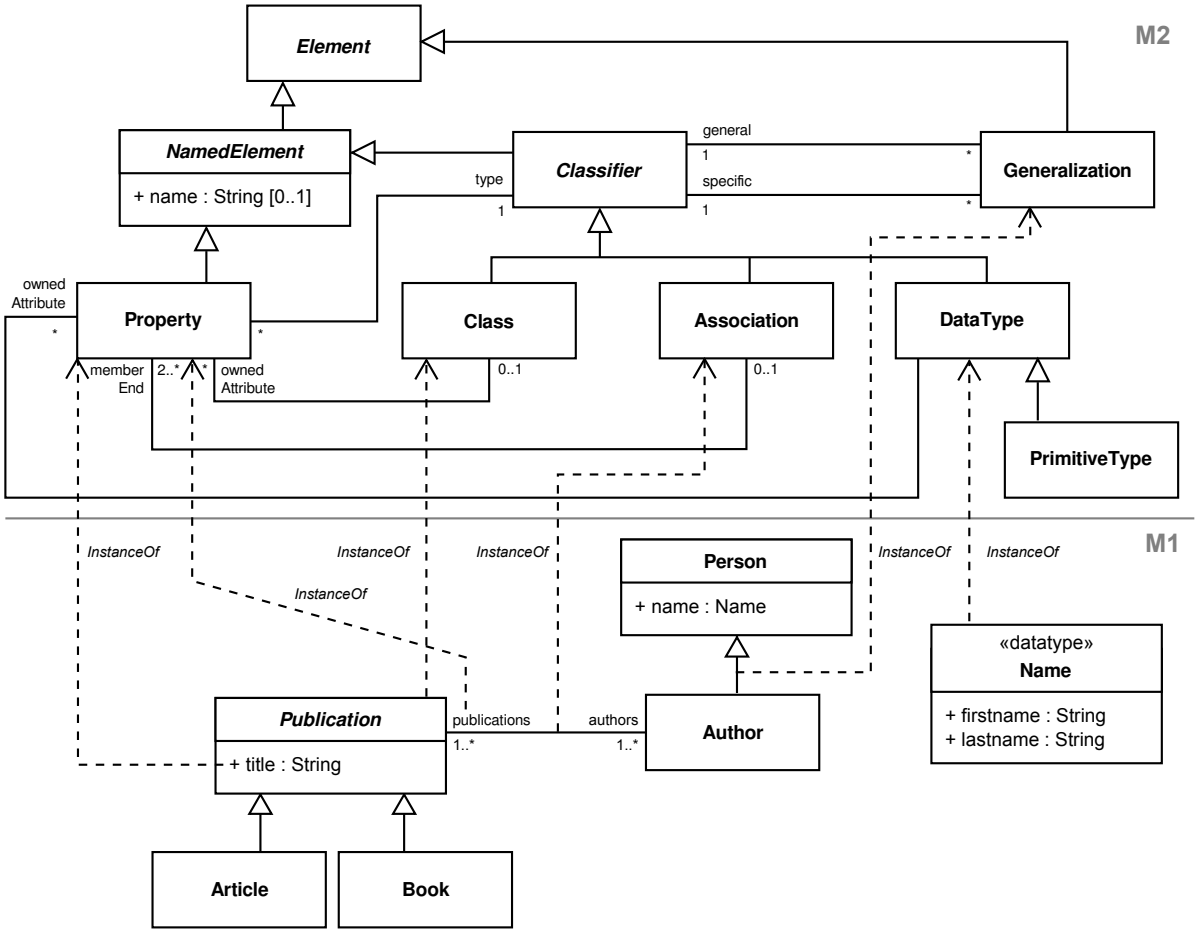


Abbildung 4.1. Beispiel eines User-Modells auf Schicht M1 als Instanz des UML-Metamodells auf Schicht M2.

Elemente, die eine `xmi:id` haben, die im Attribut `memberEnd` enthalten ist, und verarbeite sie geeignet.

- ▷ Für jedes XML-Element mit dem Namen `uml:Generalization` prüfe, ob das Element mit der `xmi:id`, die im Attribut `general` steht, den Namen `uml:Class`, `uml:DataType` oder `uml:Association` hat. Erzeuge passende Konstrukte im Zieldokument.
- ▷ Für jedes XML-Element mit dem Namen `uml:Property` generiere einen passenden Namen aus dem Attribut `name`. Prüfe, ob das Element mit der `xmi:id`, die im Attribut `type` steht, den Namen `uml:Class`, `uml:DataType` oder `uml:PrimitiveType` hat. Erzeuge passende Konstrukte im Zieldokument.

Wie zu erkennen ist, sind einige Anweisungen mehrfach vorhanden, wie z.B. die Behandlung des Attributs `name` oder die Überprüfung, um welchen Elementnamen es sich handelt. Dies ist der Tatsache geschuldet, dass bei einer modellbasierten Transformation auf syntakti-

```
 1 <xmi:XMI xmi:version="2.1">
 2    <uml:Package  xmi:id="1"  name="Library" packagedElement="2 5 7 8 10 11 12"/>
 3    <uml:DataType xmi:id="2"  name="Name" ownedAttribute="3 4" />
 4    <uml:Property xmi:id="3"  name="firstname" type="18" dataType="2" />
 5    <uml:Property xmi:id="4"  name="lastname"  type="18" dataType="2"/>
 6    <uml:Class    xmi:id="5"  name="Person" ownedAttribute="6" />
 7    <uml:Property xmi:id="6"  name="name" type="2" class="5" />
 8    <uml:Class    xmi:id="7"  name="Author"  generalization="15"/>
 9    <uml:Class    xmi:id="8"  name="Publication" isAbstract="true" ownedAttribute="9"/>
10    <uml:Property xmi:id="9"  name="title" type="18" class="8" />
11    <uml:Class    xmi:id="10" name="Article" generalization="16"/>
12    <uml:Class    xmi:id="11" name="Book"    generalization="17"/>
13    <uml:Association xmi:id="12" name="Asso1" memberEnd="13 14" />
14    <uml:Property xmi:id="13" name="publications" type="8" association="12" />
15    <uml:Property xmi:id="14" name="authors"      type="7" association="12" />
16    <uml:Generalization xmi:id="15" general="5" specific="7" />
17    <uml:Generalization xmi:id="16" general="8" specific="10" />
18    <uml:Generalization xmi:id="17" general="8" specific="11" />
19    <uml:PrimitiveType  xmi:id="18" href="UMLPrimitiveTypes.library.uml#String" />
20 </xmi:XMI>
```

Listing 4.1. XMI-Repräsentation der Instanzen aus Abbildung 4.1.

scher Ebene nicht auf die Zusammenhänge im Metamodell zurückgegriffen werden kann, wie z.B. die Information, dass es sich bei den UML-Elementtypen *Property*, *Class*, *Association* und *DataType* um ein *NamedElement* handelt und diese damit alle ein klassenabhängiges Attribut *name* besitzen.

Neben dieser allgemeinen Schwierigkeit, die bei modellbasierter Transformation auftritt, bei der nur Elemente einer Modell-Schicht (und nicht zusätzlich die der Metamodell-Schicht) zur Verfügung stehen, treten sowohl bei XMI-Dokumenten als auch bei den XML-basierten Syntaxen von OWL weitere Probleme auf, von denen zwei im Folgenden vorgestellt werden. Auch bei Syntaxen, die nicht XML-basiert sind, sind ähnliche Schwierigkeiten denkbar.

Unterschiedliche Repräsentation in XML-Dokumenten

Sowohl XMI-Dokumente als auch die XML-basierten Syntaxen von OWL haben die Eigenschaft, dass XML-Dokumente desselben Modells sehr unterschiedlich aussehen können – es besteht eine große Freiheit bei der Gestaltung der XML-Dokumente.

Obwohl Listing 4.1 und Listing 4.2 das gleiche in Abbildung 4.1 (unten) gezeigte Modell beschreiben, so sind sie schon auf den ersten Blick verschieden. Anhand von Listing 4.2 werden zwei Beispiele für unterschiedliches Aussehen eigentlich gleicher Modellelemente innerhalb eines XMI-Dokuments näher beschrieben:

- Beim Vergleich der Zeilen 13–15 mit Zeile 21 fällt auf, dass im ersten Fall das klassenabhängige Attribut als verschachteltes XML-Element `ownedAttribute` ohne Angabe von `xmi:type`, im zweiten Fall als XML-Attribut `ownedAttribute` mit einem Verweis auf die in den Zeilen 38-40 definierte Instanz des UML-Elementtyps *Property* geschrieben wurde.
- Der Vergleich der Zeilen 23–25 mit der Zeile 27 zeigt, dass die Vererbungsbeziehung im

ersten Fall als verschachteltes Element `generalization` ausgeführt wurde, im zweiten Fall als XML-Attribute mit Verweis auf das Element in Zeile 32. Beim verschachtelten Element ist keine Angabe `xmi:type` vorhanden. Ebenfalls ist die Referenz auf den spezifischen Typ (in Form des XML-Attributs `specific`) hier nicht vorhanden.

```
1 <xmi:XMI xmi:version="2.1">
2   <uml:Package xmi:id="1" name="Library">
3
4      <packagedElement xmi:type="uml:DataType" xmi:id="2" name="Name">
5          <ownedAttribute xmi:id="3" name="firstname">
6            <type xmi:type="uml:PrimitiveType" href="UMLPrimitiveTypes.library.uml#String" />
7          </ownedAttribute>
8          <ownedAttribute xmi:id="4" name="lastname">
9            <type xmi:type="uml:PrimitiveType" href="UMLPrimitiveTypes.library.uml#String" />
10         </ownedAttribute>
11     </packagedElement>
12
13     <packagedElement xmi:type="uml:Class" xmi:id="5" name="Person">
14         <ownedAttribute xmi:id="6" name="name" type="2"/>
15     </packagedElement>
16
17     <packagedElement xmi:type="uml:Class" xmi:id="7" name="Author">
18         <generalization xmi:id="15" general="5" />
19     </packagedElement>
20
21     <packagedElement xmi:type="uml:Class" xmi:id="8" name="Publication" isAbstract="true"
           ownedAttribute="9" />
22
23     <packagedElement xmi:type="uml:Class" xmi:id="10" name="Article">
24         <generalization xmi:id="16" general="8"  />
25     </packagedElement>
26
27     <packagedElement xmi:type="uml:Class" xmi:id="11" name="Book" generalization="17"/>
28
29     <packagedElement xmi:type="uml:Association" name="Asso1" xmi:id="12" memberEnd="13 14" />
30   </uml:Package>
31
32   <uml:Generalization xmi:id="17" general="8" specific="11" />
33
34   <uml:Property xmi:id="13" name="publications" type="8" association="12" />
35
36   <uml:Property xmi:id="14" name="authors" type="7" association="12" />
37
38   <uml:Property xmi:id="9" name="title">
39       <type xmi:type="uml:PrimitiveType" href="UMLPrimitiveTypes.library.uml#String" />
40   </uml:Property>
41 </xmi:XMI>
```

Listing 4.2. Alternative XMI-Repräsentation der Instanzen aus Abbildung 4.1.

Ähnliche Freiheit bei der Strukturierung gibt es auch bei XML-Schema, wo derselben Sachverhalt in drei verschiedenen *"flavours"* aufgeschrieben werden kann: *embedded types*, *named types* und *flat catalogue*.[95]

[95] Vgl. Luttenberger, N.: Grammars for XML Documents - XML Schema, Part 1 – Vorlesung "XML in Communication Systems"(WS 2011/12), Kiel 2011, S. 51.

Identische Namen für unterschiedliche Elemente

```
1 <rdf:RDF>
2   <owl:Class rdf:about="Father">
3     <rdfs:subClassOf>
4       <owl:Class>
5         <owl:intersectionOf rdf:parseType="Collection">
6           <owl:Class rdf:about="Man">
7             <rdfs:subClassOf rdf:resource="Person" />
8           </owl:Class>
9           <owl:Class rdf:about="Parent" />
10        </owl:intersectionOf>
11      </owl:Class>
12    </rdfs:subClassOf>
13  </owl:Class>
14 </rdf:RDF>
```

Listing 4.3. Ontologie in RDF/XML Syntax (`rdf:RDF`-Element gekürzt).

In Listing 4.3 wird eine weitere Schwierigkeit bei der Arbeit mit XML-Serialisierungen – in diesem Fall von OWL-Ontologien – deutlich. Sowohl in Zeile 2 als auch in Zeile 4 findet sich ein XML-Element namens `owl:Class`. Während das XML-Element in Zeile 2 eine Instanz des OWL-Elementtyps *Class* beschreibt, korrespondiert das XML-Element in Zeile 4 nicht direkt mit einem Modell-Element. Notwendig wird das zusätzliche XML-Element `owl:Class` lediglich durch die Vorgaben des RDF/XML-Formats, das es nicht erlaubt, das XML-Element `owl:intersectionOf` als Kindelement von `rdfs:subClassOf` zu verwenden.

Zusammenfassend lässt sich sagen, dass die Anweisungen modellbasierter Transformationen auf syntaktischer Ebene aufgrund der aufgezeigten Schwierigkeiten zahlreiche Sonderfälle enthalten, um diese Art von Mehrdeutigkeiten zu bearbeiten. Die Regeln werden dadurch umfangreicher und schwerer zu verstehen.

4.3 Verarbeitung auf Metamodell-Ebene

Im Gegensatz zur modellbasierten Transformation, die auf syntaktischer Ebene arbeitet, besteht bei einer Modelltransformation (→ 2.3.1 Begriff "Modelltransformation") sowohl Zugriff auf die Modelle als auch auf die Metamodelle, da sich die Transformationsregeln auf das Metamodell beziehen. Bei der Erstellung der Transformationsregeln werden ausschließlich die Metamodelle betrachtet – daher auch der Begriff "Verarbeitung auf Metamodell-Ebene". Es muss beim Schreiben der Transformationsregeln noch nicht bekannt sein, welche Modelle später mit den Regeln transformiert werden sollen. Jedes Modell, das konform zum (Eingabe-)Metamodell ist, kann mit Hilfe dieser Transformationsregeln verarbeitet werden.

Wird die Verarbeitung auf Basis der Metamodelle beschrieben, so stehen nicht nur die Namen der Element- und Beziehungstypen zur Verfügung, sondern es kann auf sämtliche – im Metamodell aufgeführte – Beziehungen zwischen diesen (wie z.B. *IsA*-Beziehungen) zugegriffen werden. Beim Beispiel in Abbildung 4.1 ist auf Metamodell-Ebene offensichtlich,

dass jede *Class* auch ein *NamedElement* ist. Vereinfacht lässt sich sagen, dass die Verarbeitung anhand von Typen, nicht anhand von Namen definiert wird.

Analoge Regeln, wie sie oben für die Verarbeitung auf syntaktischer Ebene aufgestellt wurden, könnten für eine Verarbeitung auf Metamodell-Ebene in etwa wie folgt aussehen:[96]

- ▷ Für jedes *NamedElement* generiere einen passenden Namen im Zielmodell.
- ▷ Für jede *Class* erzeuge ein passendes Element im Zielmodell. Verarbeite die Menge der *Property*-Instanzen, die als *ownedAttribute* auftreten, geeignet.
- ▷ Für jeden *DataType* erzeuge ein passendes Element im Zielmodell. Verarbeite die Menge der *Property*, die als *ownedAttribute* auftreten, geeignet.
- ▷ Für jede *Association* erzeuge ein Element im Zielmodell. Verarbeite die Menge der *Property*, die als *memberEnd* auftreten, geeignet.
- ▷ Für jede *Generalization* erzeuge ein passendes Element im Zielmodell, bei dem die *Classifier* *general* und *specific* verwendet werden.
- ▷ Für jede *Property* erzeuge ein passendes Element mit dem als *type* auftretenden *Classifier*.

Da ein Metamodell meist durch die Verwendung abstrakter Elementtypen und passender *IsA*-Beziehungen gut strukturiert ist, lassen sich oft weniger und übersichtlichere Anweisungen angeben, da Anweisungen für gleichartige Elementtypen zusammengefasst werden können. So lassen sich im Beispiel in Abbildung 4.1 alle Anweisungen, die den *name* einer *Property*, *Class*, *Association*, *DataType* und *PrimitiveType* betreffen, zu einer einzigen Anweisung für *NamedElement* zusammenfassen.

Werden die Verarbeitungsregeln anhand des Metamodells aufgestellt, lässt sich berücksichtigen, wie oft eine bestimmte Beziehung auftreten wird. Es ist also bereits beim Schreiben der Verarbeitungsregeln bekannt, ob nur ein einzelnes Element auftreten wird, ob dieses optional ist, oder ob mit einer Menge von Elementen gerechnet werden muss. So lässt sich beispielsweise dem Metamodell in Abbildung 4.1 (oben) entnehmen, dass eine Instanz von *Property* mit maximal einer Instanz von *Class* verbunden ist. Es ist außerdem bekannt, dass in diesem Fall neben der Beziehung *Property* → *Class* auch die umgekehrte Beziehung *Class* → *Property* besteht, wobei die *Property*-Instanz mit dem Rollennamen *ownedAttribute* auftritt.

[96] Die Regeln an dieser Stelle sind extrem verkürzt und dienen nur der Illustration der unterschiedlichen Herangehensweise. Damit die Regeln einfacher zu lesen sind, wird statt "x-Instanz" nur "x" geschrieben. Ebenfalls wird "die als ein Teilnehmer in einer Beziehung mit dem Rollennamen y auftreten" verkürzt zu "die als y auftreten".

4.4 QVT Relations

Als Transformationssprache für MOF-basierte Modelle bietet sich die Sprache "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)" an, eine von der OMG entwickelte Sprache für Modell-zu-Modell-Transformationen. Dass bereits in ihrem Namen "MOF" enthalten ist, macht die enge Verzahnung deutlich. Die Spezifikation der Sprache greift auf die Standards MOF und OCL zurück. QVT-Transformationen können sowohl deklarativ (QVT Relations) als auch imperativ (Operational QVT) beschrieben werden.

Die deklarative QVT Relations (QVT-R) eignet sich für die in dieser Arbeit angestrebte beispielhafte Transformation besser, da

- eine deklarative Schreibweise zu kompakteren Transformationsregeln führt, die mit weniger Codewiederholungen auskommen,
- die entwickelten Regeln, ihr Miteinander-Wirken sowie der Zusammenhang von Quell- und Zielmodell anschließend besser analysiert werden können,
- während der Ausführung einer QVT-R Transformation *trace-Klassen* erzeugt werden, die zur späteren Analyse nützlich sind,
- eine grafische Syntax für QVT-R existiert, so dass die Transformationen übersichtlich und schnell verständlich dargestellt werden können.

QVT-R sieht zwar auch die Möglichkeit vor, eine bijektive Abbildung (→ 2.3.3 Bijektive Transformationen) zwischen Instanzen zweier Metamodelle in einer Menge von Transformationsregeln anzugeben, da jedoch nicht alle Teile von UML- und OWL-Metamodell transformiert werden können, kam diese Variante nicht in Frage.

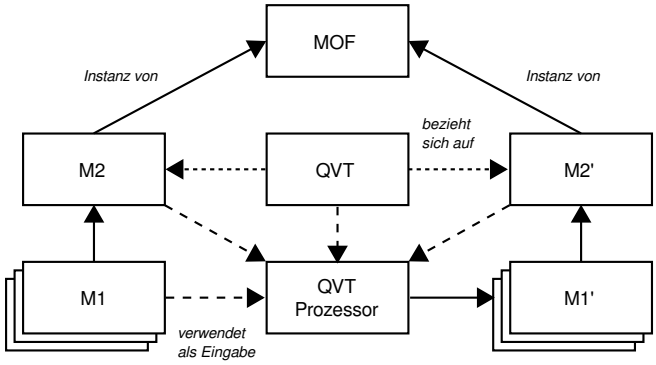


Abbildung 4.2. Übersicht der Transformation von Modellen mit Hilfe von QVT-Regeln, die sich auf MOF-konforme Metamodelle beziehen.

Abbildung 4.2 zeigt allgemein den grundlegenden Ablauf von Modelltransformationen mit QVT: Die QVT-Transformationsregeln beziehen sich auf das Eingabe-Metamodell $M2$ sowie auf das Ausgabe-Metamodell $M2'$, nicht jedoch auf die zu transformierenden Modelle.

Beide Metamodelle sind wiederum Instanzen des MOF. Zur Laufzeit verwendet eine QVT-Implementierung die Transformationsregeln, beide Metamodelle sowie das zu $M2$ konforme Eingabemodell $M1$, um anhand der Regeln ein zu $M2'$ konformes Ausgabemodell $M1'$ zu erstellen.

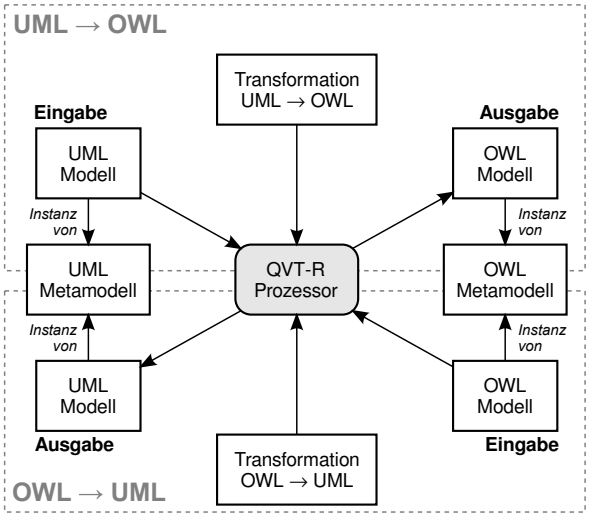


Abbildung 4.3. Transformation UML↔OWL im Überblick. Der obere Teil der Abbildung ist von links nach rechts, der untere Teil von rechts nach links zu lesen.

In Abbildung 4.3 ist der komplette Ablauf des konkreten, in dieser Arbeit beschriebenen, Transformationsprozesses für UML und OWL dargestellt. Die obere Hälfte zeigt die UML → OWL Transformation, die untere Hälfte die OWL → UML Transformation. Beiden Transformationsrichtungen gemein ist die Verwendung der UML- und OWL-Metamodelle. Vor und nach den eigentlichen Modell-zu-Modell Transformationen müssen speziell beim Umgang mit OWL-Ontologien gewisse einfache Vortransformationen erledigt werden, um zwischen konkreter und abstrakter Syntax zu übersetzen. Diese können leicht mit Hilfe des OWL API[97] oder eines XSLT-Skripts erledigt werden.

[97] Vgl. Horridge, M./Bechhofer, S.: The OWL API: A Java API for OWL ontologies, in: Hoekstra, R./Patel-Schneider, P. (Hrsg.): Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009), 2009 ⟨URL: http://ceur-ws.org/Vol-529/owled2009_submission_29.pdf⟩.

Kapitel 5

Andere Arbeiten

Eine kritische Betrachtung des scheinbaren Unterschieds zwischen Modellen auf der einen und Ontologien auf der anderen Seite wird von Atkinson et al. vorgenommen.[98] Die Autoren kommen zu dem Schluss, dass alle Ontologien auch Modelle und ebenso fast alle Modelle auch Ontologien sind.

5.1 Über das Verhältnis von UML und OWL

Es existiert eine Vielzahl von Arbeiten, die sich grundlegend mit dem Verhältnis von UML und Ontologien im Allgemeinen bzw. OWL im Speziellen beschäftigen. Dabei ist das Ziel nicht unbedingt die Transformation zwischen UML und OWL.

Einen wichtigen Meilenstein bei der Betrachtung verwandter Arbeiten stellt dabei der Aufruf der Object Management Group (OMG) im Jahre 2003 dar, Vorschläge für das Ontology Definition Model (ODM), ein MOF-kompatibles Modell zur Beschreibung von OWL-Ontologien, einzureichen. Als Reaktion auf diesen Aufruf sind eine Reihe von Beiträgen veröffentlicht worden. Neben dem Vorschlag für ein Metamodell enthalten diese Arbeiten meist Überlegungen, wie Ontologien mit Hilfe von UML modellieren werden können. Eine Transformation zwischen UML und OWL steht in der Regel nicht im Fokus dieser Arbeiten.

Daher können die Arbeiten, die sich mit dem Verhältnis von UML und OWL beschäftigen, in zwei Gruppen unterteilt werden:

1. Erweiterung der UML (vor dem ODM)
2. Arbeiten in Zusammenhang mit dem ODM

5.1.1 Erweiterung der UML (vor dem ODM)

Baclawski et al. stellen eine Erweiterung der UML vor, um die Beschreibung von – in der DARPA Agent Markup Language (DAML) geschriebenen – Ontologien mit Hilfe von UML zu verbessern.[99] Es werden eine Reihe von Stereotypen definiert. Mit diesen lassen sich gewöhnliche UML-Konstrukte wie Assoziationen markieren (z.B. «TransitiveProperty»). Daneben

[98] Vgl. Atkinson/Gutheil/Kiko: On the relationship of ontologies and models.

[99] Vgl. Baclawski et al.: Extending UML to Support Ontology Engineering for the Semantic Web, in: Gogolla, M./Kobryn, C. (Hrsg.): The Unified Modeling Language: Modeling Languages, Concepts, and Tools; 4th International Conference; Proceeedings / «UML» 2001, Berlin/Heidelberg/New York/Barcelona/Hong Kong/London/Milan/Paris/Tokyo 2001.

gibt es aber eine Reihe von Stereotypen zum Kennzeichnen in der UML nicht vorgesehener Beziehungen zwischen zwei Assoziationen (z.B. «subPropertyOf» und «samePropertyAs»). Die Autoren schlagen ferner vor, das UML-Metamodell um neue Klassen zu erweitern (genannt werden "Property" und "Restriction"). Dies widerspricht der in der UML-Spezifikation an Erweiterungen gestellten Anforderung, dass diese keine neuen Meta-Klassen einführen dürfen.[100] Da es sich bei der in dieser Arbeit verwendeten Sprache DAML um einen Vorgänger von OWL handelt, lassen sich die Ergebnisse nur bedingt auf die aktuelle Technik übertragen.

Schreiber stellt eine UML-basierte grafische Notation von OWL-Lite vor.[101] Er arbeitet mit Stereotypen (wie «subproperty», «datatype», «defined class» etc.) und verwendet neu definierte Schlüsselwörter für UML-Constraints (wie "SameClassAs"). Jede in einer Ontologie definierte Klasse soll im UML-Modell mit dem Stereotyp «defined class» gekennzeichnet werden. Neben diesen UML-konformen Erweiterungen wird ebenfalls eine nicht konforme Veränderung der grafischen Syntax von UML vorgeschlagen: Bei einer durch Angabe ihrer Population definierten Klasse soll durch eine, in die rechte obere Ecke des Klassen-Rechtecks geschriebene, Zahl die Größe der Population angegeben werden. Zum einen beschäftigt sich der Beitrag nur mit OWL-Lite, also einer stark eingeschränkten Teilmenge von OWL, zum anderen handelt es sich nach Aussage des Autors bei dem Beitrag nur um einen Entwurf, der sich mit einem ebenfalls im Entwurfsstadium befindlichen Sprachvorschlag für OWL befasse.

5.1.2 Arbeiten in Zusammenhang mit dem ODM

Djurić et al. präsentieren einen Vorschlag für ein ODM.[102,103] Im Metamodell werden viele Elementtypen verwendet, zwischen denen Vererbungsbeziehungen bestehen. Alle Elementtypen sind Untertypen von *Resource*. Zwei wichtige abstrakte Elementtypen mit zahlreichen Untertypen sind *AbstractClass*, unter dem alle Elementtypen betreffende Modellelemente zusammengefasst sind und *Property*, unter dem alle Beziehungstypen betreffende Modellelemente zusammengefasst sind. Die Beschreibung des Metamodells bleibt jedoch relativ oberflächlich, es fehlt z.B. eine Angabe der Beziehungen zwischen den beschriebenen Elementtypen. In dieser Arbeit wird auch die Transformation von UML-Modellen in OWL-Ontologien thematisiert: In Kombination mit dem von Djurić, Gašević et al. definierten UML-Profil für Ontologien "Ontology UML Profile"[104] dient das vorgeschlagene ODM als Zwischenstation für die Transformation von UML und OWL (und umgekehrt). Es werden XSLT-Transformationen für einen mehrstufigen Transformationsprozess verwendet, der vier Schritte umfasst:

[100] OMG: UML Infrastructure, S. 178: "*It is therefore forbidden to insert new metaclasses in the UML metaclass hierarchy (i.e., new super-classes for standard UML metaclasses) or to modify the standard UML metaclass definitions (e.g., by adding meta-associations).*"

[101] Vgl. Schreiber, G.: A UML Presentation Syntax for OWL Lite, 2002 ⟨URL: `http://www.swi.psy.uva.nl/usr/Schreiber/docs/owl-uml/owl-uml.html`⟩ – Zugriff am 28. Januar 2013.

[102] Vgl. Djurić, D.: MDA-based ontology infrastructure, in: Computer Science and Information Systems 1/1, Novi Sad 2004.

[103] Vgl. Djurić, D./Gašević, D./Devedžić, V.: Ontology modeling and MDA, in: Journal of Object Technology 4/1, 2005, fast identisch mit dem zuvor genannten Artikel.

[104] Vgl. Djurić et al.: A UML Profile for OWL Ontologies.

1. in XML-Syntax geschriebene Ontologien,
2. XMI-Dokumente mit Instanzen des ODM
3. UML-Modelle, die das in der Arbeit vorgestellte Ontologie UML-Profil verwenden
4. allgemeine UML-Modelle

Aufgrund der im Artikel vagen Beschreibung der Transformationen[105,106] ist nicht genau zu sagen, ob hier möglicherweise die in der (weiter unten diskutierten) Arbeit von Gašević, Djurić et al. beschriebene Transformation[107] gemeint ist. In dem Fall wären die Vorteile, die eine Transformation auf Metamodell-Ebene bietet, nicht ausgenutzt worden. Auf den heutigen Stand von OWL lässt sich die Transformation nicht übertragen, da das OWL-Metamodell von dem hier vorgestellten Metamodell deutlich abweicht.

Brockmans et al. beschreiben ein solches Metamodell für OWL.[108,109] Das Metamodell kommt mit relativ wenigen Elementtypen aus, die Untertypen von *OntologyElement* sind. Es wird viel mit klassenabhängigen Attributen gearbeitet. So enthält der Elementtyp *ObjectProperty* Wahrheitswerte, die angeben, ob es sich um eine transitive, symmetrische oder invers funktionale Object Property handelt. Neben dem Vorschlag für das Metamodell enthalten die Beiträge auch Vorschläge für ein UML-Profil zur grafischen Modellierung von Ontologien mit Hilfe von UML. Anders als die zuvor vorgeschlagenen Darstellungen kommt Brockmans et al. Vorschlag mit deutlich weniger Stereotypen aus und nutzt die existierende grafische UML-Syntax besser aus. So müssen einfache Klassen nicht mit einem Stereotyp markiert werden, die Äquivalenz zweier Klassen wird durch wechselseitige Generalisierungsbeziehungen angezeigt. Die Darstellung von Properties erfolgt mit den bei UML selten gebrauchten[110] n-stelligen Assoziationen, die mit dem Stereotypen «owl:ObjectProperty» gekennzeichnet werden. Transformationen zwischen UML und OWL werden in den Artikeln jedoch nicht thematisiert.

[105] Vgl. Djurić/Gašević/Devedžić: Ontology modeling and MDA, S. 115.
[106] Vgl. Djurić: MDA-based ontology infrastructure, S. 94f.
[107] Vgl. Gašević, D. et al.: Converting UML to OWL Ontologies, in: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York 2004.
[108] Vgl. Brockmans et al.: Visual Modeling of OWL DL Ontologies Using UML.
[109] Vgl. Brockmans, S. et al.: A Model Driven Approach for Building OWL DL and OWL Full Ontologies, in: Cruz, I. et al. (Hrsg.): The Semantic Web – ISWC 2006, Heidelberg 2006.
[110] Vgl. Brockmans et al.: Visual Modeling of OWL DL Ontologies Using UML, S. 210 Fußnote 19.

Hart et al. identifizieren drei Gruppen von UML- und OWL-Merkmalen[111] und Modellelementen:[112]

1. solche, die mehr oder weniger in beiden Sprachen vorhanden sind,
2. Merkmale, die nur UML besitzt sowie
3. Merkmale, die nur OWL besitzt.

Anhand von Beispielen wird bei den gemeinsamen Merkmalen angegeben, wie diese Beispiele in beiden Sprachen aussehen würden. Diese Aufstellung ist in wenig veränderter Form als Kapitel 16 in die ODM-Spezifikation der OMG eingeflossen.[113]

Kiko und Atkinson geben eine sehr ausführliche Übersicht über die Gemeinsamkeiten und die Unterschiede von UML und OWL.[114] Die Übersicht orientiert sich an den Ausdrücken der OWL und ist entsprechend gegliedert. Im Gegensatz zu der Arbeit von Hart et al. wird in besonderem Maße die OCL eingesetzt, um Ausdrücke der OWL wiedergeben zu können. Die OWL-Ausdrücke werden mit Freitext und formaler Definition beschrieben. Es wird ein Beispiel in konkreter visueller UML-Syntax für einen entsprechenden Sachverhalt gegeben. Zu jedem dieser Beispiele wird diskutiert, wie sich ggf. mit Hilfe von OCL die Semantik des OWL-Ausdrucks präziser ausdrücken lässt. Eine Transformation zwischen UML und OWL wird im Ausblick der Arbeit für möglich gehalten,[115] in der Arbeit selbst jedoch nicht thematisiert.

5.2 Transformationen zwischen UML und OWL

Es lassen sich zwei grundlegend verschiedene Ansätze unterscheiden:

1. XML-basierte Transformation
2. Transformationen, die nicht XML-basiert arbeiten

5.2.1 XML-basierte Transformation

Bei der im XML TS ablaufenden XML-basierten Transformation mit Hilfe der XSLT wird ausgenutzt, dass sich UML-Klassendiagramme in das XMI-Format und Ontologien in eine XML-basierte RDF-Syntax serialisieren lassen. Somit sind Quelle und Ziel XML-basierte Formate, so dass XSLT-Stylesheets verwendet werden können, um die Transformation zwischen den serialisierten Modellen durchzuführen.

[111] Im Artikel wird von *feature* gesprochen.
[112] Vgl. Hart, L et al.: OWL Full and UML 2.0 Compared, 2004 ⟨URL: http://www.omg.org/docs/ontology/04-03-01.pdf⟩.
[113] Vgl. OMG: Ontology Definition Metamodel.
[114] Vgl. Kiko, Kilian/Atkinson, C.: A Detailed Comparison of UML and OWL, Mannheim 2008 ⟨URL: http://madoc.bib.uni-mannheim.de/madoc/volltexte/2008/1898/pdf/TR2008_004.pdf⟩.
[115] Vgl. a. a. O., S. 52.

Es lassen sich bei den XML-basierten Transformationen wiederum zwei Arten von Ansätzen unterscheiden:

1. Transformation via XML Schema

2. XSLT-basierte Transformation auf syntaktischer Ebene der Modelle

Unabhängig vom gewählten Transformationsweg sind allen XML-basierten Ansätzen jedoch zahlreiche Nachteile gemein, die bereits im vorherigen Kapitel dargestellt wurden. Dazu zählen sowohl die allgemeinen Einschränkungen, die sich durch die Betrachtung von nur einer Modell-Ebene ergeben, als auch die Schwierigkeiten von XML-basierten Transformationen unter Verwendung von XMI-Dokumenten sowie Texten in den XML-basierten Syntaxen von OWL- und RDF. Auch andere Autoren weisen auf die sich durch die Verwendung von XSLT ergebenden Probleme hin.[116,117]

Transformation via XML Schema

Neben Arbeiten wie denen von Bohring und Auer[118] sowie Bedini et al.[119], die sich ganz allgemein mit der Transformation von XML Schemata (und nicht UML-Modellen) in OWL-Ontologien beschäftigen, nutzen Tschirner et al. die Tatsache aus, dass sich mit Hilfe von GML konzeptuelle Modelle, die als UML-Diagramme vorliegen, in XML Schemata transformieren lassen.[120]

Als Ziel der Arbeit wird unter anderem *"[...] a general approach for deriving [...] ontologies from [...] UML/GML data models [...]"*[121] genannt. Es bleibt aber offen, ob die in dem Beitrag vorgestellten Regeln[122] eine automatische Transformation ermöglichen sollen oder ob sie lediglich Hinweise für eine Ontologie-Erstellung von Hand sind.

XSLT auf syntaktischer Ebene der Modelle

Cranefield hat sich in zwei Artikeln mit dem Zusammenhang von UML und Ontologien beschäftigt: Cranefield und Purvis haben allgemein untersucht, wie UML und OCL genutzt

[116] Vgl. Falkovych, K./Sabou, M./Stuckenschmidt, H.: UML for the Semantic Web: Transformation-based approaches, in: Knowledge Transformation for the Semantic Web 95, 2003, S. 12.

[117] Vgl. Milanović, M. et al.: On Interchanging Between OWL/SWRL and UML/OCL, in: Proceedings of 6th Workshop on OCL for (Meta-) Models in Multiple Application Domains (OCLApps), 2006.

[118] Bohring, H./Auer, S.: Mapping XML to OWL ontologies, in: Jantke, K. P./Fähnrich, K.-P./Wittig, W. S. (Hrsg.): Leipziger Informatik-Tage, Leipzig 2005.

[119] Bedini, I/G, Gardarin/Nguyen, B: Deriving Ontologies from XML Schema, in: Cépaduès (Hrsg.): Entrepôts de données et analyse en ligne - EDA'08, Toulouse 2008.

[120] Vgl. Tschirner, S./Scherp, A./Staab, S.: Semantic access to INSPIRE, in: Grütter, R. et al. (Hrsg.): Terra Cognita 2011. Proceedings of the Terra Cognita Workshop on Foundations, Technologies and Applications of the Geospatial Web, 2011 ⟨URL: http://ceur-ws.org/Vol-798/proceedings.pdf⟩.

[121] A. a. O., S. 2.

[122] A. a. O., Abschnitt 3.

werden können, um Ontologien zu modellieren.[123] Ziel bei dieser früheren Arbeit war nicht die Transformation von UML zu OWL, sondern die Nutzung von UML als Ontologie-Modellierungssprache. Daher werden in der Arbeit auch Überlegungen angestellt, in welcher Weise mit UML-Modellen Reasoning betreiben werden kann. Eine Transformation von UML-Klassendiagrammen in Java-Programmcode sowie in RDFS stellt Cranefield in einer späteren Arbeit vor.[124] Für diese Transformation werden zwei XSLT-Stylesheets eingesetzt.

Da in der Arbeit nur RDFS verwendet wird und der Autor nicht die Absicht hatte, ein komplettes Modell zu transformieren,[125] bleibt die Transformation der Modellelemente recht oberflächlich.

Falkovych präsentiert eine Transformation von UML-Modellen in DAML+OIL (einem Vorgänger der OWL) und RDFS mit Hilfe von XSLT.[126] Die Transformation erfolgt zwischen der Serialisierung eines UML-Klassendiagramms in einer XMI-Datei und einer DAML+OIL-Ontologie in XML-Syntax. Dabei macht es sich der Autor zunutze, dass in einer XMI-Datei alle Modellelemente mit einem eindeutigen Schlüssel versehen sind – dieser wird genutzt, um den unbenannten Elementen des UML-Modells einen Namen in der Ontologie zuzuweisen. Die Arbeit enthält ein System zur Klassifikation verschiedener Arten von Assoziationen (z.B. binary_unnamed, unidirectional_following_direction_named). Für jeden Eintrag in diesem Klassifikationssystem wird eine Object Property angelegt, z.B. `<daml:ObjectProperty rdf:ID="binary_unnamed»`. Wird bei der Transformation aus einer entsprechenden Assoziation eine Object Property generiert, so wird diese als Unterbeziehungstyp dieser Basis-Object-Properties definiert (z.B. `<rdfs:subPropertyOf rdf:resource="#binary_unnamed"/>`).

Klassenabhängige Attribute werden grundsätzlich zu Data Properties transformiert – im Allgemeinen ist das nicht korrekt. Die Nutzung der Schlüssel aus dem XMI-Dokument führt teilweise zu schlecht lesbaren und schlecht nutzbaren Namen, da diese – oft kryptischen, da nicht für den menschlichen Leser bestimmten – XMI-Schlüssel u.a. in den Namen von Properties einfließen. Der Ansatz, aus Assoziationen generierte Object Properties als Unterbeziehungstypen abstrakter Basis-Beziehungstypen zu deklarieren, ist interessant. Er führt jedoch zu unübersichtlichen Ontologien, da diese Vererbungsbeziehung der Beziehungstypen im Originalmodell nicht enthalten war. Ein Nutzen dieser Markierungen wird nicht deutlich. In einem späteren Artikel weist Falkovych auf die Probleme hin, die durch die Verwendung der XSLT entstehen.[127]

[123] Vgl. Cranefield, S./Purvis, M.: UML as an ontology modelling language, in: The Information Science Discussion Paper Series 99/01, Dunedin 1999.

[124] Vgl. Cranefield, S.: Networked Knowledge Representation and Exchange using UML and RDF, in: Journal of Digital information 1/8, Austin 2001.

[125] A. a. O.: *"[...] it was not a goal to express all details of the model, only enough to facilitate serialisation of model instances."*

[126] Vgl. Falkovych, K.: Ontology Extraction from UML Diagram, Amsterdam 2002.

[127] Vgl. Falkovych/Sabou/Stuckenschmidt: UML for the Semantic Web, S. 12.

Gašević, Djurić et al. beschreiben die Transformation eines UML-Klassendiagramms in eine OWL-Ontologie mit Hilfe von XSLT.[128] Dabei muss bei der Erstellung des Klassenmodells das von Djurić, Gašević et al. definierte UML-Profil "Ontology UML Profile"[129] verwendet werden. Das UML-Modell wird als XMI-Datei abgespeichert und mit Hilfe eines XSLT-Stylesheets in eine nicht näher beschriebene "*OWL XML description*"[130] übertragen. Aufgrund der im Artikel gegebenen Beispiele lässt sich annehmen, dass es sich bei dieser konkreten Syntax um die RDF/XML-Syntax für OWL handelt.

Durch die Verwendung des "Ontology UML Profile" werden UML-Modelle unübersichtlich, da fast alle Modellelemente mit Stereotypen gekennzeichnet werden müssen, z.B. jede benannte Klassen mit dem Stereotyp «OntClass». Außerdem ist es notwendig, in UML-Notizen Hinweise für die Transformation zu geben, wie z.B. die Anweisung, dass das Ziel der Transformation eine anonyme Klasse ist. Erstaunlich ist der Hinweis, die Serialisierung des UML-Modells als XMI-Datei sei "*fairly awkward since it contains full description of an UML model.*"[131] – bei einer Serialisierung eines Modells sollte eigentlich genau dies zu erwarten sein. Ebenso erschließt sich der Hinweis, die Transformation sei "*especially difficult because each UML construct is a different stereotype.*"[132] nicht, da doch die Definition der Stereotypen ganz in der Hand der Autoren lag.

Leinhos beschreibt zwei XSLT-Stylesheets zur Transformation von UML-Modellen, die als XMI-Dateien serialisiert wurden, in OWL-Ontologien in konkreter RDF/XML-Syntax.[133] Hervorzuheben ist, dass die kompletten Transformationsregeln veröffentlicht wurden, sodass das Vorgehen im Detail nachvollzogen werden kann. Eines der Stylesheets dient zur Transformation allgemeiner UML-Klassendiagramme in OWL-Ontologien. Das andere Stylesheet dient zur Transformation speziell konstruierter UML-Klassendiagramme in OWL-Ontologien, sodass sich mehr OWL-Konstrukte nutzen lassen. Ähnlich wie bei Falkovychs Ansatz werden Aggregationen und Kompositionen durch Vererbung von Beziehungstypen markiert.

Bei der Transformation für speziell konstruierte UML-Klassendiagramme müssen bereits bei der Definition des UML-Modells besondere Rollennamen (z.B. "allValuesFrom") bei Assoziationen und Klassennamen (z.B. "unionOf") verwenden werden, die bei der Transformation in entsprechende OWL-Konstrukte übersetzt werden. Bei der Transformation allgemeiner UML-Klassendiagramme werden der Ontologie Elemente hinzugefügt, die im ursprünglichen UML-Modell nicht vorhanden sind und nicht mit der Semantik des UML-Modells übereinstimmen. So muss jede Instanz einer OWL-Klasse O, die aus einer abstrakten UML-Klasse transformiert wurde, genau eine Data Property mit dem Namen "isAbstract" besitzen. Da jede Unterklasse O' von O ebenfalls diese Einschränkung erbt, müssen auch alle Instanzen von O' eine solche Data Property besitzen.

[128] Vgl. Gašević et al.: Converting UML to OWL Ontologies.
[129] Vgl. Djurić et al.: A UML Profile for OWL Ontologies.
[130] Gašević et al.: Converting UML to OWL Ontologies.
[131] A. a. O.
[132] A. a. O.
[133] Vgl. Leinhos, S.: OWL Ontologieextraktion und -modellierung auf der Basis von UML Klassendiagrammen, Diplomarbeit Universität der Bundeswehr München, München 2006.

5.2.2 Nicht-XML-basierte Transformationen

Milanović, Gašević et al. beschreiben die Transformation von OCL-Regeln in Semantic Web Rule Language (SWRL)-Regeln mit Hilfe der Atlas Transformation Language (ATL).[134,135] Im Transformationsprozess kommen eine Reihe von Zwischenformaten zum Einsatz:

- die REWERSE Rule Markup Language (R2ML), eine MOF-basierte Sprache zur Definition von Regeln,
- das Rule Definition Metamodel (RDM),
- ein nicht näher beschriebenes MOF-basiertes Metamodell für XML,
- und das in der OCL-Spezifikation beschriebene OCL-Metamodell.

Da sich dieser Artikel nicht mit der Transformation von konzeptuellen Modellen, sondern von Regeln befasst, gehört er nur bedingt zu der relevanten Literatur. Bemerkenswert ist jedoch das Vorgehen der Transformation mit Hilfe von Metamodellen – wenn auch nicht für UML und OWL 2. Die Vielzahl der dort verwendeten Zwischenmodelle macht die Transformation jedoch unübersichtlich und fehleranfällig, da in jedem Transformationsschritt Informationen verlorengehen oder verändert werden können.

In der ODM-Spezifikation werden – neben der bereits erwähnten Gegenüberstellung von UML und OWL – in QVT-R spezifizierte Transformationsregeln für Transformationen UML → OWL und OWL → UML angegeben.[136]

Das im ODM beschriebene Metamodell für OWL 1 unterscheidet sich deutlich von dem OWL 2-Metamodell: Beim OWL 2-Metamodell stehen die Axiome im Vordergrund. Daher gibt es dort z.B. Elementtypen wie *ObjectPropertyDomain*, *ObjectPropertyRange* und *SubObjectPropertyOf* als Untertypen des abstrakten Typs *Axiom*. Dagegen wurde beim ODM der Fokus auf die Entitäten selbst (also die Klassen, Properties etc.) gelegt. Dementsprechend besitzt dort die Properties repräsentierende Klasse *RDFProperty* die klassenabhängigen Attribute *RDFSdomain*, *RDFSrange* sowie *RDFSsubPropertyOf*. Aufgrund dieses fundamentalen Unterschieds sind die im ODM angegebenen Transformationen nur begrenzt auf OWL 2 übertragbar.

Die im ODM beschriebenen Transformationen enthalten zudem eine Reihe von Fehlern[137], die Grund zum Zweifeln geben, dass sich mit Hilfe dieser Regeln wirklich komplette Modelle transformieren lassen. Dieser Verdacht wird durch die Tatsache verstärkt, dass nur ein Teil der Modellelemente behandelt wird. An vielen Stellen bleibt der Text vage und überlässt eine Formulierung der Regeln explizit dem Leser. Es scheint sich bei den in Kapitel 16 der ODM

[134] Vgl. Milanović et al.: On Interchanging Between OWL/SWRL and UML/OCL.

[135] Vgl. Milanović, M. et al.: Towards Sharing Rules Between OWL/SWRL and UML/OCL, in: Electronic Communications of the EASST Volume 5, 2006, fast identisch mit dem vorherigen Artikel.

[136] Vgl. OMG: Ontology Definition Metamodel, Kapitel 16.

[137] Eine von Zedlitz und Jörke aufgestellte Sammlung der Fehler im ODM wurde in Jörke, Jan: Alternativen für die konzeptuelle Modellierung von GML, Diplomarbeit Christian-Albrechts-Universität, Kiel 2010, Anhang A.1 veröffentlicht und im Juli 2010 an die OMG übermittelt.

vielmehr um eine Sammlung einzelner Ideen für die einfache Transformation sich ohnehin ähnelnder Modellelemente zu handeln.

Höglund et al. verwenden MOFScript, um eine Transformation von UML nach OWL 2 durchzuführen.[138] Ziel der Arbeit ist das Validieren von Metamodellen. Da sich UML-basierte Metamodelle nur schwer validieren lassen, wollen die Autoren eine Transformation der Metamodelle in eine OWL-Ontologie durchführen, um mit den dort verfügbaren Softwarewerkzeugen entsprechende Validierungen vorzunehmen. Ausführlich geht der Artikel auf verschiedene Arten von Aggregation und Komposition ein und zeigt auf, wie diese mit Hilfe von SWRL-Regeln in der transformierten Ontologie überprüft werden können.

Bei MOFScript handelt es sich um eine Sprache zur Modell-zu-Text Transformation. Das Ergebnis der Transformation ist folglich ausschließlich eine Ontologie in konkreter OWL/XML-Syntax.[139] Das OWL 2-Metamodell wird bei der Transformation nicht berücksichtigt, da direkt Text ausgegeben wird. Da das Ziel der Arbeit das Validieren von Metamodellen ist, wird die Ontologie mit einigen Modellelementen angereichert, die zwar für das Validieren benötigt werden, die allgemeine Verwendbarkeit der Ontologie aber behindern: Es werden Klassen erzeugt, deren Instanzen die direkten Instanzen einer UML-Klasse repräsentieren sollen. Für Aggregation und Komposition werden "Collection"-Klassen eingefügt, für die entsprechend zusätzliche Object Properties erzeugt werden. Einige der beschriebenen Konstrukte funktionieren nur unter sehr speziellen Randbedingungen, vgl. die Anmerkungen zu (→ 9.4 Kardinalitätsbeschränkungen).

[138] Vgl. Höglund, S. et al.: Representing and Validating Metamodels using the Web Ontology Language OWL 2. TUCS Technical Report No. 973, Turku 2010 ⟨URL: http://tucs.fi/publications/attachment.php?fname=TR973.full.pdf⟩.

[139] Die Autoren betonen, dass sie lieber die functional-style Syntax verwendet hätten, dies aber aufgrund der zu dem Zeitpunkt fehlenden SWRL-Unterstützung nicht möglich gewesen sei.

Kapitel 6

UML und OWL – Gemeinsamkeiten und Unterschiede

In diesem Kapitel werden grundlegende Unterschiede zwischen UML und OWL aufgeführt. Diese haben unmittelbar Auswirkung auf die Transformationen und machen sie in einigen Fällen unmöglich.

6.1 Behandlung von Namen

Namen von Modellelementen werden in OWL und UML unterschiedlich behandelt. Dabei sind zwei Aspekte zu berücksichtigen

- Gültigkeitsbereich von Namen
- Anwendung bzw. Nicht-Anwendung einer Unique Names Assumption (UNA)

6.1.1 Gültigkeitsbereich von Namen

Bei OWL haben alle Namen globale Gültigkeit. Das bedeutet, dass ein an einer Stelle benanntes Element von überall mit diesem Namen referenziert werden kann.

Bei UML hingegen ist die Gültigkeit von Namen auf einen Namensraum (Instanzen des Meta-Elementtyps *Namespace*) begrenzt. Als Namensraum treten Pakete, Klassen, Datentypen und Assoziationen auf, da alle diese Meta-Elementtypen Untertyp von *Namespace* sind. Folglich müssen Namen von Element- und Beziehungstypen (d.h. Klassen, Datentypen, Assoziationen etc.) nur innerhalb eines Pakets eindeutig sein. Abbildung 6.1 zeigt im oberen Teil zwei Beispiele: Die Klasse “Klasse1” im Paket “Paket1” ist trotz gleichen Namens verschieden von der Klasse “Klasse1” im Paket “Paket2”. Ebenso ist die Assoziation “beziehung” im Paket “Paket1” verschieden von der gleichnamigen Assoziation im Paket “Paket2”.

Eine lokale Gültigkeit haben auch die Namen klassenabhängiger Attribute. Ein klassenabhängiges Attribut mit dem Namen “titel” einer Klasse “Buch” bezeichnet ein anderes Attribut als eines mit dem Namen “title” einer Klasse “Person”, wie in Abbildung 6.1 Mitte abgebildet. Ebenfalls lokale Gültigkeit haben Rollennamen bei Assoziationen. Im unteren Teil der Abbildung 6.1 ist ein Beispiel dargestellt.

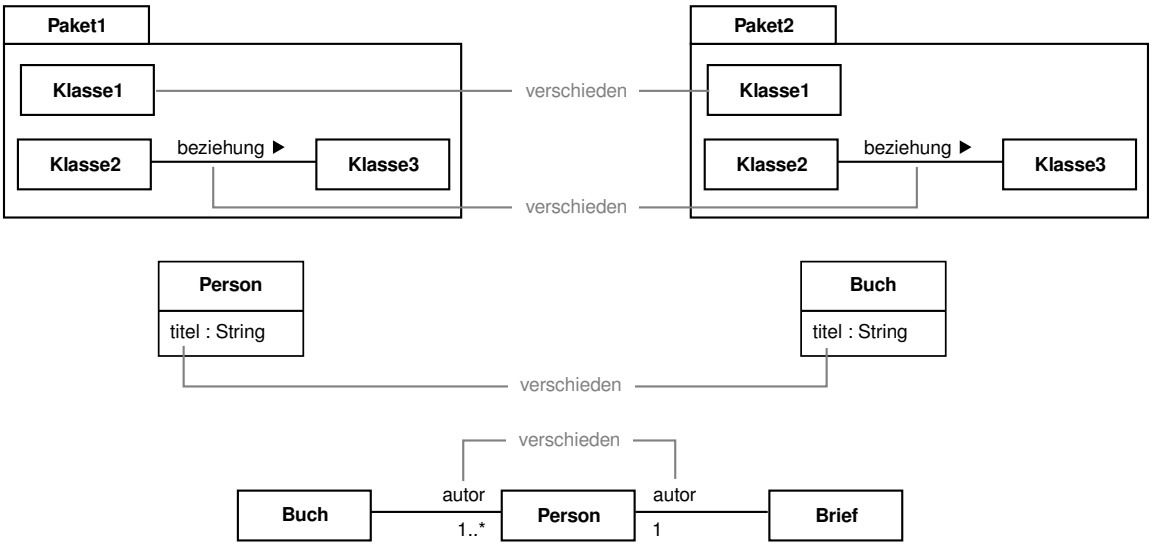


Abbildung 6.1. Beispiele für begrenzte Gültigkeit von Namen in UML.

Auch namenlose Elemente können unterschieden werden. Dies ist dann der Fall, wenn sie verschiedene Typen besitzen.[140]

Dieser Unterschied beim Gültigkeitsbereich von Namen – speziell der lokalen Gültigkeit von Attribut- und Rollennamen – ist für die Transformation UML → OWL von Bedeutung und wird im Abschnitt über Namen von Elementtypen (→ 7.2 NAMEN) noch einmal thematisiert.

6.1.2 Unique Names Assumption

"*The **unique names axiom** states a disequality for every pair of constants in the knowledge base. When this is assumed by the theorem prover, rather than written down in the knowledge base, it is called a **unique names assumption**.*"[141]

Bei UML wird eine solche Unique Names Assumption (UNA) verwendet. Diese besagt, dass es sich bei Elementen, die – innerhalb ihres Gültigkeitsbereiches (s.o.) – unterschiedliche Namen besitzen, um verschiedene Elemente handelt. Ein Modellelement besitzt also genau einen Namen. Ist ein Objekt Instanz einer Klasse mit dem Namen A, so ist es nicht möglicherweise Instanz einer Klasse mit dem Namen B – abgesehen von dem Fall, dass die Klassen in einer Vererbungsbeziehung stehen. Ohne UNA wäre dies möglich, dass es sich bei A und B um verschiedene Namen desselben Elements handeln könnte.

Weiterhin ist zu beachten, dass OWL keine UNA verwendet. Es ist daher möglich, einem einzigen Element mehrere verschiedene Namen zuzuweisen. Im Fall von Elementtypen könnte also ein Objekt, das Instanz eines Elementtyps A ist, ebenfalls Instanz eines Elementtyps

[140] OMG: UML Infrastructure, S. 75: "*The default rule is that two elements are distinguishable if they have unrelated types, or related types but different names.*"

[141] RUSSELL, Stuart J./NORVIG, Peter: Artificial Intelligence: A Modern Approach, Upper Saddle River, 2. Auflage 2003, S. 333.

B sein, – alleine dadurch, dass *A* und *B* verschiedene Namen für denselben Elementtyp sind. Für exakte Festlegungen bietet OWL mehrere Axiome an, mit denen definiert werden kann, dass sich Namen auf verschiedene Elemente beziehen. Dies wird im Abschnitt über Disjunktion (→ 10.2 Disjunktion) näher erläutert.

6.2 Open- und Closed-World-Assumption

Ein grundlegender Unterschied zwischen UML und OWL ist die Art, wie mit nicht explizit niedergeschriebenem Wissen umgegangen wird. In UML Klassenmodellen wird eine Closed World Assumption (CWA) verwendet: Alle Aussagen, die nicht explizit niedergeschrieben wurden, gelten als falsch.[142]

Diese unterschiedlichen Annahmen machen es notwendig, bei einer Transformation eines UML-Modells in ein OWL-Modell an einigen Stellen Restriktionen hinzuzufügen, damit die ursprünglichen Aussagen möglichst sinnerhaltend abgebildet werden. Insbesondere spielt der Unterschied zwischen OWA und CWA bei folgenden Modellelementen einen Rolle:

- **Beziehungstypen** — Bei UML sind zwei Beziehungstypen ohne Aussage über ihre Gleichheit verschieden. Bei OWL hingegen besteht die Möglichkeit, dass sie identisch sind. Gemäß der OWA wären sie nur dann verschieden, wenn eine Aussage über die Nicht-Gleichheit vorhanden ist. Unter anderem beim Überprüfen von Kardinalitätsbeschränkungen führt dieser Unterschied zu Problemen. Die bei UML getroffene Annahme lässt sich jedoch durch das Einfügen von Aussagen zur Disjunktion von Beziehungstypen nachbilden. (→ 9 Beziehungstypen)

- **Kardinalitätsbeschränkungen** — Sowohl UML als auch OWL sehen die Möglichkeit vor, obere und untere Schranken für die Häufigkeit des Auftretens von Beziehungen an einer Instanz anzugeben. Während die Überprüfung oberer Schranken auch unter OWA möglich ist, ist dies bei der Überprüfung unterer Schranken im Allgemeinen nicht der Fall.[143] (→ 9.4 Kardinalitätsbeschränkungen)

- **Komposition** — Ähnlich wie beim Überprüfen von Kardinalitätsbeschränkungen ist es zwar auch unter OWA möglich, zu prüfen, ob ein *Teil* fälschlicherweise zu mehr als einem *Ganzen* zugeordnet ist. Ob jedoch das *Teil* fälschlicherweise gar nicht einem *Ganzen* zugeordnet ist, lässt sich nicht überprüfen. (→ 9.6 Teil-Ganzes-Beziehungen)

[142] Russell/Norvig: Artificial Intelligence, S. 355: *"[...], so that ground atomic sentences not asserted to be true are assumed to be false."*

[143] Nur bei der Verwendung von Elementtypen mit fester Population (→ 7.5 Elementtypen mit fester Population) wäre dies möglich.

6.3 Monolevel und Multilevel Information Bases

OWL ermöglichst es, sowohl Elementtypen und ihre Beziehungen untereinander (T-Box) als auch Elemente, ihre Beziehungen untereinander und zu den Elementtypen (A-Box) in einer Ontologie aufzuschreiben. Es liegen also gleichzeitig Elemente mit verschiedenem Klassifikationslevel (→ 2.4.1 Model-Driven Architecture Technology Space) vor.

Bei UML ist dies sehr eingeschränkt möglich: "*UML does not allow a uniform, coherent representation of types, meta entity types, etc., in multilevel information bases. UML is essentially geared towards monolevel information bases, [...]*"[144] So wird bei UML zwischen dem *type level* und dem *instance level* unterschieden. Die zur Laufzeit existierenden und vom Modell beschriebenen Instanzen sind nicht Bestandteil des Modells: "*Instance level – These are the things that models represent at runtime. They don't appear in models directly [...] These classes do not appear at all in the UML2 metamodel or in UML models, [...]*"[145] Dies hat u.a. Auswirkung auf die Definition von Elementtypen mit fester Population (→ 7.5 Elementtypen mit fester Population).

Im strengen Sinne handelt es sich jedoch auch bei einer OWL-Ontologie nicht um eine Multilevel Information Base. Dazu wäre nämlich neben dem gemeinsamen Auftreten von Elementtypen und ihren Instanzen erforderlich, dass sich Elementtypen selbst wieder als Instanzen verwenden lassen. Dies ist aber nicht der Fall; es lassen sich zwar, wie in Listing 6.1 dargestellt, gleich benannte Elementtypen und Instanzen innerhalb einer Ontologie verwenden, es handelt sich aber bei beiden um verschiedene Elemente.[146]

```
ClassAssertion( a:Dog a:Brian )                    Brian is a dog.
ClassAssertion( a:Species a:Dog )                  Dog is a species.
```

Listing 6.1. Beispiel für einen Elementtyp und ein Individuum mit identischem Namen.[147]

In der ersten Zeile wird "Dog" als Name eines Elementtyps verwendet. In der zweiten Zeile bezeichnet "Dog" jedoch nicht diesen Elementtyp, sondern ein Individuum (das mit dem Elementtyp aus der ersten Zeile nichts zu tun hat).

6.4 Strukturierung

Sowohl UML als auch OWL kennen Konzepte zur Strukturierung umfangreicher Modelle, die sich in mehrere Pakete bzw. Ontologien aufteilen lassen. (→ 11.1 Pakete). Um innerhalb so

[144] Olivé: Conceptual Modeling, S. 393.

[145] OMG: Unified Modeling Language, Superstructure Version 2.4, 2011 ⟨URL: http://www.omg.org/spec/UML/2.4/Superstructure⟩, S. 13.

[146] Vgl. W3C: OWL 2 Structural specification, Abschnitt 5.9.

[147] aus Ebd. – Die Texte am Ende jeder Zeile sind als Kommentar zu verstehen. Auch wenn diese Schreibweise syntaktisch falsch ist, wird sie hier verwendet, da sie sich im Originaltext befindet.

strukturierter Modelle auf Elemente aus verschiedenen Bereichen zugreifen zu können, gibt es dementsprechend Paket-Importe und Ontologie-Importe. (→ 11.2 Importe) Jedoch gibt es in UML noch weitere Möglichkeiten der Erweiterung und Anpassung.

Wie bereits unter 3.1.4 dargestellt, kennt die UML das Konzept der Profile, die eine Einschränkung – bzw. im Fall des "UML-Profils" von ISO19100 eine Erweiterung – der Sprache darstellen. In OWL gibt es keine entsprechenden Konstrukte, das Metamodell in ähnlicher Weise zu erweitern. Insbesondere die in Profilen definierten Stereotypen lassen sich aber durch Klassen und Vererbungsbeziehungen in einer OWL-Ontologie abbilden, wie in Abbildung 6.2 skizziert. Es geht dabei jedoch die Information verloren, dass es sich um einen Stereotyp und nicht um eine normale Klasse handelte.

