JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Multi-Level Modellierung und OWL

Implementierung von Mapping und Werkzeug-Unterstützung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

MAG.RER.SOC.OEC.

im Diplomstudium

WIRTSCHAFTSINFORMATIK

Angefertigt am *Institut für Wirtschaftsinformatik – Data & Knowledge Engineering*

Betreuung / Begutachter:

o.Univ.-Prof. Dr. Michael Schrefl

Mitbetreuung:

Mag. Dr. Bernd Neumayr

Eingereicht von:

Joachim Huber

Linz, im *September 2011.*

Johannes Kepler Universität Linz
A-4040 Linz, Altenberger Str. 69, Internet: http://www.jku.at, DVR 0093696

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, September 2011

……..

Joachim Huber

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die durch ihre fachliche und persönliche Unterstützung zur Entstehung dieser Diplomarbeit beigetragen haben.

Zuerst möchte ich meinen Eltern dafür danken, dass sie durch ihre finanzielle Unterstützung mein Studium überhaupt erst möglich gemacht haben. Aber auch für ihre persönliche Unterstützung und Motivation, vor allem in schwierigen Zeiten des Studiums, möchte ich mich vielmals bedanken. Daher soll diese Arbeit auch ihnen gewidmet sein.

Herrn o.Univ.-Prof. Dr. Michael Schrefl und meinem Betreuer Mag. Dr. Bernd Neumayr möchte ich für die Unterstützung bei der Erstellung dieser Arbeit danken. Mein Betreuer trug durch seine fachliche und engagierte Betreuung sowie seinen Ratschlägen wesentlich zu dieser Arbeit bei.

Schließlich möchte ich mich noch bei meinem Bruder, meinen Freunden und ehemaligen Studienkollegen für ihre Unterstützung im Laufe meines Studiums bedanken.

Kurzfassung

In den letzten Jahren rückte der Bereich der Multi-Level Modellierung immer mehr in den Fokus des Interesses. In diesem Zusammenhang haben Neumayr und Schrefl Multi-Level Objects (M-Objects) und Multi-Level Relationships (M-Relationships) eingeführt und deren Mapping nach OWL 2 (Description Logic $\mathcal{SROIQ}$) definiert. Weiters geben sie ein Beispiel, wie ein derartiges Multi-Level Modell aussehen kann.

Mein Kollege Alois Diwold hat in seiner Diplomarbeit ein Modellierungswerkzeug für M-Objects und M-Relationships auf Basis von Protégé Frames bereits umgesetzt. Meine Diplomarbeit knüpft an seine Realisierung nahtlos an, indem ich ein Export-Plugin für Protégé erstellt habe, das den Mapping Algorithmus von Neumayr und Schrefl implementiert und die in Protégé modellierten M-Objects und M-Relationships nach OWL transformiert. Das ist auch gleichzeitig die erste große Aufgabenstellung dieser Arbeit.

Der zweite zentrale Punkt dieser Arbeit befasst sich mit der Ermittlung eines geeigneten Reasoners für in OWL abgebildete M-Objects und M-Relationships. In diesem Zusammenhang wurden von mir Reasoning – Performance Studies durchgeführt, wobei ich mich dabei auf die OWL-Reasoner Pellet, Fact++ und HermiT konzentriert habe. Leider musste ich in diesem Zusammenhang bald feststellen, dass diese Reasoner allesamt im Moment noch nicht in der Lage sind, die im Mapping Algorithmus enthaltenen Integrity Constraints (IC), eine kürzlich vorgestellte Erweiterung zu OWL, richtig zu interpretieren.

Zum Teil werden IC Verletzungen zwar bereits richtig erkannt, was aber auf die Tatsache zurückzuführen ist, dass diese Axiome auch unter Open World Assumption so interpretiert werden. In diesem Zusammenhang bleibt abzuwarten, ob um ICs erweitertes OWL von OWL-Reasonern künftig unterstützt wird. Da die Reasoner alle dieselben Ergebnisse lieferten, konnte ich nur aufgrund des Laufzeitverhaltens auf einen geeigneten Reasoner schließen.

Abstract

The area of multilevel modeling has become very popular in recent years. In this context, Neumayr and Schrefl introduced Multi-Level Objects (M-Objects) and Multi-Level Relationships (M-Relationships) and defined their mapping to OWL 2 (Description Logic $\mathcal{SROIQ}$). Furthermore they show how such a multi-level model might look like.

My colleague Alois Diwold already implemented a modeling tool for M-Objects and M-Relationships based on Protégé Frames in his thesis. My thesis builds on his realization by creating an export plugin for Protégé that implements the mapping algorithm of Neumayr and Schrefl and transforms M-Objects and M-Relationships modeled in Protégé to OWL. This is also the first major task of this work.

The second key point of this work deals with the identification of a suitable reasoner for M-Objects and M-Relationships modeled in OWL. In this context I performed Reasoning – Performance Studies, where I focused on the OWL reasoner Pellet, Fact++ and HermiT. Unfortunately, I had to recognize soon, that none of them is yet in the position to interpret correctly the Integrity Constraints (IC), a recently announced extension to OWL, contained in the mapping algorithm.

Some of the ICs were interpreted correctly, but this is due to the fact that these axioms can also be interpreted under the Open World Assumption. In this context it remains to be seen whether ICs are supported by OWL reasoners in the future. Since the evaluated reasoners provided all the same results, I was only able to choose an appropriate reasoner because of the run-time behavior.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Einführung *1*
1.2 Aufgabenstellung und Zielsetzung *3*
1.3 Aufbau der Diplomarbeit *4*

2 Grundlagen 7
2.1 Description Logics *7*
2.1.1 Motivation 7
2.1.2 DL-Aufbau 8
2.1.3 Klassen und Konzepte 8
2.1.4 Rollen und Eigenschaften 10
2.1.5 Individuen 11
2.1.6 Operatoren und Semantik 12
2.2 OWL *15*
2.2.1 Gegenüberstellung OWL 1 / OWL 2 16
2.2.2 Gemeinsamkeiten und Unterschiede 17
2.2.3 Neue Features 18
2.2.4 Darstellung der DL-Syntax in der OWL- / Manchester OWL-Syntax 19
2.3 OWL und Integrity Constraints *20*
2.3.1 Problematik 20
2.3.2 Constraints in OWL 21
2.3.3 Typische Anwendung von Integrity Constraints 24
2.4 M-Objects und M-Relationships *27*
2.4.1 Einführung 27
2.4.2 Beispiel 31
2.5 Mapping von M-Objects und M-Relationships nach OWL *34*
2.5.1 Mapping der M-Objects 34
2.5.2 Mapping der M-Relationships 39
2.6 Protégé *41*
2.6.1 Protégé Frames 42
2.6.2 Protégé OWL 44
2.6.3 Protégé OWL API vs. OWL API 3 45
2.7 Umsetzung von M-Objects und M-Relationships in Protégé Frames *46*
2.7.1 Klassen und Klassenhierarchie 47
2.7.2 Slots 49
2.7.3 Formulare / Masken 50
2.7.4 Instanzen 51

3 Realisierung Mapping Algorithmus 53
3.1 Vorbereitende Tätigkeiten *53*
3.1.1 Erstellung des Export-Plugins 53
3.1.2 Erstellung der Ontologie und des Dokuments 56
3.1.3 Erstellung der OWL Annotation Property 58
3.2 Mapping der M-Objects nach OWL *59*
3.2.1 Festlegen der Eigenschaften der Konkretisierungshierarchie 60
3.2.2 Erstellung der M-Objects und Zuweisung der M-Levels zu M-Objects 63
3.2.3 Konkretisierung der M-Objects 65
3.2.4 Eigenschaften und Werte von Eigenschaften 67
3.2.5 Attributschema 69
3.2.6 Vererbung von Eigenschaftswerten 73
3.2.7 Level-Hierarchie 75
3.2.8 Festlegung erlaubter Level-Eigenschaften 77

3.2.9 Festlegung der M-Object-Zugehörigkeit von Levels 79
3.2.10 Sicherstellung Disjoint von M-Levels 81
3.2.11 Sicherstellung Unique Name Assumption 82
3.3 Mapping der M-Relationships nach OWL *83*
3.3.1 Festlegen der Eigenschaften source und target 84
3.3.2 Erstellung und Konkretisierung der M-Relationships 85
3.3.3 Ermittlung und Zuweisung von source und target 87
3.3.4 Source- und Target-Level Konkretisierung 88
3.3.5 Sicherstellung der Navigation entlang von M-Relationships 90
3.3.6 Sicherstellung Unique Name Assumption 93

4 Reasoning – Performance Studies 95
4.1 Testumgebung *95*
4.2 Beispielerweiterung *95*
4.3 Reasoning Ergebnisse *97*
4.3.1 Test der Transitivität von concretize 98
4.3.2 Test der Integrity Constraints 99
4.3.3 Test der Vererbung von Eigenschaftswerten 102
4.3.4 Test der Disjunktheit von M-Levels 103
4.3.5 Test der Unique Name Assumption 104
4.3.6 Resümee 104
4.4 Gegenüberstellung des Laufzeitverhaltens *105*

5 Zusammenfassung 108

Literaturverzeichnis 109

A Anhang 112
A.1 Verwendete Programmversionen *112*
A.1.1 Protégé 3.4.1 112
A.1.2 Protégé 4.1 Beta 114
A.1.3 Pellet 2.1.2 115
A.1.4 Fact++ 1.5.1 115
A.1.5 HermiT 1.3.3 116
A.1.6 Entwicklungsumgebung Eclipse 3.3.2 116

Abbildungsverzeichnis

Abbildung 1: Struktur der Sprachbeschreibungen zu OWL 2 [W3C09a] 17
Abbildung 2: M-Objects [Neum09a] 28
Abbildung 3: M-Relationships [Neum09a] 30
Abbildung 4: Beispiel [Neum09b] 32
Abbildung 5: Mapping von M-Objects nach OWL [Neum09b] 36
Abbildung 6: Mapping von M-Relationships nach OWL [Neum09b] 39
Abbildung 7: Klassenhierarchie (Beispiel aus [Diwo11]) 47
Abbildung 8: Klasseneditor von CarModel (Beispiel aus [Diwo11]) 48
Abbildung 9: Klasseneditor von BookPhysicalEntity (Beispiel aus [Diwo11]) 48
Abbildung 10: Slots (Beispiel aus [Diwo11]) 49
Abbildung 11: Sloteditor von author (Beispiel aus [Diwo11]) 50
Abbildung 12: Sloteditor von concretizationOf (Beispiel aus [Diwo11]) 50
Abbildung 13: Formular / Maske von BookCategory (Beispiel aus [Diwo11]) 51
Abbildung 14: Instanzenbrowser (Beispiel aus [Diwo11]) 52
Abbildung 15: Instanzeneditor von Book (Beispiel aus [Diwo11]) 52
Abbildung 16: Menü nach eigener Plugin-Erstellung 56
Abbildung 17: Neue OWL-Ontologie mit Wurzelklasse Thing 58
Abbildung 18: OWL Annotation Property für Integrity Constraints 59
Abbildung 19: OWL Objekteigenschaften concretize und concretize_t 62
Abbildung 20: concretize als Subobjekteigenschaft von concretize_t 62
Abbildung 21: M-Levels 64
Abbildung 22: Zuweisung des Top-Levels Catalog zu Product 65
Abbildung 23: Zuweisung des Top-Levels Category zu Book und Car 65
Abbildung 24: Car konkretisiert Product 67
Abbildung 25: OWL Dateneigenschaften 68
Abbildung 26: HarryPotter4 mit OWL Dateneigenschaften author & listprice 69
Abbildung 27: Durch IC (Zeile 9) erzeugte Axiome 72
Abbildung 28: OWLSubClassOfAxiom mit Annotation 72
Abbildung 29: Durch Zeile 10 erzeugtes Axiom 74
Abbildung 30: Durch IC (Zeile 12) erzeugte Axiome 77
Abbildung 31: Durch IC (Zeile 14) erzeugte Axiome 79
Abbildung 32: Category als Subklasse von ∃concretize_t.{Product} 81
Abbildung 33: Hinzugefügte Disjoint-Klassen (Levels) zum Level Catalog 82
Abbildung 34: Unique Name Assumption für Book 83
Abbildung 35: OWL Objekteigenschaften source und target 84
Abbildung 36: M-Relationships 86
Abbildung 37: Konkretisierung von Product-producedBy-Company 87
Abbildung 38: Zuweisung von source und target zu Car-producedBy-Company 88
Abbildung 39: Durch IC (Zeile 24) erzeugte Axiome 90
Abbildung 40: Durch IC (Zeile 26) erzeugte Axiome 93
Abbildung 41: Unique Name Assumption Car-producedBy-CarManufacturer 94
Abbildung 42: Ergebnis des Tests der Transitivität (M-Objects) 98
Abbildung 43: Ergebnis des Tests der Transitivität (M-Relationships) 99
Abbildung 44: Fehlermeldung von Fact++ und HermiT 99
Abbildung 45: Fehlermeldung von Pellet 99
Abbildung 46: Testergebnis IC Zeile 16 102
Abbildung 47: Antwortzeitverhalten einzelner Reasoner 107

Tabellenverzeichnis

Tabelle 1: Darstellung von DL in OWL / Manchester OWL [Horr05a], [Horr05b]..... 20
Tabelle 2: M-Objects, Abstraktionsebenen und M-Relationships [Neum09b]..... 34
Tabelle 3: Unterschiede zwischen Protégé OWL API und OWL API 3..... 46
Tabelle 4: M-Objects, M-Levels und M-Realtionships des erweiterten Beispiels 97
Tabelle 5: Gegenüberstellung der Reasoning Ergebnisse 105
Tabelle 6: Laufzeitverhalten der Reasoner bei zunehmender Individuenanzahl 106
Tabelle 7: Eigenschaften von Protégé 3.4.1 [Prot09] 113
Tabelle 8: Eigenschaften von Protégé 4.1 Beta [Prot09] 114

Abkürzungen

ABOX	...	Assertional Box
API	...	Application Programming Interface
bzw.	...	beziehungsweise
C / C++	...	eine ISO standardisierte höhere Programmiersprache
CVS	...	Concurrent Version System
CWA	...	Closed World Assumption
DIG	...	DL Implementation Group
DL	...	Description Logics
EPL	...	Eclipse Public License
GPL	...	GNU General Public License
HTTP	...	Hypertext Transfer Protocol
IC	...	Integrity Constraint
IRI	...	Internationalized Resource Identifier
IP	...	Intellectual Property
IT	...	Informationstechnik
Java EE	...	Java Platform, Enterprise Edition
JDBC	...	Java Database Connectivity
JDK	...	Java Development Kit
JRE	...	Java Runtime Environment
JVM	...	Java Virtual Machine
LGPL	...	GNU Lesser General Public License
M-Object	...	Multi-Level-Objekt
M-Relationship	...	Multi-Level-Beziehung
OSGi	...	Open Services Gateway Initiative
OWA	...	Open World Assumption
OWL	...	Web Ontology Language
PHP	...	„Personal Home Page“ Hypertext Preprocessor
RDF	...	Resource Description Framework
RDFS	...	Resource Description Framework Schema
SOA	...	Serviceorientierte Architektur
SPARQL	...	SPARQL Protocol and RDF Query Language
$\mathcal{SROIQ}$	...	Logische Grundlage von OWL 2 DL
SVN	...	Apache Subversion

SWRL	...	Semantic Web Rule Language
TBOX	...	Terminological Box
URI	...	Uniform Resource Identifier
XML	...	Extensible Markup Language
z.B.	...	zum Beispiel

1 Einleitung

1.1 Einführung

Bei der Multi-Level Modellierung können Objekte mittels Klassifizierung, Aggregation und Generalisierung auf verschiedenen Abstraktionsebenen beschrieben und so konkretisiert werden. Die Wartung und Erweiterung derartiger Modelle stellt sich dabei als äußerst schwierig dar, wodurch sich die Komplexität um ein vielfaches erhöht. Mit den Ansätzen (1) *materialization*, (2) *powertypes* und (3) *deep instantiation* wurde bisher versucht die Komplexität dieser Modelle zu verringern. M-Objects und M-Relationships arbeiten aufbauend auf den Ergebnissen dieser Ansätze. [Neum09a]

Zunächst wird mit ihnen eine Kapselung unterschiedlicher Abstraktionsebenen erreicht, wodurch diese separat betrachtet werden können. Weiters können mit ihnen Klassifizierung, Aggregation und Generalisierungshierarchien in einer einzigen Konkretisierungshierarchie dargestellt werden. Dadurch werden mittels M-Objects und M-Relationships modulare und redundanzfreie Multi-Level-Modelle, Level-Hierarchien und Mehrfachabstraktionsbeziehungen unterstützt und die Anordnung einer Menge verwandter Objekte auf unterschiedlichen Abstraktionsebenen ermöglicht. Ziel dabei ist es gemeinsame Eigenschaften von Objekten abfragen und beschreiben zu können. [Neum09b]

Alois Diwold hat in seiner laufenden Diplomarbeit ein Modellierungswerkzeug entwickelt, welches die Modellierung von M-Objects und M-Relationships in Protégé Frames unterstützt. In meiner Diplomarbeit liegt das Interesse darin, wie sich diese M-Objects und M-Relationships in OWL darstellen und wie sie in Protégé OWL modelliert werden können.

Bei OWL handelt es sich um die W3C Standardsprache zur Modellierung von Ontologien im Semantic Web. Dabei basiert die Logik, die sich hinter OWL verbirgt auf Description Logics (DL), welche eine Gruppe von Sprachen zur Wissensrepräsentation darstellen. [Moti07]

In dieser Arbeit werden OWL und Description Logics häufig synonym verwendet, insbesondere im später angeführten Mapping Algorithmus von Neumayr und Schrefl. Das liegt daran, dass die von mir verwendete Version 2 von OWL weitgehend der DL $\mathcal{SROIQ}$ entspricht. Daher stellt OWL (zumindest so wie es in dieser Arbeit verwendet wird) eine andere Darstellung für DL dar, und umgekehrt.

Der eben erwähnte Mapping Algorithmus enthält eine Vielzahl so genannter Integrity Constraints (ICs). Diese wurden ursprünglich dazu verwendet um die Konsistenz und Korrektheit der Daten einer Datenbank zu gewährleisten. Im Zusammenhang mit ICs und OWL ergibt sich jedoch das Problem, dass jene Axiome (Aussagen darüber, was in einer bestimmten Domain der Wahrheit entspricht [W3C09c]), welche als ICs interpretiert werden sollen, in OWL eine andere Bedeutung haben als analoge Bedingungen in relationalen Datenbanken. [Moti07]

OWL ist im Moment noch nicht imstande ICs als solche zu behandeln, da sie eigentlich auch kein Bestandteil von OWL sind. Es gibt jedoch einen entsprechenden Vorschlag von Motik und Horrocks in [Moti07]. Im Moment werden ICs noch als Standardaxiome unter Open World Assumption (OWA, wird typischerweise im Semantic Web gemacht) interpretiert, was dazu führt, dass sie sehr häufig falsch behandelt werden, da die OWA unvollständige Information als nicht definiert interpretiert und diese Information daher annimmt und herleitet, wodurch Verletzungen häufig nicht erkannt werden. Den Gegensatz zur OWA stellt die Closed World Assumption (CWA, wird von Datenbanken verwendet) dar. Unter dieser wird unvollständige Information nicht angenommen oder hergeleitet, sondern ihr Fehlen als negativ betrachtet, wodurch Verletzungen erkannt werden. [Amat07] Daher stellt sich die Darstellung und Interpretation von ICs in OWL im Vergleich zur Realisierung von ICs in Datenbanken als äußerst schwierig dar.

Im Zusammenhang mit der Multi-Level Modellierung definierten Neumayr und Schrefl einen Algorithmus für das Mapping von M-Objects und M-Relationships nach OWL. Ziel dieses Algorithmus ist es (1) die Semantik von M-Objects und M-Relationships zu erhalten, um ihre grundlegenden Ziele und Anforderungen erfüllen zu können, und (2) eine Darstellung von M-Objects und M-Relationships in OWL zu geben, die es einem Reasoner ermöglicht Inkonsistenzen zu erkennen und Abfragen auf den ver-

schiedenen Abstraktionsebenen durchzuführen. [Neum09b] In dieser Arbeit soll dieser Algorithmus nun in Form eines Plugins für Protégé umgesetzt werden, womit ich auch schon zur Aufgabenstellung und Zielsetzung meiner Diplomarbeit komme.

1.2 Aufgabenstellung und Zielsetzung

1. Realisierung des Mapping Algorithmus aus [Neum09b]

   Zielsetzung

   - Erstellung eines Export-Plugins für Protégé, welches ein Mapping der M-Objects und M-Relationships (Umsetzung des Beispiels aus [Neum09b] von Alois Diwold in Protégé Frames) nach Protégé OWL ermöglicht, so dass sie in Protégé OWL richtig dargestellt werden
   - Korrekte Umsetzung der im Mapping Algorithmus enthaltenen Integrity Constraints, um es später dem Reasoner zu ermöglichen, Integritätsverletzungen zu erkennen (nur näherungsweise möglich)

2. Durchführung der Reasoning – Performance Studies

   Zielsetzung

   - Erstellung eines entsprechenden Designs der Studie, um aussagekräftige Ergebnisse zu erhalten
   - Gegenüberstellung des Antwortzeitverhaltens verschiedener Reasoner (Pellet, Fact++, HermiT)
   - Überprüfung ob die Reasoner Pellet, Fact++ und HermiT die verwendeten OWL-Sprachkonstrukte richtig verwenden und zum erwarteten Reasoning-Ergebnis kommen
   - Ermittlung des am besten geeigneten Reasoners für die Thematik dieser Arbeit

1.3 Aufbau der Diplomarbeit

Zu Beginn dieser Arbeit wird in Kapitel 2 auf jene Grundlagen eingegangen, die für das Verständnis der zentralen Kapitel der Diplomarbeit von Bedeutung sind. Zuerst erfolgt eine kleine Wiederholung von beziehungsweise wird eine kurze Einführung in Description Logics gegeben. Es wird beschrieben, wozu sie verwendet werden beziehungsweise was mit ihnen bezweckt werden soll. Es wird unter anderem erklärt, worum es sich bei Klassen (Classes), Individuen (Individuals), Rollen / Eigenschaften (Properties), ABOX (Assertional Box) und TBOX (Terminological Box) handelt beziehungsweise wie die Notation im Allgemeinen aussieht.

Da ich in meiner Arbeit sowohl Protégé 3.4.1 (unterstützt OWL 1) als auch 4.1 Beta (unterstützt OWL 2) verwende, werde ich im zweiten Punkt des Grundlagenkapitels eine kurze Gegenüberstellung dieser beiden OWL Versionen vornehmen. Ich werde dabei insbesondere auf die Gemeinsamkeiten und Unterschiede eingehen und in weiterer Folge auf jene neuen Features, welche OWL 2 bietet. Zum Abschluss der Einführung in OWL werde ich zeigen, wie sich die DL-Syntax in der OWL- bzw. Manchester OWL-Syntax darstellt.

Der Mapping Algorithmus aus [Neum09b] ist sehr Integrity Constraint lastig. Daher werden ICs in einem eigenen Unterkapitel der Grundlagen behandelt. Insbesondere welche Problematik sich mit ihnen ergibt und wie ihre Darstellung in OWL aussieht. Daran anschließend werden noch einige typische Anwendungen von Integrity Constraints in OWL gezeigt.

Da M-Objects und M-Relationships einen zentralen Bestandteil dieser Arbeit darstellen, werden ihnen gleich zwei Punkte in den Grundlagen gewidmet. Dabei wird zunächst im ersteren der beiden Punkte eine allgemeine Einführung in M-Objects und M-Relationships gegeben, wobei ich mich insbesondere auf das von Schrefl und Neumayr angeführte Beispiel beziehen werde. Nachdem ein gemeinsames Verständnis über M-Objects und M-Relationships besteht, werde ich im zweiten der beiden Punkte auf das Mapping von M-Objects und M-Relationships nach OWL eingehen.

Da ein Teil der Aufgabenstellung auf die Erstellung eines Export-Plugins für Protégé abzielt, werden im folgenden Punkt des Grundlagenkapitels zunächst die zentralen Bestandteile von Protégé Frames (Klassen, Instanzen, Slots, Facets, Formulare / Masken) und Protégé OWL (Individuen, Eigenschaften, Klassen) erklärt. Da zur einwandfreien Umsetzung des Mapping Algorithmus Protégé um die OWL API 3 (unterstützt OWL 2) erweitert werden musste, wird daran anschließend auch auf diese Syntax kurz eingegangen und verglichen, wie sie sich im Gegensatz zur Protégé OWL API darstellt.

Der Abschluss des Grundlagenkapitels widmet sich der Realisierung des zuvor erwähnten Beispiels aus [Neum09b] in Protégé Frames. Diese Umsetzung wurde von Alois Diwold im Rahmen seiner laufenden Diplomarbeit [Diwo11] vorgenommen und stellt den Einstiegspunkt für meine Diplomarbeit dar. Ich werde hier ein paar Screenshots seiner Beispielrealisierung in Protégé Frames anführen und diese erklären, damit später in Kapitel 3 klar ist, worauf ich mich beziehe, wenn die einzelnen Methoden zur Realisierung des Mapping Algorithmus erläutert werden.

Anschließend an das Grundlagenkapitel wird in Kapitel 3 erklärt, wie ich den Mapping Algorithmus aus [Neum09b] umgesetzt habe. Zunächst werden jene Tätigkeiten beschrieben, die vorgenommen werden müssen, um mit der Umsetzung des Mapping Algorithmus beginnen zu können, nämlich der Erstellung des Export-Plugins, der Initialisierung der OWL Ontologie und der Erstellung der OWL Annotation Property.

Daran anschließend werden die für das Mapping relevanten Methoden beschrieben, ohne jedoch den Source Code explizit anzuführen. Es wird dabei beschrieben, was mit einer bestimmten Methode umgesetzt wurde, auf welche Zeile des Mapping Algorithmus sich diese Methode bezieht und wie ich dabei vorgehe, um die gewünschten Informationen aus der Beispielrealisierung von Alois Diwold herauszulesen, um sie anschließend in das OWL Modell einzubauen. Anhand von Screenshots werde ich laufend zeigen, wie sich die Informationen nach Anwendung einer Methode in Protégé OWL darstellen, ohne jedoch bereits ein Reasoning durchzuführen. In diesem Kapitel geht es hauptsächlich darum, ob die Informationen vollständig und richtig abgebildet werden.

In Kapitel 4 werden schließlich die Reasoning – Performance Studies durchgeführt. Zuerst erfolgt eine Beschreibung der Testumgebung, in welcher die Studien durchgeführt wurden. Daran anschließend werde ich zeigen, wie ich das Beispiel aus [Neum09b] um eine Vielzahl von M-Objects und M-Relationships erweitert habe, um im späteren Verlauf bei der Gegenüberstellung des Antwortzeitverhaltens der einzelnen Reasoner ausdruckstarke Ergebnisse zu erhalten.

Bevor das eben erwähnte Antwortzeitverhalten getestet wird, werden jedoch noch die von den Reasonern ermittelten Ergebnisse gezeigt. Der Fokus liegt dabei vor allem darauf, ob sie die Transitivität richtig ableiten können, ob die Vererbung von Eigenschaftswerten funktioniert, ob die Axiome zur Umsetzung der Unique Name Assumption der M-Objects und M-Relationships richtig interpretiert werden und in wie weit sich die Open World Assumption (wird von diesen Reasonern verwendet) mit den beabsichtigten Ergebnissen im Hinblick auf die ICs im Moment bereits decken. Die einzelnen Ergebnisse werden abschließend im Hinblick auf die Reasoning Ergebnisse als auch das Antwortzeitverhalten zusammenfassend gegenübergestellt.

Bevor abschließend im Anhang die einzelnen von mir verwendeten Programmversionen genauer beschrieben werden, wird in einer Zusammenfassung nochmals auf die Ziele dieser Arbeit eingegangen und inwieweit diese Ziele realisiert werden konnten.

2 Grundlagen

2.1 Description Logics

Dieser Punkt soll eine kurze Einführung in Description Logics geben. Jene, die bereits mit DL zu tun hatten, können diesen Punkt als kurze Wiederholung ansehen.

2.1.1 Motivation

Wie man unschwer übersetzen kann, handelt es sich bei Description Logics um Beschreibungslogiken, welche eine Gruppe von Sprachen zur Wissensrepräsentation darstellen. In den 1970er Jahren erfreuten sich Ansätze zur Wissensrepräsentation größter Beliebtheit, wobei zwischen logisch basierten Formalismen und nicht logisch basierten Repräsentationen unterschieden wurde. [Nard03]

Spricht man von Description Logics, kommt man am Begriff der Ontologie nicht vorbei. Bei Ontologie handelte es sich ursprünglich um eine Disziplin aus dem Bereich der Philosophie, die sich mit der Natur und Organisation der Realität befasste. Man stellte sich vor allem die Frage: „Wie sollten Dinge klassifiziert werden?“. Im Bereich der Computerwissenschaften stellt eine Ontologie eine formale Darstellung von Wissen bestehend aus (1) einem Vokabular zur Beschreibung der Sichtweise auf einen Bereich, (2) einer expliziten Spezifikation der beabsichtigten Bedeutung des Vokabulars, und (3) durch Einschränkungen zusätzlich gewonnenes Wissen über diesen Bereich, dar. Idealerweise sollte eine Ontologie ein gemeinsames Verständnis für einen Interessensbereich schaffen und ein formales und maschinenmanipulierbares Modell über diesen Bereich zur Verfügung stellen. [Horr05a]

Durch Reasoner (auch Classifier genannt) kann beschriebenes Wissen ausgewertet werden. Ontologien erleichtern ihnen dabei die Interpretation von Zusammenhängen, da eine einheitliche Modellierungssprache vorhanden ist. [Berg08] Heutzutage finden Ontologien bereits in vielen Bereichen ihre Anwendung, beispielsweise in der Medi-

zin, in der Bioinformatik, bei Datenbanken, bei Benutzeroberflächen, in der Linguistik und schließlich dem Semantic Web. [Horr05a]

2.1.2 DL-Aufbau

Description Logics unterscheiden zwischen Instanzen auf der einen und Schema- und Strukturinformationen auf der anderen Seite. Während es sich bei Instanzen um Individuen handelt, welche in der so genannten ABOX (Assertional Box) deklariert werden, handelt es sich bei Schema- und Strukturinformationen um Klassen (Konzepte) und deren Beziehungen (Rollen), welche in der so genannten TBOX (Terminological Box) deklariert werden. [Berg08]

In der TBOX werden mittels Axiome Konzepte definiert, wobei ein Konzept auch durch bereits definierte Konzepte beschrieben werden kann. Mittels Konzept- und Rollenassertionen kann Wissen spezifiziert werden, da die ABOX Zusatzwissen über den Interessensbereich enthält, was mit Hilfe von Assertionen über Individuen erlangt wird. [Nard03]

2.1.3 Klassen und Konzepte

Klassen können in zwei Formen auftreten, nämlich zum einen als vollständig definierte (abgeleitete) Klassen und zum anderen als primitive und unvollständig definierte (zugesicherte) Klassen. [Nard03]

Vollständig definierte Klassen benötigen „notwendig und hinreichende“ Bedingungen. Diese werden verwendet um ein Individuum klassifizieren zu können. Notwendige und hinreichende Bedingungen werden auch als logische Äquivalenz angesehen und in DL durch $\equiv$ gekennzeichnet. Solche „Definitionen“ sind aussagekräftiger als notwendige Bedingungen alleine (auch Inklusion oder Spezialisierung genannt und durch $\sqsubseteq$ gekennzeichnet). Ein Beispiel für eine vollständig definierte Klasse könnte folgendermaßen aussehen (übernommen aus [Nard03] und ins Deutsche übersetzt):

$$\text{Frau} \equiv \text{Person} \sqcap \text{Weiblich}$$

Aus diesem Axiom kann abgeleitet werden, dass jedes Individuum, welches sowohl eine *Person* als auch *Weiblich* ist, eine *Frau* ist. Ebenso kann umgekehrt darauf geschlossen werden, dass jede *Frau* eine *Person* ist, die *Weiblich* ist. [Nard03]

Unvollständig definierte (primitive, zugesicherte) Klassen unterscheiden sich von vollständig definierten Klassen dahingehend, dass lediglich von der linken auf die rechte Seite des Axioms geschlossen werden kann, jedoch nicht umgekehrt. Ein Beispiel für eine unvollständig definierte Klasse könnte folgendermaßen aussehen: [Nard03]

$$\text{Frau} \sqsubseteq \text{Person} \sqcap \text{Weiblich}$$

Aus diesem Axiom kann abgeleitet werden, dass ein Individuum, welches als *Frau* ausgezeichnet wird, sowohl eine *Person* als auch *Weiblich* ist. Jedoch nur in diese Richtung und nicht umgekehrt, wie dies bei einer vollständig definierten Klasse möglich wäre.