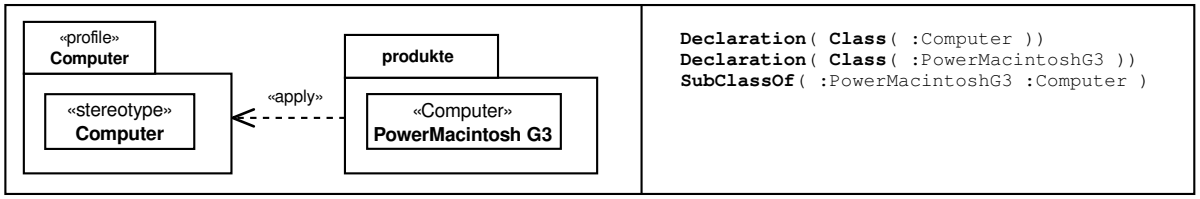


Abbildung 6.2. Beispiel einer Transformation der Anwendung eines Stereotyps in eine Vererbungsbeziehung.

In vielen Fällen sind allerdings UML-Profile zu spezifisch, um eine allgemeine Transformation angeben zu können. Hier sind Transformationsregeln, die speziell für ein UML-Profil erarbeitet wurden, zielführender.

6.5 Eingeschränkte Sichtbarkeit

In UML kann die Sichtbarkeit von Modellelementen eingeschränkt werden. Bei statischen Modellen betrifft dies vor allem klassenabhängige Attribute. Vier Varianten der Sichtbarkeit sind möglich:[148]

- public — Die Sichtbarkeit des Elements ist nicht eingeschränkt.
- private — Das Element ist nur innerhalb des Namensraums (→ 6.1.1 Gültigkeitsbereich von Namen), zu dem es gehört, sichtbar.
- protected — Das Element ist von den Elementen aus sichtbar, die eine Vererbungsbeziehung zum Namensraum, zu dem es gehört, besitzen.
- package — Das Element ist für alle Elemente, die sich im gleichen, nächstmöglich umschließenden Paket befinden, sichtbar. (Kann nicht auf Elemente angewendet werden, die direkt Bestandteil eines Pakets sind.)

[148] Vgl. OMG: UML Infrastructure, S. 90.

Die eingeschränkte Sichtbarkeit dient der Datenkapselung. Sie ist bei der Softwareentwicklung wichtig, da von außen nicht sichtbare Elemente geändert werden können, ohne andere Bestandteile der Software anpassen zu müssen.

Bei OWL gibt es keine solche Einschränkung der Sichtbarkeit. Über seinen global gültigen Internationalized Resource Identifier (IRI) kann von überall auf ein Element zugegriffen werden. Bei der Transformation OWL → UML wird daher grundsätzlich eine nicht eingeschränkte Sichtbarkeit (*public*) festgelegt. Bei der Transformation UML → OWL gehen Einschränkungen der Sichtbarkeit verloren.

6.6 Explizites und implizites Wissen

In OWL wird zwischen explizitem und implizitem Wissen unterschieden: Explizites Wissen ist innerhalb einer Ontologie in Form von Axiomen niedergeschrieben. Durch die Semantik von OWL festgelegt, ist ebenfalls implizites (d.h. nicht als Axiome niedergeschriebenes) Wissen in einer Ontologie enthalten. Dies ist in Listing 6.2 dargestellt: Die Klasse A enthält alle diejenigen Individuen, zu denen mindestes eine Object Property "op" angegeben ist. Die Klasse B enthält alle die Individuen, die mindestens zwei Object Properties "op" besitzen. Da jedes Individuum, das zwei oder mehr "op"-Properties besitzt (und damit zu Klasse B gehört) offensichtlich auch mehr als eine "op"-Property hat, gehört es folglich auch zu Klasse A. Damit ist die Klasse B eine Unterklasse von Klasse A.

```
1 Declaration( Class(:A) )
2 Declaration( Class(:B) )
3
4 Declaration( ObjectProperty( :op ))
5
6 EquivalentClasses( :A ObjectMinCardinality( 1 :op owl:Thing ) )
7 EquivalentClasses( :B ObjectMinCardinality( 2 :op owl:Thing ) )
```

Listing 6.2. Zwischen Klasse A und Klasse B gibt es eine implizite Unterklassen-Beziehung.

Soll das implizite Wissen einer Ontologie in ein UML-Modell übertragen werden, so lässt sich eine Reasoner verwenden, um es zu schlussfolgern (inferieren). Im obigen Beispiel ist dies wünschenswert, da die hinreichende Bedingung (`ObjectMinCardinality`) nicht in UML ausgedrückt werden kann (siehe nächster Abschnitt). Andererseits kann das zusätzlich geschlussfolgerte Wissen dazu führen, dass das Modell nach der Transformation verwirrend komplex wird.

Listing 6.3 zeigt ein einfaches Beispiel einer Unterklassen-Beziehung zwischen Klassen. Abbildung 6.3 stellt zwei verschiedene Ansätze der Transformation vor: auf der linken Seite ohne geschlussfolgertes Wissen, auf der rechten Seite mit einer zusätzlichen geschlussfolgerten Unterklassen-Beziehung zwischen Klasse C und Klasse A. Die Semantik beider Modelle ist die gleiche, da die *isA*-Beziehung auch in UML transitiv ist.

```
1 Declaration( Class(:A) )
2 Declaration( Class(:B) )
3 Declaration( Class(:C) )
4
5 SubClassOf( :A :B )
6 SubClassOf( :B :C )
```

Listing 6.3. Es existiert eine implizite IsA-Beziehung zwischen der Klasse A und der Klasse C.

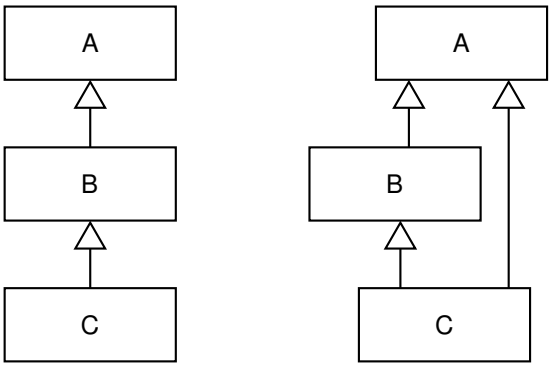


Abbildung 6.3. Das rechte Diagramm zeigt die implizite Unterklassenbeziehung zwischen Klasse A und Klasse C.

6.7 Notwendige und hinreichende Bedingungen

In OWL können außer benannten Elementtypen, denen explizit Objekte als Instanzen zugewiesen werden, auch (unbenannte) Class Expression (CE) definiert werden, bei denen die Zugehörigkeit über eine Formel festgelegt ist. Diese Formel ist dabei sowohl notwendige als auch hinreichende Bedingung für ein Objekt, Instanz des Elementtyps zu sein.

Einerseits ist jedes Objekt, das die Formel erfüllt, automatisch auch Instanz des Elementtyps. Mit anderen Worten: Das Erfüllen der Formel ϕ ist *hinreichende* Bedingung, damit das Objekt e eine Instanz des Elementtyps E ist.

$$\phi(e) \rightarrow E(e)$$

Andererseits erfüllt jedes Objekt, das Instanz des Elementtyps ist, auch die angegebene Formel. Mit andere Worten: Das Erfüllen der Formel ϕ ist *notwendige* Bedingung, damit das Objekt e eine Instanz des Elementtyps E ist.

$$E(e) \rightarrow \phi(e)$$

Auf diese Wiese können Individuen automatisch klassifiziert werden: Alleine durch das Erfüllen der Bedingung wird ein Individuum Instanz des Elementtyps. Listing 6.4 zeigt ein Beispiel für die Verwendung von CE. Der Ausdruck `DataExactCardinality(1 :hasISBN xsd:string)` in Zeile 1 beschreibt einen unbenannten Elementtyp, der diejenigen Individuen

umfasst, für die eine "hasISBN"-Eigenschaft angegeben ist.[149]

```
1 EquivalentClasses( :Book DataExactCardinality( 1 :hasISBN xsd:string) )
2
3 Declaration( NamedIndividual( :Economics )
4 DataPropertyAssertion( :hasISBN :Economics "1844801330" )
5
6 Declaration( NamedIndividual( :WealthOfNations )
7 ClassAssertion( :Book :WealthOfNations )
```

Listing 6.4. Das Individuum "Economics" ist Instanz der Klasse "Book" auch wenn es nicht ausdrücklich als `ClassAssertion`-Axiom aufgeschrieben wurde. Das Individuum "WealthOfNations" besitzt eine – nicht explizit definierte – Eigenschaft "hasISBN".

Da das Vorhandensein einer ISBN eine *hinreichende* Bedingung für ein Individuum ist, um Instanz des Elementtyps zu sein, ist das in Zeile 3 deklarierte Individuum "Economics" Instanz des Elementtyps "Book", obwohl dies nicht explizit als Axiom aufgeschrieben wurde. Alleine die Angabe der "hasISBN"-Eigenschaft in Zeile 4 reicht für eine Typisierung aus.

Da das Vorhandensein einer ISBN eine *notwendige* Bedingung für ein Individuum ist, um Instanz des Elementtyps zu sein, besitzt das in Zeile 6 deklarierte Individuum "WealthOfNations" eine "hasISBN"-Eigenschaft, da es in Zeile 7 als Instanz des Elementtyps "Book" festgelegt wurde. Dass im Beispiel keine "hasISBN"-Eigenschaft angegeben ist, ist aufgrund der OWA kein Problem – der Wert der Eigenschaft ist nicht bekannt, es existiert aber genau eine Eigenschaft für das Individuum.

UML unterstützt eine automatische Klassifikation anhand hinreichender Bedingungen nicht. Ein anonymer Elementtyp, wie er durch die obigen CE definiert wird, kann für sich alleine nicht in einem UML-Modell dargestellt werden. Es ist jedoch – wie später gezeigt wird – mit einigen Einschränkungen möglich, solche CE zu transformieren, wenn sie innerhalb einer Untertyp-Beziehung auftreten. (→ 7.3 Vererbung) Eine weitere Möglichkeit, die hinreichenden Bedingungen von OWL in UML wiederzugeben, ist die Verwendung der OCL, wie es von Kiko, Kilian und Atkinson vorgeschlagen wird.[150]

[149] Im Prinzip lassen sich unbenannte CE auch direkt in `ClassAssertion` (z.B. in Zeile 7) verwenden. Für ein besseres Verständnis wurde jedoch dem unbenannten Elementtyp mit Hilfe eines `EquivalentClasses`-Axioms "ein Name zugewiesen".

[150] Vgl. Kiko/Atkinson: A Detailed Comparison of UML and OWL.

6.8 Komplementbildung

An vielen Stellen erlaubt OWL es, mit dem Komplement von Klassen und Datentypen zu arbeiten. Bei UML ist dies im Allgemeinen nicht möglich. Nur wenn eine einzige Oberklasse für das gesamte Modell definiert wurde, kann eine als überscheidungsfrei und vollständig markierte Instanz des UML-Elementtyps *GeneralizationSet* (→ 7.6 GENERALISIERUNG) verwendet werden, um eine Klasse C und ihr Komplement $\neg C$ zu modellieren.

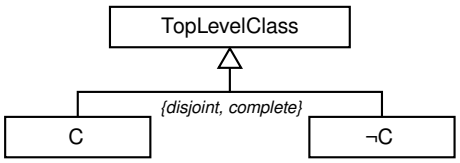


Abbildung 6.4. Dieses UML-Diagramm zeigt, wie das Komplement einer Klasse C modelliert werden kann.

6.9 Verschachtelte Class Expression

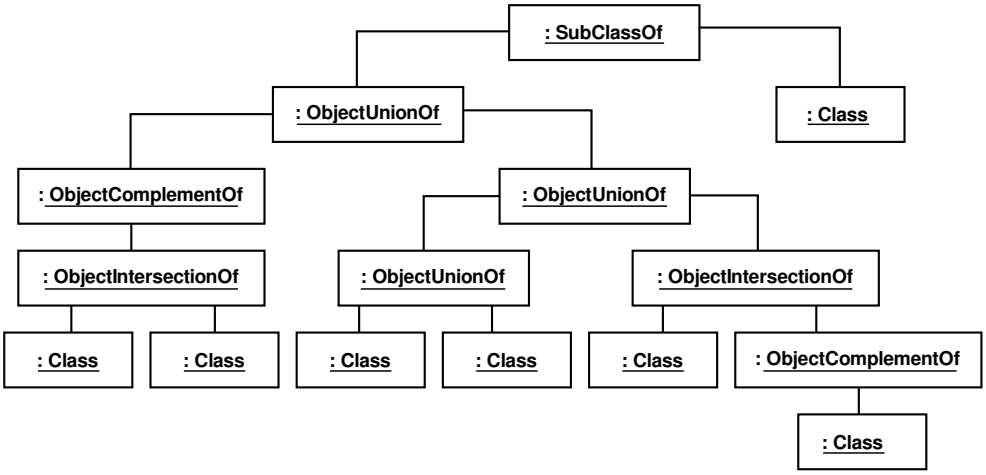


Abbildung 6.5. Beispiel für die Verschachtelung von CE.

Eine Schwierigkeit bei der Transformation OWL → UML stellen Class Expression (CE) dar, zu denen neben den benannten Klassen (Instanz des OWL-Elementtyps *Class*) auch eine Vielzahl namenloser CE zählt. Darunter sind solche, mit denen andere CE zusammengefasst werden können: z.B. `ObjectUnionOf`, `ObjectIntersection`. Mit Hilfe dieser Konstrukte können beliebig komplexe, baumförmig verschachtelte CE definiert werden. Ein Beispiel für eine solche baumförmige Verschachtelung ist in Abbildung 6.5 zu sehen.

In der Praxis treten jedoch solche stark verschachtelten CE nur selten auf. Es dominieren nicht-verschachtelte CE oder solche mit sehr wenigen Verschachtelungsebenen.

Die Transformation einer verschachtelten CE ist nur dann möglich, wenn auch alle Kindelemente (= enthaltene CE) transformiert werden können. Insbesondere die Komplementbildung (`ObjectComplementOf`-CE) stellt hierbei – wie weiter oben beschrieben – ein Problem dar.

6.10 Eigenschaften von Properties

Es ist in OWL möglich, diverse Eigenschaften ((Ir-)reflexivität, (A-)Symmetrie und Transitivität) für Properties zu definieren. Diese lassen sich nicht direkt in ein UML-Modell übertragen, da dort solche Kennzeichnungen nicht existieren. Es gibt jedoch einige Ansätze, wie diese Eigenschaften dennoch nicht verloren gehen:

- Man kann OCL-Ausdrücke angeben, um die Einschränkungen auszudrücken.[151]
- Es können Assoziationsklassen verwendet werden, die mit Hilfe von Metaklassen entsprechend den Eigenschaften der Property gekennzeichnet werden.[152]
- Analog lassen sich Assoziationsklassen nutzen und dort Attribute setzen, die die Eigenschaften beschreiben.[153,154]
- Eine weitere Möglichkeit besteht in der Nutzung von Stereotypen, mit denen angegeben wird, welche Eigenschaften OWL Properties bekommen sollen.[155]

Abbildung 6.6 gibt anhand eines einfachen Beispiels einen Überblick über die vier Möglichkeiten. Bei der Transformation von Beziehungstypen wird die Verwendung von Stereotypen zur Angabe von Eigenschaften bei Transitivität (→ 9.7 Transitivität) und Symmetrie (→ 9.8 Symmetrie) gezeigt.

[151] Vgl. Kiko/Atkinson: A Detailed Comparison of UML and OWL.
[152] Vgl. Schreiber: A UML Presentation Syntax for OWL Lite.
[153] Vgl. Brockmans et al.: Visual Modeling of OWL DL Ontologies Using UML.
[154] Vgl. Brockmans et al.: A model driven approach for building OWL.
[155] Vgl. Zedlitz, J./Jörke, J./Luttenberger, N.: From UML to OWL 2, in: Lukose, D./Ahmad, A. R./Suliman, A. (Hrsg.): Proceedings of Knowledge Technology Week 2011, Berlin/Heidelberg 2012.

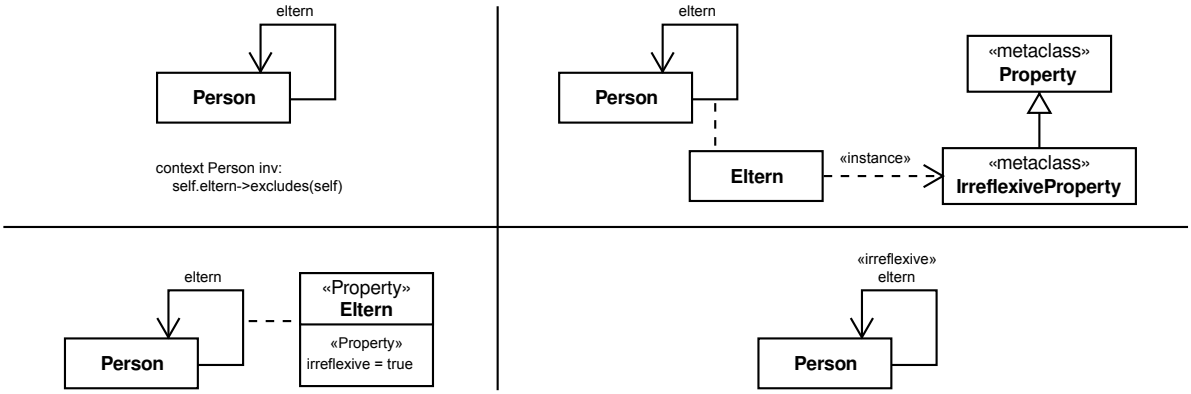


Abbildung 6.6. Vier Möglichkeiten für die Behandlung von Property-Eigenschaften in UML-Diagrammen. **Oben links:** Angabe eines OCL-Ausdrucks. **Oben rechts:** Verwendung einer Assoziationsklasse als Instanz einer Property-Metaklasse. **Unten links:** Verwendung einer Assoziationsklasse mit Attribut. **Unten rechts:** Angabe eines Stereotyps für die Assoziation.

Kapitel 7

Elementtypen

Dieses und die folgenden vier Kapitel bilden den Hauptteil der Arbeit. In ihnen werden Konzepte erläutert, die bei der Datenmodellierung verwendet werden. Jedes Kapitel beschäftigt sich mit einer bestimmten Gruppe von Konzepten:

- **Elementtypen** – Elementtypen dienen zur Klassifikation einzelner Objekte. In diesem Kapitel geht es folglich um die Beschreibung der Eigenschaften *eines* Objekts.

- **Datentypen** – Datentypen sind eine spezielle Art von Elementtypen, wobei der Wert eines Datentyps keine Identität besitzt, gleiche Werte eines Datentyps also nicht unterscheidbar sind.

- **Beziehungstypen** – Eine weitere Art von Typen sind Beziehungstypen, mit denen Beziehungen zwischen *mehreren* Objekten bzw. Werten beschrieben werden.

- **Beschränkungen** – In diesem Kapitel werden Beschränkungen betrachtet, die sich entweder auf beide vorgenannten Typen oder auf eine Kombination von Elementtypen und Beziehungstypen beziehen.

- **Strukturierung** – Um umfangreiche Modelle übersichtlich gestalten zu können, gibt es Strukturierungskonzepte. Zwei davon werden in diesem Kapitel betrachtet.

Jeder Abschnitt ist folgendermaßen aufgebaut: Zunächst wird das Modellierungskonzept allgemein vorgestellt. Es wird – sofern anwendbar – eine formale Beschreibung der Zusammenhänge des Konzepts gegeben. Darauf folgt eine Beschreibung, wie sich das Konzept in UML bzw. OWL darstellen lässt. Sofern vorhanden, wird sowohl die zugehörige konkrete Syntax als auch die abstrakte Syntax (d.h. Instanzen des Metamodells) dargestellt. Anschließend werden sowohl die Transformation UML → OWL als auch die Transformation OWL → UML mit ihren jeweiligen Besonderheiten präsentiert. Die Transformationen werden entweder durch Angabe der Transformationsregeln in grafischer QVT-R-Syntax oder durch Angabe eines Beispiels für transformierte Modelle in konkreter Syntax verdeutlicht. Jeder Abschnitt schließt mit einer Ausdeutung des Konzepts für XSD: Es wird gezeigt, wie sich das Konzept in XSD darstellen lässt, eine formale Betrachtung (z.B. auf Metamodellebene) findet jedoch nicht statt.

7.1 Allgemein

Elementtypen zählen zu den wichtigsten Bestandteilen eines konzeptuellen Modells: "*Defining the entity types [...] is a crucial task in conceptual modeling.*"[156] Elementtypen werden auch als *Konzepte* bezeichnet und sind so namensgebend für den Begriff "konzeptuelles Modell". Elementtypen dienen dazu, individuelle Objekte mit gleichen oder ähnlichen Eigenschaften zusammenzufassen. Ein solches individuelles Objekt, das einem Elementtyp zugeordnet ist, wird als *Instanz* des Elementtyps bezeichnet. Diese Instanzen von Elementtypen müssen identifizierbar sein.[157] Objekte können auch nur zeitweilig Instanz eines Elementtyps sein – dies ist aber bei der Betrachtung statischer Modelle, wie in dieser Arbeit, nicht relevant. Die Menge der Instanzen eines Elementtyps wird *Population* genannt.

Es ist prinzipiell erlaubt, dass ein Objekt Instanz von nicht nur einem Elementtyp, sondern zur selben Zeit Instanz mehrerer Elementtypen $E_1, \ldots, E_n$ ist, auch wenn diese Elementtypen nicht in einer Vererbungsbeziehung stehen. Ein solches Modell wird *multiple-classification model* genannt. Es ist solange konsistent, solange nicht zwei oder mehr der Elementtypen $E_1, \ldots, E_n$ als disjunkt (→ 10.2 Disjunktion) definiert sind.

7.1.1 Logische Repräsentation

Formal wird die Tatsache, dass ein Objekt e Instanz eines Elementtyps E ist, folgendermaßen ausgedrückt:

$$E(e)$$

Diese Beziehung, die aus einem Objekt eine Instanz eines Elementtyps macht, wird auch als *InstanceOf*-Beziehung bezeichnet.[158]

7.1.2 Repräsentation in UML

Elementtypen werden in UML *Klassen* genannt. Eine Instanz eines Elementtyps wird *Objekt* genannt. Mit UML lassen sich *multiple-classification models* erstellen, da ein Objekt Instanz mehrerer Elementtypen sein kann.[159]

Konkrete Syntax

Ein Elementtyp wird in der visuellen Syntax von UML durch ein weißes Rechteck dargestellt. Dieses ist in bis zu drei Teile unterteilt: Im obligatorischen oberen Teil ist der Name der Klasse (→ 7.2 Namen) zu finden. Darunter folgt optional ein Teil mit den klassenabhängigen

[156] Olivé: Conceptual Modeling, S. 41.
[157] A. a. O., S. 42: "*An entity type is a concept whose instances at a given time are identifiable individual objects that are considered to exist in the domain at that time.*"
[158] Vgl. a. a. O., S. 385.
[159] OMG: UML Superstructure, S. 85: "*Classification of the entity by one or more classifiers of which the entity is an instance.*"

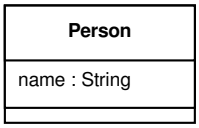


Abbildung 7.1. Beispiel für die konkrete grafische Syntax einer Klasse in UML.

Attributen (→ 9 Beziehungstypen). Im optionalen untersten Teil werden zur Klasse gehörige Operationen aufgelistet.

Abstrakte Syntax

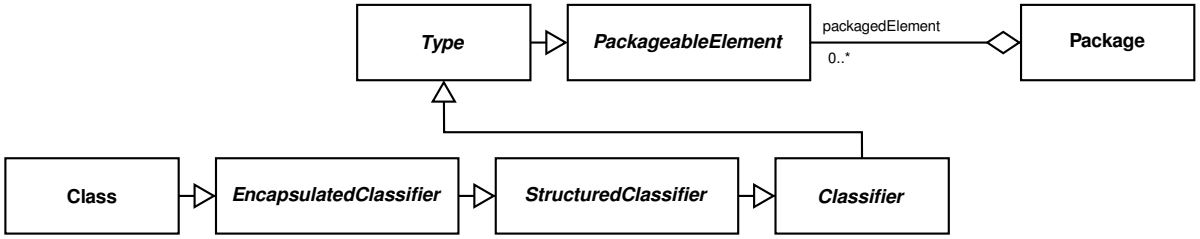


Abbildung 7.2. Ausschnitt aus dem UML-Metamodell, der die übergeordneten Elementtypen des Elementtyps *Class* sowie die Verbindung mit dem *Package* zeigt.

Eine UML-Klasse ist eine Instanz des Elementtyps *Class*. Bei *Class* handelt es sich um einen Untertyp von *PackageableElement*, dessen Instanzen Bestandteil eines Pakets sein können. Daher können Klassen direkt auf oberster Ebene einem Paket zugeordnet werden. Abbildung 7.2 zeigt den zugehörigen Abschnitt aus dem UML-Metamodell mit allen relevanten *IsA*-Beziehungen von *Class* bis *PackageableElement*.

7.1.3 Repräsentation in OWL

Elementtypen werden in OWL *Klassen* genannt. Eine Instanz eines Elementtyps wird *Indiviuum* genannt.

OWL kennt zwei besondere Elementtypen: `owl:Thing` und `owl:Nothing`. Der Elementtyp `owl:Thing` ist dabei als Menge aller Objekte, der Elementtyp `owl:Nothing` als leere Menge definiert.[160]

Neben benannten Klassen wird in OWL mit den allgemeineren Class Expression (CE) gearbeitet. Insgesamt kennt OWL 2 achtzehn Arten von CE, die in Abbildung 3.6 gut zu erkennen sind. Bei den meisten erfolgt die Zuordnung von Instanzen über hinreichende Bedingungen. CE im Allgemeinen werden nicht hier, sondern später in den Abschnitten zu (→ 7.6 Generalisierung), (→ 7.7 Schnittmenge), (→ 9.4 Kardinalitätsbeschränkungen) sowie (→ 9.5 Wertebeschränkungen) behandelt.

[160] Vgl. W3C: OWL 2 Structural specification, Abschnitt 5.1.

Konkrete Syntax

Mit Hilfe folgender konkreter Syntax wird die Existenz einer Klasse definiert:

```
Declaration( Class( name ) )
```

Gemäß der *Typing Constraints* müssen Klassen deklariert werden. Ihr Name darf sich nicht mit dem von einem Datentyp – jedoch mit dem einer Property, da hierbei keine Verwechslung droht – überschneiden.[161]

Abstrakte Syntax

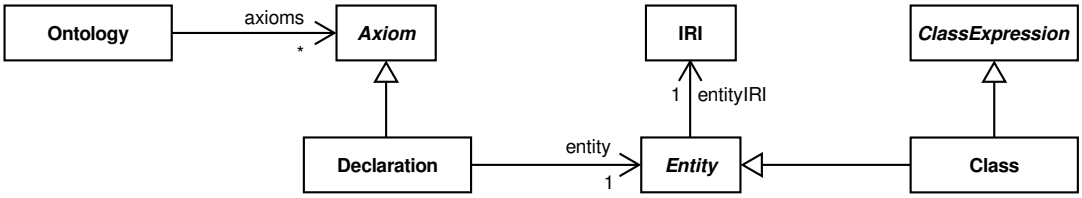


Abbildung 7.3. Ausschnitt aus dem OWL-Metamodell, der die zur Definition einer Klasse benötigten Elementtypen zeigt.

Eine Klasse wird durch eine Instanz von *Class* repräsentiert. Diese Instanzen von *Class* sind nicht unmittelbare Bestandteile einer Ontologie. Erst über ein `Declaration`-Axiom (Instanz von *Declaration*) werden sie mittelbarer Bestandteil einer Ontologie. Abbildung 7.3 zeigt den entsprechenden Ausschnitt aus dem OWL-Metamodell. In Abbildung 7.4 ist ein Objektdiagramm der abstrakten Syntax zu sehen.

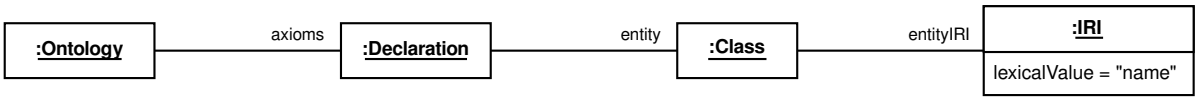


Abbildung 7.4. Objektdiagramm, das die abstrakte Syntax des bei der konkreten Syntax angegebenen Beispiels für die Definition einer Klasse zeigt.

7.1.4 Transformation UML → OWL

Das Konzept, dass Instanzen bzw. Individuen zu Klassen gehören, existiert sowohl in UML als auch in OWL. Da beide Konzepte sehr ähnlich sind, ist eine Transformation von Klassen einfach möglich, wie sie in Abbildung 7.5 gezeigt wird.

[161] Vgl. W3C: OWL 2 Structural specification, Abschnitt 5.8.1.

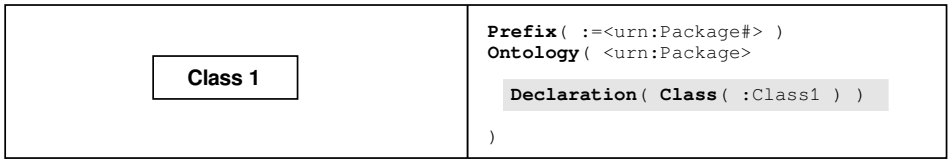


Abbildung 7.5. Beispiel einer Transformation einer Klasse (Instanz des UML-Elementtyps *Class*) in eine Klassendeklaration (Instanzen der OWL-Elementtypen *Class* und *Declaration*).

7.1.5 Transformation OWL → UML

Bei der Transformation muss zwischen benannten (d.h. deklarierten) Klassen auf der einen und unbenannten CE auf der anderen Seite unterschieden werden. Der einfachste Fall liegt vor, wenn es sich um eine benannte Klasse handelt. Dann kommt die in Abbildung 7.6 gezeigte QVT-Transformation zur Anwendung, die eine UML-Klasse mit einem aus dem IRI abgeleiteten Namen erzeugt.

Transformationen von unbenannten CE werden in den späteren Abschnitten zur Generalisierung (`ObjectUnionOf`) und Schnittmenge (`ObjectIntersectionOf`) vorgestellt.

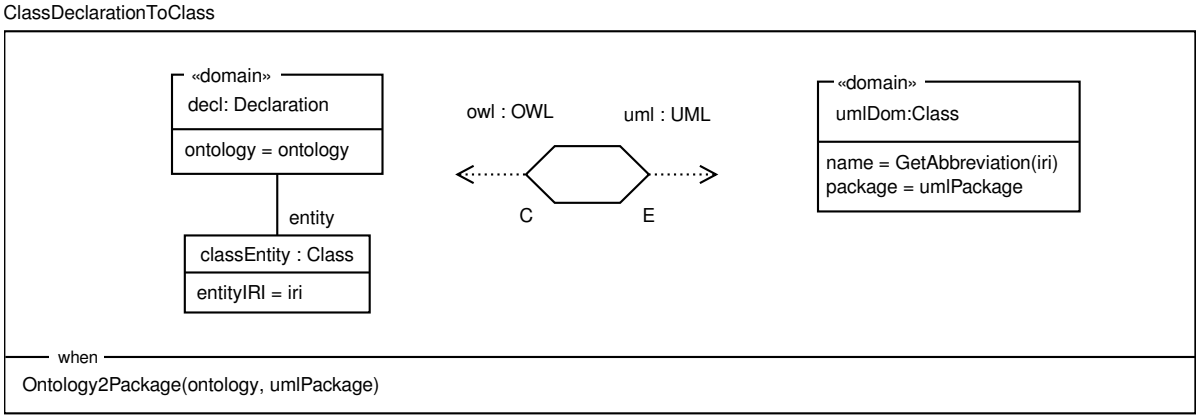


Abbildung 7.6. QVT-Regel zur Transformation von einer Instanz des OWL-Elementtyps *Class* mit dazugehöriger *Declaration*-Instanz in eine Instanz des UML-Elementtyps *Class*.

7.1.6 Ausdeutung für XSD

In XML-Schema lassen sich Elementtypen mit Hilfe von `complexType`-Elementen (Instanzen von *ComplexTypeDefinition*) definieren.

```
1 <xsd:complexType name="Name">
2   . . .
3 </xsd:complexType>
```

Eine Instanz eines solchen Elementtyps ist ein XML-Element. Der Elementtyp eines Elements kann auf zwei Arten festgelegt werden:

- durch den Namen des Elements und die Zuordnung des Namens zu einem Elementtypen in einer *ElementDeclaration*. Dabei kommen sowohl Top-Level-Deklarationen, d.h. direkt mit der *Schema*-Instanz in Beziehung stehende, als auch in anderen `complexType` enthaltene Deklarationen in Frage.
- durch explizite Angabe eines Typs mit Hilfe des `type`-Attributs aus dem Namespace `http://www.w3.org/2001/XMLSchema-instance` – meist als QName (→ 7.2 Namen) `xsi:type` geschrieben.

Enthält das XML-Schema folgende Angaben,

```
1 <xsd:element name="person" type="Person" />
2
3 <xsd:complexType name="Person">
4    <xsd:sequence>
5       <xsd:element name="name" type="Name"/>
6    </xsd:sequence>
7 </xsd:complexType>
8
9 <xsd:complexType name="Name"> . . . </xsd:complexType>
```

so wird im folgenden Instanz-Dokument dem äußersten XML-Element `person` über die *ElementDeclaration* in Zeile 1 des Schemas der Typ `Person` zugewiesen. Der Typ des inneren Elements `name` wird durch die, in der *ComplexTypeDefinition* (Zeilen 3-7 des Schemas) enthaltene, *ElementDeclaration* in Zeile 5 auf `Name` festgelegt.

```
1 <person>
2    <name> . . . </name>
3 </person>
```

Soll dem `name`-Element in Zeile 2 des Instanz-Dokuments ein anderer Typen, z.B. `ExtendedName` zugewiesen werden (bei dem es sich um einen Untertypen von Name handeln muss, damit das Dokument bzgl. des Schemas valide bleibt), so lässt sich dies mit der zweiten Art der Typ-Zuweisung folgendermaßen realisieren:

```
1 <person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2    <name xsi:type="ExtendedName"> . . . </name>
3 </person>
```

7.2 Namen

Für eine eindeutige Identifizierung muss ein Elementtyp einen innerhalb eines Modells eindeutigen Namen besitzen.[162]

7.2.1 Logische Repräsentation

Es gibt verschiedene Arten, Modellelemente *identifizierbar* zu machen, von denen Olivé sechs Methoden nennt.[163] Bis auf die Identifikation der Datentypen eignen sich alle Methoden auch zur Identifikation von Elementtypen. Häufig sind folgende zwei Arten anzutreffen:

Ein Elementtyp wird identifizierbar, wenn das Metamodell einen Beziehungstyp der Art

$$Name(Elementtyp, String)$$

enthält, bei dem es sich um eine einfache Referenz zu *Elementtyp* handelt. Dabei kann sowohl der Name des Beziehungstyps als auch der Datentyp des "Namen" (im obigen Beispiel *String*) variieren.

Eine zweite Möglichkeit, einen Elementtyp E identifizierbar zu machen, besteht in der Verwendung einer zusammengesetzten Referenz, bei der jeder der Elementtypen $E_1, \ldots, E_n$ identifizierbar ist:

$$\{R_1(E, E_1), \ldots, R_n(E, E_n)\}$$

Ein Beispiel für eine Identifikation mit Hilfe einer zusammengesetzten Referenz ist die Modellierung einer Straßenkreuzung, die durch eine Instanz des Elementtyps "Ort" und zwei Instanzen des Elementtyps "Straße" identifiziert wird:

E	Kreuzung
R_1	befindetSichIn
E_1	Ort
R_2	straße1
R_3	straße2
E_2, E_3	Straße

Wichtig ist, dass die Elementtypen "Ort" und "Straße" identifizierbar sind. Dies kann z.B. über den Namen als einfache Referenz erreicht werden.

7.2.2 Repräsentation in UML

Die UML-Spezifikation legt fest, dass jedes Modellelement innerhalb seines Paketes (→ 11.1 Pakete) einen eindeutigen Namen besitzen muss. Dadurch ist sichergestellt, dass jedes Modellelement durch Angabe seines Namens und der Position in der Pakethierarchie eindeutig identifiziert werden kann.

[162] Vgl. Olivé: Conceptual Modeling, S. 44.
[163] Vgl. a. a. O., S. 107 f.

So ist z.B. die UML-Klasse namens “Kurs” in einem Paket “Universität” unterschiedlich von einer Klasse “Kurs” in einem Paket “Finanzwirtschaft”, obwohl beide Klassen denselben Namen besitzen.

Durch folgende zusammengesetzten Referenz werden Instanzen von *Class* (wie jede Instanz des UML-Elementtyps *Type*) identifizierbar:

$$name(Class, String)$$
$$package(Class, Package)$$

Ähnliches gilt für die Namen von klassenabhängigen Attributen (→ 9 Beziehungstypen). Die Namen dieser Attribute müssen nur innerhalb der Klasse, zu der sie gehören, eindeutig sein.

$$name(Property, String)$$
$$class(Property, Class)$$

So ist z.B. ein Attribut “titel”, das zu einer Klasse “Person” gehört, unterschiedlich von einem Attribut “titel” in einer Klasse “Buch”.

Konkrete Syntax

Im Falle von Elementtypen wird der Name, wie in Abbildung 7.1 zu sehen, fett gedruckt in den obersten Teil des Rechtecks geschrieben. Die Platzierung der Namen bei anderen Modellelementen wird in dem jeweiligen Abschnitt beschrieben, der sich mit dem Element befasst.

Abstrakte Syntax

Dadurch, dass der UML-Elementtyp *Class* – wie in Abbildung 7.7 zu sehen – ein Untertyp von *NamedElement* ist, ist ein klassenabhängiges Attribut *name* vorhanden, dessen Wert der Name der Klasse ist.

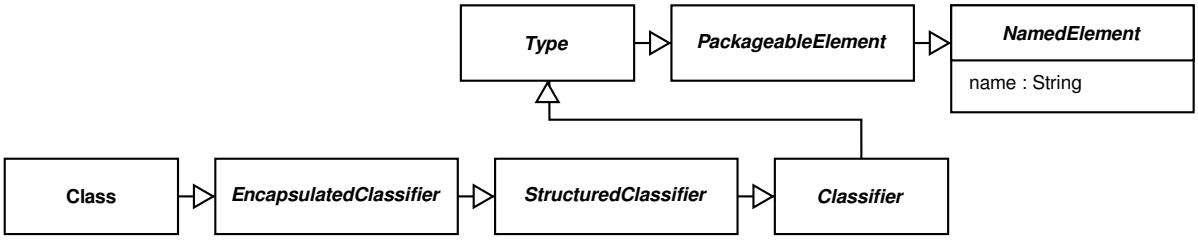


Abbildung 7.7. Ausschnitt aus dem UML-Metamodell, das den Zusammenhang zwischen *Class* und *NamedElement* zeigt.

7.2.3 Repräsentation in OWL

OWL verwendet als Namen von Elementtypen und anderen Elementen (Instanzen von *Entity*) sogenannte Internationalized Resource Identifier gemäß RFC 3987. Dabei haben alle zugewiesenen Namen globale Gültigkeit, unabhängig davon, in welchem Zusammenhang sie eingesetzt werden.

Es handelt sich also um eine einfache Referenz, mit Hilfe derer Elementtypen identifizierbar werden:

$$entityIRI(Entity, IRI)$$

Ein IRI selbst ist wiederum durch seinen lexikalischen Wert *lexicalValue* identifizierbar:

$$lexicalValue(IRI, String)$$

Die OWL 2 DL Namenskonventionen schreiben vor, dass alle Elemente – mit Ausnahme von Individuen – benannt sein müssen. Sind alle Elemente benannt (d.h. es taucht in der Ontologie kein Bezeichner auf, zu dem kein Typ festgelegt wurde), so wird von einer Ontologie mit *consistent declarations* gesprochen. Es darf keine Überschneidung bei den Namen von Klassen und Datentypen geben. Ebenfalls darf es bei den Namen der Properties keine Überschneidungen geben.[164]

Wichtig für die Transformation ist außerdem, dass (benannte) Klassen vom Typ *Class* zwar als *Entity* einen Namen besitzen, eine Vielzahl von Axiomen sich jedoch auf *ClassExpression* bezieht, deren Unterklasse *Class* ebenfalls ist. Diese *ClassExpression* haben keinen Namen und können auch nicht selbst als Axiom Bestandteil einer Ontologie sein. Die vielfältigen Möglichkeiten, wie (anonyme) *ClassExpression* definiert werden können, werden bei den einzelnen Transformationsregeln vorgestellt.

Konkrete Syntax

Es gibt für IRIs eine Langschreibweise, bei der der IRI in spitze Klammern eingebettet wird, sowie die in der OWL-Spezifikation beschriebene Möglichkeit, IRIs verkürzt zu schreiben. Dazu wird für einen Teil eines IRI ein Präfix deklariert, der dann als Abkürzung verwendet werden kann.[165]

Ein IRI in vollständiger Langschreibweise sieht folgendermaßen aus:

```
<http://example.org#Test>
```

Nach der Definition des Präfix

```
Prefix(abc:=<http://example.org#>)
```

[164] Vgl. W3C: OWL 2 Structural specification, Abschnitt 5.8.1.
[165] Vgl. a. a. O., Abschnitt 2.4.

kann in der Ontologie auch diese verkürzte Schreibweise verwendet werden:

```
abc:Test
```

Die Zuordnung eines IRI zu einem (Meta-)Elementtyp erfolgt mit Hilfe eines `Declaration`-Axioms. So definieren die Axiome

```
Declaration( Class( abc:Firma ) )
Declaration( ObjectProperty( abc:arbeitetBei ) )
```

die Zuordnung des IRI `<http://example.org#Firma>` zu einer Instanz des Meta-Elementtyps *Class* und des IRI `<http://example.org#arbeitetBei>` zu einer Instanz des Meta-Elementtyps *ObjectProperty*.

Abstrakte Syntax

Ein IRI besitzt ein klassenabhängiges Attribut, welches den lexikalischen Wert des IRI beinhaltet. Zwei IRIs werden als *strukturell äquivalent*[166] angesehen, wenn ihre lexikalischen Werte identisch sind.

Wie in Abbildung 7.8, die einen Ausschnitt aus dem OWL-Metamodell zeigt, zu sehen ist, wird jeder Instanz von *Entity* eine *IRI*-Instanz zugeordnet, über deren Wert die *Entity*-Instanz referenziert werden kann.

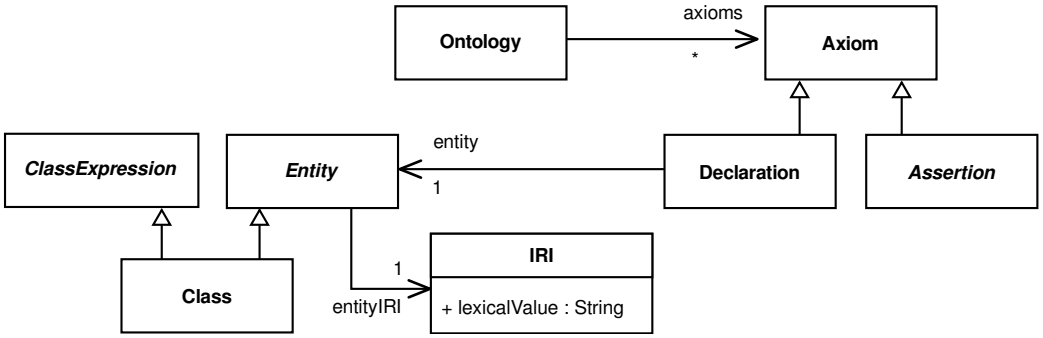


Abbildung 7.8. Ausschnitt aus dem OWL-Metamodell, der die Zuordnung von Namen zu Elementen zeigt.

Elemente, die über einen Namen verfügen und über diesen referenziert werden können, leiten sich vom abstrakten Element *Entity* ab. Diese Elemente sind jedoch nicht unmittelbar Teil einer Ontologie, sondern werden erst durch die Verwendung in einem `Declaration`-Axiom Bestandteil der Ontologie. Dieser Zusammenhang ist in Abbildung 7.8 dargestellt.

[166] Vgl. W3C: OWL 2 Structural specification, Abschnitt 2.1.

Tabelle 7.1. Übersicht über die Erzeugung von IRIs.

Modellelement	**Meta-Elementtyp**	**generierter IRI**
Paket P	*Package*	<urn:P>
Klasse C im Paket P	*Class*	<urn:P#C>
klassenabhängiges Attribute A der Klasse C	*Property*	<urn:P#C_A>
Endpunkt T_1:N_1 einer Assoziation A mit Enden T_1:N_1 und T_2:N_2	*Property*	<urn:P#T_2_N_1>

7.2.4 Transformation UML → OWL

Da – anders als bei UML – ein OWL-Modellelement nicht nur innerhalb eines Pakets, sondern global einen eindeutigen Namen benötigt, muss bei der Transformation jedem Modellelement ein solcher global eindeutiger Name zugewiesen werden. Daher werden in den Transformationsregel entsprechende global eindeutige IRI generiert. Tabelle 7.1 gibt einen Überblick, auf welche Weise IRIs erzeugt werden, so dass die Namen global eindeutig sind.

7.2.5 Transformation OWL → UML

Da Namen in der UML nur innerhalb eines Pakets eindeutig sein müssen, ist es sinnvoll, die *remaining characters* einer verkürzt geschriebenen IRI als Name für UML-Elemente zu verwenden (vorausgesetzt, der Präfix-IRI ist dem Paket zugeordnet). Dadurch werden die erzeugten UML-Modell besser lesbar. Eine QVT-Query sorgt für diese Namenszuordnung.

7.2.6 Ausdeutung für XSD

Zur Referenzierung von Modellelementen wird eine Kombination von Namensraum (ein Uniform Resource Identifier (URI)) und einem lokalen Namen verwendet. Als abkürzende Schreibweise gibt es die Möglichkeit, einem Namensraum-URI ein Präfix zuzuweisen und die Namen in der Form `Präfix:lokaler Name` zu schreiben. Für diese Form existiert bei XSD der Datentyp `QName`. Innerhalb eines Namensraums (also auch innerhalb eines Schemas, das sich in einem `targetNamespace` befindet) müssen die Namen eindeutig sein.

Fast alle XSD-Modellelemente (bis auf die "Helfer"-Modellelemente (→ 3.3.2 XML Schema Component Model)) können benannt werden, bei Attribut- und Elementdeklarationen ist dies sogar obligatorisch. Die Vergabe des Namens erfolgt über das `name`-Attribut.

```
1 <xsd:complexType name="Name des Typs"> . . . </xsd:complexType>
2
3 <xsd:element name="Name des Elements"> . . . </xsd:element>
```

7.3 Vererbung

Oft tritt bei der konzeptuellen Modellierung die Situation auf, dass die Population eines Elementtyps notwendigerweise ebenfalls Population eines anderen Elementtyps ist. In diesem Fall wird von einer Vererbungsbeziehung zwischen diesen beiden Elementtypen gesprochen. Elementtypen, die in einer Vererbungsbeziehung stehen, werden oft mit folgenden Begriffspaaren bezeichnet:

- subtype – supertype
- Kindtyp – Elterntyp
- Untertyp – Über-/Obertyp
- hyponym – hypernym[167]

7.3.1 Logische Repräsentation

Stehen zwei Elementtypen in einer Vererbungsbeziehung, so ist jede Instanz des Elementtyps E_1 ebenfalls Instanz des Elementtyps E_2.

$$E_1(e) \rightarrow E_2(e)$$

Diese Beziehung wird oft als *IsA*-Beziehung bezeichnet.[168]

7.3.2 Repräsentation in UML

In UML ist die Instanz einer Klasse ebenfalls Instanz ihrer Superklassen.[169] Die Vererbungsbeziehung von Elementtypen ist damit transitiv.

Konkrete Syntax

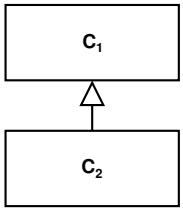


Abbildung 7.9. Grafische UML-Syntax der Vererbung zweier Elementtypen.

[167] Vgl. Olivé: Conceptual Modeling, S. 214.
[168] Vgl. a. a. O.
[169] OMG: UML Infrastructure, S. 52: *"..., each instance of the specific classifier is also an instance of the general classifier."*

Grafisch wird eine Vererbungsbeziehung zweier Elementtypen – wie in Abbildung 7.9 gezeigt – durch einen Pfeil zwischen den sie repräsentierenden Symbolen dargestellt. Die Pfeilspitze in Form eines weißen Dreiecks zeigt auf die Oberklasse.

Abstrakte Syntax

Eine Vererbungsbeziehung wird durch eine Instanz des UML-Elementtyps *Generalization* dargestellt, wie in Abbildung 7.10 gezeigt. Über die Beziehung mit Rollennamen *general* ist der Obertyp, über die Beziehung mit Rollennamen *specific* ist der Untertyp zugeordnet. Dabei muss genau eine Klasse als Obertyp und eine Klasse als Untertyp definiert sein. Es gibt hingegen keine Einschränkungen, in wie vielen Vererbungsbeziehungen eine Klasse stehen darf.

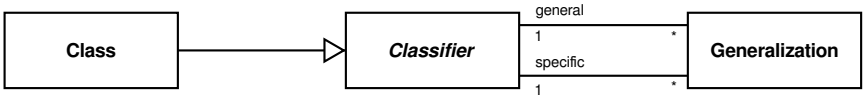


Abbildung 7.10. Ausschnitt aus dem UML-Metamodell, das die Nachbarschaft des UML-Elementtyps *Generalization* zeigt.

7.3.3 Repräsentation in OWL

In OWL ist die Vererbungsbeziehung von Elementtypen über Untermengen der Klassen-Interpretation definiert.[170] Da die Untermengen-Beziehung transitiv ist, ist auch die Vererbungsbeziehung von Elementtypen in OWL transitiv.

Konkrete Syntax

Eine Vererbungsbeziehung zwischen zwei Elementtypen wird in OWL durch das `SubClassOf`-Axiom beschrieben:

`SubClassOf(`E_1`,`E_2`)`

Alle Individuen, die Instanz der Klasse E_1 sind, sind auch Instanz der Klasse E_2.

Abstrakte Syntax

Eine Vererbungsbeziehung wird mit Hilfe einer Instanz des OWL-Elementtyps *SubClassOf*, bei der es sich um ein Axiom handelt und somit top-level einer Ontologie zugeordnet ist, dargestellt. Abbildung 7.11 zeigt *SubClassOf* mit zugehörigen Modellelementen. Über die Beziehung mit Rollennamen *superClassExpression* ist der Obertyp, über die Beziehung mit Rollennamen *subClassExpression* der Untertyp zugeordnet. Dabei muss genau eine CE als

[170] Vgl. W3C: OWL 2 Direct Semantics, Abschnitt 2.3.1.

Obertyp und eine CE als Untertyp definiert sein. Es gibt dagegen keine Einschränkungen, in wie vielen Vererbungsbeziehungen eine CE stehen darf.

Abbildung 7.11. Ausschnitt aus dem OWL-Metamodell, der die Nachbarschaft von *SubClassOf* zeigt.

7.3.4 Transformation UML → OWL

Aufgrund der sehr ähnlichen Struktur und Semantik (speziell Transitivität) von *Generalization*-Elementen auf der einen und *SubClassOf*-Axiomen auf der anderen Seite, ist eine Transformation leicht möglich. Zu jeder Instanz des UML-Elementtyps *Generalization* wird eine Instanz des OWL-Elementtyps *SubClassOf* angelegt, wenn sich die beiden verbundenen Elementtypen transformieren lassen. Da es sich bei dem `SubClassOf`-Element um ein Axiom handelt, ist es notwendig, dies mit der beinhaltenden Ontologie zu verknüpfen. Abbildung 7.12 zeigt die entsprechende QVT-Regel.

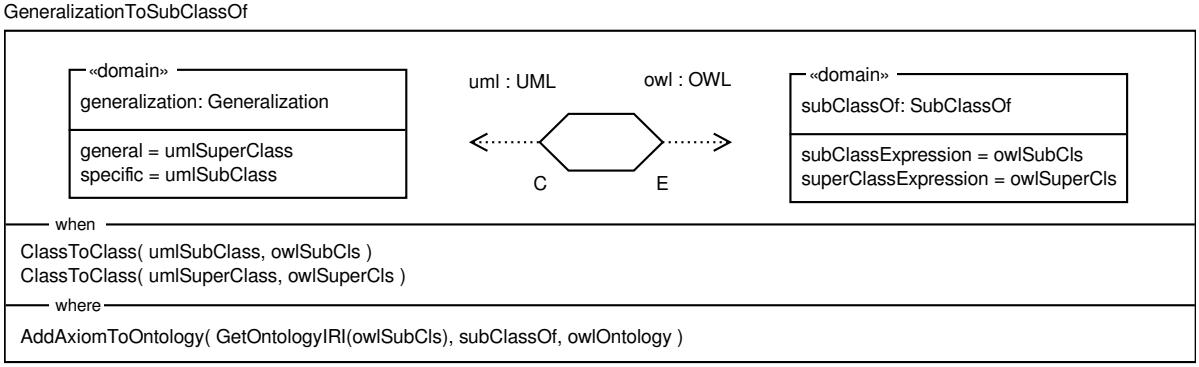


Abbildung 7.12. QVT-Regel zum Transformieren einer Instanz des UML-Elementtyps *Generalization* in eine Instanz des OWL-Elementtyps *SubClassOf*.

7.3.5 Transformation OWL → UML

Bei der Transformation eines Axioms der Form `SubClassOf(`C_c C_p`)` müssen folgende Fälle unterschieden werden:

(a) Sowohl Unterklasse C_c als auch Oberklasse C_p lassen sich in eine UML-Klasse transformieren.

(b) Bei einer von beiden Elementtypen handelt es sich um eine CE, bei der die Zugehörigkeit durch notwendige und hinreichende Bedingungen (→ 6.7 Notwendige und hinreichende Bedingungen) beschrieben ist.

(c) Mindestens einer der beiden Elementtypen lässt sich aus anderen Gründen (z.B. wegen Komplementbildung) nicht in eine UML-Klasse transformieren.

Für den Fall (a), dass sich beide Elementtypen abbilden lassen, ist die Transformation einfach. Abbildung 7.13 zeigt ein Beispiel für die Transformation eines `SubClassOf`-Axioms in diesem einfachsten Fall. Es wird, wie in Abbildung 7.14 gezeigt, mit Hilfe einer Instanz des UML-Elementtyps *Generalization* eine Vererbungsbeziehung zwischen den beiden transformierten UML-Klassen hergestellt.

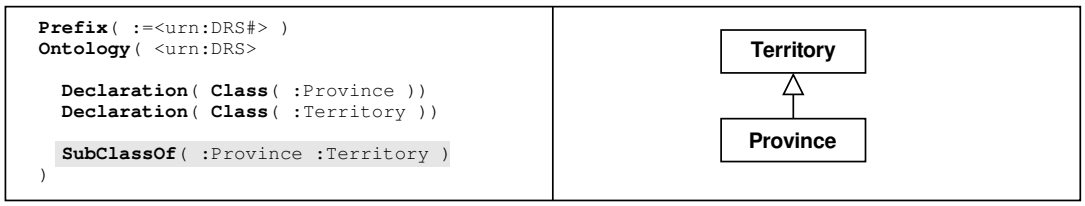


Abbildung 7.13. Beispiel für die Transformation eines `SubClassOf`-Axioms.

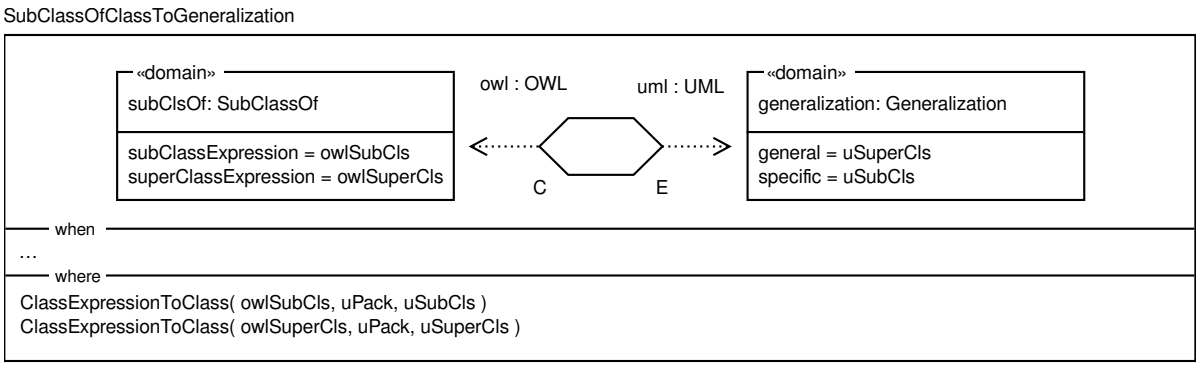


Abbildung 7.14. QVT-Regel, die Unterklassen-Beziehungen zwischen zwei Klassen in eine Generalisierung transformiert.

```
1 SubClassOf(
2   DataMinCardinality( 1 :hatISBN )
3   :Buch
4 )
```

Listing 7.1. Die Zugehörigkeit zur Unterklasse C_c ist über eine hinreichende Bedingung definiert: "Jedes Objekt, das mindestens eine ISBN hat, ist ein Buch." (Es gibt auch Bücher, die nicht mindestens eine ISBN haben.)

Handelt es sich im Fall (b) bei einer der beiden Elementtypen um eine durch notwendige und hinreichende Bedingungen definierte CE, so ist zu unterscheiden, ob es sich dabei um die Unterklasse C_c oder die Oberklasse C_p handelt.

Handelt es sich um die Unterklasse C_c, liegt ein Fall wie in Listing 7.1 vor. Der Elementtyp C_c ist über eine Formel ϕ definiert. Jedes Individuum e ist genau dann Instanz des Elementtyps,

wenn es die Formel erfüllt:

$$C_c(e) \leftrightarrow \phi(e)$$

Dass ein Individuum e Instanz von C_p ist, wird durch $C_p(e)$ angegeben. Aus der Vererbungsbeziehung

$$C_c(e) \rightarrow C_p(e)$$

folgt nun (durch Ersetzen von $C_c(e)$) unmittelbar

$$\phi(e) \rightarrow C_p(e)$$

Das Erfüllen der Formel $\phi(e)$ ist also hinreichende Bedingung dafür, dass ein Objekt Instanz des Elementtyps C_p ist. Da UML automatische Klassifikation anhand hinreichender Bedingungen nicht unterstützt, ist dieser Fall nicht transformierbar.

```
1 SubClassOf(
2   :Buch
3   DataMinCardinality( 1 :hatISBN )
4 )
```

Listing 7.2. Die Klasse C_c = `Buch` ist Unterklasse der mit Hilfe der hinreichenden Bedingung "hat mindestens eine ISBN" definierten anonymen Klasse C_p.

Anders sieht die Situation im Fall (b) aus, wenn der über notwendige und hinreichende Bedingungen definierte Elementtyp als Obertyp C_p auftritt. Listing 7.2 zeigt ein Beispiel für diesen Fall. Alleine für sich genommen wäre der Elementtyp C_p nicht in eine UML-Klasse transformierbar. Dadurch, dass der Elementtyp C_p jedoch innerhalb des `SubClassOf`-Axioms als Obertyp auftritt, geht die notwendige Bedingung der CE auf den Unter-Elementtyp C_c über, wie im Folgenden zu sehen ist:

Der Elementtyp C_p ist über eine Formel ϕ definiert. Jedes Individuum e ist genau dann Instanz des Elementtyps, wenn es die Formel erfüllt:

$$C_p(e) \leftrightarrow \phi(e)$$

Dass ein Individuum e Instanz des Unter-Elementtyps C_c ist, wird durch $C_c(e)$ angegeben. Aus der Vererbungsbeziehung

$$C_c(e) \rightarrow C_p(e)$$

folgt nun (durch Ersetzen von $C_p(e)$) unmittelbar

$$C_c(e) \rightarrow \phi(e)$$

Das Erfüllen der Formel $\phi(e)$ ist also notwendige Bedingung dafür, dass ein Objekt Instanz des Elementtyps C_c ist. Im Beispiel von Listing 7.2 bedeutet dies: "Jedes Buch muss mindestens eine ISBN besitzen." Solche notwendigen Bedingungen sind wiederum problemlos in UML transformierbar.

7.3.6 Ausdeutung für XSD

XSD kennt zwei Arten der Vererbung zwischen komplexen Typen: Erweiterung `extension` und Spezialisierung `restriction`. Die Art der Vererbung beeinflusst, ob der Untertyp

▷ mehr Beziehungen als der Obertyp bzw.

▷ die gleichen oder weniger Beziehungen mit eingeschränktem Zielbereich besitzt.

Auf die *IsA*-Beziehung der Elementtypen hat die Wahl jedoch keinen Einfluss.

```
1 <xsd:complexType name="E1">
2    <xsd:complexContent>
3       <xsd:restriction base="E2"/>
4    </xsd:complexContent>
5 </xsd:complexType>
```

Da jedes XML-Element vom Typ *E*2 auch vom Typ *E*1 ist, kann es – z.B. mit Hilfe von `xsi:type` (→ 7 Elementtypen) – überall dort eingesetzt werden, wo auch der Typ *E*1 zulässig ist.

7.4 Abstrakte Elementtypen

Als *abstrakt* wird Elementtyp bezeichnet, der nicht instantiiert werden kann: An abstract class(ifier) "*does not provide a complete declaration and can typically not be instantiated.*"[171]

Somit könnte man annehmen, Modelle, die abstrakte Klassen enthalten, seien damit per Definition inkonsistent.[172] Wahler et. al. geben einen Lösungsvorschlag für dieses scheinbare Problem: Eine abstrakte Klasse *C* in einem Modell ist nur dann sinnvoll, wenn auch eine Unterklasse *C′* definiert wird. Andernfalls wäre *C* überflüssig und könnte aus dem Modell entfernt werden. Da es Instanzen von *C′* geben kann, kann es auch für *C* Instanzen geben, da gemäß der UML-Spezifikation die Instanzen einer Klasse automatisch auch Instanzen all ihrer Superklassen sind.[173]

7.4.1 Repräsentation in UML

Die UML erlaubt es, abstrakte Klassen zu definieren, zu denen keine direkten Instanzen (siehe Abbildung 7.15) erzeugt werden dürfen. Nur für Unterklassen ist eine Instantiierung zulässig.

Konkrete Syntax

In UML existieren abstrakte Elementtypen als abstrakte Klassen. Abstrakte Klassen werden in UML gekennzeichnet, indem der Klassenname kursiv geschrieben oder die Klasse mit dem Stereotyp «abstract» versehen ist, wie in Abbildung 7.16 zu sehen.

[171] OMG: UML Infrastructure, S. 84.
[172] Vgl. Wahler, M. et al.: Efficient Analysis of Pattern-Based Constraint Specifications, in: Software and Systems Modeling 9/2, Heidelberg 2010 ⟨URL: http://dx.doi.org/10.1007/s10270-009-0123-6⟩, S. 5.
[173] Vgl. a. a. O.

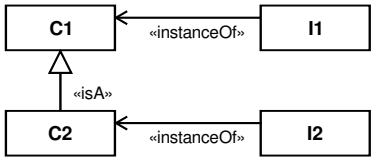


Abbildung 7.15. Direkte und indirekte Instanzen einer Klasse: I1 ist eine direkte Instanz von C1, I2 ist eine indirekte Instanz.

Abbildung 7.16. Konkrete Syntax für abstrakte Klassen. Links die Darstellung mit kursiver Schrift, rechts mit dem Stereotyp «abstract».

Abstrakte Syntax

In der abstrakten Syntax findet sich die Information, dass eine Klasse abstrakt ist, in Form eines klassenabhängigen Attributes des Elementtyps *Classifier* wieder, siehe Abbildung 7.17.

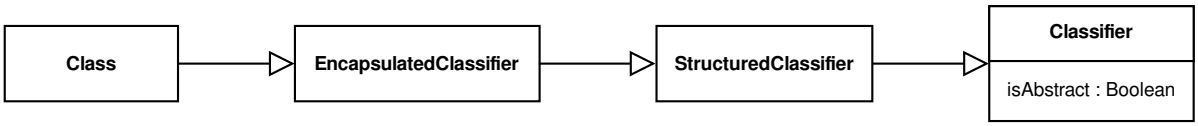


Abbildung 7.17. Auszug aus dem UML-Metamodell, der zeigt, dass *Class* ein Untertyp von *Classifier* mit dem Attribut *isAbstract* ist.