Eine Sonderform von Klassen stellen so genannte Nominals (auch als *enumerated classes* bezeichnet) dar. Dabei handelt es sich um Klassen, bei denen die Menge der Objekte genau vorgegeben ist, das heißt genau auf diese Objekte beschränkt sind. Diese Klassen werden in DL in geschwungenen Klammern { } dargestellt. Nehmen Sie als Beispiel die Benelux-Länder, die durch die Klasse

$$\{\text{Belgien, Niederlande, Luxemburg}\}$$

ausgedrückt werden. [Horr07a]

Von großer Bedeutung im Zusammenhang mit Klassen und Konzepten sind die beiden Konzepte *Top* und *Bottom*. Während *Top* ($\top$) alle Objekte einer Wissensbasis bezeichnet, stellt *Bottom* ($\bot$) ein unerfüllbares Konzept dar. Nehmen Sie als Beispiel die folgenden beiden Axiome:

(1) $\top \sqsubseteq \text{Ding} \sqcup \text{Lebewesen}$

(2) $\text{Arbeiter} \sqcap \text{Student} \sqsubseteq \bot$

Während in (1) festgelegt wird, dass das Top-Konzept sowohl alle Individuen des Konzepts *Ding* als auch des Konzepts *Lebewesen* umfasst, stellt das Axiom in (2) sicher, dass es sich bei den beiden Klassen *Arbeiter* und *Student* um zwei disjunkte (nicht überlappende) Klassen handelt.

2.1.4 Rollen und Eigenschaften

Konzepte können simple Rollen (häufig auch als Eigenschaften oder Attribute bezeichnet) besitzen, welche es ermöglichen, zwei Individuen in Beziehung zu setzen. Rollen stehen in der Modellierung gleichberechtigt neben Konzepten und sind über eine Domain definiert und besitzen eine Range, wobei man unter Domain das Quell- und unter Range das Zielkonzept versteht. Ein Beispiel hierfür könnte folgendermaßen aussehen: [Berg08]

(1) $\exists \text{hatKind}.\top \sqsubseteq \text{Person}$

(2) $\top \sqsubseteq \forall \text{hatKind}.\text{Person}$

(3) hatKind(Helga, Joachim)

Während in (1) festgelegt wird, dass die Domain der *hatKind*-Beziehung eine *Person* ist, wird in (2) festgelegt, dass die Range dieser Beziehung ebenfalls eine *Person* darstellt. In (3) werden die beiden Individuen *Helga* und *Joachim* durch die *hatKind*-Beziehung miteinander verlinkt, woraus logisch gefolgert werden kann, dass *Helga* und Joachim *Personen* sind.

Will man nun beispielsweise ausdrücken, dass einer von *Joachims* Elternteilen *Helga* ist, könnte man dies mit der Rolle *hatElternteil* ausdrücken. In diesem Fall stellt die Rolle *hatElternteil* die so genannte inverse Rolle von *hatKind* dar, da man entweder sagen könnte „Helga hat ein Kind namens Joachim“ oder „Joachim hat einen Elternteil namens Helga“. Inverse Rollen werden mit $^{-}$ gekennzeichnet. Infolge dessen würde in unserem Fall die inverse Rolle von *hatKind* durch $hatKind^{-}$ dargestellt wer-

den. Um nun festzulegen, dass es sich bei der Rolle *hatElternteil* um die inverse Rolle von *hatKind* handelt, würde man folgenden Ausdruck anwenden: [Berg08]

$$\text{hatElternteil} \equiv \text{hatKind}^{-}$$

Neben inversen gibt es noch funktionale, symmetrische und transitive Rollen. Bei funktionalen Rollen handelt es sich um Rollen, welche genau einen Zielwert umfassen und für die Repräsentation von einwertigen Attributen Verwendung finden. Symmetrische Rollen können in beide Richtungen gelesen werden. Zum Beispiel folgt aus „Michael hat einen Mitspieler namens Rene" umgekehrt „Rene hat einen Mitspieler namens Michael“.

Zu guter Letzt können mit transitiven Rollen Transitivitäten ausgedrückt werden. Transitive Rollen können folgendermaßen beispielhaft erklärt werden: Joachim hat einen Untergebenen namens Helmut, Helmut hat wiederum einen Untergebenen namens Roland, der wiederum einen Untergebenen in Markus hat. Da Joachim in Helmut einen Untergebenen hat, Helmut in weiterer Folge Roland und Roland schlussendlich Markus als Untergebenen hat, sind sowohl Helmut, Roland und Markus die Untergebenen von Joachim (vorausgesetzt, dass *hatUntergebenen* als transitive Eigenschaft angesehen wird). Um in DL-Syntax die Rolle *hatUntergebenen* als transitiv zu definieren schreibt man:

$$\text{hatUntergebenen}^{+} \sqsubseteq \text{hatUntergebenen}$$

Wie sie im angeführten Axiom sehen, stellt die transitive Rolle (Eigenschaft) von *hatUntergebenen* eine Subeigenschaft von *hatUntergebenen* dar, gekennzeichnet durch $^{+}$. [Berg08]

2.1.5 Individuen

Durch Klassifizierung werden den Individuen Klassen zugewiesen, beispielsweise in 2.1.4 *Joachim* und *Helga* die Klasse *Person*. Weiters könnte dem Individuum *Joachim* die Klasse *Student* und dem Individuum *Helga* die Klasse *Arbeiter* zugewie-

sen werden. Einem Individuum können also durchaus mehrere Klassen zugewiesen werden. Wird eine Klasse einem Individuum zugewiesen, so erhält das Individuum die Merkmale sowie alle abgeleiteten Merkmale dieser Klasse, wobei abgeleitete Zuordnungen immer auf den Merkmalen des Individuums basieren. [Berg08]

2.1.6 Operatoren und Semantik

Aus Konzepten und Rollen können unter Zuhilfenahme verschiedener Operatoren umfangreiche Ausdrücke gebildet werden. Um den Rahmen dieses Kapitels nicht zu sprengen, werde ich hier nur auf die wichtigsten Operatoren (bzw. jene Operatoren, die in Punkt 2.5 beim Mapping Algorithmus auftreten) eingehen. Für ausführlichere Erklärungen und Beschreibungen verweise ich auf [Nard03].

Komplement

Komplemente drücken das genaue Gegenstück eines Konzepts aus. In DL wird dieser Sachverhalt wie folgt ausgedrückt:

$$\neg C$$

Beispiel: Frau $\equiv$ Person $\sqcap \neg$ Männlich

Im angeführten Beispiel wird *Frau* als äquivalent mit *nicht Männlichen Personen* definiert. Man würde natürlich mit *Person* $\sqcap$ *Weiblich* dasselbe Konzept beschreiben, unter der Annahme, dass *Weiblich* ebenfalls bereits definiert wurde, es nur diese beiden Geschlechter gibt und es sich darüber hinaus bei den beiden Geschlechtern um disjunkte Klassen handelt.

Konzeptkonjunktion

Konzeptkonjunktionen dienen dazu Mengen von Individuen im Hinblick darauf zu definieren, dass nur jene Individuen betrachtet werden, welche sowohl zum ersten als auch zum zweiten Konzept gehören, beispielsweise zu den beiden Konzepten *C* und *D*, siehe folgendes Beispiel: [Nard03]

C ⊓ D
Beispiel: Mann ≡ Person ⊓ Männlich

Im angeführten Beispiel werden jene Individuen, die zum Konzept *Person* UND zum Konzept *Männlich* gehören, im Konzept *Mann* zusammengefasst. *Mann* stellt also die Schnittmenge der beiden anderen Konzepte dar.

Vereinigung

Im Gegensatz zur Konzeptkonjunktion wird bei der Vereinigung nicht die Schnittmenge, sondern die Vereinigungsmenge zweier Mengen von Individuen ermittelt. DL sieht hierfür folgende Darstellung vor:

C ⊔ D
Beispiel: Elternteil ≡ Mutter ⊔ Vater

Im Beispiel werden alle Individuen des Konzepts *Vater* als auch all jene des Konzepts *Mutter* im Konzept *Elternteil* zusammengefasst.

Kardinalitäten

Kardinalitäten werden häufig als charakteristisches Merkmal von Description Logics bezeichnet, obwohl sie durchaus auch in anderen Modellierungssprachen vorkommen. Mit ihnen wird die Anzahl von Rollen begrenzt, wobei es 3 Arten von möglichen Einschränkungen gibt: maximale (≤), minimale (≥) und exakte (=) Kardinalität. Zum Beispiel würde

(≥ 3 hatKind) ⊓ (≤ 2 hatWeiblichenAngehörigen)

die Klasse jener Individuen beschreiben, die mindestens 3 Kinder und höchstens 2 weibliche Angehörige haben. Würden die beiden Kardinalitäten ≤ und ≥ in diesem Beispiel durch eine exakte Kardinalität ersetzt werden, würde es jene Individuen beschreiben, die genau 3 Kinder und genau 2 weibliche Angehörige haben. [Nard03]

Allquantor

Der Allquantor wird für Restriktionen von Rollen und Werten verwendet. Damit werden Konzepte als jene Mengen von Individuen definiert, die über die angegebene Rolle (*R*) nur mit Instanzen des angegebenen Konzepts (*C*) in Beziehung stehen. In DL wird dies folgendermaßen ausgedrückt: [Nard03]

$$\forall R.C$$
Beispiel: NurSöhne $\equiv \forall$hatKind.Mann

Durch den Allquantor wird im angeführten Beispiel erreicht, dass das Konzept *NurSöhne* nur jene Objekte umfasst, die ausschließlich *männliche Kinder* haben, das heißt, jene Individuen die kein *nicht-männliches Kind* haben.

Existenzquantor

Der Existenzquantor wird ebenfalls für Restriktionen von Rollen und Werten verwendet. Will man zum Beispiel das Konzept „Individuen die eine Tochter haben“ beschreiben, kommt der Existenzquantor zum Einsatz. Es werden mit diesem Konzept also alle Individuen beschrieben, die zumindest eine Tochter haben, es spielt dabei aber keine Rolle, ob das Individuum nur Töchter hat, wie das durch den Allquantor erreicht werden würde. Eine solche Rollenrestriktion wird in DL folgendermaßen ausgedrückt: [Nard03]

$$\exists R.C$$
Beispiel: Mutter $\equiv$ Frau $\sqcap \exists$hatKind.Person

Im angeführten Beispiel wird eine *Mutter* dadurch beschrieben, dass es sich bei dem Individuum um eine *Frau* handelt, welche in zumindest einer *hatKind* Beziehung mit einer Person steht.

2.2 OWL

OWL ist die W3C Standardsprache zur Modellierung von Ontologien im Semantic Web und kann als äußerst ausdruckstarke Sprache zur Darstellung von Wissen angesehen werden. Die Logik, welche sich hinter OWL verbirgt, basiert auf Description Logics. [Moti07]

Klassen, Individuen und Rollen / Eigenschaften stellen wichtige Bestandteile einer OWL-Ontologie dar. Klassen werden dazu verwendet, um Mengen von Individuen zusammenzufassen und können benannt (mit einem Namen versehen) oder anonym (besitzen keinen Namen) sein. Darüber hinaus können sie übergeordnete Klassen (Superklassen) und untergeordnete Klassen (Subklassen) besitzen. Zu guter Letzt kann festgelegt werden, ob es sich bei zwei Klassen um nicht überlappende (Disjoint Classes) oder gleiche (Equivalent Classes) Klassen handelt. [Horr07b] Auf sie wurde in Punkt 2.1.3 bereits intensiver eingegangen.

Individuen werden dazu verwendet, um Objekte innerhalb eines bestimmten Interessensbereichs darzustellen. Sie können Instanzen von Klassen darstellen und werden daher in diesem Zusammenhang auch häufig als Instanzen und nicht als Individuen bezeichnet. [Horr07b]

In OWL wird zwischen zwei Arten von Eigenschaften unterschieden: (1) Objekteigenschaften und (2) Dateneigenschaften. Durch erstgenannte Eigenschaft werden zwei Individuen (Objekte) zueinander in Beziehung gesetzt, beispielsweise durch die Eigenschaft *hatUntergebenen* die beiden Individuen *Helmut* und *Joachim*, bei der z.B. *Helmut* einen Untergebenen von *Joachim* darstellt. Diese Eigenschaften können funktional, transitiv oder symmetrisch sein und eine inverse Eigenschaft besitzen (siehe Punkt 2.1.4). Darüber hinaus gibt es so genannte Dateneigenschaften. Diese Eigenschaften setzen im Gegensatz zu Objekteigenschaften nicht zwei Individuen, sondern ein Individuum und einen konkreten Datenwert in Verbindung, beispielsweise das Individuum *Auto* mit der Eigenschaft *Höchstgeschwindigkeit* und dem konkreten Wert 200 km/h. [Horr07b]

Da in dieser Arbeit immer wieder von der Webontologiesprache OWL die Rede ist und sowohl Version 1 als auch Version 2 zum Einsatz kommen, werde ich hier kurz auf die Gemeinsamkeiten und Unterschiede dieser beiden Versionen eingehen beziehungsweise welche neuen Features OWL 2 im Vergleich zu seinem Vorgänger bietet.

2.2.1 Gegenüberstellung OWL 1 / OWL 2

Vielleicht fragen Sie sich, warum ich mich nicht einfach für eine der beiden Versionen entschieden habe. Das hat folgende Gründe:

(1) der Mapping Algorithmus aus [Neum09b] macht die Verwendung von OWL 2 erforderlich

(2) M-Objects und M-Relationships wurden von Alois Diwold in seiner Diplomarbeit in Protégé 3 Frames implementiert. Protégé 3 unterstützt jedoch lediglich OWL 1, daher möchte ich diese Version hier auch nicht außen vor lassen.

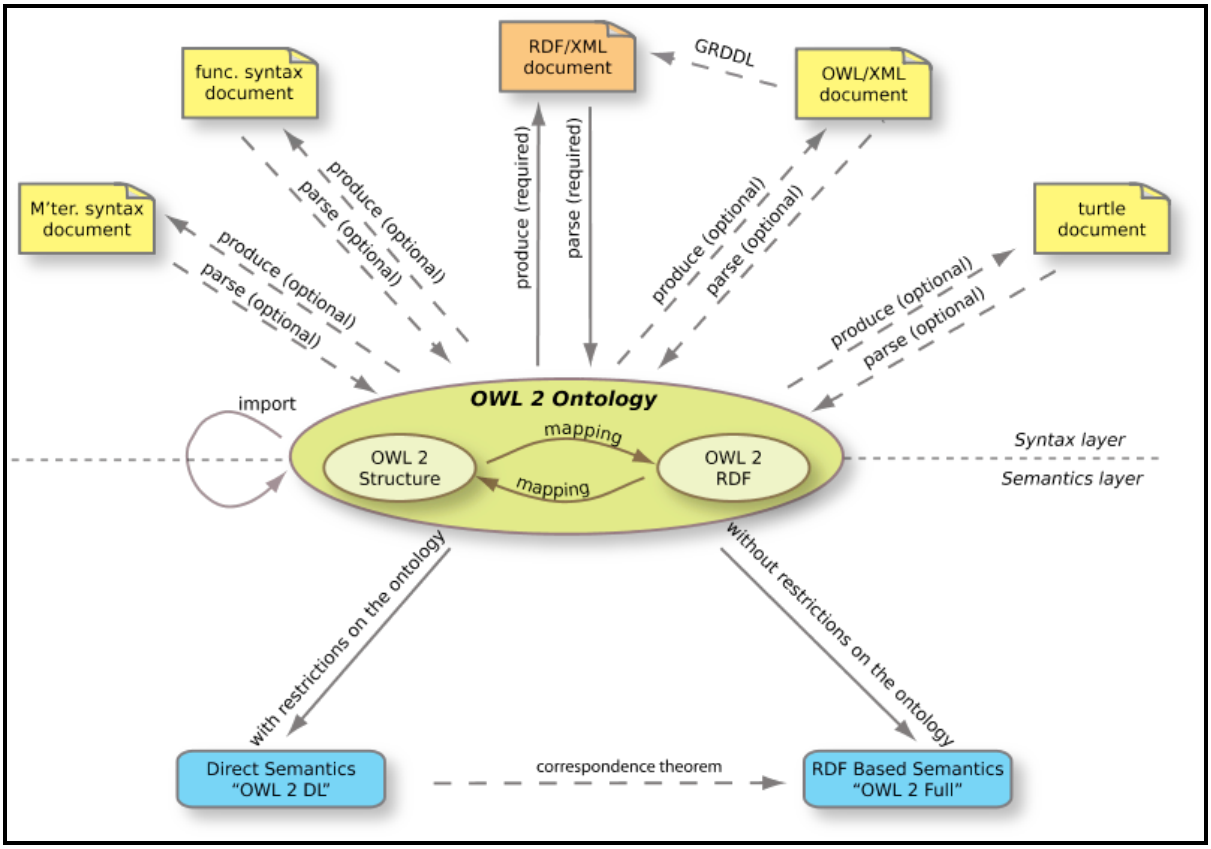

Abbildung 1: Struktur der Sprachbeschreibungen zu OWL 2 [W3C09a]

Abbildung 1 soll einen kleinen Überblick über die Struktur der Sprachbeschreibungen zu OWL 2 geben. Während die Ellipse in der Mitte eine OWL 2 Ontologie darstellt (kann einerseits eine abstrakte Struktur oder ein RDF-Graph sein), repräsentieren die fünf Rechtecke oberhalb jeweils eine konkrete Syntax, mit denen Ontologien serialisiert und ausgetauscht werden können. Die beiden blauen Kästen unterhalb stellen die beiden semantischen Spezifikationen dar, welche die Bedeutung einer OWL 2 Ontologie definieren. [W3C09a] Für eine intensivere Betrachtung der Struktur von OWL-Ontologien verweise ich auf [W3C09c].

2.2.2 Gemeinsamkeiten und Unterschiede

Obwohl sich einige der in Abbildung 1 dargestellten Blöcke auch in OWL 1 wiederfinden, weisen die beiden Versionen doch wesentliche Unterschiede auf:

- Die funktionale Syntax von OWL 2 leitet sich aus der abstrakten OWL Syntax von OWL 1 ab. Die Syntax der beiden ist zwar unterschiedlich, jedoch ist ihre

Rolle in der Gesamtstruktur von OWL identisch. Die funktionale Syntax von OWL 2 kann einerseits mehr RDF Graphen umfassen und kommt andererseits näher an die RDF Graphen-Darstellung heran.

- Beide Versionen ermöglichen ein präzises Mapping von Ontologie-Strukturen zu RDF Graphen. Jedoch ist nur OWL 2 in der Lage, ein Mapping auch in die entgegengesetzte Richtung vorzunehmen.
- Die beiden Semantiken „RDF-basiert“ und „Direkt“ von OWL 2 tauchen in der Vorgängerversion unter den Namen „RDF kompatible modelltheoretische Semantik“ und „Direkt modelltheoretische Semantik“ auf.
- Eine Präsentationssyntax für XML steht ebenfalls für beide Versionen zur Verfügung, wobei jene von OWL 1 nicht als empfehlenswert anzusehen ist. Darüber hinaus existiert die Manchester OWL-Syntax in dieser Version nicht.
- In OWL 1 erfolgt die Definition der Profile (Subsprachen) OWL Full, OWL DL und OWL Lite, die aufgrund der Rückwärtskompatibilität auch Profile in OWL 2 darstellen. Zusätzlich werden in OWL 2 die drei Profile OWL 2 EL, OWL 2 QL und OWL 2 RL definiert.
- Die Beziehungen zwischen den beiden Semantiken „RDF-basiert“ und „Direkt“, sowie die Rolle von RDF/XML und anderen Syntaxen hat sich nicht verändert. Alle Ontologien aus OWL 1 stellen auch gültige Ontologien in OWL 2 dar. [W3C09a]

2.2.3 Neue Features

Da es sich bei OWL 2 um eine Erweiterung von OWL 1 handelt, ist klar, dass diese neuere Version im Vergleich zur älteren einige neue Features bietet. Manche dieser Neuerungen betreffen die Syntax, während andere neue Ausdrücke zur Verfügung stellen, beispielsweise: [W3C09a]

- Schlüssel
- Eigenschaftsketten
- umfangreichere Datentypen und Datenbereiche
- qualifizierte Kardinalitätsrestriktionen
- asymmetrische und reflexive Eigenschaften
- Annotierungen (Annotations)

Diese Aufzählung soll lediglich einen kleinen Überblick der neuen Features von OWL 2 geben. Sollten Sie an einer detaillierten Beschreibung aller Neuerungen interessiert sein, verweise ich auf [W3C09b]. Als letztes ist in diesem Punkt noch zu erwähnen, dass OWL 2 im Unterschied zu OWL 1 unter bestimmten Umständen erlaubt ein und dieselbe URI für eine Klasse und ein Individuum (so genanntes Punning) zu verwenden. Dadurch wird erreicht, dass sich die Anzahl an RDF Graphen, welche durch DL Reasoner gehandhabt werden können, etwas erweitert hat. [W3C09a]

2.2.4 Darstellung der DL-Syntax in der OWL- / Manchester OWL-Syntax

Da die Aufgabe dieser Arbeit unter anderem darin besteht ein Mapping nach OWL zu ermöglichen, werde ich hier kurz darauf eingehen, wie die Syntax von DL in OWL aussieht. Da Protégé die Manchester OWL-Syntax verwendet, führe ich hier zusätzlich an, wie sich die Syntax von DL in dieser Syntax darstellt, damit in Kapitel 3, wo Sie mit dieser Syntax konfrontiert werden, keine Verständnisprobleme auftreten. Die folgende Tabelle, entnommen aus [Horr05a] und [Horr05b] stellt hierfür eine gute Übersicht dar.

DL-Syntax	OWL-Syntax	Manchester OWL-Syntax
$C_1 \sqcap \ldots \sqcap C_n$	intersectionOf	C_1 and ... and C_n
$C_1 \sqcup \ldots \sqcup C_n$	unionOf	C_1 or ... or C_n
$\neg C$	complementOf	not C
$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$	oneOf	$\{x_1 \ldots x_n\}$
$\forall P.C$	allValuesFrom	P only C
$\exists P.C$	someValuesFrom	P some C
$\leq nP$	maxCardinality	P max n
$\geq nP$	minCardinality	P min n
$C_1 \sqsubseteq C_2$	subClassOf	C_1 SubClassOf C_2
$C_1 \equiv C_2$	equivalentClass	C_1 EquivalentTo C_2
$C_1 \sqsubseteq \neg C_2$	disjointWith	C_1 DisjointWith C_2

$\{x_1\} \equiv \{x_2\}$	sameIndividualAs	$\{x_1\}$ SameAs $\{x_2\}$
$\{x_1\} \sqsubseteq \neg \{x_2\}$	differentFrom	$\{x_1\}$ DifferentFrom $\{x_2\}$
$P_1 \sqsubseteq P_2$	subPropertyOf	P_1 SubPropertyOf P_2
$P_1 \equiv P_2$	equivalentProperty	P_1 EquivalentTo P_2
$P_1 \equiv P_2^-$	inverseOf	P_1 InverseOf P_2
$P^+ \sqsubseteq P$	transitiveProperty	Transitive P
$\top \sqsubseteq \leq 1P$	functionalProperty	Functional P
$\top \sqsubseteq \leq 1P^-$	inverseFunctionalProperty	InverseFunctional P

Tabelle 1: Darstellung von DL in OWL / Manchester OWL [Horr05a], [Horr05b]

2.3 OWL und Integrity Constraints

Da im Punkt 2.5 bei der Vorstellung des Mapping Algorithmus häufig von Integrity Constraints (ICs) die Rede ist und diese darüber hinaus einen zentralen Bestandteil dieses Algorithmus darstellen, soll dieser Punkt ein Verständnis für sie schaffen.

2.3.1 Problematik

Integrity Constraints (Integritätsbedingungen) werden verwendet, um die Konsistenz und Korrektheit der Daten einer Datenbank zu gewährleisten. Wird nun versucht derartige Integritätsbedingungen in OWL umzusetzen, entsteht das Problem, dass die Axiome, welche als ICs interpretiert werden sollen, eine andere Bedeutung haben als analoge Bedingungen in Datenbanken. Das führt auch dazu, dass ihre Interpretation in OWL häufig als zweckwidrig bzw. ungeeignet erscheint. [Moti07]

Nehmen Sie als Beispiel eine Anwendung zur Verwaltung von Steuerrückzahlungen, die es verlangt, dass zu jeder Person eine Sozialversicherungsnummer existiert. In einer relationalen Datenbank würde dies mittels einer Inklusionsabhängigkeit gelöst werden, welche festlegt, dass zu jeder Person eine Sozialversicherungsnummer vor-

handen sein muss. Bei einer Datenbankaktualisierung würde diese Bedingung als Check interpretiert werden, was bedeutet, dass sobald eine neue Person zur Datenbank hinzugefügt wird auch eine Prüfung erfolgt, ob für diese Person eine Sozialversicherungsnummer spezifiziert wurde. Ist das nicht der Fall, wird die Datenbankaktualisierung verweigert. [Moti07]

Dieselbe Bedingung würde in OWL mittels einer *existential restriction* (Existenzrestriktion) ausgedrückt werden, jedoch zu einem anderen Ergebnis führen. Wird eine Person ohne Sozialversicherungsnummer zu der Ontologie hinzugefügt, würde das zu keiner Fehlermeldung führen. Es würde lediglich zu der Schlussfolgerung führen, dass die betreffende Person über eine Sozialversicherungsnummer verfügt, diese jedoch nicht bekannt ist. [Moti07] Um die soeben beschriebene Problematik zu lösen, schlagen Motik, Horrocks und Sattler in [Moti07] eine Erweiterung von OWL vor, bei der die Integrity Constraints die gleiche Bedeutung wie in relationalen Datenbanken erhalten sollen. Darüber hinaus geben sie eine gute Gegenüberstellung von OWL und relationalen Datenbanken, worauf ich hier aber nicht näher eingehen möchte.

2.3.2 Constraints in OWL

Nehmen Sie als Ausgangsbeispiel die Anwendung des Österreichischen Fußballverbandes zur Verwaltung aller aktiven Fußballspieler in Österreich. Um für einen Verein spielberechtigt zu sein benötigt jeder Spieler einen Spielerpass mit einer eindeutigen Spielerpass-Identifikationsnummer (*SPN*, ähnlich anzusehen wie die Sozialversicherungsnummer bei der Anwendung zur Verwaltung von Steuerrückzahlungen). Nehmen sie darüber hinaus die folgende ABOX und TBOX als gegeben an, wobei (6) als Constraint und (5) als normales Standardaxiom behandelt werden sollen (angelehnt an [Moti07]):

ABOX (A_1)

(1) Spieler(Klaus)
(2) hatSPN(Klaus, nr98765)
(3) SPN(nr98765)
(4) Spieler(Hannes)

TBOX (T_1)

(5) Spieler $\sqsubseteq$ Person
(6) Spieler $\sqsubseteq \exists$hatSPN.SPN

Aufgrund von (5) erhalten wir *Person(Klaus)* und *Person(Hannes)*. Da für *Klaus* eine *SPN* spezifiziert wurde, wird der Constraint aus (6) aufgrund von (1), (2) und (3) für ihn erfüllt. Für *Hannes* wurde hingegen keine *SPN* spezifiziert, wodurch der Constraint verletzt wäre. [Moti07] Da wie bereits erwähnt Constraints eigentlich kein Bestandteil von OWL sind und OWL daher bisher noch nicht in der Lage ist zwischen Standardaxiomen und Constraints zu unterscheiden, wird das Axiom aus (6) unter Open World Assumption ebenfalls als Standardaxiom interpretiert und dadurch die Verletzung nicht erkannt. Das liegt daran, weil unter OWA fehlende Information defaultmäßig angenommen und hergeleitet wird.

Das ist jedoch genau das, was mittels Integrity Constraints verhindert werden soll. Ihr Ziel ist es, inkonsistente, fehlerhafte oder unvollständige Daten zu erkennen und IC-Verletzungen anzuzeigen.

Um nun zwischen jenen Axiomen, welche neue Fakten erzeugen und jenen Axiomen, welche lediglich dazu verwendet werden, um zu prüfen, ob alle benötigten Informationen abgeleitet werden können, zu unterscheiden, wird die DL Wissensbasis um Constraints erweitert. Die erweiterte DL Wissensbasis *K* setzt sich dann wie folgt zusammen: [Moti07]

- *S* ... eine endliche Menge von Standard TBOX Axiomen
- *C* ... eine endliche Menge von Constraint TBOX Axiomen
- *A* ... eine endliche Menge von ABOX Assertionen ($\neg$) A(a), R(a,b), a $\approx$ b oder a $\not\approx$ b, wobei A ein atomares Konzept, R eine Rolle und a bzw. b Individuen darstellen

Dabei stellt eine Interpretation *I* ein Modell (gültige Interpretation) dar, wenn *C* erfüllt ist. Das alleine reicht jedoch noch nicht aus, um das gewünschte Ergebnis zu erhalten. Die Constraints sollen außerdem prüfen, ob die Fakten, welche aus der Vereinigung von *A*, *S* und *C* abgeleitet werden können auch aus der Vereinigung von *A* und *S* alleine abgeleitet werden können. Dazu prüft *C* jedes dieser so genannten *minimal models*, ob es den Constraint erfüllt oder verletzt. Um Ihnen das ganze etwas ver-

ständlicher darlegen zu können, werfen Sie einen Blick auf das folgende Beispiel (angelehnt an [Moti07]):

ABOX (A_2)	Standard TBOX (S_2)	Constraint TBOX (C_2)
Türke(Murat)	Türke ⊑ Europäer ⊔ Asiat	Europäer ⊑ Mensch Asiat ⊑ Mensch

Standard TBOX (S_3)
Türke ⊑ (Europäer ⊓ Mensch) ⊔ (Asiat ⊓ Mensch)

Aus der Vereinigung von A_2 und S_2 ergeben sich folgende zwei *minimal models*, wobei keines der beiden die in C_2 enthaltenen Constraints erfüllt: [Moti07]

I_1 = {*Türke(Murat), Europäer(Murat)*}
I_2 = {*Türke(Murat), Asiat(Murat)*}

Würde dahingegen nun S_2 durch S_3 ersetzt werden, würde man aus der Vereinigung von A_2 und S_3 folgende beiden *minimal models* erhalten, welche C_2 erfüllen: [Moti07]

$I_3 = I_1 \cup$ {*Mensch(Murat)*}
$I_4 = I_2 \cup$ {Mensch*(Murat)*}

Ein Problem mit *minimal models* und Constraints ergibt sich dann, wenn man ausdrücken will, dass sich ein Individuum von einem anderen Individuum unterscheidet. Normalerweise würde dies mit einer so genannten *Unique Name Assumption* ausgedrückt werden, welche jedoch von OWL nicht gemacht wird. Motik, Horrocks und Sattler lösen dieses Problem dadurch, dass sie die Vereinigung von *A* und *S* in einem *Herbrand Modell* interpretieren. In einem *Herbrand Modell* wird jede Konstante von sich selbst interpretiert, wobei ≈ eine Kongruenzrelation darstellt. Werfen Sie hierzu einen Blick auf folgendes Beispiel (angelehnt an [Moti07]):

ABOX (A_4)	Standard TBOX (S_4)	Constraint TBOX (C_4)
Mensch(Joe) Tier(Garfield)		Mensch ⊓ Tier ⊑ ⊥

Da hier kein Axiom impliziert, dass *Joe* und *Garfield* als gleiche Individuen interpretiert werden sollen, werden sie defaultmäßig als unterschiedlich angesehen. Dadurch wird zwar der Constraint aus C_4 erfüllt, was jedoch keine optimale Lösung darstellt, da es sich bei *Joe* und *Garfield* durchaus auch um gleiche Individuen handeln kann. Dieses Problem kann durch *Minimization* von ≈ gemeinsam mit der *Minimization* aller Prädikate gelöst werden. Dadurch ergibt sich als einzig gültige Interpretation I_5 das *minimal model {Mensch(Joe), Tier(Garfield)}*, wodurch einerseits der Constraint aus C_4 erfüllt wird und darüber hinaus sichergestellt ist, dass *Joe* und *Garfield* als unterschiedliche Individuen interpretiert werden. [Moti07]

2.3.3 Typische Anwendung von Integrity Constraints

Participation Constraints

Diese ICs bestehen aus zwei Konzepten (*C, D*) und einer Relation (*R*), die diese beiden Konzepte in Beziehung zueinander setzt. Sie sind mit Inklusionsabhängigkeiten in relationalen Datenbanken vergleichbar. Sie legen fest, dass jede Instanz von *C* mit einer oder mehreren Instanzen von *D* in einer *R*-Beziehung stehen muss. Außerdem geben sie die Kardinalität der Relation vor, wobei nur Werte größer oder gleich 0 erlaubt sind. Werfen Sie dazu einen Blick auf das folgende Beispiel (angelehnt an [Moti07]):

ABOX (A_5)	(7)
Mensch(Murat) hatStaatsbürgerschaft(Murat,Türkei) hatStaatsbürgerschaft(Murat,Belgien)	Mensch ⊑ ≤1 hatStaatsbürgerschaft.Land

Durch (7) wird ausgedrückt, dass ein *Mensch* maximal die *Staatbürgerschaft* eines *Landes* haben darf (Doppelstaatsbürgerschaften werden in diesem Beispiel nicht

berücksichtigt). Würde (7) nun ein Teil einer Standard TBOX sein, würde $A \cup S$ erfüllt werden, da man aus (7) *Türkei* $\approx$ *Belgien* erhalten würde. Wäre (7) nun jedoch ein Axiom einer Constraint TBOX, so würde das *minimal model* aus $A \cup S$ exakt die gleichen Fakten enthalten wie A_5. Auch $\approx$ würde minimiert werden, wodurch klar wird, dass es sich bei den beiden *Ländern Türkei* und *Belgien* um unterschiedliche Instanzen handelt. Dadurch würde der Constraint verletzt werden, da *Murat* nicht die *Staatsbürgerschaft* von zwei *Ländern* haben darf, womit der Constraint auch richtig interpretiert werden würde. [Moti07]

Typing Constraints

Integrity Constraints können auch dafür verwendet werden, um zu prüfen, ob Objekte vom richtigen Typ sind. Ein typisches Beispiel hierfür sind *domain* und *range* Restriktionen, bei denen der *Typing Constraint* für eine Rolle (*R*) und ein Konzept (*C*) festlegt, dass *R* nur Objekte (von oder zu) verknüpfen kann, die ausdrücklich in der Form von *C* typisiert sind. Werfen Sie dazu einen Blick auf die folgenden beiden Constraints (angelehnt an [Moti07]):

(8) Domain Constraint

$\exists autor.\top \sqsubseteq Buch \sqcup Zeitschrift \sqcup Blog$

(9) Range Constraint

$\top \sqsubseteq \forall hatStaatsbergerschaft.Land$

Während in (8) festgelegt wird, dass *autor* lediglich für Objekte vergeben werden darf, die entweder ein *Buch*, eine *Zeitschrift* oder ein *Blog* sind, wird in (9) festgelegt, dass man ausschließlich die *Staatsbürgerschaft* von *Ländern* haben kann. Würde nun die ABOX lediglich aus *hatStaatsbürgerschaft(Murat,Türkei)* bestehen und (9) ein Axiom einer Standard TBOX sein, würden wir aus (9) *Land(Türkei)* erhalten und $A \cup S$ erfüllen. Handelt es sich bei (9) wiederum um ein Axiom einer Constraint TBOX, würde das *minimal model* lediglich den Fakt aus der ABOX enthalten, woraus nicht eindeutig hervorgeht, dass es sich bei *Türkei* um ein *Land* handelt, wodurch der Constraint verletzt wäre. [Moti07]

Restriktion bekannter (benannter) Individuen

Mittels Constraints können bekannte (benannte) Individuen eingeschränkt werden. In dem in Punkt 2.3.2 erwähnten Beispiel könnte es beispielsweise zwei Arten von Personen geben, nämlich zum einen jene Individuen, die aktive Fußballer eines Vereins darstellen und jene Individuen, welche ihre aktive Karriere bereits beendet haben. Das Vorhandensein einer *SPN* wird in diesem Fall lediglich für die Gruppe der erstgenannten Individuen erforderlich sein. Um nun diese beiden Gruppen von Individuen voneinander unterscheiden zu können, könnte ein neues Konzept eingeführt werden, dass die erste Gruppe beschreibt und eine Sub-Menge von *Person* darstellt. [Moti07]

$$\text{AktiverSpieler} \sqsubseteq \text{Person}$$

Für alle im Konzept *AktiverSpieler* zusammengefassten Personen muss gelten, dass sowohl *Person* als auch die *SPN* dieser Person bekannt sind. ICs können zwar die Existenz von bestimmten Individuen feststellen, jedoch können sie dabei nicht zwischen bekannten (benannten) und unbekannten (unbenannten) Individuen unterscheiden. Motik, Horrocks und Sattler bedienen sich hier eines Tricks, um dieses Problem zu umgehen. Sie führen ein neues (spezielles) Konzept *O* ein, dass alle benannten Individuen bezeichnet und formulieren die folgenden beiden Constraints: [Moti07]

(10) $\text{AktiverSpieler} \sqsubseteq O$

(11) $\text{AktiverSpieler} \sqsubseteq \exists \text{hatSPN}.(O \sqcap \text{SPN})$

Durch das Hinzufügen der zuvor definierten Assertion *O(a)* zu jedem Individuum *a* der ABOX wird für jedes *minimal model* aus $A \cup S$ sichergestellt, dass *O* als Menge jener Objekte interpretiert wird, welche auch bekannt sind. Damit stellt (10) sicher, dass jeder *AktiverSpieler* und (11) dass jede *SPN* ebenfalls bekannt ist. [Moti07]

2.4 M-Objects und M-Relationships

Da ein zentraler Bestandteil dieser Arbeit M-Objects und M-Relationships sind, erfolgt nun eine kurze Einführung darin, was man unter ihnen versteht und welchen Zweck sie erfüllen. Daran folgend wird das Beispiel aus [Neum09b] vorgestellt, welches auch in Kapitel 3 verwendet wird. Für eine tiefer gehende Einführung in M-Objects und M-Relationships verweise ich auf [Neum09a] und [Neum10b]. Für den Einsatz von M-Objects und M-Relationships für die Modellierung von Data Warehouses verweise ich auf [Neum10a].

2.4.1 Einführung

Die Modellierung von Objekten auf unterschiedlichen Abstraktionsebenen hat in den letzten Jahren zusehends an Bedeutung gewonnen. Die Organisation von Objekten erfolgt häufig in Hierarchien, welche aus mehreren Levels beziehungsweise Ebenen bestehen. Dabei kann es sich um Produkthierarchien, Dimensionshierarchien oder Taxonomien im Allgemeinen handeln. Ein *ProductCatalog* könnte beispielsweise aus den drei Levels *category*, *model* und *physical entity* mit den Instanzen *Book*, *Illuminati* und *myCopyOfIlluminati* bestehen. Sind diese Objekte auf jedem Level einheitlich beziehungsweise besitzen dieselbe Struktur, stellt sich die Konzeptmodellierung derartiger Hierarchien einfach dar. Unterscheiden sich jedoch Objekte, welche auf dem gleichen Level liegen, in ihren Subhierarchien, ist die Modellierung nicht mehr ganz so unkompliziert. Nehmen Sie die beiden *ProductCategories Book* und *Car*. Beide könnten gemeinsam haben, dass eine *taxRate* für sie zu zahlen ist. Jedoch können sie sich beispielsweise auf der Sub-Ebene *ProductModel* dahingehend unterscheiden, dass ein *Book* die Eigenschaften *author*, nrOf*Pages*, usw. hat, während ein *Car* auf dieser Ebene eine *maxSpeed* als Eigenschaft aufweist. [Neum09a]

Objekte können sich also einerseits aufgrund ihrer Eigenschaften unterscheiden. Darüber hinaus können sich Objekte desselben Levels dadurch unterscheiden, dass in der Subhierarchie eines Objekts ein zusätzlicher Level eingefügt wird, beispielsweise könnte für *Car* ein Level namens *brand* eingeführt werden. Jedoch enthält nur die Subhierarchie von *Car* diesen Level, nicht jedoch die Subhierarchie von *Book*. [Neum09a]

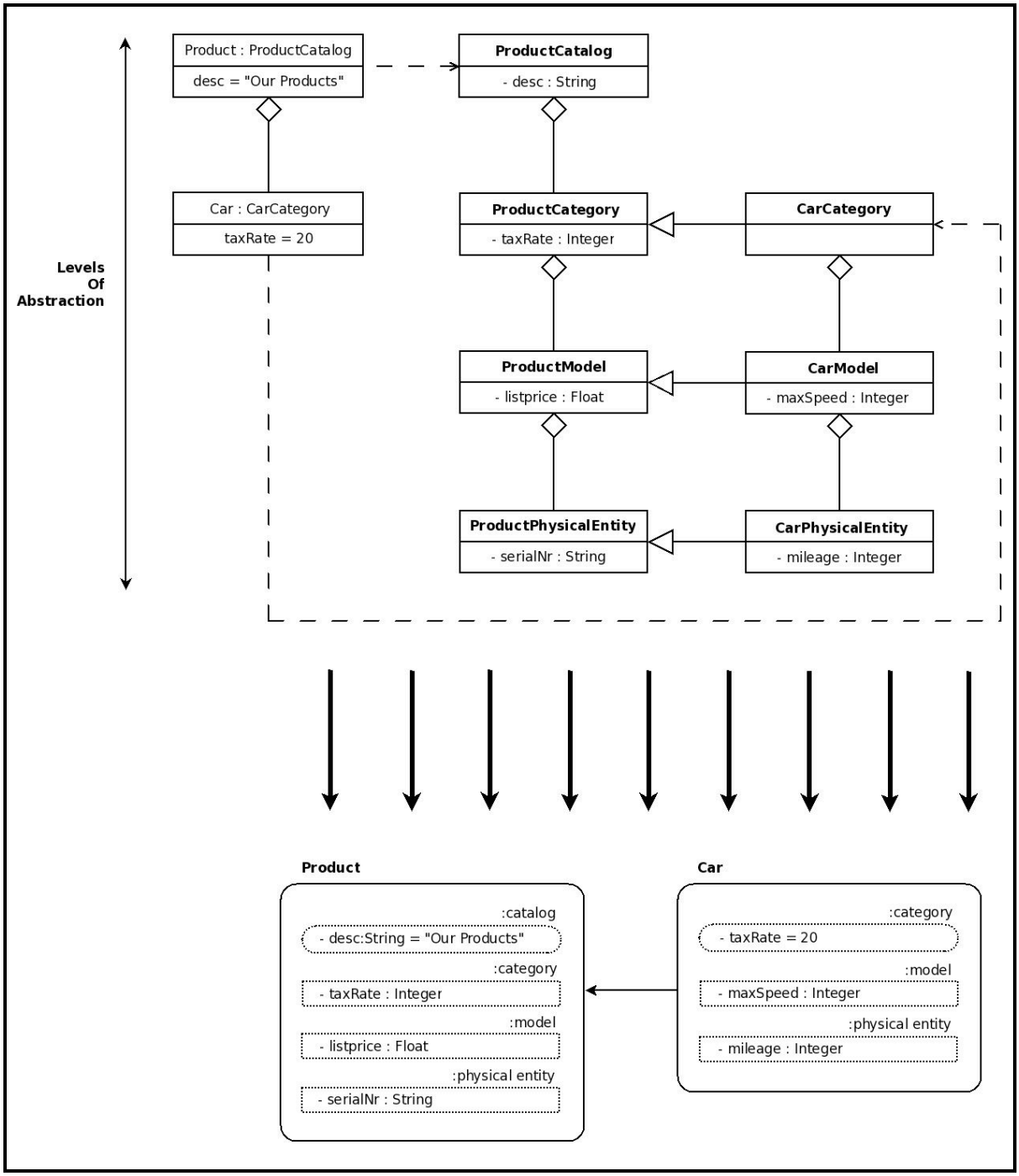

Abbildung 2: M-Objects [Neum09a]

Derartige Hierarchien können, wie in Abbildung 2 oberhalb zu sehen ist, in UML durch Aggregation, Generalisierung und Klassifikation dargestellt und modelliert werden. Erhöht sich nun die Komplexität des Modells, beispielsweise durch die Einführung eines neuen Konzepts für einen bestimmten Bereich, dann wird offensichtlich, dass die Erweiterbarkeit dieser Darstellung schnell an ihre Grenzen stößt. Es müssten für jedes neue Konzept eine Instanz und mehrere Klassen eingeführt werden,

zusammen mit Aggregationen, Generalisierungen und Instanzierungsbeziehungen, was einen erheblichen Wartungsaufwand des Modells bedeuteten würde. [Neum09a]

In den letzten Jahren wurden *materialization*, *powertypes* und *deep instantiation* als Techniken vorgeschlagen, um die Komplexität derartiger Modelle zu reduzieren. Diese Techniken erleichtern zwar einerseits die Multi-Level Modellierung und unterstützen die Tiefencharakterisierung, jedoch unterstützen sie keine Subhierarchien, wenn diese nicht einheitlich sind. Multi-Level-Objekte und Multi-Level-Beziehungen bauen auf diesen Techniken auf und ermöglichen eine natürlich intuitive Darstellung der Konkretisierung von Objekten und Beziehungen entlang mehrerer Abstraktionsebenen. [Neum09a] Für einen Vergleich dieser Techniken verweise ich auf [Neum11].

Multi-Level-Objekte fassen Abstraktionsebenen zusammen und ordnen diese vom abstraktesten zum konkretesten Level. Dabei beschreiben sie sich selbst, indem sie für die Eigenschaften auf der Ebene des Top-Levels Werte spezifizieren. Darüber hinaus beschreiben sie jene Eigenschaften, die sie mit den Objekten ihrer Subhierarchie gemeinsam haben. Konkretisiert nun ein Objekt ein anderes Objekt, so enthält dieses Objekt alle Level vom übergeordneten Objekt mit Ausnahme des Top-Levels. [Neum09a]

Die Konkretisierungsbeziehung zwischen zwei M-Objects wird vielfältig interpretiert. In Abbildung 2 (im unteren Teil der Abbildung zu sehen) stellt das M-Object *Car* eine Instanz des M-Objects *Product* dar und bezieht sich dabei auf den zweiten Top-Level von *Product*, nämlich *category*, und gibt dabei Werte für die Attribute dieser Ebene vor (in diesem Fall für *taxRate*). Dabei stellen *category*, *model* und *physical entity* des M-Objects *Car* Subklassen des jeweiligen Levels des M-Objects *Product* dar. Darüber hinaus können Beziehungen auf unterschiedlichen Ebenen beschrieben werden. Solche M-Relationships können unterschiedliche Rollen einnehmen, abhängig davon, welche Levels und M-Objects sie verbinden. In Abbildung 3 wird unterhalb gezeigt, wie die M-Relationship *producedBy* die beiden M-Objects *Product* und *Company* in Beziehung setzt. Man achte darauf, dass nicht nur die beiden Objekte, sondern auch die Abstraktionsebenen *category* und *industrial sector*, *model* und *enterprise* und zu guter Letzt *physical entity* und *factory* in Beziehung gesetzt werden. [Neum09a]

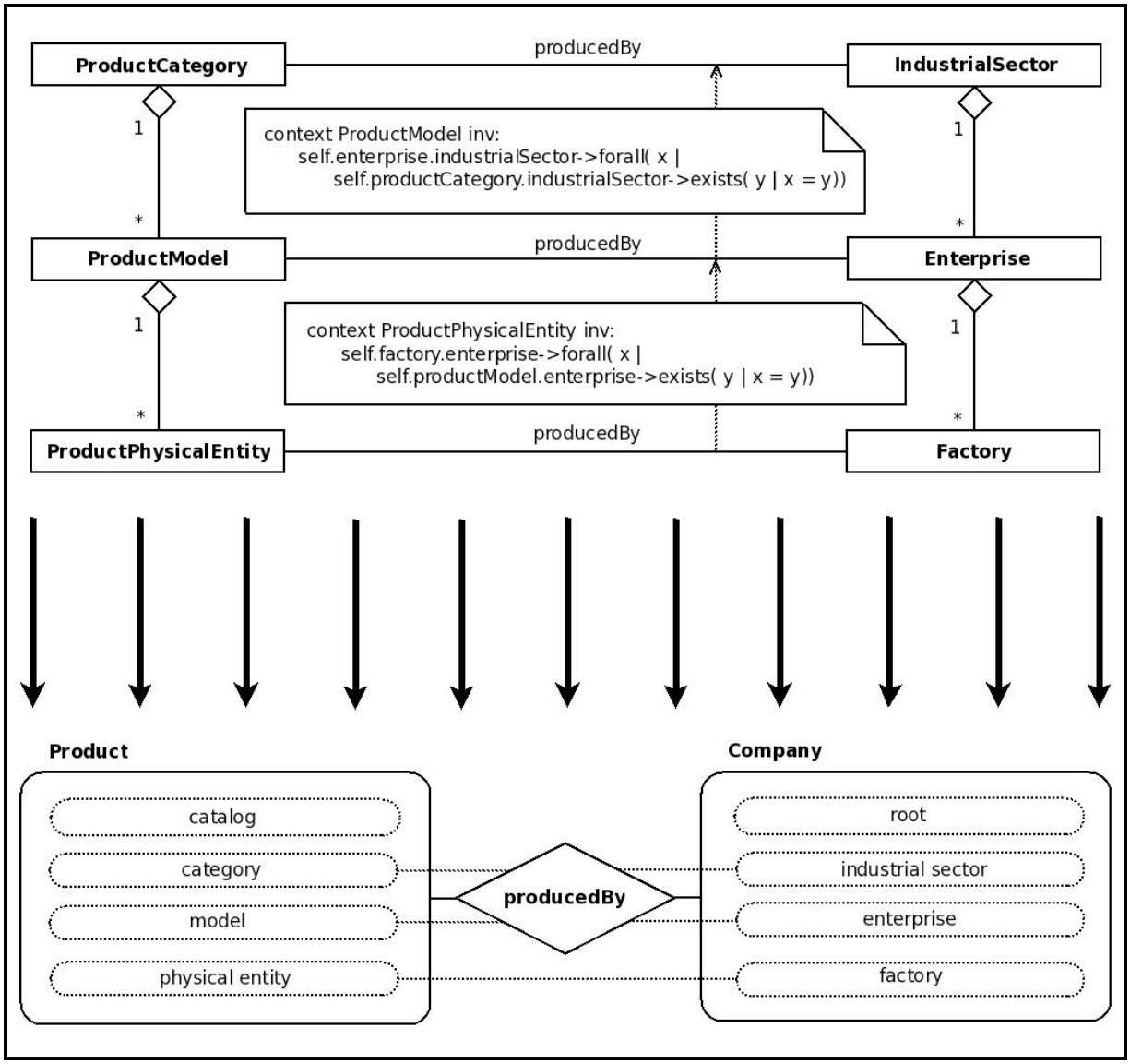

Abbildung 3: M-Relationships [Neum09a]

Durch M-Objects und M-Relationships soll folgendes erreicht werden:

- **Unterschiedliche Abstraktionsebenen:** Es soll möglich sein, Objekte auf unterschiedlichen Abstraktionsebenen zu beschreiben.

- **Erweiterbarkeit:** Multi-Level Modellierung soll ein hohes Maß an Erweiterbarkeit im Hinblick auf die Modellierung von unterschiedlichen Subhierarchien bieten, beispielsweise wenn Levels, Beziehungen oder Attribute hinzugefügt werden.

- **Vermeidung von Fragmentierung und Redundanz:** Jede Information über ein Bereichsobjekt soll lokal zu diesem Objekt beschrieben werden um Redundanz und eine Trennung der Information zu vermeiden.

- **Beziehungen:** Der Ansatz soll die Modellierung von Beziehungen zwischen Objekten und deren Spezifizierung auf unterschiedlichen Abstraktionsebenen unterstützen.

- **Abfragen:** Um mit Multi-Level Modellen arbeiten zu können, müssen Abfragemöglichkeiten unterstützt werden. [Neum09a]

2.4.2 Beispiel

Da ich mich im Laufe dieser Arbeit sehr häufig auf das Beispiel aus [Neum09b] beziehe, dient dieser Punkt dazu, dieses Beispiel vorzustellen. Außerdem soll die folgende Erläuterung dabei helfen, den in Punkt 2.5 vorgestellten Mapping Algorithmus anhand eines konkreten Beispiels erklären zu können. In Abbildung 4 wird dieses veranschaulicht.

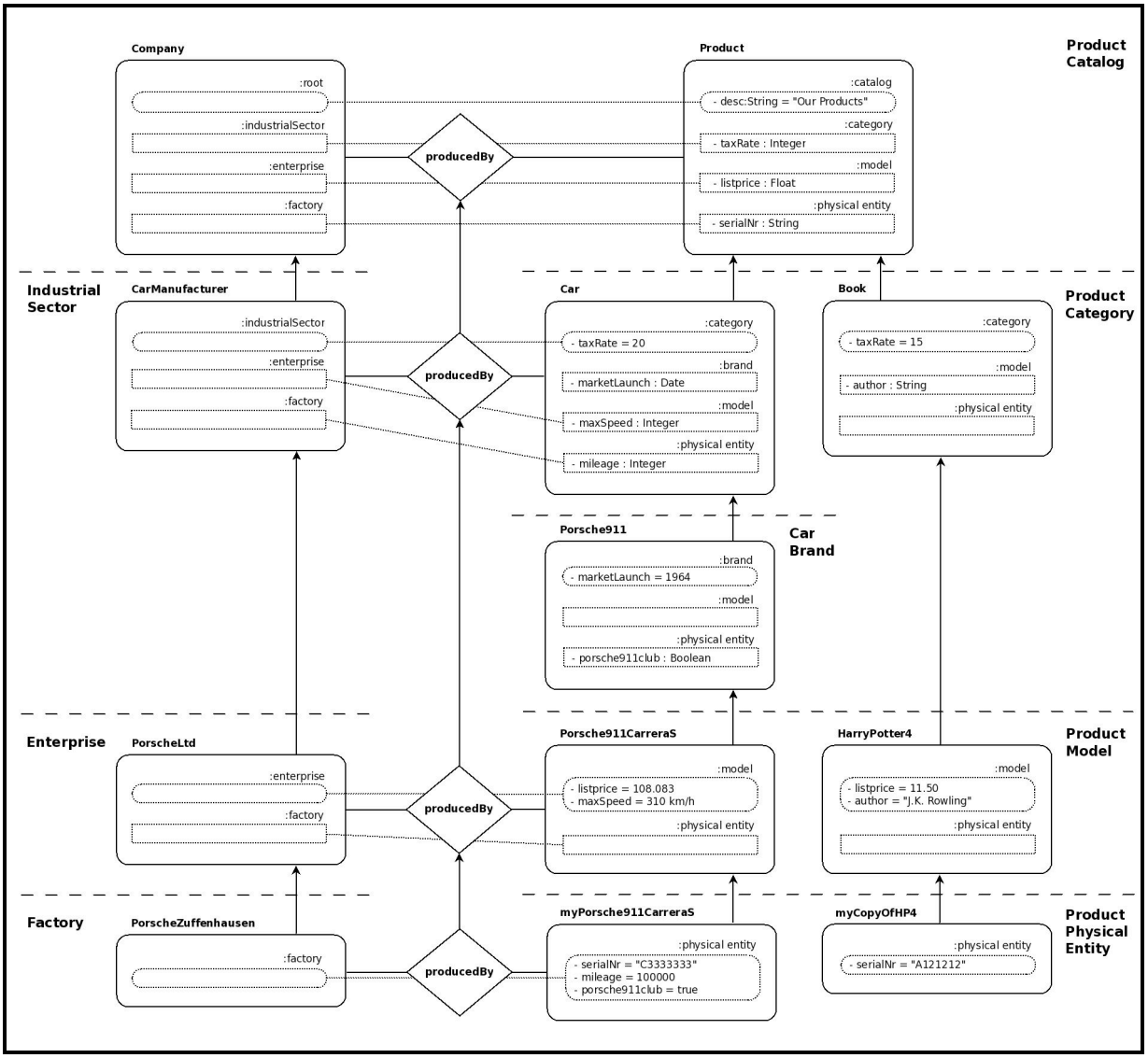

Abbildung 4: Beispiel [Neum09b]

Wie in der Abbildung zu sehen ist wird ein *ProductCatalog* auf drei Abstraktionsebenen (Levels) beschrieben, nämlich *category*, *model* und *physical entity*. Dabei hat jede *ProductCategory* eine *taxRate*, jedes *ProductModel* einen *listprice* und jede *physical entity* von *Product* eine *serialNr* als Eigenschaft. [Neum09b]

Die beiden M-Objects *Book* und *Car* stellen Konkretisierungen von *Product* dar und erben somit die Abstraktionsebenen dieses übergeordneten M-Objects. Sie haben gemeinsam, dass sie auf dem Level *ProductCategory* das Attribut *taxRate* aufweisen, jedoch mit unterschiedlicher Ausprägung (15 bzw. 20). Sie unterscheiden sich weiters darin, dass *Book* auf Ebene des *ProductModels* zusätzlich zum *listprice* das Attribut *author*, *Car* hingegen zusätzlich das Attribut *maxSpeed* aufweist. Auf Ebene der *physical entity* ändert sich bei *Book* nichts. *Car* hingegen hat auf diesem Level

zusätzlich zum *listprice* die beiden Attribute *mileage* und *porsche911club*. Der letzte Unterschied zwischen diesen beiden M-Objects liegt darin, dass bei *Car* ein zusätzlicher Level mit dem Namen *brand* eingeführt wurde, welcher wiederum die Eigenschaft *marketLaunch* aufweist. [Neum09b]

Weiters ist ein *Product* mit einer *Company* verbunden, welches es herstellt. Eine *Company* wird ebenfalls auf mehreren Abstraktionsebenen beschrieben, welche in diesem Fall *industrialSector*, *enterprise* und *factory* lauten. Eine *Company*, welche ein Produzent von *Car* ist, gehört zu einem spezifischen *industrialSector*, der hier als *CarManufacturer* bezeichnet wurde. Auf Ebene der *physical entity* von *Car* wird auch eine Verbindung zu jener *factory* hergestellt, in welcher *Car* produziert wurde, um beispielsweise später eventuelle Qualitätsmängel auf genau jene *factory* zurückführen zu können. Dazu muss *factory* zu jener *Company* gehören, die das betreffende *CarModel* produziert. [Neum09b]

Das ganze könnte natürlich noch weitergeführt werden. Beispielsweise ist *HarryPotter4* eine Konkretisierung von *Book*. Da *Book* wiederum eine Konkretisierung von *Product* darstellt, konkretisiert *HarryPotter4* indirekt auch *Product*. Das M-Object *myCopyOfHP4* konkretisiert dann direkt oder indirekt *Product*, *Book* und *HarryPotter4*.

Wie unschwer zu erkennen ist, werden immer alle Abstraktionsebenen von einem übergeordneten an ein oder mehrere untergeordnete M-Objects vererbt, mit Ausnahme des Top-Levels des übergeordneten M-Objects. Es ergibt sich eine lineare Ordnung der Abstraktionsebenen entlang der M-Objects vom abstraktesten bis hin zum konkretesten. Die Attribute des Top-Levels eines jeden M-Objects werden durch konkrete Werte spezifiziert. Der Sinn eines Top-Levels liegt darin, das betreffende M-Object selbst zu beschreiben. So ist beispielsweise bei *Car* der entsprechende Top-Level *Category*, welcher das Attribut tax*Rate* mit dem konkreten Wert 15 enthält. Alle anderen Abstraktionsebenen beschreiben lediglich Attribute, welche untergeordnete M-Objects gemeinsam haben. [Neum09b]

Aus diesem Beispiel ergeben sich folgende M-Objects, Abstraktionsebenen und M-Relationships:

M-Objects	Product, Book, Car, Porsche911, HarryPotter4, Porsche911CarreraS, myCopyOfHP4, myPorsche911CarreraS, Company, CarManufacturer, PorscheLtd, PorscheZuffenhausen
Abstraktions-ebenen	catalog, category, model, physical entity, brand, root, industrialSector, enterprise, factory
M-Relationships	Product-producedBy-Company, Car-producedBy-CarManufacturer, Porsche911CarreraS-producedBy-PorscheLtd, myPorsche911CarreraS-producedBy-PorscheZuffenhausen

Tabelle 2: M-Objects, Abstraktionsebenen und M-Relationships [Neum09b]

2.5 Mapping von M-Objects und M-Relationships nach OWL

Aufbauend auf der Einführung in M-Objects und M-Relationships, welche Schrefl und Neumayr in [Neum09a] geben und worauf ich mich in 2.4.1 bereits bezogen habe, beschreiben sie in [Neum09b] ihren Algorithmus für das Mapping von M-Objects beziehungsweise M-Relationships nach OWL.