7.4.2 Repräsentation in OWL

OWL kennt kein Sprachkonstrukt, mit dem angegeben werden kann, dass eine Klasse kein Individuum enthalten darf, das ihr direkt und keiner Unterklasse zugeordnet ist. Angenommen, es gäbe eine OWL-Klasse C, die kein Individuum enthalten dürfe, und eine nicht-leere Klasse C', die Unterklasse von C ist:

$$C'(x) \rightarrow C(x)$$

Sei nun i ein Individuum der Klasse C':

$$C'(i)$$

Aus der Kombination beider Aussagen folgt unmittelbar: $C(i)$, was jedoch ein Widerspruch zu der Annahme ist, die Klasse C enthalte keine Individuen.

7.4.3 Transformation UML → OWL

In der Regel tauchen abstrakte Klassen in Verbindung mit Generalisierungen auf. In Verbindung damit ist es möglich, sie wie "normale" Klassen zu behandeln. Es bleibt jedoch

die Einschränkung, dass nicht sichergestellt werden kann, dass eine abstrakte Klasse keine direkten Instanzen enthält.

7.4.4 Transformation OWL → UML

Bei der Transformation einer Instanz des OWL-Elementtyps `ObjectUnionOf` in Verbindung mit Vererbungsbeziehungen spielen abstrakte Elementtypen eine Rolle. Dies wird im Abschnitt über Generalisierungen (→ 7.6 Generalisierung) behandelt.

7.4.5 Ausdeutung für XSD

XSD sieht abstrakte Elementtypen vor. Es lassen sich sowohl komplexe Typen (`complexType`) als auch einfache Typen (`simpleType`) als abstrakt markieren. Diese können dann nicht direkt Elementen als Typen zugewiesen werden. Sie können jedoch als Basis anderer Typen bei Erweiterung oder Spezialisierung dienen.

```
1 <complexType name="E1" abstract="true">
2 </complexType>
```

7.5 Elementtypen mit fester Population

Ein Elementtyp mit fester Population besteht aus einer vorgegebenen Menge von Objekten, die die Population dieses Elementtyps bilden. Es ist nicht möglich, weitere Objekte als Instanz dieses Elementtyps auszuweisen.

7.5.1 Logische Repräsentation

Ist ein Objekt e Instanz eines Elementtyps E mit der festen Population $\{e_1, \ldots e_n\}$, so gilt:

$$E(e) \rightarrow e \in \{e_1, \ldots e_n\}$$

7.5.2 Repräsentation in UML

UML besitzt keine Möglichkeit, anzugeben, dass die Population eines Elementtyps nur aus einer festgelegten Menge von Objekten bestehen darf. Für Datentypen ist es mit Hilfe von *Enumeration* möglich, Elementtypen mit fester Population zu definieren (→ 8 Datentypen).

Die konkrete und abstrakte Syntax für eine Enumeration wird im Abschnitt über Datentypen (→ 8 Datentypen) vorgestellt.

7.5.3 Repräsentation in OWL

In OWL lassen sich Elementtypen mit fester Population sowohl für Individuen als auch für Werte von Datentypen definieren. Bei einer festen Menge von Individuen handelt es sich

um eine CE, bei einer vorgegebenen Menge von Werten um eine Data Range. Die dabei aufgeführten Individuen müssen dabei nicht unbedingt Instanz desselben Elementtyps sein. Ebenso müssen die aufgeführten Werte nicht demselben Datentyp angehören.

Konkrete Syntax

Mit Hilfe der `ObjectOneOf`-CE wird ein Elementtyp mit einer festen, angegebenen Population definiert:

`ObjectOneOf(` $o_1 \ldots o_n$ `)`

Die konkrete Syntax für Datentypen mit festgelegter Wertemenge wird in Abschnitt 8.4 beschrieben.

Abstrakte Syntax

In abstrakter Syntax wird eine `ObjectOneOf`-CE durch eine Instanz von *ObjectOneOf* dargestellt, die in Beziehung zu einer oder mehreren Instanzen von *Individual* steht. Der entsprechende Ausschnitt aus dem Metamodell ist in Abbildung 7.18 dargestellt.

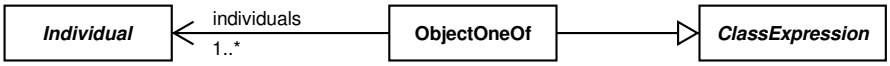


Abbildung 7.18. Der *ObjectOneOf* betreffende Auszug aus dem OWL-Metamodell.

Die abstrakte Syntax für Datentypen mit festgelegter Wertemenge werden in Abschnitt 8.4 beschrieben.

7.5.4 Transformation UML ↔ OWL

Da sich in UML nur Datentypen mit fester Population beschreiben lassen, ist eine Transformation der `ObjectOneOf`-CE nicht möglich. Beide Transformationsrichtungen für Datentypen werden jeweils in den entsprechenden Abschnitten in Kapitel 8 beschrieben.

7.5.5 Ausdeutung für XSD

Für `complexType` gibt es keine Möglichkeit, die Menge der Elemente festzulegen, die diesen Typ haben. Für Datentypen lässt sich eine Menge erlaubter Werte angeben – Näheres dazu im Abschnitt über Datentypen (→ 8 Datentypen).

7.6 Generalisierung

Bei einer Generalisierung handelt es sich um eine Verallgemeinerung einer Vererbungsbeziehung. Es werden meist mehr als zwei Elementtypen in Beziehung gesetzt. Außerdem ist es möglich, anzugeben, ob es sich um eine vollständige und/oder überschneidungsfreie Generalisierung handelt.

7.6.1 Logische Repräsentation

Formal lässt sich die Tatsache, dass ein Elementtyp E die Elementtypen $E_1, \ldots, E_n$ generalisiert, folgendermaßen ausdrücken:

$$E_i(e) \rightarrow E(e) \qquad i = 1, \ldots n$$

Handelt es sich um eine vollständige Generalisierung, so muss zusätzlich jede Instanz des Elementtyps E auch Instanz von mindestens einem der Elementtypen $E_1, \ldots, E_n$ sein:

$$E_(e) \rightarrow E_1(e) \vee \cdots \vee E_n(e) \qquad i = 1, \ldots n$$

Bei einer überschneidungsfreien Generalisierung darf jede Instanz des Elementtyps E höchstens Instanz von einem der Elementtypen $E_1, \ldots, E_n$ sein:

$$E_i(e) \rightarrow \neg E_1(e) \wedge \cdots \wedge \neg E_{i-1}(e) \wedge \neg E_{i+1}(e) \wedge \cdots \wedge \neg E_n(e)$$

7.6.2 Repräsentation in UML

In UML lässt sich die Generalisierung von Elementtypen ähnlich wie die Vererbung von Elementtypen darstellen. Auch Angaben darüber, ob eine Generalisierung komplett und/oder überschneidungsfrei (disjunkt) ist, lassen sich in UML ausdrücken, die dafür vier Eigenschaften kennt:

- complete – Es gelten die oben beschriebenen Einschränkungen für eine vollständige Generalisierung.
- incomplete – Es werden keine Einschränkungen hinsichtlich der Vollständigkeit der Generalisierung gemacht.
- disjoint – Es gelten die oben beschriebenen Einschränkungen für eine überschneidungsfreie Generalisierung.
- overlapping – Es werden keine Einschränkungen hinsichtlicher der Überschneidungsfreiheit der Generalisierung gemacht.

Konkrete Syntax

Grafisch wird in UML eine Generalisierung ähnlich wie die Vererbung von Elementtypen dargestellt. Eine Baumstruktur verbindet die spezialisierten Klassen mit der generalisierten Klasse. Die Pfeilspitze in Form eines weißen Dreiecks zeigt auf die generalisierte Klasse.

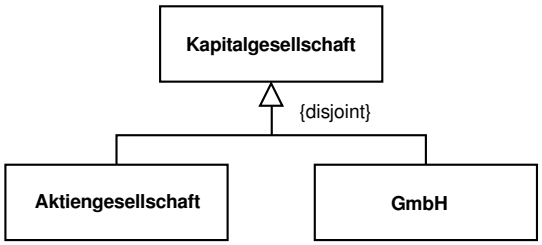


Abbildung 7.19. Beispiel für ein *GeneralizationSet*.

Die Eigenschaften der Generalisierung werden grafisch durch die Angabe der oben aufgezählten vier Schlüsselwörter in geschweiften Klammern in der Nähe des gemeinsamen Teils des Baumes oder der Pfeilspitze dargestellt. Abbildung 7.19 zeigt ein Beispiel für eine Generalisierung in UML. Weitere Beispiele mit unterschiedlichen Eigenschaften sind in Abbildung 7.25 und Abbildung 7.26 zu finden.

Abstrakte Syntax

Mit einer Instanz von *GeneralizationSet* können mehrere – jeweils zwei Elementtypen verbindende – Instanzen von *Generalization* zusammengefasst werden. Mit Hilfe der Attribute *isCovering* sowie *isDisjoint* der *GeneralizationSet*-Instanz lassen sich die Eigenschaften der Generalisierung abbilden. Da eine *Generalization* für *Classifier* und nicht nur speziell für *Class* definiert ist, lassen sich auch Generalisierungen von Datentypen (→ 8 DATENTYPEN) erstellen. Abbildung 7.20 zeigt den entsprechenden Abschnitt im UML-Metamodell, Abbildung 7.21 ein Objektdiagramm mit dem zum obigen Beispiel passenden Modell.

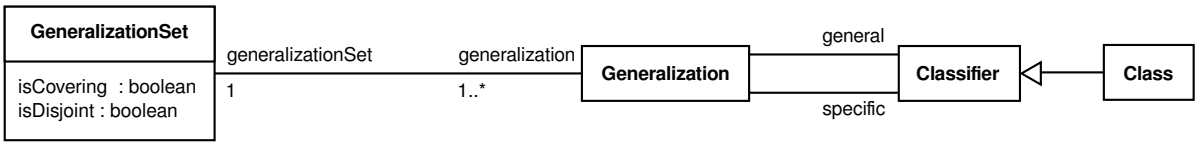


Abbildung 7.20. Der *GeneralizationSet* betreffende Auszug aus dem UML-Metamodell.

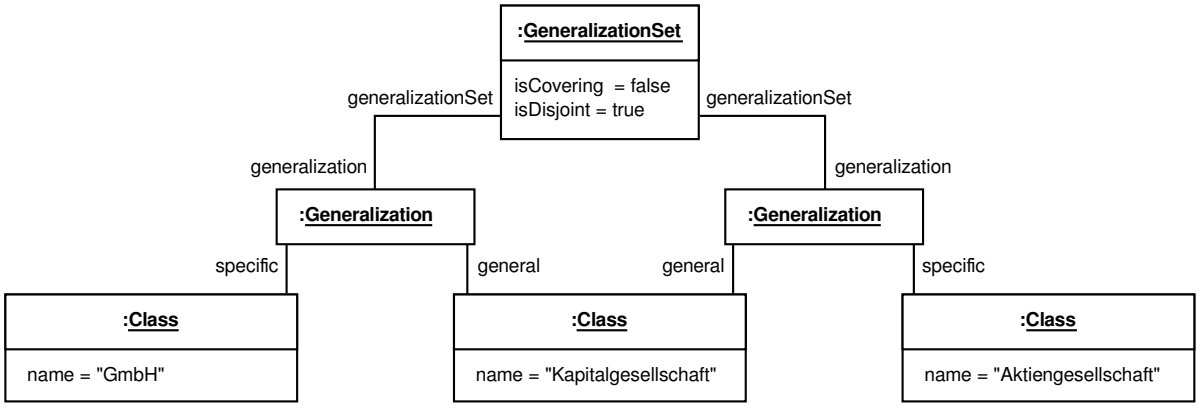


Abbildung 7.21. Objektdiagramm, das die abstrakte Syntax des in Abb. 7.19 gezeigten Beispiels darstellt.

7.6.3 Repräsentation in OWL

Neben der einfachen Vererbung mit Hilfe eines `SubClassOf`-Axioms existiert in OWL ein weiteres Axiom, mit dem für die Elementtypen $E_1, E_2, \ldots, E_n$ beschrieben werden kann, dass $E_2 \ldots E_n$ eine vollständige, überschneidungsfreie Partition von E_1 darstellen.

Konkrete Syntax

Definiert wird diese vollständige, überschneidungsfreie Partition in OWL mit Hilfe des `DisjointUnion`-Axioms, das eine *Class* mit mindestens zwei *ClassExpresion* verbindet:

```
DisjointUnion( :Person :NaturalPerson :LegalPerson )
```

Abstrakte Syntax

Bei der abstrakten Syntax ist zu beachten, dass es sich bei `DisjointUnion` um ein Axiom handelt, das also auf oberster Ebene in einer Ontologie auftritt. Während die Elementtypen, die die Teile der Partition bilden, beliebige *ClassExpression* sein dürfen, muss es sich bei dem zu unterteilenden Elementtypen um eine benannte *Class* handeln. Abbildung 7.22 zeigt den zugehörigen Abschnitt aus dem OWL-Metamodell, Abbildung 7.23 ein Objektdiagramms des zum obigen Beispiel gehörenden Modells.

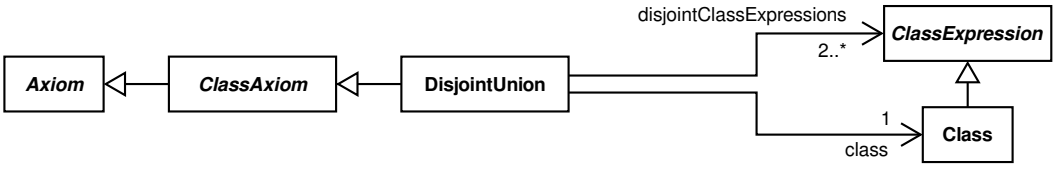


Abbildung 7.22. Der *DisjointUnion* betreffende Auszug aus dem OWL-Metamodell.

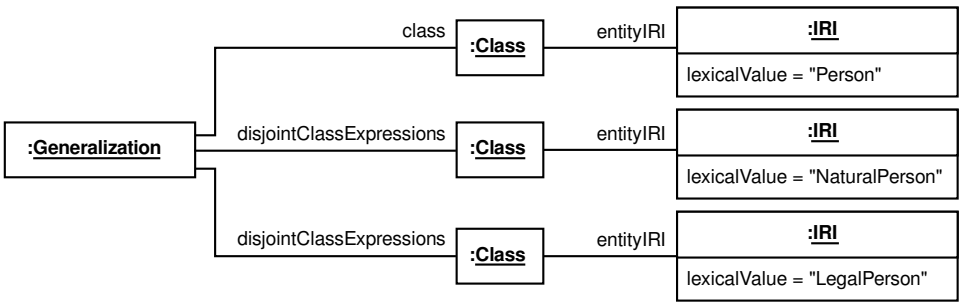


Abbildung 7.23. Objektdiagramm, das die abstrakte Syntax des obigen Beispiels für die konkrete Syntax zeigt.

7.6.4 Transformation UML → OWL

Aufgrund der bei UML möglichen Angaben zur Vollständigkeit und/oder Überschneidungsfreiheit von Generalisierungsbeziehungen lassen sich bei der Transformation verschiedene Fälle identifizieren, die im Folgenden vorgestellt werden:

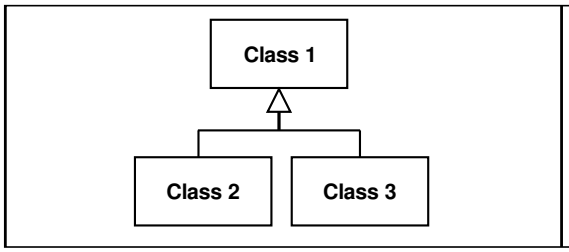
- ▻ Allgemeiner Fall
- ▻ Überschneidungsfreie Generalisierung
- ▻ Überschneidungsfreie und vollständige Generalisierung
- ▻ Vollständige (aber nicht überschneidungsfreie) Generalisierung
- ▻ Generalisierung von Datentypen — wird im Kapitel 8 über Datentypen behandelt

Allgemeiner Fall

Die Konzepte von Spezialisierung bzw. Generalisierung in UML und OWL sind sehr ähnlich. Wenn E' eine spezialisierter Untertyp eines Elementtyps E ist, und i eine Instanz bzw. ein Individuum ist, dann gilt in beiden Fällen $E'(i) \rightarrow E(i)$. Daher ist die Transformation relativ leicht. Für jede Generalisierungsbeziehung "E ist Generalisierung von E'" (bzw. "E' ist Spezialisierung von E)" wird das Axiom `SubClassOf( E' E )` der Ontologie hinzugefügt. Ein Beispiel für diesen allgemeine Fall ist in Abbildung 7.24 dargestellt.

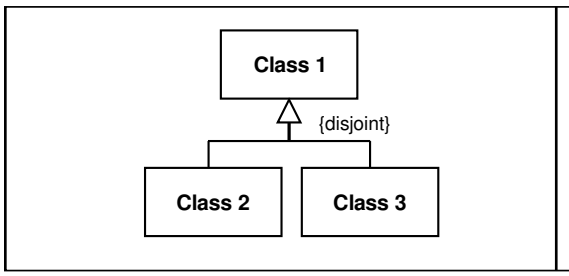
Überschneidungsfreie Generalisierung

Eine Generalisierung ist überschneidungsfrei (disjunkt), wenn eine Instanz eines Untertyps nicht gleichzeitig Instanz eines anderen Untertyps der Generalisierung sein darf. Diese Art von Einschränkung kann durch das Hinzufügen eines `DisjointClasses`-Axioms (Instanz des OWL-Elementtyps *DisjointClasses*), das alle Unter-Elementtypen der Generalisierung enthält, erreicht werden. Dieser Fall ist in Abbildung 7.25 dargestellt.




```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Class1 ) )
  Declaration( Class( :Class2 ) )
  Declaration( Class( :Class3 ) )

  SubClassOf( :Class2 :Class1 )
  SubClassOf( :Class3 :Class1 )
)
```

Abbildung 7.24. Beispiel für die Transformation einer Generalisierungsbeziehung ohne Einschränkungen.

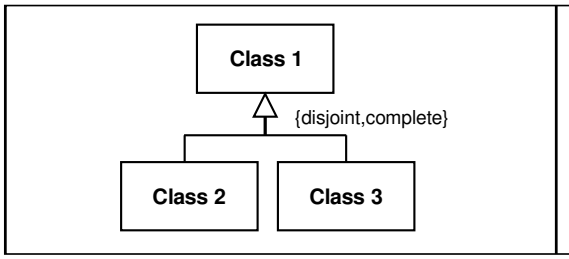



```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Class1 ) )
  Declaration( Class( :Class2 ) )
  Declaration( Class( :Class3 ) )

  SubClassOf( :Class2 :Class1 )
  SubClassOf( :Class3 :Class1 )
  DisjointClasses( :Class2 :Class3 )
)
```

Abbildung 7.25. Beispiel für die Transformation einer überschneidungsfreien (aber nicht unbedingt vollständigen) Generalisierung.

Überschneidungsfreie und vollständige Generalisierung

Ist eine Generalisierung überschneidungsfrei und vollständig, so kann ein stärkeres Axiom verwendet werden. Ein `DisjointUnion(` $E\ E'_1\ \ldots\ E'_n$`)`-Axiom (bei dem es sich letztlich nur um eine abkürzende Schreibweise handelt, siehe Tabelle 3.1) besagt, dass jedes Individuum, das Instanz des Elementtyps E ist, Instanz von genau einem Elementtyp E_i ist und jedes Individuum, das Instanz von E_i ist, automatisch auch zur Population von E gehört. Dieser Fall ist in Abbildung 7.26 dargestellt.




```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Class1 ) )
  Declaration( Class( :Class2 ) )
  Declaration( Class( :Class3 ) )

  DisjointUnion( :Class1 :Class2 :Class3 )
)
```

Abbildung 7.26. Beispiel für die Transformation einer überschneidungsfreien und vollständigen Generalisierung.

Vollständige Generalisierung

Ähnlich ist die Situation, wenn die Generalisierung zwar vollständig, aber nicht überschneidungsfrei ist. In diesem Fall wird das `DisjointClasses`-Axiom der in der abkürzenden Schreibweise eines `DisjointUnion`-Axioms zusammengefassten Axiome (siehe Tabelle 3.1) nicht benötigt. Somit ergibt sich die in Abbildung 7.27 dargestellte Lösung unter Verwendung einer `ObjectUnionOf`-CE und eines `EquivalentClasses`-Axioms.

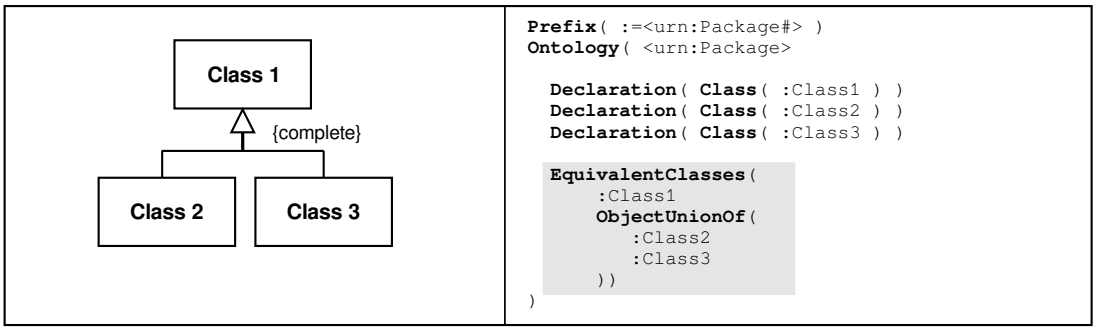


Abbildung 7.27. Beispiel für die Transformation einer vollständigen aber nicht überschneidungsfreien Generalisierung.

7.6.5 Transformation OWL → UML

ObjectUnionOf

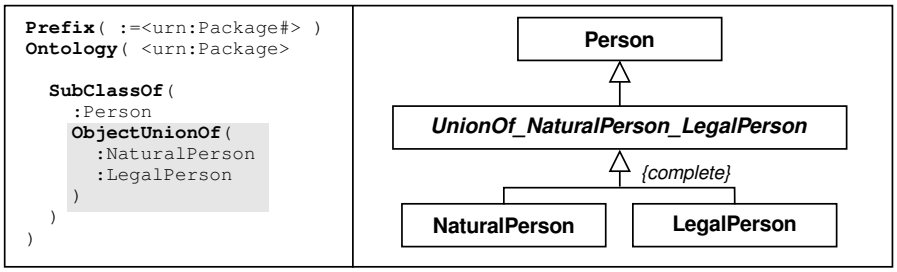


Abbildung 7.28. Beispiel für die Transformation einer Instanz des OWL-Elementtyps *ObjectUnionOf* in eine abstrakte Klasse (Instanz des UML-Elementtyps *Class*) und Vererbungsbeziehungen mit *GeneralizationSet*.

Eine `ObjectUnionOf`($E_1 \ldots E_n$) definiert einen Elementtyp, zu dessen Population die alle diejenigen Individuen gehören, die Instanz von einem oder mehreren Elementtypen $E_1 \ldots E_n$ sind. Lassen sich $E_1 \ldots E_n$ in UML-Klassen $C_1 \ldots C_n$transformieren, so kann die `ObjectUnionOf` als abstrakte Klasse abgebildet werden. Zwischen der abstrakten Klasse und den Bestandteilen der Union $C_1 \ldots C_n$ wird jeweils eine Generalisierungsbeziehung (Instanz des UML-Elementtyps *Generalization*) hergestellt. Diese Generalisierungen werden in einer Instanz des UML-Elementtyps *GeneralizationSet* zusammengefasst, die als `complete` gekennzeichnet

wird. Dies ist möglich, da in UML die Bildung einer Vereinigung von Elementtypen $E_1 \ldots E_n$ semantisch äquivalent zu einem abstrakten Elementtyp und der Angabe von Untertypen ist.[174] Abbildung 7.28 zeigt ein Beispiel für die Transformation einer `ObjectUnionOf`.

DisjointUnion

Es handelt sich bei dem `DisjointUnion`-Axiom um eine syntaktische Abkürzung. Ein Axiom `DisjointUnion(`$A\ B_1\ \ldots\ B_n$`)` ist semantisch äquivalent zu den drei Axiomen

```
SubClassOf( A ObjectUnionOf( B1 ... Bn ))
SubClassOf( ObjectUnionOf( B1 ... Bn ) A )
DisjointClasses( B1 ... Bn )
```

In diesem speziellen Fall lassen sich die oben geschilderten Probleme mit `SubClassOf`-Axiomen sowie hinreichenden Bedingungen für eine Klassenzugehörigkeit jedoch umgehen und die Semantik des Ausdrucks kann transformiert werden.

Die in der Langschreibweise des `DisjointUnion`-Axioms auftretende `ObjectUnionOf(` $B_1\ \ldots\ B_n$ `)` wird wie oben beschrieben in $n+1$ UML-Klassen, n Generalisierungsbeziehungen und eine als *complete* markierte Instanz des UML-Elementtyps *GeneralizationSet* transformiert. Die Oberklasse ist in diesem Fall jedoch nicht abstrakt und bekommt einen Namen, der dem IRI der OWL-Klasse A entspricht. Außerdem wird die Instanz des UML-Elementtyps *GeneralizationSet* als *disjoint* markiert, um die Disjunktheit der Klassen $B_1 \ldots B_n$ abzubilden.

Abbildung 7.29 zeigt ein Beispiel für die Transformation einer `DisjointUnion`.

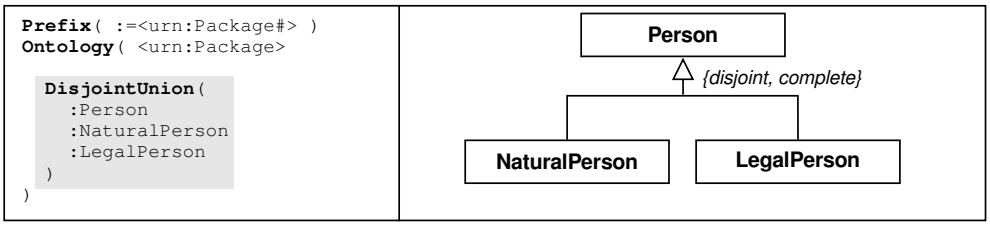


Abbildung 7.29. Beispiel für die Transformation einer Instanz des OWL-Elementtyps *DisjointUnion* in eine als überschneidungsfrei und vollständig gekennzeichnete Instanz des UML-Elementtyps *GeneralizationSet*.

7.6.6 Ausdeutung für XSD

Bei XSD existiert keine Möglichkeit, mit der sich einschränken ließe, dass es zu einem *ComplexTypeDefinition* keine weiteren Untertypen als bisher definierte geben darf. Die Vollständigkeit kann also nicht dargestellt werden. Aufgrund der fehlenden Möglichkeit, einem Element mehr als einen Typ zuzuweisen, ist die Modellierung der Überschneidungsfreiheit nicht erforderlich.

[174] Vgl. Olivé: Conceptual Modeling, S. 168.

7.7 Schnittmenge

Ein Elementtyp kann durch eine Schnittmenge anderer Elementtypen definiert werden. Ein Objekt ist genau dann Instanz dieses Elementtyps, wenn er auch Instanz einer Menge anderer Elementtypen ist.[175]

7.7.1 Logische Repräsentation

Formal lässt sich die Tatsache, dass ein Elementtyp E Schnittmenge einer Menge anderer Elementtypen $E_1 \dots E_n$ ist, folgendermaßen ausdrücken:

$$E(e) \leftrightarrow E_1(e) \wedge \dots \wedge E_n(e)$$

7.7.2 Repräsentation in UML

Für die Darstellung einer Schnittmenge von Elementtypen wäre ein dem *GeneralizationSet* entsprechendes Modellelement für Spezialisierungen notwendig. Ein solches gibt es bei UML jedoch nicht.

7.7.3 Repräsentation in OWL

In OWL steht eine CE zur Verfügung, um als Schnittmenge definierte Elementtypen zu definieren.

Konkrete Syntax

Mit der CE `ObjectIntersectionOf` wird ein Elementtyp definiert, dessen Instanzen genau diejenigen Objekte sind, die gleichzeitig Instanz eines jeden der Elementtypen $E_1 \dots E_n$ sind.

`ObjectIntersectionOf(` $E_1 \dots E_n$ `)`

Abstrakte Syntax

In abstrakter Syntax wird die Schnittmenge durch eine Instanz von *ObjectIntersectionOf* repräsentiert, die mit zwei oder mehr Instanzen von *ClassExpression* in Beziehung steht.

7.7.4 Transformation OWL → UML

Da UML kein dem *GeneralizationSet* vergleichbares Modellelement für Spezialisierungen enthält, lässt sich die Semantik dieses Elements nicht vollständig ausdrücken. Lassen sich die in der `ObjectIntersectionOf`-CE auftretenden Elementtypen $E_1 \dots E_n$ in UML-Klassen

[175] Vgl. Olivé: Conceptual Modeling, S. 169.

$C_1 \ldots C_n$ transformieren, so ist es lediglich möglich, eine abstrakte Klasse zu definieren, die Unterklasse der Klassen $C_1 \ldots C_n$ ist.

Aufgrund der im Abschnitt 6.7 diskutierten Probleme lässt es sich jedoch nicht erzwingen, dass alle Objekte, die Instanz aller Klassen $C_1 \ldots C_n$ sind, ebenfalls Instanz der abstrakten Unterklasse sein müssen.

7.7.5 Ausdeutung für XSD

XSD sieht keine Möglichkeit vor, einen Elementtyp als Schnittmenge anderer Typen zu definieren. Eine Instanz von *ComplexTypeDefinition* kann nur maximal einen Basistypen haben und diesen einschränken. Ein Schnitt zweier Elementtypen ist jedoch nicht möglich.

Kapitel 8

Datentypen

8.1 Allgemein

Bei einem Datentyp handelt es sich um eine spezielle Art von Elementtyp. In den vorherigen Abschnitten ging es in der Regel um Klassen als Elementtypen. Der Unterschied zwischen Klassen und Datentypen besteht darin, dass der Wert eines Datentyps keine Identität besitzt, gleiche Werte eines Datentyps also nicht unterscheidbar sind. Eine Ganzzahl mit dem Wert 42 bezeichnet also stets denselben Wert, unabhängig davon, für welches klassenabhängige Attribut sie verwendet wird. Bei einem Attribut, dessen Typen Klassen sind, sind die Werte Zeiger auf Instanzen der Klasse. Sie haben eine eigene Identität und sind somit unterscheidbar. Eine Instanz der Klasse "Firma" ist verschieden von einer anderen Instanz einer "Firma". Es ist jedoch auch möglich, dass die Werte verschiedener klassenabhängiger Attribute auf dieselbe Instanz zeigen.

8.1.1 Logische Repräsentation

Ein Datentyp besteht aus drei Komponenten: dem Wertebereich (engl. value space), dem lexikalischen Bereich (engl. lexical space) sowie einer wohldefinierten Abbildung von lexikalischem Bereich in den Wertebereich.[176] Der Wertebereich ist die – unter Umständen unendliche – Menge der zu beschreibenden Werte. Der lexikalische Bereich beschreibt, wie Werte des Datentyps *syntaktisch* aussehen müssen. Mit Hilfe der wohldefinierten Abbildung werden syntaktisch zulässige Werte auf Elemente des Wertebereichs abgebildet. Dabei ist es möglich, dass viele (sogar unendlich viele) syntaktisch verschiedene Werte auf dasselbe Element des Wertebereichs abgebildet werden.

Ein Beispiel: Sei der Wertebereich die Menge der rationalen Zahlen, die sich mit endlich vielen Nachkommastellen darstellen lassen. Der lexikalische Bereich umfasse die Zeichenketten, die dem regulären Ausdruck `[+-]?[0-9]+\.[0-9]+` genügen. Als Abbildung wird die natürliche Interpretation von Dezimalzahlen als rationale Zahlen verwendet. Dann bezeichnen die Zeichenketten "2.7", "+2.7" sowie "2.70" (und viele weitere) dieselbe Zahl 2,7.

[176] Vgl. Hitzler, P. et al.: Semantic Web, Berlin/Heidelberg 2008, S. 51f.

8.1.2 Repräsentation in UML

Neben einer Menge von im Standard vordefinierten (primitiven) Datentypen[177] erlaubt UML die Definition weiterer Datentypen in Klassendiagrammen. Dabei kann es sich um primitive Datentypen, zusammengesetzte Datentypen sowie Aufzählungen handeln. Details zu den drei Arten werden in den nachfolgenden Abschnitten beschrieben.

Im Unterschied zu Instanzen von UML-Klassen, die von sich aus verschieden sind, werden alle Instanzen eines Datentyps, die denselben Wert haben, als gleich angesehen: "*All copies of an instance of a data type and any instances of that data type with the same value are considered to be equal instances.*"[178]

Konkrete Syntax

Datentypen werden ähnlich wie Klassen mit einem Rechteck-Symbol dargestellt, das mit dem Schlüsselwort «datatype» (zusammengesetzte Datentypen), «primitive» (primitive Datentypen) oder «enumeration» (Aufzählungen) versehen wird. Der Name des Datentyps befindet sich im oberen Abschnitt. Abbildung 8.1 zeigt Beispiele für alle drei Fälle.

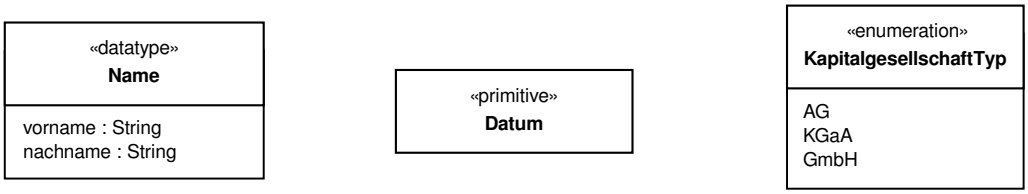


Abbildung 8.1. Beispiele für Datentypen in UML. Links: benutzerdefinierter Datentyp mit zwei Bestandteilen. Mitte: benutzerdefinierter primitiver Datentyp. Rechts: Aufzählung mit drei erlaubten Werten.

Abstrakte Syntax

Ein UML-Datentyp ist eine Instanz des Elementtyps *DataType*. Bei *DataType* handelt es sich um einen Untertyp von *PackageableElement*, dessen Instanzen Bestandteil eines Pakets sein können. Daher können Datentypen direkt auf oberster Ebene einem Paket zugeordnet werden. Abbildung 8.2 zeigt den zugehörigen Abschnitt aus dem UML-Metamodell mit allen relevanten *IsA*-Beziehungen von *DataType* bis *PackageableElement*.

[177] Vgl. OMG: UML Infrastructure, S. 171 ff.
[178] OMG: UML Superstructure, S. 63.

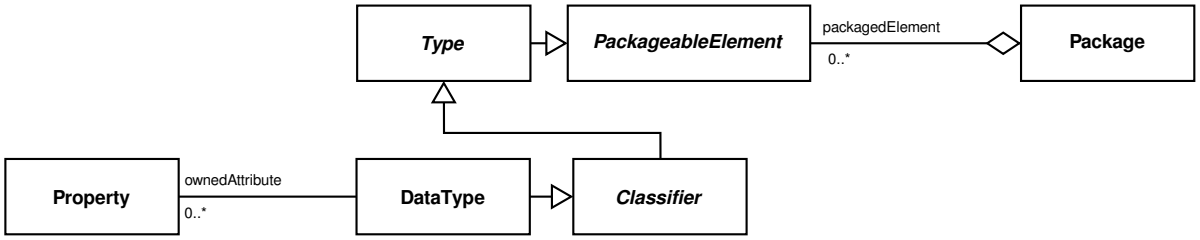


Abbildung 8.2. Ausschnitt aus dem UML-Metamodell, der den Elementtyp *DataType* sowie die Verbindung mit dem *Package* zeigt.

Obwohl die grafische Repräsentation eines Datentyps im Allgemeinen (Instanz von *DatyType*) sowie eines primitiven Datentyps (Instanz von *PrimitiveType*) und einer Aufzählung (Instanz von *Enumeration*) im Speziellen der Darstellung einer Klasse (Instanz von *Class*) ähnelt, die mit dem Stereotyp «datatype», «primitve» bzw. «enumeration» gekennzeichnet ist, so handelt es sich doch um verschiedene Elemente des Metamodells, wie in Abbildung 8.3 dargestellt.

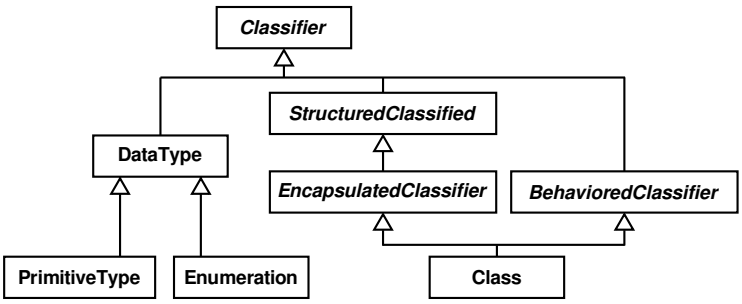


Abbildung 8.3. Auszug aus dem UML-Metamodell, das den Unterschied zwischen Klassen und Datentypen zeigt.

8.1.3 Repräsentation in OWL

In OWL werden drei Arten von Datentypen unterschieden:

- *rdfs:Literal* als Grunddatentyp
- die Datentypen der *OWL 2 datatype map*, die im Wesentlichen eine Teilmenge der Datentypen der XSD-Spezifikation ist
- Datentypen, die innerhalb der Ontologie mit Hilfe von `DatatypeDefinition` definiert wurden

Der Wertebereich des Grunddatentyps *rdfs:Literal* ist dabei als Vereinigung der Wertebereiche aller anderen Datentypen definiert. Eine weitere Menge vordefinierter Datentypen bildet die *OWL 2 datatype map*. Dabei handelt es sich um eine Teilmenge der Datentypen, die in

der XSD-Spezifikation definiert sind.[179] Wertebereich, lexikalischer Bereich sowie die bei der Definition eigener Datentypen erlaubten Einschränkungen des Wertebereichs werden aus der XSD-Spezifikation übernommen. Zusätzlich definiert die *OWL 2 datatype map* die Datentypen `owl:real`, `owl:rational` sowie `rdf:XMLLiteral`.

Mit Hilfe der Datentypen lassen sich Mengen von Werten (Instanzen von Datentypen), so genannte *Data Ranges* definieren. Dafür stehen die üblichen mengentheoretischen Operationen zur Verfügung. Mit Hilfe von `DataOneOf` kann eine Menge beschrieben werden, die genau aus den aufgeführten Werten besteht. Weiterhin ist es mit Hilfe von `DatatypeRestriction` möglich, eine Menge von Werten zu definieren, indem der Wertebereich eines Datentyps mit Hilfe von *constraining facets* eingeschränkt wird. Welche Beschränkungen erlaubt sind, ist in der *OWL 2 datatype map* definiert. Für einen Zahl-Datentyp sind z.B. folgende Beschränkungen möglich: kleiner gleich, größer gleich, kleiner sowie größer.[180]

Hinsichtlich der in einer Ontologie selbst definierten Datentypen ist jedoch eine wichtige Einschränkungen zu beachten: Sie können nicht in der A-Box einer Ontologie verwendet werden: "*The lexical form of each literal occurring in an OWL 2 DL ontology must belong to the lexical space of the literal's datatype.*"[181] Die Definition eigener Datentypen bezieht sich jedoch auf den Wertebereich; der lexikalische Bereich wird ausschließlich in der *OWL 2 datatype map* festgelegt.

Folgendes einfaches Beispiel zeigt eine Situation, bei der ohne diese Einschränkung eine Entscheidung über die Konsistenz einer Ontologie nicht eindeutig wäre:

```
1 Declaration( Datatype( :unklarerDatentyp ) )
2 DatatypeDefinition( :unklarerDatentyp
3     DataOneOf( "30"^^xsd:integer "30"^^xsd:string  )
4 )
5 DataPropertyRange( :alter xsd:integer )
6
7 DataPropertyAssertion( :alter :TimmiTester "30"^^:unklarerDatentyp )
```

Die `"30"^^:unklarerDatentyp` könnte für `"30"^^xsd:integer` stehen, in diesem Fall wäre die Ontologie konsistent, da als Zielbereich `xsd:integer` angegeben ist. Genauso wäre es jedoch auch möglich, dass sie für `"30"^^xsd:string` steht – dann läge eine inkonsistente Ontologie vor. Um diese Art von Nicht-Eindeutigkeit zu vermeiden, ist die Einschränkung bezüglich selbst definierter Datentypen notwendig.

[179] Im Detail sind dies: anyURI, base64Binary, boolean, byte, dateTime, dateTimeStamp, decimal, double, float, hexBinary, int, integer, language, long, Name, NCName, negativeInteger, NMTOKEN, nonNegativeInteger, nonPositiveInteger, normalizedString, positiveInteger, short, string, token, unsignedByte, unsignedInt, unsignedLong und unsignedShort.

[180] Die zugehörigen *normative constraining facets* sind: xsd:minInclusive, xsd:maxInclusive, xsd:minExclusive, xsd:maxExclusive – vgl. W3C: OWL 2 Structural specification, Abschnitt 4.1.

[181] A. a. O., Abschnitt 5.7.

Konkrete Syntax

Ein Datentyp wird definiert, indem einer *DataRange* mit Hilfe eines `DatatypeDefinition`-Axioms ein IRI zugewiesen wird. Gemäß der OWL 2 DL Richtlinie muss dieser IRI als Name für einen Datentyp deklariert werden.

```
Declaration( Datatype( :ISBN13 ) )
DatatypeDefinition(
      :ISBN13
      DatatypeRestriction(
           xsd:string xsd:pattern "[0-9]3-[0-9]-[0-9]5-[0-9]3-[0-9]"
      )
)
```

Abstrakte Syntax

In der abstrakten Syntax fällt auf, dass ein *Datatype* nur mittelbar (über eine *DatatypeDefinition*-Instanz) mit seinem Wertebereich (einer Instanz einer Unterklasse von *DataRange*) verbunden ist. Es ist daher auch möglich, einen Datentypen zu verwenden, dem kein Wertebereich zugeordnet ist – definitionsgemäß hat dieser dann den Wertebereich von *rdfs:Literal*. Da die zur Definition von Wertemengen (und damit Datentypen) verwendeten Unterklassen von *DataRange* wiederum Referenzen auf *DataRange* haben und auch *Datatype* eine Unterklasse von *DataRange* ist, sind beliebig verschachtelte Konstruktionen von datentypdefinierenden Elementen möglich.

Abbildung 8.4 zeigt den zugehörigen Abschnitt aus dem OWL-Metamodell, Abbildung 8.5 ein Objektdiagramm des zum obigen Beispiel gehörenden Modells.

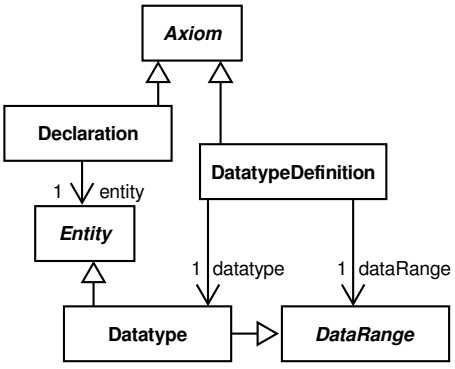


Abbildung 8.4. Der die Datentypen betreffende Auszug aus dem OWL-Metamodell.

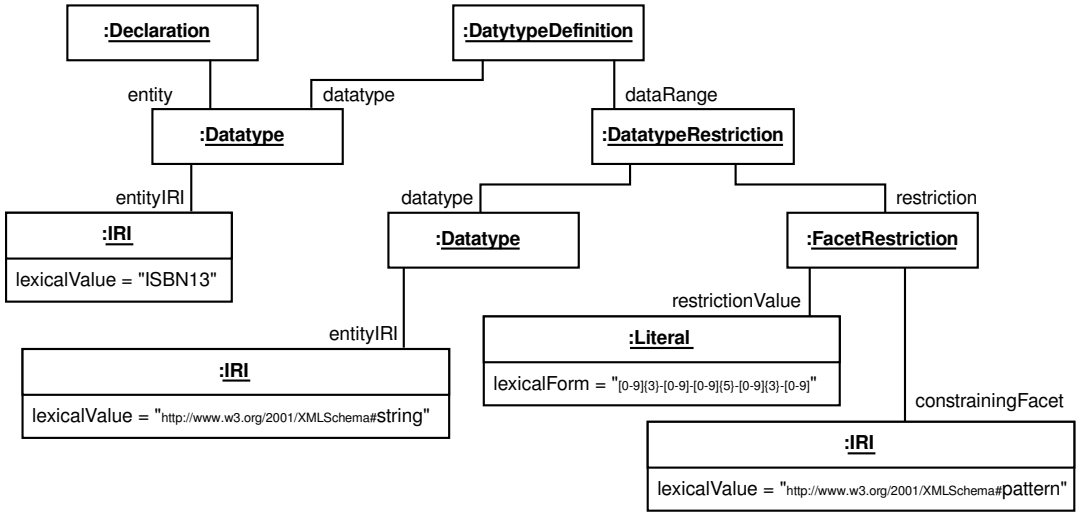


Abbildung 8.5. Objektdiagramm, das die abstrakte Syntax des obigen Beispiels für die Definition des Datentyps "ISBN13" zeigt.

8.2 Primitive Datentypen

Primitive Datentypen haben keine innere Struktur. Als Beispiele sind Zeichenketten, Wahrheitswerte oder Zahlen zu nennen.

8.2.1 Repräsentation in UML

Primitive Datentypen werden in UML insofern speziell behandelt, als der UML-Standard bereits einige primitive Datentypen bereit stellt: Wahrheitswerte (Boolean), Ganzzahlen (Integer), natürliche Zahlen plus "unbegrenzt" (UnlimitedNatural) und Zeichenketten (String). Primitive Datentypen in UML besitzen keine Identität: "*Instances of primitive types do not have identity. If two instances have the same representation, then they are indistinguishable.*"[182]

Konkrete Syntax

Primitive Datentypen werden in UML üblicherweise nicht grafisch dargestellt. Sollte es notwendig sein, würden sie sie als Rechteck mit dem Schlüsselwort «primitive» gezeichnet werden. Abbildung 8.1 (Mitte) zeigt ein Beispiel dafür.

Abstrakte Syntax

In abstrakter Syntax finden sich primitive Datentypen als Instanzen von *PrimitiveType* wieder.

[182]OMG: UML Superstructure, S. 142.

8.2.2 Repräsentation in OWL

Bei OWL bestehen Datentypen aus genau einem Literal,[183] besitzen also keine weitere Struktur und sind damit immer primitive Datentypen. Die konkrete und abstrakte Syntax von Datentypen wurde bereits im vorherigen allgemeinen Abschnitt über Datentypen vorgestellt.

8.2.3 Transformation UML → OWL

Für die Transformation von primitiven Datentypen müssen drei Fälle unterschieden werden:

(a) Der Datentyp ist einer der vier primitiven UML-eigenen Datentypen "Boolean", "Integer", "String" und "UnlimitedNatural".

(b) Der Datentyp ist ein primitiver Datentyp, der aus der Menge der XSD-Datentypen stammt, die auch bei OWL verwendet werden.

(c) Die Definition des (benutzerdefinierten) Datentyps ist im UML-Modell enthalten.

UML-eigener Datentyp

OWL verwendet die Datentyp-Definitionen aus der XSD-Spezifikation. Daher wird im Fall (a) ein Datentyp in seinen entsprechenden Datentyp aus XML Schema transformiert. Primitive UML-eigene Datentypen lassen sich daran erkennen, dass sie in einem Paket namens "UMLPrimitiveTypes" enthalten sind. Der Zusammenhang von UML-eigenen Datentypen und XSD-Datentypen wird in Tabelle 8.1 gezeigt.

Tabelle 8.1. Abbildung der primitiven UML-eigenen Datentypen auf XSD-Datentypen.

UML-Datentyp	XSD-Datentyp
Boolean	http://www.w3.org/2001/XMLSchema#boolean
Integer	http://www.w3.org/2001/XMLSchema#integer
String	http://www.w3.org/2001/XMLSchema#string
UnlimitedNatural	http://www.w3.org/2001/XMLSchema#nonNegativeInteger

Für eine präzise Wiedergabe des UML-Datentyps "UnlimitedNatural", einschließlich des Sternchen-Zeichens für "unbegrenzt", müsste ein Datentyp definiert werden, der eine Vereinigung des XSD-Datentyps "nonNegativeInteger" und dem Zeichen "*" ist:

```
Declaration( Datatype( :UnlimitedNatural ) )
DatatypeDefinition( :UnlimitedNatural
      DataUnionOf(
           xsd:nonNegativeInteger
           DataOneOf( "*")
      ) )
```

[183] Vgl. W3C: OWL 2 Structural specification, Abschnitt 7.

Aufgrund der oben beschriebenen Einschränkungen hinsichtlich der Verwendbarkeit benutzerdefinierter Datentypen in OWL und der Tatsache, dass "UnlimitedNatural" in der Praxis – außer bei der Definition von UML-Modellen – so gut wie nicht vorkommt, scheint es praktikabler, an dieser Stelle "nonNegativeInteger" zu verwenden.

XSD-Datentyp

Im Fall (b) ist die Transformation noch offensichtlicher, da hier direkt ein Datentyp referenziert wird, der auch in OWL zur Verfügung steht. Der Paketname, in dem die primitiven Datentypen enthalten sind, ist von der verwendeten Typbibliothek abhängig. Gängig ist eine Bezeichnung wie "XMLPrimitiveTypes", die auch im Folgenden verwendet wird. Bei Einsatz einer anderen Bibliothek wäre lediglich dieser Name anzupassen, am grundlegenden Vorgehen ändert sich nichts. Liegt ein primitiver Datentyp aus dem Paket "XMLPrimitiveTypes" vor, so wird sein Name verwendet, um durch Ergänzen des XSD-Namensraums[184] die Referenz auf den XSD-Datentyp in der Ontologie zu bilden.

Benutzerdefinierter Datentyp

Für benutzerdefinierte Datentypen des Falls (c) wird in der Ontologie mit Hilfe des `Datatype`-Axioms einer neuer Datentyp definiert. Da in OWL Datentypen – wie alle OWL-Modellelemente – über eindeutige IRIs identifiziert werden, muss während der Transformation ein entsprechender Bezeichner definiert werden. Da in der UML Elemente (also auch Datentypen) eindeutig über ihren Namen und ihr umschließendes Paket identifiziert werden, wird der Bezeichner aus einer Kombination von Paket- und Datentypnamen gebildet.

8.2.4 Transformation OWL → UML

Primitive Datentypen stellen bei der Transformation eine Schwierigkeit dar, da OWL sehr vielfältige Möglichkeiten zur Definition neuer Datentypen bietet. Einige – und vermutlich die am meisten verwendeten – lassen sich jedoch transformieren. Die primitiven Datentypen von OWL stammen aus der XSD-Spezifikation; für diese Datentypen gibt es gängige UML-Bibliotheken. Daher reicht es zum Transformieren aus, eine solche Bibliothek in den Transformationsprozess einzubinden und für einen OWL-Datentyp anhand seines IRI die entsprechende Instanz des UML-Elementtyps *PrimitiveType* herauszusuchen.

8.2.5 Ausdeutung für XSD

XSD bietet reichhaltige Unterstützung für primitive Datentypen, die dort `simpleType` genannt werden. Neben einer Vielzahl von vordefinierten Datentypen besteht die Möglichkeit, sich eigene `simpleType` zu definieren. Dafür stehen drei Möglichkeiten zur Verfügung:

▷ Ausgehend von einem Basistypen werden Einschränkungen festgelegt (`restriction`).

[184] `http://www.w3.org/2001/XMLSchema`

- Es wird ein Datentyp angegeben, der – durch Leerzeichen getrennt – beliebig oft wiederholt werden darf. Ein solcher Datentyp ist semantisch äquivalent zu der Angabe, dass der Beziehungstyp, für den er verwendet wird, beliebig oft auftreten darf (`list`).
- Es wird eine Vereinigung zweier anderer Datentypen gebildet (`union`).

In diesem Beispiel wird, ausgehend vom Datentyp `xsd:string`, ein regulärer Ausdruck zum Einschränken des lexikalischen Bereichs angegeben:

```
1 <simpleType name="ISBN13">
2    <restriction base="xsd:string">
3       <pattern value="[0-9]{3}-[0-9]-[0-9]{5}-[0-9]{3}-[0-9]" />
4    </restriction>
5 </simpleType>
```

8.3 Zusammengesetzte Datentypen

Im Gegensatz zu primitiven Datentypen haben zusammengesetzte Datentypen eine innere Struktur. Beispiele für zusammengesetzte Datentypen sind:

- Name, der aus Vorname und Familienname besteht
- physikalische Messgröße, die aus Wert und Maßeinheit besteht
- Adresse, bestehend aus Straßenname, Hausnummer, Postleitzahl und Ort

8.3.1 Repräsentation in UML

In UML dürfen Datentypen – wie auch Klassen – abhängige Attribute (und auch Operationen, die hier jedoch nicht betrachtet werden) besitzen. Somit eignen sie sich zur Beschreibung von Strukturen.[185]

Konkrete Syntax

Zusammengesetzte Datentypen werden ähnlich wie Klassen mit einem Rechteck-Symbol dargestellt, das mit dem Schlüsselwort «datatype» versehen wird. Der Name des Datentyps befindet sich im oberen Abschnitt. Im darunter befindlichen Abschnitt werden die Attribute des Datentyps mit Namen und – in der Regel primitivem – Datentyp aufgeführt. Ein Beispiel eines Datentyps mit zwei Bestandteilen ist in Abbildung 8.1 (links) zu sehen.

Abstrakte Syntax

Bei einem zusammengesetzten Datentypen handelt es sich um eine direkte Instanz des Elementtyps *DataType*. Die einzelnen Bestandteile des Datentyps sind als Instanzen von *Property* in der Rolle *ownedAttribute* mit ihr verknüpft, wie in Abbildung 8.2 dargestellt ist.

[185] Vgl. Balzert: Softwaretechnik, S. 182.

8.3.2 Repräsentation in OWL

Bei OWL bestehen Datentypen aus genau einem Literal, besitzen also keine weitere Struktur.[186]

Als Erbe von RDF bestünde theoretisch die Möglichkeit, einen *blank node* (einen unbenannten Knoten) und die RDF-Anweisung `parseType="Resource"` zu verwenden, um zusammengesetzte Daten zu realisieren, wie in Listing 8.1 gezeigt wird. Allerdings wird `parseType="Resource"` weder bei OWL 1 noch bei OWL 2 erwähnt, sodass dies kein für OWL gültiges Konstrukt sein dürfte.

Auch wenn eine solche Schreibweise in OWL möglich wäre und einem solchen anonymen Individuum ein Elementtyp zugewiesen werden könnte, so ließe sich bei der Definition des Elementtyps nicht mehr erkennen, ob es sich um einen zusammengesetzten Datentypen oder einen "normalen" Elementtyp handeln soll.

```
 1 <rdf:RDF xml:base="http://example.com/persons/"
 2    xmlns="http://example.com/persons/"
 3    xmlns:owl="http://www.w3.org/2002/07/owl#"
 4    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 5
 6    <owl:Ontology rdf:about="http://example.com/persons/"/>
 7
 8    <owl:Class rdf:about="Person" />
 9
10    <owl:NamedIndividual rdf:about="Timmi">
11       <rdf:type rdf:resource="Person"/>
12       <hasName rdf:parseType="Resource">
13          <first>Timmi</first>
14          <last>Tester</last>
15       </hasName>
16    </owl:NamedIndividual>
17 </rdf:RDF>
```

Listing 8.1. Eine Ontologie in RDF/XML-Syntax mit einem strukturierten Wert als *blank node* (Zeilen 13–16).

8.3.3 Transformation UML → OWL

Die Transformation zusammengesetzter Datentypen, d.h. solcher mit abhängigen Attributen, erfolgt ähnlich wie die von Elementtypen. Datentypen in UML haben zwei Eigenschaften, auf die bei der Transformation Rücksicht genommen werden muss:

1. Die Werte besitzen keine Identität.

2. Jeden Wert gibt es nur einmal.

Da die Transformation analog zu der von Elementtypen erfolgt, handelt es sich im Modell folglich bei den Instanzen des aus einem Datentyp resultierenden Elementtyps auch um Individuen. Da in OWL jedes Individuum einen Namen besitzen muss, ergäbe sich dadurch bei Eigenschaft (1) eine gegenüber UML veränderte Semantik: Während in UML die Instanz

[186] Vgl. W3C: OWL 2 Structural specification, Abschnitt 7.

des Datentyps keine Identität besitzt, muss dem entsprechenden Individuum bei OWL ein IRI zugewiesen werden, mit dem es referenziert (und auch identifiziert) werden kann.

Die Eigenschaft (2), dass es jeden Wert nur einmal geben darf, lässt sich mit Hilfe von `HasKey`-Axiomen (→ 10 BESCHRÄNKUNGEN) gewährleisten. Für jeden UML-Datentyp D mit den abhängigen Attributen $a_1 \dots a_n$, der auf eine OWL-Klasse C mit den Data Properties $dp_1 \dots dp_n$ abgebildet wird, wird folgendes Axiom zur Ontologie hinzugefügt:

`HasKey(` C `()` `(` $dp_1 \dots dp_n$ `) )`

Dadurch wird sichergestellt, dass es sich bei jedem Auftreten eines Individuums mit denselben Werten von $dp_1 \dots dp_n$ um genau ein und dasselbe Individuum handelt. Abbildung 8.6 zeigt ein Beispiel für eine solche Transformation eines zusammengesetzten Datentyps.

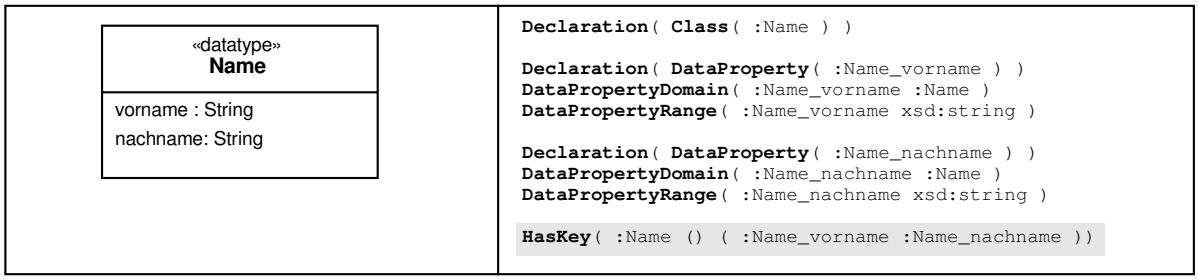


Abbildung 8.6. Beispiel für die Transformation eines zusammengesetzten Datentyps (d.h. Instanz des UML-Elementtyps *DataType*) mit abhängigen Attributen in Instanzen der OWL-Elementtypen *Class*, *DataProperty* und *HasKey*.

8.3.4 Ausdeutung für XSD

Die Definition zusammengesetzter Datentypen ist in XSD nur bedingt möglich. Ein `simpleType` kommt dafür nicht in Frage, da er (bis auf die Wiederholungen einer Liste) keine innere Struktur aufweist. Zusammengesetzte Datentypen lassen sich jedoch nachbilden, indem ein `complexType` E und eine Reihe von Beziehungstypen definiert wird, bei denen E als erster Teilnehmer auftritt und der zweite Teilnehmer ein Datentyp ist. Hier ist dies am Beispiel eines zusammengesetzten Datentypen namens “Name”, der die zwei Bestandteile “vorname” und “nachname” besitzt, gezeigt:

```
1 <complexType name="Name">
2    <sequence>
3       <element name="vorname" type="xsd:string" />
4       <element name="nachname" type="xsd:string" />
5    </sequence>
6 </complexType>
```

8.4 Aufzählungen

Bei Aufzählungen handelt es sich um eine spezielle Form eines primitiven Datentyps. Anders als bei einem allgemeinen primitiven Datentyp sind bei einer Aufzählung Wertebereich und lexikalischer Bereich gleich große, fest definierte, endliche Mengen. Die Abbildung von lexikalischem Bereich in den Wertebereich ist eine eins-zu-eins Abbildung.

Ein Beispiel für einen Aufzählungs-Datentyp sind die deutschen Namen der Wochentage, die aus sieben möglichen Werten bestehen.

8.4.1 Repräsentation in UML

In UML treten Aufzählungen als Instanzen von *Enumeration* auf und geben explizit eine Liste von erlaubten Werten vor.

Konkrete Syntax

Aufzählungen werden mit dem auch für Klassen verwendeten Rechteck-Symbol dargestellt, das mit dem Schlüsselwort «enumeration» versehen wird. Der Name der Aufzählung befindet sich im oberen Abschnitt. Im unteren Abschnitt werden die erlaubten Werte aufgeführt, einer pro Zeile. Ein Beispiel für eine Aufzählung mit drei erlaubten Werten zeigt Abbildung 8.1 (rechts).

Abstrakte Syntax

Da es sich bei *Enumeration* um einen *DataType* handelt, der wiederum Unterklasse von *NamedElement* ist, besitzt eine *Enumeration* einen Namen. Die für die Aufzählung erlaubten Werte sind Instanzen des Typs *EnumerationLiteral*, die mit der *Enumeration*-Instanz verknüpft sind. Zu beachten ist, dass es sich bei einem *EnumerationLiteral* ebenfalls um ein *NamedElement* handelt. Der Wert findet sich daher als Name des *EnumerationLiteral* wieder. Abbildung 8.7 zeigt den für *Enumeration* relevanten Abschnitt aus dem UML-Metamodell. Abbildung 8.8 zeigt das Objektdiagramm des zum obigen Beispiel (Abbildung 8.1 rechts) gehörenden Modells.

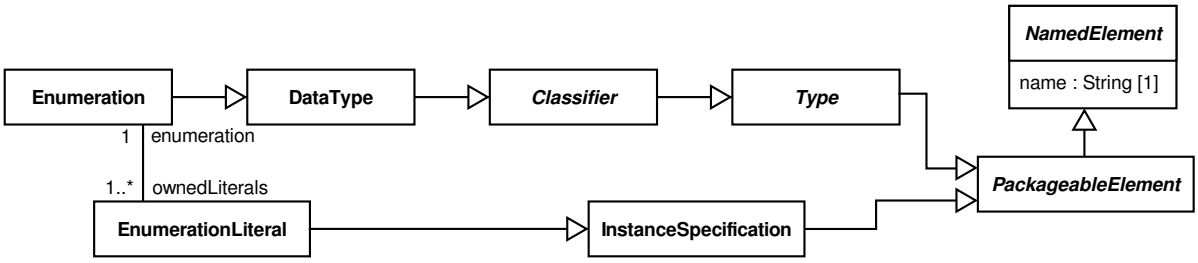


Abbildung 8.7. Der *Enumeration* betreffende Auszug aus dem UML-Metamodell.

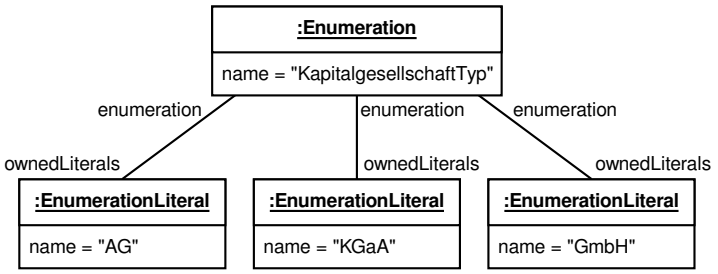


Abbildung 8.8. Objektdiagramm, das die abstrakte Syntax des obigen Beispiels (Abbildung 8.1 rechts) für die Definition des Aufzählung-Datentyps "KaptialgesellschaftTyp" zeigt.

8.4.2 Repräsentation in OWL

In OWL ist es möglich, eine Data Range zu definieren, die aus einer Menge von angegebenen Literalen besteht. Dabei müssen die Literale nicht demselben Datentyp angehören.