2.5.1 Mapping der M-Objects

Die erste Hälfte ihres Algorithmus beschäftigt sich dabei mit dem Mapping der M-Objects nach OWL und wird in Abbildung 5 veranschaulicht. Die darin enthaltenen Variablen sind folgendermaßen zu verstehen: [Neum09b]

o ... M-Object, bestehend aus L_o und A_o

o' ... übergeordnetes M-Object, welches von o konkretisiert wird

a ... Attribut

l ... Level

l' ... übergeordneter Level von Level l

$\hat{l}$ ... Top-Level

l_o ... Funktion, die Attribute von o zu Levels zuordnet (l_o: $A_o \rightarrow L_o$)

$l_o(a)$ ... Attribut-Level-Zuordnung von Attribut a bei Objekt o

d_o ... Funktion, die Attributen von o einen Datentyp zuordnet (d_o: $A_o \rightarrow D$)

$d_o(a)$ ... Attribut-Datentyp-Zuordnung des Datentyps von Attribut a bei Objekt o

v_o ... Funktion, die Attributen von o einen Wert zuweist

$v_o(a)$ ... Attribut-Wert-Zuweisung des Werts von Attribut a bei Objekt o

L_o ... Menge der Levels (entnommen aus L) von o

A_o ... Menge der Attribute (entnommen aus A) von o

$\hat{A}_o$ ... Menge der Top-Level-Attribute von o

P_o ... Eine Relation zur Darstellung der Level-Hierarchie im M-Object o

O ... Menge von M-Objects

L ... Universum von Levels

H ... Konkretisierungshierarchie

A ... Universum von Attributen

D ... Universum von Datentypen

1: **assert:** $\top \sqsubseteq \leq 1$ concretize

2: **assert:** concretize $\sqsubseteq$ concretize_t

3: **assert:** concretize_t^{+} $\sqsubseteq$ concretize_t

4: **for all** $o \in O$ **do**

5: **assert:** [$\hat{l}_o$] ([o])

6: **if** $\exists o'$: (o,o') $\in H$ **then assert:** concretize([o],[o'])

7: **for all** $a \in \hat{A}_o$: $v_o(a)$ is defined **do assert:** [a]([o],[$v_o(a)$])

8: **for all** $a \in (A_o \setminus \hat{A}_o)$ **do**

9: **assert IC:** $\exists$concretize_t.{[o]} $\sqcap$ [$l_o(a)$] $\sqsubseteq$ $\forall$[a].[$d_o(a)$] $\sqcap$ $=1$ [a].$\top$

10: **if** $v_o(a)$ is defined **then assert:** [$l_o(a)$] $\sqcap$ $\exists$concretize_t.{[o]} $\sqsubseteq$ $\exists$[a].{[$v_o(a)$]}

11: **for all** (l,l') $\in P_o$: $l' \neq \hat{l}_o \wedge (\nexists o' \in O : (o,o') \in H \wedge (l,l') \in P_{o'})$ **do**

12: **assert IC:** $\exists$concretize_t.{[o]} $\sqcap$ [l] $\sqsubseteq$ $\exists$concretize_t.($\exists$concretize_t.{[o]} $\sqcap$ [l'])

13: **for all** $a \in A_o$: $\nexists o' \in O : (o,o') \in H \wedge a \in A_{o'}$ **do**
14: **assert IC:** $\exists[a].\top \sqsubseteq (\exists \text{concretize\_t}.\{[o]\} \sqcup \{[o]\}) \sqcap [l_o(a)]$
15: **for all** $l \in L_o : l \neq \hat{l}_o \wedge (\nexists o' \in O : (o,o') \in H \wedge l \in L_{o'})$ **do**
16: **assert IC:** $[l] \sqsubseteq \exists \text{concretize\_t}.\{[o]\}$
17: **for all** $l \in L$, $l' \in (L \setminus \{l\})$ **do assert:** $[l] \sqcap [l'] \sqsubseteq \bot$
18: **for all** $o \in o, o' \in (O \setminus \{o\})$ **do assert:** $[o] \not\approx [o']$

Abbildung 5: Mapping von M-Objects nach OWL [Neum09b]

M-Objects werden in OWL durch Individuen (beispielsweise *Car*) und Abstraktionsebenen als primitive Klassen (beispielsweise *Category*) dargestellt. Dabei ist jedes M-Object mittels einer Klassenassertion einer Abstraktionsebene zugewiesen. Untergeordnete M-Objects werden durch die funktionale Eigenschaft *concretize* mit übergeordneten M-Objects verbunden, beispielsweise ist *Book* ein Mitglied der Klasse *Category* und konkretisiert *Product*. Im Mapping Algorithmus wird dies durch die Zeilen 5 und 6 umgesetzt und würde in Bezug auf das Beispiel aus Abbildung 4 folgenden Mapping-Output erzeugen: [Neum09b]

5: Category(Car)
6: concretize(Car,Product)

Werte von Top-Level-Attributen werden in OWL durch Eigenschaftsassertionen dargestellt, beispielsweise hat die *ProductCategory Car* eine *taxRate* in der Höhe von 20. Im Mapping Algorithmus wird dies in Zeile 7 realisiert und würde folgenden Mapping-Output liefern: [Neum09b]

7: taxRate(Car,20)

Das Axiom $\exists$concretize_t.{[*o*]} $\sqcap$ [*l*] stellt eine Klasse von Individuen dar, welche ein bestimmtes Abstraktionslevel besitzen und direkt oder indirekt eine Konkretisierung eines bestimmten Individuums darstellen. Die Klasse aller *CarModels* würde daher durch $\exists$concretize_t.{Car} $\sqcap$ Model ausgedrückt werden. Dabei stellt *concretize_t* die transitive „Supereigenschaft“ von *concretize* dar. Dadurch, dass in Zeile 6 zuvor festgelegt wurde, dass *Car* eine Konkretisierung von *Product* darstellt, kann der Reasoner schließen, dass $\exists$concretize_t.{Car} $\sqcap$ Model und $\exists$concretize_t.{Car} $\sqcap$ Physica-

lEntity von ∃concretize_t.{Product} ⊓ Model bzw. ∃concretize_t.{Product} ⊓ PhysicalEntity subsummiert werden. Durch diese tolle Eigenschaft von OWL werden sowohl Klassenvererbungen als auch Konsistenzprüfungen erleichtert. [Neum09b]

Gemeinsame Charakteristiken der Mitglieder eines bestimmten Levels eines M-Objects werden durch so genannte Subklassenaxiome einschließlich der jeweiligen Klassen vor dem Subklassensymbol angezeigt. Attribute werden durch Dateneigenschaften mit dazugehörigen Werten und Wertrestriktionen dargelegt und stehen nach dem Subklassensymbol. Dies wird in den Zeilen 8 – 9 umgesetzt. Die Werte der Attribute können mit M-Objects auf darunterliegenden Ebenen geteilt werden, was durch Zeile 10 im Mapping Algorithmus ausgedrückt wird. Um die Semantik des M-Object-Ansatzes zu bewahren, werden die Axiome als Integrity Constraints interpretiert. Die folgenden 3 Zeilen zeigen beispielhaft für das M-Object *Car*, wie der Mapping-Output von Zeile 9 des Mapping Algorithmus aussehen würde. [Neum09b]

9: **IC:** ∃concretize_t.{Car} ⊓ Brand ⊑ ∀marketLaunch.Date ⊓ =1 marketLaunch.⊤
9: **IC:** ∃concretize_t.{Car} ⊓ Model ⊑ ∀maxSpeed.Integer ⊓ =1 maxSpeed.⊤
9: **IC:** ∃concretize_t.{Car} ⊓ PhysicalEntity ⊑ ∀mileage.Integer ⊓ =1 mileage.⊤

In den Zeilen 11 – 12 wird folgendes umgesetzt: „ein Level *l* eines M-Objects *o* stellt sicher, dass Konkretisierungen von *o* auf darunterliegenden Ebenen ebenfalls eine Konkretisierung von *o* auf dem Level *l* konkretisieren.“ Dadurch wird eine stabile Aufwärtsnavigation erlaubt und heterogene Levelhierarchien unterstützt, da es ermöglicht wird, dass ein M-Object neue Abstraktionsebenen einführt, welche jedoch nur für Nachkommen dieses M-Objects gelten. Der Mapping-Output von Zeile 12 würde für *Car* folgendermaßen aussehen: [Neum09b]

12: **IC:** ∃concretize_t.{Car} ⊓ Model ⊑ ∃concretize_t.(∃concretize_t.{Car} ⊓ Brand)

Der Mapping Algorithmus stellt darüber hinaus in den Zeilen 13 – 14 sicher, dass ein Attribut lediglich in ein einziges Level eines M-Objects aufgenommen werden kann, wie die folgenden Zeilen für das M-Object *Car* zeigen. [Neum09b]

14: **IC:** ∃marketLaunch.⊤ ⊑ (∃concretize_t.{Car} ⊔ {Car}) ⊓ Brand
14: **IC:** ∃maxSpeed.⊤ ⊑ (∃concretize_t.{Car} ⊔ {Car}) ⊓ Model
14: **IC:** ∃mileage.⊤ ⊑ (∃concretize_t.{Car} ⊔ {Car}) ⊓ PhysicalEntity

Weiters wird in den Zeilen 15 – 16 des Mapping Algorithmus sichergestellt, dass jeder Level genau zu einem bestimmten M-Object gehört, was durch die folgende Zeile, wiederum für das M-Object *Car*, erreicht wird. [Neum09b]

16: **IC:** Brand ⊑ ∃concretize_t.{Car}

In Zeile 17 wird sichergestellt, dass alle Levels paarweise disjunkt sind. Um den Rahmen nicht zu sprengen, hier nur für den Level *Category* gezeigt. [Neum09b]

17: Category ⊓ Catalog ⊑ ⊥
17: Category ⊓ Model ⊑ ⊥
17: Category ⊓ PhysicalEntity ⊑ ⊥
17: Category ⊓ Brand ⊑ ⊥
17: Category ⊓ Root ⊑ ⊥
17: Category ⊓ IndustrialSector ⊑ ⊥
17: Category ⊓ Enterprise ⊑ ⊥
17: Category ⊓ Factory ⊑ ⊥

Zum Abschluss des Mappings der M-Objects nach OWL wird schließlich in Zeile 18 die Unique Name Assumption sichergestellt, was bedeutet, dass jedes Individuum einen einzigartigen Namen hat und sich von allen anderen Individuen unterscheidet. Hier wiederum beispielsweise für das M-Object *Car* gezeigt. [Neum09b]

18: Car ≉ Product
18: Car ≉ Book
18: Car ≉ Porsche911
18: Car ≉ Porsche911CarreraS
18: Car ≉ MyPorsche911CarreraS
18: Car ≉ HarryPotter4
18: Car ≉ MyCopyOfHP4

18: Car $\not\approx$ Company

18: Car $\not\approx$ CarManufacturer

18: Car $\not\approx$ PorscheLtd

18: Car $\not\approx$ PorscheZuffenhausen

2.5.2 Mapping der M-Relationships

Nachdem das Mapping der M-Objects nach OWL durchgeführt wurde, wird im zweiten Teil des Mapping Algorithmus das Mapping der M-Relationships nach OWL vorgenommen, veranschaulicht in Abbildung 6. Darin bezeichnet *R* eine Menge von M-Relationships $r = (s_r, t_r, C_r)$ und *HR* eine Konkretisierungshierarchie. [Neum09b]

19: **assert:** $\top \sqsubseteq \leq 1 source \sqcap \leq 1 target$

20: **for all** $r \in R$ **do**

21: **if** $\exists r' : (r,r') \in H_R$ **then assert:** concretize([*r*],[*r'*])

22: **assert:** source([*r*],[s_r])

23: **assert:** target([*r*],[t_r])

24: **assert IC:** $\exists$concretize_t.{[*r*]} $\sqsubseteq$ ($\forall$source.($\exists$concretize_t.{[s_r]} $\sqcup$ {[s_r]}) $\sqcap$ $\forall$target.$\exists$concretize_t.{[t_r]}) $\sqcup$ ($\forall$source.$\exists$concretize_t.{[s_r]} $\sqcap$ $\forall$target.($\exists$concretize_t.{[t_r]} $\sqcup$ {[t_r]}))

25: **for all** $(I,I') \in C_r : I \neq \hat{I}_{sr} \vee I' \neq \hat{I}_{tr}$ **do**

26: **assert IC:** $\exists$concretize_t.{[*r*]} $\sqcap$ ($\exists$source.$\exists$concretize_t.[*I*] $\sqcup$ $\exists$target.$\exists$concretize_t.[*I'*]) $\sqsubseteq$ $\exists$concretize_t.($\exists$concretize_t.{[*r*]} $\sqcap$ $\exists$source.[*I*] $\sqcap$ $\exists$target.[*I'*])

27: **for all** $r' \in R : r \neq r'$ **do assert:** [*r*] $\not\approx$ [*r'*]

Abbildung 6: Mapping von M-Relationships nach OWL [Neum09b]

Ich werde nun wiederum Zeile für Zeile des Algorithmus erklären, wobei ich als Beispiel immer auf die Beziehung der M-Objects *Car* und *CarManufacturer* eingehen werde, welche durch die M-Relationship *producedBy* verbunden sind.

M-Relationships werden in OWL als Individuen dargestellt, wobei jedes Individuum mit seiner übergeordneten Beziehung und seinem Ausgangs- und Ziel-M-Object ver-

bunden ist. Dies wird mittels Eigenschaftsassertionen, welche die funktionalen Rollen (Properties) *concretize*, *source* und *target* verwenden, umgesetzt. Der Name der M-Relationship ergibt sich dabei immer aus dem Ausgangs- und Ziel-M-Object und der M-Relationship, welche die beiden M-Objects zueinander in Beziehung setzt. Im angeführten Beispiel würde sich daher der Name *Car-producedBy-CarManufacturer* ergeben. *Source* ist dabei *Car* und *target* ist *CarManufacturer.* Die Beziehung stellt eine Konkretisierung von *Product-producedBy-Company* dar. Die folgenden Zeilen zeigen den Mapping-Output der Zeilen 21 – 23 aus Abbildung 6. [Neum09b]

21: concretize(Car-producedBy-CarManufacturer, Product-producedBy-Company)
22: source(Car-producedBy-CarManufacturer, Car)
23: target(Car-producedBy-CarManufacturer, CarManufacturer)

Alle direkten oder indirekten Konkretisierungen der M-Relationship *Car-producedBy-CarManufacturer* haben eine direkte oder indirekte Konkretisierung von *Car* oder *Car* selbst als Ausgangs-M-Object und eine direkte oder indirekte Konkretisierung von *CarManufacturer* oder *CarManufacturer* selbst als Ziel-M-Object. Entweder *Car* oder *CarManufacturer* müssen konkretisiert sein, was durch die folgende Zeile (Zeile 24 des Mapping Algorithmus) erreicht wird. [Neum09b]

24: **IC:** ∃concretize_t.{Car-producedBy-CarManufacturer} ⊑
(∀source.(∃concretize_t.{Car} ⊔ {Car}) ⊓
∀target.∃concretize_t.{CarManufacturer}) ⊔ (∀source.∃concretize_t.{Car} ⊓
∀target.(∃concretize_t.{CarManufacturer} ⊔ {CarManufacturer}))

Wie bereits bei den Zeilen 11 – 12 des Mappings der M-Objects nach OWL beschrieben wurde, ermöglichen Levels eine Aufwärtsnavigation. Analog dazu stellen Beziehungslevels (*connection levels*) von M-Relationships die Navigation von M-Relationships auf höher gelegenen Ebenen sicher. Zum Beispiel müssen alle M-Relationships die unterhalb des Beziehungslevels (*model*, *enterprise*) liegen und eine Konkretisierung von *Car-producedBy-CarManufacturer* darstellen ebenfalls eine M-Relationship konkretisieren, welche *Car-producedBy-CarManufacturer* auf dem Level (*model*, *enterprise*) konkretisiert. In Abbildung 6 wird das durch die Zeilen 25 – 26 umgesetzt und ist wiederum als Integrity Constraint zu interpretieren. Die folgende

Zeile zeigt abermals den Mapping-Output dieser Zeile für das erwähnte Beispiel. [Neum09b]

26: **IC:** ∃concretize_t.{Car-producedBy-CarManufacturer} ⊓
(∃source.∃concretize_t.Model ⊔ ∃target.∃concretize_t.Enterprise) ⊑
∃concretize_t.(∃concretize_t.{Car-producedBy-CarManufacturer} ⊓
∃source.Model ⊓ ∃target.Enterprise)

Zum Abschluss des Mappings der M-Relationships nach OWL wird in Zeile 27 die Unique Name Assumption der M-Relationships sichergestellt, was sich im Output folgendermaßen darstellt: [Neum09b]

27: Car-producedBy-CarManufacturer ≉ Product-producedBy-Company
27: Car-producedBy-CarManufacturer ≉ Porsche911CarreraS-producedBy-PorscheLtd
27: Car-producedBy-CarManufacturer ≉ myPorsche911CarreraS-producedBy-PorscheZuffenhausen

2.6 Protégé

Protégé wurde vom Stanford Center for Biomedical Informatics Research entwickelt. Es handelt sich dabei sich um eine Open Source Plattform, welche eine Vielzahl an Werkzeugen zur Verfügung stellt um Modelle und wissensbasierte Anwendungen mittels Ontologien zu bauen. Es unterstützt die Erstellung, die Visualisierung und das Manipulieren von Ontologien durch unterschiedliche Repräsentationen. Kurz gesagt handelt es sich bei Protégé um einen „Ontologieeditor“. Die Modellierung wird dabei in zwei Formen unterstützt: [Prot10a]

- **Protégé Frames Editor:** ermöglicht es, frame-basierte Ontologien zu entwickeln. [Prot10a]
- **Protégé OWL Editor:** ist darauf hin ausgelegt, Ontologien für das semantische Web mittels OWL zu erstellen. [Prot10a]

Weitere Informationen zu Protégé und den von mir verwendeten Programmversionen finden Sie im Anhang.

2.6.1 Protégé Frames

Eine Ontologie besteht in Protégé Frames aus Klassen, Slots, Facets und Axiome. Die Wissensbasis beinhaltet dabei zum einen die Ontologie und zum anderen Instanzen von Klassen mit bestimmten Werten für die Slots. Bei Klassen handelt es sich dabei um Konzepte einer bestimmten Domäne. Während die Slots die Eigenschaften und Attribute dieser Klassen beschreiben, werden durch Facets wiederum die Eigenschaften der Slots beschrieben. Zu guter Letzt werden mittels Axiome die Nebenbedingungen spezifiziert. [Noy00] Ich werde nun etwas konkreter auf die einzelnen Bestandteile von Protégé Frames eingehen.

Klassen und Instanzen

Klassen werden in Protégé Frames in einer taxonomischen Hierarchie abgebildet, in der die standardmäßig vordefinierte Klasse *:THING* immer die Wurzelklasse darstellt. Ist nun zum Beispiel die Klasse *Redakteur* eine Subklasse der Klasse *Mitarbeiter*, so stellt jede Instanz von *Redakteur* gleichzeitig eine Instanz der Klasse *Mitarbeiter* dar. Derartige Subklassenbeziehungen werden in Protégé in Form eines Baums visualisiert. [Noy00]

Darüber hinaus unterstützt Protégé Frames Mehrfachvererbung, was bedeutet, dass eine Subklasse durchaus mehr als nur eine Superklasse aufweisen kann. Zum Beispiel könnte *Redakteur* sowohl eine Subklasse von *Mitarbeiter* als auch von *Autor* darstellen, wenn davon ausgegangen wird, dass ein *Redakteur* sowohl ein *Mitarbeiter* als auch ein *Autor* einer bestimmten Zeitung ist. In Protégé Frames können sowohl Instanzen als auch Klassen Instanzen einer Klasse sein. Hat eine Klasse A wiederum eine Klasse B als Instanz, handelt es sich bei der Klasse A um eine so genannte Metaklasse. [Noy00]

Slots

Slots beschreiben Eigenschaften von Klassen und Instanzen, beispielsweise den Namen eines Mitarbeiters. Beim Slot selbst handelt es sich um einen Frame, der unabhängig von einer Klasse definiert wird. Wird ein Slot zu einer Ontologie hinzugefügt, so kann dieser mit einem Wert versehen werden. Würde zum Beispiel der Slot *Name* zu einer Ontologie hinzugefügt werden, so könnte dieser Slot anschließend zu den Klassen *Zeitung* und *Mitarbeiter* hinzugefügt werden (vorausgesetzt, die beiden Klassen wurden ebenfalls zuvor zu der Ontologie hinzugefügt). Der Slot kann anschließend bei *Zeitung* mit dem Namen der Zeitung (z.B. dem String „New York Times“) und bei Mitarbeiter mit dem Namen des Mitarbeiters (z.B. dem String „Huber“) „befüllt“ werden. Daraus kann geschlossen werden, dass ein und derselbe Slot durchaus für mehrere Klassen verwendet werden kann. [Noy00]

Es gibt zwei Arten von Slots: *own slots* und *template slots*. *Own slots* beschreiben Eigenschaften von Klassen und Instanzen, wobei diese Slots nicht von den Subklassen (und deren Instanzen) geerbt werden. *Template slots* hingegen können lediglich zu Klassen hinzugefügt werden und beschreiben die Eigenschaften der Klasseninstanzen. Im Gegensatz zu *own slots* erben hier die Subklassen dieser Klasse den entsprechenden Slot. Bei den Instanzen dieser Klasse wird der *template slot* wiederum zu einem *own slot*. [Noy00]

Facets

Durch Facets können erlaubte Werte für Slots festgelegt werden. Zum Beispiel kann mittels Kardinalitäten und Restriktionen festgelegt werden, wie viele Werte ein gewisser Slot aufnehmen darf und von welchem Datentyp (String, Integer, Float, usw.) diese Werte sein müssen. Darüber hinaus kann bei numerischen Werten ein Minimum- bzw. Maximum-Wert festgelegt werden. Nehmen Sie als Beispiel den Slot *Gehalt*. Für diesen könnte festgelegt werden, dass der darin enthaltene Wert nicht kleiner als 15.000 sein darf und der Wert vom Datentyp Float sein muss. Protégé Frames bietet in diesem Bereich noch eine Vielzahl weiterer Möglichkeiten. [Noy00]

Formulare / Masken

Ein weiterer zentraler Bestandteil von Protégé Frames sind Formulare / Masken (Forms). Solche Formulare ermöglichen es Informationen zu Instanzen schnell und einfach zu erfassen und anzuzeigen. Wird beispielsweise zu einer Klasse ein *template slot* hinzugefügt, so generiert Protégé Frames automatisch ein Formular für die Instanzen dieser Klasse. Dabei bestimmen die Slots sowie deren Kardinalitäten und Datentypen das standardmäßige Layout und den Inhalt des Formulars. Benutzer können diese automatisch generierten Formulare für jede Klasse nach ihren Wünschen abändern, um ihren Anforderungen besser zu entsprechen. Auch in diesem Bereich bietet Protégé Frames eine Reihe von Möglichkeiten, beispielsweise kann das Layout des Formulars verändert oder eine andere Anzeige der in den Slots vorhandenen Werte gewählt werden, usw. [Noy00]

2.6.2 Protégé OWL

Protégé OWL unterstützt, wie der Name schon sagt, die Webontologiesprache OWL und stellt eine Erweiterung von Protégé dar. Das Programm bietet einem Anwender folgende Features: [Prot11]

- Laden und speichern von OWL- und RDF-Ontologien
- Erstellung und Visualisierung von Klassen, Rollen (Eigenschaften) und SWRL Regeln
- Definition von logischen Klassen-Charakteristiken als OWL-Ausdrücke
- Ausführen von Reasonern wie Pellet, HermiT, Fact++, Racer, usw.
- Erstellung von OWL Individuals (Individuen) für das Semantic Web

Durch die gebotene Update-Funktion lässt sich das Programm nach eigenen Wünschen durch eine Vielzahl von Plugins erweitern. Aufgrund der Open Source Java API wird es Anwendern darüber hinaus ermöglicht, eigene Plugins und Interfaces für das Programm zu schreiben. [Prot11] Für eine intensivere Betrachtung von Protégé OWL und dessen Handhabung zur Modellierung von OWL Modellen verweise ich auf [Knub04a], [Knub04b] sowie [Knub04c].

2.6.3 Protégé OWL API vs. OWL API 3

Da ich leider im Laufe der Programmierung für diese Arbeit feststellen musste, dass es mit der API von Protégé OWL nicht möglich war, den Mapping Algorithmus von Schrefl und Neumayr (siehe Punkt 2.5) vollständig umzusetzen, musste ich mich neben dieser API auch der OWL API 3 bedienen.

Vor allem im Zusammenhang mit der Realisierung der Integrity Constraints (siehe Punkt 2.3) traten große Probleme auf.

Daher möchte ich Ihnen in der folgenden Tabelle kurz zeigen, welche Unterschiede die Protégé OWL API und die OWL API 3 (unterstützt OWL 2) aufweisen. Da der Unterschied dieser beiden OWL APIs doch sehr erheblich ist, sind in Tabelle 3 nur jene Klassen und Methoden angeführt, die für mein Programm relevant sind.

Protégé OWL API	OWL API 3
OWLDatatypeProperty	OWLDataProperty
OWLUnionClass	OWLObjectUnionOf
OWLIntersectionClass	OWLObjectIntersectionOf
OWLEnumeratedClass	OWLObjectOneOf
OWLMaxCardinality	OWL[Object,Data]MaxCardinality
OWLMinCardinality	OWL[Object,Data]MinCardinality
OWLCardinality	OWL[Object,Data]ExactCardinality
OWLAllValuesFrom	OWL[Object,Data]AllValuesFrom
OWLSomeValuesFrom	OWL[Object,Data]SomeValuesFrom
createOWLDatatypeProperty()	getOWLDataProperty()
createOWLObjectProperty()	getOWLObjectProperty()
createOWLNamedSubclass()	getOWLSubClassOfAxiom()
createOWLNamedIndividual()	getOWLNamedIndividual()
createOWLMaxCardinality()	getOWL[Object,Data]MaxCardinality
createOWLMinCardinality()	getOWL[Object,Data]MinCardinality
createOWLCardinality()	getOWL[Object,Data]ExactCardinality
addPropertyValue()	getOWL[Object,Data]PropertyAssertionAxiom
createOWLSomeValuesFrom()	getOWL[Object,Data]AllValuesFrom()

createOWLAllValuesFrom()	getOWL[Object,Data]AllValuesFrom()
createOWLIntersectionClass()	getOWLObjectIntersectionOf()
createOWLUnionClass()	getOWLObjectUnionOf()
createOWLEnumeratedClass()	getOWLObjectOneOf()
addSuperproperty()	getOWLSub[Object,Data]PropertyOfAxiom()
addDisjointClass()	getOWLDisjointClassesAxiom()
addDifferentFrom()	getOWLDifferentIndividualsAxiom()

Tabelle 3: Unterschiede zwischen Protégé OWL API und OWL API 3

2.7 Umsetzung von M-Objects und M-Relationships in Protégé Frames

Diwold hat im Rahmen seiner laufenden Diplomarbeit [Diwo11] bereits eine Umsetzung des Beispiels aus Abbildung 4 in Protégé Frames vorgenommen, was gleichzeitig den Einstiegspunkt für meine Diplomarbeit darstellt.

Um Ihnen dabei zu helfen zu verstehen, auf welche Daten ich mich in Kapitel 3 beziehe, werde ich in den folgenden Unterpunkten anhand einiger Screenshots zeigen, welche Klassen beziehungsweise Klassenhierarchie, Slots, Formulare / Masken und Instanzen er dabei verwendet hat.

Ich weise jedoch darauf hin, dass ich auf einer Beta-Version seiner Umsetzung aufgebaut habe und sich diese daher noch ändern kann.

2.7.1 Klassen und Klassenhierarchie

In Abbildung 7 ist die von Alois Diwold verwendete Klassenhierarchie zu sehen. Von zentraler Bedeutung sind hierbei die Klassen MConnenctionlevel, MLevel, MObject und MRelationship.

Wie leicht zu erkennen ist, besitzt die Klasse MObject eine Vielzahl von Unterklassen. Der Grund dafür liegt darin, dass für jede Ebene, auf welchem ein bestimmtes M-Object beschrieben wird, eine Klasse erzeugt wurde.

Sehen sie sich beispielsweise das M-Object *Product* an. *Product* wird - wie bereits in Abbildung 4 zu sehen war - auf den Abstraktionsebenen *catalog*, *category*, *model* und *physical entity* beschrieben. Daraus ergeben sich die Klassen *ProductCatalog*, *ProductCategory*, *ProductModel* und *ProductPhysicalEntity*. Wie ebenfalls bereits bekannt, wird das M-Object *Product* von *Car* und *Book* konkretisiert.

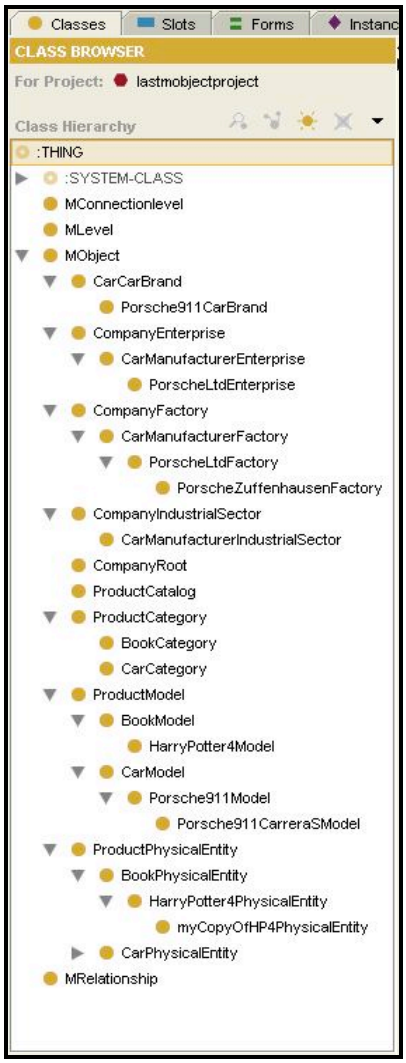

Abbildung 7: Klassenhierarchie (Beispiel aus [Diwo11])

Dadurch, dass alle Levels mit Ausnahme des Top-Levels an Konkretisierungen eines M-Objects vererbt werden, ergeben sich die Klassen *BookCategory* und *CarCategory*, *BookModel* und *CarModel* und *CarPhysicalEntity* und *BookPhysicalEntity*. Diese Klassen bilden dann die Subklassen der entsprechenden Klassen des konkretisierten M-Objects, siehe beispielsweise die beiden Klassen *BookModel* und *CarModel* als Subklassen von *ProductModel*. Dies wird so lange weitergeführt, bis die unterste Ebene erreicht wurde, wodurch sich eine übersichtliche Klassenhierarchie ergibt.

Zu jeder Klasse werden im Klasseneditor eine Reihe von Informationen angezeigt. Abbildung 8 veranschaulicht den oberen Teil des Klasseneditors für die Klasse *CarModel*. Im linken oberen Bereich wird der Name der Klasse angegeben und ob diese eine konkrete oder abstrakte Rolle einnimmt (in diesem Fall eine konkrete). Darunter werden im Bereich der Template Slots jene Slots angezeigt, welche von der betreffenden Klasse bereitgestellt werden. In Abbildung 8 sind das *concretizationOf*, *definesClass*, *listprice* und *maxSpeed*, wobei bei den ersten 3 Slots das blaue Viereck in Klammern steht. Das bedeutet, dass diese Slots von einer übergeordneten Klasse geerbt wurden, *maxSpeed* hingegen nicht.

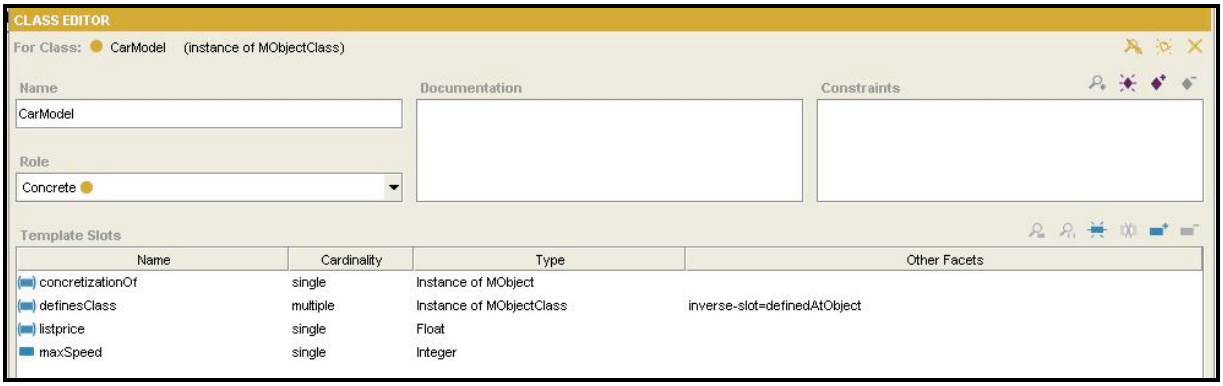

Abbildung 8: Klasseneditor von CarModel (Beispiel aus [Diwo11])

Am unteren Ende des Klasseneditors wird für jede Klasse angezeigt, welchen Typ sie konkretisiert, bei welchem Objekt sie definiert wird und welchen Level sie definiert. In Abbildung 9 wird das für die Klasse *BookPhysicalEntity* gezeigt. Diese Klasse konkretisiert die Klasse *BookModel*, wird beim M-Object *Book* definiert und definiert den Level *PhysicalEntity*.