Konkrete Syntax

Die Definition der Data Range mit fester Literal-Menge erfolgt mit Hilfe des Schlüsselworts `DataOneOf` und einer Auflistung der erlaubten Literale.

```
DatatypeDeclaration( :KapitalgesellschaftTyp
     DataOneOf(
           "AG" ^^xsd:string
           "KGaA" ^^xsd:string
           "GmbH" ^^xsd:string
) )
```

Abstrakte Syntax

Ein Datentyp mit festgelegter Wertemenge wird in abstrakter Syntax mit einer Instanz von *DataOneOf* dargestellt. Diese steht in Beziehung zu einer oder mehreren Instanzen von *Literal*, deren *lexicalForm*-Attribut je einen Wert der Wertemenge enthält. Der Datentyp des Literals selbst wird durch eine Beziehung zu einer Instanz von *Datatype* angegeben. Damit die durch `DataOneOf` definierte Data Range Bestandteil der Ontologie wird, muss sie – wie jeder innerhalb der Ontologie definierte Datentyp – mit Hilfe einer Instanz von *DatatypeDefinition* mit einem Datentyp (Instanz von *Datatype*) verknüpft werden. Der Zusammenhang dieser Meta-Elementtypen wird in Abbildung 8.9 gezeigt. Abbildung 8.10 zeigt ein Objektdiagramm der abstrakten Syntax für das obige Beispiel in konkreter Syntax.

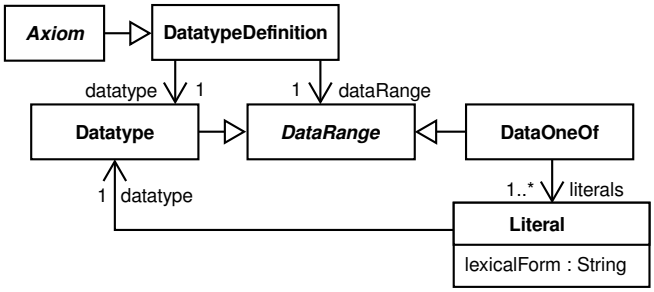


Abbildung 8.9. Der *DataOneOf* betreffende Auszug aus dem OWL-Metamodell.

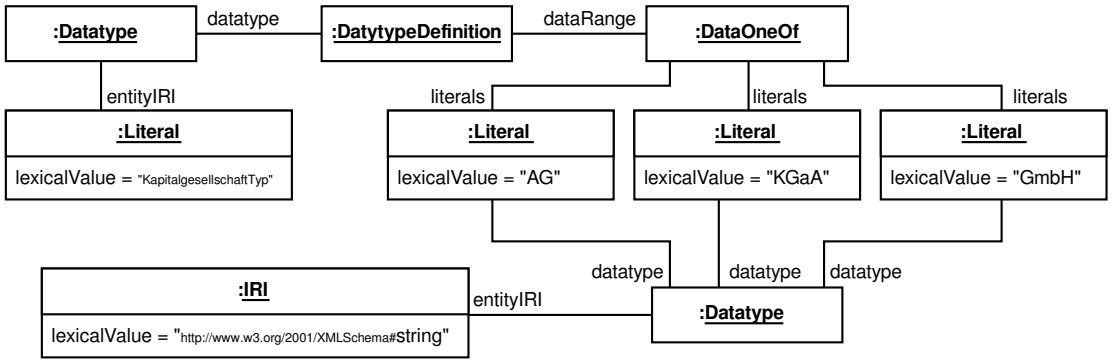


Abbildung 8.10. Objektdiagramm, das die abstrakte Syntax für das Beispiel im Abschnitt zur konkreten Syntax von `DataOneOf` zeigt.

8.4.3 Transformation UML → OWL

In der UML wird eine Instanz des Elementtyps *Enumeration* verwendet, um einen Datentypen mit einer vorgefertigten Liste erlaubter Werte zu definieren. Eine solche *Enumeration*-Instanz lässt sich in einen benannten OWL-Datentyp transformieren. Neben der Deklaration der Existenz des Datentyps (Instanz des OWL-Elementtyps *Declaration*) muss die Definition des Datentyps mit Hilfe einer Instanz des OWL-Elementtyps *DatatypeDefinition* angegeben werden. Die in der *Enumeration*-Instanz angegebenen erlaubten Werte werden innerhalb einer Instanz des OWL-Elementtyps *DataOneOf* (als Instanzen von *Literal*) aufgeführt. Abbildung 8.11 zeigt ein Beispiel für eine solche Transformation.

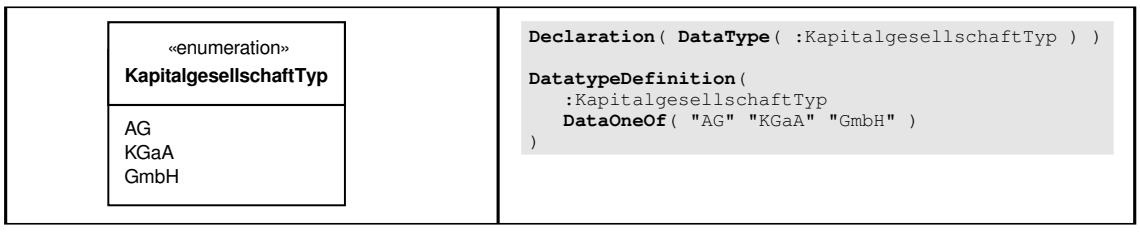


Abbildung 8.11. Beispiel für die Transformation einer Instanz des UML-Elementtyps *Enumeration*.

8.4.4 Transformation OWL → UML

Abbildung 8.12. Beispiel für die Transformation einer Datentypendefinition mit Hilfe des `DataOneOf`-Axioms in eine Instanz des UML-Elementtyps *Enumeration*.

Mit Hilfe des Axioms `DataOneOf` definierte Listen lassen sich als Instanzen des UML-Elementtyps *Enumeration* darstellen, wie in Abbildung 8.12 gezeigt. Taucht das `DataOneOf`-Axiom außerhalb einer namensgebenden `DatatypeDefintion` auf, wird ein Name (aus den enthaltenen Literalen) generiert, da in UML eine *Enumeration*-Instanz als *NamedElement* zwingend einen Namen benötigt.

Aus den lexikalischen Werten der einzelnen im `DataOneOf`-Axiom auftretenden Literale (Instanzen des OWL-Elementtyps *Literal*) werden die zur *Enumeration*-Instanz gehörenden *EnumerationLiteral*-Elemente erzeugt. Es ist in OWL möglich, den Literalen eines `DataOneOf`-Axioms einen oder auch verschiedene Datentypen zuzuweisen. Dies ist auch bei den *EnumerationLiteral*-Instanzen möglich, bei denen das *classifier*-Attribut mit dem entsprechende Datentyp belegt wird. Abbildung 8.13 zeigt die entsprechende QVT-Regel.

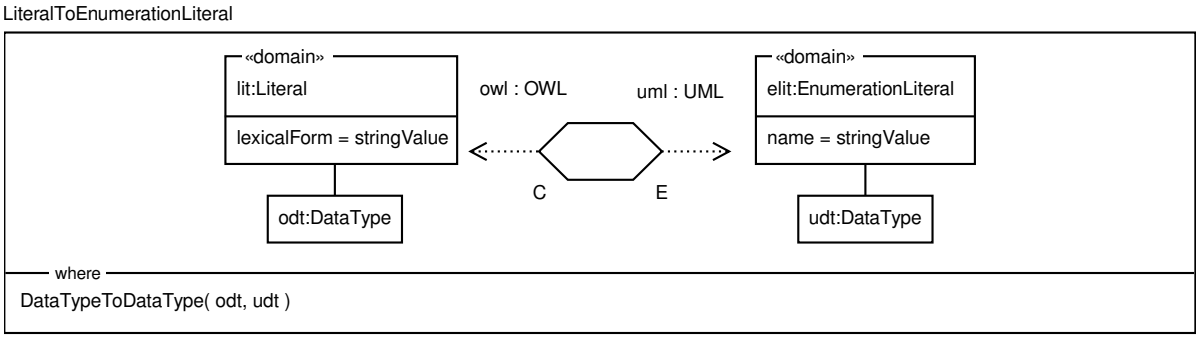


Abbildung 8.13. QVT-Regel für die Transformation von Literalen einer Instanz des OWL-Elementtyps *DataOneOf* in Instanzen des UML-Elementtyps *EnumerationLiteral*.

8.4.5 Ausdeutung für XSD

Indem der Basistyp eines `simpleType` auf eine feste Menge von Werten einschränkt wird, lässt sich ein Aufzählungstyp definiert:

```
<simpleType name="KapitalgesellschaftTyp">
   <restriction base="xsd:string">
      <enumeration value="AG" />
      <enumeration value="KGaA" />
      <enumeration value="GmbH" />
   </restriction>
</simpleType>
```

8.5 ISO 19100 Codelist

Bei einer *Codelist* handelt es sich um eine Erweiterung des UML-Elementtyps *Enumeration*, die im "UML-Profile" von ISO 19103 bzw. GML definiert ist.[187,188,189]

Zunächst ist in ISO 19103 von einer "enumeration", also einem Aufzählungsdatentyp die Rede: ("*«CodeList» is a flexible enumeration [...]*"[190] Etwas später wird im selben Absatz jedoch bereits die Darstellung in Form einer Klasse mit klassenabhängigen Attributen für die Wert-Schlüssel-Paare nahegelegt: *"[...] with an attribute name for each value and the code(key) represented as an initial value."* Auch in der GML-Spezifikation wird von Klassen gesprochen: "*A UML class with stereotype «CodeList»*".[191]

Einer solchen Klasse (sic!), die mit dem Stereotyp *Codelist* versehen ist, wird eine neue Semantik zugewiesen: Zusätzlich zu den festen Werten einer Aufzählung (wie bei einer UML-*Enumeration*) darf eine Codelist weitere Werte enthalten. Der GML-Standard schreibt für diese zusätzlichen Werte eine spezielle lexikalische Form vor.

8.5.1 Transformation UML → OWL

Ähnlich wie eine Instanz des UML-Elementtyps *Enumeration* wird auch eine Instanz des Elementtyps *Codelist* auf einen OWL-Datentyp abgebildet. Ein `DataUnionOf`-Axiom verbindet dabei zwei Wertebereiche (Instanzen von *DataRange*):

- die vordefinierten Werte einer gewöhnlichen Instanz des UML-Elementtyps *Enumeration* in Form einer Instanz des OWL-Elementtyps *DataOneOf*
- die zusätzlichen Werte der *Codelist*-Instanz, spezifiziert durch eine Instanz des OWL-Elementtyps *DataTypeRestriction*.

[187] Vgl. ISO: Norm 19103, S. 23.
[188] Vgl. OGC: GML Standard 3.2.1, S. 15.
[189] Vgl. a. a. O., S. 337.
[190] ISO: Norm 19103, S. 23.
[191] OGC: GML Standard 3.2.1, S. 347.

Die Instanz des OWL-Elementtyps *DataTypeRestriction* ermöglicht einen eleganten Weg, den Datentyp mit den zusätzlich erlaubten Werten zu versehen – OWL verwendet hier dieselbe Syntax wie XML Schema, die in der GML-Spezifikation verwendet wird.

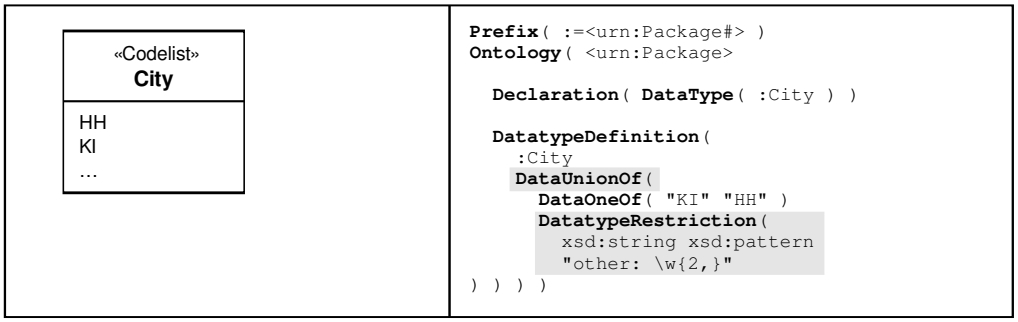


Abbildung 8.14. Beispiel für die Transformation einer GML-*Codelist*.

8.5.2 Ausdeutung für XSD

Die GML-Spezifikation gibt vor, wie eine Codelist in XSD umzusetzen ist.[192] Es werden drei Instanzen von *SimpleTypeDefinition* erzeugt: Eine umfasst die explizit aufgezählten Werte der Codelist, eine zweite deckt zusätzliche Werte (gemäß der von GML vorgegeben Syntax) ab, eine dritte verbindet die beiden genannten Typen in einer `union`. Die Codelist aus Abbildung 8.14 sieht in XML-Schema folgendermaßen aus:

```
<simpleType name="City">
   <union memberTypes="tns:CityEnumerationType tns:CityOtherType"/>
</simpleType>

<simpleType name="CityEnumerationType">
   <restriction base="string">
      <enumeration value="HH"/>
      <enumeration value="KI"/>
   </restriction>
</simpleType>

<simpleType name="CityOtherType">
   <restriction base="string">
      <pattern value="other: \w{2,}"/>
   </restriction>
</simpleType>
```

[192] Vgl. OGC: GML Standard 3.2.1, S. 347.

8.6 Generalisierung

Bei Datentypen handelt es sich letztlich nur um eine spezielle Art von Elementtypen, sodass auch bei ihnen die Möglichkeit besteht, Generalisierungsbeziehungen (→ 7.6 Generalisierung) zu definieren.

8.6.1 Repräsentation in UML

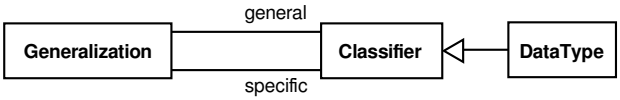


Abbildung 8.15. Der *Generalization* und *DataType* betreffende Auszug aus dem UML-Metamodell.

Da eine *Generalization* für *Classifier* und damit auch für *DataType* definiert ist, lassen sich auch Generalisierungen von Datentypen darstellen. Abbildung 8.15 zeigt, wie *DataType* in Verbindung mit *Generalization* genutzt werden kann.

8.6.2 Repräsentation in OWL

OWL bietet keine allgemeine Möglichkeit, Vererbungen bzw. Generalisierungen von Datentypen zu definieren.

8.6.3 Transformation UML → OWL

Da OWL keine Möglichkeit für eine Vererbung/Generalisierung von Datentypen vorsieht, ist eine Transformation im Allgemeinen nicht möglich. Im Spezialfall einer vollständigen Generalisierung von primitiven Datentypen (z.B. Aufzählungen) ist eine Transformation jedoch möglich.

Handelt es sich bei den Bestandteilen einer als vollständig markierten Instanz des UML-Elementtyps *GeneralizationSet* nicht um Klassen (Instanzen des UML-Elementtyps *Class*), sondern um Datentypen (Instanzen des UML-Elementtyps *DataType*), so kann als Ziel nicht wie im allgemeinen Abschnitt zu Generalisierung (→ 7.6 Generalisierung) eine `ObjectUnionOf`-CE verwendet werden, da diese – wie es der Name schon sagt – nur für Klassen nutzbar ist. Stattdessen kommt bei einer vollständigen Generalisierung von Datentypen eine Instanz des OWL-Elementtyps *DataUnionOf* zum Einsatz. Die in der *DataUnionOf*-Instanz zusammengefassten Unter-Datentypen bilden somit eine neue Data Range. Mit Hilfe einer Instanz des OWL-Elementtyps *DatatypeDefinition* wird dieser Menge von Datentypen ein Name zugewiesen. Dies ist der Name des Ober-Datentyps aus UML. Abbildung 8.16 zeigt die zugehörige Transformationsregel. Ein Beispiel für eine solche Transformation ist in Abbildung 8.17 dargestellt.

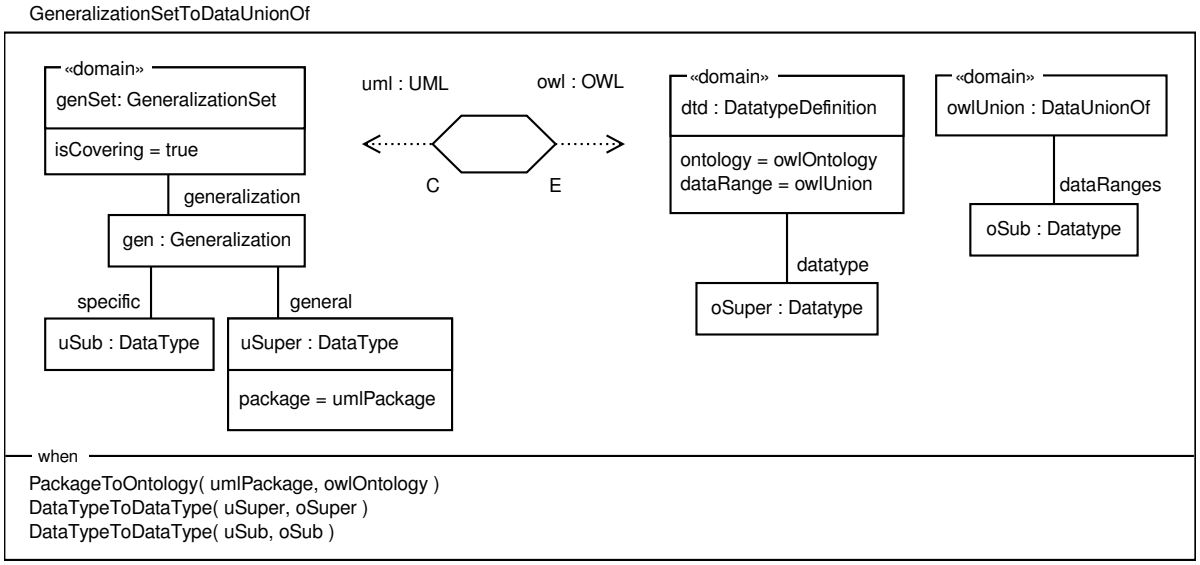


Abbildung 8.16. QVT-Regel für die Transformation einer vollständigen Generalisierung von Datentypen.

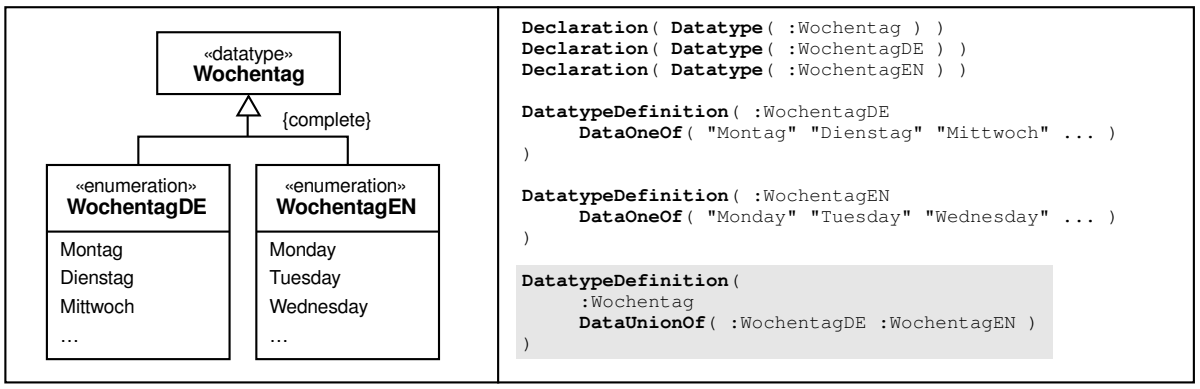


Abbildung 8.17. Beispiel für die Transformation einer Generalisierungsbeziehung zwischen Datentypen.

Kapitel 9

Beziehungstypen

9.1 Allgemein

In einem Softwaresystem gibt es typischerweise eine Vielzahl von Beziehungen zwischen Instanzen der Elementtypen. Die Eigenschaften dieser Beziehungen werden mit Hilfe von *Beziehungstypen* (=engl. relationship types) beschrieben. Statt des Begriffs *Beziehungstyp* wird oftmals der Begriff *Beziehung* verwendet. Das ist nicht ganz korrekt, da mit *Beziehung* eine konkrete Instanz eines Beziehungstyps zwischen Instanzen von Elementtypen gemeint ist. Ein Beziehungstyp beschreibt allgemein, welche Eigenschaften für alle zugehörigen Beziehungen gelten.

Zu einer Beziehung gehören *Teilnehmer* (=beteiligte Instanzen eines Elementtyps), die eine bestimmte *Rolle* spielen. Übertragen auf Beziehungstypen sind die Teilnehmer Elementtypen, die eine Rolle in dem Beziehungstyp spielen. Es ist möglich, auf die Angabe der Rollen innerhalb der Beziehung bzw. des Beziehungstyps zu verzichten.

Verbindet ein Beziehungstyp ausschließlich Instanzen eines einzigen Elementtyps miteinander, so wird von einem *rekursiven* Beziehungstyp gesprochen.

Hat ein Beziehungstyp zwei Teilnehmer, so wird dies als *binärer* Beziehungstyp bezeichnet. Zugehörige Beziehungen heißen entsprechend binäre Beziehungen. Da beliebige n-stellige Beziehungen in binäre Beziehungen transformiert werden können,[193,194] **werden im Folgenden nur binäre Beziehungstypen betrachtet.**

9.1.1 Logische Repräsentation

Beziehungstypen können in der Form

$$R(p_1 : E_1, p_2 : E_2)$$

dargestellt werden. Dabei ist R ein Beziehungstyp, p_1 und p_2 sind Rollen, E_1 und E_2 sind Elementtypen.

Wird auf die Bezeichnung der Rollen verzichtet, so lässt sich ein Beziehungstyp folgendermaßen darstellen:

$$R(E_1, E_2)$$

[193] Vgl. Olivé: Conceptual Modeling, Kapitel 6.
[194] Vgl. Hesse/Mayr: Modellierung, S. 388.

9.1.2 Repräsentation in UML

Im UML können binäre Beziehungstypen auf zwei Arten dargestellt werden, als Assoziation oder als klassenabhängiges Attribut.

Assoziationen werden als Linien zwischen zwei Elementtypen dargestellt. Der Name des Beziehungstyps kann mit einem Pfeil versehen an die Linie geschrieben werden. Die Rollennamen werden in die Nähe ihrer zugehörigen Elementtypen geschrieben. Eine Besonderheit bei UML ist es, dass es möglich sein muss, alleine aufgrund des Rollennamens eindeutig von einem Elementtyp zum anderen zu navigieren.[195]

Attribute werden im mittleren Bereich eines Elementtyps aufgeführt, wobei lediglich die Angaben p_2 und E_2 notiert werden. E_1 ist aufgrund der Positionierung innerhalb des Symbols für den Elementtyp klar. Einen separaten Namen des Beziehungstyps und einen Rollennamen p_1 gibt es nicht, der zweite Teilnehmer des Beziehungstyps wird als Eigenschaft des ersten Elementtyps gesehen.

Konkrete Syntax

Im folgenden Beispiel sind die beiden Beziehungstypen als Assoziation (“arbeitet bei”) bzw. Attribut (“name”) dargestellt. Die grafische Syntax ist in Abbildung 9.1 zu sehen.

$$ArbeitetBei(Arbeitnehmer : Person, Arbeitgeber : Firma)$$
$$Name(Person, String)$$

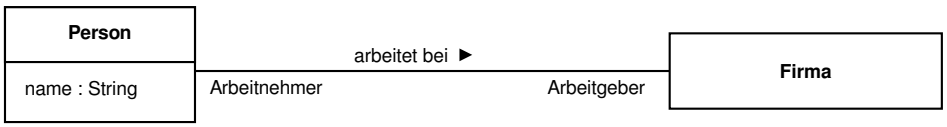


Abbildung 9.1. Beispiel für die grafische Syntax einer Assoziation (“arbeitet bei”) und eines klassenabhängigen Attributs (“name”).

Abstrakte Syntax

Unterscheidet sich die konkrete graphische Syntax der beiden Varianten Assoziation und Attribut deutlich voneinander, so werden beide in der abstrakten Syntax durch eine *Property*-Instanz repräsentiert. Der Ausschnitt aus dem UML-Metamodell, der den Zusammenhang dieser Elemente zeigt, ist in Abbildung 9.2 dargestellt.

Im Falle eines klassenabhängigen Attributs steht eine *Property*-Instanz in der Rolle *ownedAttribute* in Beziehung mit einem Elementtyp (Instanz von *StructuredClassifier*). Handelt es sich bei diesem Elementtyp um eine Instanz von *Class*, so existiert eine Beziehung zwischen der *Property*-Instanz und der *Class*-Instanz, die in der Rolle *class* auftritt.

[195] Vgl. Olivé: Conceptual Modeling, S. 71.

Handelt es sich um eine Assoziation, so steht eine *Association*-Instanz in Beziehung zu (mindestens)[196] zwei Instanzen von *Property*, die in der Rolle *membersEnd* auftreten. In diesem Fall existiert ebenfalls eine Beziehung zwischen jeder der beiden *Property*-Instanzen und der *Association*-Instanz, die in der Rolle *association* auftritt.

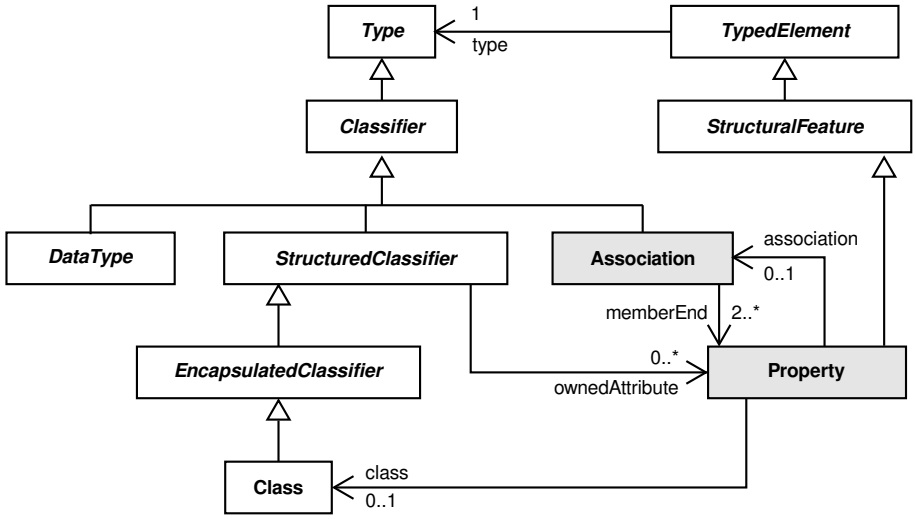


Abbildung 9.2. Auszug aus dem UML-Metamodell, das die beiden Möglichkeiten zeigt, wie Elementtypen verbunden werden können.

Abbildung 9.3 zeigt ein Objektdiagramm, das die abstrakte Syntax des zur konkreten UML-Syntax gezeigten Beispiels darstellt und die Zusammenhänge beispielhaft deutlich macht.

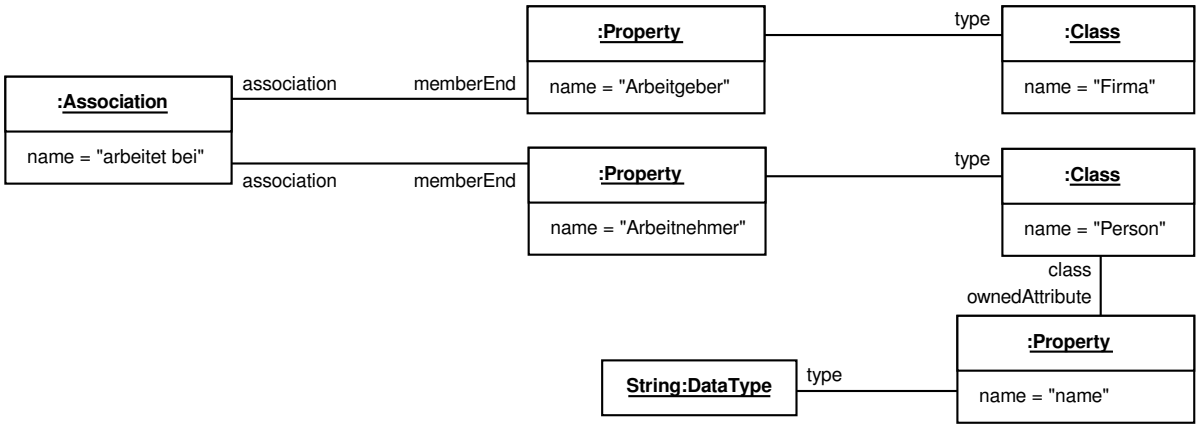


Abbildung 9.3. Objektdiagramm, das die abstrakte Syntax des zur konkreten UML-Syntax gezeigten Beispiels darstellt.

[196] Wie oben angemerkt, werden in dieser Arbeit nur binäre Beziehungstypen betrachtet.

9.1.3 Repräsentation in OWL

OWL kennt zwei verschiedene Konstrukte zum Verbinden von Elementen:

- Object Properties – für Beziehungen zwischen zwei Instanzen von Klassen
- DataProperties – für Beziehungen zwischen einer Instanz einer Klasse und einer Instanz eines Datentyps (→ 8 Datentypen).

Muss im Folgenden nicht zwischen Object Properties und DataProperties unterschieden werden, so wird einfach von *Property* die Rede sein.

Der Name des Beziehungstyps R wird in OWL als Name der Property verwendet. Die Rollen p_1 und p_2 lassen sich bei einer Property nicht angeben. Bei der Deklaration einer Property, d.h. der Angabe, dass eine Property mit diesem Namen existiert, wird zunächst keine nähere Aussage über die in Beziehung stehenden Elementtypen E_1 und E_2 gemacht. Eine Instanz eines solchen Beziehungstyps kann daher zur Verbindung beliebiger Objekte verwendet werden.

Erst durch die Verwendung weiterer Axiome werden Aussagen über den Definitions- und Zielbereich einer Property gemacht und so die Elementtypen E_1 und E_2 bestimmt. Zu beachten ist, dass bei OWL zu einer einzigen Property mehrere Axiome angegeben werden können, die sich auf Definitions- und Zielbereich beziehen:

```
ObjectPropertyRange( :hasAuthor :Person )
ObjectPropertyRange( :hasAuthor :Author )
```

Die Bedeutung dieser Schreibweise ist, dass ein Individuum, das zu einer so spezifizierten Beziehung gehört, sowohl Instanz des einen als auch des anderen Elementtyps sein muss.[197]

Konkrete Syntax

Mit Hilfe einer `Declaration` wird die Existenz einer Object Property angezeigt und ihr ein Name zugewiesen. Mit Hilfe der Axiome `ObjectPropertyDomain` und `ObjectPropertyRange` können Aussagen über die Elementtypen der an der Beziehung beteiligten Objekte gemacht werden.

```
Declaration( ObjectProperty( R ) )
ObjectPropertyDomain( R E1 )
ObjectPropertyRange( R E2 )
```

Ebenso wie für eine Object Property gibt es entsprechende Schlüsselwörter auch für eine Data Property:

```
Declaration( DataProperty( R ) )
DataPropertyDomain( R E1 )
DataPropertyRange( R E2 )
```

[197] Vgl. W3C: OWL 2 Direct Semantics, Abschnitt 2.3.2.

Abstrakte Syntax

In abstrakter Syntax gehört zu einer Object Property mit (optionaler) Angabe von Definitions- und Zielbereich eine Instanz einer *Declaration*, die die Instanz von *ObjectProperty* mit einer Ontologie verbindet – Abbildung 9.4 zeigt den zugehörigen Abschnitt aus dem OWL-Metamodell. Eine Instanz von *ObjectProperty* kann nicht direkt in Beziehung mit einer Instanz von *Ontology* stehen, sondern stets in Verbindung mit einem Axiom.

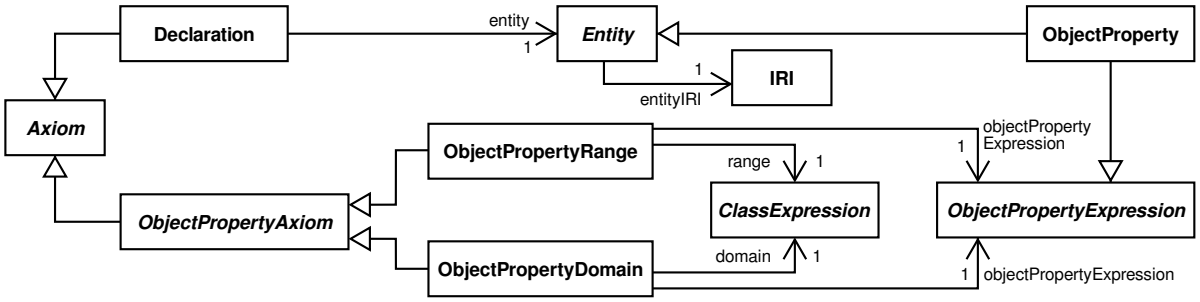


Abbildung 9.4. Der *ObjectProperty* sowie die Definitions- und Zielbereich definierende Axiome `ObjectPropertyDomain` und `ObjectPropertyRange` betreffende Auszug aus dem OWL-Metamodell.

Der Definitions- bzw. Zielbereich der Object Property wird mit Hilfe von Instanzen der Elementtypen *ObjectPropertyDomain* bzw. *ObjectPropertyRange* festgelegt. Diese Elementtypen sind Untertypen von *ObjectPropertyAxiom* sowie *Axiom*. Daher können ihre Instanzen direkt Bestandteil einer Ontologie sein. Sowohl Instanzen von *ObjectPropertyDomain* als auch *ObjectPropertyRange* sind mit jeweils genau einer Instanz von *ClassExpression* bzw. *ObjectPropertyExpression* verbunden. Wie oben beschrieben, können jedoch mehrere Instanzen von *ObjectPropertyDomain* bzw. *ObjectPropertyRange* in Verbindung mit einer *ObjectProperty*-Instanz gesetzt werden.

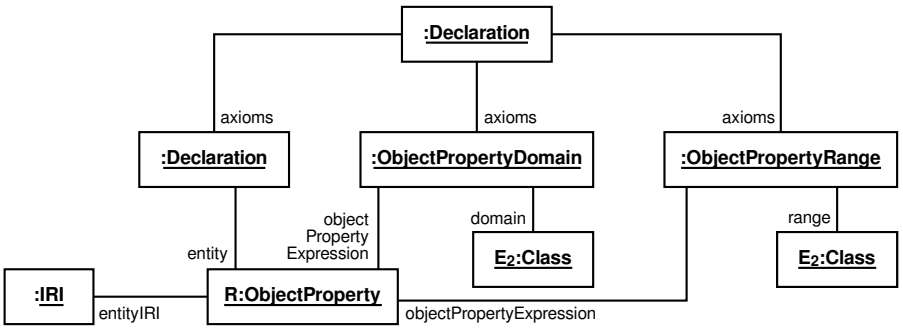


Abbildung 9.5. Objektdiagramm, das die abstrakte Syntax des im Abschnitt zur konkreten Syntax gezeigten Beispiels für die Definition einer Object Property mit Definitions- und Zielbereich darstellt.

Die Abbildung 9.5 zeigt ein Objektdiagramm des zum obigen Beispiel der konkreten Syntax gehörenden Modells. Auch hier lässt sich deutlich erkennen, dass zwar die Axiome zum Definieren von Definitions- und Zielbereich direkt zur Ontologie gehören, die Object

Property selbst aber nur mittelbar über die *Declaration*-Instanz.

Die abstrakte Syntax für Data Properties ist ähnlich der für Object Properties. Der entsprechende Ausschnitt aus dem OWL-Metamodell ist in Abbildung 9.6 zu sehen. Der wichtigste Unterschied besteht darin, dass der Zielbereich einer Data Property eine Instanz von *DataRange* (und nicht *ClassExpression* wie bei einer Object Property) ist.

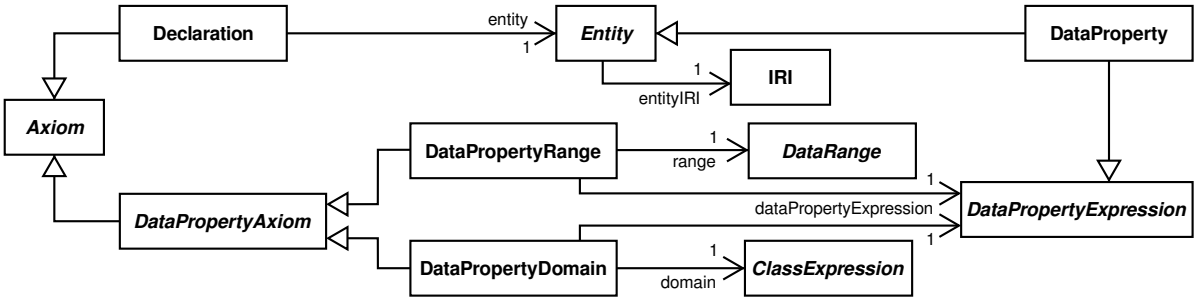


Abbildung 9.6. Der *DataProperty* sowie die Definitions- und Zielbereich definierende Axiome `DataPropertyDomain` und `DataPropertyRange` betreffende Auszug aus dem OWL-Metamodell.

9.1.4 Transformation UML → OWL

Wie bereits geschildert, sieht die UML zwei verschiedene Möglichkeiten vor, um Klassen miteinander zu verbinden: Assoziationen und klassenabhängige Attribute. Im UML-Metamodell werden beide Varianten durch denselben Elementtyp *Property* repräsentiert, wie in Abbildung 9.2 zu sehen ist. In beiden Fällen wird der Typ der *Property*-Instanz durch eine Beziehung zu einer Instanz des (abstrakten) UML-Elementtyps *Type* in der Rolle *type* angegeben. Aufgrund dieser Gemeinsamkeit ist es sinnvoll, die Transformation von Assoziationen und klassenabhängigen Attributen gemeinsam zu betrachten, da sie sich nur in wenigen Dingen unterscheiden.

Zunächst zu den Unterschieden, die bei der Abbildung von Assoziationen und klassenabhängigen Attributen zu beachten sind:

Da der eine Assoziation repräsentierende UML-Elementtyp *Association* ein Untertyp von *Classifier* ist, sind in einem UML-Modell alle Assoziationen direkte Bestandteile eines UML-Pakets. Das OWL-Konzept, das einer Assoziation ähnlich ist, ist eine Object Property, die über ihre Deklaration ebenfalls Bestandteil einer Ontologie ist. Da Pakete auf Ontologien abgebildet werden (→ 11.1 Pakete), passt dies gut zusammen. Assoziationen können unidirektional oder bidirektional sein. Eine unidirektionale Assoziation wird auf *eine* Object Property abgebildet. Da eine bidirektionale Assoziation gleichwertig zu zwei gerichteten, zueinander inversen Assoziationen zwischen beiden Elementtypen ist,[198] werden für eine bidirektionale Assoziation *zwei* Object Properties erzeugt, eine für jede Richtung. Damit die Information erhalten bleibt, dass beide Object Properties zu einer einzigen bidirektionalen Assoziation gehören, wird ein

[198] Vgl. Balzert: Softwaretechnik, S. 165.

entsprechendes `InverseObjectProperties`-Axiom zur Ontologie hinzugefügt – siehe dazu auch den Abschnitt über Inverse (→ 9.9 Inverse).

Die Transformation klassenabhängiger Attribute ist etwas komplizierter, da es zu ihnen kein offensichtlich korrespondierendes Konzept in OWL gibt. Das Hauptproblem besteht darin, dass bei OWL Klassen keine anderen Modellelemente enthalten, so wie es bei UML der Fall ist. Aber auch hier lässt sich die Verbindung durch eine Object Property bzw. eine Data Property abbilden.

Sowohl im Fall einer Assoziation als auch im Fall klassenabhängiger Attribute ist die Wahl, ob eine Instanz des UML-Elementtyps *Property* in eine Object Property oder eine Data Property abgebildet wird, davon abhängig, auf welche Art von Elementtyp der Typ der *Property*-Instanz verweist: Handelt es sich dabei um die Instanz des UML-Elementtyps *Class* oder um einen zusammengesetzten Datentyp (d.h. Instanz des UML-Elementtyps *DataType* mit abhängigen Attributen), so ist eine Object Property zu verwenden. Ist es hingegen die Instanz eines einfachen Datentyps (d.h. eine Instanz des UML-Elementtyps *PrimitiveDataType* oder *Enumeration*), so wird eine Data Property verwendet.

Anders als in der UML, wo die Typen einer Assoziation (Definitionsbereich und Zielbereich) stets angegeben sind, ist es in OWL nicht notwendig, Definitionsbereich und Zielbereich für Properties anzugeben. In diesem Fall wird für den jeweiligen Bereich `owl:Thing` angenommen. Dies entspricht der OWA: Sind keine weiteren Informationen über eine Verbindung bekannt, so kann sie zwischen Individuen beliebiger Elementtypen existieren. Um jedoch die Properties ähnlich wie bei der CWA eines UML-Modells einzuschränken, ist es notwendig, entsprechende Axiome für Definitions- und Zielbereich anzugeben, die die erlaubten Klassen und Datentypen spezifizieren.

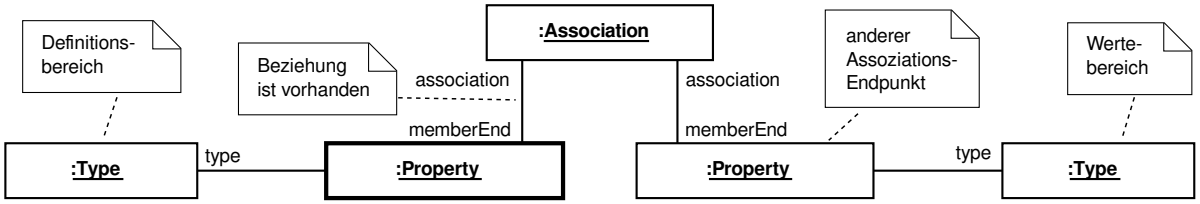


Abbildung 9.7. Wahl von Defintions- und Zielbereich im Falle einer Assoziation. Betrachtet wird die dicker umrandete *Property*-Instanz.

Der **Zielbereich** einer Instanz des UML-Elementtyps *Property* kann folgendermaßen bestimmt werden (vgl. den betreffenden Auszug aus dem UML-Metamodell in Abbildung 9.2): Es handelt sich dabei um den Elementtyp, der durch die Instanz des UML-Meta-Elementtyps *Type* in der Rolle *type* mit der *Property*-Instanz in Beziehung steht.

Um den **Definitionsbereich** zu bestimmen, muss wiederum zwischen Assoziationen und klassenabhängigen Attributen unterschieden werden:

▷ Bei klassenabhängigen Attributen existiert eine Beziehung zwischen der *Property*-Instanz und einer *Class*-Instanz in der Rolle *class*. Der durch diese *Class*-Instanz repräsentierte

Elementtyp ist der gesuchte Typ.

▷ Ist die *Property*-Instanz Teil einer Assoziation (und es gibt eine Beziehung zu einer *Association* in der Rolle *association*), so muss der Typ des anderen Assoziationsendpunktes gewählt werden. Abbildung 9.7 verdeutlicht die Auswahl der beiden Bereiche.

Bedingt durch die OWA wäre es möglich, dass zwei OWL-Properties, die das Transformationsergebnis verschiedener Instanzen des UML-Elementtyps *Property* sind, als eine einzige interpretiert werden. Um dies zu vermeiden und die CWA der UML bestmöglich abzubilden, werden alle diejenigen Properties, die nicht in einer Vererbungsbeziehung (→ 9.3 VERERBUNG) stehen, als verschieden ("disjoint") markiert. Dazu werden der Ontologie `DisjointObjectProperties`- bzw. `DisjointDataProperties`-Axiome hinzugefügt.

Abbildung 9.8. Beispiel für die Transformation von klassenabhängigen Attributen in Object- und Data Properties.

9.1.5 Transformation OWL → UML

In OWL wird, wie oben geschildert, zwischen Data Properties, mit der Klassen und Datentypen verbunden werden, sowie Object Properties, um Klassen mit Klassen zu verbinden, unterschieden. Für die Repräsentation von Data Properties werden in UML klassenabhängige Attribute verwendet. Bei der Betrachtung von Object Properties muss berücksichtigt werden, dass es für die Verbindung zwischen Klassen in UML zwei verschiedene Modellelemente gibt: Assoziationen und klassenabhängige Attribute. Bis auf einige – unten diskutierte – Ausnahmen werden Object Properties in klassenabhängige Attribute transformiert, wie in Abbildung 9.9 gezeigt ist. Die dazugehörige QVT-Regel ist in Abbildung 9.10 dargestellt.

Aufgrund der deklarativen Natur von QVT ist es einfach, verschiedene Regeln zu spezifizieren, die sich um einzelne Aspekte der Transformation kümmern. Eine Data Property (Instanz

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Book ) )
  Declaration( Class( :Person ) )

  Declaration( ObjectProperty( :author ) )
  ObjectPropertyDomain( :author :Book )
  ObjectPropertyRange( :author :Person )

  Declaration( DataProperty( :title ) )
  DataPropertyDomain( :title :Book ) )
  DataPropertyRange( :title xsd:string )
)
```

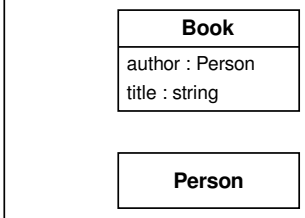


Abbildung 9.9. Beispiel für die Transformation von Object- und Data Properties in klassenabhängige Attribute.

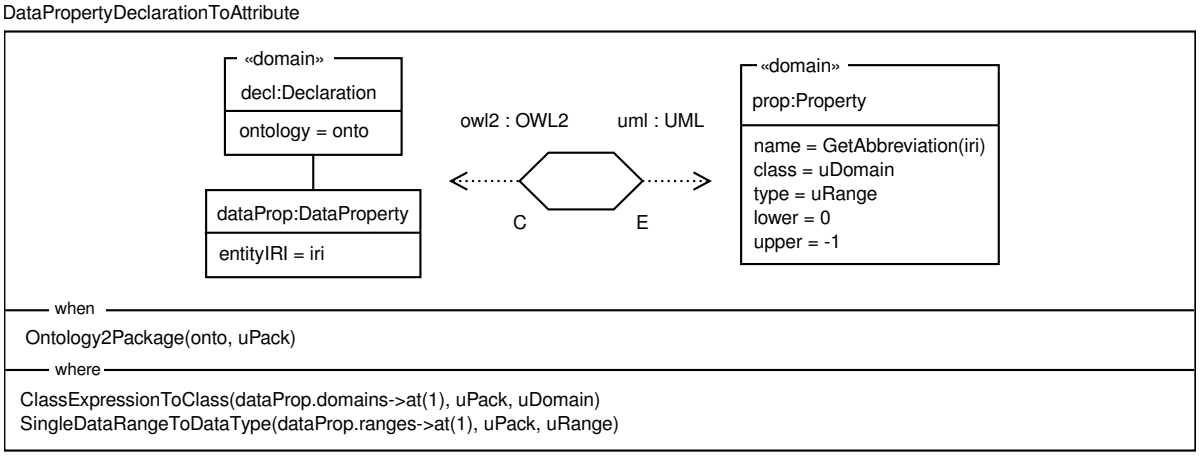


Abbildung 9.10. QVT-Regel zum Transformieren einer Data Property.

des OWL-Elementtyps *DataProperty*) wird immer auf eine Instanz des UML-Elementtyps *Property* abgebildet. Die entsprechende Regel ist in Abbildung 9.10 zu sehen. Eine quasi identische Regel existiert für Object Property, wobei hier in der `when`-Klausel weitere Bedingungen enthalten sind. Diese sorgen dafür, dass in bestimmten Fällen die Regel nicht angewendet wird, da hier eine Instanz des UML-Elementtyps *Association* benötigt wird.

Gesondert behandelt werden solche Object Property-Verbindungen, zu denen innerhalb der Ontologie eine inverse Verbindung existiert. Eine solche kann auf mehrere Arten spezifiziert sein: durch explizite Angabe mit Hilfe des `InverseObjectProperties`-Axioms, durch Verwendung einer anonymen inversen `InverseObjectProperty` oder durch Kennzeichnung einer Object Property als symmetrisch oder invers-funktional. Diese Fälle werden in den entsprechenden Abschnitten (→ 9.9 Inverse) und (→ 9.8 Symmetrie) besprochen.

Wie oben geschildert, muss damit gerechnet werden, dass für Werte- oder Bildbereich mehr als eine Klasse angegeben ist. Da in UML diese Notation nicht möglich ist, wird in solchen Fällen eine Hilfsklasse eingefügt, die von allen beteiligten Definitions- bzw. Zielbe-

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  ObjectPropertyRange( :hasAuthor :Person )
  ObjectPropertyRange( :hasAuthor :Author )
)
```

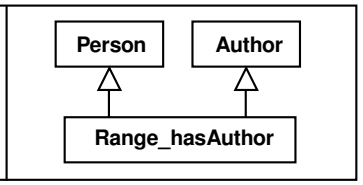


Abbildung 9.11. Beispiel für die Transformation von mehreren Axiomen, die den Zielbereich einer Object Property angeben, in UML-Klassen mit Vererbungsbeziehungen.

reichsklassen erbt. Abbildung 9.11 zeigt ein Beispiel für eine solche Konstruktion. An den Stellen, wo die betroffene Object Property verwendet wird, wird im UML-Modell entsprechend die neue Hilfsklasse verwendet.

Eine Möglichkeit, Object Properties mit `owl:Thing` als Definitions- und Zielbereich in UML abzubilden, besteht darin, eine einzige Oberklasse C_{super} zu definieren, die Oberklasse aller anderen Klassen im Modell ist und `owl:Thing` entspricht. In diesem speziellen Fall einer einzigen Oberklasse ließe sich eine Object Property ohne Definitions- und Zielbereich als Instanz des UML-Elementtyps *Association* mit zwei *members ends* vom Typ C_{super} darstellen.

9.1.6 Ausdeutung für XSD

Mit Hilfe von Beziehungstypen werden zwei Elementtypen in Beziehung miteinander gesetzt. Dieser Beziehung (bzw. den Rollen, die die teilnehmenden Instanzen der Elementtypen spielen) kann ein Name gegeben werden. In XSD stellen die in Complex Types enthaltenen Elemente eine solche benannte Verbindung dar. Daher lässt sich ein solches Element als Beziehungstyp verstehen.

Das anfangs genannte Beispiel des Beziehungstyps $Name(Person, String)$, der als zweiten Teilnehmer einen Datentypen besitzt, lässt sich in XSD als ein Element vom Typ `Person` mit einem Kindelement `name` vom Typ `xsd:string` darstellen. Auch allgemeine Beziehungstypen – die als zweiten Teilnehmer nicht unbedingt einen Datentyp besitzen – lassen sich auf diese Weise darstellen. Ein Beispiel dafür ist der anfangs genannte Beziehungstyp $ArbeitetBei(Arbeitnehmer : Person, Arbeitgeber : Firma)$. Hierbei besteht die Möglichkeit, als Elementnamen den Namen des Beziehungstyps "ArbeitetBei" oder den Namen der Rolle des zweiten Teilnehmers "Arbeitgeber" zu verwenden.

```
1 <xsd:complexType name="Person">
2    <xsd:sequence>
3       <xsd:element name="name" type="xsd:string" />
4       <xsd:element name="ArbeitetBei" type="Firma" />
5    </xsd:sequence>
6 </xsd:complexType>
```

Eine Schwierigkeit entsteht dann, wenn

- sich – wie in dem Beispiel *ArbeitetBei* zu vermuten – mehrere Beziehungen auf dasselbe Objekt (als zweiter Teilnehmer) beziehen, oder
- beide Rollen eines Beziehungstyps (bzw. ein Beziehungstyp und sein inverser (→ 9.9 Inverse) dargestellt werden sollen.

Dies lässt sich zwar korrekt in XSD notieren (wieder am Beispiel von *ArbeitetBei*, diesmal unter Nutzung der Rollennamen):

```
<xsd:complexType name="Person">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="Arbeitgeber" type="Firma" />
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Firma">
   <xsd:sequence>
      <xsd:element name="Arbeitnehmer" type="Person" />
   </xsd:sequence>
</xsd:complexType>
```

es gibt allerdings kein endlich großes XML-Dokument, das valide bezüglich dieses Schemas ist und ein Element der Typen `Person` oder `Firma` enthält.[199]

Ein – auch bei GML gewählter – Ausweg für die praktische Nutzung eines solchen Modells besteht darin, Referenzen auf XML-Elemente zu verwenden. In dem Fall muss jedoch auf die Typsicherheit, die das Validieren gegen ein XML-Schema bietet, verzichtet werden.

```
<xsd:complexType name="Person">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="Arbeitgeber" type="FirmaReference" />
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="PersonReference">
   <xsd:sequence />
   <xsd:attributeGroup ref="xlink:simpleLink"/>
</xsd:complexType>

<xsd:complexType name="FirmaReference">
   <xsd:sequence />
   <xsd:attributeGroup ref="xlink:simpleLink"/>
</xsd:complexType>

<xsd:complexType name="Firma">
   <xsd:sequence>
      <xsd:element name="Arbeitnehmer" type="PersonReference" />
   </xsd:sequence>
</xsd:complexType>
```

[199] Jedes Element vom Typ `Person` erfordert ein Element vom Typ `Firma`, das ein Element vom Typ `Person` erfordert, das ein Element vom Typ `Firma` erfordert, ...

9.2 Ordnungen

In manchen Fällen ist es notwendig, Aussagen darüber zu treffen, in welcher Reihenfolge die Instanzen einer Beziehung auftreten. Ein Beispiel dafür ist eine Warteschlange, bei der die Abfolge der Elemente entscheidend ist, um z.B. ein first-in-first-out Prinzip zu realisieren.

9.2.1 Logische Repräsentation

Eine mögliche Repräsentation einer geordneten Beziehung besteht darin, dem Beziehungstyp ein weiteres Element hinzuzufügen, das die Position innerhalb der geordneten Liste angibt.[200]

$$R(p_1 : E_1, p_2 : E_2, ordnung : n), n \in \mathbb{N}$$

9.2.2 Repräsentation in UML

Konkrete Syntax

Für geordnete Beziehungen sieht die UML das Schlüsselwort *ordered* vor, das in die Nähe des geordneten Teilnehmers geschrieben wird. Abbildung 9.12 zeigt ein Beispiel.

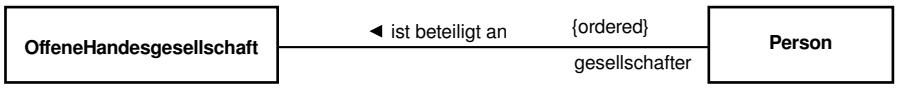


Abbildung 9.12. Darstellung einer geordneten Beziehung in grafischer UML-Syntax.

Abstrakte Syntax

Im UML-Metamodell findet sich die Information, dass es sich um eine geordnete Beziehung handelt, in der Eigenschaft *isOrdered* der *Property*-Klasse des entsprechenden Member-Ends wieder.

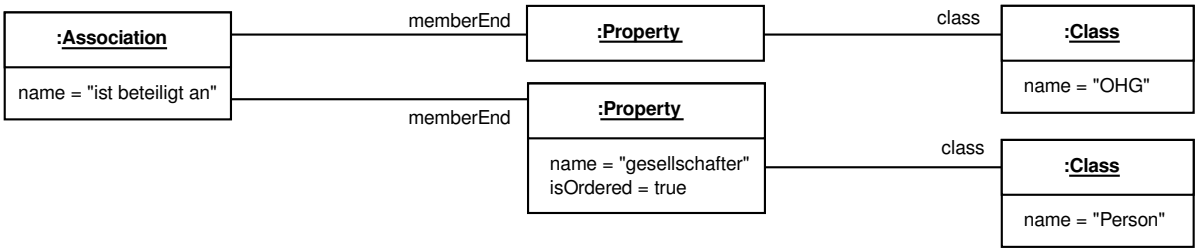


Abbildung 9.13. Objektdiagramm, das die abstrakte Syntax des Beispiels aus Abbildung 9.12 zeigt.

[200] Vgl. Olivé: Conceptual Modeling, S. 74.

9.2.3 Repräsentation in OWL

OWL kennt keine Möglichkeit, eine Reihenfolge festzulegen, in der die Properties zwischen Individuen bzw. Individuum und Wert auftreten. Da es sich bei der Beziehung zwischen Ontologien und Axiomen – und damit auch Assertions – um eine ungeordnete Menge handelt,[201] ist die Abfolge, in der die Assertions in der serialisierten Ontologie auftreten, ohne Bedeutung.

Dies ist jedoch nicht damit zu verwechseln, dass innerhalb des OWL-Metamodells durchaus geordnete Beziehungstypen auftreten, deren Reihenfolge entscheidend ist.[202]

9.2.4 Ausdeutung für XSD

Durch die aus der Serialisierung von XML folgende Dokumentenreihenfolge, ergibt sich automatisch eine Ordnung der Instanzen des Beziehungstyps. Es ist daher nicht erforderlich, explizit anzugeben, dass es sich um eine geordnete Beziehung handeln soll.

Anmerkung: Die Reihenfolge, mit der Instanzen eines Beziehungstyps im Dokument auftreten, legt eine Ordnung unter ihnen fest. Die Verwendung einer `sequence` regelt dagegen die Reihenfolge, in der die Instanzen *verschiedener* Beziehungstypen im Dokument auftreten müssen. Bei Verwendung von `all` tritt das Problem nicht auf, da hier für die enthaltenen Beziehungstypen nur die Kardinalitäten 0..1 zulässig sind.

9.3 Vererbung

Manchmal ist es notwendig, dass aus einer Beziehung zwischen zwei Objekten automatisch eine andere Beziehung folgt. Anders ausgedrückt, die Population eines Beziehungstyps R_2 schließt die Population eines anderen Beziehungstyps R_1 mit ein – alle Instanzen von R_2 sind automatisch auch Instanzen von R_1. Weitere Sprechweisen: R_1 ist Untertyp von R_2, R_2 subsumiert R_1, R_1 erbt von R_2.

Ein beispielhafte Anwendung dafür ist:

```
ArbeitetBei( Arbeitnehmer: Person, Arbeitgeber: Firma )
Leitet( Manager: Person, Firma )
```

Unter der Annahme, dass jeder Manager auch bei der Firma arbeitet, die er leitet, ist *Leitet* ein Untertyp von *ArbeitetBei*.

[201] Vgl. W3C: OWL 2 Structural specification, Abschnitt 3 und Abschnitt 9.
[202] Vgl. a. a. O., Abschnitt 2.1.

9.3.1 Logische Repräsentation

Logisch lässt sich die Tatsache, dass $R_1(p_{1,1} : E_1, \ldots, p_{1,n} : E_n)$ ein Unter-Beziehungstyp von $R_2(p_{2,1} : E_1, \ldots, p_{2,n} : E_n)$ ist, mit Hilfe folgender Formel ausdrücken:

$$R_1(e_1, e_2) \rightarrow R_2(e_1, e_2)$$

9.3.2 Repräsentation in UML

In der UML wird eine Vererbung von Beziehungstypen ähnlich wie die Vererbung zwischen Elementtypen realisiert. Werden die beiden Beziehungstypen durch eine Assoziation repräsentiert, so lässt sich eine *Generalization* zwischen diesen Assoziationen erstellen.

Werden klassenabhängige Attribute verwendet, so lässt sich ein untergeordnetes (engl: subsetted) Attribut angeben.

Konkrete Syntax

Grafisch wird die Vererbung zwischen zwei Beziehungstypen dadurch symbolisiert, dass ein Pfeil mit weißem Dreieck zwischen den die Assoziationen repräsentierenden Linien/Pfeilen eingezeichnet wird.

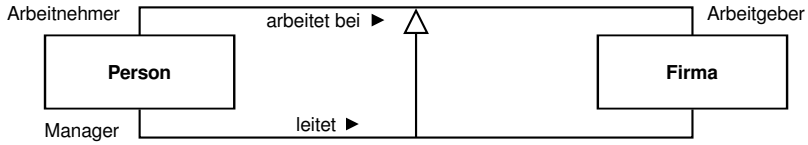


Abbildung 9.14. Vererbung zwischen zwei Beziehungstypen in konkreter grafischer UML-Syntax.

Abstrakte Syntax

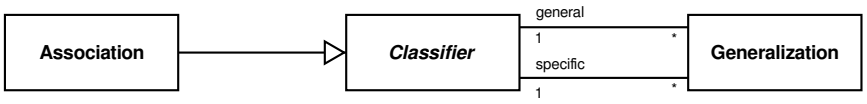


Abbildung 9.15. Ausschnitt aus dem UML-Metamodell, der den die Vererbung von Beziehungstypen im Falle von Assoziationen betreffenden Ausschnitt zeigt.

In abstrakter Syntax wird die Vererbung von Beziehungstypen als eine *Generalization*-Instanz dargestellt, die zwei *Assoziation*-Instanzen miteinander verbindet. Jede dieser *Assoziation*-Instanzen repräsentiert dabei – wie oben geschildert – einen der beiden Beziehungstypen. Der Unterbeziehungstyp tritt dabei in der Rolle *specific*, der Ober-Beziehungstyp in der Rolle *general* auf. Dieser Zusammenhang zwischen den Meta-Elementtypen ist in Abbildung 9.15 dargestellt. Die Abbildung 9.16 zeigt ein Objektdiagramm, das die für das obige Beispiel benötigten Instanzen der UML-Meta-Elementtypen zeigt.

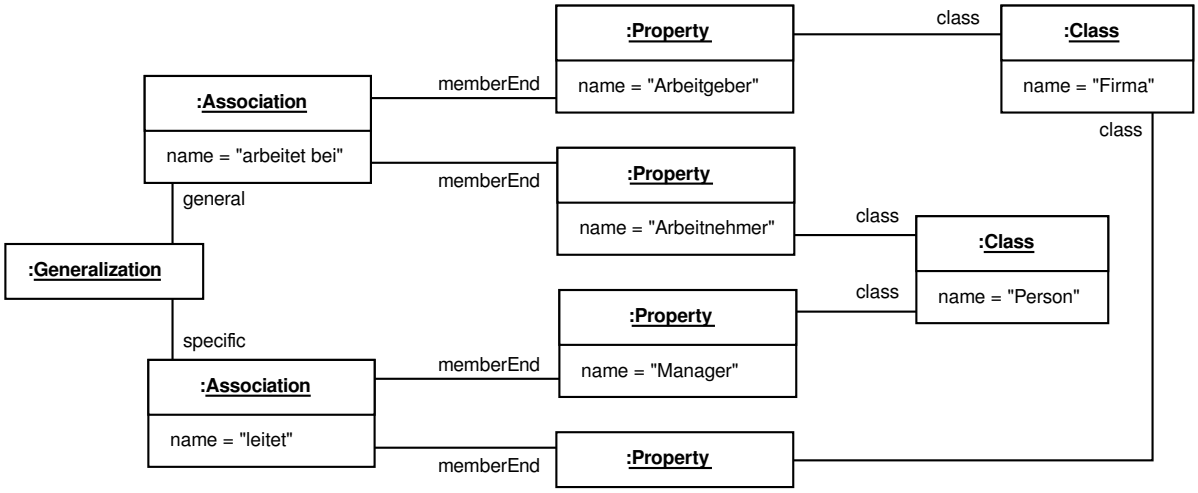


Abbildung 9.16. Objektdiagramm, das die abstrakte Syntax des in Abb. 9.14 gezeigten Beispiels darstellt.

Bei der Verwendung klassenabhängiger Attribute, die in Form von *Property*-Instanzen vorliegen, werden diese – wie in Abbildung 9.17 zu sehen – in eine *subsettedProperty*-Beziehung gesetzt.

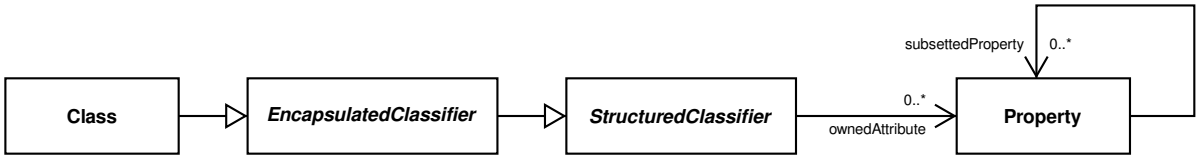


Abbildung 9.17. Ausschnitt aus dem UML-Metamodell, der den die Vererbung von Beziehungstypen im Fall klassenabhängiger Attribute betreffenden Ausschnitt zeigt.