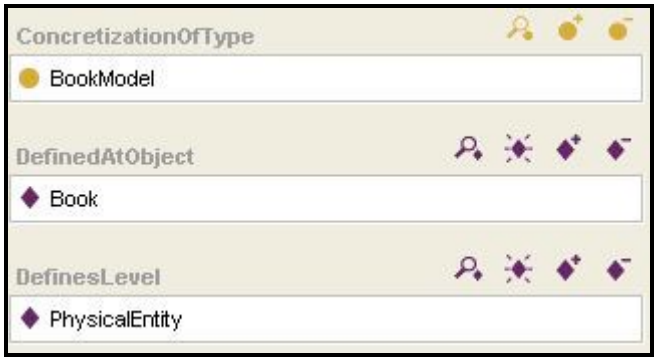

Abbildung 9: Klasseneditor von BookPhysicalEntity (Beispiel aus [Diwo11])

2.7.2 Slots

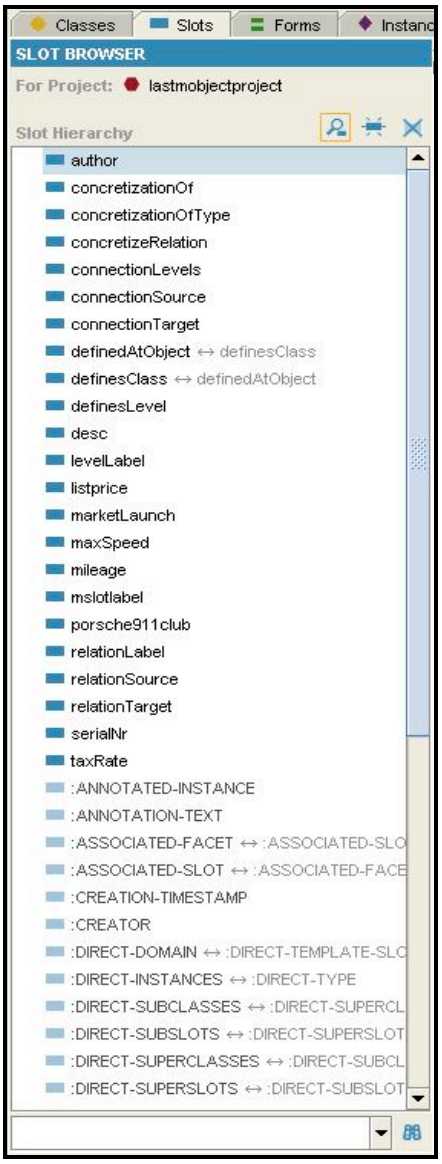

Abbildung 10: Slots (Beispiel aus [Diwo11])

Einige der von Diwold erstellten Slots sind in Abbildung 10 zu sehen. Diese enthalten sowohl Daten- als auch Objekteigenschaften. Dateneigenschaften sind beispielsweise *author*, *taxRate* und *serialNr*. Objekteigenschaften hingegen sind zum Beispiel *concretizationOf*, *definedAtObject* und *definesClass*.

Wie zu jeder Klasse ein entsprechender Klasseneditor existiert, gibt es auch für Slots einen solchen Editor. In Abbildung 11 beziehungsweise Abbildung 12 ist zu sehen, dass ein Sloteditor jede Menge Felder zur Verfügung stellt.

Diese Felder sind hier *Name*, *Value Type*, *Minimum*, *Maximum*, *Mslotlabel*, *Documentation*, *Cardinality* (*required*, *multiple*, *at least*, *at most*), *Inverse Slot*, *Template Values*, *Default Values* und *Domain*. Darüber hinaus wird bei Objekteigenschaften das Feld *Allowed Classes* angezeigt, welches darüber Auskunft gibt, in welchen Klassen diese Eigenschaft vorkommen darf.

In Abbildung 11 ist die Dateneigenschaft *author* mit dem Datentyp String belegt. Das Label, welches angibt, wie diese Eigenschaft in Protégé angezeigt wird, lautet ebenfalls *author*. Darüber hinaus wurde festgelegt, dass die Eigenschaft *author* maximal einmal (im Feld *at most* mit 1 festgelegt) vorkommen kann und dass die Klasse *BookModel* die Domain dieser Eigenschaft darstellt. Die Objekteigenschaft *concretizationOf* in Abbildung 12 verhält sich analog zur Dateneigenschaft *author*, jedoch mit dem Unterschied, dass zusätzlich das Feld *Allowed Classes* angezeigt wird beziehungsweise dass im Feld *Value Type* kein Datentyp, sondern ein Objekt (in diesem Fall eine Instanz) eingetragen ist.

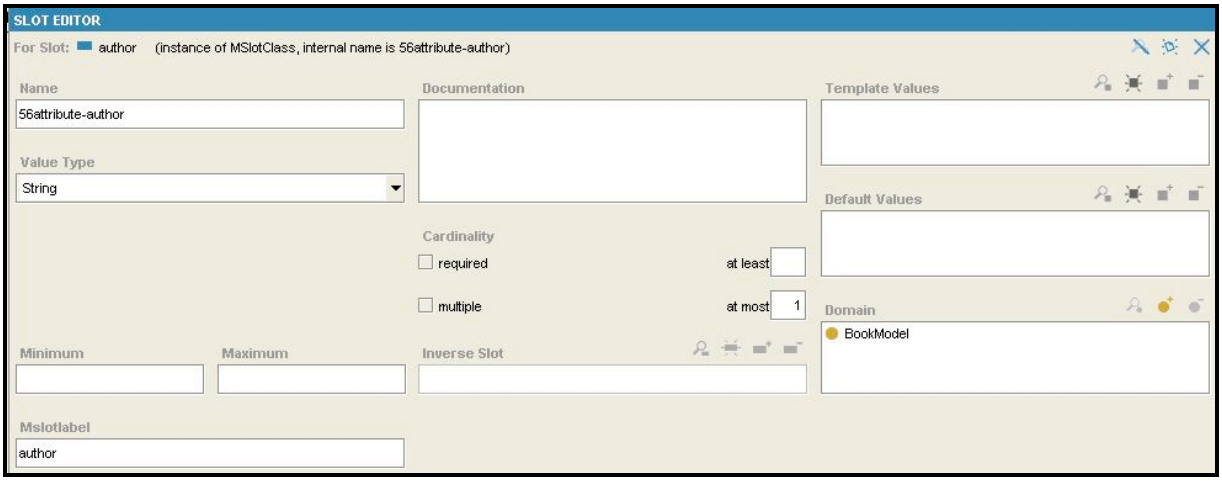

Abbildung 11: Sloteditor von author (Beispiel aus [Diwo11])

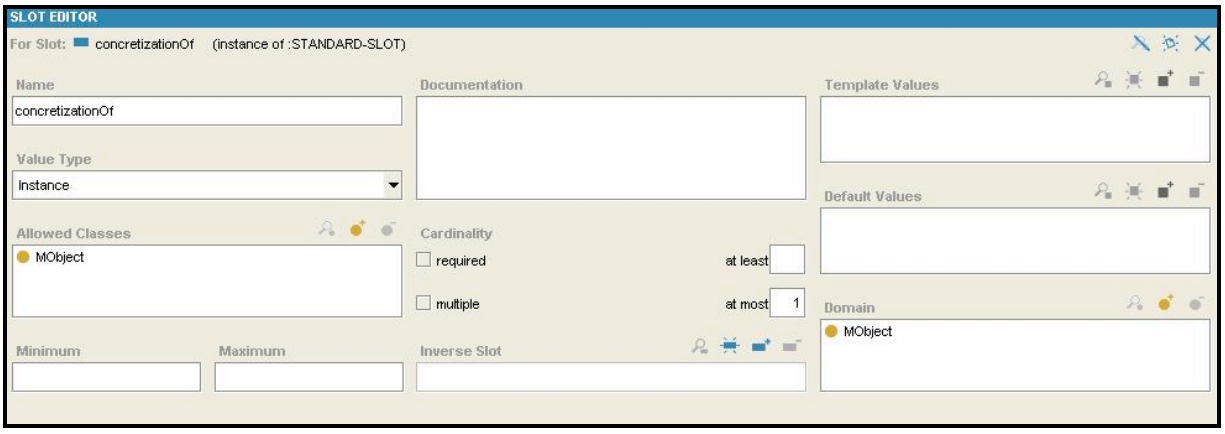

Abbildung 12: Sloteditor von concretizationOf (Beispiel aus [Diwo11])

2.7.3 Formulare / Masken

Wie bereits in 2.6.1 erklärt wurde, legen Formulare / Masken (Forms) fest, wie die Slots einer Klasse angezeigt werden. In Abbildung 13 wird beispielhaft für die Klasse *BookCategory* gezeigt, wie sich die einzelnen Slots dieser Klasse darstellen. Hier wird der Slot *TaxRate* durch ein Integerfeld, *ConcretizationOf* durch ein Instanzenfeld und *DefinesClass* durch ein Instanzenlistenfeld dargestellt. *DefinesClass* wird aus dem Grund durch ein Listenfeld dargestellt, da ein Objekt durchaus mehrere Klassen definieren kann.

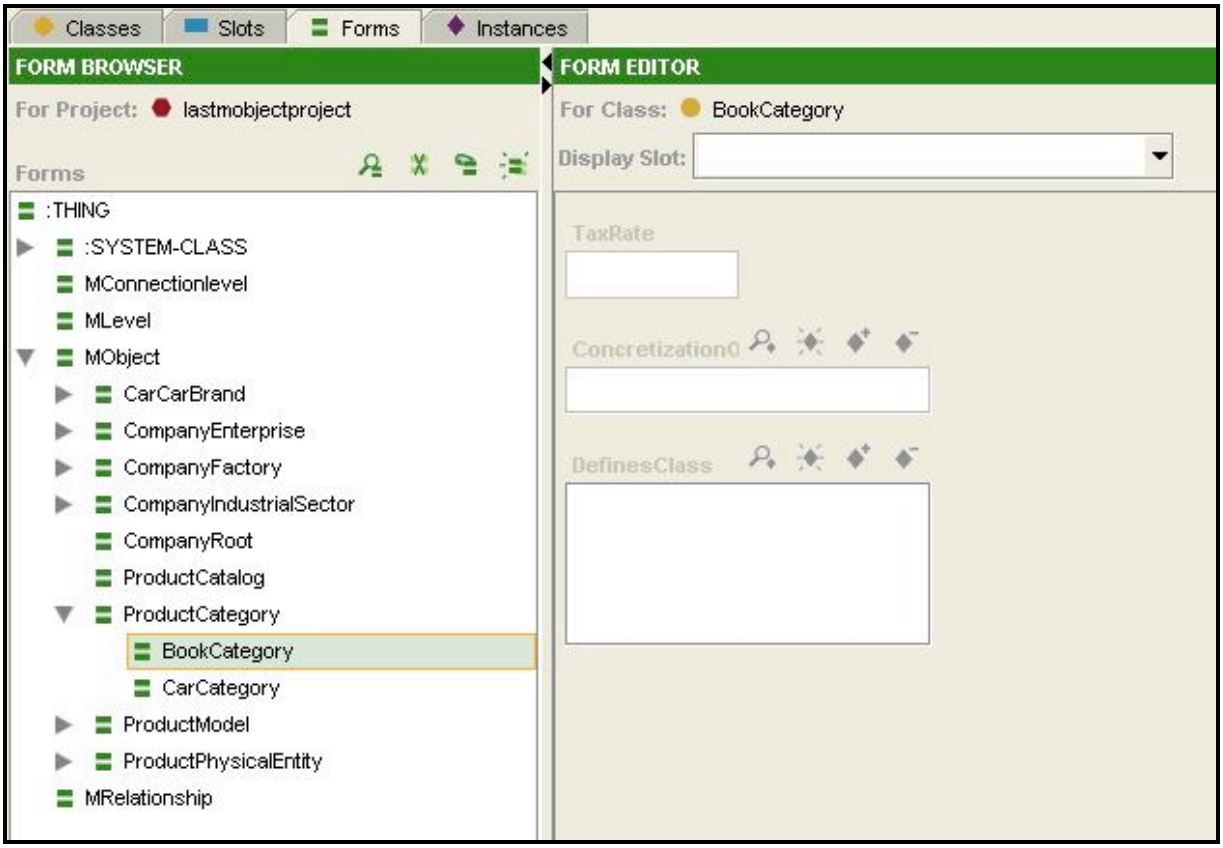

Abbildung 13: Formular / Maske von BookCategory (Beispiel aus [Diwo11])

2.7.4 Instanzen

Abbildung 14 zeigt alle direkten und indirekten Instanzen der Klasse *MObject*. Neben jeder Instanz steht in Klammer jene Klasse, welche dieses M-Object direkt instanziiert. In Abbildung 15 werden die einzelnen Slots zum M-Object *Book* angezeigt. Nun wird auch ersichtlich, wofür die Formulare in 2.7.3 festgelegt wurden. *TaxRate* ist hier mit dem Wert 15 belegt, *Book* stellt eine Konkretisierung von *Product* (im Slot *ConcretizationOf* angegeben) dar und definiert die Klassen *BookCategory*, *BookModel* und *BookPhysicalEntity* (im Slot *DefinesClass* festgehalten).

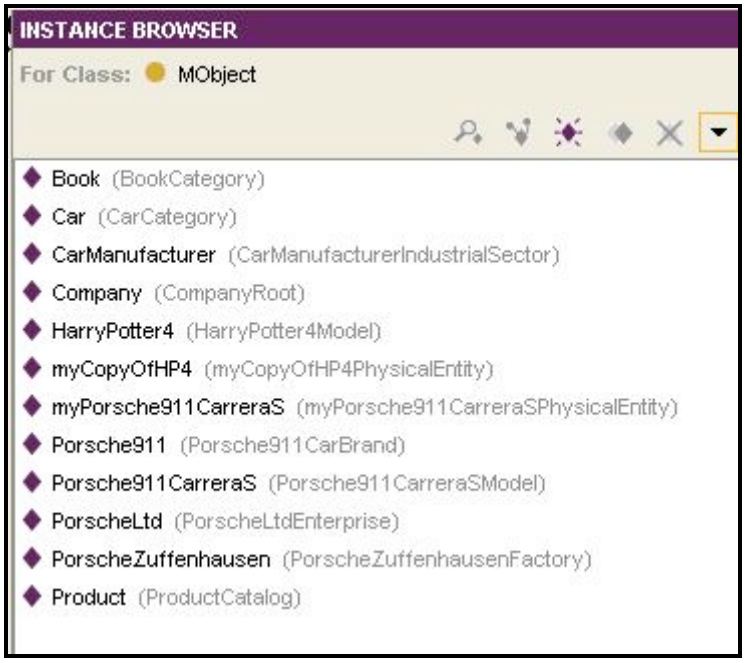

Abbildung 14: Instanzenbrowser (Beispiel aus [Diwo11])

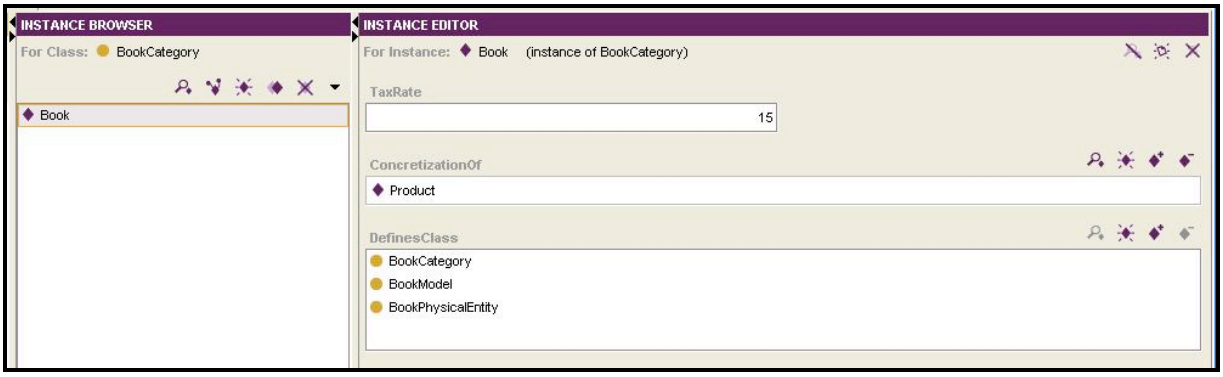

Abbildung 15: Instanzeneditor von Book (Beispiel aus [Diwo11])

3 Realisierung Mapping Algorithmus

3.1 Vorbereitende Tätigkeiten

Bevor ich mit der eigentlichen Umsetzung des Mapping Algorithmus aus [Neum09b] beginnen konnte, waren noch eine Reihe vorbereitender Tätigkeiten notwendig, beschrieben in den folgenden Punkten 3.1.1 - 3.1.3. Es war unter anderem erforderlich einige Methoden zu schreiben, die zwar mit dem Mapping Algorithmus selbst nichts zu tun hatten, aber ohne die das Mapping erst gar nicht möglich gewesen wäre. Diese werden nun unter anderem kurz beschrieben.

3.1.1 Erstellung des Export-Plugins

Bevor ich mit der Realisierung des Mapping Algorithmus beginnen konnte, musste ich es dem Benutzer zunächst ermöglichen mein Plugin, dass diesen Algorithmus enthält, in Protégé aufrufen zu können. Ein Plugin stellt nicht mehr als eine Erweiterung eines bereits existierenden Programms dar. Protégé ermöglicht es Entwicklern sechs verschiedene Arten von Plugins zu schreiben: [Prot10b]

- Tab widget
- Slot widget
- Back-end
- Create project
- Export
- Project

Da es sich bei dem von mir zu erstellenden Plugin um ein Export-Plugin handelt, werde ich hier nur darauf eingehen, wie die Vorgehensweise bei der Erstellung eines derartigen Plugins aussieht. Sollten Sie daran interessiert sein, wie die Erstellung der anderen angeführten Plugins funktioniert, verweise ich auf [Prot10b], wo Sie sehr

gute Beispiele zu deren Erstellung finden. Bei der Erstellung des Plugins im Rahmen dieser Arbeit bin ich etwas anders vorgegangen als in [Prot10c] vorgeschlagen wird, nämlich folgendermaßen:

1. Im ersten Schritt habe ich den Source Code von Protégé Frames als auch von Protégé OWL unter http://protege.stanford.edu/download/registered.html runtergeladen.

2. Um anschließend diesen Source Code nach meinen Wünschen abändern und erweitern zu können, wurden im nächsten Schritt in der Entwicklungsumgebung Eclipse zwei Projekte mit den Namen *Core* und *OWL* erstellt und der jeweilige Code in diese beiden Projekte importiert. Damit Eclipse keine Fehlermeldungen zurückgibt, musste angegeben werden, dass *Core* von *OWL* verwendet wird. Sie können hier davon ausgehen, dass ich Eclipse gemäß der beispielhaften Vorgehensweise in [Prot10c] zuvor konfiguriert habe. Um die bereits erwähnte OWL API 3 verwenden zu können, mussten in den *Build-Path* von *OWL* die entsprechenden *jar-files* der OWL API 3 aufgenommen werden.

3. Da der heruntergeladene Source Code von Protégé bereits Klassen zur Erstellung eines Export-Plugins nach OWL zur Verfügung stellt, konnte ich mir die Arbeit ersparen, die für dieses Plugin relevanten Klassen alle neu zu schreiben. Daher wurden zunächst die entsprechenden Klassen einfach dupliziert und neu benannt. Das bereits vorhandene Plugin bot jedoch bei weitem nicht die Funktionen die mein Plugin im späteren Verlauf zur Verfügung stellen soll. Daher war es notwendig, die einzelnen Klassen abzuändern beziehungsweise zu erweitern. Konkret wurden in diesem Schritt dabei folgende Klassen dupliziert, neu benannt und nach meinen Wünschen abgeändert:

   - **JenaExportPlugin → JenaMOWLExportPlugin**
     Damit das Plugin in Protégé nicht mit “OWL” angezeigt wird, musste in der Methode *getName()* der Rückgabewert von “OWL” auf “M-OWL” abgeändert werden. Damit die exportierte Datei auch im gewünschten Pfad gespeichert wird, musste darüber hinaus der *run()*-Methode der Klasse

MOWLKnowledgeBaseCopier der Dateipfad als Parameter mit übergeben werden. Ansonsten wurden in dieser Klasse keine weiteren Änderungen vorgenommen.

- **ProtegeSaver → MOWLProtegeSaver**
  Diese Klasse wird von der Klasse *JenaMOWLExportPlugin* aufgerufen. Da ich bis auf den Konstruktor dieser Klasse nichts weiter benötigt habe, wurden alle darin enthaltenen Methoden entfernt.

- **KnowledgeBaseCopier → MOWLKnowledgeBaseCopier**
  Zunächst wurden in dieser Klasse die benötigten Pakete der OWL API 3 importiert und alle nicht benötigten Methoden entfernt. An ihre Stelle treten später die für die Realisierung des Mapping Algorithmus benötigten Methoden. Bestehen blieben in dieser Klasse lediglich der Konstruktor und die *run()*-Methode.

4. Um mein Plugin in Protégé anzeigen zu können, musste im vierten Schritt noch eine kleine Erweiterung vorgenommen werden. Hierzu habe ich in Protégé OWL die im Ordner META-INF enthaltene Datei MANIFEST.MF um folgende Zeilen erweitert:

   Name: edu/stanford/smi/protegex/owl/jena/export/JenaMOWLExportPlugin.class
   Export-Plugin: True

5. Nach der Angabe von edu.stanford.smi.protegex.owl.ProtegeOWL als Mainklasse konnte Protégé nun von Eclipse aus gestartet werden. Die folgende Abbildung zeigt, dass bei „Export to format“ nun „M-OWL“ als neuer Menüeintrag vorhanden ist.

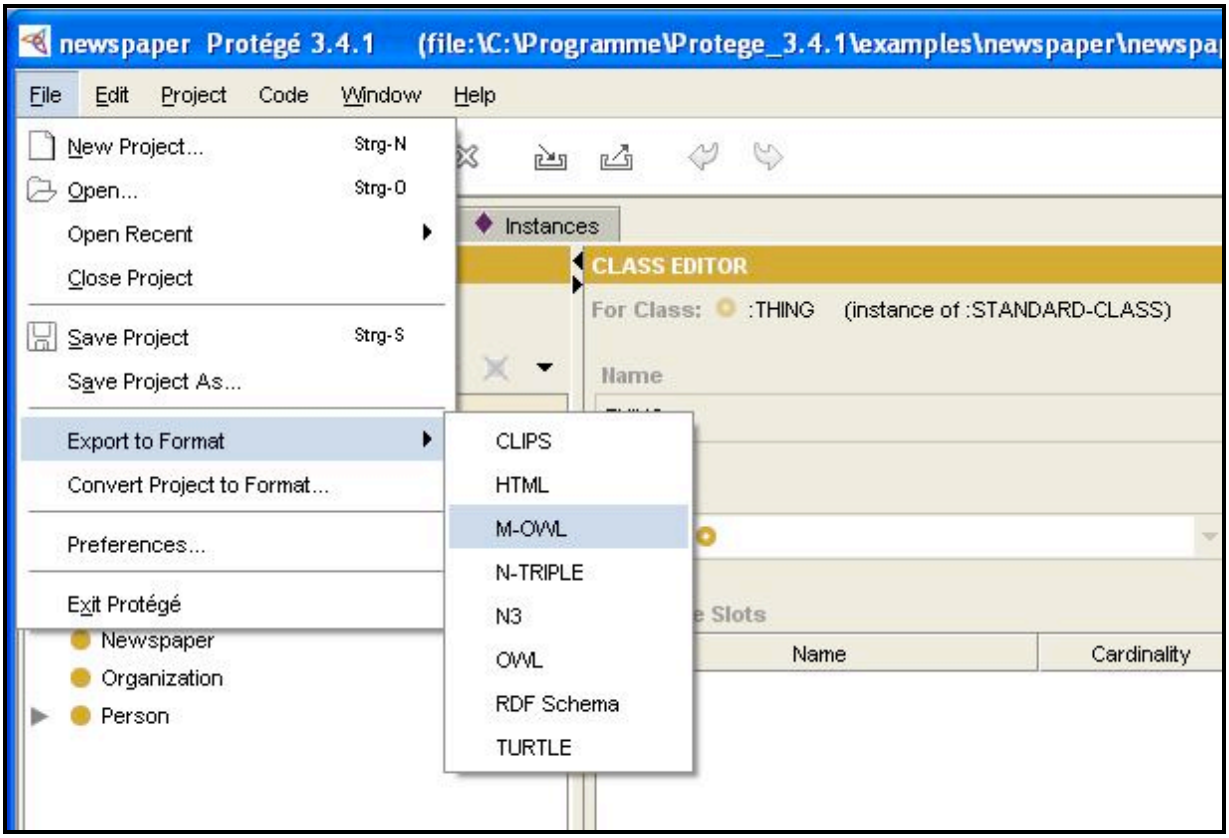

Abbildung 16: Menü nach eigener Plugin-Erstellung

Wird nun der Menüeintrag „M-OWL“ ausgewählt, erscheint ein Fenster in dem Dateiname und Speicherort gewählt werden können. Durch einen anschließenden Klick auf „OK“ wird das Plugin ausgeführt. Ich möchte hier darauf hinweisen, dass dieses Plugin bis zu diesem Zeitpunkt lediglich eine leere OWL-Datei erstellt.

3.1.2 Erstellung der Ontologie und des Dokuments

Wie bereits in den Grundlagen erwähnt, stellt den zentralen Bestandteil bzw. den Grundbaustein von OWL immer eine Ontologie dar, in welche die Klassen, Individuen, Axiome, Eigenschaften, usw. aufgenommen werden. Daher wird zunächst eine solche Ontologie benötigt. Beim Start von Protégé OWL wird unter anderem die Möglichkeit geboten eine bereits existierende OWL Ontologie zu öffnen. Eine weitere Option stellt die Erzeugung einer neuen OWL Ontologie dar. Da beim Export nach OWL erst eine neue Ontologie erzeugt werden musste, war in meinem Fall die zweite Möglichkeit von Interesse. Erfolgen keine genaueren Angaben, wird diese Ontologie von Protégé OWL Default mäßig erstellt und benannt.

Um die erstellte Ontologie in weiterer Folge in einer OWL Datei speichern zu können (um sie später in Protégé OWL öffnen zu können), musste im nächsten Schritt ein OWL Dokument erstellt werden, welches die Ontologie einschließlich der gemappten Klassen, Instanzen, Eigenschaften, usw. beinhaltet. Bei der Erstellung der OWL Ontologie und des Dokuments wurden zunächst die Wurzelklasse (wird im Rahmen der Default mäßigen Erstellung der Ontologie ebenfalls Default mäßig erstellt) und das dazugehörige OWL Modell ermittelt. Durch den Import des Pakets *org.semanticweb.owlapi.model* wird unter anderem der Zugriff auf die Klasse IRI ermöglicht, welche die Methode *create(String str)* beziehungsweise *create(File file)* zur Verfügung stellt.

Diese Methoden ermöglichen das Erzeugen einer IRI (Internationalized Resource Identifier, dienen der eindeutigen Identifikation einer bestimmten Ressource) für die Ontologie (hier wurde der Default mäßige Name der Ontologie des zuvor ermittelten OWL Modells verwendet) und einer IRI für das Dokument (hier wurde der als Parameter übergebene Pfad verwendet). Durch den Import der Klassen *org.semanticweb.owlapi.-apibinding.OWLManager* und *org.semanticweb.owlapi.util.-SimpleIRIMapper* konnten ein *OWLOntologyManager* und ein *IRIMapper* erzeugt werden. Während die Aufgabe des *OWLOntologyManagers* unter anderem darin besteht, Axiome einer Ontologie hinzuzufügen, besteht die Aufgabe des *IRIMappers* in der Überführung der Ontologie in das OWL Dokument.

Dieser *IRIMapper* wird vom *OWLOntologyManager* (von mir als *manager* bezeichnet und in weiterer Folge nur mehr unter diesem Namen verwendet) mittels der Methode *addIRIMapper(IRI mapperIRI)* der Ontologie hinzugefügt. Daran anschließend wurde mittels der vom *manager* bereitgestellten Methode *createOntology(IRI ontologyIRI)* die Ontologie erzeugt und mittels der Methode *saveOntology(IRI ontologyIRI)* die Ontologie gespeichert und dem Dokument hinzugefügt. Somit war die Voraussetzung für die Erstellung der Klassen, Individuen, Eigenschaften, usw. geschaffen. In Abbildung 17 sehen Sie die neu erstellte OWL Ontologie mit der Default mäßig erstellten Wurzelklasse *Thing*.

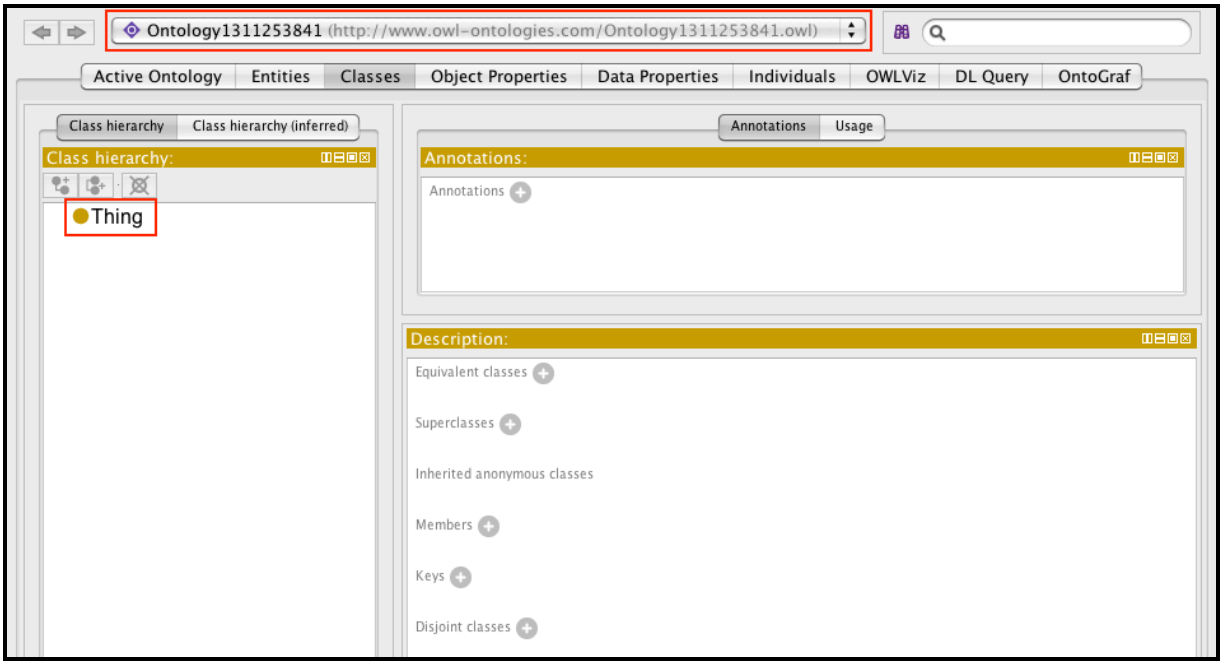

Abbildung 17: Neue OWL-Ontologie mit Wurzelklasse Thing

3.1.3 Erstellung der OWL Annotation Property

Wie bereits häufig in dieser Arbeit erwähnt wurde, wird um ICs erweitertes OWL im Moment von Protégé noch nicht unterstützt. Protégé OWL bietet jedoch die Möglichkeit mittels so genannter OWL Annotation Properties Axiome mit einer Anmerkung zu versehen. Diese Annotationen stellen zwar keine Lösung des Problems (Behandlung von ICs als *Closed World* und nicht als *Open World* Axiome) dar, geben jedoch einen Anknüpfungspunkt für spätere Erweiterungen. Aus diesem Grund werden die in den Punkten 3.2.5, 3.2.7 - 3.2.9, 3.3.4 und 3.3.5 erstellten Integrity Constraints von mir mittels OWL Annotation Property als ICs gekennzeichnet.