9.3.3 Repräsentation in OWL

OWL ermöglicht es, auszudrücken, dass die Population eines Beziehungstyps R_1 die Population eines anderen Beziehungstyps R_2 einschließt. Dabei wird unterschieden, ob zwei Object Properties oder zwei Data Properties in einer Vererbungsbeziehung stehen.

Konkrete Syntax

Zwei Properties können folgendermaßen in eine Vererbungsbeziehung gesetzt werden:

```
SubObjectPropertyOf( a, b )
SubDataPropertyOf( a, b )
```

Dabei wird *a* zu einem Untertyp von *b*.

Abstrakte Syntax

In abstrakter Syntax wird die Vererbung von Properties durch eine Instanz des Elementtyps *SubObjectProperty* bzw. *SubDataProperty* repräsentiert.

Während *SubDataProperty*-Instanzen (siehe Abbildung 9.18) jeweils eine Verbindung zu einer *DataPropertyExpression* als Obertyp und eine als Untertyp besitzen, kann eine Instanz von *SubObjectProperty* mit mehr als zwei *ObjectPropertyExpressions* in Verbindung stehen (siehe Abbildung 9.19).

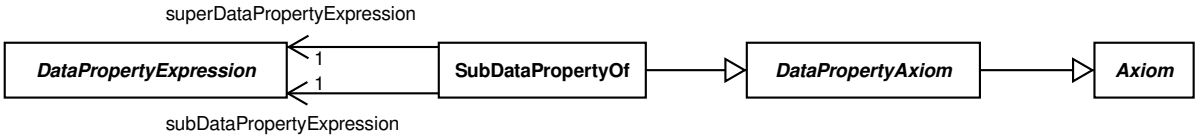


Abbildung 9.18. Ausschnitt aus dem OWL-Metamodell, der den *SubDataProperty* betreffenden Ausschnitt zeigt.

Abbildung 9.19. Ausschnitt aus dem OWL-Metamodell, der den *SubObjectProperty* betreffenden Ausschnitt zeigt.

9.3.4 Transformation UML → OWL

Eine Generalisierungsbeziehung (Instanz des UML-Elementtyps *Generalization*) zwischen zwei Instanzen des UML-Elementtyps *Association* kann in OWL mit Hilfe von Instanzen des OWL-Elementtyps *SubPropertyOf* abgebildet werden. Ein Beispiel für eine solche Transformation ist in Abbildung 9.20 zu sehen.

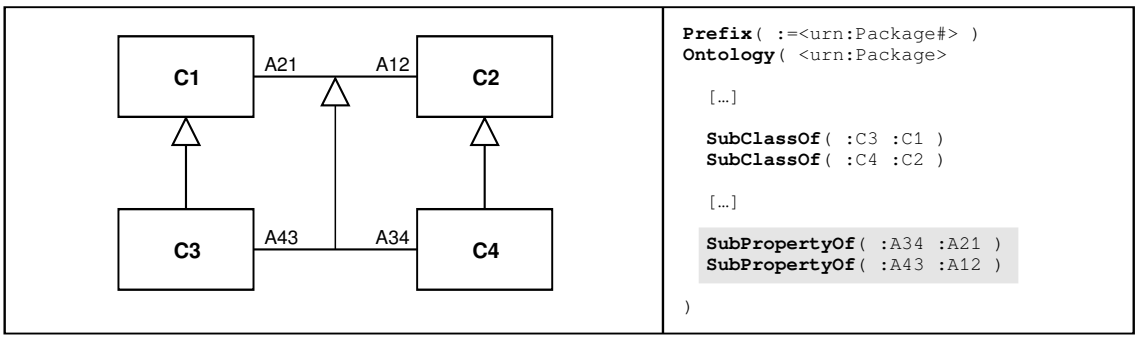


Abbildung 9.20. Beispiel für die Transformation von generalisierten Assoziationen.

Da eine bidirektionale Assoziation in zwei Object Properties transformiert wird, muss ebenfalls die Generalisierungsbeziehung zwischen zwei bidirektionalen Assoziationen in zwei Instanzen des OWL-Elementtyps *SubPropertyOf* transformiert werden.

9.3.5 Transformation OWL → UML

Ähnlich wie die Definition von Unter-Beziehungstypen bei OWL mit Hilfe von Instanzen des OWL-Elementtyps *SubObjectPropertyOf* kennt auch UML die Angabe übergeordneter Beziehungstypen: Das klassenabhängige Attribut *subsettedProperty* an einer Instanz des UML-Elementtyps *Property* ermöglicht die Angabe einer übergeordneten *Property*-Instanz. Die hierfür notwendige einfache QVT-Regel ist in Abbildung 9.21 zu sehen.

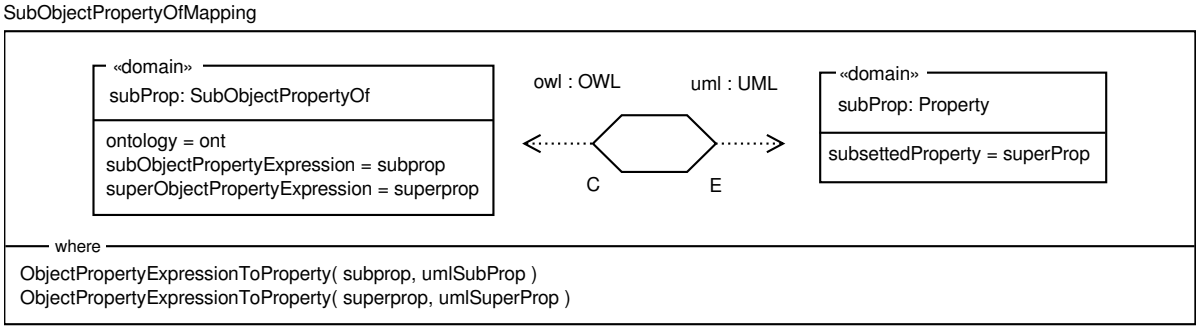


Abbildung 9.21. QVT-Regel für die Transformation der Instanz des OWL-Elementtyps *SubObjectPropertyOf*.

9.3.6 Ausdeutung für XSD

Werden, wie oben dargestellt, Beziehungstypen als verschachtelte Elemente aufgefasst, so lässt sich die Vererbung von Beziehungstypen mit Hilfe von so genannten *substitution groups* modellieren. Enhält eine *ElementDeclaration A* ein oder mehrere Referenzen auf eine andere *ElementDeclaration B* (in Form des klassenabhängigen Attributs *substitutionGroup*), so lässt sich das Element *A* überall dort einsetzen, wo das Element *B* angegeben ist.

In konkreter Syntax sieht das eingangs vorgestellte Beispiel mit den Beziehungstypen *ArbeitetBei* und *Leitet* folgendermaßen aus:

```
 1 <xsd:complexType name="Person">
 2    <xsd:sequence>
 3       <xsd:element name="name" type="xsd:string" />
 4       <xsd:element ref="arbeitetBei" />
 5    </xsd:sequence>
 6 </xsd:complexType>
 7
 8 <xsd:element name="arbeitetBei" type="Firma" />
 9
10 <xsd:element name="leitet" substitutionGroup="arbeitetBei" type="Firma"/>
```

Anmerkung: Da sich die Angabe beim Attribut `substitutionGroup` nur auf eine Top-Level-*ElementDeclaration* beziehen kann, ist es notwendig, das Element `arbeitetBei` entsprechend top-level zu deklarieren (Zeile 8) und auch innerhalb des Typs `Person` entsprechend zu referenzieren (Zeile 4).

9.4 Kardinalitätsbeschränkungen

Kardinalitätsbeschränkungen stellen eine der wichtigsten Arten der Beschränkungen bei konzeptueller Modellierung dar. Sie ermöglichen unter anderem eine technische Beschränkung der Menge der Beziehungen, die ein Objekt mit anderen Objekten eingehen kann. So kann beispielsweise im Vorfeld Speicherplatz entsprechend reserviert werden. Aber auch für die Semantik eines Modells spielen Kardinalitätsbeschränkungen eine große Rolle, da sie es in vielen Fällen erst ermöglichen, die genaue Bedeutung der auftretenden Typen erkennen zu können.[203]

Da OWL nur binäre Beziehungen zwischen Objekten kennt, werden nur Kardinalitätsbeschränkungen für diese Art von Beziehungen betrachtet.

9.4.1 Logische Repräsentation

Sei $R(p_1 : E_1, p_2 : E_2)$ ein binärer Beziehungstyp zwischen Instanzen p_1 des Elementtyps E_1 und Instanzen p_2 des Elementtyps E_2. Die Kardinalitätsbeschränkung (oder auch kurz: Kardinalität) zwischen p_1 und p_2 in Relation R, geschrieben $Card(p_1; p_2; R)$, ist ein Paar

$$Card(p_1; p_2; R) = (min, max)$$

das angibt, wie viele Elemente des Typs E_2 mindestens und höchstens in Relation R mit einem Element des Typs E_1 stehen dürfen. Die minimale Kardinalität min muss größer oder gleich null sein, die maximale Kardinalität max muss größer als null und nicht kleiner als min sein. Formal:

$$E_1(e_1) \rightarrow min \leqslant |\{e_2|R(e_1, e_2)\}| \leqslant max$$

$\{e_2|R(e_1, e_2)\}$ ist dabei die Menge derjenigen Elemente e_2, die in Beziehung R zum Element e_1 stehen.

Als Kurzschreibweise werden verwendet:

$$C_{max}(p_1; p_2; R) \Leftrightarrow E_1(e_1) \rightarrow |\{e_2|R(e_1, e_2)\}| \leqslant max$$
$$C_{min}(p_1; p_2; R) \Leftrightarrow E_1(e_1) \rightarrow min \leqslant |\{e_2|R(e_1, e_2)\}|$$

[203] Vgl. Olivé: Conceptual Modeling, S. 83.

Es handelt sich um eine *funktionellen Abhängigkeit* zwischen p_1 und p_2 (bezüglich der Relation R), wenn $C_{max}(p_1; p_2; R) = 1$ gilt.

Bei folgenden Kardinalitäten handelt es sich eigentlich nicht um Einschränkungen, es wird auch von *unbeschränkten* Relationen gesprochen:

- $C_{min}(p_1; p_2; R) = 0$
- $C_{max}(p_1; p_2; R) = \infty$

9.4.2 Repräsentation in UML

In UML werden Kardinalitäten *Multiplizität* genannt. Der Wert der Kardinalität

$$Card(p_1; p_2; R) = (min, max)$$

wird in der Nähe von p_2 geschrieben, bei klassenabhängigen Attributen in eckige Klammern. Dabei werden *min* und *max* durch zwei Punkte (..) getrennt aufgeschrieben. Handelt es sich um eine nach oben unbeschränkte Beziehung

$$C_{max}(p_1; p_2; R) = \infty$$

so wird für die obere Schranke ein Sternchen (*) verwendet. Im Falle von

$$Card(p_1; p_2; R) = (0, \infty)$$

wird nur ein Sternchen ohne untere Schranke notiert. Ist keine Kardinalität angegeben, wird der Standardwert

$$Card(p_1; p_2; R) = (1, 1)$$

verwendet.

Konkrete Syntax

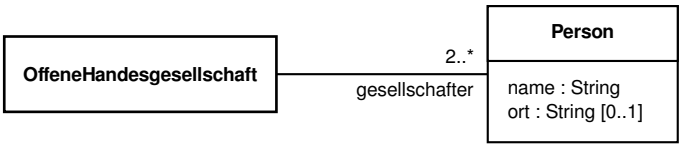


Abbildung 9.22. Beispiel für drei verschiedene Kardinalitätsbeschränkungen in grafischer UML-Syntax.

Die Abbildung 9.22 zeigt drei verschiedene Kardinalitätsbeschränkungen: Eine Offene Handelsgesellschaft muss zwischen zwei und einer unbeschränkten Anzahl (*) Personen als Gesellschafter haben. Eine Person muss genau einen Namen (Standardwert, da sonst keine Angabe vorhanden) haben. Es kann für eine Person kein oder höchstens ein Ort (0..1) angegeben sein.

Abstrakte Syntax

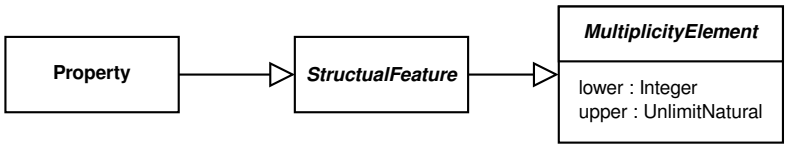


Abbildung 9.23. Ausschnitt aus dem UML-Metamodell, der *Property* mit seinen Oberklassen zeigt.

Im UML-Metamodell finden sich die Kardinalitätsbeschränkungen in Form eines *MultiplicityElement* wieder, das Oberklasse von *Property* ist, wie in Abbildung 9.23 zu sehen ist.

9.4.3 Repräsentation in OWL

OWL kennt Konstrukte zum Beschreiben der Kardinalität. Es sind Class Expression (CE), die dazu dienen, bestimmte Mengen von Individuen zu beschreiben. Im folgenden Abschnitt wird wieder eine binäre Relation $R(x, y)$ zweier Individuen x und y betrachtet.

Die CE `ObjectMinCardinality( n R )` beschreibt die Menge der Individuen, die zu mehr als n Individuen in Beziehung R stehen: $\{x : |\{y : R(x, y)\}| \geqslant n\}$.[204] Entsprechende CE gibt es auch für Mengen von Individuen, die mit weniger als n Individuen (*ObjectMaxCardinality*) oder genau n Individuen (*ObjectExactCardinality*) in Beziehung stehen.

Handelt es sich bei dem Typ von y um einen Datentyp, so kommen die CE *DataMinCardinality*, *DataMaxCardinatlity* bzw. *DataExactCardinatliy* zum Einsatz.

Weiterhin kann eine Einschränkung darauf gemacht werden, welchem Elementtyp y angehören muss. Die bereits oben durch die CE `ObjectMinCardinality( n R )` beschriebene Menge kann weiter eingeschränkt werden, indem die Zugehörigkeit von y zu einem Elementtyp E verlangt wird: `ObjectMinCardinality( n R E )` Diese CE beschreibt die Menge der Individuen, die zu mehr als n Instanzen des Elementtyps E in Beziehung R stehen: $\{x : |\{y : R(x, y) \wedge E(y)\}| \geqslant n\}$.[205] Auch für diese Art der Einschränkung gibt es – wie für den obigen allgemeinen Fall – fünf weitere CE.

Konkrete Syntax

Die konkrete Syntax für eine CE, die die Menge der Individuen beschreibt, die zu mehr als n Instanzen des Elementtyps E in Beziehung R stehen, sieht folgendermaßen aus:

```
ObjectMinCardinality( n R E )
```

[204] Vgl. W3C: OWL 2 Direct Semantics, Abschnitt 2.2.3.
[205] Vgl. a. a. O.

Abstrakte Syntax

In abstrakter Syntax werden kardinalitätsbeschränkende CE durch Instanzen der Elementtypen *ObjectMinCardinality*, *ObjectMaxCardinality*, *ObjectExactCardinality*, *DataMinCardinality*, *DataMaxCardinality* sowie *DataExactCardinality* repräsentiert, die Unterklassen von *ClassExpression* sind. Abbildung 9.24 zeigt diesen Zusammenhang am Beispiel der abstrakten Syntax einer `ObjectMinCardinality`-CE.

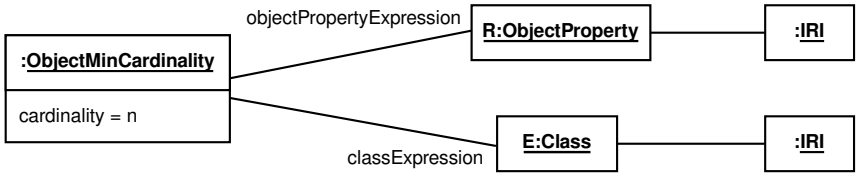


Abbildung 9.24. Objektdiagramm, das die abstrakte Syntax kardinalitätsbeschränkender CE am Beispiel von *ObjectMinCardinality* zeigt.

9.4.4 Transformation UML → OWL

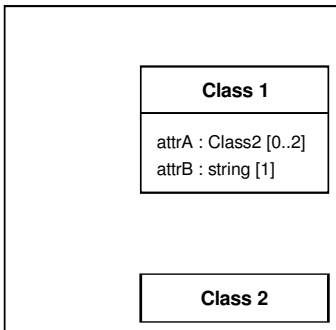
In UML-Modellen können sowohl Assoziationen als auch klassenabhängige Attribute mit Kardinalitätsbeschränkungen versehen werden. Es lässt sich eine untere und/oder obere Schranke für die Häufigkeit des Auftretens einer Assoziation bzw. eines Attributes angeben. OWL besitzt ebenfalls sechs Elementtypen zum Beschränken von Kardinalitäten einer Property: *ObjectMinCardinality*, *ObjectMaxCardinality* und *ObjectExactCardinality* sowie die entsprechenden Elemente für eine Data Properties.

Zwar handelt sich sich bei `ExactCardinality` (wie in Tabelle 3.1 zu sehen) nur um eine Abkürzung eines Paares von `MinCardinality`- und `MaxCardinality`-Elementen, jedoch erhöht die Verwendung dieser Abkürzung die Verständlichkeit einer Ontologie. Daher wird eine UML-Kardinalitätsbeschränkung, bei der untere und obere Schranke denselben Wert haben, in eine Instanz des OWL-Elementtyps *ExactCardinality* mit entsprechendem Wert übersetzt.

Hat die obere Schranke den Wert 1, so wird zusätzlich eine Instanz des OWL-Elementtyps *FunctionalObjectProperty* bzw. *FunctionalDataProperty* erzeugt. Hierbei handelt es sich zwar ebenfalls nur um eine verkürzende Schreibweise, aber auch hier wird durch das zusätzliche Axiom die Verständlichkeit der Ontologie verbessert, da sofort die Art der Property zu erkennen ist.

Es ist jedoch nicht möglich und ausreichend, der Ontologie einfach entsprechende CE für die Kardinalitätsbeschränkungen hinzuzufügen. Zum einen handelt es sich bei CE nicht um Axiome, daher lassen sie sich nicht direkt einer Ontologie zuordnen. Zum anderen bezieht sich in der UML die Einschränkung der Kardinalitäten von klassenabhängigen Attributen und Assoziationen stets auf eine konkrete Klasse. Da in OWL Properties nicht unmittelbar in Klassen enthalten sind (siehe oben), wirken die für Properties spezifizierten Kardinalitätsbeschränkungen in Form von CE hier zunächst nicht auf eine Klasse.

Diese Schwierigkeiten können dadurch gelöst werden, dass der Ontologie für jede Kardinalitätsbeschränkung Instanzen des OWL-Elementtyps *SubClassOf* hinzugefügt werden. Als Untertyp tritt die OWL-Klasse auf, die der UML-Klasse entspricht, für die der Beziehungstyp samt seiner Kardinalitätsbeschränkung definiert wurde. Als Obertyp tritt die zu der Kardinalitätsbeschränkung passende CE auf. So werden die durch die CE definierten Kardinalitätsbeschränkungen an die Klasse vererbt, die in Verbindung mit der Assoziation bzw. dem klassenabhängigen Attribut steht.




```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Class1 ) )
  Declaration( Class( :Class2 ) )
  DisjointClasses( :Class1 :Class2 )

  Declaration( ObjectProperty( :Class1_attrA ) )
  ObjectPropertyDomain( :Class1_attrA :Class1 )
  ObjectPropertyRange( :Class1_attrA :Class2 )
  SubClassOf(
    :Class1
    ObjectMinCardinality( 0 :Class1_attrA :Class2 ) )
  SubClassOf(
    :Class1
    ObjectMaxCardinality( 2 :Class1_attrA :Class2 ) )

  Declaration( DataProperty( :Class1_attrB ) )
  DataPropertyDomain( :Class1_attrB :Class1 ) )
  DataPropertyRange( :Class1_attrB xsd:string )
  SubClassOf(
    :Class1
    DataExactCardinality( 1 :Class1_attrB xsd:string ) )
)
```

Abbildung 9.25. Beispiel für die Transformation von klassenabhängigen Attributen und Assoziationen mit Mengenbeschränkungen.

Abbildung 9.25 zeigt jeweils für Object und Data Properties ein Beispiel, an dem dieses Vorgehen deutlich wird: Eine Instanz der "Class1" darf mit null bis zwei Instanzen der "Class2" in Beziehung stehen (wobei diese in der Rolle "attrA" auftreten). Entsprechend werden bei der Transformation zwei Vererbungsbeziehungen zwischen der OWL-Klasse "Class1" und zwei CE definiert.

Ein Reasoner kann mit Hilfe dieser Kardinalitätsbeschränkungen die Konsistenz der Ontologie überprüfen. Jedoch ist es nur möglich, die Überschreitung oberer Schranken zu bemerken. Die Verletzung einer unteren Schranke kann aufgrund der verwendeten OWA prinzipiell nicht erkannt werden. Allein durch die Angabe einer unteren Schranke ist festgelegt, dass es eine entsprechende Anzahl von Beziehungen in der Ontologie gibt. Die scheinbar noch "fehlenden" Beziehungen sind einfach nicht explizit in der Ontologie aufgeführt wurden.

Höglund et al. beschreiben, wie sich mit einem Reasoner auch Verletzungen einer unteren Schranke prüfen lassen.[206] In der Diskussion mit dem Autor[207] hat sich jedoch ergeben, dass dies im Allgemeinen nicht funktioniert. Nur für den Spezialfall, dass eine Closed World erzwungen wird, lässt sich die Verletzung einer unteren Schranke entdecken. Dazu

[206] Vgl. Höglund et al.: Representing and Validating Metamodels
[207] Schriftwechsel per E-Mail im Oktober 2010.

müssen alle relevanten Klassen abgeschlossen sein, d.h. die zugehörigen Individuen müssen aufgezählt werden.

```
EquivalentClasses( :C ObjectOneOf( :I1 :I2 ... :In ) )
```

Außerdem müssen sowohl alle relevanten Klassen und alle Individuen als verschieden voneinander markiert sein.

```
DifferentIndividuals( :I1 :I2 ... :In )
DisjointClasses( :C1 :C2 ... :Cn )
```

9.4.5 Transformation OWL → UML

Mit Hilfe der CE `ObjectMinCardinality` und `ObjectMaxCardinality` bzw. denen für Data Properties werden in OWL anonyme Elementtypen definiert, die Einschränkungen bezüglich des Auftretens einer Property vorgeben.[208] Wird ein durch solche Kardinalitätsbeschränkungen definierter anonymer Elementtyp als Obertyp C_p innerhalb eines `SubClassOf`(C_c C_p)-Axioms verwendet, so gehen die Kardinalitätsbeschränkungen auf die klassenabhänigen Attribute der Unterklasse C_c über. Abbildung 9.26 zeigt ein Beispiel für die Transformation dieser Art Kardinalitätsbeschränkungen.

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Book ) )
  Declaration( Class( :Person ) )
  Declaration( ObjectProperty( :author ) )
  Declaration( DataProperty( :title ) )

  SubClassOf( :Book ObjectMinCardinality( 1 :author :Person ))
  SubClassOf( :Book ObjectMaxCardinality( 5 :author :Person ))
  SubClassOf( :Book DataExactCardinatliy( 1 :title xsd:string ))
)
```

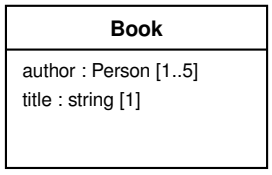


Abbildung 9.26. Beispiel für die Transformation verschiedener Axiome zur Beschränkung der Kardinalität in Kardinalitätsbeschränkungen klassenabhängiger Attribute.

Funktionale Abhängigkeit

OWL erlaubt es, Properties als funktional zu kennzeichnen. Bei den beiden Axiomen handelt es sich jedoch nur um syntaktische Abkürzungen (siehe Tabelle 3.1) für Unterklasse- und Kardinalitäts-Axiome. Daher können sie in Kardinalitätsbeschränkungen 0..1 der entsprechenden Instanz des UML-Elementtyps *Property* transformiert werden. Abbildung 9.27 zeigt ein Beispiel für eine solche Transformation, die zugehörige QVT-Regel ist in Abbildung 9.28 zu sehen.

[208] Bei `DataExactCardinality`, `DataSomeValuesFrom`, `ObjectExactCardinality` sowie `ObjectSomeValuesFrom` handelt es sich lediglich um syntaktische Abkürzungen für die vier anderen Kardinalitäts-CE, siehe auch Tabelle 3.1.

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( ObjectProperty( :mainAuthor ))
  Declaration( DataProperty( :hasTitle ))

  ObjectPropertyDomain( :mainAuthor :Article )
  DataPropertyDomain( :hasTitle :Article )

  FunctionalObjectProperty( :mainAuthor)
  FunctionalDataProperty( :hasTitle )
)
```

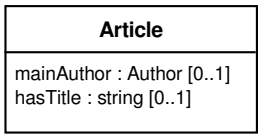


Abbildung 9.27. Beispiel für die Transformation von `FunctionalObjectPropert-` und `Functional-DataProperty`-Axiomen in Kardinalitätsbeschränkungen klassenabhängiger Attribute.

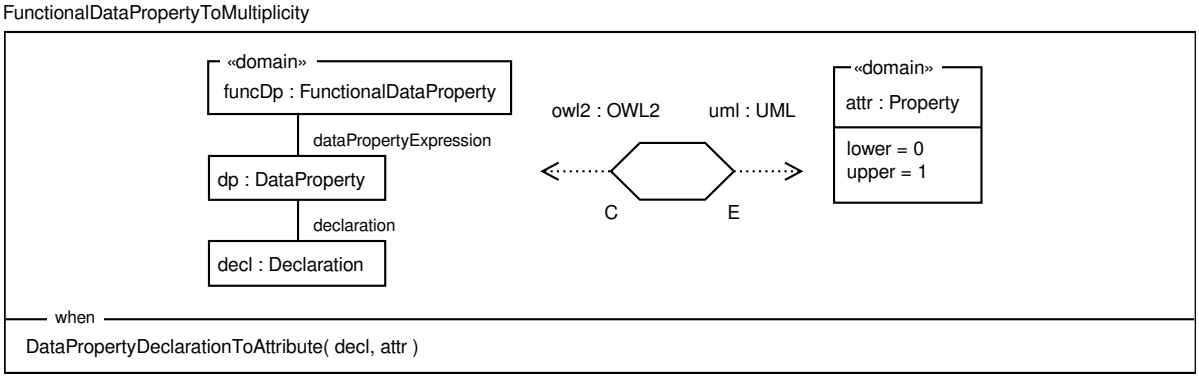


Abbildung 9.28. QVT-Regel zum Tranformieren eines `FunctionalDataProperty`-Axioms in Kardinalitätsbeschränkungen einer `Property`.

Bei dem `InverseFunctionalObjectProperty`-Axiom zum Kennzeichnen einer inversfunktionaler Object Property handelt es sich ebenfalls um syntaktische Abkürzung. Allerdings verhindern in diesem Fall die in Abschnitt 7.3.5 geschilderten Probleme mit `SubClassOf`-Axiomen sowie hinreichenden Bedingungen für eine Klassenzugehörigkeit eine analoge Transformation. Erhalten werden kann die Einschränkung dadurch, dass die OWL-Property auf ein *memberEnd* einer Instanz des UML-Elementtyps *Association* abgebildet und am zweiten *memberEnd* die Kardinalitätsbeschränkung 0..1 vorgenommen wird. Abbildung 9.29 zeigt ein Beispiel für eine Transformation mit einem `InverseFunctionalObjectProperty`-Axiom.

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( ObjectProperty( :isFatherOf ))
  ObjectPropertyDomain( :isFatherOf :Man )
  ObjectPropertyRange( :isFatherOf :Person )

  InverseFunctionalObjectProperty( :isFatherOf )
)
```

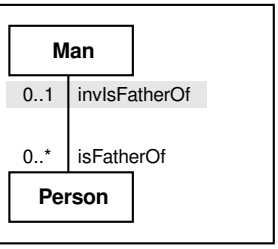


Abbildung 9.29. Beispiel für die Transformation eines `InverseFunctionalObjectProperty`-Axioms in Kardinalitätsbeschränkungen einer Assoziation.

9.4.6 Ausdeutung für XSD

Bei XSD können für Beziehungen sowohl im Fall von Attributen als auch im Fall von Elementen Kardinalitätsbeschränkungen angeben werden. Bei den Attributen ist das Maximalvorkommen stets 1, beim Mindestvorkommen kann zwischen 0 und 1 gewählt werden. Bei Elementen besteht eine freie Wahl der Schranken, wobei es beim Maximalvorkommen neben Zahlenwerten auch möglich ist, `unbounded` anzugeben. Dies bedeutet, dass die betreffende Beziehung unbegrenzt oft auftreten darf. Werden keine expliziten Festlegungen für Kardinalitätsbeschränkungen getroffen, so gilt für Attribute die Kardinalitätsbeschränkung 0..1, für Elemente sind untere und obere Schranke auf 1 gesetzt, d.h. die Beziehung ist obligatorisch.

In diesem Beispiel darf es zwei bis fünf Beziehungen mit dem Namen *N* zu Elementen des Typs *E* geben:

```
1 <sequence>
2    <element
3       name="N"
4       type="E"
5       minOccurs="2"
6       maxOccurs="5" />
7 </sequence>
```

Im XML Schema Component Model befinden sich die Kardinalitätsbeschränkungen in der *Particle*-Instanz, die die *ElementDeclaration*-Instanz mit den beiden am Beziehungstyp beteiligten Elementtypen (Instanzen von *ComplexType*) verbindet. Ein Objektdiagramm für das obige Beispiel ist in Abbildung 9.30 zu sehen.

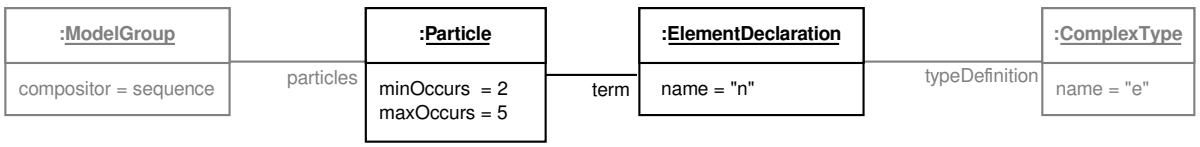


Abbildung 9.30. Objektdiagramm mit Instanzen des XML Schema Component Model für das Beispiel in konkreter XML-Syntax.

9.5 Wertebeschränkungen

Bei einer Wertebeschränkung ist der zweite Teilnehmer eines Beziehungstyps auf exakt einen Wert bzw. ein Objekt festgelegt.

9.5.1 Logische Repräsentation

Eine Wertebeschränkung eines Beziehungstyps kann so interpretiert werden, dass bei einem binärer Beziehungstyp $R(p_1 : E_1, p_2 : E_2)$ der Elementtyp E_2 eine feste Population (→ 7.5 Elementtypen mit fester Population) mit genau einem Element hat $E_2 = \{a\}$.

9.5.2 Repräsentation in UML

In UML kann festgelegt werden, dass ein klassenabhängiges Attribut bei der Instantiierung der Klasse, zu der es gehört, mit einem festen Wert vorbelegt wird. Um auch bei einer dynamischen Nutzung die Änderung dieses Wertes zu verhindern, kann zusätzlich dieser Wert als unveränderlich markiert werden.

Konkrete Syntax

In der grafischen Syntax erfolgt die Angabe des Wertes nach einem Gleichheitszeichen hinter dem Typ des klassenabhängigen Attributs, wie in Abbildung 9.31 dargestellt ist.

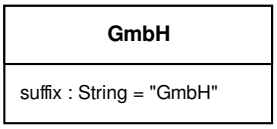


Abbildung 9.31. Beispiel für die Angabe einer Wertebeschränkung in grafischer UML-Syntax.

Abstrakte Syntax

Wie am Anfang dieses Kapitels gesehen, handelt es sich bei klassenabhängigen Attributen um Instanzen von *Property*. Eine *Property* enthält ein klassenabhängiges Attribut *default* mit dem Initialwert. Da – wie in Abbildung 9.32 zu sehen ist – jede *Property* auch ein *StructuralFeature* ist, besitzt es ebenfalls ein klassenabhängiges Attribut *isReadOnly*. Ist dieses auf "wahr" gesetzt, so darf der Wert nicht verändert werden.

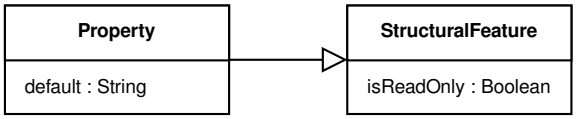


Abbildung 9.32. Ausschnitt aus dem UML-Metamodell, der die für eine Wertebeschränkung relevanten Attribute von *Property* und *StructuralFeature* zeigt.

9.5.3 Repräsentation in OWL

OWL bietet sowohl für Object als auch für Data Properties die Möglichkeit, den zweiten Teilnehmer auf genau einen Wert festzulegen. Eine Wertebeschränkung könnte auch durch Definition eines einelementigen Elementtyps und entsprechendem Kardinalitäts-Axiom realisieren werden. OWL bietet jedoch eine abkürzende Schreibweise, die die Semantik für einen menschlichen Leser schnell deutlich macht.

Konkrete Syntax

Die konkrete Syntax für eine CE, die die Menge der Individuen beschreibt, die zu einem Individuum *i* in Beziehung *R* stehen, sieht folgendermaßen aus:

```
ObjectHasValue( R i )
```

Analog dazu beschreibt folgender Ausdruck die Menge der Individuen, die zu einem Wert *a* in Beziehung *R* stehen:

```
DataHasValue( R a )
```

Abstrakte Syntax

In abstrakter Syntax wird die Festlegung einer Object Property auf ein Individuum durch eine Instanz des Typs *ObjectHasValue* dargestellt, die in Verbindung mit einer Instanz von *ObjectPropertyExpression* und einer *Individual*-Instanz steht. Die Festlegung einer Data Property auf einen Wert wird durch eine Instanz des Typs *DataHasValue* dargestellt, die in Verbindung mit einer *DataPropertyExpression*-Instanz und dem entsprechenden Wert in Form einer *Literal*-Instanz steht. Abbildung 9.33 zeigt den entsprechenden Ausschnitt aus dem OWL-Metamodell.

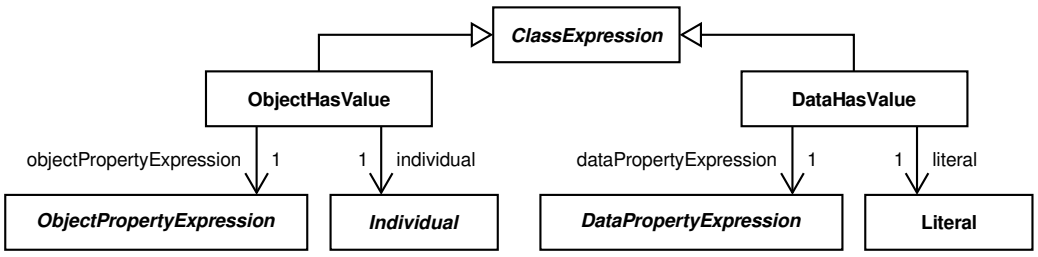


Abbildung 9.33. Ausschnitt aus dem OWL-Metamodell, der die Elementtypen *ObjectHasValue* und *DataHasValue* zeigt.

9.5.4 Transformation UML → OWL

So wie Kardinalitätsbeschränkungen klassenabhängiger Attribute in `SubClassOf`-Axiome der enthaltenen Klasse transformiert werden, wird auch eine Wertebeschränkung in ein `SubClassOf`-Axiom transformiert. Als Obertyp kommt eine Instnaz von *DataHasValue* (eine *DataRange*) zum Einsatz. Dort werden der festgelegte Wert und die Data Property angegeben, die aus dem klassenabhängigen Attribut erzeugt wurde.

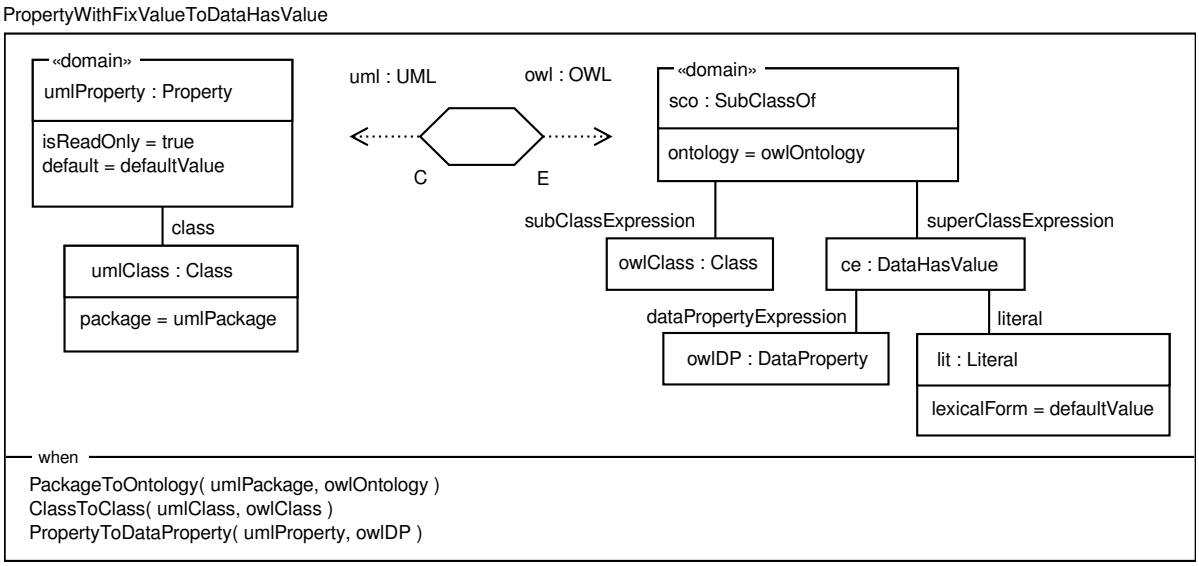


Abbildung 9.34. QVT-Transformationsregel zur Behandlung von klassenabhängigen Attributen mit festem Wert.

Die Transformation lässt sich durch eine zusätzliche Transformationsregel realisieren, die auf solche Instanzen des UML-Elementtyps *Property* angewendet wird, bei denen das klassenabhängige Attribut `isReadOnly = true` und ein Wert für das klassenabhängige Attribut `default` gesetzt ist. Der Zusammenhang zwischen UML- und OWL-Klassen sowie der Instanz des UML-Elementtyps *Property* und der Instanz des OWL-Elementtyps *DataProperty* wird über entsprechende `when`-Regeln sichergestellt. Die Transformationsregel ist in Abbildung 9.34 zu dargestellt.

9.5.5 Transformation OWL → UML

Wird ein mit Hilfe der `DataHasValue`-CE definierter Elementtyp als Obertyp C_p in einem `SubClassOf(`C_c C_p`)`-Axiom verwendet, so wird dieser feste Wert zu einer notwendigen Bedingung für Instanzen der Klasse C_c. Jede Instanz dieser Klasse wird genau den festgelegten Wert besitzen. In einem UML-Modell lässt sich dieser Wert bei einer Instanz des UML-Elementtyps *Property* mit Hilfe des klassenabhängigen Attributs *default* setzen. Ein Beispiel für diesen Fall ist in Abbildung 9.35 dargestellt.

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Book ) )
  Declaration( DataProperty( :title ) )
  Declaration( DataProperty( :printed ) )

  SubClassOf( :Book DataExactCardinatliy( 1 :title xsd:string ))

  SubClassOf( :Book DataHasValue( :printed "true"^^xsd:boolean ))
)
```

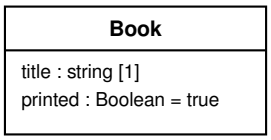


Abbildung 9.35. Beispiel für die Transformation eines Axiomes zur Wertebeschränkung in eine Wertebeschränkung eines klassenabhängigen Attributs.

9.5.6 Ausdeutung für XSD

XSD sieht sowohl für Attribute als auch für Elemente die Möglichkeit vor, einen festen Wert anzugeben. Dazu wird das Attribut `fixed` verwendet:

```
1 <xsd:complexType name="GmbH">
2    <xsd:sequence>
3       <xsd:element name="name" type="xsd:string" />
4       <xsd:element name="suffix" type="xsd:string" fixed="GmbH" />
5    </xsd:sequence>
6 </xsd:complexType>
```

9.6 Teil-Ganzes-Beziehungen

Bei Teil-Ganzes-Beziehungen (auch als Komposition und Aggregation bekannt) handelt es sich um eine spezielle Art von Beziehungstypen, die eine wichtige Rolle bei der Modellierung spielen.[209] Mit ihnen lässt sich zum einen eine bestimmte Semantik ausdrücken, zum anderen können weitere Einschränkungen für die jeweiligen Beziehungstypen auferlegt werden.

Eine Teil-Ganzes-Beziehung ist antisymmetrisch, d.h. wenn T ein Teil von G ist, dann kann G nicht Teil von T sein. Umstritten ist dagegen, ob Teil-Ganzes-Beziehungen transitiv sind.[210]

Da eine Teil-Ganzes-Beziehung ein binärer Beziehungstyp ist, treffen die zuvor gemachten Aussagen und Beobachtungen für allgemeine Beziehungstypen auch für diese Art von Beziehungstyp zu.

9.6.1 Logische Repräsentation

Eine Teil-Ganzes-Beziehung lässt sich als binäre Relation mit den beiden Rollen *part* und *whole* darstellen:

$$IsPartOf(part : Entity_1, whole : Entity_2)$$

[209] Vgl. Olivé: Conceptual Modeling, S. 137.
[210] Vgl. a. a. O., S. 142.

9.6.2 Repräsentation in UML

Die UML kennt zwei verschiedene Arten von Teil-Ganzes-Beziehungen: Aggregation und Komposition. Sie unterscheiden sich sowohl in (grafischer) Syntax als auch in ihrer Semantik.

Eine Aggregation ist antisymmetrisch und transitiv, so dass die verbundenen Objekte einen gerichteten azyklischen Graphen bilden.[211] Es ist möglich, dass ein *Teil* Bestandteil von mehreren *Ganzen* ist.

Eine Komposition ist eine stärkere Form der Aggregation, da hier neben Antisymmetrie und Transitivität zusätzlich ein *Teil* – zu einem Zeitpunkt – nur Bestandteil eines einzigen *Ganzen* sein darf. Weiterhin hängt die Existenz von *Teil* von der Existenz eines *Ganzen* ab, es kann also nicht ohne ein *Ganzes* existieren.

Konkrete Syntax

Wie in Abbildung 9.36 gezeigt, wird eine Aggregation durch eine Linie mit einer transparenten/weißen Raute auf Seite des Ganzen dargestellt, eine Komposition mit einer gefüllten/schwarzen Raute.

Abbildung 9.36. Beispiel für die grafische Syntax einer Aggregation und einer Komposition.

Abstrakte Syntax

Im Metamodell werden für Aggregationen und Kompositionen ebenfalls Instanzen von *Association* und *Property* verwendet, wobei die Eigenschaft *aggregation* gesetzt ist: Im Falle einer Aggregation wird der Wert *AggregationType::shared*, bei einer Komposition der Wert *AggregationType::composite* verwendet. Abbildung 9.37 zeigt ein Beispiel für die abstrakte Syntax von Aggregation und Komposition.

9.6.3 Repräsentation in OWL

OWL kennt keine speziellen Konstrukte zum Kennzeichnen von Teil-Ganzes-Beziehungen.

9.6.4 Transformation UML → OWL

Bei Aggregationen und Kompositionen handelt es sich um spezielle Formen der Assoziation zwischen zwei Klassen. Wie andere Assoziationen zwischen zwei Klassen auch werden diese in Object Properties transformiert. Außerdem werden die oben genannten zusätzlichen Einschränkungen berücksichtigt:

[211] Vgl. Balzert: Softwaretechnik, S. 171.

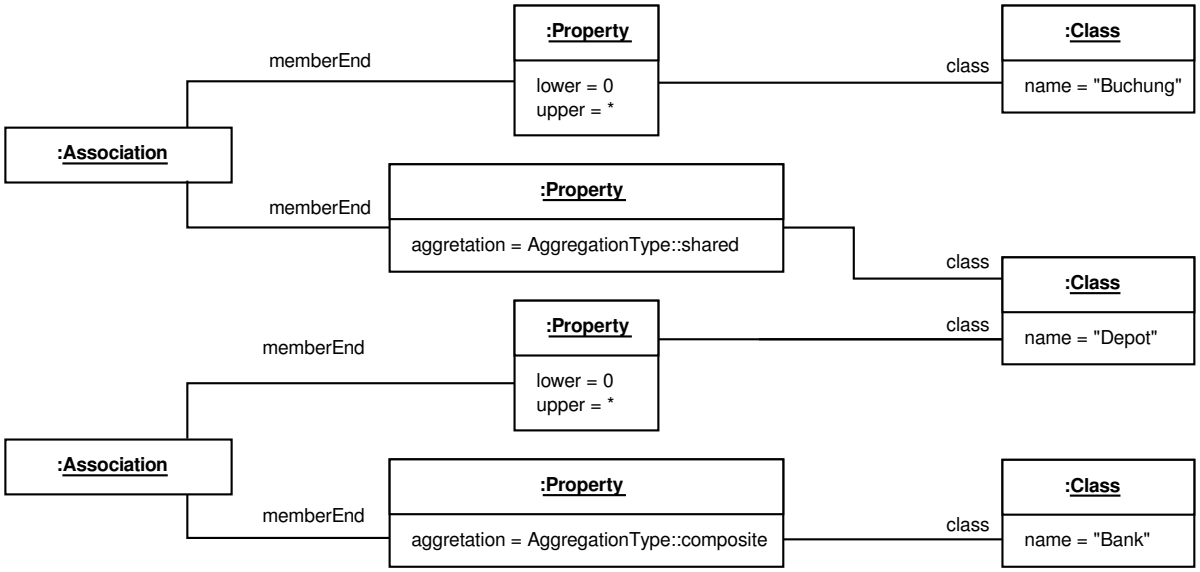


Abbildung 9.37. Objektdiagramm, das die abstrakte Syntax des Beispiels aus Abbildung 9.36 zeigt.

a) Aggregationen sind antisymmetrisch,

b) ein Objekt darf nicht in einer Aggregationsbeziehung zu sich selbst stehen – das wäre in Widerspruch zur Antisymmetrie,

c) ein Objekt darf nicht Teil von mehr als einer Komposition sein,

d) ein Objekt einer Klasse, die Bestandteil einer Komposition ist, darf nicht alleine existieren.

Die Asymmetrie kann durch Hinzufügen eines `AsymmetricObjectProperty`-Axioms für die aus der Assoziation mit Aggregations- oder Kompositionseigenschaft transformierte ObjectProperty erreicht werden.

Die Einschränkung (b) kann nach OWL transformiert werden, indem für die jeweilige Assoziation mit Aggregations- oder Kompositionseigenschaft ein `IrreflexiveObjectProperty`-Axiom der Ontologie hinzugefügt wird. Dieses Axiom verbietet, dass es eine entsprechende Object Property zwischen dem Individuum und sich selbst geben darf.

Einschränkung (c) kann durch ein `FunctionalObjectProperty`- oder ein `InverseFunctionalObjectProperty`-Axiom erreicht werden. Wenn die Assoziation mit Kompositionseigenschaften bidirektional navigierbar ist, macht es keinen Unterschied, welche Art von Axiom verwendet wird. Handelt es sich um eine einseitig navigierbare Assoziation, muss folgende Wahl getroffen werden: Ist die Assoziation von “Teil” zum “Ganzen” navigierbar, so ist ein `FunctionalObjectProperty`-Axiom zu verwenden. Eine Verbindung zwischen einem Individuum der “Teil”-Klasse zu mehr als einem Individuum der “Ganzen”-Klasse würde die Ontologie inkonsistent machen. Ein `InverseFunctionalObjectProperty`-Axiom wird verwendet, wenn die Assoziation vom “Ganzen” zum “Teil” navigierbar ist.

Dadurch, dass bei OWL die OWA verwendet wird, ist eine Einschränkung der Form (d) nicht möglich. Das betreffende Individuum könnte Teil einer Komposition sein, die nicht explizit in der Ontologie aufgeführt ist.

9.6.5 Ausdeutung für XSD

Den baumförmigen Aufbau eines XML-Dokumente lässt sich so deuten, dass es sich bei allen Beziehungen zwischen übergeordnetem und untergeordnetem Elementen um Teil-Ganzes-Beziehungen handelt.

Das Beispiel aus dem UML-Diagramm oben ließe sich z.B. folgendermaßen in XSD schreiben:

```
<complexType name="Bank">
   <sequence>
      <element name="depot" minOccurs="1" maxOccurs="unbounded" />
   </sequence>
</complexType>

<complexType name="Depot">
   <sequence>
      <element name="buchung" minOccurs="0" maxOccurs="unbounded" />
   </sequence>
</complexType>

<complexType name="Buchung">
   <sequence />
</complexType>
```

Allerdings ist diese Semantik nicht in allen Fällen intendiert, sondern ergibt sich aus dem praktischen Grund, dass sich ein XML-Dokument bequem baumförmig aufschreiben lässt. Ansonsten gibt es bei XSD kein besonderes Kennzeichen dafür, dass es sich um eine Teil-Ganzes-Beziehung handelt.

9.7 Transitivität

Bei einem binären, rekursiven Beziehungstypen lässt sich angeben, dass die zugehörigen Beziehungen *transitiv* sind. Ein Beispiel dafür ist die Beziehungen "Nachkomme" zwischen Personen. Ist jemand der Nachkomme einer Person, so sind auch wiederum alle seine Nachkommen Nachkommen.

9.7.1 Logische Repräsentation

Ein Beziehungstyp R ist transitiv, wenn gilt:

$$R(a,b) \wedge R(b,c) \rightarrow R(a,c)$$

9.7.2 Repräsentation in UML

Graphisch lässt sich nicht darstellen, dass ein Beziehungstyp transitiv ist. Dies ist nur durch die Angabe einer OCL-Invariante möglich. Ist der Beziehungstyp $R(p_1 : E, p_2 : E)$ transitiv, so lässt sich dies mit folgender Invariante ausdrücken:

```
context E inv RIsTransitive:
        p2 → includesAll(p2.p2)
```

9.7.3 Repräsentation in OWL

In OWL lässt sich mit Hilfe eines Axioms angeben, dass eine Object Property transitiv ist.

Konkrete Syntax

Mit folgendem Axiom wird definiert, dass es sich bei *op* um eine transitive Object Property handelt:

```
TransitiveObjectProperty( op )
```

Abstrakte Syntax

Eine Instanz von *TransitiveObjectProperty* ist über eine Assoziation mit der als transitiv zu definierenden Instanz von *ObjectPropertyExpression* verbunden, wie in Abbildung 9.38 gezeigt.

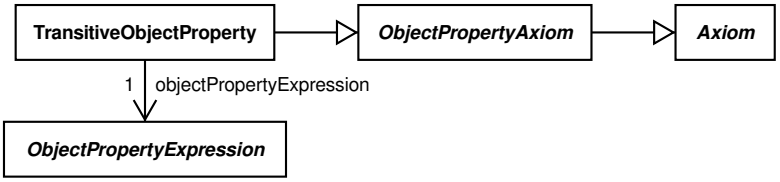


Abbildung 9.38. Der *TransitiveObjectProperty* betreffende Auszug aus dem OWL-Metamodell.

9.7.4 Transformation UML → OWL

Werden zur Angabe der Eigenschaften von Beziehungstypen Stereotypen verwendet, wie in Abschnitt 6.10 beschrieben, so wird für eine mit dem Stereotyp «transitive» gekennzeichnete Instanz des UML-Elementtyps *Property* ein entsprechendes `TransitiveObjectProperty`-Axiom erzeugt. Der Aufruf der Regel zum Erzeugen des Axioms kann in der `where`-Klausel einer Regel für die Behandlung des UML-Elementtyps *Property* erfolgen.

9.7.5 Transformation OWL → UML

Existiert für eine Object Property ein `TransitiveObjectProperty`-Axiom, so lässt sich die transformierte Instanz des UML-Elementtyps *Property* mit einem Stereotyp «transitive» versehen – sofern Stereotypen zur Kennzeichnung verwendet werden.

9.7.6 Ausdeutung für XSD

Da bei XSD alle Beziehungen explizit angegeben werden, existiert keine Möglichkeit, einen Beziehungstypen als transitiv zu kennzeichnen.

9.8 Symmetrie

Bei einem binären, rekursiven Beziehungstypen lässt sich angeben, dass die zugehörigen Beziehungen *symmetrisch* sind. Ein Beispiel dafür ist die natürliche "ist Nachbar von"-Beziehung. Wenn A der Nachbar von B ist, dann ist ebenfalls B der Nachbar von A.

9.8.1 Logische Repräsentation

Ein Beziehungstyp R ist symmetrisch, wenn gilt:

$$R(a, b) \rightarrow R(b, a)$$

9.8.2 Repräsentation in UML

Graphisch lässt sich nicht darstellen, dass ein Beziehungstyp symmetrisch ist. Dies ist nur durch die Angabe einer OCL-Invariante möglich. Ist der Beziehungstyp $R(p_1 : E, p_2 : E)$ symmetrisch, so lässt sich dies mit folgender Invariante ausdrücken:

```
context E inv RIsSymmetric:
        p1.p1 → includes(self)
```

9.8.3 Repräsentation in OWL

In OWL lässt sich mit Hilfe eines Axioms angeben, dass eine Object Property symmetrisch ist.

Konkrete Syntax

Mit folgendem Axiom wird definiert, dass es sich bei *op* um eine symmetrische Object Property handelt:

```
SymmetricObjectProperty( op )
```

Abstrakte Syntax

Eine Instanz von *SymmetricObjectProperty* ist über eine Assoziation mit der als symmetrisch zu definierenden Instanz von *ObjectPropertyExpression* verbunden, wie in Abbildung 9.39 gezeigt.

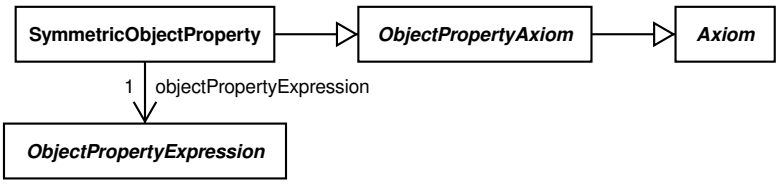


Abbildung 9.39. Der *SymmetricObjectProperty* betreffende Auszug aus dem OWL-Metamodell.

9.8.4 Transformation UML → OWL

Werden zur Angabe der Eigenschaften von Beziehungstypen Stereotypen verwendet, wie in Abschnitt 6.10 beschrieben, so wird für eine mit dem Stereotyp «symmetric» gekennzeichnete Instanz von *Property* ein entsprechendes `SymmetricObjectProperty`-Axiom erzeugt. Der Aufruf der Regel zum Erzeugen des Axioms kann in der `where`-Klausel einer Regel für *Property* erfolgen.

9.8.5 Transformation OWL → UML

Bei der Transformation symmetrischer Object Properties wird anders als im allgemeinen Fall (→ 9 Beziehungstypen) stets eine Instanz des UML-Elementtyps *Association* erzeugt, um so eine beidseitige Navigierbarkeit herzustellen.

Zusätzlich kann – sofern die in Abschnitt 6.10 beschriebenen Stereotypen zur Angabe von Eigenschaften für Beziehungstypen verwendet werden – für eine Object Property, zu der ein `SymmetricObjectProperty`-Axiom existiert, die daraus transformierte Instanz des UML-Elementtyps *Property* mit einem Stereotyp «symmetric» versehen werden.

9.8.6 Ausdeutung für XSD

Da bei XSD alle Beziehungen explizit angegeben werden, existiert keine Möglichkeit, einen Beziehungstypen als symmetrisch zu kennzeichnen.

9.9 Inverse

In konzeptuellen Modellen taucht oft die Situation auf, dass zwei Elementtypen E_1 und E_2 Teilnehmer in einem Beziehungstyp $R(E_1, E_2)$ sind, aber die Freiheit bestehen soll, Instanzen beider Elementtypen als ersten oder zweiten Teilnehmer zu verwenden. Um dies zu ermöglichen, lässt sich ein inverser Beziehungstyp $R_{inv}(E_1, E_2)$ definieren.

Als Beispiel soll ein Buch dienen, das Kapitel beinhaltet. Zu den Elementtypen *Buch* und *Kapitel* gibt es einen Beziehungstypen $beinhaltet(Buch, Kapitel)$. Instanzen des Typs *Buch* müssen dabei stets als der erste Teilnehmer einer solchen Beziehung auftreten. Wird ein zu *beinhaltet* inversen Beziehungstyp $istEnthaltenIn(Kapitel, Buch)$ definiert, so lassen sich zu *beinhaltet* äquivalente Aussagen mit Instanzen des Typs *Kapitel* als erstem Teilnehmer machen.

9.9.1 Logische Repräsentation

Ist ein Beziehungstyp R_{inv} als invers zum Beziehungstyp R definiert, so gilt:

$$R(a, b) \rightarrow R_{inv}(b, a)$$

9.9.2 Repräsentation in UML

UML sieht zwar keine Möglichkeit vor, explizit zwei beliebige Assoziationen als invers zueinander zu markieren, jedoch lassen sich die beiden Enden einer binären, bidirektionalen Assoziation als zwei inverse Beziehungen verstehen.

9.9.3 Repräsentation in OWL

Die Definition von inversen Beziehungstypen ist nur bei Object Properties möglich, da ein Wert (eine Instanz eines Datentyps) selbst keine Properties besitzen kann. OWL kennt zwei Möglichkeiten, inverse Beziehungstypen zu verwenden:

1. Zwei zuvor definierte Object Properties werden mit Hilfe des `InverseObjectProperties`-Axioms als invers zueinander gekennzeichnet.
2. Mit Hilfe der Object Property Expression `ObjectInverseOf` kann die Inverse einer Object Property direkt verwendet werden, ohne ihr einen Namen zuzuweisen.

Zu beachten ist dabei, dass für zwei Individuen a und b die inverse Object Property $op_{inv}(b, a)$ automatisch Bestandteil der Ontologie ist, sobald $op(a, b)$ in der Ontologie enthalten ist.

Konkrete Syntax

Um die zwei zuvor definierten, benannten Object Properties op_1 und op_2 als invers zueinander zu kennzeichnen, wird folgendes Axiom verwendet:

```
InverseObjectProperties( op1, op2)
```

Die Inverse einer zuvor definierten, benannten Object Property op wird mit folgender Schreibweise verwendet:

```
ObjectInverseOf( op )
```

Abstrakte Syntax

Werden zwei Object Properties als invers gekennzeichnet, so wird dafür eine Instanz von *InverseObjectProperties* verwendet. Diese ist mit den zwei *ObjectPropertyExpression*-Instanzen verbunden, die invers zueinander sein sollen. Der entsprechende Ausschnitt aus dem OWL-Metamodell ist in Abbildung 9.40 (oben) zu sehen.

Die Inverse einer zuvor definierten, benannten Object Property wird durch eine Instanz von *InverseObjectProperty* repräsentiert. Bei dieser handelt es sich um eine *ObjectPropertyExpression*, so dass sie wie eine gewöhnliche Object Property verwendet werden kann. Über eine Assoziation ist sie mit der zu ihr inversen Instanz von *ObjectProperty* verbunden. Der entsprechende Ausschnitt aus dem OWL-Metamodell ist in Abbildung 9.40 (unten) zu sehen.

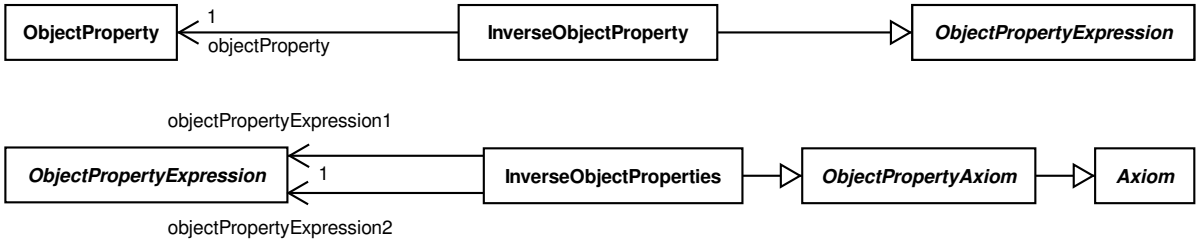


Abbildung 9.40. Ausschnitte aus dem OWL-Metamodell, welche die für *InverseObjectProperty* (oben) und *InverseObjectProperties* (unten) relevanten Teile zeigen.

9.9.4 Transformation UML → OWL

Da bidirektionale Assoziationen bei der Transformation UML → OWL in zwei – zunächst unabhängige – Object Properties transformiert werden, sie jedoch gleichwertig zu zwei gerichteten, zueinander inversen Assoziation sind (vergleiche Abschnitt 9.1.4), wird für eine solche bidirektionale Assoziation eine Instanz des OWL-Elementtyps *InverseObjectProperties* erzeugt und Beziehungen zu den entsprechenden Instanzen von *ObjectProperty* hergestellt. Die QVT-Transformationsregel ist in Abbildung 9.41 zu sehen.

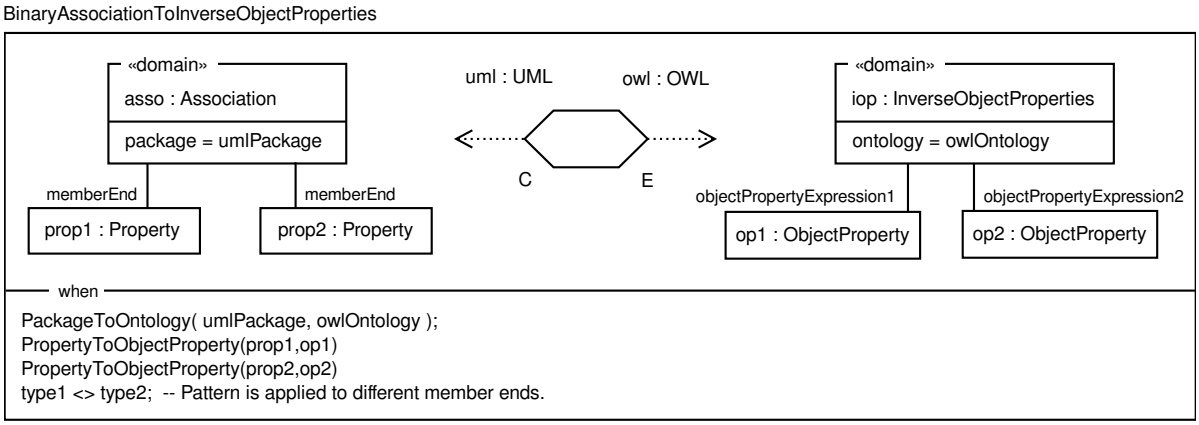


Abbildung 9.41. QVT-Regel zum Transformieren zweier Instanzen des UML-Elementtyps *Property*, die gemeinsam in einer binären, bidirektionalen *Association* auftreten.

9.9.5 Transformation OWL → UML

Für Object Property-Verbindungen, zu denen innerhalb der Ontologie eine inverse Verbindung existiert, werden andere Transformationsregeln als im allgemeinen Fall eingesetzt. Wie im Abschnitt zur konkreten Syntax gesehen, können inverse Verbindungen auf mehrere Arten spezifiziert sein:

- durch explizite Angabe mit Hilfe des `InverseObjectProperties`-Axioms,
- durch Verwendung einer anonymen inversen `InverseObjectProperty` oder
- durch Kennzeichnung einer Object Property als invers-funktional.

In allen Fälle kommen keine klassenabhängigen Attribute zum Einsatz, da ansonsten der Zusammenhang zwischen beiden Verbindungen nicht deutlich wird. Stattdessen wird eine Instanz des UML-Elementtyps *Association* als Transformationsziel verwendet. Die beiden als *members ends* auftretenden Instanzen des UML-Elementtyps *Property* werden wechselseitig als Wert des *opposite*-Attribut gesetzt. In Abbildung 9.42 wird dies für den Fall von zwei benannten, explizit als invers gekennzeichneten Object Properties gezeigt. Die zugehörige QVT-Regel ist in Abbildung 9.43 zu sehen.

```
Prefix( :=<urn:Package#> )
Ontology( <urn:Package>

  Declaration( ObjectProperty( :hasAuthor ))
  ObjectPropertyDomain( :hasAuthor :Book )
  ObjectPropertyRange( :hasAuthor :Author )

  Declaration( ObjectProperty( :wrote ))

  InverseObjectProperties( :hasAuthor :wrote )
)
```

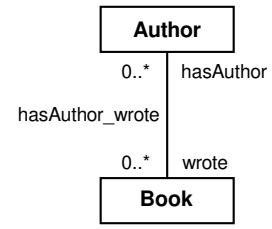


Abbildung 9.42. Beispiel für die Transformation einer inversen Object Property in eine Instanz UML-Elementtyps *Association*.

ObjectPropertyWithInverseToAttribute

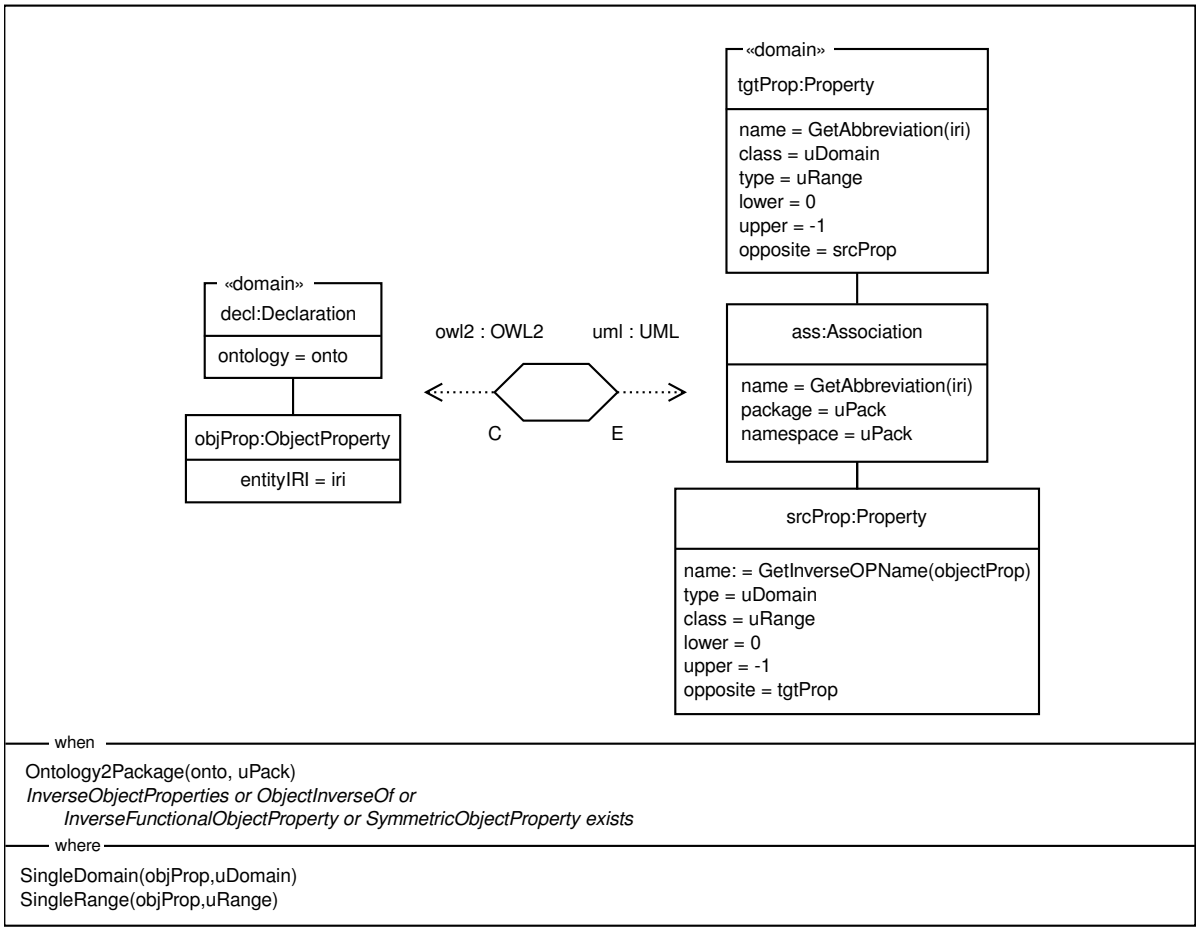


Abbildung 9.43. QVT-Regel zur Transformation einer Instanz des OWL-Elementtyps *ObjectProperty* mit Inverser in eine Instanz des UML-Elementtyps *Association*.

9.9.6 Ausdeutung für XSD

XSD selbst kennt keine Markierung für inverse Beziehungstypen. Bei der Definition von GML Application Schemas mit Hilfe von XSD ist es jedoch vorgesehen, inverse Beziehungstypen zu kennzeichnen.

```
1 <element name="owner" type="ex:PersonPropertyType" maxOccurs="unbounded">
2    <annotation>
3       <appinfo><gml:reverseProperty>ex:owns</gml:reverseProperty></appinfo>
4    </annotation>
5 </element>
```

Listing 9.1. Ausschnitt aus einem XML-Schema mit GML-eigenen Element `gml:reverseProperty`.[212]

Sind Beziehungen ausschließlich über Kindelemente spezifiziert (und nicht über Referenzen), so könnten die Beziehung zum Kindelement und die Beziehung zum Elternelement im XML-Dokument als invers zueinander ansehen. Dies ist jedoch keine Eigenschaft von XSD.

[212] OGC: GML Standard 3.2.1, S. 354.

Kapitel 10

Beschränkungen

Einige sehr häufige Beschränkungen wurden bereits in vorherigen Abschnitten betrachtet, wie z.B. die Kardinalitätsbeschränkungen (→ 9.4 Kardinalitätsbeschränkungen) oder die Überschneidungsfreiheit bei Generalisierungen (→ 7.6 Generalisierung). Dieses Kapitel widmet sich weiteren Beschränkungen für Elementtypen und Beziehungstypen, nämlich Schlüsseln für Elementtypen, Disjunktion von Element- und Beziehungstypen, sowie "bedingten Beziehungstypen", bei denen eine Kombination von Element- und Beziehungstypen betrachtet wird.

10.1 Schlüssel

Mit Hilfe von Schlüssel-Beschränkungen (engl. key constraints) lässt sich sicherstellen, dass es keine zwei verschiedenen Instanzen eines Elementtyps gibt, bei denen alle im Schlüssel angegebenen Beziehungen einen identischen Wert aufweisen bzw. auf dasselbe Objekt zeigen. Es gibt einfache Schlüssel, die auf lediglich einer Beziehung basieren, sowie zusammengesetzte Schlüssel, die auf mehreren Beziehungen basieren.

Abbildung 10.1. UML-Diagramm als Beispiel, für welche Sachverhalte Schlüssel eingesetzt werden können.[213]

Abbildung 10.1 zeigt ein Beispiel, wie Schlüssel eingesetzt werden können:

- Für den auf der rechten Seite abgebildeten Elementtyp *Country* könnte ein einfacher Schlüssel für den Beziehungstyp *name* definiert werden. So wäre sichergestellt, dass es keine zwei verschiedenen Instanzen von *Country* gibt, die denselben Wert für *name* haben, d.h. es gibt keine zwei Staaten mit identischen Namen.

- Für den auf der linken Seite abgebildeten Elementtyp *Town* könnte eine zusammengesetzter Schlüssel für die Beziehungstypen *longitude* und *latitude* definiert werden. Damit

[213] Vereinfacht nach Olivé: Conceptual Modeling, S. 197.

wäre sichergestellt, dass es nur maximal eine Instanz von *Town* geben kann, bei der *longitude* und *latitude* denselben Wert hat, d.h. es gibt maximal eine Stadt, die sich an einer geographischen Koordinate befindet.

10.1.1 Logische Repräsentation

Sind E, E_i (für $i = 1 \dots n$) Beziehungstypen und sind $R_i(E, E_i)$ (für $i = 1 \dots n$) Beziehungstypen (wobei nicht ausgeschlossen ist, dass $E = E_i$ ist), so ist ein Schlüssel durch folgende Implikation definiert:

$$\forall x, y, a_1, \dots a_n : E(x) \wedge E(y) \wedge R_1(x, a_1) \wedge R_1(y, a_1) \wedge \dots \wedge R_n(x, a_n) \wedge R_n(y, a_n) \rightarrow x = y$$

Bei einem einfachen Schlüssel spielt nur ein Beziehungstyp eine Rolle und die Implikation reduziert sich auf:

$$\forall x, y, a : E(x) \wedge E(y) \wedge R(x, a) \wedge R(y, a) \rightarrow x = y$$

10.1.2 Repräsentation in UML

UML bietet die Möglichkeit, für einen Elementtyp einen einzigen Schlüssel zu definieren. Dazu werden diejenigen klassenabhängigen Attribute (Instanzen von *Property*) markiert, die Bestandteil dieses Schlüssels sind. Mit Hilfe der so markierten Attribute kann eine Instanz des Elementtyps identifiziert werden.[214]

Konkrete Syntax

Ist für einen Elementtyp ein Schlüssel definiert, so wird hinter die klassenabhängigen Attribute, die Bestandteil dieses Schlüssel sind, das Schlüsselwort `{ID}` geschrieben. Abbildung 10.2 zeigt ein Beispiel dafür.

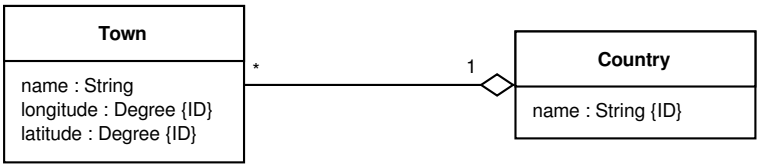


Abbildung 10.2. UML-Diagramm mit Schlüssel-Definitionen.

[214] OMG: UML Infrastructure, S. 98: *"[...] indicates this property can be used to uniquely identify an instance of the containing Class."*

Abstrakte Syntax

Die Angabe, ob ein klassenabhängiges Attribut Bestandteil eines Schlüssels ist, erfolgt mit Hilfe des klassenabhängigen Attributs *isID* der Meta-Klasse *Property*. Abbildung 10.3 zeigt dies am Beispiel der Klasse `Town` aus Abbildung 10.2.

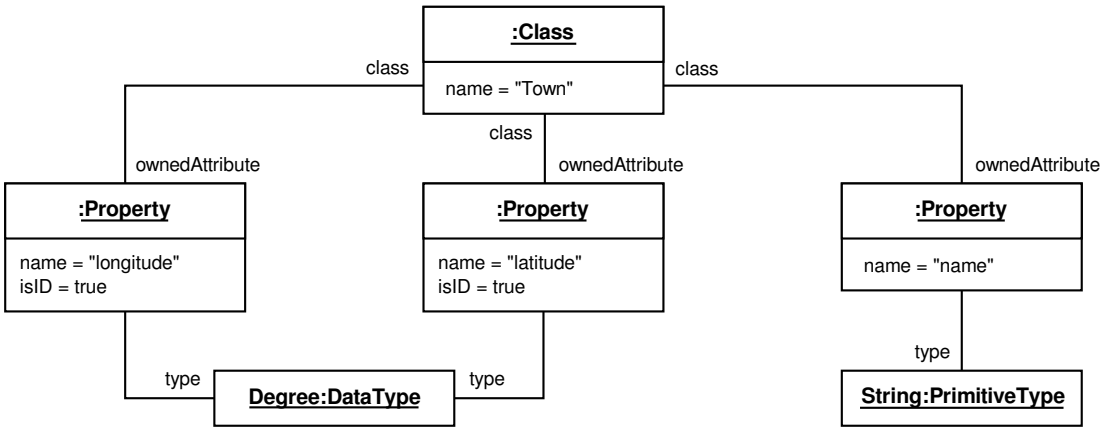


Abbildung 10.3. Abstrakte Syntax der Klasse `Town` aus Abbildung 10.2.

10.1.3 Repräsentation in OWL

OWL bietet ein Axiom zum Definieren von zusammengesetzten Schlüsseln an. Ein solcher Schlüssel kann nicht nur für benannte Elementtypen, sondern für jede CE definiert werden. Die zu berücksichtigen Beziehungstypen werden in zwei Mengen unterteilt, nämlich in Object Properties und Data Properties.

Mit den Axiomen `FunctionalObjectProperty` und `FunctionalDataProperty` bietet OWL noch eine weitere Möglichkeit, besonders starke einfache Schlüssel zu definieren. Eine Identität wird dabei unabhängig vom Elementtyp des Objekts alleine auf Basis der Beziehung definiert.

Konkrete Syntax

Die beiden eingangs beschriebenen Schlüssel lassen sich mit folgenden beiden Axiomen in OWL ausdrücken:

```
HasKey ( :Country () (:name) )
HasKey ( :Town () (:longitude :latitude) )
```

Die leeren Klammern in beiden Axiomen würde die IRIs von Object Properties aufnehmen, sollten solche in der Schlüssel-Beschränkung auftreten – im angegeben Beispiel ist dies nicht der Fall.

Abstrakte Syntax

In abstrakter Syntax wird die Definition eines Schlüssels durch eine Instanz von *HasKey* repräsentiert. Diese steht in Beziehung mit einer Instanz von *ClassExpression*, als dem Elementtyp, für den der Schlüssel definiert wird. Ebenso gibt es Beziehungen zu den Instanzen der Beziehungstypen, die im Schlüssel verwendet werden. Diese sind unterteilt in Object Properties und Data Properties. Der entsprechende Teil des Metamodelles ist in Abbildung 10.4 zu sehen.

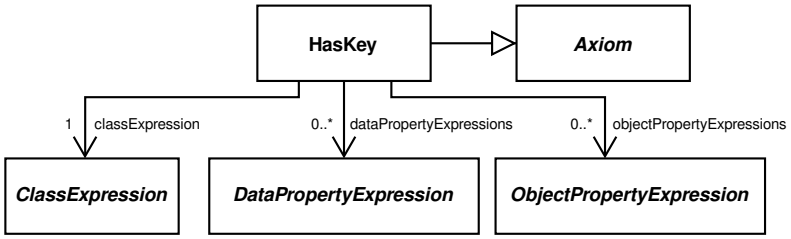


Abbildung 10.4. Der *HasKey* betreffende Auszug aus dem OWL-Metamodell.

Abbildung 10.5 zeigt die Instanzen der abstrakten Syntax für das obige Beispiel des zusammengesetzten Schlüssels für den Elementtyp *Town*.

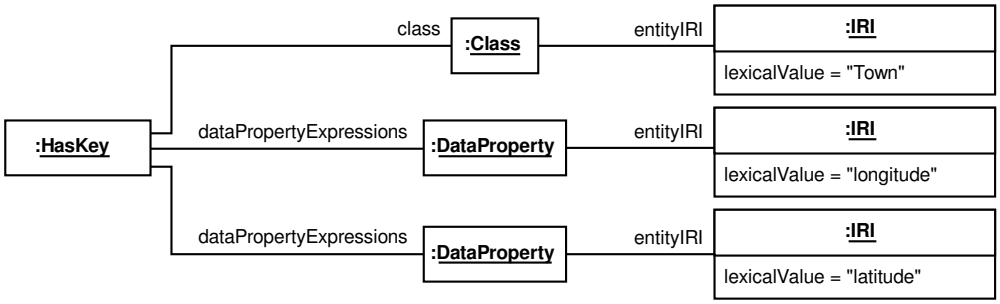


Abbildung 10.5. Objektdiagramm, das ein Beispiel der zu einem zusammengesetzten Schlüssel gehörenden Elemente zeigt.

10.1.4 Transformation UML → OWL

Für jede Klasse, die klassenabhängige Attribute hat (d.h. die mit Instanzen des UML-Elementtyps *Property* mit Rollenname *ownedAttribute* in Beziehung steht), die mit `isID=true` gekennzeichnet sind, wird eine entsprechende Instanz des OWL-Elementtyps `HasKey` der Ontologie hinzugefügt.

Auch bei der Transformation UML → OWL zusammengesetzter Datentypen (siehe Abschnitt 8.3.3) spielen `HasKey`-Axiome eine Rolle, da mit ihnen sichergestellt wird, dass es nur genau eine Instanz einer aus einem zusammengesetzten Datentyp transformierten OWL-Klasse gibt.

10.1.5 Transformation OWL → UML

Die Information, dass die in `HasKey`-Axiomen auftretenden Properties einen Schlüssel bilden, lässt sich dadurch transformieren, dass bei den aus diesen Properties generierten klassenabhängigen Attributen (Instanzen des UML-Elementtyps *Property*) der Wert des klassenabhängigen Attributs *isID* gesetzt wird. Abbildung 10.6 zeigt die entsprechende QVT-Transformationsregel. Die Transformation funktionaler Object und Data Properties wird in Abschnitt 9.4.5 behandelt.

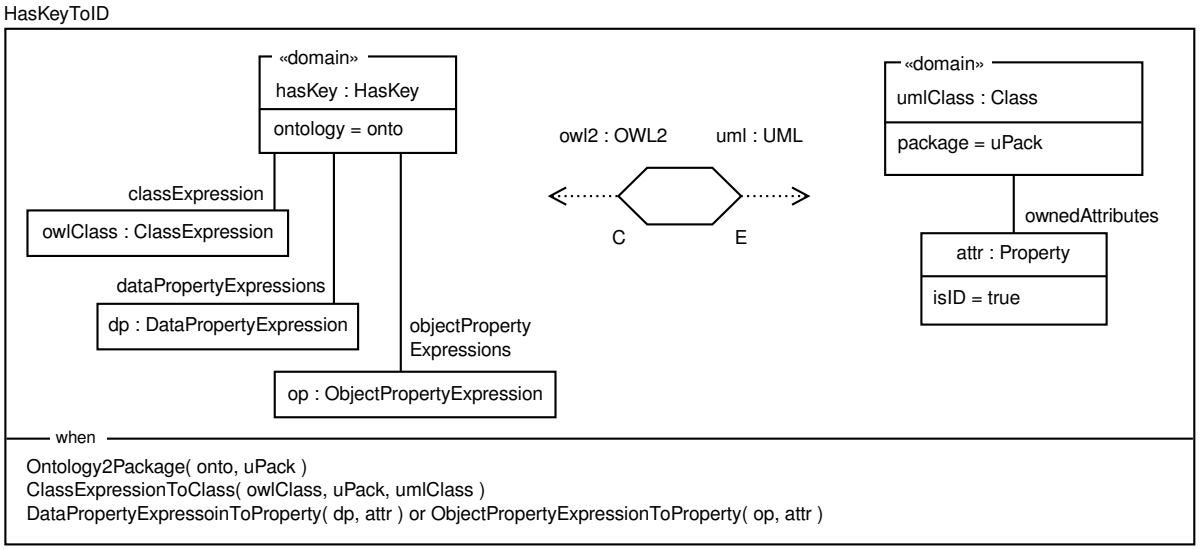


Abbildung 10.6. QVT-Regel zur Transformation von `HasKey`-Axiomen.

10.1.6 Ausdeutung für XSD

Bei XSD gibt es zwei Möglichkeiten, Schlüssel zu definieren. Die erste Möglichkeit eignet sich dazu, einem Element – unabhängig von seinem Elementtyp – eine eindeutige Kennzeichnung zu geben (einfacher Schlüssel). Dazu wird der Datentyp `xsd:ID` verwendet. Dieser Datentyp ist so definiert, dass jeder Wert des Datentyps nur genau einmal in einem XML-Dokument vorkommen darf. Wird also der Datentyp `xsd:ID` als zweiter Teilnehmer in einem Beziehungstyp verwendet, so ist sichergestellt, dass es nur genau eine Beziehung pro Wert gibt.

```
1 <complexType>
2    <sequence>
3       <element name="eindeutigeKennung" type="xsd:ID" />
4    </sequence>
5 </complexType>
```

Die zweite Möglichkeit erlaubt die Definition einer Art zusammengesetzter Schlüssel. Dafür werden *identity constraints* (Instanzen von *Identity-constraintDefintion*) verwendet. Es handelt sich dabei allerdings nur bedingt um echte Schlüssel, da sie nicht für einen Elementtyp, sondern für den Namen eines XML-Elements definiert werden. Wenn jedoch auf den Zusammenhang von Elementnamen und Elementtyp geachtet wird, können *identity constraints* als zusammengesetzte Schlüssel verwendet werden, wie hier für das Beispiel aus Abbildung 10.1 zu sehen ist:

```
<element name="town" type="tns:Town" />
<complexType name="Town">
   <sequence>
      <element name="name" type="xsd:string" />
      <element name="longitude" type="xsd:decimal" />
      <element name="latitude" type="xsd:decimal" />
   </sequence>
</complexType>

<key name="townPosition">
   <selector xpath=".//town"/>
   <field xpath="longitude"/>
   <field xpath="latitude"/>
</key>
```

Mit Hilfe der `selector`-Angabe werden die Elemente (bzw. die Elementtypen, da hier eine eindeutige Beziehung zwischen Elementtypen und Elementnamen besteht) ausgewählt, für die der Schlüssel gelten soll. Die `field`-Angaben zählen die zu berücksichtigenden Elemente (bzw. die von ihnen repräsentierten Beziehungstypen) auf.

10.2 Disjunktion

Manchmal ist es – gerade bei Sprachen ohne UNA – notwendig, zu markieren, dass zwei Modellelemente verschieden sind. Sind zwei Elementtypen als disjunkt markiert, so besitzen sie keine überlappende Population.

10.2.1 Logische Repräsentation

Sind zwei Elementtypen E_1 und E_2 als disjunkt definiert, so gilt für ein Objekt e:

$$E_1(e) \rightarrow \neg E_2(e)$$

10.2.2 Repräsentation in UML

Laut Standard sind in UML Elementtypen, die in Verbindung mit einem *GeneralizationSet* auftreten, disjunkt, sofern nichts anderes angegeben ist.[215] Eine Möglichkeit, Klassen außerhalb

[215] OMG: UML Superstructure, S. 79: "*default is {incomplete, disjoint}*"

eines *GeneralizationSet* als disjunkt zu markieren, gibt es nicht.[216]

Allerdings wird oft – abweichend von den eben genannten Festlegungen im Standard – mit der Annahme gearbeitet, dass alle Klassen, die nicht in einer Vererbungsbeziehung miteinander stehen, paarweise disjunkt sind: "*Here, we follow a typical assumption in UML class diagrams, namely hat all classes not in the same hierarchy are a priori disjoint.*"[217] und "*all classes not in the same hierarchy are a priori disjoint*"[218].

Konkrete Syntax

Um – zusätzlich zur Vorgabe im Standard – deutlich zu machen, dass die Unter(beziehungs)-typen in einer Instanz eines *GeneralizationSet* disjunkt sind, wird das Schlüsselwort `{disjoint}` in die Nähe des gemeinsamen Teils des Baumes oder der Pfeilspitze geschrieben.

Die Abbildung 7.19 zeigt ein Beispiel einer Instanz eines *GeneralizationSet*, bei dem die Untertypen explizit als disjunkt gekennzeichnet sind.

Abstrakte Syntax

Die Information, dass es sich um disjunkte Typen handelt, taucht in abstrakter Syntax in Form eines klassenabhängigen Attributs *isDisjoint* des Elementtyps *GeneralizationSet* auf, siehe auch Abbildung 7.20.[219]

10.2.3 Repräsentation in OWL

OWL kennt zwei Axiome zum Kennzeichnen, dass Elementtypen disjunkt sind, sowie jeweils ein Axiom zur Kennzeichnung, dass Object Properties bzw. Data Properties disjunkt sind. Mit jedem dieser Axiome lässt sich nicht nur eine Aussage über je zwei Typen treffen, sondern es ist auch möglich, mit Mengen von mehr als zwei Elementen zu arbeiten. Die Bedeutung ist dann die, dass die aufgeführten Typen paarweise disjunkt sind.

[216] OMG: Ontology Definition Metamodel, S. 25: "*UML disjointedness requires disjoint classes to have a common super-type, [...]*"

[217] Cadoli, M. et al.: Finite Model Reasoning on UML Class Diagrams Via Constraint Programming, in: Basili, R./Pazienza, M. T. (Hrsg.): AI*IA '07: Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, Berlin/Heidelberg 2007, S. 41.

[218] Berardi, D./Calvanese, D./Giacomo, G. de: Reasoning on UML Class Diagrams is EXPTIME-hard, in: Calvanese, D./De Giacomo, G./Franconi, E. (Hrsg.): Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rom 2003 ⟨URL: `http://ceur-ws.org/Vol-81/berardi-1.pdf`⟩.

[219] Der UML-Standard OMG: UML Superstructure enthält an dieser Stelle widersprüchliche Definitionen: Während auf Seite 79 als Vorgabe "*default is {incomplete, disjoint}*" gemacht wird, ist auf Seite 78 der Vorgabewert für *isDisjoint* mit `false` angegeben.

Konkrete Syntax

Mit Hilfe des `DisjointClasses`-Axiom lässt sich angegeben, dass Elementtypen disjunkt sind:

```
DisjointClasses( C1 ...Cn )
```

Auch im `DisjointUnion`-Axiom, das eine syntaktische Abkürzung ist (siehe Tabelle 3.1), ist ein `DisjointClasses`-Axiom enthalten, so dass es auch zur Kennzeichnung disjunkter Elementtypen verwendet werden kann:

```
DisjointUnion( C C1 ...Cn )
```

Sollen Object Properties und Data Properties als disjunkt definiert werden, so stehen dafür die Axiome `DisjointObjectProperties` und `DisjointDataProperties` zur Verfügung:

```
DisjointObjectProperties( op1 ...opn )
DisjointDataProperties( dp1 ...dpn )
```

Abstrakte Syntax

Abbildung 10.7 gibt einen Überblick über die zur Kennzeichnung von Disjunktionen vorhandenen OWL-Elementtypen. Es handelt sich bei allen vier Typen um Untertypen von *Axiom*.

Sowohl Instanzen von *DisjointClasses* als auch *DisjointUnion* haben mindestens zwei Beziehungen zu Instanzen von *ClassExpression*. Eine Instanz von *DisjointUnion* muss darüber hinaus noch auf eine *Class*-Instanz verweisen, die die Vereinigung der *ClassExpression*-Instanzen bildet.

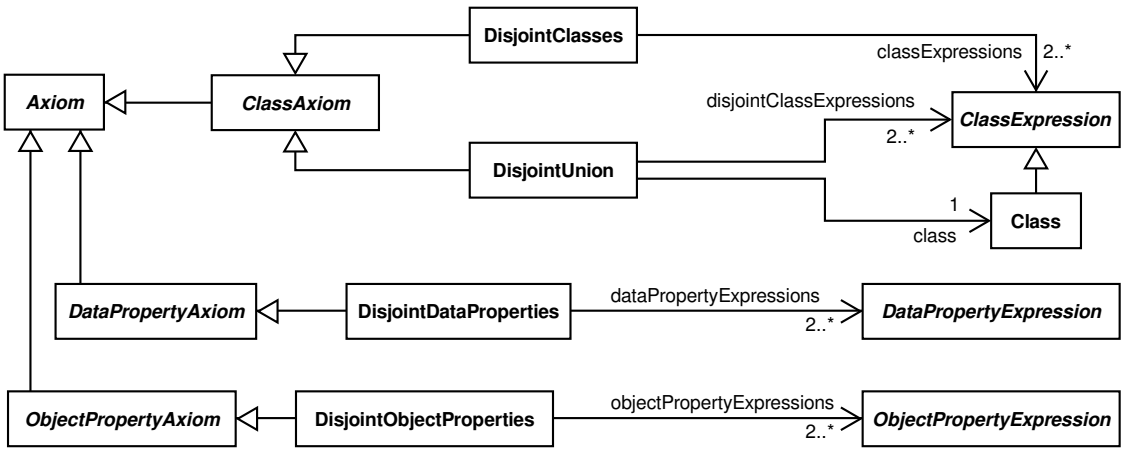


Abbildung 10.7. Der *DisjointClasses*, *DisjointUnion*, *DisjointObjectProperties* und *DisjointDataProperties* betreffende Auszug aus dem OWL-Metamodell.

10.2.4 Transformation UML → OWL

Wird der Annahme gefolgt, dass alle Klassen, die nicht in einer Vererbungsbeziehung miteinander stehen, disjunkt sind, so ist es dementsprechend notwendig, `DisjointClasses`-Axiome paarweise für alle Klassen, die nicht in einer Vererbungsbeziehung stehen, einzufügen.

Für Klassen (und Assoziationen) in einer Generalisierungsbeziehung wird nur dann keine Instanz des OWL-Elementtyps *DisjointClasses* mit entsprechenden Beziehungen erzeugt, wenn in der Instanz des UML-Elementtyps *GeneralizationSet* ausdrücklich angegeben ist, dass es sich um nicht-disjunkte Typen handelt.

10.2.5 Transformation OWL → UML

Da UML kein allgemeines Kennzeichen für die Kennzeichnung von Disjunktionen hat, ist eine allgemeine Transformation nicht möglich. Aufgrund der Tatsache, dass UML eine UNA verwendet, also Elemente mit unterschiedlichem Namen grundsätzlich als verschieden angesehen werden, besteht in vielen Anwendungsfällen keine Notwendigkeit, Typen als disjunkt zu kennzeichnen, wie dies bei OWL erforderlich ist.

Eine besondere Rolle spielt das `DisjointUnion`-Axiom, da es neben der Aussage über die Disjunktion der Elementtypen noch weitere Aussagen enthält. Seine Transformation wird im Abschnitt zur Generalisierung von Elementtypen (→ 7.6 Generalisierung) beschrieben.

10.2.6 Ausdeutung für XSD

Aufgrund der nicht vorhandenen Möglichkeit, einem Element mehr als einen Typ zuzuweisen, ist die Modellierung der Disjunktion nicht erforderlich.

10.3 Bedingte Beziehungstypen

Bei bedingten Beziehungstypen handelt es sich um eine Menge von Beziehungstypen, von denen für ein Objekt jeweils nur genau eine einzige Instanz dieser Beziehungstypen auftreten darf.

Ein Beispiel ist eine Adresse, bei der entweder eine Hausanschrift oder ein Postfach angeben ist. Eine Adresse darf eine Hausanschrift, ein Postfach aber nicht beides oder keines besitzen.

10.3.1 Logische Repräsentation

Ist E ein Elementtyp und $\{R(E, E_1) \ldots R(E, E_n)\}$ eine Menge von Beziehungstypen (mit E als erstem Teilnehmer), so darf für ein Objekt e, das eine Instanz von E ist, nur genau eine Beziehung $R(e, e_i)$ auftreten (wobei e_i eine Instanz von E_i ist).

10.3.2 Repräsentation in OWL

UML sieht kein spezielles Konstrukt vor, mit dessen Hilfe solche bedingten Beziehungstypen definiert werden könnten. Das ISO 19100 "UML-Profile" definiert jedoch einen weiteren Meta-Elementtyp *Union*, der eine entsprechende Semantik besitzt: Von der Menge der Eigenschaften einer Union darf nur eine einzige verwendet werden. Im UML-Klassendiagramm wird eine *Union* dargestellt, indem eine Klasse mit dem Stereotyp «Union» versehen wird. Wie bereits unter 3.4 geschildert, handelt es sich hierbei nicht um einen "echten" UML-Stereotypen, da eine Veränderung an der Semantik des Modellelements vorgenommen wird. Die Menge der Eigenschaften der *Union* wird durch die Menge der klassenabhängigen Attribute gebildet.

10.3.3 Transformation UML → OWL

Es wurden zwei verschiedene Abbildungen zur Transformation einer Instanz des Elementtyps *Union* nach OWL entwickelt. Die erste Variante funktioniert nur dann, wenn die Typen aller klassenabhängigen Attribute entweder nur Datentypen oder nur Klassen sind. In diesem Fall werden die Attribute in Object Properties oder Data Properties – nicht jedoch in eine Mischung von beiden – übersetzt.

Mit der zweiten Variante hingegen lassen sich auch Mischungen von Klassen und Datentypen transformieren. Es wird jedoch eine größere Anzahl von Axiomen benötigt, um die Semantik nachbilden zu können.

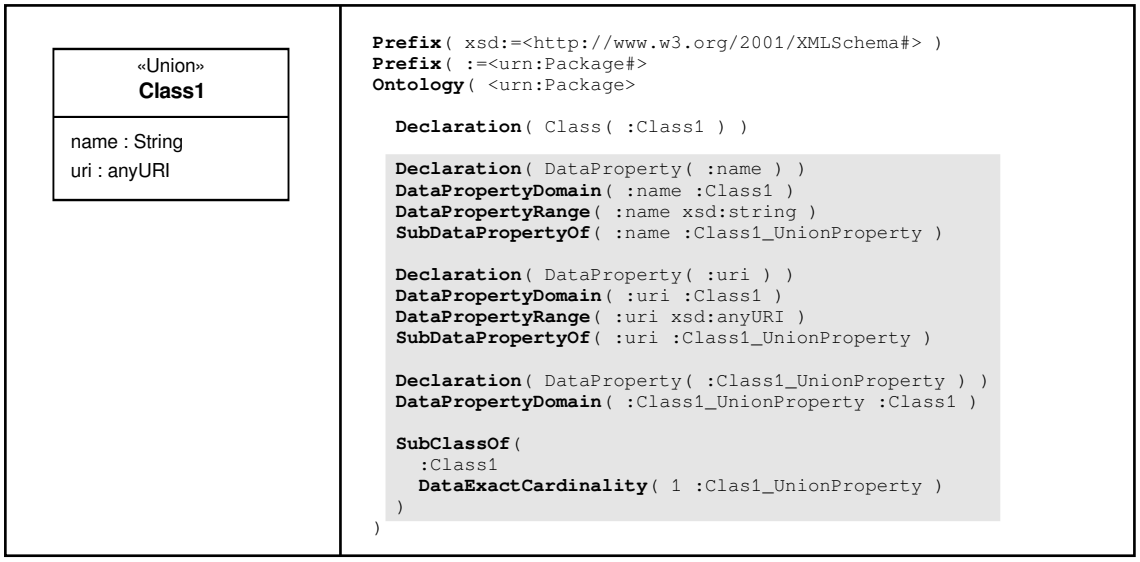
Variante 1

Sei C eine Klasse, die eine Instanz von *Union* repräsentiert und habe sie die Eigenschaften $p_1 \dots p_n$.

Um sicherzustellen, dass nur eine einzige Eigenschaft $p_x \in p_1 \dots p_n$ für ein Individuum angegeben ist, werden der Ontologie eine Hilfseigenschaft p_{Union} mit Definitionsbereich C sowie die Axiome $p_i \sqsubseteq p_{Union} \forall i \in 1..n$ hinzugefügt.

Mit Hilfe eines `DataExactCardinality`-Axioms wird die Anzahl der p_{Union}-Eigenschaften auf genau eine pro Individuum der Klasse C festgelegt. Dadurch wird verhindert, dass zwei oder mehr verschiedene Eigenschaften gesetzt werden. Aufgrund der OWA kann jedoch nicht sichergestellt werden, dass überhaupt explizit eine Eigenschaft gesetzt wurde – dieses Problem wurde bereits weiter oben im Abschnitt 9.4.4 diskutiert.

Abbildung 10.8 zeigt ein Beispiel für die Transformation einer Union mit der ersten Variante.




```
Prefix( xsd:=<http://www.w3.org/2001/XMLSchema#> )
Prefix( :=<urn:Package#>
Ontology( <urn:Package>

  Declaration( Class( :Class1 ) )

  Declaration( DataProperty( :name ) )
  DataPropertyDomain( :name :Class1 )
  DataPropertyRange( :name xsd:string )
  SubDataPropertyOf( :name :Class1_UnionProperty )

  Declaration( DataProperty( :uri ) )
  DataPropertyDomain( :uri :Class1 )
  DataPropertyRange( :uri xsd:anyURI )
  SubDataPropertyOf( :uri :Class1_UnionProperty )

  Declaration( DataProperty( :Class1_UnionProperty ) )
  DataPropertyDomain( :Class1_UnionProperty :Class1 )

  SubClassOf(
    :Class1
    DataExactCardinality( 1 :Clas1_UnionProperty )
  )
)
```

Abbildung 10.8. Erste Variante für die Transformation einer ISO 19100-Union.

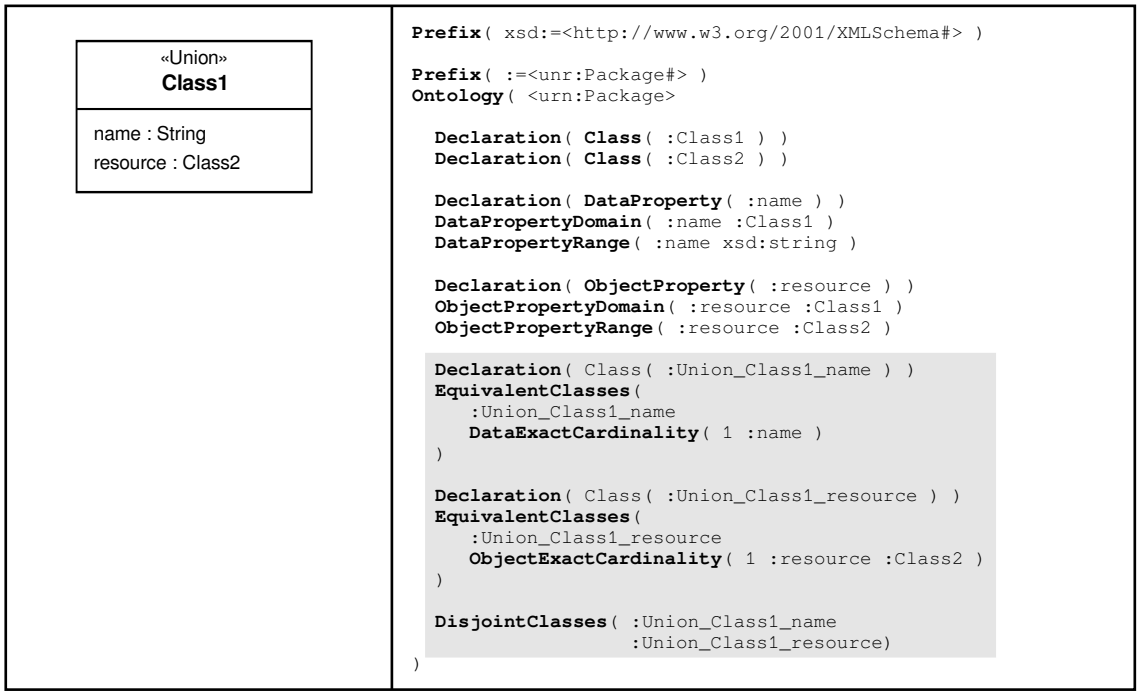
Variante 2

Für jede Eigenschaft $p_i \in p_1 \ldots p_n$ der *Union* wird eine OWL-Hilfsklasse C_i erzeugt. Mit Hilfe eines `DisjointClasses`-Axioms wird festgelegt, dass alle diese n Klassen paarweise verschieden sind. Für jede Klasse wird außerdem ausgesagt, dass sie äquivalent zur Menge der Individuen ist, die durch p_i mit genau einem Individuum bzw. Literal verbunden sind:

`EquivalentClasses(` C_i `DataExactCardinality( 1` p_i `) )` bzw.
`EquivalentClasses(` C_i `ObjectExactCardinality( 1` p_i `) )`

Abbildung 10.9 zeigt ein Beispiel für die Transformation einer Union mit der zweiten Variante.

Während der Ontologie bei der ersten Variante nur $(n + 3)$ Axiome pro UML-Eigenschaft der Union hinzugefügt werden, benötigt die zweite Variante $(2n + 1)$ zusätzliche Axiome pro Eigenschaft. Daher ist es geschickt, die erste Variante dann einzusetzen, wenn die Union ausschließlich aus Datentypen bzw. Klassen besteht, und auf die zweite Variante nur dann zurückzugreifen, wenn eine Mischung aus beiden vorliegt.




```
Prefix( xsd:=<http://www.w3.org/2001/XMLSchema#> )

Prefix( :=<unr:Package#> )
Ontology( <urn:Package>

  Declaration( Class( :Class1 ) )
  Declaration( Class( :Class2 ) )

  Declaration( DataProperty( :name ) )
  DataPropertyDomain( :name :Class1 )
  DataPropertyRange( :name xsd:string )

  Declaration( ObjectProperty( :resource ) )
  ObjectPropertyDomain( :resource :Class1 )
  ObjectPropertyRange( :resource :Class2 )

  Declaration( Class( :Union_Class1_name ) )
  EquivalentClasses(
     :Union_Class1_name
     DataExactCardinality( 1 :name )
  )

  Declaration( Class( :Union_Class1_resource ) )
  EquivalentClasses(
     :Union_Class1_resource
     ObjectExactCardinality( 1 :resource :Class2 )
  )

  DisjointClasses( :Union_Class1_name
                   :Union_Class1_resource)
)
```

Abbildung 10.9. Zweite Variante für die Transformation einer ISO 19100-Union.

10.3.4 Ausdeutung für XSD

Wie in der GML-Spezifikation beschrieben, lässt sich eine Union durch einen `complexType` mit einer `choice` darstellen.[220]

```
1 <complexType name="Class1">
2    <choice>
3       <element name="name" type="string"/>
4       <element name="uri" type="anyURI"/>
5    </choice>
6 </complexType>
```

Listing 10.1. Wiedergabe einer Union im XML Schema.

[220] Vgl. OGC: GML Standard 3.2.1, S. 349.

Kapitel 11

Strukturierung

11.1 Pakete

Mit dem Begriff *Paket* wird ein Strukturierungskonzept bezeichnet, um auch bei einer großen Vielzahl von Elementtypen und Beziehungstypen einen Überblick zu behalten. In einem Paket können Modellelemente zu größeren Einheiten zusammengefasst werden. Neben dem nicht eindeutig definierten Begriff *Paket* werden auch die Begriffe *Subsystem*, sowie die englischen *subject* und *category* verwendet.[221]

Da es sich bei Paketen um reine Strukturierungselemente handelt, haben sie keine logische Bedeutung für die in den Paketen enthaltenen Modellelemente.

11.1.1 Repräsentation in UML

Ein UML-Element, mit dessen Hilfe andere Modellelemente zusammengefasst werden können, ist das Paket. Pakete haben einen Namen, der innerhalb des beschriebenen Modells eindeutig sein muss. Da auch Pakete selbst Teil eines Pakets sein können, ist eine Verschachtelung von Paketen möglich.

Konkrete Syntax

Pakete werden in der grafischen Syntax als Rechtecke mit einem Reiter in der linken oberen Ecke dargestellt. Wird der Inhalt des Pakets dargestellt, so wird der Name des Pakets in diesen Reiter geschrieben (wie in Abbildung 11.1 links). Wird der Inhalt des Pakets nicht gezeigt, so wird der Paketname in das Rechteck geschrieben (Abbildung 11.1 rechts).

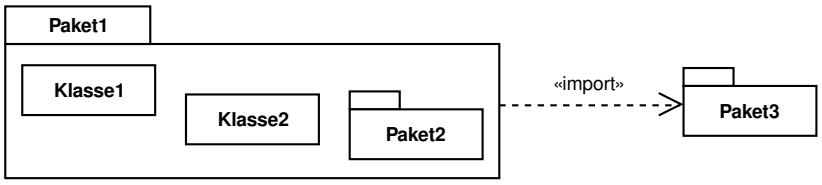


Abbildung 11.1. Beispiel für die Darstellung von Paketen in grafischer UML-Syntax.

Es existiert auch noch eine alternative Darstellung mit Hilfe von Linien und Pluszeichen.[222] Diese soll hier jedoch nicht weiter thematisiert werden.

[221] Vgl. Balzert: Softwaretechnik, S. 145.
[222] Vgl. a. a. O.

Abstrakte Syntax

Bei Paketen handelt es sich um Instanzen des Meta-Elementtyps *Package*. Diese enthalten Instanzen von *PackageableElement*, zu denen z.B. Klassen, Datentypen und Assoziationen zählen. Da ein Paket selbst Instanz von *PackageableElement* ist, kann es wiederum in Paketen enthalten sein. Als Instanz von *NamedElement* besitzt ein Paket einen Namen. In Abbildung 11.2 ist der entsprechende Ausschnitt aus dem UML-Metamodell zu sehen.

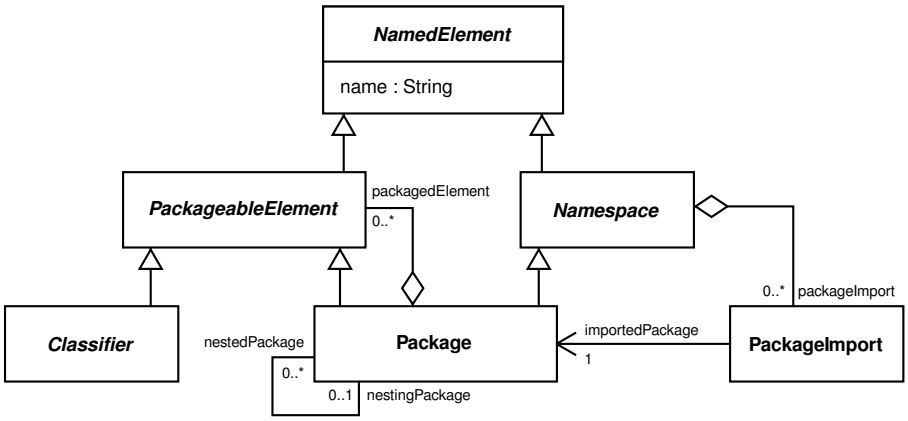


Abbildung 11.2. Ausschnitt aus dem UML-Metamodell, der die für Pakete und Paketimporte relevanten Modellelemente zeigt.

11.1.2 Repräsentation in OWL

Ein Element, mit dessen Hilfe andere Modellelemente zusammengefasst werden können, ist die Ontologie. Mit Hilfe eines IRI wird einer Ontologie ein Bezeichner gegeben. Es ist nicht möglich, Ontologien zu verschachteln. Ein Import ist jedoch möglich (→ 11.2 Importe).

Konkrete Syntax

Eine Ontologie wird mit dem Schlüsselwort `Ontology` eingeleitet. Es folgt in Klammern zunächst der `ontologyIRI`, mit dessen Hilfe die Ontologie identifiziert werden kann. Anschließend folgen die in der Ontologie enthaltenen Axiome.

```
Ontology ( <http://example.net> ...)
```

Abstrakte Syntax

In abstrakter Syntax wird eine Ontologie durch eine Instanz von *Ontology* dargestellt. Diese hat Beziehungen zu einer Instanz von *IRI* (in der Rolle *ontologyIRI*) und einer beliebigen Anzahl enthaltener Axiome (als Instanzen von *Axiom*). Abbildung 11.3 zeigt den entsprechenden Ausschnitt aus dem Metamodell.

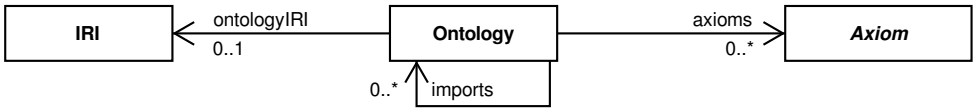


Abbildung 11.3. Ausschnitt aus dem OWL-Metamodell, der *Ontology* und in Beziehung stehende Elementtypen zeigt.

11.1.3 Transformation UML → OWL

Da sich die Konzepte von UML-Paketen und OWL-Ontologien ähneln, bietet es sich an, Pakete in Ontologien zu transformieren.

Da es sich bei dem UML-Elementtyp *Profile* um einen Untertyp des UML-Elementtyps *Package* handelt, ist zu beachten, dass UML-Profile, die Instanzen des UML-Elementtypes *Profile* sind, nicht als Pakete transformiert werden. Daher muss in den Transformationsregeln angegeben sein, dass für diese keine Transformation stattfinden soll. Mit Hilfe einer entsprechenden `when`-Klausel ist dies leicht möglich.

Bei der Namensgebung der Ontologie werden zwei Fälle unterschieden:

- Entspricht der Name des Pakets bereits den syntaktische Regeln für einen IRI, so wird der Paketname als Wert des *lexicalValue*-Attributs der Instanz des OWL-Elementtyps *IRI*, die als *ontologyIRI* dient, verwendet.
- Andernfalls wird der Wert des *lexicalValue*-Attributs nach dem Muster `"urn:" + Paketname` gebildet.

11.1.4 Transformation OWL → UML

Eine Ontologie wird in ein UML-Paket transformiert, da es ebenfalls einen Namen zur Identifikation besitzt und wie eine Ontologie alle weiteren Modellelemente enthält. Als Name des Pakets wird der lexikalische Wert des *ontologyIRI*-Attributs der *Ontology*-Instanz verwendet.

Die einfache QVT-Transformationsregel, die zu jeder Instanz des OWL-Elementtyps *Ontology* eine Instanz des UML-Elementtyps *Package* erzeugt, ist in Abbildung 11.4 zu sehen.

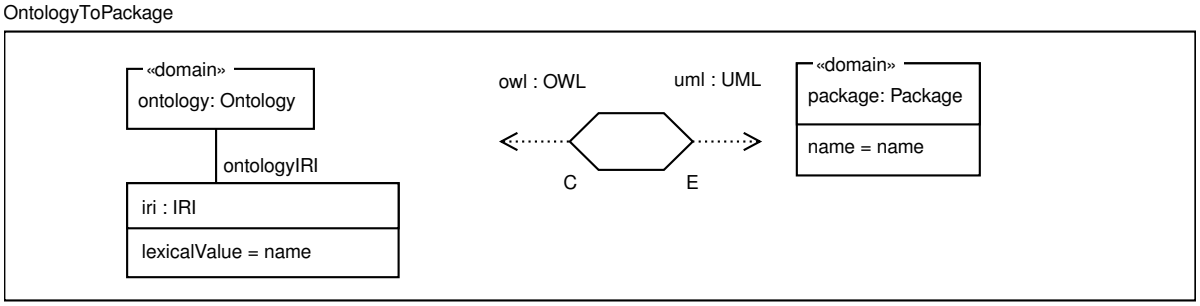


Abbildung 11.4. QVT-Regel zum Transformieren von *Ontology*.

11.1.5 Ausdeutung für XSD

Das Strukturierungskonzept von XSD ist das der Schemata. In einer Instanz von *Schema* werden Definitionen und Deklarationen zusammengefasst (siehe auch Abbildung 3.8). Ein Schema bezieht sich auf einen Namensraum, den `targetNamespace`. Da es zu einem Namensraum üblicherweise nur ein Schema gibt, lässt sich dieser als Name des Schemas verstehen.

```
1 <schema targetNamespace="http://example.org" elementFormDefault="qualified">
2   . . .
3 </schema>
```

11.2 Importe

Bei der Verwendung eines Strukturierungskonzept, wie dem der Pakete, besteht meist die Notwendigkeit, von einem Paket auf Elemente anderer Pakete zugreifen zu müssen. Damit ein Informationssystem Kenntnis von den anderen Paketen besitzt, muss eine Verbindung zwischen den Paketen hergestellt werden. Diese Verbindung wird *Import* genannt.

Da es sich bei Importen um reine Strukturierungselemente handelt, haben sie keine logische Bedeutung für die beteiligten Modellelemente.

11.2.1 Repräsentation in UML

In UML lässt sich zwischen zwei Paketen eine Import-Beziehung angeben. Ist eine solche Import-Beziehung spezifiziert, kann vom importierenden Paket aus auf Elemente des importierten Pakets zugegriffen werden, ohne qualifizierende Namen verwenden zu müssen.[223] Ein Paket kann beliebig viele andere Pakete importieren.

Die Angabe von Import-Beziehungen macht es außerdem bei komplexen UML-Modellen einfacher, die Struktur des Modells zu erkennen, da deutlich wird, welcher Teil von welchem abhängt.

Konkrete Syntax

Der Import eines Paketes wird in grafischer Syntax durch einen gestrichelten Pfeil vom importierenden Paket zum importierten Paket dargestellt. Dieser Pfeil kann mit dem Schlüsselwort «import» versehen werden. Ist kein Schlüsselwort vorhanden, wird implizit «import» angenommen. Dies ist in Abbildung 11.1 zu sehen.

Abstrakte Syntax

In der abstrakten Syntax wird der Import eines Pakets durch eine Instanz von *PackageImport* repräsentiert. Diese hat eine Beziehung zu der *Package*-Instanz, die importiert wird und als

[223] Vgl. Balzert: Softwaretechnik, S. 146.

Rolle *importedPackage* auftritt. Ein Paket kann als Instanz von *Namespace* beliebig viele Pakete importieren. Abbildung 11.2 zeigt den entsprechenden Ausschnitt aus dem Metamodell.

11.2.2 Repräsentation in OWL

Um auf Elemente anderer Ontologien zugreifen zu können, müssen diese importiert werden. Dabei werden alle in den importierten Ontologien enthaltenen Axiome in die *axiom closure* der importierenden Ontologie übernommen.[224]

Konkrete Syntax

Mit Hilfe des Schlüsselworts `Import` innerhalb einer Ontologie und der Angabe des `ontologyIRI` der zu importierenden Ontologie wir die Import-Beziehung hergestellt.

```
Ontology( <http://www.example.com/importing-ontology>
      Import( <http://www.example.com/my/2.0> )
      ...
)
```

Abstrakte Syntax

In abstrakter Syntax tritt eine Import-Beziehung als Instanz des rekursiven *imports*-Beziehungstyps auf, siehe Abbildung 11.3.

Da in einer serialisierten OWL-Ontologie nur der IRI der importierten Ontologie enthalten ist, gibt es noch zwei weitere Assoziationen für eine *Ontology*-Instanz: *directlyImportsDocuments* und *directlyImports*. Diese werden aber bereits beim Parsen des Ontologie-Dokuments aufgelöst, so dass bei der Verarbeitung der abstrakten Syntax nur noch der *imports*-Beziehungstyp berücksichtigt werden muss.

11.2.3 Transformation UML → OWL

Werden die von einem Paket importieren Pakete ebenfalls in eine Ontologie transformiert, so werden aus Instanzen des UML-Elementtyps *PackageImport* entsprechend *imports*-Referenzen auf die transformierten Instanzen des OWL-Elementtyps *Ontology* gesetzt. Die entsprechende QVT-Transformationsregel ist in Abbildung 11.5 zu sehen.

[224] Vgl. W3C: OWL 2 Structural specification, Abschnitt 3.4.

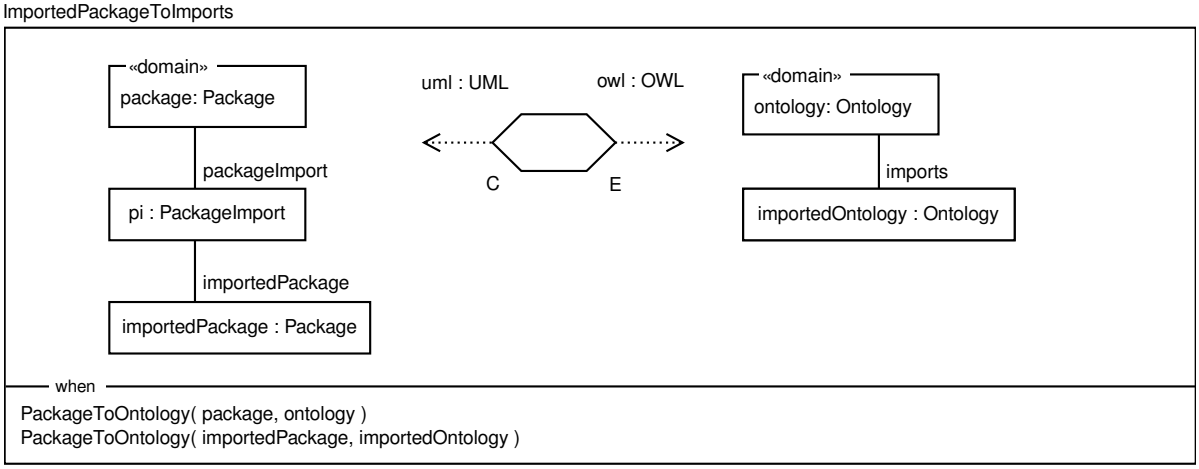


Abbildung 11.5. QVT-Regel zum Transformieren von Instanzen des UML-Elementtyps *PackageImport*.

11.2.4 Transformation OWL → UML

Da Ontologien in Pakete transformiert werden, wird der Import einer Ontologie auf einen Paket-Import abgebildet. Dazu wird für jede *imports*-Beziehung einer Instanz des OWL-Elementtyps *Ontology* eine Instanz des UML-Elementtyps *PackageImport* erzeugt. Diese QVT-Transformationsregel ist in Abbildung 11.6 zu sehen.

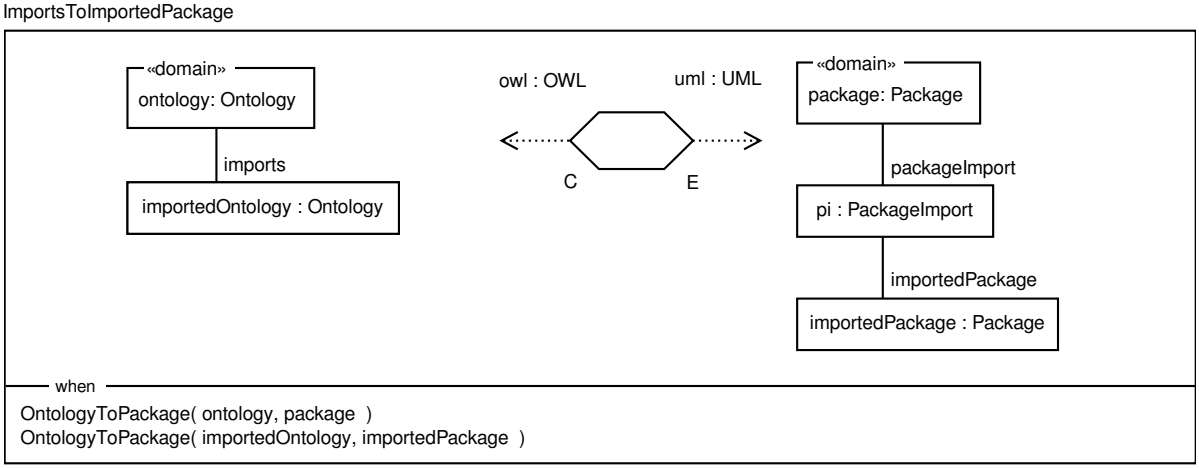


Abbildung 11.6. QVT-Regel zum Transformieren der *imports*-Beziehung von Instanzen des OWL-Elementtyps *Ontology*.

11.2.5 Ausdeutung für XSD

Sollen Definitionen aus anderen XML Schemata verwendet werden, so müssen diese importiert werden. Der Namespace des importierten Schemas wird benötigt, um mit Hilfe qualifizierter Namen auf Elemente des importierten Schemas zugreifen zu können. Mit Hilfe der `schemaLocation`-Angabe kann dem XML-Parser ein Hinweis gegeben werden, wo das XSD-Dokument zu finden ist.

```
1 <import
2    namespace="http://www.w3.org/XML/1998/namespace"
3    schemaLocation="http://www.w3.org/2001/xml.xsd">
```

Kapitel 12

Untersuchung der entwickelten Regeln

Bereits in den vorherigen Kapiteln wurde deutlich, dass sich nicht alle Konstrukte von UML bzw. OWL in den jeweils anderen Technologieraum übertragen lassen. Dieses Kapitel beschäftigt sich zum einen damit, um welchen Teil der UML- und OWL-Metamodelle es sich dabei genau handelt. Zum anderen beschäftigt es sich mit der Fragestellung, ob die in den vorherigen Kapiteln vorgestellten Transformationsregeln korrekt und – in ihren zuvor angegebenen Grenzen – vollständig sind. Dazu werden drei Arten von Untersuchungen durchgeführt:

1. Abdeckung der Metamodelle
2. Analyse einzelner Transformationsregeln
3. Automatisches Überprüfen der Transformationen

Ein Vorteil bei einer Transformation mit Hilfe von QVT-R liegt darin, dass während der Durchführung einer Transformation so genannte *trace-Klassen*[225] und deren Instanzen geschrieben werden. Während die Instanzen der trace-Klassen von den Eingabemodellen abhängen, sind die trace-Klassen selbst nicht von den verarbeiteten Modellen, sondern nur von den Transformationsregeln und den Metamodellen abhängig. Diese Aufzeichnungen werden für die Untersuchungen 1) und 2) verwendet.

Die **Untersuchung 1** der Abdeckung der Metamodelle zeigt auf, welcher Teil der UML- und OWL-Metamodelle von den Transformationen überhaupt erfasst wird. Durch eine grafische Übersicht mit Hilfe von Diagrammen der Metamodelle lassen sich die entsprechenden Teile einfach identifizieren. So wird schnell deutlich, welche (Meta-)Elementtypen nicht verarbeitet werden und bei deren Verwendung daher Informationsverlust droht. Eine detailliertere tabellarische Übersicht der Elementtypen eines Metamodells ermöglicht eine Gegenüberstellung von Definitions- und Zielbereich sowie die Angabe möglicher Einschränkungen bei der Transformation. Für die nicht bearbeiteten Elementtypen wird erläutert, warum keine Transformation durchgeführt wird.

Die weiteren Untersuchungen sind notwendig, da eine Betrachtung der Abdeckung der Metamodelle noch nicht ausreichend für die Bewertung der Transformationen ist, wie im Folgenden an zwei Beispielen gezeigt wird.

Selbst eine vollständige Abdeckung der Metamodelle garantiert noch keine semantikerhaltenden Transformationen. Auch mit trivialen und sinnlosen Transformationen lässt sich

[225] Siehe OMG: QVT Specification 1.0, S. 5.

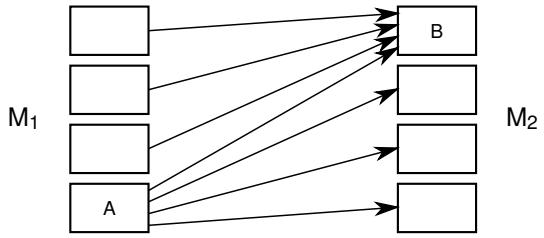


Abbildung 12.1. Skizze trivialer Transformationen, die Instanzen aller Eingabe-Elementtypen auf Instanzen eines Ausgabe-Elementtyps abbildet.

eine solche vollständige Abdeckung erzielen. Die Skizze in Abbildung 12.1 zeigt ein Beispiel für eine solche Transformation: Alle Elementtypen des einen Metamodells M_1 werden auf einen einzigen Elementtyp B des anderen Metamodells M_2 abgebildet. So wird für M_1 eine vollständige Abdeckung erreicht. Um außerdem eine vollständige Abdeckung der Elementtypen von M_2 zu erreichen, wird ein Transformationsregel angegeben, die aus jeder Instanz des Elementtyps A jeweils eine Instanz jeden Elementtyps des Metamodells M_2 erzeugt. Dadurch wird ebenfalls eine vollständige Abdeckung von M_2 erreicht.

Es lassen sich ebenfalls zwei Transformationen T und T' definieren, bei denen die Hintereinanderausführung auf ein Modell m wiederum $T'(T(m)) = m$ ergibt, ohne dass jedoch mit dem Ergebnis der Transformation $T(m)$ sinnvoll gearbeitet werden kann.

Als Beispiel für diesen Sachverhalt soll eine triviale Transformation T dienen, die das Modell m (z.B. unter Nutzung einer konkreten Syntax) in einen Kommentar des Modells $m' = T(m)$ abbildet. Die Rücktransformation T' stellt das das Modell entsprechend aus dem Kommentar wieder her. Diese beiden Transformationen würden die Bedingung $T'(T(m)) = m$ erfüllen. Es ist jedoch offensichtlich, dass in m' keines der Elemente von m irgendeine Bedeutung hätte. Die Listings 12.1 und 12.2 zeigen ein Beispiel für eine solche Transformation vom Ontology TS in den XML TS.

```
1 Prefix(:=<urn:m#> )
2 Ontology(<urn:m>
3         Declaration(Class(:Book))
4         Declaration(Class(:Author))
5         Declaration(ObjectProperty(:hasAuthor))
6         ObjectPropertyDomain(:hasAuthor :Book)
7         ObjectPropertyRange(:hasAuthor :Author)
8 )
```

Listing 12.1. Einfache Ontologie als Beispiel für ein Modell m.

```
1 <schema xmlns="http://www.w3.org/2001/XMLSchema">
2   <annotation>
3    <documentation>Prefix(:=&lt;urn:m#&gt; ) Ontology(&lt;urn:m&gt; Dclaration(Class(:Book))
         Declaration(Class(:Author)) Declaration(ObjectProperty(:hasAuthor)) ObjectPropertyDomain(:
         hasAuthor :Book) ObjectPropertyRange(:hasAuthor :Author))</documentation>
4   </annotation>
5 </schema>
```

Listing 12.2. Eine triviale Transformation von m in ein XML-Schema.