Da folgende Annotation Property im Laufe des Mappings sehr häufig zum Einsatz kommt, wurde sie gleich zu Beginn erstellt und dem Set *annotations* hinzugefügt, damit im späteren Verlauf bei der Realisierung des Mapping Algorithmus jederzeit bei Bedarf darauf zugegriffen werden konnte und nicht laufend neu erzeugt werden musste. Die Annotation Property wurde durch

factory.getOWLAnnotation(factory.getOWLAnnotationProperty (OWLRDFVocabulary.RDF_DESCRIPTION.getIRI()), factory.getOWLLiteral(“Integrity Constraint“))

erzeugt. Wie unschwer zu erkennen ist, entschied ich mich dabei für eine beschreibende Annotation Property (Description, siehe Abbildung 18). Die Annotation Property hätte aber durchaus auch in anderer Form (Kommentar, Titel, Identifier, Label, usw.) den Axiomen hinzugefügt werden können.

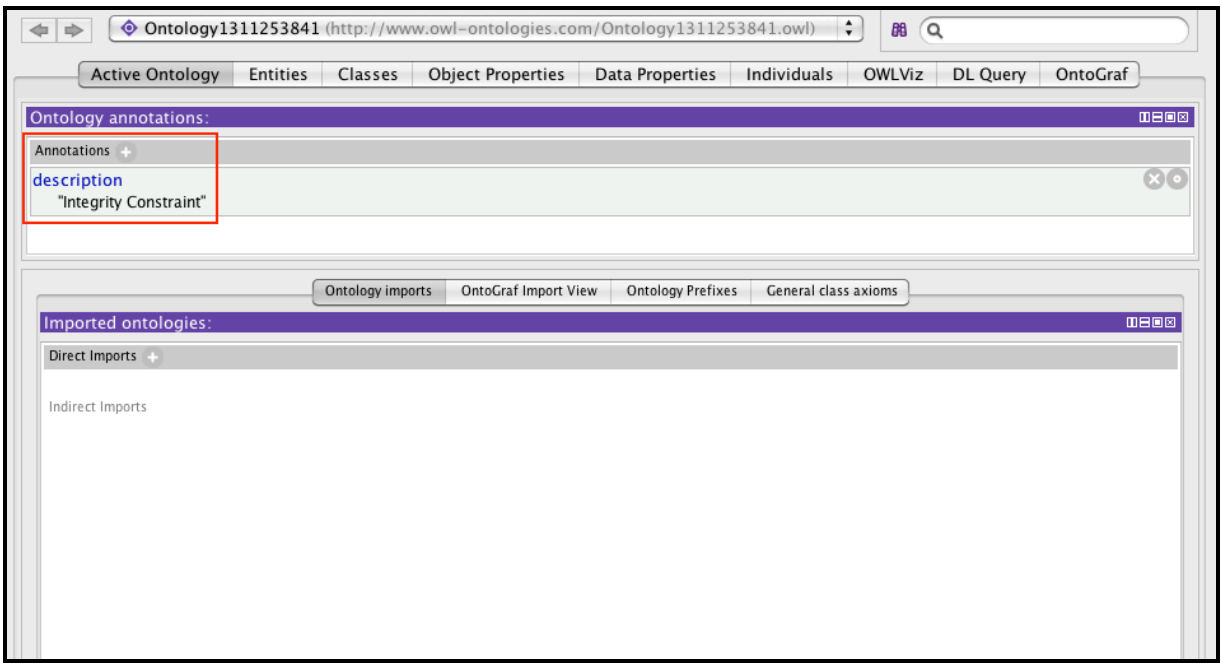

Abbildung 18: OWL Annotation Property für Integrity Constraints

3.2 Mapping der M-Objects nach OWL

Nun konnte mit der konkreten Umsetzung des Mapping Algorithmus (siehe Punkt 2.5.1 und 2.5.2) aus [Neum09b] begonnen werden. In diesem Punkt wird zunächst erläutert, wie das Mapping der M-Objects nach OWL realisiert wurde und erst anschließend in Punkt 3.3 das Mapping der M-Relationships nach OWL gezeigt. Die folgenden Unterpunkte beziehen sich dabei auf die Zeilen 1 – 18 des Mapping Algorithmus, wobei immer explizit angeführt wird, welche Zeile(n) im aktuellen Punkt umgesetzt werden. Es wurde dabei von mir eine getrennte Abarbeitung gewählt, sprich,

es wurde jede Zeile durch eine eigene Methode umgesetzt. Der Grund dafür liegt darin, dass ich so meiner Meinung nach eine übersichtlichere Struktur bei der Programmierung schaffen konnte.

3.2.1 Festlegen der Eigenschaften der Konkretisierungshierarchie

In den Zeilen 1 – 3 des Mapping Algorithmus werden die Eigenschaften der Konkretisierungshierarchie festgelegt. Konkret handelt es sich dabei um folgende drei Zeilen:

1: **assert:** $\top \sqsubseteq \leq 1$ concretize
2: **assert:** concretize $\sqsubseteq$ concretize_t
3: **assert:** concretize_t^{+} $\sqsubseteq$ concretize_t

In der ersten Zeile wird festgelegt, dass jedes M-Object maximal ein anderes M-Object konkretisiert. In den Zeilen 2 – 3 wird weiters festgelegt, dass die Objekteigenschaft *concretize* eine Subeigenschaft von *concretize_t* darstellt und *concretize_t* wiederum eine Supereigenschaft der transitiven Hülle von *concretize_t*. Um mit diesen Eigenschaften im Rahmen des Mappings arbeiten zu können, mussten diese der Ontologie zunächst hinzugefügt werden.

Wie bereits bekannt ist, unterscheidet OWL zwischen Objekt- und Dateneigenschaften. Bei *concretize* und *concretize_t* handelt es sich um Objekteigenschaften. Wie ebenfalls bereits bekannt ist, besteht die Aufgabe solcher OWL Objekteigenschaften darin Objekte, welche in einer OWL Ontologie enthalten sind, miteinander zu verlinken. Um diese erzeugen zu können musste zuerst mittels des *managers* eine sogenannte *OWLDataFactory* (von mir kurz als *factory* bezeichnet) erstellt werden. Diese *factory* stellt eine Vielzahl von Methoden zur Verfügung um unter anderem Klassen, Axiome oder Entitäten erzeugen zu können. Durch

```
factory.getOWLObjectProperty(IRI.create(ontologyIRI + "#concretize"))
```

wird beispielsweise die Objekteigenschaft *concretize* erzeugt. Gleichermaßen wurde bei der Erstellung von *concretize_t* verfahren. In einem nächsten Schritt musste noch angegeben werden, dass es sich bei *concretize* um eine funktionale und bei *concretize_t* um eine transitive Objekteigenschaft handelt (siehe Punkt 2.1.4). Dies wurde für *concretize* durch folgende drei Zeilen erreicht:

```
OWLAxiom axiom = factory.getOWLFunctionalObjectPropertyAxiom(concretize)
AddAxiom addAxiom = new AddAxiom(ontology, axiom)
manager.applyChange(addAxiom)
```

Für *concretize_t* verhält es sich sehr ähnlich, mit dem Unterschied, dass nicht die Methode *getOWLFunctionalObjectPropertyAxiom(OWLObjectProperty concretize)* sondern die Methode *getOWLTransitiveObjectPropertyAxiom(OWLObjectProperty concretize_t)* zum Einsatz kommt.

Da diese drei Zeilen eigentlich am Ende jeder von mir geschriebenen Methode vorkommen werde ich diese in der Folge nicht mehr extra anführen sondern lediglich das erstellte Axiom erwähnen. Die Zeilen zwei und drei bleiben dabei immer gleich.

Mit der Angabe, dass es sich bei *concretize* um eine funktionale Objekteigenschaft und bei *concretize_t* um eine transitive Objekteigenschaft handelt, wurde bereits die erste und dritte Zeile des Mapping Algorithmus umgesetzt. Der folgenden Abbildung können Sie entnehmen, wie *concretize* und *concretize_t* nun bei den OWL Objekteigenschaften in Protégé angezeigt werden.

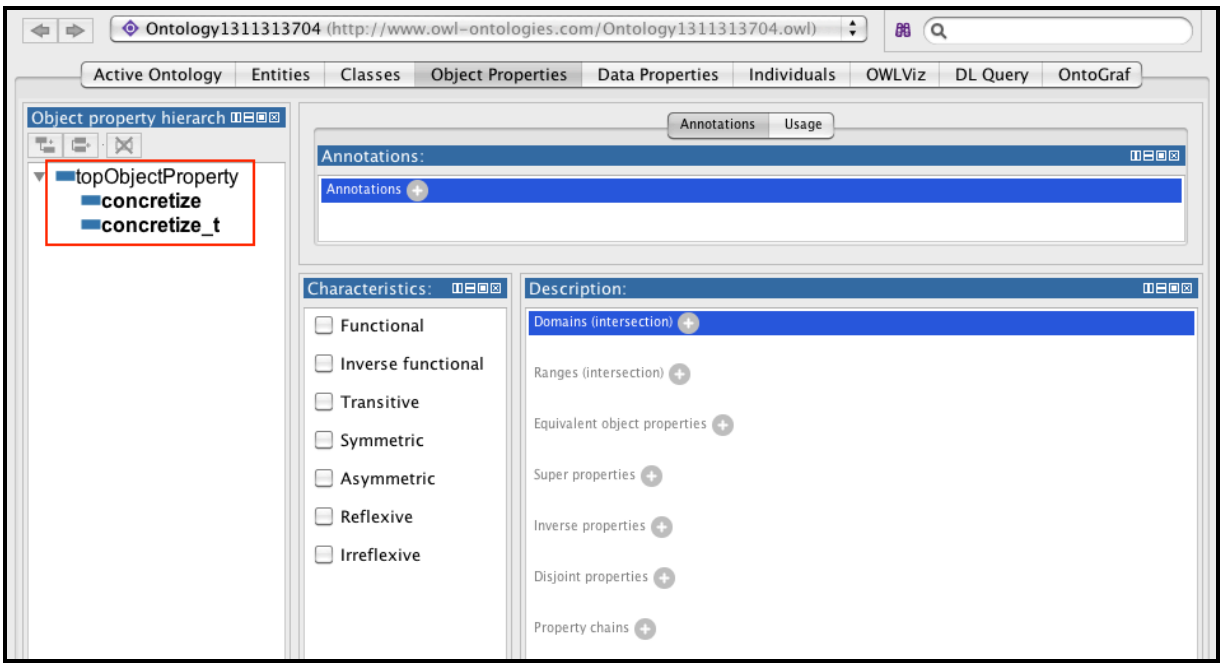

Abbildung 19: OWL Objekteigenschaften concretize und concretize_t

Mit der abermals von der *OWLDataFactory* bereitgestellten Methode *getOWLSubObjectPropertyOfAxiom(OWLObjectProperty concretize, OWLObjectProperty concretize_t)* konnte anschließend festgelegt werden, dass *concretize* eine Subobjekteigenschaft von *concretize_t* darstellt (zweite Zeile des Mapping Algorithmus). Wie in Abbildung 20 zu sehen ist, wird nun in Protégé *concretize* als Subobjekteigenschaft von *concretize_t* angeführt.

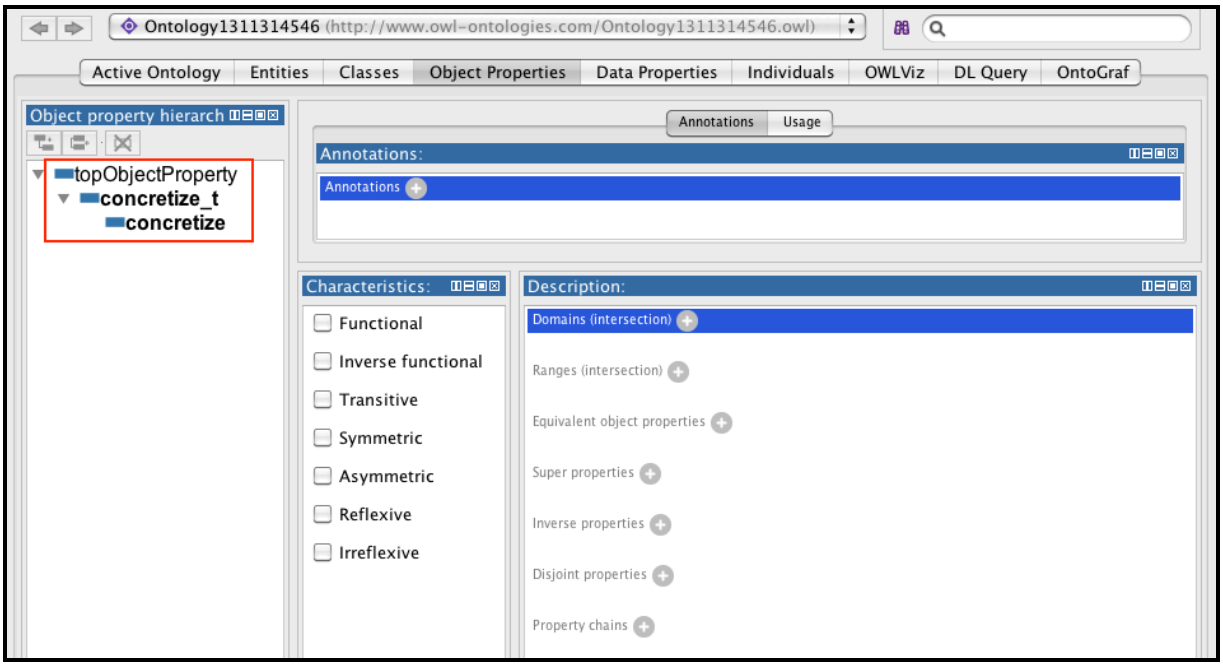

Abbildung 20: concretize als Subobjekteigenschaft von concretize_t

3.2.2 Erstellung der M-Objects und Zuweisung der M-Levels zu M-Objects

Durch die Zeilen 4 und 5 des Mapping Algorithmus wird den M-Objects ihr jeweiliger Top-Level zugewiesen beziehungsweise anders ausgedrückt, jener Top-Level, den es definiert. Beispielsweise wird dem M-Object *Product* der Top-Level *Catalog* oder dem M-Object *Book* der Top-Level *Category* zugewiesen. Dies wird im Mapping Algorithmus durch folgende beiden Zeilen ausgedrückt:

4: **for all** $o \in O$ **do**
5: **assert:** [$\hat{I}_o$] ([o])

M-Levels werden in Protégé OWL als Klassen (*OWLClasses*) dargestellt. Sie dienen dazu um Mengen von Individuen zusammenzufassen bzw. zu klassifizieren. Um mit den M-Objects und M-Levels arbeiten zu können, mussten diese zunächst aus dem realisierten Beispiel aus [Diwo11] ausgelesen werden. Dabei wurden zuerst die M-Levels ermittelt und der Ontologie hinzugefügt und anschließend den ermittelten M-Objects zugewiesen.

Um die M-Levels zu erzeugen, wurde zunächst mittels der von *OWLDataFactory* bereitgestellten Methode *getOWLClass(IRI OWLClassIRI)* die Klasse *MObject* erstellt. Diese neue Klasse wird dabei automatisch zu einer Subklasse der bereits in Punkt 3.1.2 erzeugten Wurzelklasse *Thing*, welche wiederum automatisch beim Erstellen der Ontologie erzeugt wurde. Nachdem diese Klasse erzeugt war, wurde mittels eines Iterators über alle Instanzen des realisierten Beispiels aus [Diwo11] iteriert, wobei darauf zu achten war, dass es sich nicht bei jeder Instanz um einen Level handeln musste. Lediglich bei jenen Instanzen, welche den Slot *levelLabel* enthielten, handelte es sich um M-Levels.

Aus den gefundenen M-Levels wurden anschließend OWL Klassen erzeugt. Nachdem nun alle relevanten M-Levels erzeugt waren, mussten diese noch der Klasse *MObject* als Subklassen zugewiesen werden. Dies konnte mit dem Axiom *OWLSubclassOfAxiom(OWLClass subclass, OWLClass MObject)* realisiert werden. Somit waren alle Abstraktionsebenen der Beispielrealisierung aus [Diwo11] in der Ontologie hinterlegt, zu sehen in der folgenden Abbildung.

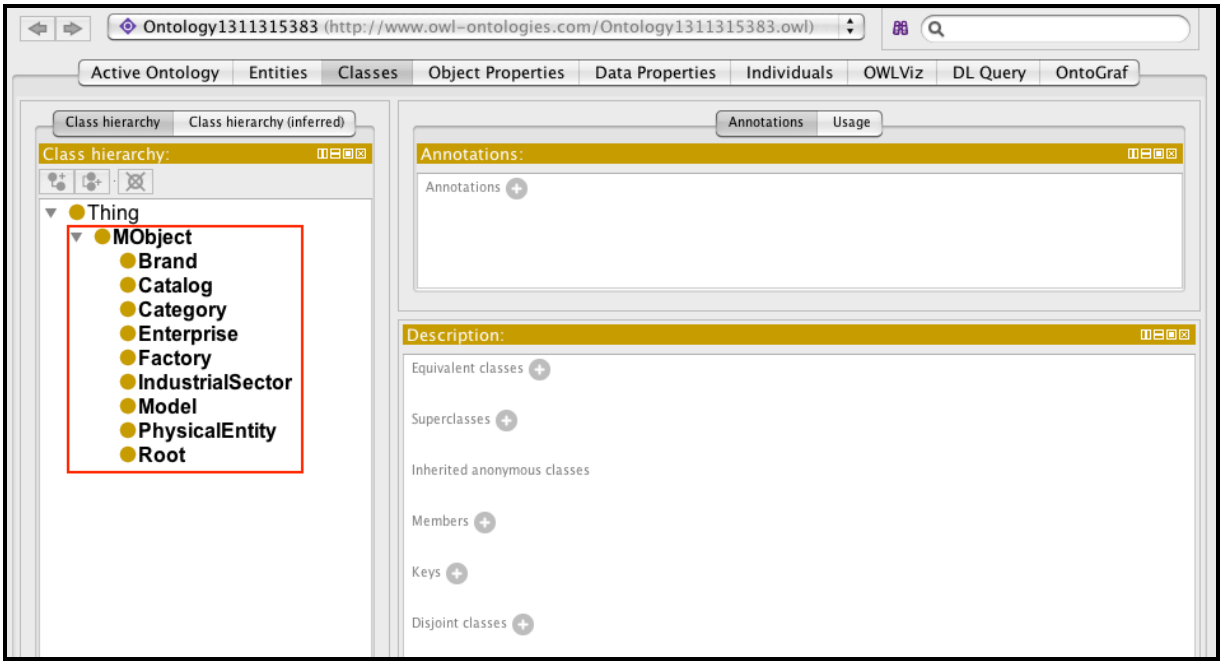

Abbildung 21: M-Levels

Bei der Erstellung der M-Objects wurde sehr ähnlich verfahren. Dabei wurde wiederum über alle Instanzen der Realisierung aus [Diwo11] iteriert, wobei nur jene Instanzen berücksichtigt wurden, welche eine Instanz einer so genannten *Default Simple Instance* darstellten und einen eigenen Slot namens *concretizationOf* hatten, da es sich nur bei diesen Instanzen um die gewünschten M-Objects handelte. Von jeder ermittelten Instanz konnte dann über den direkten Typ dieser Instanz die jeweilige Klasse ermittelt werden. Über den in dieser Klasse enthaltenen Slot *definesLevel* konnte anschließend ermittelt werden, welchen Top-Level dieses M-Object definiert.

Mittels der von *OWLDataFactory* bereitgestellten Methode *getOWLNamedIndividual(IRI OWLNamedIndividualIRI)* konnten anschließend aus den ermittelten Instanzen *OWLNamedIndividuals* erzeugt und schließlich mittels des Axioms *OWLClassAssertionAxiom(OWLClass mlevel, OWLNamedIndividual owlNamedIndividual)* den M-Objects der entsprechende M-Level zugewiesen werden. In den folgenden Abbildungen wird beispielhaft gezeigt, wie den M-Objects *Product*, *Book* und *Car* ihr jeweiliger Top-Level (*Catalog* bei *Product, Category* bei *Book* und *Car*) zugewiesen wurde.

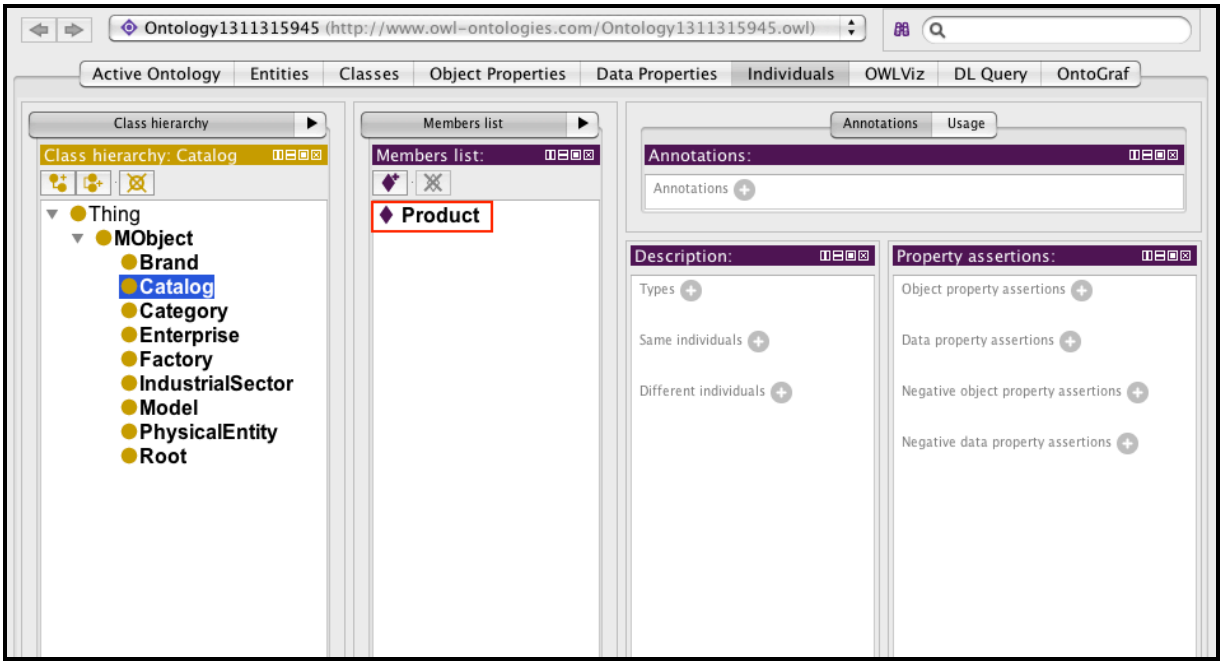

Abbildung 22: Zuweisung des Top-Levels Catalog zu Product

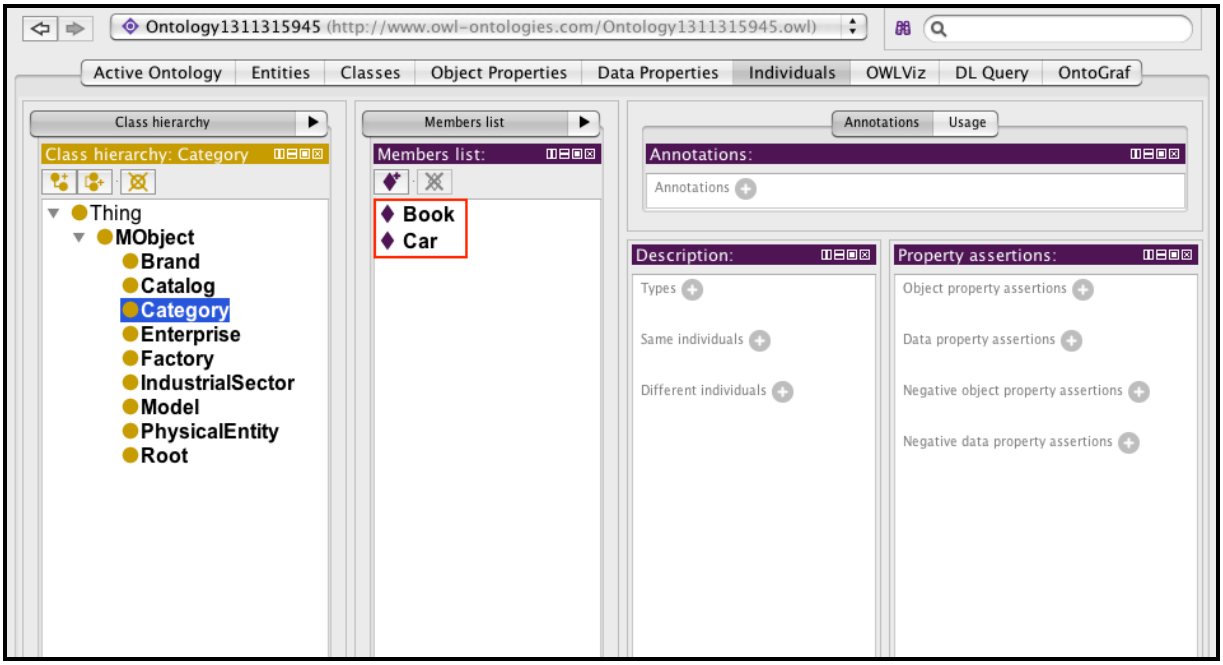

Abbildung 23: Zuweisung des Top-Levels Category zu Book und Car

3.2.3 Konkretisierung der M-Objects

Als nächstes erfolgt in den Zeilen 4 und 6 des Mapping Algorithmus die Konkretisierung der M-Objects, d. h. es wird jedem M-Object jenes M-Object zugewiesen, wel-

ches es konkretisiert. Dabei handelt es sich um folgende beiden Zeilen des Mapping Algorithmus:

4: **for all** $o \in O$ **do**

6: **if** $\exists o'$: $(o,o') \in H$ **then assert:** concretize([*o*],[*o'*])

Um jederzeit einen schnellen Zugriff auf alle erstellten M-Objects zu haben, wurde in diesem Zusammenhang die Hilfsmethode *getMObjects()* geschrieben, welche eine Collection aller erstellten M-Objects zurückliefert. Darüber hinaus liefert die Hilfsmethode *getTopMObjects()* eine Collection jener Objekte, welche keine anderen Objekte konkretisieren, was im Beispiel aus [Neum09b] auf die beiden M-Objects *Product* und *Company* zutrifft.

Somit konnte über alle M-Objects (mit Ausnahme der Top-M-Objects) iteriert und die jeweilig entsprechende Instanz in der Realisierung aus [Diwo11] ermittelt werden. Dort besitzt jede Instanz einen Slot namens *concretizationOf*, in welchem die Instanz festgehalten wird, welche sie konkretisiert. Zu dieser Instanz konnte anschließend das entsprechende *OWLNamedIndividual* im OWL Modell ermittelt werden, womit die beiden für die Konkretisierung benötigten M-Objects bekannt waren.

Da die OWL Objekteigenschaft *concretize* bereits in Punkt 3.2.1 erstellt wurde, musste im OWL Modell dieses bereits bestehende Feld nur mehr bei dem betreffenden M-Object mit jenem M-Object befüllt werden, welches es konkretisiert. Das konnte mittels der von der *OWLDataFactory* bereitgestellten Methode *getOWLObjectPropertyAssertionAxiom(OWLObjectProperty concretize, OWLNamedIndividual individual1, OWLNamedIndividual individual2)* realisiert werden. Die folgende Abbildung zeigt die durchgeführte Konkretisierung am Beispiel von *Car* und *Product*.

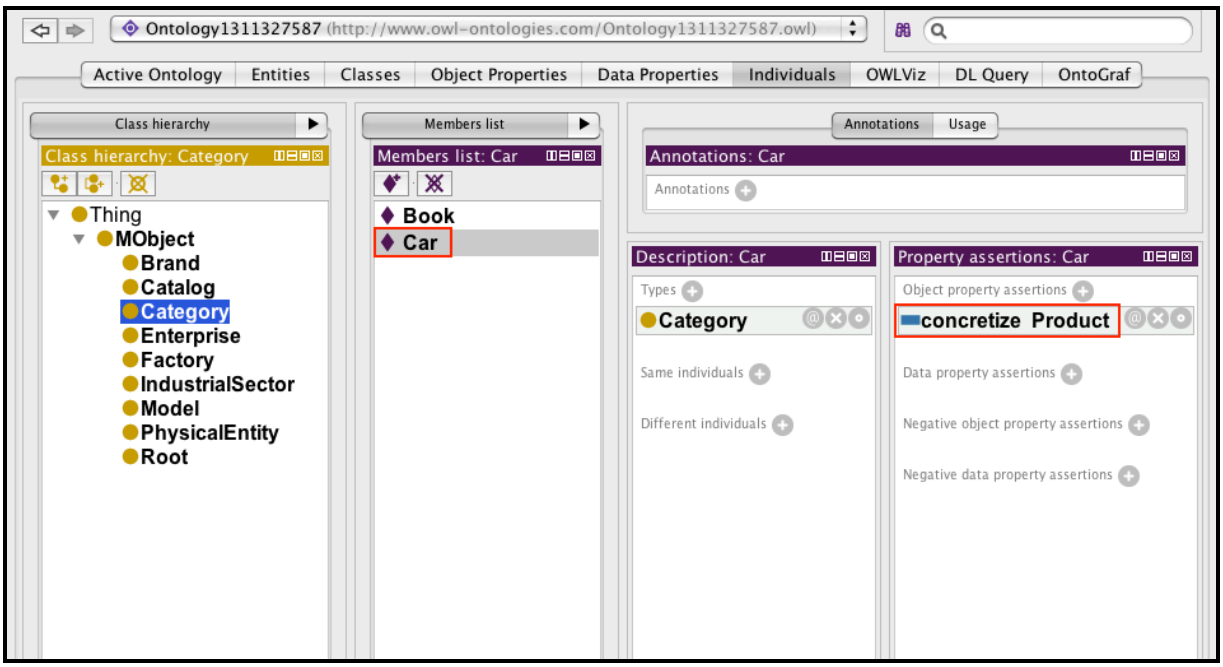

Abbildung 24: Car konkretisiert Product

3.2.4 Eigenschaften und Werte von Eigenschaften

Nachdem die Konkretisierung der M-Objects vorgenommen wurde, erfolgt nun in den Zeilen 4 und 7 des Mapping Algorithmus die Erstellung der Dateneigenschaften (einschließlich der darin enthaltenen Werte) und deren Zuweisung zu den entsprechenden M-Objects, ausgedrückt durch folgende Zeilen:

4: **for all** $o \in O$ **do**

7: **for all** $a \in \hat{A}_o : v_o(a)$ is defined **do assert:** $[a]([o],[v_o(a)])$

Zur Umsetzung dieser Zeilen wurde über alle Slots der Beispielrealisierung aus [Diwo11] iteriert und jene Slots berücksichtigt, welche einen eigenen Slot namens *mSlotLabel* besaßen, da es sich nur bei diesen Slots um die gewünschten Dateneigenschaften handelte. Aus den ermittelten Slots wurde anschließend mit Hilfe der von der *OWLDataFactory* bereitgestellten Methode *getOWLDataProperty(IRI OWLDataPropertyIRI)* eine entsprechende OWL Dateneigenschaft für das OWL Modell erzeugt.

Anschließend wurde wiederum unter Verwendung der Hilfsmethode *getMObjects()* über alle M-Objects iteriert und zu jedem die entsprechende Instanz in der Beispiel-

realisierung aus [Diwo11] ermittelt. Von dieser Instanz wurden dann jene Slots berücksichtigt, die einer Dateneigenschaft entsprachen. Anschließend wurde diese Eigenschaft inklusive der darin enthaltenen Daten (und des Datentyps) mittels der ebenfalls von der OWLData*Factory* bereitgestellten Methode *getOWLDataPropertyAssertionAxiom(OWLDataProperty property, OWLNamedIndividual mobject, OWLLiteral literal)* dem jeweiligen *OWLNamedIndividual* des OWL Modells hinzugefügt. Während Abbildung 25 die erstellten OWL Dateneigenschaften zeigt, werden in Abbildung 26 beispielhaft die zugewiesenen OWL Dateneigenschaften au*thor* (mit String „J.K. Rowling“) und *listprice* (mit float 11.5) zum M-Object *HarryPotter4* gezeigt.

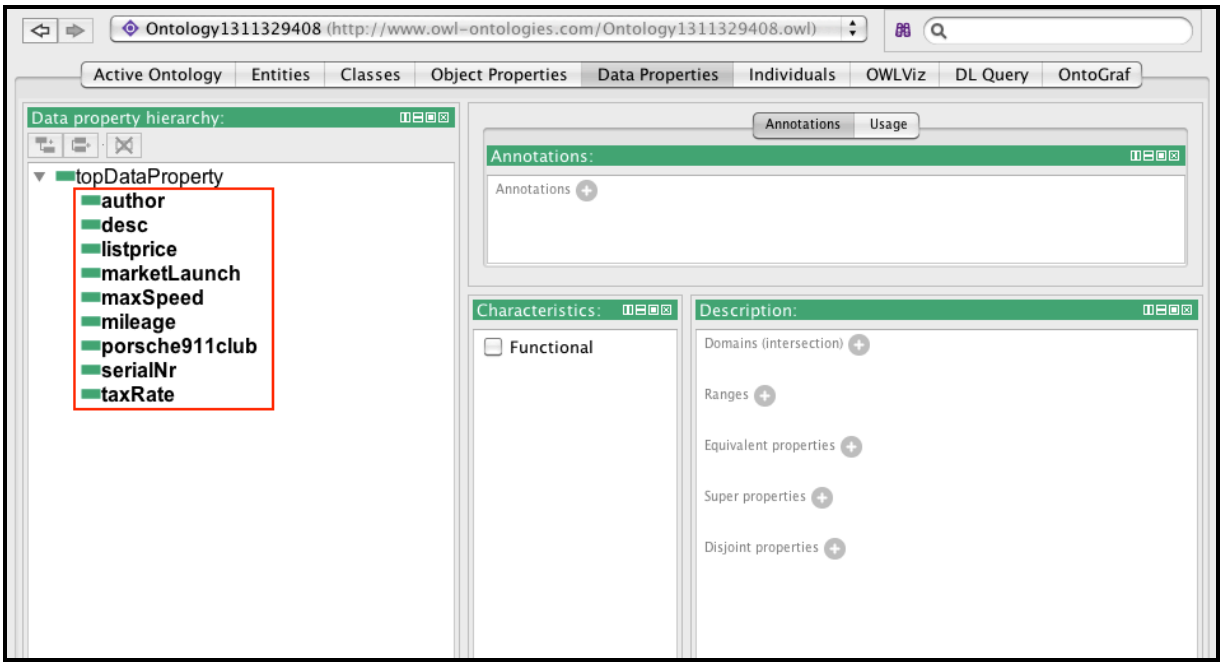

Abbildung 25: OWL Dateneigenschaften

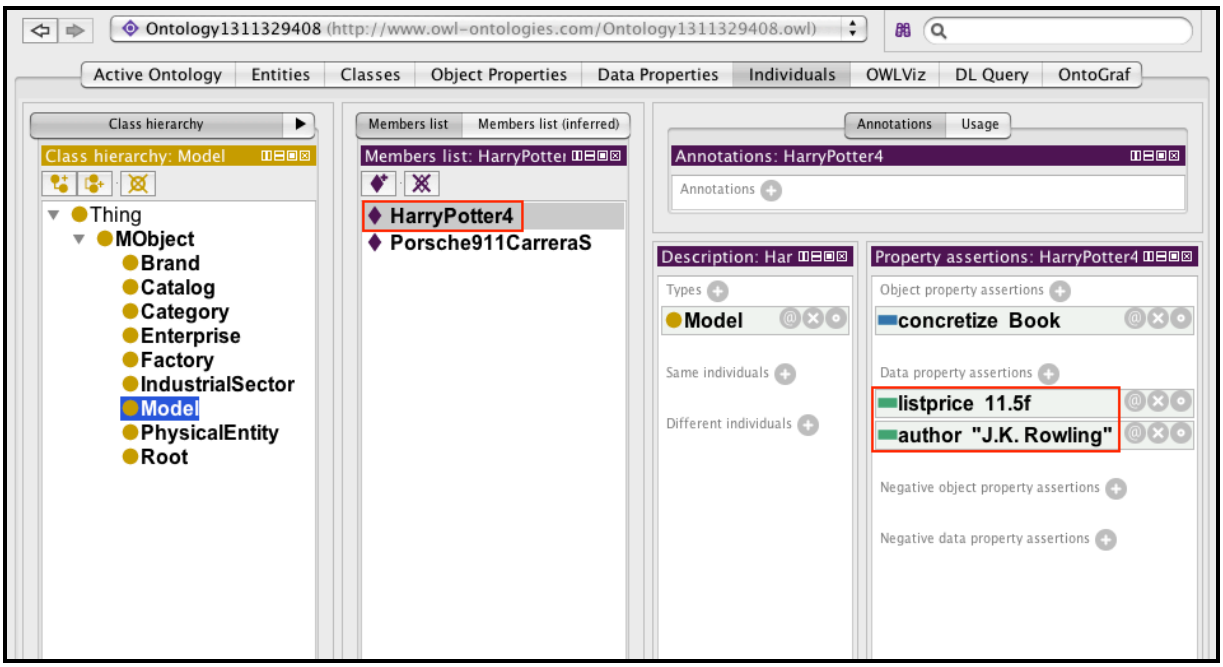

Abbildung 26: HarryPotter4 mit OWL Dateneigenschaften author & listprice

3.2.5 Attributschema

Waren die bisher realisierten Zeilen des Mapping Algorithmus noch nicht so spannend bzw. aufwändig, so wird es in den folgenden Zeilen etwas interessanter:

4: **for all** $o \in O$ **do**

8: **for all** $a \in (A_o \setminus \hat{A}_o)$ **do**

9: **assert IC:** $\exists$concretize_t.$\{[o]\} \sqcap [l_o(a)] \sqsubseteq \forall[a].[d_o(a)] \sqcap =1\ [a].\top$

In diesen Zeilen erfolgt die Realisierung des ersten Integrity Constraints. Dieser IC stellt sicher, dass auf einem bestimmten Level eines M-Objects eine Dateneigenschaft nur genau einen definierten Datentyp (beispielsweise muss bei *Product* auf dem Level *Category taxrate* immer vom Typ Integer sein) und nur genau einen Wert haben darf. Im eben genannten Beispiel soll es also nicht erlaubt sein mehrere Werte für *taxrate* zu vergeben beziehungsweise Werte die nicht vom Typ Integer sind.

Dieser IC wird auf jeden Level eines M-Objects angewendet mit Ausnahme des Top-Levels. Konkretisiert ein M-Object ein anderes M-Object, dann wird bei diesem der IC nur auf jene Levels angewendet, die Dateneigenschaften enthalten, welche im

übergeordneten M-Object nicht enthalten sind, da die bereits vorhandenen Dateneigenschaften ohnehin vom übergeordneten M-Object geerbt werden. Somit wird der IC auf jede Dateneigenschaft genau einmal angewendet. Um so gut wie möglich erklären zu können, wie bei der Realisierung dieser Anforderung vorgegangen wurde, werde ich meine Vorgehensweise nun schrittweise erläutern. Es wurde wiederum über alle M-Objects des OWL Modells iteriert und für jedes M-Object folgende Schritte durchgeführt:

Schritt 1: Zunächst wurde mittels der von der *OWLDataFactory* bereitgestellten Methode *getOWLObjectOneOf(OWLNamedIndividual individual)* eine so genannte *OWLObjectOneOf* Klasse erzeugt. Aus dieser Klasse und der Objekteigenschaft *concretize_t* konnte anschließend mittels der Methode *getOWLObjectSomeValuesFrom(OWLObjectProperty concretize_t, OWLObjectOneOf oneOf)* der Teil ∃concretize_t.{[o]} aus Zeile 9 des Mapping Algorithmus realisiert werden.

Schritt 2: In der Beispielrealisierung aus [Diwo11] enthält jede Instanz einen Slot namens *definesClass*, welcher alle Klassen beinhaltet, die von dieser Instanz definiert werden. Diese Klassen enthalten wiederum einen Slot namens *definesLevel*, welche den Level enthalten, der von dieser Klasse definiert wird. Handelte es sich bei diesem Level um den Top-Level, wurde diese Klasse übersprungen. Bei allen anderen Klassen wurde anschließend abgefragt, ob sie eine Dateneigenschaft enthalten, die sie nicht von einer übergeordneten Klasse geerbt haben. War das der Fall wurde mit Hilfe des ermittelten Levels der entsprechende M-Level des OWL Modells ermittelt. Zu guter Letzt wurde mit dem Ergebnis aus Schritt 1 und dem ermittelten M-Level mittels der Methode *getOWLObjectIntersectionOf(OWLSomeValuesFrom owlSome, OWLClass mlevel)* die Realisierung des linken Teiles (Subklasse) des IC abgeschlossen, was in Zeile 9 des Mapping Algorithmus dem Ausdruck ∃concretize_t.{[o]} ⊓ [lo(a)] entspricht.

Schritt 3: Da in Schritt 2 bereits die relevanten Dateneigenschaften ermittelt wurden, wurde in diesem Schritt zunächst die entsprechende OWL Dateneigenschaft des OWL Modells ermittelt. Daran anschließend wurde aus der Beispielrealisierung aus [Diwo11] der Datentyp dieser Eigenschaft ausgelesen. Als nächstes wurde ähnlich wie in Schritt 1 vorgegangen, nur dass diesmal die Methode *getOWLDataAllValues-*

From(OWLDataProperty property, OWLDatatype type) angewendet wurde, wodurch der Teil ∀[a].[do(a)] aus Zeile 9 des Mapping Algorithmus realisiert wurde.

Schritt 4: In diesem Schritt wurde eine Kardinalität mit dem Wert 1 erzeugt und der jeweiligen OWL Dateneigenschaft zugewiesen, so dass sichergestellt wurde, dass sie nur einen Wert annehmen kann. Das entspricht in Zeile 9 des Mapping Algorithmus dem Ausdruck =1 [a].⊤

Schritt 5: Die Ergebnisse aus Schritt 3 und 4 wurden ähnlich wie in Schritt 2 mittels der Methode getOWLObjectIntersectionOf(OWLDataAllValuesFrom owlAll, OWLDataExactCardinality cardinality) zusammengefügt, womit der rechte Teil des IC aus Zeile 9 des Mapping Algorithmus (Superklasse, ∀[a].[do(a)] ⊓ =1 [a].⊤) ebenfalls umgesetzt war.

Schritt 6: Im 6. und somit letzten Schritt wurde mittels der Methode *getOWLSubClassOfAxiom(OWLObjectIntersectionOf intersection1, OWLObjectIntersectionOf intersection2, Set<OWLAnnotation> annotations)* dem Ergebnis aus Schritt 2 das Ergebnis aus Schritt 5 als Superklasse zugewiesen. Da es sich in diesem Punkt um einen Integrity Constraint handelt, habe ich, wie Sie der Methode bereits entnehmen konnten, darüber hinaus dem OWL*SubClassOfAxiom* die in Punkt 3.1.3 erstellte Annotation Property als Parameter übergeben.

Somit war die Umsetzung dieses IC abgeschlossen. Da es sich bei den erstellten Klassen um keine benannten Klassen handelt und diese darüber hinaus auch keine Subklassen einer benannten Klasse sind, werden sie in Protégé OWL bei den General Class Axioms angezeigt, wie in Abbildung 27 zu sehen ist. Abbildung 28 zeigt, wie die Annotation Property dem *OWLSubClassOfAxiom* hinzugefügt wurde.

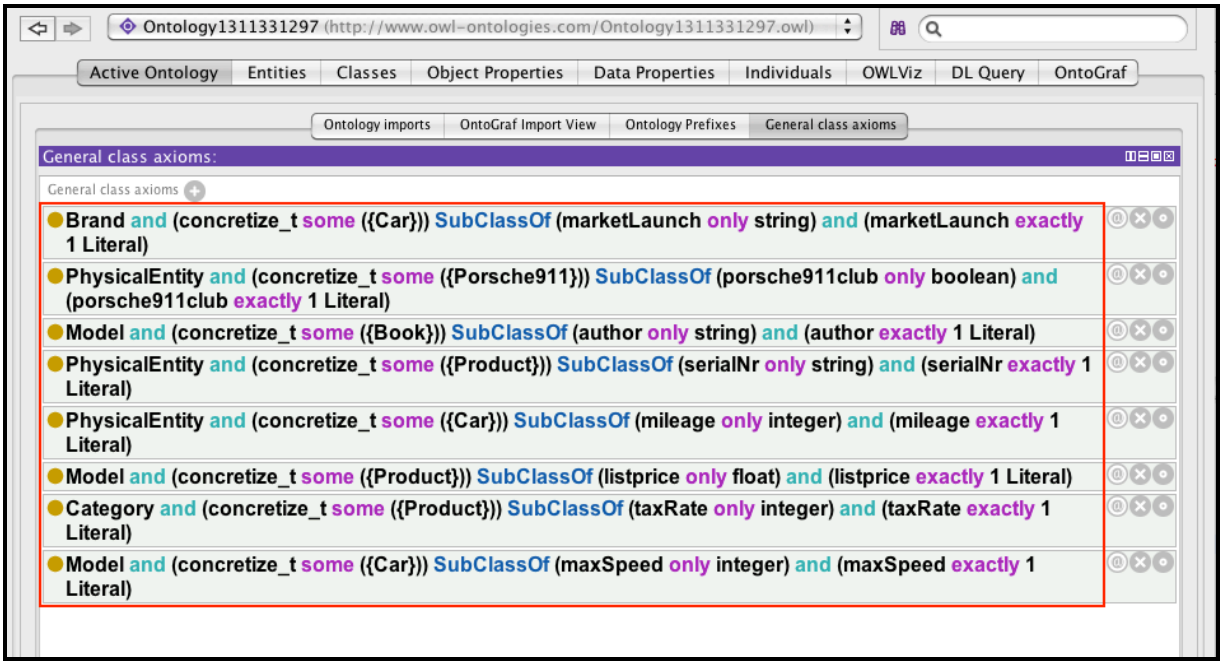

Abbildung 27: Durch IC (Zeile 9) erzeugte Axiome

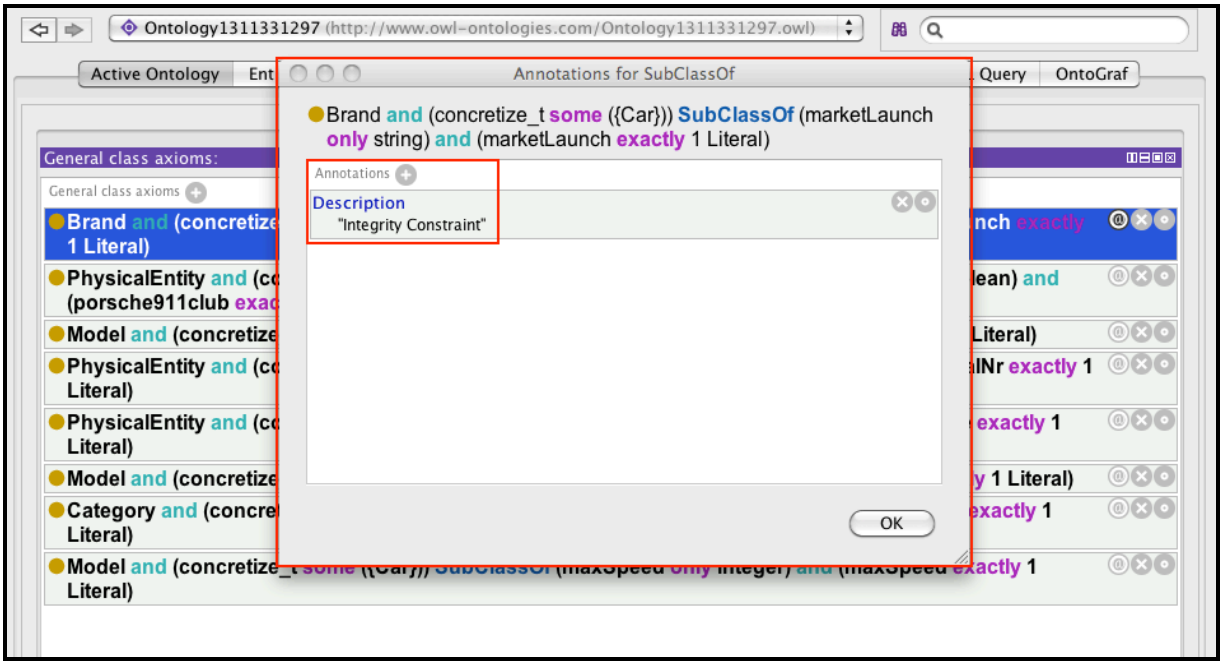

Abbildung 28: OWLSubClassOfAxiom mit Annotation

3.2.6 Vererbung von Eigenschaftswerten

Dieser Punkt bezieht sich auf die Zeilen 4, 8 und 10 des Mapping Algorithmus:

4: **for all** $o \in O$ **do**

8: **for all** $a \in (A_o \setminus \hat{A}_o)$ **do**

10: **if** $v_o(a)$ is defined **then assert:** $[l_o(a)] \sqcap \exists \text{concretize\_t}.\{[o]\} \sqsubseteq \exists [a].\{[v_o(a)]\}$

Werden auf einem Level eines M-Ojects, welches kein Top-Level darstellt, Eigenschaftswerte definiert, so müssen diese Werte von M-Objects auf darunterliegenden Ebenen geteilt werden. Würde beispielsweise in Abbildung 4 beim M-Object *Product* auf dem Level *Category* für die Dateneigenschaft *taxrate* ein Wert (zum Beispiel 20) definiert werden, dann würden die beiden M-Objects *Car* und *Book* diesen Wert erben. Bei der Realisierung wurde wiederum über alle M-Objects iteriert und für jedes M-Object folgende Schritte durchgeführt:

Zunächst wurde zum jeweiligen M-Object die entsprechende Instanz in der Beispielrealisierung aus [Diwo11] ermittelt. Anschließend wurden die im Slot *definesClass* enthaltenen Klassen dieser Instanz ausgelesen, ähnlich der Vorgehensweise in 3.2.5. Diese Klassen wurden in weiterer Folge durchlaufen. War eine Klasse ident zum direkten Typen dieser Instanz so wurde diese Klasse übersprungen. Ansonsten wurde anschließend geprüft, ob eine dieser Klassen eine Dateneigenschaft (*D*) enthielt, die sie nicht von einer übergeordneten Klasse geerbt hatte und *D* darüber hinaus einen Wert enthielt.

War das der Fall, mussten die Subklassen dieser Klasse ermittelt werden, da die in den Subklassen enthaltenen Instanzen diesen Wert erben und somit für die entsprechenden M-Objects des OWL Modells von Interesse waren. Neben den Subklassen wurden die in den Slots *definedAtObject* (M-Object) und *definesLevel* (M-Level) enthaltenen Objekte ermittelt. Aus dem soeben ermittelten M-Object wurde anschließend eine *OWLObjectOneOf* Klasse und aus dem ermittelten Level eine *OWLClass* erstellt. In weiterer Folge konnte mit der Objekteigenschaft *concretize_t* und der eben erstellten *OWLObjectOneOf* Klasse eine *OWLObjectSomeValuesFrom* Klasse erstellt werden, mit der wiederum in Kombination mit der zuvor erstellten *OWLClass*

eine *OWLObjectIntersectionOf* Klasse erzeugt werden konnte. Somit war der Subklassenteil der Zeile 10 des Mapping Algorithmus ($[I_o(a)] \sqcap \exists$concretize_t.$\{[o]\}$) abgeschlossen.

Zur Umsetzung des Superklassenteiles wurde zunächst aus dem in *D* enthaltenen Wert eine *OWLDataOneOf* Klasse und anschließend aus dieser Klasse in Kombination mit *D* eine *OWLDataSomeValuesFrom* Klasse erstellt (entspricht $\exists[a].\{[v_o(a)]\}$ im Mapping Algorithmus). Zum Abschluss wurde aus der zuvor erstellten *OWLObjectIntersectionOf* Klasse und der eben erstellten *OWLDataSomeValuesFrom* Klasse ein *OWLSubClassOfAxiom* erzeugt, womit die Zeile 10 des Mapping Algorithmus vollständig realisiert war.

In Abbildung 4 bzw. der Beispielrealisierung aus [Diwo11] wird dieser Teil des Mapping Algorithmus nicht berücksichtigt. Daher musste ich, um ein Ergebnis zu erhalten, das Beispiel dahingehend abändern, dass *Product* auf dem Level *Category* einen Wert von 20 für *taxrate* festlegt. Da es sich wie in Punkt 3.2.5 um anonyme Klassen handelt, werden auch diese Axiome bei den General Class Axioms angezeigt, wie in Abbildung 29 zu sehen ist (in diesem Fall nur ein erzeugtes Axiom). Da es sich jedoch um keinen IC handelt, wird hier auf das Hinzufügen der Annotation Property verzichtet.

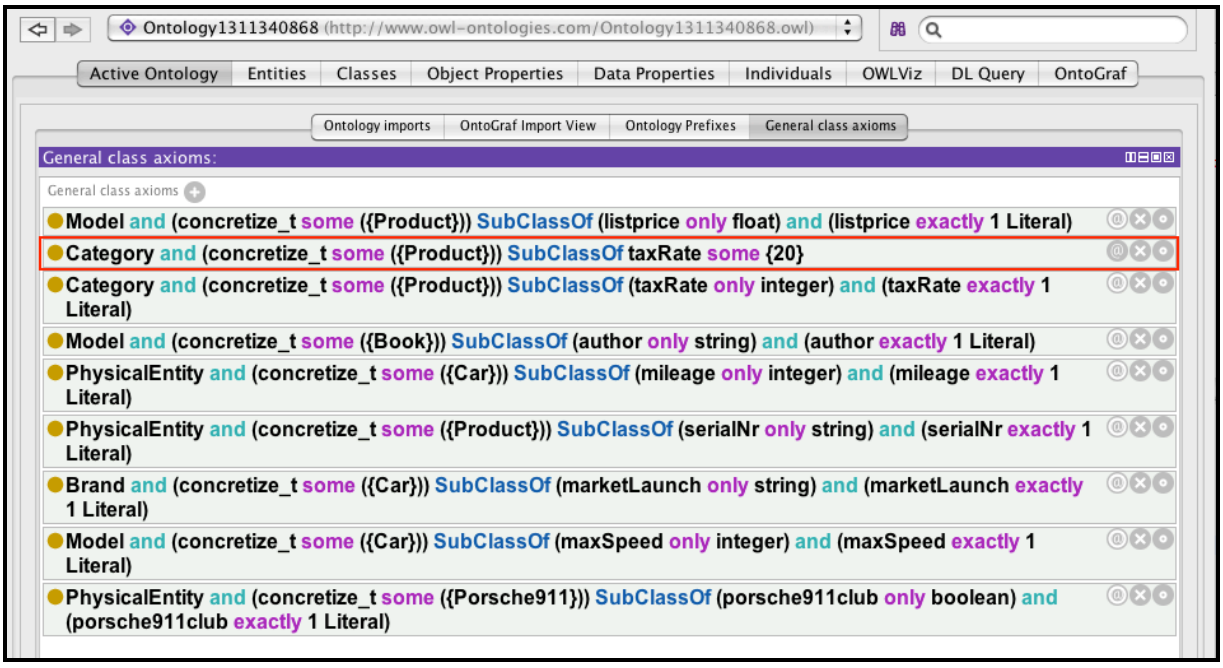

Abbildung 29: Durch Zeile 10 erzeugtes Axiom

3.2.7 Level-Hierarchie

In den Zeilen 4, 11 und 12 des Mapping Algorithmus erfolgt die Umsetzung des nächsten Integrity Constraints:

4: **for all** $o \in O$ **do**
11: **for all** $(l,l') \in P_o : l' \neq \hat{l}_o \wedge (\nexists o' \in O : (o,o') \in H \wedge (l,l') \in P_{o'})$ **do**
12: **assert IC:** $\exists$concretize_t.{[o]} $\sqcap$ [l] $\sqsubseteq \exists$concretize_t.($\exists$concretize_t.{[o]} $\sqcap$ [l'])

Auf dem ersten Blick wirkt dieser IC noch etwas komplizierter als jener aus Punkt 3.2.5. Wie Sie jedoch schnell merken werden, unterscheidet er sich nicht großartig vom ersten IC. Dieser Integrity Constraint soll die einzelnen Levels eines M-Objects (mit Ausnahme des Top-Levels) zueinander in Beziehung setzen. Nehmen sie beispielsweise das M-Object *Product*. *Product* definiert die Levels *Catalog*, *Category*, *Model* und *PhysicalEntity*, wobei *Catalog* als Top-Level in diesem Punkt nicht weiter von Bedeutung ist. Der unterste Level von *Product* ist *PhysicalEntity*, dessen direkt übergeordneter Level *Model* ist. Der übergeordnete Level von *Model* ist wiederum *Category*. Es ergibt sich also die Level-Hierarchie *Category* gefolgt von *Model* und schließlich gefolgt von *PhysicalEntity*.

Bei den beiden Konkretisierungen von *Product*, nämlich *Car* und *Book*, braucht diese Hierarchie nicht erneut festgelegt werden, da sie diese erben. Wird jedoch ein neuer Level (siehe *Brand* beim M-Object *Car*) eingeführt, muss die Hierarchie angepasst werden. In diesem Fall bleibt *Model* weiterhin der übergeordnete Level von *PhysicalEntity*, daher braucht hier auch nichts angepasst werden. *Model* hingegen hat nun *Brand* als neuen übergeordneten Level, was zur Folge hat, dass die Hierarchie von *Car* dementsprechend angepasst werden muss. Bei der Realisierung bin ich dabei folgendermaßen vorgegangen:

Schritt 1: Die Umsetzung dieses IC erfolgte top down, also vom abstraktesten bis hin zum konkretesten M-Object. Daher wurden zunächst mittels der Methode *getTopMObjects()* jene M-Objects aus dem OWL Modell ermittelt, welche keine Konkretisierung eines anderen M-Objects darstellten. Danach wurde aus der Beispielrealisierung aus [Diwo11] die entsprechende Instanz ermittelt und aus dessen Slot *defines*-

Class jene Klassen ausgelesen, welche dieses M-Object definieren. Von jeder dieser Klassen wurden anschließend die Werte aus den Slots *concretizationOfType* und *definesLevel* ermittelt. Danach musste geprüft werden, ob es sich beim ermittelten Level aus dem Slot *definesLevel* um den Top-Level dieses M-Objects handelte. Hierfür wurde die Hilfsmethode *getTopMLevel(String mObjectName)* verwendet, welche zu einem M-Object den jeweiligen Top-Level zurückliefert. Handelte es sich um den Top-Level, so wurde dieser Level der Collection *levels* hinzugefügt. In dieser Collection wurden alle Levels gesammelt, die für diesen IC nicht weiter von Bedeutung waren. Handelte es sich nicht um den Top-Level (*TL1*), musste anschließend geprüft werden, ob die übergeordnete Klasse den Top-Level (*TL2*) definierte. War das der Fall, wurde der Level ebenfalls *levels* hinzugefügt, wenn nicht, war ein relevantes Level für diesen IC gefunden.

Schritt 2: Da in Schritt 1 bereits das relevante M-Object und der relevante Level (*TL1*) ermittelt wurden, konnte der Subklassenteil des IC (∃concretize_t.{[*o*]} ⊓ [*l*]) bereits umgesetzt werden. Zuerst wurde mittels des M-Objects eine *OWLObjectOneOf* Klasse und daraus anschließend in Kombination mit *concretize_t* eine *OWLObjectSomeValuesFrom* Klasse erstellt. Danach wurde aus *TL1* und *OWLObjectSomeValuesFrom* eine *OWLObjectIntersectionOf* Klasse erstellt, womit der Subklassenteil dieses ICs abgeschlossen war.

Schritt 3: In Schritt 1 wurde ebenfalls bereits der übergeordnete Level (*TL2*) ermittelt. Wie bereits bekannt, durfte es sich bei diesem Level nicht um den Top-Level des M-Objects handeln. Da im Superklassenteil dieses ICs die gleiche *OWLObjectSomeValuesFrom* Klasse noch einmal vorkommt wie im Subklassenteil konnte in diesem Schritt sofort eine neue *OWLObjectIntersectionOf* Klasse aus *TL2* und dieser Klasse erzeugt werden. Aus der eben erzeugten Klasse und *concretize_t* wurde anschließend erneut eine *OWLObjectSomeValuesFrom* Klasse erzeugt, wodurch die Realisierung des Superklassenteiles ebenfalls abgeschlossen war.

Schritt 4: Zum Abschluss wurde aus dem Ergebnis aus Schritt 2 und dem Ergebnis aus Schritt 3 ein *OWLSubClassOfAxiom* (inklusive Annotation Property, siehe Punkt 3.2.5) erzeugt und der übergeordnete Level ebenfalls der Collection *levels* hinzugefügt. Nachdem alle Top-M-Objects abgearbeitet waren, wurden die Schritte 1 – 4

auch für alle anderen M-Objects durchgeführt, bis schließlich alle M-Objects abgearbeitet waren.

Abbildung 30 zeigt die durch diesen Integrity Constraint erzeugten Axiome (aus Gründen der besseren Leserlichkeit wurden die in 3.2.5 und 3.2.6 erzeugten Axiome aus dieser Abbildung entfernt).