Auch in diesem Fall kann die Abdeckung der Metamodelle nichts Auffälliges zeigen, wie ebenfalls anhand der Skizze in Abbildung 12.1 erläutert werden kann: Alle Elementtypen des Eingabemodells M_1 werden verarbeitet und auf Instanzen des Kommentar-Elementtyps B abgebildet. So ist für M_1 eine vollständige Abdeckung erreicht. Zur Erreichung einer vollständigen Abdeckung von M_2 kann das gleiche Vorgehen wie im vorherigen Beispiel gewählt werden.

Es ist also notwendig, die Transformationsregeln weiteren Untersuchungen zu unterziehen. Dazu werden mit **Untersuchung 2** einzelne, zueinander inverse Transformationsregeln mit ihren Abhängigkeiten untereinander sowie den Abhängigkeiten zu den Meta-Elementtypen betrachtet. Hierbei würden Regeln, die nur künstlich die Abdeckung der Metamodelle erhöhen sollen, schnell auffallen, da sich entweder

- eine Regel auf Instanzen ungewöhnlich vieler Elementtypen bezieht oder
- eine Regel Instanzen ungewöhnlich vieler Elementtypen erzeugt.

Im Einzelfall ist die Analyse inverser Transformationsregeln mit abhängigen Regeln und zugehörigen Meta-Elementtypen sehr umfangreich. Es werden daher im Rahmen dieser Ausarbeitung nur einige Beispiele vorgestellt, die das Vorgehen erläutern und Ergebnisse für verschiedene Fälle zeigen:

1. komplette Transformation in beide Richtungen ohne Informationsverlust
2. bereits bekannter und dokumentierter Verlust von Information
3. beispielhafte Transformation zum Erhöhen der Abdeckung der Metamodelle, die bei Untersuchung 1 nicht auffallen würden

Aufgrund der Komplexität der manuellen Analyse der Transformationsregeln und der damit verbundenen Gefahr, Fehler zu übersehen, ist eine automatische Überprüfung der Transformationen wünschenswert. Eine solche wird mit **Untersuchung 3** vorgestellt. Dazu werden die beiden Transformationen automatisiert hintereinander ausgeführt und anschließend das Eingabemodell und das Ausgabemodell geeignet miteinander verglichen. Dieses Verfahren sowie ein "geeigneter" Vergleich werden in dem entsprechenden Abschnitt beschrieben.

Nicht untersucht werden Metriken für die QVT-R Transformationsregeln selbst, da

- es in dieser Arbeit um den Umfang der transformierbaren Bestandteile der beiden Metamodelle und den Semantikerhalt der transformierten Modell-Elemente geht,
- es keine verschiedenen Regelsätze für die gleiche Aufgabenstellung gibt, die verglichen werden könnten,
- die erstellten Transformationsregeln übersichtlich und verständlich aber nicht z.B. in Hinblick auf schnelle Transformation optimiert sein sollen und

- es nur wenig Literatur im Bereich von Metriken zur Bewertung von QVT-R Transformationen gibt.[226] Kapova et al. nennen zwanzig Metriken für QVT-R, die automatisch bestimmt werden können. Sie weisen jedoch darauf hin, dass noch keine ausreichenden Informationen vorliegen, um aus den ermittelten Zahlen für eine Metrik Rückschlüsse auf die Qualität der Transformationsregeln selbst treffen zu können.[227]

12.1 Abdeckung der Metamodelle

Ein wichtiger Aspekt bei der Bewertung der Transformationen ist die Menge der von den Regeln abgedeckten Metamodell-Elemente. Sie zeigt, welcher Teil der UML- und OWL-Metamodelle von den Transformationen überhaupt erfasst wird. Es werden vier Aufstellungen benötigt:

1. OWL als Definitionsbereich der Transformation OWL → UML
2. OWL als Zielbereich der Transformation UML → OWL
3. UML als Definitionsbereich der Transformation UML → OWL
4. UML als Zielbereich der Transformation OWL → UML

Mit Hilfe der trace-Klassen lässt sich bestimmen, welche Elementtypen des Metamodells bei der Transformation UML → OWL sowie der Transformation OWL → UML berücksichtigt werden.

12.1.1 Abdeckung der OWL-Meta-Elementtypen

Wie im Abschnitt 3.2.3 geschildert enthält das OWL-Metamodell insgesamt 69 Klassen, die bei der Definition des Schemas einer Ontologie zum Einsatz kommen.

Abbildung 12.2 gibt einen Überblick darüber, welche Teile des OWL-Metamodells von den beiden Transformationen OWL → UML und UML → OWL behandelt werden. Bei zehn der nicht markierten Elementtypen handelt es sich um abstrakte Klassen. Für die anderen nicht transformierten Elementtypen wird im Folgenden begründet, warum eine Transformation nicht möglich ist.

Tabelle 12.1 gibt eine genaue Aufstellung über die OWL-Meta-Elementtypen, die im Definitionsbereich der Transformation OWL → UML bzw. im Zielbereich der Transformation UML → OWL liegen. Ein "•" gibt dabei an, dass der entsprechende Elementtyp in der Transformation behandelt wird. Ein "○" bedeutet, dass die Transformation mit gewissen Einschränkungen möglich ist. Bei Elementtypen, bei denen eine Transformation nicht oder nur mit Einschränkungen möglich ist, verweist die Zahl auf die der Tabelle folgenden Erläuterungen.

[226] Kapova, L. et al.: Evaluating Maintainability with Code Metrics for Model-to-Model Transformations, in: Heineman, G. T./Kofron, J./Plasil, F. (Hrsg.): Research into Practice – Reality and Gaps, Berlin/Heidelberg 2010: "*However, for relational modeltransformation languages like QVT Relations there is not even a comparable amount of literature.*"

[227] Vgl. a. a. O.

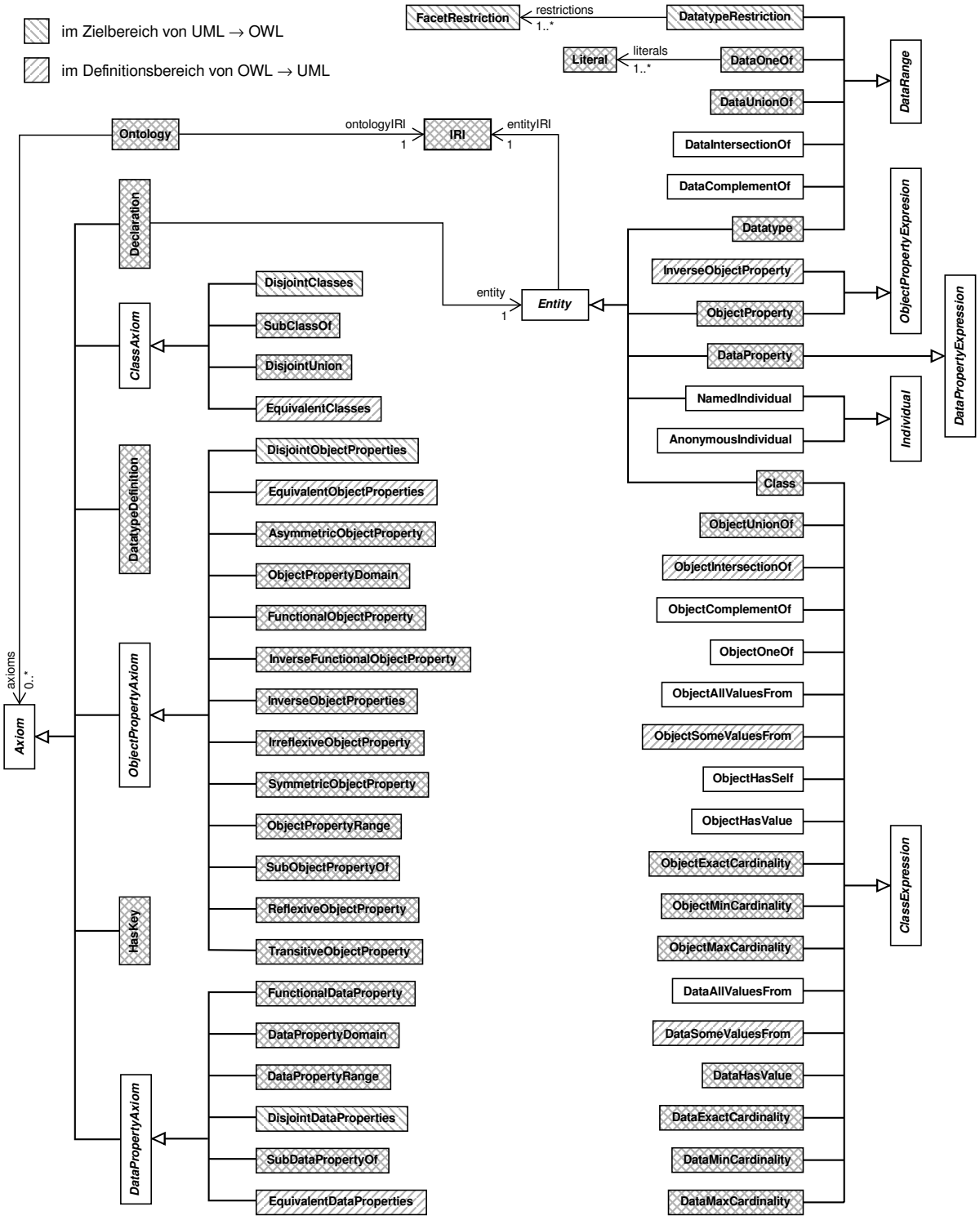


Abbildung 12.2. Auftreten der Elementtypen des OWL-Metamodells im Definitions- bzw. Zielbereich der Transformationen OWL → UML und UML → OWL.

Tabelle 12.1. Abdeckung der OWL-Meta-Elementtypen.

OWL-Meta-Elementtyp	**Im Definitionsbereich von OWL → UML**	**Im Zielbereich von UML → OWL**
Ontology	•	•
Imports	•	•
IRI	•	•
FacetRestriction	-[1)]	•
Literal	•	•
AnonymousIndividual	-[2)]	-[3)]
InverseObjectProperty	•	-[4)]
Entities		
Class	•	•
DataProperty	•	•
Datatype	•	•
ObjectProperty	•	•
NamedIndividual	-[2)]	-[3)]
Class Expressions		
DataAllValuesFrom	-[11)]	-[11)]
DataExactCardinality	•	•
DataHasValue	•	•
DataMaxCardinality	•	•
DataMinCardinality	•	•
DataSomeValuesFrom	•	-[10)]
ObjectAllValuesFrom	-[11)]	-[11)]
ObjectComplementOf	-[7)]	-[7)]
ObjectExactCardinality	•	•
ObjectHasSelf	-[9)]	○[8)]
ObjectHasValue	-[20)]	-[20)]
ObjectIntersectionOf	○[5)]	-[6)]
ObjectMaxCardinality	•	•
ObjectMinCardinality	•	•
ObjectOneOf	-[20)]	-[20)]
ObjectSomeValuesFrom	•	-[10)]
ObjectUnionOf	•	•
Class Axioms		
DisjointClasses	○[15)]	•
DisjointUnion	•	•

Weiter auf der nächsten Seite ...

Tabelle 12.1 – fortgesetzt von vorheriger Seite

OWL-Meta-Elementtyp	**Im Definitionsbereich von OWL → UML**	**Im Zielbereich von UML → OWL**
EquivalentClasses	•	-[14]
SubClassOf	•	•
Object Property Axioms		
AsymmetricObjectProperty	○[18]	○[18]
DisjointObjectProperties	-[17]	•
EquivalentObjectProperties	•	-[16]
FunctionalObjectProperty	•	•
InverseFunctionalObjectProperty	•	•
InverseObjectProperties	•	•
IrreflexiveObjectProperty	○[18]	○[18]
ObjectPropertyDomain	•	•
ObjectPropertyRange	•	•
ReflexiveObjectProperty	○[18]	○[18]
SubObjectPropertyOf	•	•
SymmetricObjectProperty	○[19]	○[18]
TransitiveObjectProperty	○[18]	○[18]
Data Property Axioms		
DataPropertyDomain	•	•
DataPropertyRange	•	•
DisjointDataProperties	-[17]	•
EquivalentDataProperties	•	-[16]
FunctionalDataProperty	•	•
SubDataPropertyOf	•	•
Data Ranges		
DataComplementOf	-[7]	-[7]
DataIntersectionOf	-[13]	-[13]
DataOneOf	•	•
DatatypeRestriction	-[12]	•
DataUnionOf	•	•
Sonstige Axioms		
DatatypeDefinition	•	•
Declaration	•	•
HasKey	•	•

1. **FacetRestriction** (OWL → UML)
   In der UML gibt es keine Entsprechungen für die komplexen Möglichkeiten eigene Datentypen zu definieren, wie sie Instanzen des OWL-Elementtyps *FacetRestriction* bieten.

2. **AnonymousIndividual** und **NamedInvididual** (OWL → UML)
   Da sich die Transformation nur auf die T-Box der Ontologie bezieht, treten Individuen nur in Verbindung mit den CE *ObjectHasValue* und *ObjectOneOf* auf. Diese können jedoch nicht transformiert werden, siehe Anmerkung 20. Daher werden auch Individuen nicht transformiert.

3. **AnonymousIndividual** und **NamedInvididual** (UML → OWL)
   Da Elementtypen und Beziehungstypen (in Modellen auf Ebene M1), jedoch keine Instanzen transformiert werden, werden entsprechend bei der Transformation keine Individuen in der Ontologie erzeugt.

4. **InverseObjectProperty** (UML → OWL)
   Da UML keine anonymen Beziehungstypen kennt, die als invers zu einem anderen (benannten) Beziehungstyp definiert sind, wird dieses Axiom nicht erzeugt. Verwendet wird jedoch das für zwei benannte Beziehungstypen genutzte Axiom `InverseObjectProperties`.

5. **ObjectIntersectionOf** (OWL → UML)
   Es wird für die in einer `ObjectIntersectionOf` aufgeführten Klassen eine abstrakte Unterklasse definiert, die von allen aufgeführten Klassen erbt. Es ist jedoch aufgrund der im Abschnitt 6.7 diskutierten Probleme nicht möglich, zu erzwingen, dass jedes Objekt, das Instanz aller aufgeführter Klassen ist, ebenfalls Instanz der abstrakten Unterklasse ist. (→ 7.7 Schnittmenge) Daher geht hier ein Teil der Semantik des Axioms verloren.

6. **ObjectIntersectionOf** (UML → OWL)
   UML kennt kein Konstrukt, um eine Schnittmenge von Klassen zu definieren. Daher wird bei der Transformation UML → OWL keine solche CE erzeugt.

7. **DataComplementOf** und **ObjectComplementOf** (UML ↔ OWL)
   Aufgrund der im Abschnitt 6.8 diskutierten Probleme ist es allgemein nicht möglich, per Komplement definierte Elementtypen in UML abzubilden.

8. **ObjectHasSelf** (UML → OWL)
   Da es in UML keinen Elementtyp gibt, der dadurch definiert ist, dass ein Objekt mit sich selbst in einer Beziehung steht, wird dieser Meta-Elementtyp bei der Transformation UML → OWL nicht erzeugt.

9. **ObjectHasSelf** (OWL → UML)
   Werden, wie in Abschnitt 6.10 beschrieben, vordefinierte Stereotypen zur Kennzeichnung der Eigenschaften von Object Properties (siehe auch Anmerkung 18) verwendet, so kann die aus der Object Property der `ObjectHasSelf`-CE transformierte Assoziation mit einem

solchen Stereotyp als reflexiv gekennzeichnet werden. Die eigentliche Semantik eines Elementtyps wird dabei jedoch nicht abgebildet.

10. **DataSomeValuesFrom** und **ObjectSomeValuesFrom** (UML ↔ OWL)
Da es sich bei beiden Axiomen nur um Abkürzungen handelt (siehe Tabelle 3.1), werden sie bei der Transformation UML → OWL nicht erzeugt. Stattdessen werden `DataMinCardinality`- bzw. `ObjectMinCardinality`-Axiome verwendet.

11. **DataAllValuesFrom** und **ObjectAllValuesFrom** (UML ↔ OWL)
Bei diesen CE werden Instanzen über hinreichende Bedingungen (→ 6.7 Notwendige und hinreichende Bedingungen) ausgewählt. Da diese allgemein nicht in UML wiedergegeben werden können, scheidet eine Transformation aus. Auch die ausführliche Schreibweise der Axiome (es handelt sich um Abkürzungen, siehe Tabelle 3.1) eignet sich nicht für eine Transformation, da in dem Fall die Komplementbildung hinderlich ist.

12. **DatatypeRestriction** (UML ↔ OWL)
In der UML gibt es keine Entsprechungen für die komplexen Möglichkeiten von OWL, eigene primitive Datentypen zu definieren.

13. **DataIntersectionOf** (UML ↔ OWL)
UML sieht keine Möglichkeit vor, einen neuen Datentypen als Schnittmenge zweier anderer Datentypen zu definieren.

14. **EquivalentClasses** (UML → OWL)
Da es in UML – bis auf gegenseitige Generalisierungsbeziehungen, die als solche auch transformiert werden – keine Möglichkeit gibt, zwei Klassen als äquivalent zu kennzeichnen, wird kein `EquivalentClasses`-Axiom erzeugt.

15. **DisjointClasses** (OWL → UML)
Durch die UNA von UML ist bereits sichergestellt, dass zwei Klassen als unterschiedlich angesehen werden, wenn sie einen unterschiedlichen Namen besitzen. Es kann jedoch auch beabsichtigt sein, durch Verwendung dieses Axioms zu verhindern, dass ein Objekt Instanz mehrerer der im Axiom aufgelisteten Elementtypen ist (→ 7 Elementtypen). Dieser Sachverhalt lässt sich nur dann transformieren, wenn alle im Axiom aufgeführten Klassen eine gemeinsame Oberklasse haben (→ 10.2 Disjunktion).

16. **EquivalentDataProperties** und **EquivalentObjectProperties** (UML → OWL)
Da es in UML – bis auf gegenseitige Generalisierungsbeziehungen, die als solche auch transformiert werden – keine Möglichkeit gibt, zwei Assoziationen als äquivalent zu kennzeichnen, werden keine `EquivalentDataProperties` und `EquivalentObjectProperties`-Axiome erzeugt.

17. **DisjointDataProperties** und **DisjointObjectProperties** (OWL → UML)
Durch die UNA von UML ist bereits sichergestellt, dass zwei Properties als unterschiedlich angesehen werden, wenn sie einen unterschiedlichen Namen besitzen.

18. **Eigenschaften von Object Properties** (UML ↔ OWL)
    Um zwischen Eigenschaften von Assoziationen auf der einen Seite und entsprechenden Eigenschaften von Object Properties (Axiome `AsymmetricObjectProperty`, `IrreflexiveObjectProperty`, `ReflexiveObjectProperty`, `TransitiveObjectProperty`) auf der anderen Seite transformieren zu können, ist es notwendig, die Assoziationen im UML-Modell geeignet zu kennzeichnen. Dafür lassen sich, wie in Abschnitt 6.10 beschrieben, vordefinierte Stereotypen verwenden. Bei einer Transformation, die von der Verwendung dieser Stereotypen anhängt, handelt es sich jedoch nicht mehr um eine Transformation allgemeiner UML-Modelle.

19. **SymmetricObjectProperty** (OWL → UML)
    Beim Vorliegen einer symmetrischen Object Property werden zwei zueinander inverse Beziehungstypen erstellt. Es wird jedoch nicht erzwungen, dass aus der Existenz einer Instanz des einen Beziehungstyps automatisch auch die Existenz des inversen Beziehungstyps folgt. Daher geht hier ein Teil der Semantik dieses Axioms verloren.

20. **ObjectHasValue** und **ObjectOneOf** (UML ↔ OWL)
    Da UML keine Unterstützung für Elementtypen mit fester Population (→ 7.5 Elementtypen mit fester Population) bietet, ist es nicht möglich, die CE `ObjectOneOf` zu transformieren. Beim der CE `ObjectHasValue` handelt sich um eine Abkürzung (siehe Tabelle 3.1), die eine Instanz des OWL-Elementtyps *ObjectOneOf* enthält. Daher ist auch hier eine Transformation nicht möglich.

12.1.2 Abdeckung der UML-Meta-Elementtypen

Abbildung 12.3 zeigt, welche Elementtypen des UML-Metamodells im Definitions- bzw. Zielbereich der Transformationen UML → OWL und OWL → UML liegen.[228] Auf den ersten Blick scheinen nur relativ wenige Elementtypen behandelt zu werden. Dieser Eindruck täuscht jedoch, da es sich bei den nicht markierten Elementtypen um abstrakte Typen handelt. Lediglich *InstanceSpecification* ist ein instantiierbarer Elementtyp – aber auch dieser tritt nur als Obertyp von *EnumerationLiteral* auf, hat also selbst keine direkten Instanzen.

Tabelle 12.2 gibt eine genaue Aufstellung über die UML-Meta-Elementtypen, die im Definitionsbereich der Transformation UML → OWL bzw. im Zielbereich der Transformation OWL → UML liegen. Ein "•" gibt dabei an, dass der entsprechende Elementtyp in der Transformation behandelt wird. Ein "○" bedeutet, dass die Transformation mit gewissen Einschränkungen möglich ist. Bei Elementtypen, bei denen eine Transformation nicht oder nur mit Einschränkungen möglich ist, verweist die Zahl auf die der Tabelle folgenden Erläuterungen.

[228] Um das Diagramm nicht noch unübersichtlicher zu machen, wurden mehrfache *IsA*-Beziehungen und folgende abstrakte Elementtypen mit ihren *IsA*-Beziehungen entfernt: *Property* → *ConnectableElement* → *TypedElement*, *StructuralFeature* → *Feature* → *RedefinableElement* → *NamedElement*, *Classifier* → *RedefinableElement* → *NamedElement*, *Association* → *Relationship* → *Element*, *Generalization* → *DirectedRelationship* → *Relationship* → *Element*, *Class* → *EncapsulatedClassifier* → *StructuredClassifier*.

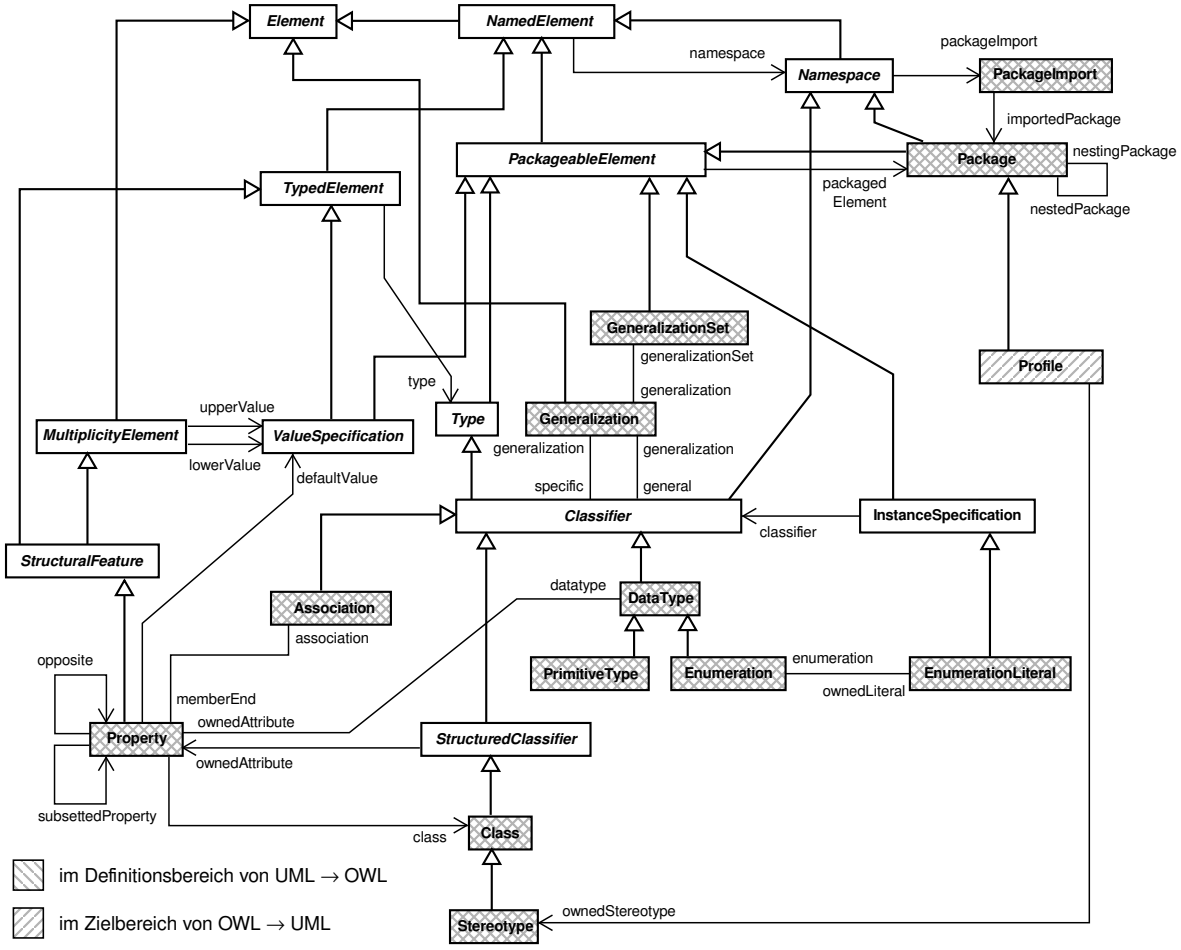


Abbildung 12.3. Auftreten der Elementtypen des UML-Metamodells im Definitions- bzw. Zielbereich der Transformationen UML → OWL und OWL → UML.

Tabelle 12.2. Abdeckung der UML-Meta-Elementtypen.

UML-Meta-Elementtyp	**Im Definitionsbereich von UML → OWL**	**Im Zielbereich von OWL → UML**
Association	•	•
Class	•	•
DataType	○[1)]	•
Enumeration	•	•
EnumerationLiteral	•	•
Generalization (Class)	•	•
Generalization (Association)	•	•
GeneralizationSet	•	•
Package	•	•
PackageImport	•	•
Profile	○[2)]	-[3)]
Property	•	•
Stereotype	○[4)]	○[5)]

1. **DataType** (UML → OWL)
   Ein zusammengesetzter Datentyp wird in eine OWL-Klasse transformiert. Da Instanzen dieser Klasse eine Identifikation besitzen, gibt es hier eine Abweichung der Semantik. Nach der Transformation ist in der Ontologie nicht mehr zu erkennen, dass es sich um einen zusammengesetzten Datentypen gehandelt hat. Bei einer Rücktransformation würde so aus einem zusammengesetzten Datentypen eine Klasse.

2. **Profile** (UML → OWL)
   Es geht bei der Transformation von Profilen Information verloren. Es wird nur die Semantik bestimmter Profile (ISO 19110) berücksichtigt. Hat man sich ein Profil mit Stereotypen zur Kennzeichnung der Eigenschaften von Beziehungstypen (siehe Abschnitt 6.10) angelegt, so kann auch dieses transformiert werden. Im allgemeinen Fall werden Stereotypen jedoch auf Klassen und Vererbungsbeziehungen abgebildet.

3. **Profile** (OWL → UML)
   Da OWL kein den UML-Profilen entsprechendes Konzept kennt (siehe Abschnitt 6.4), werden bei der Transformation keine Instanzen des UML-Elementtyps *Profile* erzeugt.

4. **Stereotype** (UML → OWL)
   Die Markierung einer Klasse C mit einem Stereotyp S wird in eine Vererbungsbeziehung zweier UML-Klassen C und S transformiert. Dabei geht die Information, dass es sich bei S um einen Stereotyp und nicht um einen Elementtyp handelt, verloren.

5. **Stereotype** (OWL → UML)
   Da bei der Transformation UML → OWL Stereotypen in normale OWL-Klassen transformiert werden, ist kein Unterschied zu anderen Klassen der Ontologie festzustellen. Daher werden nach einer Hin- und Rücktransformation keine Stereotypen mehr zu finden sein.
   Werden – wie in Abschnitt 6.10 beschrieben, Stereotypen zur Angabe der Eigenschaften von Beziehungstypen verwendet, so wird bei der Transformation OWL → UML auf entsprechende Instanzen des UML-Elementtyps *Stereotype* verwiesen, die bereits in einem, in die Transformation eingebundenen, UML-Profil definiert sein müssen.

12.2 Analyse einzelner Transformationsregeln

Mit Hilfe der trace-Klassen und den Metamodellen lässt sich überprüfen, ob die Transformationen wie gewünscht wirken und ob alle relevanten Modellelemente mit einbezogen werden. Werden die trace-Klassen der Transformation UML → OWL, die trace-Klassen der Transformation OWL → UML sowie die beiden Metamodelle kombiniert, so lässt sich analysieren, ob beide Transformationen invers zueinander sind.

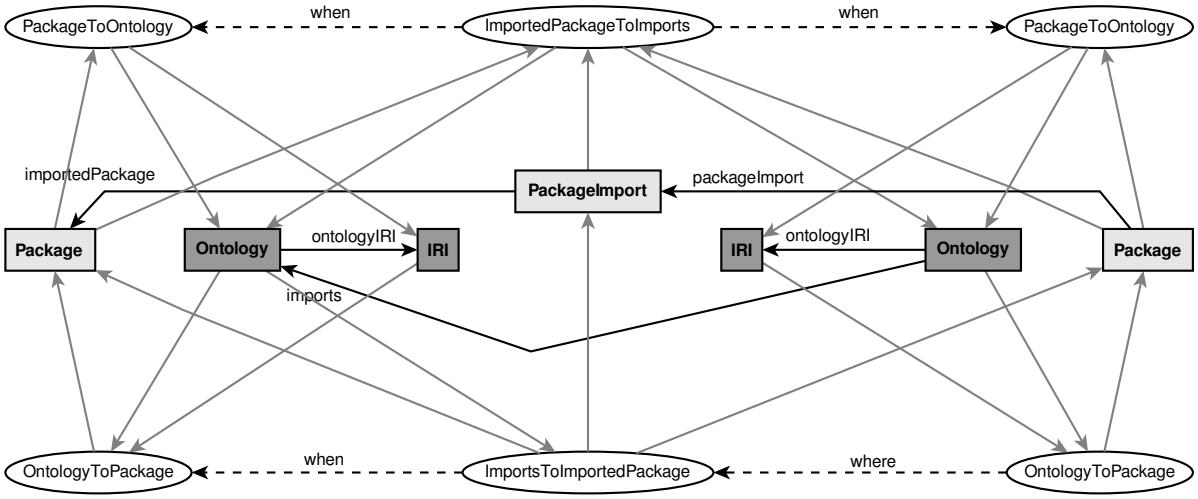


Abbildung 12.4. Analyse der Transformation von Paket- bzw. Ontologie-Importen.

Abbildung 12.4 zeigt eine solche Analyse beispielhaft für die Transformation von Paket- bzw. Ontologie-Importen. Relevant für dieses einfache und noch übersichtliche Beispiel sind auf der einen Seite die beiden Top-Relationen `ImportedPackageToImports` und `PackageToOntology`, auf der anderen Seite die Relation `ImportsToImportedPackage` und die Top-Relation `OntologyToPackage`.

In der oberen Zeile der Abbildung sind – als Ellipsen dargestellt – die Instanzen der für die Transformationsrichtung UML → OWL relevanten Regeln zu sehen. Die gestrichelten Pfeile geben die Abhängigkeit der Regeln über `when`-Bedingungen an. `PackageToOntology` taucht in der Abbildung doppelt auf, da – wie in der grafischen Syntax der Regeln (Abbildung 11.5)

zu sehen – die when-Bedingungen von `ImportedPackageToImports` zwei Abhängigkeiten zu `PackageToOntology` enthalten: erstens für das importierende Paket und zweitens für das importierte Paket. Die in Richtung der Regeln zeigenden grauen Pfeile geben an, dass der betreffende Elementtyp innerhalb der `checkonly domain` verwendet wird. Ein ausgehender Pfeil zeigt entsprechend auf Elementtypen, die innerhalb der `enforce domain` zum Einsatz kommen.

In der Mitte der Abbildung sind – mit Rechtecken dargestellt – Instanzen der Elementtypen des UML-Metamodells (hellgrau) und des OWL-Metamodells (dunkelgrau) zu sehen. Die schwarzen Pfeile geben im Metamodell definierte Beziehungen zwischen den Instanzen an. Als Beschriftung wird der Rollenname des zweiten Teilnehmers verwendet.

In der untersten Zeile befinden sich die Instanzen der für die Transformationsrichtung OWL → UML relevanten Regeln, ebenfalls als Ellipsen dargestellt. Auch hier geben die gestrichelten Pfeile die Abhängigkeit der Regeln an, in diesem Fall eine `when`- und eine `where`-Abhängigkeit. Hier taucht `OntologyToPackage` zweimal auf, da die Regel in einem Fall als Top-Relation angewendet wird und im zweiten Fall innerhalb der `when`-Bedingung von `ImportsToImportedPackage`, angewendet auf die importierte Ontologie.

Wie an dem Diagramm zu erkennen ist, verhalten sich die beiden Transformationen invers zueinander, so dass die Transformation beliebig oft wiederholt werden kann, ohne dass Modellelemente verloren gehen oder neu hinzukommen.

Informationsverlust

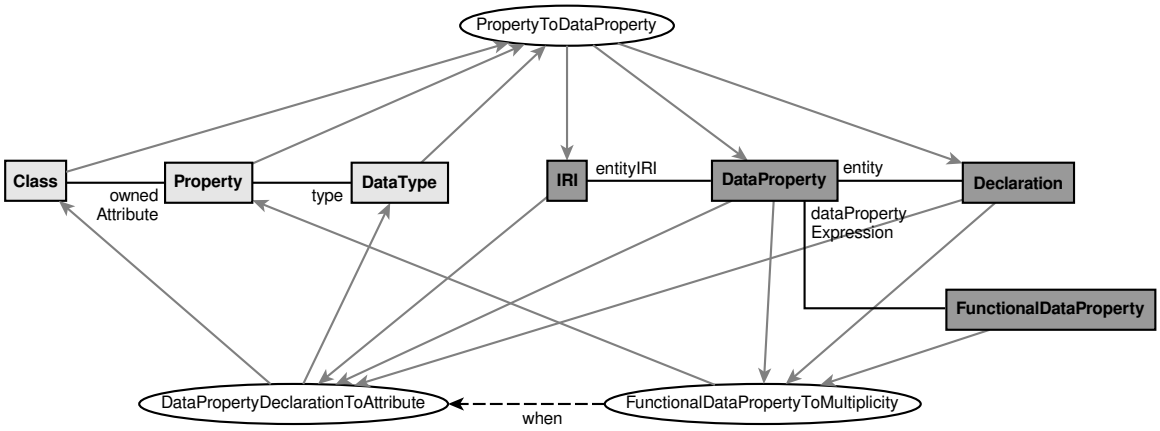


Abbildung 12.5. Analyse der Transformation von funktionalen Data Properties.

Ein Beispiel für den Fall, in dem beide Transformationen nicht invers zueinander sind und Information verlorengeht, ist in Abbildung 12.5 dargestellt. Der Übersichtlichkeit halber ist die Darstellung auf die notwendigsten Elemente beschränkt. So sind die Regeln, mit denen sichergestellt wird, dass die Elemente zum passenden UML-Paket bzw. zur passenden Ontologie gehören (und zugehörige Modellelemente), nicht enthalten.

Auch hier ist in der obersten Zeile die Instanz der für die Transformationsrichtung UML → OWL relevanten Regel `PropertyToDataProperty` zu sehen. Im mittleren Bereich befinden sich die Instanzen der Elementtypen des UML-Metamodells (hellgrau) und des OWL-Metamodells (dunkelgrau) mit ihren Beziehungen. In der untersten Zeile sind die Instanzen der Regeln für die Transformationsrichtung OWL → UML zu sehen, die beiden Top-Relationen `FunctionalDataPropertyToMultiplicty` und `DataPropertyDeclaration-ToAttribute` mit einer `when`-Abhängigkeit.

Wie zu erkennen ist, taucht der OWL-Elementtyp *FunctionalDataProperty* nur in der `checkonly domain` der Transformationsrichtung OWL → UML auf (grauer Pfeil). Eine Verbindung zu den Regeln der Transformationsrichtung UML → OWL gibt es nicht, d.h. es wird bei der Transformation UML → OWL nie eine Instanz von *FunctionalDataProperty* erzeugt.

Beispiel für "betrügerische" Regeln

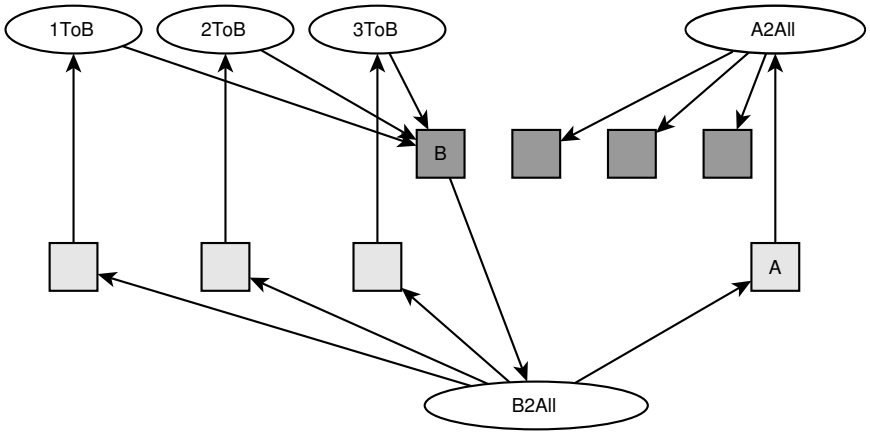


Abbildung 12.6. Analyse einer "betrügerischen" Transformation.

Um zu zeigen, wie mit Hilfe der Analyse einzelner Regeln "betrügerische" Transformationen (siehe einleitender Text zu diesem Kapitel) entdeckt werden können, wird hier eine – ansonsten nicht für die Arbeit verwendete – Transformation gezeigt, die wie die Skizze in Abbildung 12.1 arbeitet. Beide Beispiel-Metamodelle haben jeweils vier Elementtypen. Da alle Elementtypen beider Metamodelle bei der Transformation berücksichtigt werden, liegt die Abdeckung beider Metamodelle bei 100%.

Wie jedoch bei der Analyse in Abbildung 12.6 zu sehen ist, beziehen sich die Regeln `B2All` und `A2All` jeweils auf eine (im Vergleich mit der Gesamtzahl der Elementtypen im Metamodell) sehr große Anzahl verschiedener Elementtypen. Dies ist ein Hinweis darauf, dass es sich hier vermutlich nicht um semantikerhaltende Transformationen handelt, sondern nur um solche, die die Abdeckung der Metamodelle erhöhen sollen. Bei größeren Metamodellen fällt diese Häufung noch stärker ins Auge.

12.3 Automatisches Überprüfen der Transformation

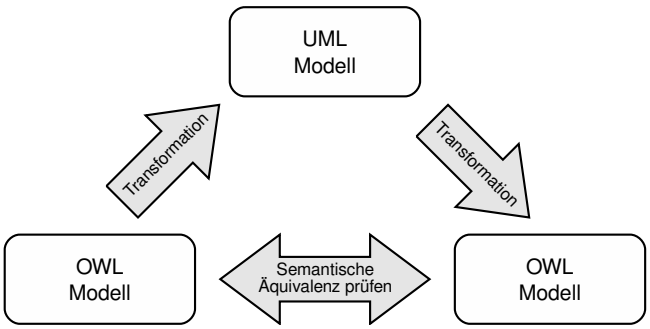


Abbildung 12.7. Vorgehen für die Überprüfung der Korrektheit der Transformationen.

Um eine Aussage darüber treffen zu können, ob die Transformationen für bestimmte Teile der Metamodelle korrekt formuliert sind, werden – wie in Abbildung 12.7 skizziert – beide Transformationen hintereinander ausgeführt und anschließend Eingabemodell und Ausgabemodell geeignet miteinander verglichen.

Dass ein OWL-Modell als Eingabemodell (und damit auch als Ausgabemodell) dient, hat den Vorteil, dass beim anschließenden Vergleich auf die im Ontology TS zur Verfügung stehenden Softwarewerkzeuge wie Reasoner zurückgegriffen werden kann. Im Folgenden wird dargestellt, wie ein "geeigneter" Vergleich aussehen kann:

Sei U das UML-Metamodell und O das OWL-Metamodell, u ein zu U konformes und o ein zu O konformes Modell. $\vec{T}_{UO}$ und $\vec{T}_{OU}$ seien die in den Kapiteln 7 bis 11 beschriebene Transformationen UML → OWL bzw. OWL → UML. Idealerweise sollte nun die Hintereinanderausführung der Transformationen $o_2 = \vec{T}_{UO}(\vec{T}_{OU}(o_1))$ eine Ontologie dergestalt erzeugen, dass o_1 und o_2 *semantisch äquivalent* sind.

12.3.1 "Semantisch äquivalent"

Es lässt sich beobachten, dass es strukturell unterschiedliche Modelle M_1 und M_2 gibt, bei denen jede Instanz m, die konform zu M_1 ist, ebenfalls konform zu M_2 ist. Die Modelle beschreiben also die gleiche (statische) Semantik.

Eine ähnliche Beobachtung lässt sich bei den zwei Beispielen aus dem XML TS in den Listings 12.3 und 12.4 machen: Die Struktur beider Modelle ist identisch. Beide Modelle haben denselben Informationsumfang, sie unterscheiden sich lediglich durch die Namensgebung der Elemente. Durch eine einfache Umbenennung (im Beispiel: a→r, b→s) kann jede Instanz, die konform zum ersten Modell ist, in eine zum zweiten Modell konforme Instanz überführt werden.

Insgesamt lässt sich also die Prüfung auf semantische Äquivalent auf die Frage reduzieren, ob zwischen beiden Ontologien jeweils ein *Total Ontology Mapping* existiert.

```
<schema>
  <element name="e" type="tns:eType" />
  <complexType name="eType">
    <sequence>
      <element name="a" type="tns:aType" />
      <element name="b" type="tns:bType" />
    </sequence>
  </complexType>
  <complexType name="aType">
    <sequence>
      <element name="x" minOccurs="3" maxOccurs="3" type="string" />
    </sequence>
  </complexType>
  <complexType name="bType">
    <sequence>
      <element name="x" minOccurs="2" maxOccurs="2" type="string" />
    </sequence>
  </complexType>
</schema>
```

Listing 12.3. Gekürztes Beispiel für ein XML-Schema.

```
<schema>
  <element name="e" type="tns:eType" />
  <complexType name="eType">
    <sequence>
      <element name="r" type="tns:rType" />
      <element name="s" type="tns:sType" />
    </sequence>
  </complexType>
  <complexType name="rType">
    <sequence>
      <element name="x" minOccurs="3" maxOccurs="3" type="string" />
    </sequence>
  </complexType>
  <complexType name="sType">
    <sequence>
      <element name="x" minOccurs="2" maxOccurs="2" type="string" />
    </sequence>
  </complexType>
</schema>
```

Listing 12.4. Gekürztes Beispiel für ein XML-Schema.

Total Ontology Mapping

Für den Begriff *Total Ontology Mapping* wird die Definition von Kalfoglou und Schorlemme verwendet.[229]

Eine Ontologie ist ein Paar $O = (S, A)$, wobei S die Signatur der Ontologie ist und A die Menge der Axiome. Die Signatur beschreibt das verwendete Vokabular der Ontologie. Die Menge der Axiome beschreibt, wie die Wörter aus S in Beziehung zueinander gesetzt werden.

Ein Total Ontology Mapping von einer Ontologie $O_1 = (S_1, A_1)$ zu einer Ontologie $O_2 = (S_2, A_2)$ ist ein Morphismus $f : S_1 \to S_2$, der die beiden Signaturen der Ontologien so aufeinander abbildet, dass $A_2 \models f(A_1)$, d.h. alle Interpretationen, die die Axiome von O_2 erfüllen, erfüllen auch die umbenannten Axiome von O_1.[230]

Bestimmung eines Total Ontology Mappings für OWL

In OWL bilden die Instanzen des Elementtyps *Entity* die *Signatur* der Ontologie. Die Elemente der Signatur unterteilen sich in die disjunkten Mengen *Class*, *ObjectProperty*, *DataProperty*, *AnnotationProperty*, *Datatype*, *NamedIndividual*. Die Signatur hat also die Form $S = (S_C, S_{OP}, S_{DP}, S_{DT}, S_I)$.[231] Da die Mengen disjunkt sind, muss nur innerhalb einer Menge nach Umbenennungen gesucht werden, was die Komplexität der Suche deutlich verringert.

Der Einfachheit halber wird angenommen, dass in S_1 nur Elemente auftreten, die auch in A_1 verwendet werden und in S_2 nur Elemente auftreten, die in A_2 verwendet werden. Ansonsten können nicht verwendete Elemente gestrichen werden, ohne die Aussage der Axiome zu verändern.

Weiterhin wird angekommen, dass die Bestandteile der Signaturen beider Ontologien dieselbe Größe haben: $|S_{X1}| = |S_{X2}|$, $X \in \{C, OP, DP, DT, I\}$. Sollte dies nicht der Fall sein, so wird der kleineren Menge eine entsprechende Menge bislang nicht verwendeter Elemente hinzugefügt.

Um eine übersichtliche Notation zu wahren, werden nur die Teilmenge S_{C1} und S_{C2} für die Klassen genauer betrachtet. Die übrigen vier Teilmengen S_{OP}, S_{DP},S_{DT} und S_I werden analog behandelt.

Für S_{C1} und S_{C2} wird eine beliebige Reihenfolge der Elemente festgelegt, so dass zwei geordnete Listen $S_{C1} = (c_1, \dots, c_n)$ und $S_{C2} = (d_1, \dots, d_n)$ entstehen. Nun wird für alle möglichen Permutationen $\sigma_C : N \to N$, $N = \{1, \dots n\}$ durchprobiert, ob sich jedes Axiom $a \in f(A_1)$ aus A_2 herleiten lässt, wobei $f : (S_{C1}, \dots) \to (S_{C2}, \dots)$ und $\sigma_C(c_i) = d_i \ \forall i \in \{1, \dots n\}$.

Anhand eines Beispiels soll die Arbeitsweise des Algorithmus verdeutlicht werden.

[229] Vgl. Kalfoglou, Y./Schorlemmer, M.: Ontology mapping: the state of the art, in: The Knowledge Engineering Review 18/1, 2003 ⟨URL: http://drops.dagstuhl.de/opus/volltexte/2005/40⟩.

[230] Vgl. a. a. O.

[231] Die Annotationen bleiben – wie weiter oben geschildert – unberücksichtigt.

```
1 Prefix(:=<urn:01#> )
2 Ontology(<urn:01>
3         Declaration(Class(:Author))
4         Declaration(Class(:Book))
5         Declaration(ObjectProperty(:hasAuthor))
6         ObjectPropertyRange(:hasAuthor :Author)
7         ObjectPropertyDomain(:hasAuthor :Book)
8 )
9
10 Prefix(:=<urn:02#> )
11 Ontology(<urn:02>
12         Declaration(Class(:Person))
13         Declaration(Class(:Buch))
14         Declaration(Class(:Autor))
15         Declaration(ObjectProperty(:wurdeGeschriebenVon))
16         SubClassOf( :Autor :Person )
17         ObjectPropertyDomain(:wurdeGeschriebenVon :Buch)
18         ObjectPropertyRange(:wurdeGeschriebenVon :Autor)
19 )
```

Listing 12.5. Zwei Beispiel-Ontologien in OWL Functional Style Syntax.

Die Ontologien in Listing 12.5 haben folgende Signaturen:

- $S_{C1} = \{\text{Author}, \text{Book}\}$, $S_{OP1} = \{\text{hasAuthor}\}$, $S_{DP1} = \emptyset$, $S_{DT1} = \emptyset$, $S_{I1} = \emptyset$
- $S_{C2} = \{\text{Person}, \text{Buch}, \text{Autor}\}$, $S_{OP2} = \{\text{wurdeGeschriebenVon}\}$, $S_{DP2} = \emptyset$, $S_{DT2} = \emptyset$, $S_{I2} = \emptyset$

Da S_{C2} ein Element mehr enthält als S_{C1}, muss S_{C1} entsprechend um ein Element erweitert werden: $S_{C1} = \{\text{Author}, \text{Book}, \text{X}\}$. S_{OP1} und S_{OP2} enthalten nur jeweils ein Element, daher existiert nur eine mögliche Permutation $\sigma_{OP1} = (1)$. Für σ_C müssen sechs mögliche Permutationen überprüft werden.

Zunächst soll das weitere Vorgehen bei einer nicht passenden Permutation $\sigma_C = id$ demonstriert werden. In diesem Fall ist

$f :=$ ((Author , Book, X), (hasAuthor),$\emptyset$, $\emptyset$, $\emptyset$)
$\rightarrow$ ((Person , Buch, Autor), (wurdeGeschriebenVon),$\emptyset$, $\emptyset$, $\emptyset$)

Es ist also zu prüfen, ob das Axiom

f(`ObjectPropertyRange(:hasAuthor :Author)`)
= `ObjectPropertyRange(:wurdeGeschriebenVon :Person)`

aus den Axiomen A_2 der zweiten Ontologie hergeleitet werden kann. Dies ist nicht der Fall, somit kann mit Hilfe dieser Permutation kein Total Ontology Mapping hergestellt werden.

Als nächste soll die durch $\sigma(1) := 3, \sigma(2) := 2, \sigma(3) := 1$ gegebene Permutation untersucht werden. In diesem Fall ist

$f :=$ ((X, Book, Author), (hasAuthor), $\emptyset$, $\emptyset$, $\emptyset$)
$\rightarrow$ ((Person , Buch, Autor), (wurdeGeschriebenVon), $\emptyset$, $\emptyset$, $\emptyset$)

Nun wird überprüft, ob sich die zwei umbenannten Axiome

f(`ObjectPropertyRange(:hasAuthor :Author)`)
= `ObjectPropertyRange(:wurdeGeschriebenVon :Autor)`
f(`ObjectPropertyDomain(:hasAuthor :Book)`)
= `ObjectPropertyDomain(:wurdeGeschriebenVon :Buch)`

aus den Axiomen A_2 der zweiten Ontologie herleiten lassen. Dies ist für beide Axiome der Fall. Somit ist ein Total Ontology Mapping $O_1 \rightarrow O_2$ gefunden. Dass die Axiome A_2 der zweiten Ontologie ein weiteres Axiom enthalten, ist hierfür nicht von Bedeutung. Wichtig wäre es erst bei der Suche nach einem Total Ontology Mapping $O_2 \rightarrow O_1$, das eben wegen dieses zusätzlichen Axioms – das auch nicht aus den anderen Axiomen von O_1 hergeleitet werden kann – nicht existiert.

Anmerkung: Ohne eine getrennte Betrachtung der fünf Teilmengen der Signatur wären $4! = 24$ Prüfungen notwendig gewesen, so sind es nur $3! \cdot 1! = 6$ Prüfungen.

12.3.2 Anwendung auf die Transformationsregeln

Die in den vorherigen Abschnitten beschriebene Verfahrensweise

1. Anwenden der Transformation $\vec{T}_{OU}$ auf die Eingabe-Ontologie o. Ergebnis ist $m = \vec{T}_{OU}(o)$.
2. Anwenden der Transformation $\vec{T}_{UO}$ auf das UML-Modell m. Ergebnis ist $o' = \vec{T}_{UO}(m)$.
3. Mit Hilfe des Algorithmus prüfen, ob ein Total Ontology Mapping zwischen o und o' existiert.
4. Mit Hilfe des Algorithmus prüfen, ob ein Total Ontology Mapping zwischen o' und o existiert.

kann nun auf Instanzen einzelner Meta-Elementtypen oder beliebiger Kombinationen von Meta-Elementtypen angewendet werden.

Entsprechend der theoretischen Überlegungen, bei welchen Teilen der Metamodelle eine verlustfreie Transformation in beide Richtungen möglich ist, wurden zahlreiche Test-Ontologien erstellt. Diese wurden wie beschrieben transformiert und erfolgreich auf semantische Äquivalenz getestet.

Kapitel 13

Anwendungsfall: Digitale Reichsstatistik

Um die Anwendung der Transformationsregeln in der Praxis zu demonstrieren, wird in diesem Kapitel ein Beispiel vorgestellt, das im Rahmen des ZBW-Projekts "Digitale Reichsstatistik" realisiert wurde.

Ziel dieser Betrachtung ist nicht, ein allgemeines oder für einen speziellen Anwendungszweck optimales Datenmodell aufzustellen. Vielmehr soll lediglich demonstriert werden, wie die Transformation genutzt werden kann, um von einem in einen anderen Technologieraum (→ 2.4 Technologieraum) zu übersetzen.

13.1 Übersicht zur Reichsstatistik

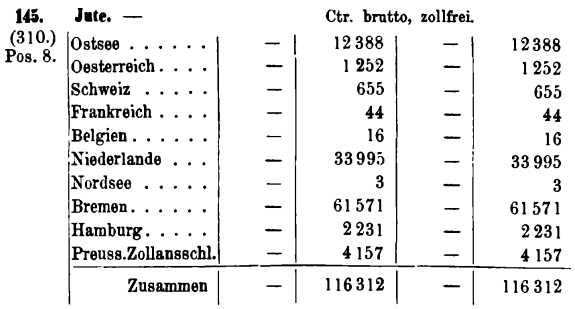
Bei der "alten Folge" der "Statistik des Deutschen Reichs" (1873–1883) handelt es sich um eine Veröffentlichung verschiedenster Statistiken der deutschen Staaten im Zeitraum der Jahre 1873 bis 1883. Die Veröffentlichung erfolgte in unsystematischer Reihenfolge und ist in 63 Bände zusammengefasst. Neben nur einmalig auftretenden statistischen Zusammenstellungen[232] und regelmäßig wiederholten Informationen zu "Waarenverkehr"[233], dem "Verkehr auf den deutschen Wasserstraßen"[234] oder der "Statistik der Seeschiffahrt"[235] sind die "Monatshefte zur Statistik des Deutschen Reichs" eines Jahres in jeweils einem Band zusammengefasst.

Dieses ca. 30.000 Seiten umfassende Druckwerk wird im Rahmen des DFG-finanzierten ZBW-Projekts "Digitale Reichsstatistik" gescannt und im Volltext erschlossen, sodass die Daten elektronisch als PDF-Dokumente und Dateien für Tabellenkalkulationsprogramme zur Verfügung stehen.

Informationen über die Wareneinfuhr machen den Großteil der Informationen der Statistik des Deutschen Reichs aus und werden in insgesamt 24 Bänden aufgeführt. Daher ist es sinnvoll, diese Art von Informationen für eine Modellierung zu betrachten, um so einen möglichst großen Teil der Reichsstatistik zu erfassen. Die Informationen sind in Tabellen mit unterschiedlichem Aggregationsgrad enthalten, wie in den Abbildungen 13.1 bis 13.3 zu sehen ist.

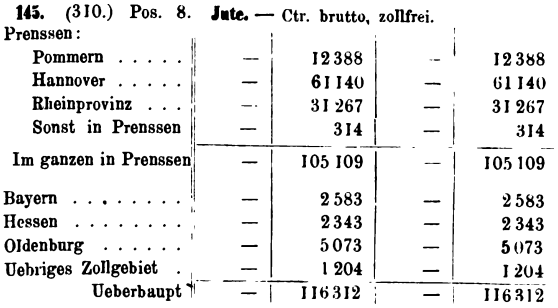
[232] Zum Beispiel zur Tabakwirtschaft in Band 42 oder zur Zollverwaltung in Band 6.
[233] 24 Bände: 3–5, 9–11, 16–17, 22–23, 27–28, 32–33, 39–40, 45–46, 49–50, 54–55 und 60–61.
[234] 11 Bände: 7, 12, 19, 24, 29, 36, 41, 47, 52, 58 und 63.
[235] 10 Bände: 13, 18, 21, 26, 31, 38, 44, 51, 56 und 62.

Abbildung 13.1. Diese Tabelle gibt Auskunft darüber, über welche Außengrenze Ware eingeführt wurde. Die ersten drei Spalten nach dem Namen der Außengrenze geben verschiedene Arten des Imports an, bei der letzten Spalte handelt es sich um eine Summe der vorherigen drei Spalten.[236]



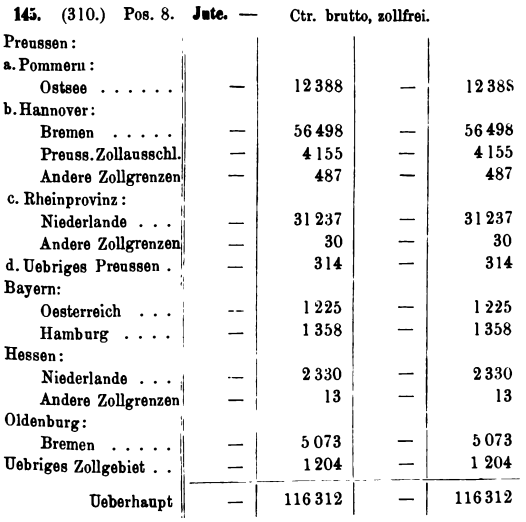
145. (310.) Pos. 8. Jute. —		Ctr. brutto, zollfrei.		
Ostsee	—	12388	—	12388
Oesterreich	—	1252	—	1252
Schweiz	—	655	—	655
Frankreich	—	44	—	44
Belgien	—	16	—	16
Niederlande	—	33995	—	33995
Nordsee	—	3	—	3
Bremen	—	61571	—	61571
Hamburg	—	2231	—	2231
Preuss. Zollansschl.	—	4157	—	4157
Zusammen	—	116312	—	116312

Abbildung 13.2. Diese Tabelle gibt Auskunft darüber, in welche Länder des Deutschen Reiches (bei Preußen zusätzlich in welche Provinz) Waren eingeführt wurden.[237]



145. (310.) Pos. 8. Jute. — Ctr. brutto, zollfrei.				
Preussen:				
Pommern	—	12388	—	12388
Hannover	—	61140	—	61140
Rheinprovinz	—	31267	—	31267
Sonst in Preussen	—	314	—	314
Im ganzen in Preussen	—	105109	—	105109
Bayern	—	2583	—	2583
Hessen	—	2343	—	2343
Oldenburg	—	5073	—	5073
Uebriges Zollgebiet	—	1204	—	1204
Ueberhaupt	—	116312	—	116312

Abbildung 13.3. Die Informationen der vorherigen beiden Tabellen werden zusätzlich sowohl nach Außengrenze als auch nach Land aufgeschlüsselt angegeben.[238]



145. (310.) Pos. 8. Jute. —		Ctr. brutto, zollfrei.		
Preussen:				
a. Pommern:				
Ostsee	—	12388	—	12388
b. Hannover:				
Bremen	—	56498	—	56498
Preuss. Zollausschl.	—	4155	—	4155
Andere Zollgrenzen	—	487	—	487
c. Rheinprovinz:				
Niederlande	—	31237	—	31237
Andere Zollgrenzen	—	30	—	30
d. Uebriges Preussen	—	314	—	314
Bayern:				
Oesterreich	—	1225	—	1225
Hamburg	—	1358	—	1358
Hessen:				
Niederlande	—	2330	—	2330
Andere Zollgrenzen	—	13	—	13
Oldenburg:				
Bremen	—	5073	—	5073
Uebriges Zollgebiet	—	1204	—	1204
Ueberhaupt	—	116312	—	116312

[236] Statistik des Deutschen Reiches, Band 3 [1873], Seite 38.
[237] Statistik des Deutschen Reiches, Band 3 [1873], Seite 113.
[238] Statistik des Deutschen Reiches, Band 3 [1873], Seite 221.

Den statistischen Angaben über die Menge der importierten Waren ist ein systematisches Warenverzeichnis vorangestellt. Ein Ausschnitt ist ein Abbildung 13.4 zu sehen. Dieses systematische Warenverzeichnis definiert eine Taxonomie der in der Statistik auftretenden Waren und Warengattungen. Auch für die heutige vom Bundesamt für Statistik geführte Außenhandelsstatistik gibt es ein entsprechendes "Warenverzeichnis für die Außenhandelsstatistik", siehe Abbildung 13.5. Um historische und aktuelle Daten vergleichen zu können, ist es erforderlich, die verwendeten Begriffe in Beziehung zueinander zu setzen. Werden Waren und Warengruppen in Form von Ontologien notiert, so ist dies möglich, indem die entsprechenden Klassen und Individuen in Beziehungen ($=, \neq, \subseteq, \supseteq$) gesetzt werden. Damit dieses Wissen genutzt werden kann, um bequem in den historischen Daten suchen zu können, müssen auch diese in Form einer Ontologie vorliegen.

Nummer des statistischen Waarenverzeichnisses.	Nummer des systematischen Waarenverzeichnisses.	Tarifs.	Waaren-Gattung mit Angabe des Maasstabes bezw. des Zollsatzes.
			8. Flachs und andere vegetabilische Spinnstoffe, mit Ausnahme der Baumwolle, roh, geröstet, gebrochen oder gehechelt.
142.	307.	8.	Flachs . . . . . Ctr. frei.
143.	308.	8.	Hanf . . . . . „
144.	309.	8.	Heede und Werg . . . . . „
145.	310.	8.	Jute . . . . . „
146.	313.	8.	Andere vegetabilische Spinnstoffe etc. . „
			9. Getreide und andere Erzeugnisse des Landbaues.
147.	1.	9 a.	Weizen . . . . . Ctr. frei.
148.	2.	9 a.	Roggen . . . . . „
149.	3.	9 a.	Gerste . . . . . „
150.	4.	9 a.	Hafer . . . . . „
151.	5.	9 a.	Mais . . . . . „
152.	9.	9 a.	Alles übrige Getreide . . . . . „
153.	12.	9 a.	Malz . . . . . „
154.	10.	9 a.	Hülsenfrüchte . . . . . „
155.	44.	9 b. 1.	Anis, Fenchel, Kümmel, Koriander . . . „
156.	45.	9 b. 2.	Senf, roher (Senfsaat) . . . . . „

Abbildung 13.4. Ausschnitt aus dem systematischen Warenverzeichnis.[239]

Juraplattenkalke	2530 90
Juraplattenkalke	6802
Justiertische für Druckereien	8442 30
Jute	5303 1
Juteabfälle	5303 90
Jutegarne	5307
Jutegewebe	5310
Jutesäcke gewebt	6305 10
Jutewerg	5303 90
K	
Kabel aus Aluminiumdraht	7614
Kabel aus Kupferdraht	7413 00

Abbildung 13.5. Ausschnitt aus dem alphabetischen Warenverzeichnis.[240]

[239] Statistik des Deutschen Reiches, Band 3 [1873], Seite 142.
[240] Statistisches Bundesamt, Warenverzeichnis für die Außenhandelsstatistik, 2012, Seite 740.

Im Folgenden sollen die beiden Möglichkeiten gezeigt werden, UML und OWL zur konzeptuellen Modellierung zu verwenden und die Modelle mit Hilfe der Transformationsregeln in den anderen Technologieraum abzubilden.

13.2 Modellierung mit UML

Eine Möglichkeit der Modellierung des Wareneingangs ist es, eine Klasse `Wareneingang` zu definieren und pro Zeile der statistischen Tabellen eine Instanz dieser Klasse zu erzeugen. Die Klasse besitzt Assoziationen zur importierten Ware (Instanz der Klasse `Ware`), zur Herkunft (Instanz der Klasse `Herkunft`) und zum Zielgebiet (Land bzw. Provinz – Instanz der abstrakten Klasse `Gebiet`). Weitere klassenabhängige Attribute enthalten die vier in den Tabellen angegebenen Mengenangaben ("auf Niederlagen", "unmittelbar", "von Niederlagen", Summe der vorherigen beiden). Um die Angaben mehrerer Jahre zu erfassen, wird ein klassenabhängiges Attribut `jahr` für das Jahr benötigt. Da pro Jahr und Ware drei Tabellen mit unterschiedlichem Aggregationsgrad verfügbar sind (siehe Abbildungen 13.1 bis 13.3) und auch jede Tabelle selbst wiederum Summen enthält, lassen sich Objekte zur Wareneinfuhr mit Hilfe einer rekursiven Assoziation in eine "ist Teil von"-Beziehung setzen. Die Klasse `Wareneingang` ist in Abbildung 13.6 dargestellt.

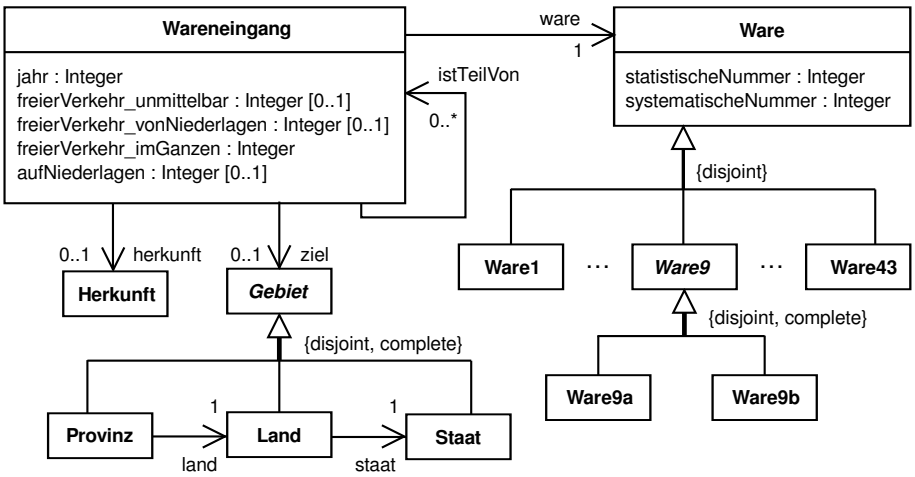


Abbildung 13.6. UML-Modell einer Beispielmodellierung für den Warenimport.

Wie oben geschildert, ist es sinnvoll, die Waren baumförmig anzuordnen. Dazu gibt es eine allgemeine Oberklasse `Ware`, die zwei klassenabhängige Attribute für statistische Nummer und systematische Nummer besitzt, siehe Abbildung 13.6. Für jede Warengruppe existiert eine Unterklasse, wobei manche von ihnen (wie z.B. die Klasse `Ware9` für die Warengruppe 9) abstrakt sind, da sie weiter strukturiert sind. Die Generalisierung der Waren kann nicht als vollständig gekennzeichnet werden, da die Reichsstatistik ein paar wenige Waren enthält, die nicht den 43 Warengruppen zugeordnet sind.

Bei der Modellierung der Zielgebiete (abstrakte Klasse `Gebiet`) ist zu beachten, dass die

Statistik meist Informationen nach Ländern gruppiert enthält. Für die preußischen Provinzen findet jedoch eine zusätzlich Unterteilung statt, wie in Abbildung 13.2 und 13.3 zu sehen ist. Außerdem gibt es Zahlen, die für den gesamten Staat "Deutsches Reich" gelten, wie die gesamte Tabelle in Abbildung 13.1 sowie die Summen in den Tabellen in Abbildung 13.3 und Abbildung 13.2.

Somit ergibt sich das in Abbildung 13.6 gezeigte Modell für den Warenimport. Auf dieses wird nun die vorgestellte Transformation UML → OWL angewendet. Das Ergebnis der automatischen Transformation ist im Anhang A zu finden, an dieser Stelle wird in Abbildung 13.7 nur eine – wiederum automatisch generierte – Visualisierung der Ontologie in der in 3.2.5 vorgestellten graphischen Syntax gezeigt.

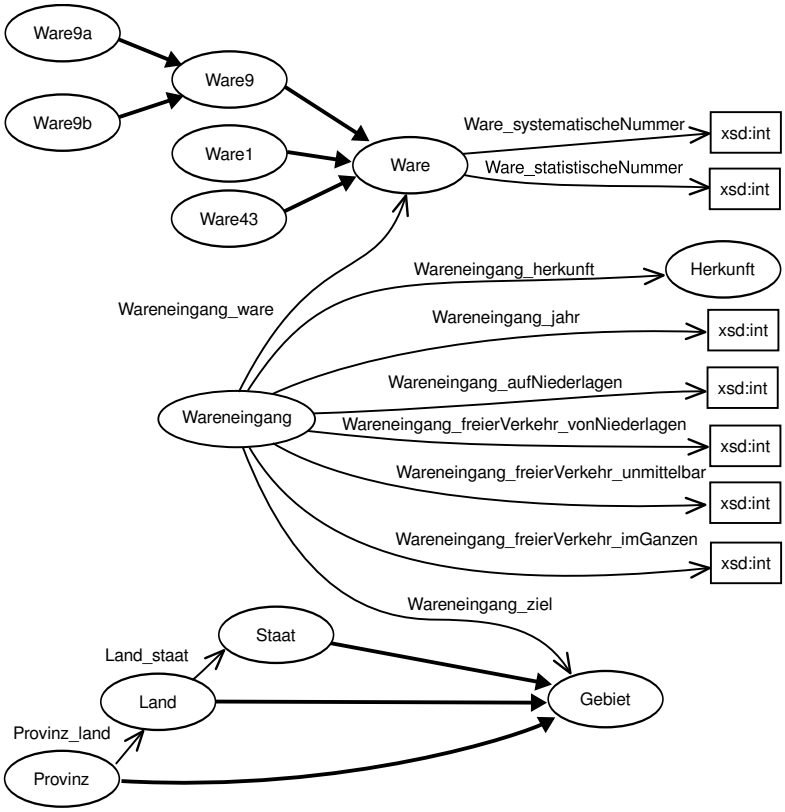


Abbildung 13.7. Ergebnis der Transformation UML → OWL für die Beispielmodellierung mit UML aus Abbildung 13.6.

Leicht zu erkennen ist, dass für jede UML-Klasse eine entsprechende OWL-Klasse mit dem gleichen Namen existiert. Die acht Generalisierungen sowie die Assoziationen zwischen den Klassen finden sich ebenfalls wieder. Zusammenfassend lässt sich sagen, dass in diesem Fall eine Transformation problemlos und nahezu[241] verlustfrei möglich ist.

[241] Wie in Abschnitt 7.4 diskutiert, kann die Information, dass eine Klasse abstrakt ist, nicht transformiert werden.

13.3 Modellierung mit OWL

In diesem Abschnitt wird ein Beispiel für die Modellierung des Warenimports mit Hilfe von OWL beschrieben. Ein Teil der Ontologie in konkreter Syntax ist in nachfolgendem Listing sowie in Abbildung 13.8 im Überblick – unter Verwendung der in 3.2.5 vorgestellten graphischen Syntax – zu sehen.

```
Prefix(:=<urn:DRS#>)
Ontology(<urn:DRS>
    Declaration( Class( :Wareneingang ) )

    Declaration( Class( :Gebiet ) )
    Declaration( Class( :Provinz ) )
    Declaration( Class( :Land ) )
    Declaration( Class( :Staat ) )
    DisjointUnion( :Gebiet :Provinz :Land :Staat )

    Declaration( ObjectProperty( :land ) )
    ObjectPropertyDomain( :land :Provinz )
    ObjectPropertyRange( :land :Land )
    SubClassOf( :Provinz ObjectExactCardinality( 1 :land :Land ) )

    Declaration( ObjectProperty( :staat ) )
    ObjectPropertyDomain( :staat :Land )
    ObjectPropertyRange( :staat :Staat )
    SubClassOf( :Land ObjectExactCardinality( 1 :staat :Staat ) )

    Declaration( Class( :Ware ) )
    Declaration( ObjectProperty( :ware ) )
    ObjectPropertyDomain( :ware :Wareneingang )
    ObjectPropertyRange( :ware :Ware )
    SubClassOf( :Wareneingang ObjectExactCardinality( 1 :ware :Ware ) )

    Declaration( Class( :Herkunft ) )
    Declaration( ObjectProperty( :herkunft ) )
    ObjectPropertyDomain( :herkunft :Wareneingang )
    ObjectPropertyRange( :herkunft :Herkunft )

    Declaration( ObjectProperty( :ziel ) )
    ObjectPropertyDomain( :ziel :Wareneingang )
    ObjectPropertyRange( :ziel :Gebiet )

    Declaration( DataProperty( :aufNiederlagen ) )
    DataPropertyDomain( :aufNiederlagen :Wareneingang )
    DataPropertyRange( :aufNiederlagen xsd:int )
    FunctionalDataProperty( :aufNiederlagen )

    Declaration( DataProperty( :freierVerkehr_unmittelbar ) )
    DataPropertyDomain( :freierVerkehr_unmittelbar :Wareneingang )
    DataPropertyRange( :freierVerkehr_unmittelbar xsd:int )
    FunctionalDataProperty( :freierVerkehr_unmittelbar )

    Declaration( DataProperty( :freierVerkehr_vonNiederlagen ) )
    DataPropertyDomain( :freierVerkehr_vonNiederlagen :Wareneingang )
    DataPropertyRange( :freierVerkehr_vonNiederlagen xsd:int )
    FunctionalDataProperty( :freierVerkehr_vonNiederlagen )

    Declaration( DataProperty( :freierVerkehr_imGanzen ) )
```

```
52     DataPropertyDomain( :freierVerkehr_imGanzen :Wareneingang )
53     DataPropertyRange( :freierVerkehr_imGanzen xsd:int )
54     SubClassOf( :Wareneingang DataExactCardinality( 1 :freierVerkehr_imGanzen xsd:int) )
55
56     Declaration( DataProperty( :jahr ) )
57     DataPropertyDomain( :jahr :Wareneingang )
58     DataPropertyRange( :jahr xsd:int )
59     SubClassOf( :Wareneingang DataExactCardinality( 1 :jahr xsd:int ) )
60
61     Declaration( ObjectProperty( :istTeilVon ) )
62     ObjectPropertyDomain( :istTeilVon :Wareneingang )
63     ObjectPropertyRange( :istTeilVon :Wareneingang )
64
65     DisjointClasses( :Wareneingang :Herkunft :Gebiet :Ware )
66
67     Declaration( DataProperty( :statistischeNummer ) )
68     DataPropertyDomain( :statistischeNummer :Ware )
69     DataPropertyRange( :statistischeNummer xsd:int )
70     SubClassOf( :Ware DataExactCardinality( 1 :statistischeNummer xsd:int) )
71
72     Declaration( DataProperty( :systematischeNummer ) )
73     DataPropertyDomain( :systematischeNummer :Ware )
74     DataPropertyRange( :systematischeNummer xsd:int )
75     SubClassOf( :Ware DataExactCardinality( 1 :systematischeNummer xsd:int) )
76
77     Declaration( Class( :Ware1 )
78     SubClassOf( :Ware1 :Ware )
79
80     Declaration( Class( :Ware9 )
81     Declaration( Class( :Ware9a )
82     Declaration( Class( :Ware9b )
83     SubClassOf( :Ware9 :Ware )
84     DisjointUnion( :Ware9 :Ware9a :Ware9b )
85
86     Declaration( Class( :Ware43 )
87     SubClassOf( :Ware43 :Ware )
88
89     DisjointClasses( :Ware1 :Ware9 :Ware43 )
90 )
```

Listing 13.1. Teil der Ontologie einer Beispielmodellierung für den Warenimport.

Jeweils eine Zeile der statistischen Tabellen wird als ein Individuum repräsentiert. Zur Klassifikation dieser Individuen wird eine Klasse `Wareneingang` benötigt (Zeile 3).

Die Zielgebiete der Waren sind in unterschiedlicher Genauigkeit (Provinz, Land, Staat) angegeben. Daher ist es sinnvoll, eine Oberklasse `Gebiet` als `DisjointUnion` zu definieren (Zeilen 5–9). Für die Beziehungen zwischen den drei Unterklassen werden zwei Object Properties definiert. Da die Angaben obligatorisch sind, werden entsprechende `ObjectExact-Cardinality`-CE verwendet (Zeilen 11–19).

Über Object Properties steht die Klasse `Wareneingang` obligatorisch mit der importierten Ware (`Ware`), optional zur Herkunft (`Herkunft`) und optional zum Zielgebiet (`Gebiet`) in Beziehung (Zeilen 21–34).

Vier Data Properties nehmen die Werte der vier verschiedenen angegebenen Mengenangaben (“auf Niederlagen”, “unmittelbar”, “von Niederlagen”, Summe der vorherigen

beiden) auf. Der Definitionsbereich ist für alle vier `Wareneingang`. Da jeder Eintrag jeweils höchstens eine dieser vier Mengenangaben besitzt, werden die Data Properties als funktional gekennzeichnet. Die Summe ist obligatorisch, daher wird hier wieder eine `DataExactCardinality`-CE (Zeilen 35–54) eingesetzt.

Damit Angaben mehrerer Jahre erfasst werden können, wird (in den Zeilen 56–59) eine Data Property mit Definitionsbereich `Wareneingang` definiert. Als Zielbereich der Data Property werden die ganzen Zahlen (`xsd:int`) festgelegt. Da sich jede Angabe auf genau ein Jahr bezieht, wird für diese Data Property eine entsprechende `DataExactCardinality`-CE eingefügt.

Da pro Jahr und Ware drei Tabellen mit unterschiedlichem Aggregationsgrad verfügbar sind und auch jede Tabelle selbst wiederum Summen enthält, lassen sich Objekte zur Wareneinfuhr mit Hilfe der `istTeilVon` Object Property in Beziehung setzen (Zeilen 61–63).

Schließlich wird definiert, dass es sich bei den zuvor definierten Klassen um verschiedene Klassen (und nicht eine Klasse mit verschiedenen Namen) handelt (Zeile 65).

Für die statistische und systematische Nummer einer Ware werden zwei Data Properties definiert (Zeilen 67–75). Jede Warengruppe wird als eine Klasse modelliert, die Unterklasse von `Ware` ist (Zeilen 77–87). Alle Klassen der Warengruppen sind untereinander disjunkt, was durch ein entsprechendes `DisjointClasses`-Axiom in Zeile 89 definiert wird. Da die Reichsstatistik ein paar wenige Waren enthält, die nicht den 43 Warengruppen zugeordnet sind, kann in diesem Fall keine `DisjointUnion` verwendet werden. Bei den beiden Untergruppen von `Ware9` ist dies jedoch wiederum möglich (Zeilen 80–84).

Bei einem Vergleich der in Listing 13.1 definierten und in Abbildung 13.8 als Übersicht dargestellten Ontologie mit dem Transformationergebnis UML → OWL in Abbildung 13.7 ist leicht eine große Ähnlichkeit zu erkennen. Es ist daher auch zu erwarten, dass auch das Ergebnis der Transformation OWL → UML dem UML-Modell in Abbildung 13.6 ähneln wird.

Das in Abbildung 13.9 abgebildete UML-Klassendiagramm zeigt das Ergebnis der Anwendung der Transformationsregeln auf die oben definierte Ontologie. Alle in der Ontologie definierten Klassen und Beziehungen finden sich auch in diesem Modell wieder. Die Kardinalitätsbeschränkungen sind mit den beabsichtigten Werten vorhanden. Im Vergleich mit dem Klassendiagramm in Abbildung 13.8 fällt auf, dass die Klassen `Gebiet` und `Ware9` nicht abstrakt sind – das ist in der Ontologie nicht angegeben. Ebenfalls ist zu bemerken, dass bei den *IsA*-Beziehungen zu `Ware` keine Instanz des UML-Elementtyps *GeneralizationSet* verwendet wird. Das liegt daran, dass – anders als bei den Oberklassen `Gebiet` und `Ware9` kein `DisjointUnion`-Axiom zum Einsatz kommt, da die Generalisierung nicht vollständig ist. Die unabhängigen `SubClassOf`-Axiome werden daher zu unabhängigen *Generalization*-Instanzen transformiert.

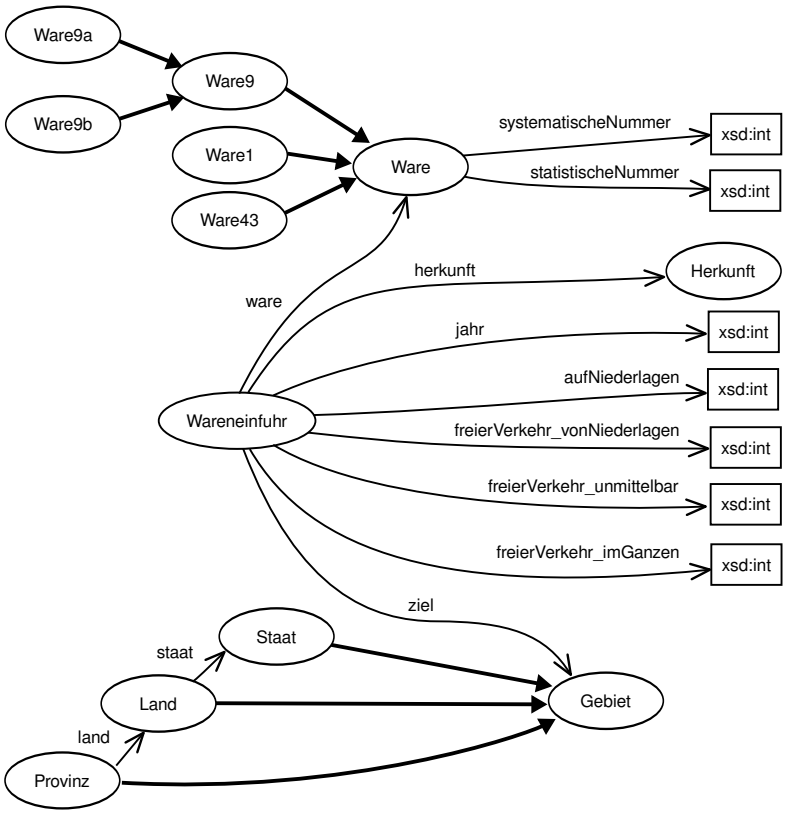


Abbildung 13.8. Visualisierung der in Teilen in Listing 13.1 gezeigten Ontologie mit der Modellierung des Warenimports.

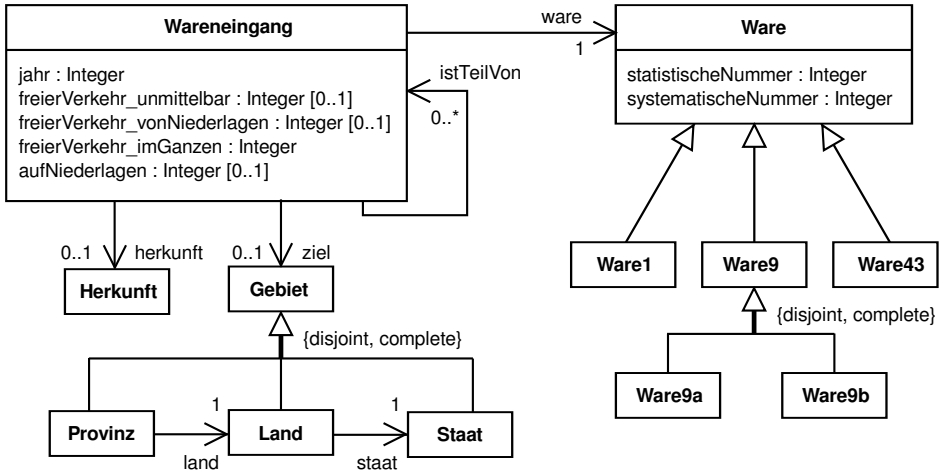


Abbildung 13.9. Ergebnis der Transformation OWL → UML für die Beispielmodellierung mit Hilfe von OWL.

13.4 Vorteilung der Nutzung von OWL

In diesem Abschnitt sollen einige Vorteile aufgezeigt werden, die die Nutzung von OWL für diesen Anwendungsfall bringt.

Beim Zugriff auf die Daten ermöglicht einem die Nutzung einer Ontologie eine leichte Suche innerhalb des Datenbestandes, wobei die Beziehungen der Waren und Warengruppen untereinander ausgenutzt werden können. So ließe sich beispielsweise eine Antwort auf die relativ komplexe Frage ermitteln, ob es Länder gibt, bei denen es im Zeitraum der Reichstatistik von 1873 bis 1883 einen kontinuierlichen Anstieg des Getreideimports aus Russland gegeben hat. Anzumerken ist, dass die Reichsstatistik keinen summierten Wert für "Getreide" aufführt, sondern dieser sich aus den Werten der einzelnen Waren der Warengruppe 9 (="Getreide") zusammensetzt.

Neben der Suche im Datenbestand besteht bei ausreichend strenger Datenmodellierung die Möglichkeit, die Konsistenz der Daten zu überprüfen und so Fehler zu finden. Dies schließt mehrere Arten von Fehlern mit ein, deren Detektion im Folgenden näher erläutert werden soll:

- Fehler in der Modellierung/Terminologie
- Fehler bei der Dateneingabe
- Fehler in den historischen Daten

Fehler in der Terminologie

```
1  FunctionalDataProperty( :statistischeNummer )
2
3  Declaration(Class( :Ware7 ))
4  SubClassOf( :Ware7 :Ware )
5  EquivalentClasses( :Ware8
6     DataAllValuesFrom( :statistischeNummer
7        DatatypeRestriction( xsd:int
8           xsd:minInclusive "122"^^xsd:int
9           xsd:maxInclusive "141"^^xsd:int  )
10    ) )
11
12 Declaration(Class( :Ware8 ))
13 SubClassOf( :Ware8 :Ware )
14 EquivalentClasses( :Ware8
15    DataAllValuesFrom( :statistischeNummer
16       DatatypeRestriction( xsd:int
17          xsd:minInclusive "142"^^xsd:int
18          xsd:maxInclusive "146"^^xsd:int  )
19    ) )
```

Listing 13.2. Beispiel für einen Fehler in der Definition der Warengruppen.

Als Beispiel für die Aufdeckung eines Fehlers in der Terminologie soll eine in Listing 13.2 gezeigte Definition der Warengruppen dienen. Wie in der Modellierung zuvor ist jede

Warengruppe über eine eigene Klasse definiert, die Unterklasse einer allgemeinen Klasse für `Ware` ist. Im Beispiel wird die Klasse mit Hilfe von hinreichenden Bedingungen definiert, die ausnutzen, dass die Waren einer Klasse mit fortlaufenden Nummern (Data Property "statistischeNummer") im Warenverzeichnis versehen sind. In Zeile 5 hat sich nun ein Tippfehler eingeschlichen, der fälschlicherweise die Klasse "Ware8" mit den eigentlich für "Ware7" geltenden Zahlen verbindet. Eine händische Fehlersuche wäre sehr aufwendig, mit Hilfe eines Reasoners lässt sich jedoch schnell eine Inkonsistenz aufdecken:

```
Ware8 equivalentTo statistischeNummer only int[>= "142"^^int, <= "146"^^int]
Functional statistischeNummer
Ware8 equivalentTo statistischeNummer only int[<= "141"^^int, >= "122"^^int]
```

Fehler bei der Dateneingabe

Die Nutzung der Datentypen von OWL macht es möglich, einige Fehler bei der Dateneingabe aufzudecken. Ist der Zielbereich für die vier Mengenangaben eines Wareneingangs auf positive Ganzzahlen eingeschränkt (`DataPropertyRange`-Axiom), so können Falsch- oder Leereingaben für diese Angabe detektiert werden.

```
1 Declaration( NamedIndividual( :B2-S113-145-1 ))
2 ClassAssertion( :Wareneingang :B2-S113-145-1 )
3 ObjectPropertyAssertion( :ware :B2-S113-145-1 :Jute )
4 DataPropertyAssertion( :jahr :B2-S113-145-1 "1872"^^xsd:positiveInteger )
5 DataPropertyAssertion( :feierVerkehr_unmittelbar :B2-S113-145-1 "l2388"^^xsd:positiveInteger )
6 DataPropertyAssertion( :feierVerkehr_imGanzen    :B2-S113-145-1 ""^^xsd:positiveInteger )
7 ObjectPropertyAssertion( :istTeilVon :B2-S113-145-1 :B2-S113-145-10 )
```

Listing 13.3. Beispiel für einen Fehler bei der Eingabe der Assertions.

Listing 13.3 zeigt zwei solcher Fehler: In Zeile 5 ist – für einen menschlichen Korrekturleser schwer zu erkennen – statt einer Ziffer ein Buchstabe enthalten. In Zeile 6 ist der Wert komplett leer geblieben. Mit Hilfe eines Reasoners ist es wieder einfach, diese Fehler zu finden:

```
B2-S113-145-1 freierVerkehr_imGanzen "()"^^positiveInteger
B2-S113-145-1 freierVerkehr_imGanzen "l2388"^^positiveInteger
```

Fehler in den historischen Daten

Eine dritte Kategorie von Fehlern, nämlich solche, die bereits als Fehler in den historischen Daten selbst enthalten sind und auch bei einer fehlerfreien Transkription in der Ontologie enthalten sind, lässt sich mit Hilfe einer Regelsprache (hier: Semantic Web Rule Language (SWRL)) aufspüren. So wird mit Hilfe der SWRL-Regel

$$\begin{aligned} & Wareneingang(?w) \\ \wedge\, & freierVerkehr\_unmittelbar(?w, ?f1) \\ \wedge\, & freierVerkehr\_vonNiederlagen(?w, ?f2) \\ \wedge\, & add(?sum, ?f1, ?f2) \\ \rightarrow\, & freierVerkehr\_imGanzen(?w, ?sum) \end{aligned}$$

für jede Instanz der Klasse `Wareneingang` eine Data Property `freierVerkehr_imGanzen` eingefügt, die als Wert die Summe der Werte der Properties `freierVerkehr_unmittelbar` und `freierVerkehr_vonNiederlagen` zugeordnet bekommt. Da es sich bei `freierVerkehr_imGanzen` um eine funktionale Property handelt, würde ein von der bereits vorhandenen Angabe abweichender Wert zu einer Inkonsistenz führen.

Kapitel 14

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein systematischer Ansatz für eine automatische Transformation konzeptueller Modelle zwischen dem Model-Driven Architecture Technology Space (MDA TS) und dem Ontology Engineering Technology Space (Ontology TS) vorgestellt. Dazu wurde – anders als bei bisherigen Arbeiten – eine Herangehensweise gewählt, die von konkreter Syntax bzw. XML-Serialisierung abstrahiert und auf Ebene der Metamodelle von UML und OWL arbeitet. So ließ sich zugleich unabhängig von einzelnen Beispielmodellen zeigen, welche Modellelemente transformiert werden können und welche nicht.

Für eine Vielzahl von Modellierungskonzepten, die sich in die vier Gruppen

- Elementtypen,
- Beziehungstypen,
- Beschränkungen und
- Strukturierung

einteilen lassen, wurde eine formale Beschreibung gegeben und untersucht, wie sich das jeweilige Konzept mit UML bzw. OWL repräsentieren lässt. In den Fällen, in denen unter Beibehaltung der Semantik eine Transformation durchführen lässt, wurden die Transformationen ausführlich sowohl als Freitext als auch formal in Form von QVT-R-Transformationsregeln beschrieben. Die QVT-R-Regeln beziehen sich dabei nur auf Elemente der Metamodelle, so dass die Transformation unabhängig von einzelnen Modellen (die Instanzen dieser Metamodelle sind) ist.

Dadurch, dass die Transformationen in beide Richtungen vollständig und formal in QVT-R aufgeschrieben vorliegen, lassen sich die Überlegungen nachvollziehen und bei Bedarf schnell an einzelnen Beispielen testen. In dieser vollständigen und formalen Beschreibung unterscheidet sich diese Arbeit von vorherigen Arbeiten, die entweder gar keine formale Beschreibung oder nur Fragmente[242] angeben.

Es hat sich gezeigt, dass sich in UML beschriebene Datenmodelle relativ gut mit Ontologien darstellen lassen. Insbesondere wenn dabei gewisse einschränkende Regeln eingehalten werden, wie sie z.B. die ISO 19100 Normenfamilie vorgibt, lässt sich die Semantik des Datenmodells gut übertragen. Als problematisch zu nennen sind hier lediglich die in UML mögliche Einschränkung der Sichtbarkeit von Modellelementen, abstrakte Klassen, bestimmte

[242] Siehe dazu auch die Kritik an Kapitel 16 des ODM in Abschnitt 5.2.2.

Arten der Generalisierung (überschneidungsfrei aber nicht vollständig), Aggregation und Komposition (die bis auf kleine Ausnahmen als gewöhnliche Beziehungstypen behandelt werden) und die Erweiterung mittels Stereotypen.

Schon der unterschiedliche Umfang der Metamodelle lässt erahnen, dass OWL wesentlich komplexere Möglichkeiten bei der Modellierung bietet.[243] Die Transformation allgemeiner Ontologien in UML-Datenmodelle ist daher nicht immer möglich. Insbesondere problematisch ist die Definition von Elementtypen mit Hilfe verschachtelter Class Expressions sowie durch hinreichende Bedingungen. Aber selbst in diesen Fällen ist oft eine Transformation möglich, z.B. bei Kardinalitätsbeschränkungen, die als Obertyp auftreten.

OWL-Konstrukte wie Komplementbildung und globale Properties lassen sich im Allgemeinen nicht transformieren. Unter der speziellen Voraussetzung, dass ein einziger Elementtyp als Obertyp aller anderen Elementtypen festgelegt wurde, ist jedoch auch in diesem Fall eine Transformation möglich.

Das Anwendungsbeispiel in Kapitel 13 demonstriert, wie gut sich die Transformationsregeln auf konkrete Modelle anwenden lassen. Unter der Annahme, dass es sich bei dem vorgestellten Beispiel um ein typisches handelt, ist festzustellen, dass der Technologieraum relativ frei gewählt werden kann, da leicht und ohne große Verluste von einem zum anderen Technologieraum gewechselt werden kann.

Ausblick

Die informal aufgeschriebene Ausdeutung für XSD ließe sich formalisieren, um so besser bewerten zu können, welche Modellierungskonzepte sich auch in den XML TS übertragen lassen. Mit Hilfe des XML Schema Component Model lassen sich die in der GML-Spezifikation vorgeschlagenen Regeln auch als QVT-R Transformationen auf Metamodell-Ebene umsetzen. Dies wurde im Umfeld dieser Arbeit – jedoch hier nicht dargestellt – für die Transformation UML → XSD prototypisch durchgeführt. Schon bei dieser prototypischen Umsetzung sind einige Ungereimtheiten in den vorgeschlagenen Regeln aufgefallen. Eine formal beschriebene Transformation deckt diese leicht auf.

In vielen Fälle lassen sich durch die Verwendung von OCL-Ausdrücken in UML-Datenmodellen Modellierungskonzepte im Allgemeinen und OWL-Konstrukte, wie die Definition von Elementtypen durch hinreichende Bedingungen, im Speziellen umsetzen. Da für OCL eine MOF-kompatible abstrakte Syntax vorhanden ist, kann auch diese Transformation – wie die in dieser Arbeit beschriebenen Transformationen – auf Metamodell-Ebene durchgeführt werden. Durch die zusätzliche Verwendung von OCL ergeben sich jedoch einige Schwierigkeiten:

- Die Transformation OWL → UML ist relativ einfach, da zu einem OWL-Konstrukt ein vordefinierten OCL-Ausdruck ins Datenmodell eingefügt werden kann. In UML-Modellen können

[243] Aus der Vielzahl der Element- und Beziehungstypen und der daraus resultierenden größeren Komplexität folgt jedoch nicht automatisch eine größere Ausdrucksstärke. Das Metamodell könnte lediglich kleinteiliger strukturiert sein.

hingegen beliebige OCL-Ausdrücke auftreten. Diese müssten bei einer Transformation UML → OWL bearbeitet werden.

- ▻ Es ist nicht klar, ob alle atomaren OCL-Ausdrücke mit OWL darstellbar sind. Eventuell lässt sich auf Seiten von OWL auf Regelsprachen (z.B. die SWRL mit ihren *built-ins*) zurückgreifen. Diese machen dann aber wiederum die Transformation OWL → UML komplizierter.
- ▻ Der Sprachumfang von OCL ist sehr groß, durch Verschachteln von Ausdrücken lassen sich beliebig komplizierte OCL-Ausdrücke erzeugen. Dies macht zum einen die Transformation komplex, zum anderen ist unklar, ob sich diese komplexen Ausdrücke in einer Ontologie wiedergeben lassen.

Es lässt sich spekulieren, dass in Zukunft konzeptuelle Modelle nur noch mit OWL formuliert werden. Insbesondere in Fällen, bei denen es einschränkende Regeln für die Verwendung von UML-Konstrukten gibt, wie bei der ISO 19100 Normenfamilie, scheint dies plausibel. Es gibt bereits Arbeiten, die sich damit beschäftigen, die Vorteile, welche sich durch die Nutzung von UML ergeben, auch auf OWL zu übertragen. Auch für Anwendungen bei denen bisher vor allem XSD verwendet wird, wird nach OWL-Alternativen gesucht:

- ▻ Wichtig für die Software-Entwicklung ist die Möglichkeit, automatisch Programmcode aus UML-Datenmodellen generieren zu können. Ähnliche Überlegungen gibt es auch für Modelle, die in Form von OWL-Ontologien aufgeschrieben sind. Alcaraz Calero et al. etwa geben einen Überblick über entsprechende Ansätze.[244] In diesem Artikel wird auch das von XSD bekannte *language binding*, d.h. das Generieren von Programmcode aus XML-Schemata sowie das automatische Serialisieren und Deserialisieren von XML-Dokumenten, für OWL thematisiert.
- ▻ Es gibt Ansätze, OWL im Bereich der Maschine-zu-Maschine-Kommunikation einzusetzen. So beschreiben Liebig et al. eine "OWLLink" genannte Schnittstelle die OWL 2-Syntaxen zum Austausch von Nachrichten verwendet.[245]

Im Bereich der Verständlichkeit ist UML derzeit überlegen. Gäbe es eine entsprechend intuitive grafische Syntax für OWL mit einer Auswahl von Software-Werkzeugen zum Umgang mit dieser Syntax, so würde dies sicherlich zum verstärkten Einsatz von OWL bei der Erstellung konzeptueller Modelle beitragen.

[244] Alcaraz Calero, J. M. et al.: Towards an Architecture to Bind the Java and OWL Languages, in: Journal of Research and Practice in Information Technology 44/1, Sydney 2012.
[245] Liebig, T. et al.: OWLlink, in: Semantic Web – Interoperability, Usability, Applicability 2/1, Amsterdam 2011.

Anhang A

Ergebnis der Transformation UML → OWL für das Beispiel aus Kapitel 13

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(:=<urn:DRS#>)
Ontology( <urn:DRS>

    Declaration( Class( <urn:DRS#Gebiet> ) )
    DisjointUnion( <urn:DRS#Gebiet> <urn:DRS#Provinz> <urn:DRS#Land> <urn:DRS#Staat> )

    Declaration( Class( <urn:DRS#Herkunft> ) )

    Declaration( Class( <urn:DRS#Land> ) )
    SubClassOf( <urn:DRS#Land> <urn:DRS#Gebiet>)
    SubClassOf( <urn:DRS#Land>
                ObjectExactCardinality( 1 <urn:DRS#Land_staat> <urn:DRS#Staat>) )

    Declaration( Class( <urn:DRS#Provinz> ) )
    SubClassOf( <urn:DRS#Provinz> <urn:DRS#Gebiet>)
    SubClassOf( <urn:DRS#Provinz>
                ObjectExactCardinality( 1 <urn:DRS#Provinz_land> <urn:DRS#Land>) )

    Declaration( Class( <urn:DRS#Staat> ) )
    SubClassOf( <urn:DRS#Staat> <urn:DRS#Gebiet>)

    Declaration( Class( <urn:DRS#Ware> ) )
    SubClassOf( <urn:DRS#Ware>
                DataExactCardinality( 1
                                      <urn:DRS#Ware_statistischeNummer>
                                      <http://www.w3.org/2001/XMLSchema#int>) )
    SubClassOf( <urn:DRS#Ware>
                DataExactCardinality( 1
                                      <urn:DRS#Ware_systematischeNummer>
                                      <http://www.w3.org/2001/XMLSchema#int>)  )

    Declaration( Class( <urn:DRS#Ware1> ) )
    SubClassOf( <urn:DRS#Ware1> <urn:DRS#Ware>)

    Declaration( Class( <urn:DRS#Ware9> ) )
    SubClassOf( <urn:DRS#Ware9> <urn:DRS#Ware>)

    Declaration( Class( <urn:DRS#Ware9a> ) )
    SubClassOf( <urn:DRS#Ware9a> <urn:DRS#Ware9>)

    Declaration( Class( <urn:DRS#Ware9b> ) )
    SubClassOf( <urn:DRS#Ware9b> <urn:DRS#Ware9>)

    Declaration( Class( <urn:DRS#Ware43> ) )
    SubClassOf( <urn:DRS#Ware43> <urn:DRS#Ware>)
```

```
47    Declaration( Class( <urn:DRS#Wareneingang> ) )
48    SubClassOf( <urn:DRS#Wareneingang>
49                DataExactCardinality( 1
50                                      <urn:DRS#Wareneingang_jahr>
51                                      <http://www.w3.org/2001/XMLSchema#int>) )
52    SubClassOf( <urn:DRS#Wareneingang>
53                DataExactCardinality( 1
54                                      <urn:DRS#Wareneingang_freierVerkehr_imGanzen>
55                                      <http://www.w3.org/2001/XMLSchema#int>) )
56    SubClassOf( <urn:DRS#Wareneingang>
57                DataMaxCardinality( 1
58                                    <urn:DRS#Wareneingang_aufNiederlagen>
59                                    <http://www.w3.org/2001/XMLSchema#int>) )
60    SubClassOf( <urn:DRS#Wareneingang>
61                DataMaxCardinality( 1
62                                    <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
63                                    <http://www.w3.org/2001/XMLSchema#int>) )
64    SubClassOf( <urn:DRS#Wareneingang>
65                DataMaxCardinality( 1
66                                    <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen>
67                                    <http://www.w3.org/2001/XMLSchema#int>) )
68    SubClassOf( <urn:DRS#Wareneingang>
69                ObjectExactCardinality( 1 <urn:DRS#Wareneingang_ware> <urn:DRS#Ware>) )
70    SubClassOf( <urn:DRS#Wareneingang>
71                ObjectMinCardinality( 0 <urn:DRS#Wareneingang_herkunft> <urn:DRS#Herkunft>) )
72    SubClassOf( <urn:DRS#Wareneingang>
73                ObjectMaxCardinality( 1 <urn:DRS#Wareneingang_herkunft> <urn:DRS#Herkunft>) )
74    SubClassOf( <urn:DRS#Wareneingang>
75                ObjectMinCardinality( 0 <urn:DRS#Wareneingang_ziel> <urn:DRS#Gebiet>) )
76    SubClassOf( <urn:DRS#Wareneingang>
77                ObjectMaxCardinality( 1 <urn:DRS#Wareneingang_ziel> <urn:DRS#Gebiet>) )
78    SubClassOf( <urn:DRS#Wareneingang>
79                ObjectMinCardinality( 0 <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Wareneingang>) )
80
81    Declaration( DataProperty( <urn:DRS#Wareneingang_aufNiederlagen> ) )
82    DataPropertyDomain( <urn:DRS#Wareneingang_aufNiederlagen> <urn:DRS#Wareneingang>)
83    DataPropertyRange( <urn:DRS#Wareneingang_aufNiederlagen>
84                       <http://www.w3.org/2001/XMLSchema#int>)
85
86    Declaration( DataProperty( <urn:DRS#Wareneingang_freierVerkehr_imGanzen> ) )
87    DataPropertyDomain( <urn:DRS#Wareneingang_freierVerkehr_imGanzen> <urn:DRS#Wareneingang>)
88    DataPropertyRange( <urn:DRS#Wareneingang_freierVerkehr_imGanzen>
89                       <http://www.w3.org/2001/XMLSchema#int>)
90
91    Declaration( DataProperty( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar> ) )
92    DataPropertyDomain( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar> <urn:DRS#Wareneingang>)
93    DataPropertyRange( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
94                       <http://www.w3.org/2001/XMLSchema#int>)
95
96    Declaration( DataProperty( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen> ) )
97    DataPropertyDomain( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen> <urn:DRS#Wareneingang>)
98    DataPropertyRange( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen>
99                       <http://www.w3.org/2001/XMLSchema#int>)
100
101   Declaration( DataProperty( <urn:DRS#Wareneingang_jahr> ) )
102   DataPropertyDomain( <urn:DRS#Wareneingang_jahr> <urn:DRS#Wareneingang>)
103   DataPropertyRange( <urn:DRS#Wareneingang_jahr>
104                      <http://www.w3.org/2001/XMLSchema#int>)
105
```

```
106     Declaration( DataProperty( <urn:DRS#Ware_statistischeNummer> ) )
107     DataPropertyDomain( <urn:DRS#Ware_statistischeNummer> <urn:DRS#Ware>)
108     DataPropertyRange( <urn:DRS#Ware_statistischeNummer>
109                        <http://www.w3.org/2001/XMLSchema#int>)
110
111     Declaration( DataProperty( <urn:DRS#Ware_systematischeNummer> ) )
112     DataPropertyDomain( <urn:DRS#Ware_systematischeNummer> <urn:DRS#Ware>)
113     DataPropertyRange( <urn:DRS#Ware_systematischeNummer> <http://www.w3.org/2001/XMLSchema#int>)
114
115     Declaration( ObjectProperty( <urn:DRS#Land_staat> ) )
116     ObjectPropertyDomain( <urn:DRS#Land_staat> <urn:DRS#Land>)
117     ObjectPropertyRange( <urn:DRS#Land_staat> <urn:DRS#Staat>)
118     FunctionalObjectProperty( <urn:DRS#Land_staat>)
119
120     Declaration( ObjectProperty( <urn:DRS#Provinz_land> ) )
121     ObjectPropertyDomain( <urn:DRS#Provinz_land> <urn:DRS#Provinz>)
122     ObjectPropertyRange( <urn:DRS#Provinz_land> <urn:DRS#Land>)
123     FunctionalObjectProperty( <urn:DRS#Provinz_land>)
124
125     Declaration( ObjectProperty( <urn:DRS#Wareneingang_herkunft> ) )
126     ObjectPropertyDomain( <urn:DRS#Wareneingang_herkunft> <urn:DRS#Wareneingang>)
127     ObjectPropertyRange( <urn:DRS#Wareneingang_herkunft> <urn:DRS#Herkunft>)
128     FunctionalObjectProperty( <urn:DRS#Wareneingang_herkunft>)
129
130     Declaration( ObjectProperty( <urn:DRS#Wareneingang_istTeilVon> ) )
131     ObjectPropertyDomain( <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Wareneingang>)
132     ObjectPropertyRange( <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Wareneingang>)
133
134     Declaration( ObjectProperty( <urn:DRS#Wareneingang_ware> ) )
135     ObjectPropertyDomain( <urn:DRS#Wareneingang_ware> <urn:DRS#Wareneingang>)
136     ObjectPropertyRange( <urn:DRS#Wareneingang_ware> <urn:DRS#Ware>)
137     FunctionalObjectProperty( <urn:DRS#Wareneingang_ware>)
138
139     Declaration( ObjectProperty( <urn:DRS#Wareneingang_ziel> ) )
140     ObjectPropertyDomain( <urn:DRS#Wareneingang_ziel> <urn:DRS#Wareneingang>)
141     ObjectPropertyRange( <urn:DRS#Wareneingang_ziel> <urn:DRS#Gebiet>)
142     FunctionalObjectProperty( <urn:DRS#Wareneingang_ziel>)
143
144     DisjointClasses( <urn:DRS#Gebiet> <urn:DRS#Ware1> )
145     DisjointClasses( <urn:DRS#Gebiet> <urn:DRS#Ware43> )
146     DisjointClasses( <urn:DRS#Gebiet> <urn:DRS#Ware9> )
147     DisjointClasses( <urn:DRS#Gebiet> <urn:DRS#Ware9a> )
148     DisjointClasses( <urn:DRS#Gebiet> <urn:DRS#Ware9b> )
149     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Gebiet> )
150     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Land> )
151     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Provinz> )
152     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Staat> )
153     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Ware1> )
154     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Ware43> )
155     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Ware9> )
156     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Ware9a> )
157     DisjointClasses( <urn:DRS#Herkunft> <urn:DRS#Ware9b> )
158     DisjointClasses( <urn:DRS#Land> <urn:DRS#Ware1> )
159     DisjointClasses( <urn:DRS#Land> <urn:DRS#Ware43> )
160     DisjointClasses( <urn:DRS#Land> <urn:DRS#Ware9> )
161     DisjointClasses( <urn:DRS#Land> <urn:DRS#Ware9a> )
162     DisjointClasses( <urn:DRS#Land> <urn:DRS#Ware9b> )
163     DisjointClasses( <urn:DRS#Provinz> <urn:DRS#Ware1> )
164     DisjointClasses( <urn:DRS#Provinz> <urn:DRS#Ware43> )
```

```
165     DisjointClasses( <urn:DRS#Provinz> <urn:DRS#Ware9> )
166     DisjointClasses( <urn:DRS#Provinz> <urn:DRS#Ware9a> )
167     DisjointClasses( <urn:DRS#Provinz> <urn:DRS#Ware9b> )
168     DisjointClasses( <urn:DRS#Staat> <urn:DRS#Ware1> )
169     DisjointClasses( <urn:DRS#Staat> <urn:DRS#Ware43> )
170     DisjointClasses( <urn:DRS#Staat> <urn:DRS#Ware9> )
171     DisjointClasses( <urn:DRS#Staat> <urn:DRS#Ware9a> )
172     DisjointClasses( <urn:DRS#Staat> <urn:DRS#Ware9b> )
173     DisjointClasses( <urn:DRS#Ware1> <urn:DRS#Ware9> <urn:DRS#Ware43> )
174     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Gebiet> )
175     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Herkunft> )
176     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Land> )
177     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Provinz> )
178     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Staat> )
179     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Ware> )
180     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Ware1> )
181     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Ware43> )
182     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Ware9> )
183     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Ware9a> )
184     DisjointClasses( <urn:DRS#Wareneingang> <urn:DRS#Ware9b> )
185     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Gebiet> )
186     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Herkunft> )
187     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Land> )
188     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Provinz> )
189     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Staat> )
190     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Ware9a> )
191     DisjointClasses( <urn:DRS#Ware> <urn:DRS#Ware9b> )
192
193     DisjointDataProperties( <urn:DRS#Wareneingang_aufNiederlagen>
194                             <urn:DRS#Ware_statistischeNummer> )
195     DisjointDataProperties( <urn:DRS#Wareneingang_aufNiederlagen>
196                             <urn:DRS#Ware_systematischeNummer> )
197     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_imGanzen>
198                              <urn:DRS#Wareneingang_aufNiederlagen> )
199     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_imGanzen>
200                              <urn:DRS#Ware_statistischeNummer> )
201     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_imGanzen>
202                              <urn:DRS#Ware_systematischeNummer> )
203     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
204                              <urn:DRS#Wareneingang_aufNiederlagen> )
205     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
206                              <urn:DRS#Wareneingang_freierVerkehr_imGanzen> )
207     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
208                              <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen> )
209     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
210                              <urn:DRS#Ware_statistischeNummer> )
211     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_unmittelbar>
212                              <urn:DRS#Ware_systematischeNummer> )
213     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen>
214                              <urn:DRS#Wareneingang_aufNiederlagen> )
215     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen>
216                              <urn:DRS#Wareneingang_freierVerkehr_imGanzen> )
217     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen>
218                              <urn:DRS#Ware_statistischeNummer> )
219     DisjointDataProperties( <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen>
220                              <urn:DRS#Ware_systematischeNummer> )
221     DisjointDataProperties( <urn:DRS#Wareneingang_jahr>
222                              <urn:DRS#Wareneingang_aufNiederlagen> )
223
```

```
    DisjointDataProperties( <urn:DRS#Wareneingang_jahr>
                            <urn:DRS#Wareneingang_freierVerkehr_imGanzen> )
    DisjointDataProperties( <urn:DRS#Wareneingang_jahr>
                            <urn:DRS#Wareneingang_freierVerkehr_unmittelbar> )
    DisjointDataProperties( <urn:DRS#Wareneingang_jahr>
                            <urn:DRS#Wareneingang_freierVerkehr_vonNiederlagen> )
    DisjointDataProperties( <urn:DRS#Wareneingang_jahr> <urn:DRS#Ware_statistischeNummer> )
    DisjointDataProperties( <urn:DRS#Wareneingang_jahr> <urn:DRS#Ware_systematischeNummer> )
    DisjointDataProperties( <urn:DRS#Ware_statistischeNummer> <urn:DRS#Ware_systematischeNummer> )
    DisjointObjectProperties( <urn:DRS#Provinz_land> <urn:DRS#Land_staat> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_herkunft> <urn:DRS#Land_staat> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_herkunft> <urn:DRS#Provinz_land> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_herkunft> <urn:DRS#Wareneingang_ziel> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Land_staat> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Provinz_land> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Wareneingang_herkunft> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_istTeilVon> <urn:DRS#Wareneingang_ziel> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ware> <urn:DRS#Land_staat> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ware> <urn:DRS#Provinz_land> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ware> <urn:DRS#Wareneingang_herkunft> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ware> <urn:DRS#Wareneingang_istTeilVon> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ware> <urn:DRS#Wareneingang_ziel> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ziel> <urn:DRS#Land_staat> )
    DisjointObjectProperties( <urn:DRS#Wareneingang_ziel> <urn:DRS#Provinz_land> )
)
```

Anhang B

Abkürzungen

CE Class Expression

CWA Closed World Assumption

DAML DARPA Agent Markup Language

DL Description Logics

GFM General Feature Model

GML Geography Mark-Up Language

IRI Internationalized Resource Identifier

MDA TS Model-Driven Architecture Technology Space

MOF Meta-Object Facility

OCL Object Constraint Language

ODM Ontology Definition Model

Ontology TS Ontology Engineering Technology Space

OWA Open World Assumption

OWL Web Ontology Language

OMG Object Management Group

QVT Meta Object Facility (MOF) 2.0 Query/View/Transformation

QVT-R QVT Relations

RDF Resource Description Framework

RDFS Resource Description Framework Schema

SWRL Semantic Web Rule Language

UML Unified Modeling Language

UNA Unique Names Assumption

URI Uniform Resource Identifier

W3C World Wide Web Consortium

XMI XML Metadata Interchange

XML Extensible Markup Language

XML TS XML Technology Space

XSLT Extensible Stylesheet Language Transformations

XSD XML Schema Definition Language

ZBW Deutsche Zentralbibliothek für Wirtschaftswissenschaften

Literaturverzeichnis

Alcaraz Calero, J. M. et al.: Towards an Architecture to Bind the Java and OWL Languages, in: Journal of Research and Practice in Information Technology 44/1, Sydney 2012, S. 17–41

Angele, J./Kifer, M./Lausen, G.: Ontologies in F-logic, in: **Staat, S./Studer, R. (Hrsg.):** Handbook on Ontologies, Berlin/Heidelberg 2009, S. 45–70

Atkinson, C./Gutheil, M./Kiko, K.: On the relationship of ontologies and models, in: Proceedings of the 2nd Workshop on Meta-Modelling and Ontologies (WoMM06), Bonn 2006 ⟨URL: `http://subs.emis.de/LNI/Proceedings/Proceedings96/GI-Proceedings-96-3.pdf`⟩

Atkinson, C./Kuhne, T.: Model-driven development: A Metamodeling Foundation, in: IEEE Software 20/5, Los Alamitos 2003, S. 36–41

Baader, F. et al. (Hrsg.): The Description Logic Handbook: Theory, Implementation and Applications, Cambridge 2003

Baclawski et al.: Extending UML to Support Ontology Engineering for the Semantic Web, in: **Gogolla, M./Kobryn, C. (Hrsg.):** The Unified Modeling Language: Modeling Languages, Concepts, and Tools; 4th International Conference; Proceeedings / «UML» 2001, Berlin/Heidelberg/New York/Barcelona/Hong Kong/London/Milan/Paris/Tokyo 2001, S. 342–360

Balzert, H.: Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering, Band 1, Heidelberg, 3. Auflage 2009

Bārzdinš, Jānis et al.: UML Style Graphical Notation and Editor for OWL 2, in: Perspectives in Business Informatics Research. 9th International Conference, BIR 2010, Rostock Germany, September 29–October 1, 2010. Proceedings, Berlin/Heidelberg 2010, S. 102–114

Bedini, I/G, Gardarin/Nguyen, B: Deriving Ontologies from XML Schema, in: **Cépaduès (Hrsg.):** Entrepôts de données et analyse en ligne - EDA'08, Toulouse 2008, S. 3–17

Berardi, D./Calvanese, D./Giacomo, G. de: Reasoning on UML Class Diagrams is EXPTIME-hard, in: **Calvanese, D./De Giacomo, G./Franconi, E. (Hrsg.):** Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rom 2003 ⟨URL: `http://ceur-ws.org/Vol-81/berardi-1.pdf`⟩, S. 28–37

Bohring, H./Auer, S.: Mapping XML to OWL ontologies, in: **Jantke, K. P./Fähnrich, K.-P./ Wittig, W. S. (Hrsg.):** Leipziger Informatik-Tage, Leipzig 2005, S. 147–156

Brockmans, S. et al.: A Model Driven Approach for Building OWL DL and OWL Full Ontologies, in: **Cruz, I. et al. (Hrsg.):** The Semantic Web – ISWC 2006, Heidelberg 2006, S. 187–200

Brockmans, S. et al.: A Metamodel and UML Profile for Rule-extended OWL DL Ontologies, in: **Cruz, I. et al. (Hrsg.):** The Semantic Web – ISWC 2006, Heidelberg 2006, S. 303–316

Brockmans, S. et al.: Visual Modeling of OWL DL Ontologies Using UML, in: **McIlraith, Sheila A./Plexousakis, Dimitris/Harmelen, Frank van (Hrsg.):** The Semantic Web – ISWC 2004, Heidelberg 2004 ⟨URL: http://www.aifb.uni-karlsruhe.de/WBS/sbr/publications/iswc04_20sbr.pdf⟩, S. 198–213

Cadoli, M. et al.: Finite Model Reasoning on UML Class Diagrams Via Constraint Programming, in: **Basili, R./Pazienza, M. T. (Hrsg.):** AI*IA '07: Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, Berlin/Heidelberg 2007

Cranefield, S.: Networked Knowledge Representation and Exchange using UML and RDF, in: Journal of Digital information 1/8, Austin 2001

Cranefield, S./Purvis, M.: UML as an ontology modelling language, in: The Information Science Discussion Paper Series 99/01, Dunedin 1999

Czarnecki, K./Helsen, S.: Feature-based Survey of Model Transformation Approaches, in: IBM Systems Journal 45/3, Riverton 2006, S. 621–645 ⟨URL: http://gsd.uwaterloo.ca/sites/default/files/ibm06.pdf⟩

Djurić, D./Gašević, D./Devedžić, V.: Ontology modeling and MDA, in: Journal of Object Technology 4/1, 2005, S. 109–128

Djurić, D.: MDA-based ontology infrastructure, in: Computer Science and Information Systems 1/1, Novi Sad 2004, S. 91–116

Djurić, D. et al.: A UML Profile for OWL Ontologies, in: Model Driven Architecture. European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers, Berlin/Heidelberg 2005 ⟨URL: http://www.springerlink.com/content/49yb6365gymtryfg/⟩, S. 204–219

Eisenhut, C./Kutzner, T.: Vergleichende Untersuchungen zur Modellierung und Modelltransformation in der Region Bodensee im Kontext von INSPIRE, München 2010

Falkovych, K.: Ontology Extraction from UML Diagram, Amsterdam 2002

Falkovych, K./Sabou, M./Stuckenschmidt, H.: UML for the Semantic Web: Transformation-based approaches, in: Knowledge Transformation for the Semantic Web 95, 2003, S. 92–107

Flynn, J.: VisioOWL, 2006 ⟨URL: http://mysite.verizon.net/jflynn12/VisioOWL/VisioOWL.htm⟩ – Zugriff am 9. Februar 2012

Gašević, D. et al.: Converting UML to OWL Ontologies, in: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York 2004, S. 488–489

Grose, T.J./Doney, G.C./Brodsky, S.A.: Mastering XMI: Java Programming with XMI, XML, and UML, New York 2002

Gruber, T. R.: Ontology, in: **Liu, L./Özsu, M. T. (Hrsg.):** Encyclopedia of Database Systems, Berlin/Heidelberg 2009

Hart, L et al.: OWL Full and UML 2.0 Compared, 2004 ⟨URL: http://www.omg.org/docs/ontology/04-03-01.pdf⟩

Hesse, W./Mayr, H.C.: Modellierung in der Softwaretechnik: eine Bestandsaufnahme, in: Informatik-Spektrum 31/5, Berlin/Heidelberg 2008, S. 377–393

Hitzler, P. et al.: Semantic Web, Berlin/Heidelberg 2008

Horridge, M./Bechhofer, S.: The OWL API: A Java API for OWL ontologies, in: **Hoekstra, R./Patel-Schneider, P. (Hrsg.):** Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009), 2009 ⟨URL: http://ceur-ws.org/Vol-529/owled2009_submission_29.pdf⟩

Höglund, S. et al.: Representing and Validating Metamodels using the Web Ontology Language OWL 2. TUCS Technical Report No. 973, Turku 2010 ⟨URL: http://tucs.fi/publications/attachment.php?fname=TR973.full.pdf⟩

ISO: Norm ISO 19101 Geographic information – Reference model, Genf 2002

ISO: Norm ISO 19109 Geographic information – Rules for application schema, Genf 2005

ISO: Norm ISO/TS 19103:2005 Geographic information – Conceptual schema language, Genf 2005

Jörke, Jan: Alternativen für die konzeptuelle Modellierung von GML, Diplomarbeit Christian-Albrechts-Universität, Kiel 2010

Kalfoglou, Y./Schorlemmer, M.: Ontology mapping: the state of the art, in: The Knowledge Engineering Review 18/1, 2003, S. 1–31 ⟨URL: http://drops.dagstuhl.de/opus/volltexte/2005/40⟩

Kapova, L. et al.: Evaluating Maintainability with Code Metrics for Model-to-Model Transformations, in: **Heineman, G. T./Kofron, J./Plasil, F. (Hrsg.):** Research into Practice – Reality and Gaps, Berlin/Heidelberg 2010, S. 151–166

Katifori, A. et al.: Ontology Visualization Methods—A Survey, in: ACM Computing Surveys (CSUR) 39/4, New York 2007 ⟨URL: http://disi.unitn.it/~p2p/RelatedWork/Matching/a10-katifori.pdf⟩

Kiko, Kilian/Atkinson, C.: A Detailed Comparison of UML and OWL, Mannheim 2008 ⟨URL: http://madoc.bib.uni-mannheim.de/madoc/volltexte/2008/1898/pdf/TR2008_004.pdf⟩

Kurtev, I./Bézivin, J./Aksit, M.: Technological spaces: An initial appraisal, in: International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, 30 Oct - 1 Nov 2002, Irvine, USA, 2002 ⟨URL: http://eprints.eemcs.utwente.nl/10206/01/0363TechnologicalSpaces.pdf⟩, S. 1–6

Leinhos, S.: OWL Ontologieextraktion und -modellierung auf der Basis von UML Klassendiagrammen, Diplomarbeit Universität der Bundeswehr München, München 2006

Liebig, T. et al.: OWLlink, in: Semantic Web – Interoperability, Usability, Applicability 2/1, Amsterdam 2011, S. 23–32

Luttenberger, N.: Grammars for XML Documents - XML Schema, Part 1 – Vorlesung "XML in Communication Systems"(WS 2011/12), Kiel 2011

Milanović, M. et al.: Towards Sharing Rules Between OWL/SWRL and UML/OCL, in: Electronic Communications of the EASST Volume 5, 2006

Milanović, M. et al.: On Interchanging Between OWL/SWRL and UML/OCL, in: Proceedings of 6th Workshop on OCL for (Meta-) Models in Multiple Application Domains (OCLApps), 2006, S. 81–95

Milvich, M.: Ein Semantisches Web für die Universitätsbibliothek Heidelberg, Masterthesis Fachhochschule Karlsruhe, Karslruhe 2005

Nolte, S.: QVT-Relations Language, Berlin/Heidelberg 2009

OGC: Geography Markup Language (GML) Encoding Standard 3.2.1, 2007 ⟨URL: http://www.opengeospatial.org/standards/gml⟩

Olivé, A.: Conceptual Modeling of Information Systems, Berlin/Heidelberg/New York 2007

OMG: MOF 2 XMI Mapping Specification, 2001 ⟨URL: http://www.omg.org/spec/XMI/⟩

OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0, 2008 ⟨URL: http://www.omg.org/spec/QVT/1.0/PDF/⟩

OMG: Ontology Definition Metamodel, 2009 ⟨URL: http://www.omg.org/spec/ODM/1.0/⟩

OMG: Unified Modeling Language, Infrastructure Version 2.4, 2011 ⟨URL: http://www.omg.org/spec/UML/2.4/Infrastructure⟩

OMG: Unified Modeling Language, Superstructure Version 2.4, 2011 ⟨URL: http://www.omg.org/spec/UML/2.4/Superstructure⟩

Pan, J.Z./Horrocks, I.: Metamodeling architecture of web ontology languages, in: **Cruz, I.F. et al. (Hrsg.):** The Emerging Semantic Web: selected papers from the first Semantic Web Working Symposium, Amsterdam 2002, S. 21

Parreiras, F.S./Staab, S./Winter, A.: On Marrying Ontological and Metamodeling Technical Spaces, in: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York 2007, S. 439–448

Pörnbacher, C.: Modellgetriebene Entwicklung der Steuerungssoftware automatisierter Fertigungssysteme, München 2011

Rector, A. et al.: OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns, in: **Carbonell, J. G./Siekmann, J. (Hrsg.):** Engineering Knowledge in the Age of the Semantic Web, Berlin/Heidelberg 2004, S. 63–81

Reussner, R./Hasselbring, W. (Hrsg.): Handbuch der Software-Architektur, Heidelberg, 2. Auflage 2009

Russell, Stuart J./Norvig, Peter: Artificial Intelligence: A Modern Approach, Upper Saddle River, 2. Auflage 2003

Schreiber, G.: A UML Presentation Syntax for OWL Lite, 2002 ⟨URL: http://www.swi.psy.uva.nl/usr/Schreiber/docs/owl-uml/owl-uml.html⟩ – Zugriff am 28. Januar 2013

Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions, in: **Engels, G. et al. (Hrsg.):** Model Driven Engineering Languages and Systems. 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings. Berlin/Heidelberg 2007, S. 1–15

Tantau, Till: Syntax versus Semantik – Vorlesung zu Logik für Informatiker (WS 2006/07), Lübeck 2006

Tschirner, S./Scherp, A./Staab, S.: Semantic access to INSPIRE, in: **Grütter, R. et al. (Hrsg.):** Terra Cognita 2011. Proceedings of the Terra Cognita Workshop on Foundations, Technologies and Applications of the Geospatial Web, 2011 ⟨URL: http://ceur-ws.org/Vol-798/proceedings.pdf⟩, S. 75–87

W3C: XML Schema Part 1: Structures, 2004 ⟨URL: http://www.w3.org/TR/xmlschema-1/⟩

W3C: OWL 2 Web Ontology Language Direct Semantics, 2012 ⟨URL: http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/⟩

W3C: OWL 2 Web Ontology Language Document Overview, 2012 ⟨URL: http://www.w3.org/TR/2012/REC-owl2-overview-20121211/⟩

W3C: OWL 2 Web Ontology Language RDF-Based Semantics, 2012 ⟨URL: http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/⟩

W3C: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, 2012 ⟨URL: http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/⟩

Wahler, M. et al.: Efficient Analysis of Pattern-Based Constraint Specifications, in: Software and Systems Modeling 9/2, Heidelberg 2010, S. 225–255 ⟨URL: http://dx.doi.org/10.1007/s10270-009-0123-6⟩

Zedlitz, J./Jörke, J./Luttenberger, N.: From UML to OWL 2, in: **Lukose, D./Ahmad, A. R./Suliman, A. (Hrsg.):** Proceedings of Knowledge Technology Week 2011, Berlin/Heidelberg 2012, S. 154–163