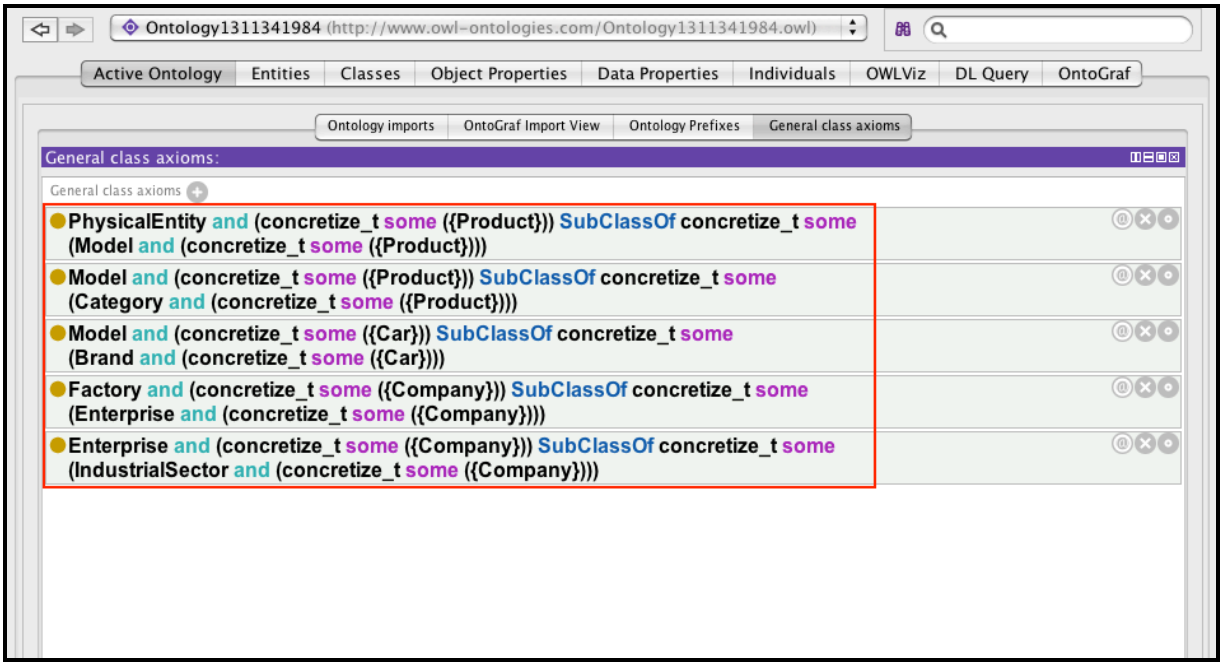

Abbildung 30: Durch IC (Zeile 12) erzeugte Axiome

3.2.8 Festlegung erlaubter Level-Eigenschaften

Durch die Zeilen 4, 13 und 14 des Mapping Algorithmus wird sichergestellt, dass eine Dateneigenschaft nur auf einem bestimmten Level eines M-Objects (beziehungsweise Konkretisierungen dieses M-Objects) vorkommen darf:

4: **for all** $o \in O$ **do**

13: **for all** $a \in A_o : \nexists o' \in O : (o,o') \in H \wedge a \in A_{o'}$ **do**

14: **assert IC:** $\exists[a].\top \sqsubseteq (\exists \text{concretize\_t}.\{[o]\} \sqcup \{[o]\}) \sqcap [I_o(a)]$

Betrachten wir beispielsweise die Dateneigenschaft *taxrate*. Der IC stellt sicher, dass *taxrate* nur auf dem Level *Category* vorkommen darf und darüber hinaus, dass *taxrate* lediglich bei jenem M-Object, welches es definiert (*Product*) und bei Konkretisie-

rungen von *Product* die diesen Level ebenfalls enthalten (*Book*, *Car*) vorkommen darf. Bei der Umsetzung bin ich folgendermaßen vorgegangen:

Schritt 1: Zuerst wurde über alle Instanzen der Beispielrealisierung aus [Diwo11] iteriert und zu jeder Instanz (vorausgesetzt es handelte sich um eine *Default Simple Instance* mit einem eigenen Slot namens *concretizationOf*) alle im Slot *definesClass* enthaltenen Klassen ausgelesen. Anschließend wurden zu jeder dieser Klassen jene Dateneigenschaften ermittelt, die nicht von einer übergeordneten Klasse geerbt wurden. Somit erhielt ich zu jedem M-Object jene Dateneigenschaften, welche für diesen IC relevant waren.

Schritt 2: Im zweiten Schritt wurde zunächst die entsprechende Dateneigenschaft im OWL Modell ermittelt und daran anschließend aus dieser eine *OWLDataSomeValuesFrom* Klasse erstellt. Als Datentyp wurde der Eigenschaft dabei Literal (entspricht dem Top-Datentyp) übergeben. Somit war der Subklassenteil des IC ($\exists[a].\top$) abgeschlossen.

Schritt 3: Als nächstes wurde aus der Instanz (Schritt 1) eine *OWLObjectOneOf* Klasse erstellt. Aus dieser Klasse und *concretize_t* konnte anschließend eine *OWLObjectSomeValuesFrom* Klasse erstellt werden. In weiterer Folge wurde aus der *OWLObjectOneOf* Klasse und der *OWLObjectSomeValuesFrom* Klasse eine *OWLObjectUnionOf* Klasse erstellt (entspricht $\exists$concretize_t.{[*o*]} $\sqcup$ {[*o*]} aus Zeile 14 des Mapping Algorithmus). Aus der in Schritt 1 ausgelesenen Klasse wurde anschließend aus dem Slot *definesLevel* der entsprechende Level ermittelt. Mit diesem Level und der erzeugten *OWLObjectUnionOf* Klasse wurde zum Abschluss des Superklassenteils dieses ICs (($\exists$concretize_t.{[*o*]} $\sqcup$ {[*o*]}) $\sqcap$ [l_o(*a*)]) eine *OWLObjectIntersectionOf* Klasse erstellt.

Schritt 4: Im letzten Schritt wurden die Ergebnisse aus Schritt 2 und 3 wieder mittels eines *OWLSubClassOfAxioms* (inklusive Annotation Property) zueinander in Beziehung gesetzt.

Die durch diesen IC erzeugten Axiome werden in Abbildung 31 veranschaulicht, wobei ich wiederum aus Gründen der besseren Leserlichkeit jene Axiome aus der Abbildung entfernt habe, welche in den Punkten 3.2.5 - 3.2.7 erzeugt wurden.

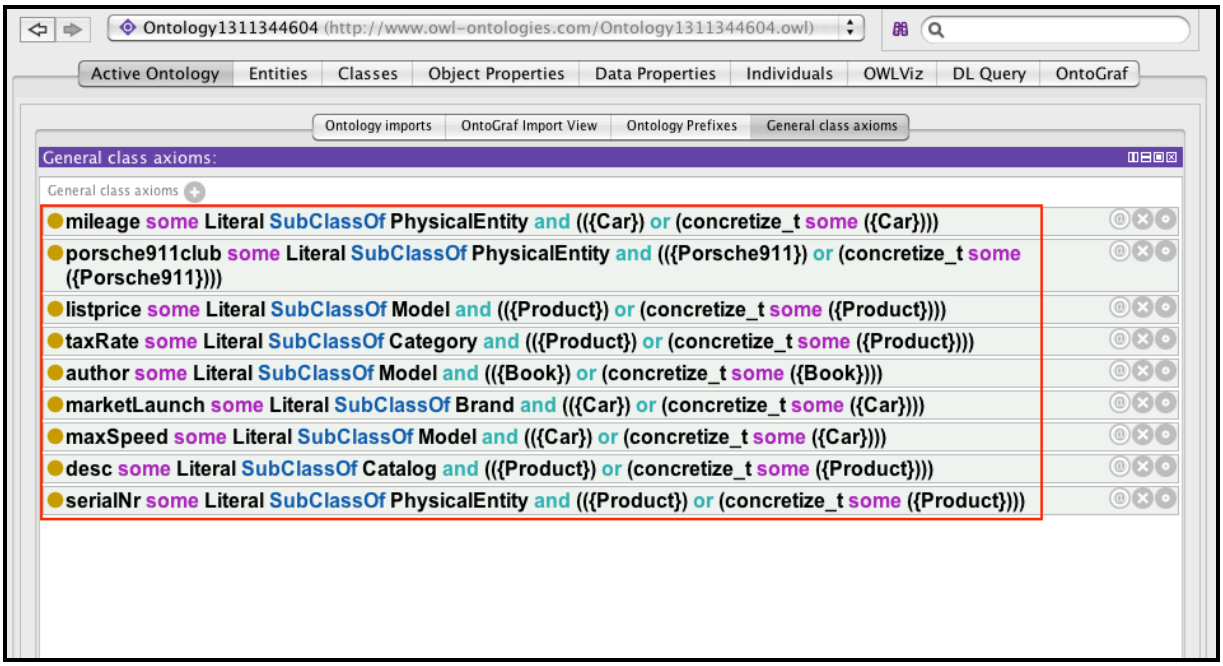

Abbildung 31: Durch IC (Zeile 14) erzeugte Axiome

3.2.9 Festlegung der M-Object-Zugehörigkeit von Levels

Der letzte der 4 ICs im Rahmen des Mappings der M-Objects nach OWL stellte sich als der am einfachsten zu realisierende dar, ausgedrückt durch die Zeilen 4, 15 und 16 des Mapping Algorithmus:

4: **for all** $o \in O$ **do**

15: **for all** $l \in L_o : l \neq \hat{l}_o \wedge (\nexists o' \in O : (o,o') \in H \wedge l \in L_{o'})$ **do**

16: **assert IC:** $[l] \sqsubseteq \exists$concretize_t.$\{[o]\}$

Dieser IC stellt sicher, dass ein Level zu einem bestimmten M-Object (und dessen Konkretisierungen) gehört. Dieser IC wird nur auf jenen Levels angewendet, welche nicht den Top-Level darstellen beziehungsweise nicht bereits durch ein übergeordnetes M-Object definiert wurden, beispielsweise wird bei *Car* lediglich der Level *Brand* berücksichtigt, da alle anderen Levels bereits durch *Product* definiert wurden. Um diesen Umstand zu realisieren bin ich folgendermaßen vorgegangen:

Schritt 1: Zunächst wurde über alle Instanzen der Beispielrealisierung aus [Diwo11] iteriert und geprüft, ob es sich bei der Instanz um eine Instanz einer *Default Simple Instance* handelte, ob die Instanz einen eigenen Slot namens *concretizationOf* besaß und die Instanz in der Collection der Top-M-Objects enthalten war. Die Prüfung, ob es sich um ein Top-M-Object handelte wurde aus dem Grund vorgenommen, da diese M-Objects keine Konkretisierungen anderer M-Objects darstellten und daher auch keine Levels enthalten konnten, die bereits in einem übergeordneten M-Object definiert wurden. Anschließend wurden jene im Slot *definesClass* enthaltenen Klassen, welche durch diese Instanz definiert werden, ermittelt. Bekanntermaßen enthalten diese Klassen den Slot *definesLevel* aus dem anschließend der entsprechende Level ermittelt werden konnte. Darüber hinaus enthalten diese Klassen einen Slot namens *concretizationOfType*. War dieser Slot leer, wurde der ermittelte Level der Collection *levels* hinzugefügt, da es sich dabei um den Top-Level handelte und daher in diesem Punkt nicht weiter von Belang war. Andernfalls war ein entsprechender Level für diesen IC gefunden, welcher auch gleichzeitig die Subklasse (linker Teil des IC) darstellt.

Schritt 2: Da bereits das benötigte M-Object bekannt war, konnte daraus im nächsten Schritt eine *OWLObjectOneOf* Klasse erzeugt werden. Aus dieser Klasse und *concretize_t* wurde anschließend eine *OWLObjectSomeValuesFrom* Klasse erstellt, wodurch der rechte Teil des IC (∃concretize_t.{[*o*]}) ebenfalls bereits realisiert war.

Schritt 3: Im letzten Schritt wurde mittels *OWLSubClassOfAxiom* (inklusive Annotation Property) dem ermittelten Level aus Schritt 1 die erzeugte *OWLObjectSomeValuesFrom* Klasse aus Schritt 2 als Superklasse zugewiesen. Der eben bearbeitete Level wurde anschließend ebenfalls der Collection *levels* hinzugefügt, da er für diesen IC nicht weiter von Bedeutung war.

Diese Schritte wurden für jeden gefundenen Level wiederholt. Nachdem jeder Level der Top-M-Objects abgearbeitet war, wurde top down in den Konkretisierungen dieser M-Objects nach Levels gesucht, die noch nicht in der Collection *levels* enthalten waren. Wurde ein noch nicht vorhandener Level gefunden, wurde dieser ebenfalls entsprechend der Schritte 1 – 3 bearbeitet. So wurde erreicht, dass auch keine Le-

vels, welche erst weiter unten in der Hierarchie eingeführt wurden, übersehen werden konnten.

Abbildung 32 zeigt beispielhaft für den Level *Category*, dass dieser nun eine Subklasse von ∃concretize_t.{Product} darstellt.

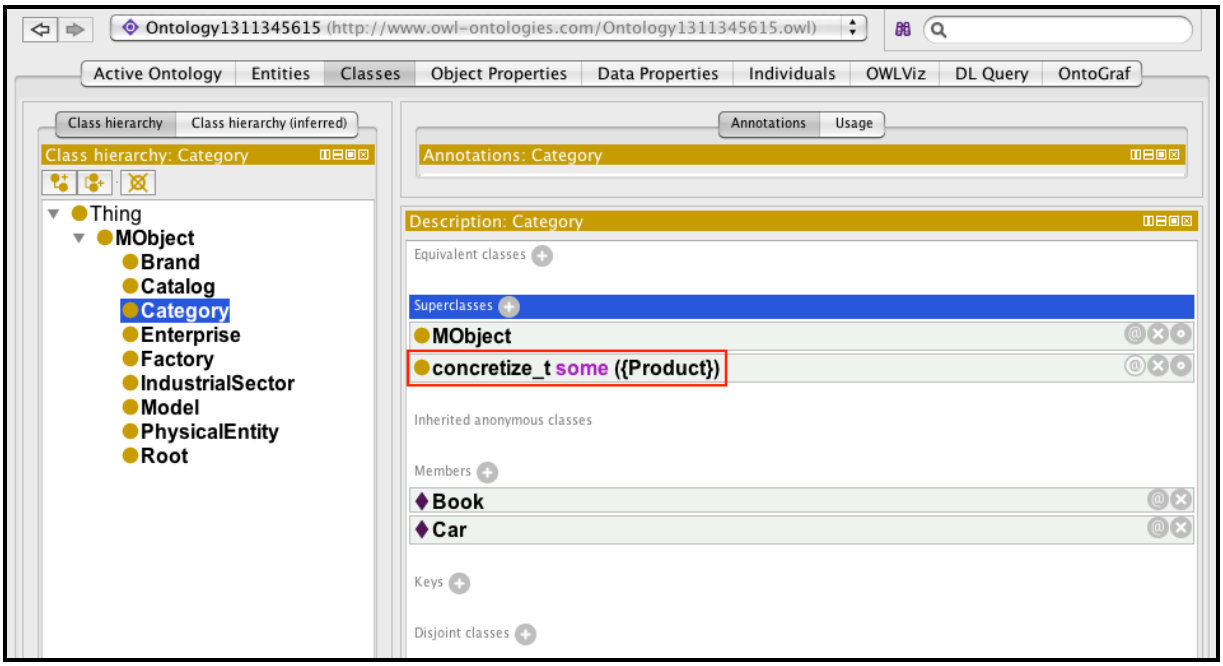

Abbildung 32: Category als Subklasse von ∃concretize_t.{Product}

3.2.10 Sicherstellung Disjoint von M-Levels

In Zeile 17 des Mapping Algorithmus wird sichergestellt, dass jeder Level in einer Disjoint-Beziehung mit allen anderen Levels steht:

17: **for all** $l \in L$, $l' \in (L \setminus \{l\})$ **do assert:** $[l] \sqcap [l'] \sqsubseteq \bot$

Dabei wurden zunächst mittels der von mir zuvor definierten Methode *getMLevels()* alle Levels des OWL Modells ermittelt und anschließend daran jedem Level mittels der von der *OWLDataFactory* bereitgestellten Methode *getOWLDisjointClassesAxiom(OWLClass class1, OWLClass class2)* alle anderen Levels als Disjoint-Klassen hinzugefügt.

Die folgende Abbildung zeigt beispielhaft für den Level *Catalog*, wie die einzelnen Levels als Disjoint-Klassen (Disjoint-Levels) hinzugefügt wurden.

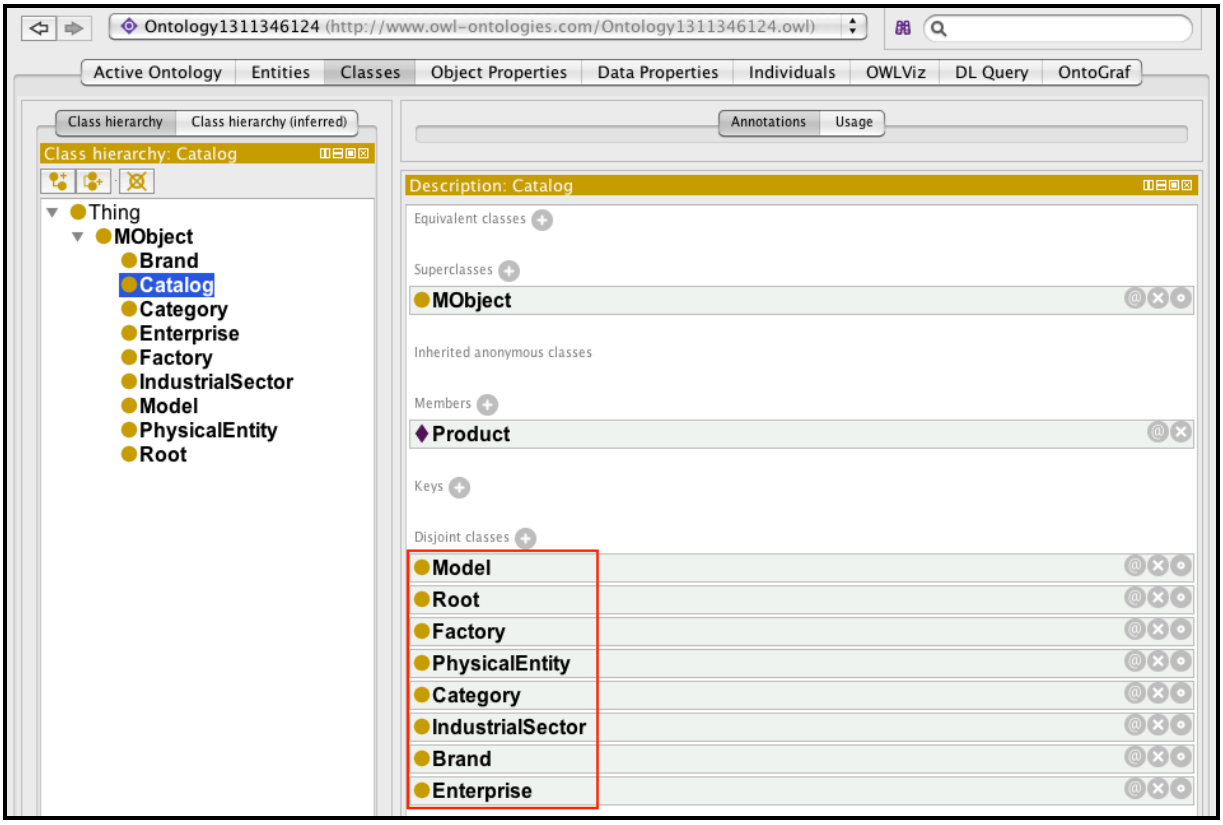

Abbildung 33: Hinzugefügte Disjoint-Klassen (Levels) zum Level Catalog

3.2.11 Sicherstellung Unique Name Assumption

In Zeile 18 des Mapping Algorithmus wird das Mapping der M-Objects nach OWL abgeschlossen:

18: **for all** $o \in O, o' \in (O \setminus \{o\})$ **do assert:** $[o] \not\approx [o']$

Durch diese Zeile soll sichergestellt werden, dass es nicht möglich beziehungsweise erlaubt sein darf mehrere M-Objects mit dem gleichen Namen zu versehen. Darüber hinaus wird sichergestellt, dass es sich bei allen M-Objects des OWL Modells um unterschiedliche Individuen handelt.

Die Vorgehensweise bei der Realisierung dieser Zeile ist schnell erklärt. Zuerst wurden die einzelnen M-Objects des OWL Modells ermittelt und jedem M-Object mittels

der von der OWLData*Factory* bereitgestellten Methode *getOWLDifferentIndividualsAxiom(OWLNamedIndividual individual1, OWLNamedIndividual individual2)* alle anderen M-Objects als unterschiedliche M-Objects hinzugefügt. Damit wurde sichergestellt, dass es sich bei allen um einzigartige Individuen handelt. Abbildung 34 zeigt diesen Umstand beispielhaft für *Book*.

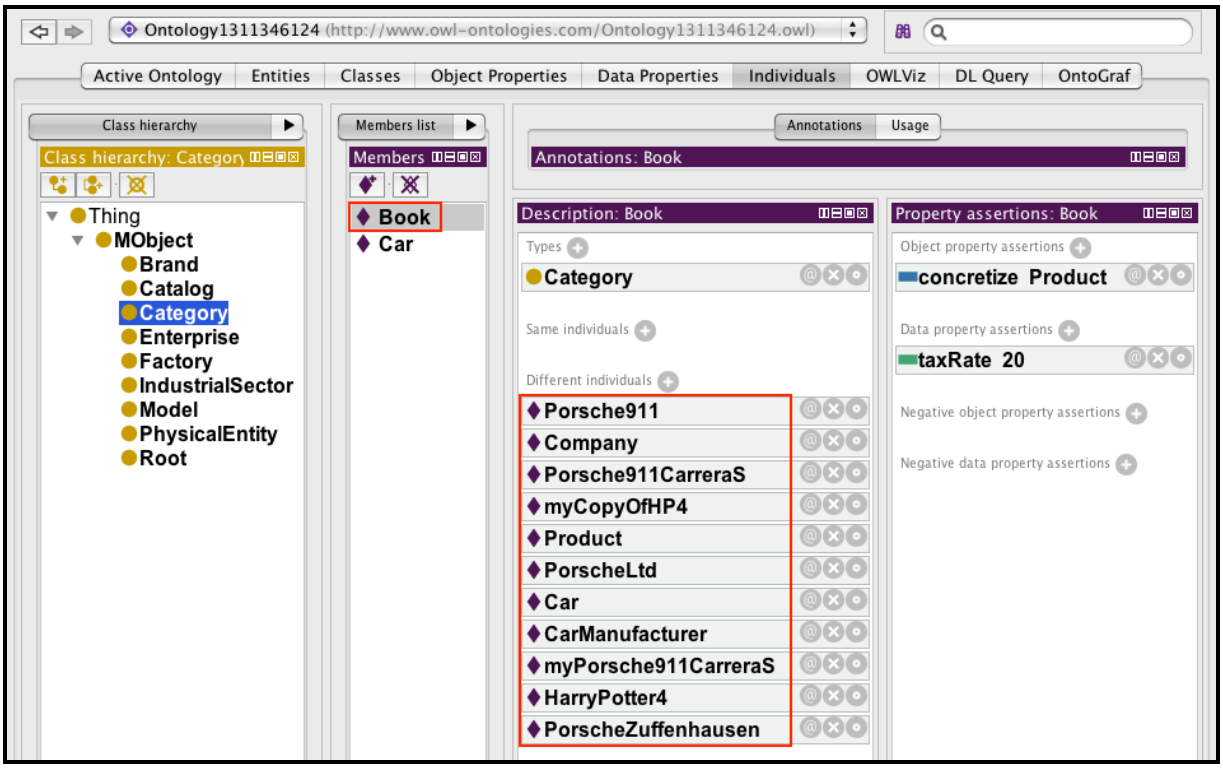

Abbildung 34: Unique Name Assumption für Book

3.3 Mapping der M-Relationships nach OWL

Nachdem nun alle M-Objects erfolgreich nach OWL gemappt wurden, geht es in diesem Punkt um die Realisierung der Zeilen 19 – 27 des Mapping Algorithmus, in denen das Mapping der M-Relationships nach OWL umgesetzt wird. Wie sie schnell erkennen werden, stellen sich manche Zeilen sehr ähnlich zu jenen aus Punkt 3.2 dar.

3.3.1 Festlegen der Eigenschaften source und target

In Zeile 19 des Mapping Algorithmus werden die OWL Objekteigenschaften *source* und *target* festgelegt:

19: **assert:** $\top \sqsubseteq \leq 1 source \sqcap \leq 1 target$

Durch diese beiden Eigenschaften wird das Start- und Ziel-M-Object einer M-Relationship festgelegt. Um mit diesen OWL Objekteigenschaften arbeiten zu können, mussten diese zunächst erzeugt und anschließend festgelegt werden, dass es sich dabei um funktionale Eigenschaften handelt, wodurch diese Zeile des Mapping Algorithmus auch bereits umgesetzt war. Da ich dabei gleichermaßen vorgegangen bin, wie bei der OWL Objekteigenschaft *concretize* (siehe Punkt 3.2.1), werde ich hier darauf verzichten, auf meine Vorgehensweise bei deren Erstellung genauer einzugehen. In der folgenden Abbildung ist zu sehen, wie sich diese beiden OWL Objekteigenschaften nach deren Erstellung in Protégé OWL darstellen.

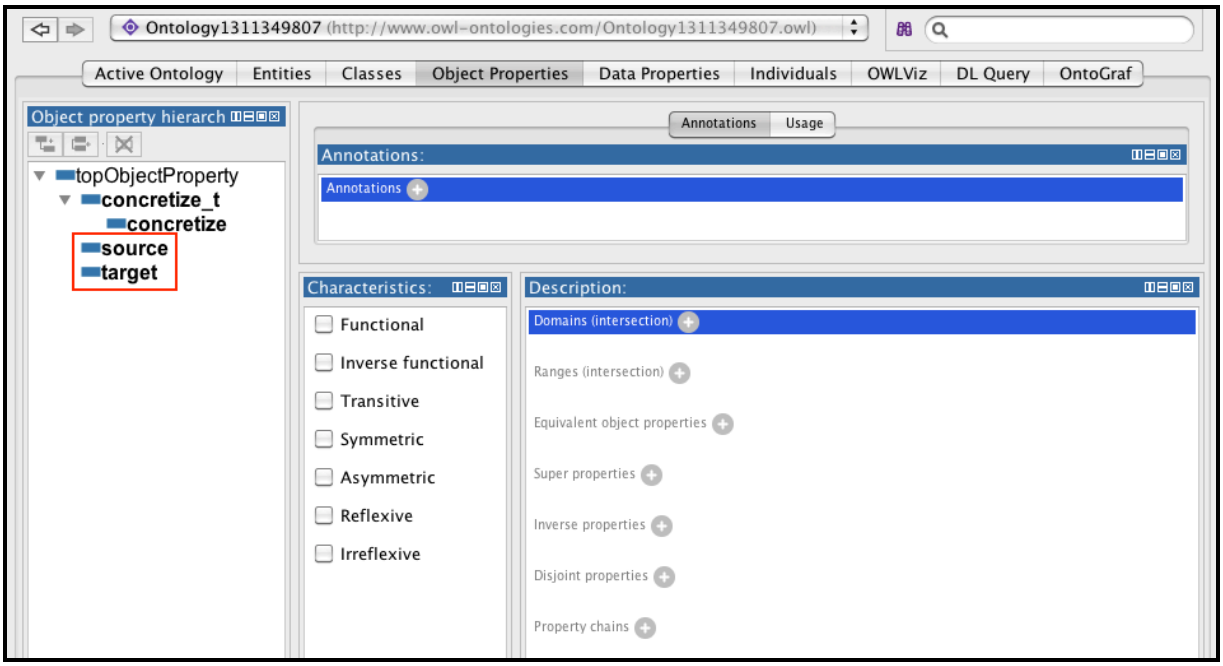

Abbildung 35: OWL Objekteigenschaften source und target

3.3.2 Erstellung und Konkretisierung der M-Relationships

In den Zeilen 20 und 21 des Mapping Algorithmus erfolgt die Konkretisierung der M-Relationships (analog zur Konkretisierung der M-Objects):

20: **for all** $r \in R$ **do**
21: **if** $\exists r' : (r,r') \in H_R$ **then assert:** concretize([r],[r'])

Da diese noch nicht erstellt wurden, musste dieser Schritt vor deren Konkretisierung noch durchgeführt werden. Dabei wurde zunächst, ähnlich zu den M-Objects, mittels der von der OWLData*Factory* zur Verfügung gestellten Methode *getOWLClass(IRI OWLClassIRI)* die Klasse *MRelationship erzeugt*.

Da M-Relationships in Protégé als Instanzen (Individuen) dargestellt werden, musste über alle Instanzen der Beispielrealisierung aus [Diwo11] iteriert werden. Berücksichtigung fanden jedoch nur jene Instanzen, welche eine Instanz einer *Default Simple Instance* waren und einen eigenen Slot namens *concretizeRelation* hatten, da es sich nur bei diesen Instanzen um die gewünschten M-Relationships handelte.

Mittels der von der OWLData*Factory* bereitgestellten Methode *getOWLNamedIndividual(IRI OWLNamedIndividualIRI)* konnte anschließend die entsprechende M-Relationship für OWL erzeugt werden. Mittels des Axioms *OWLClassAssertionAxiom(OWLClass MRelationship, OWLNamedIndividual owlNamedIndividual)* wurden die M-Relationships in weiterer Folge der Klasse *MRelationship* hinzugefügt, siehe Abbildung 36 auf der nächsten Seite.

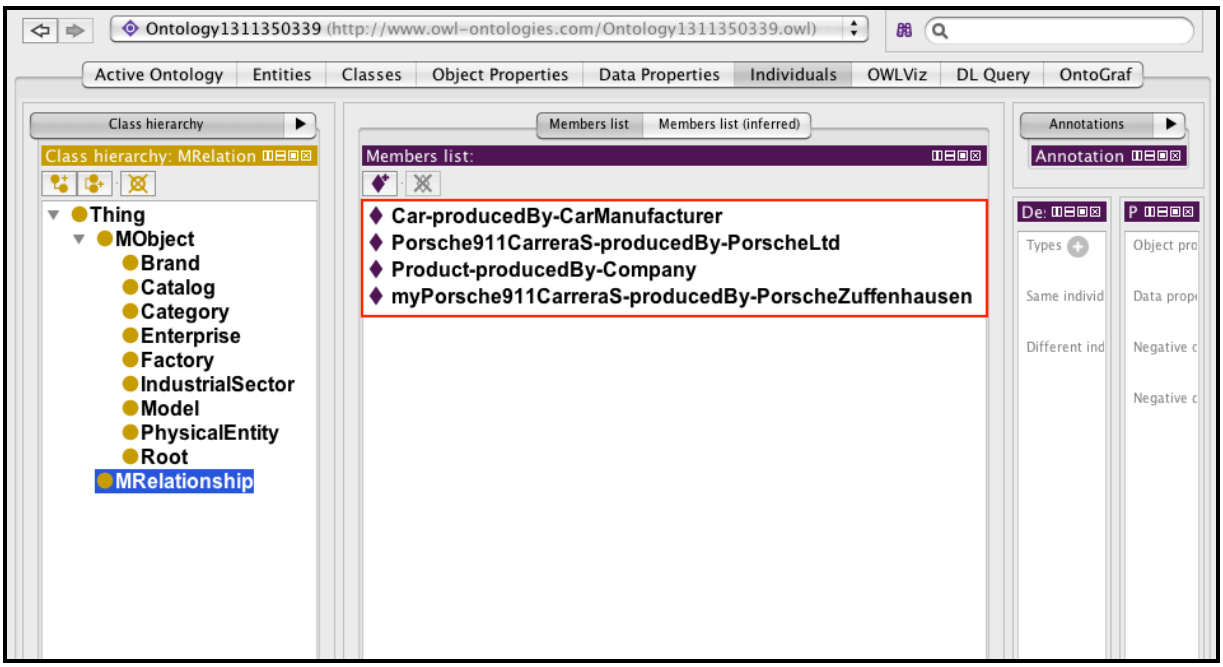

Abbildung 36: M-Relationships

Um die eben erstellten M-Relationships in weiterer Folge zu konkretisieren, wurden zunächst mittels der zuvor definierten Methode *getMRelationships()* alle bereits erstellten M-Relationships des OWL Modells ausgelesen und anschließend über diese iteriert. Zum jeweilig aktuellen M-Relationship wurde anschließend die entsprechende Instanz in der Beispielrealisierung aus [Diwo11] ermittelt.

War der Slot *concretizeRelation* dieser Instanz leer, war diese M-Relationship nicht weiter von Bedeutung, da sie keine Konkretisierung einer anderen M-Relationship darstellte. Man kann sagen, dass es sich dabei um eine Top-M-Relationship handelt. Ansonsten wurde die im Slot *concretizeRelation* enthaltene M-Relationship ausgelesen und anschließend mittels der von der *OWLDataFactory* bereitgestellten Methode *getOWLObjectPropertyAssertionAxiom(OWLObjectProperty concretize, OWLNamedIndividual individual1, OWLNamedIndividual individual2)* die Konkretisierung vorgenommen.

Abbildung 37 zeigt beispielhaft die Konkretisierung der M-Relationship *Product-producedBy-Company* durch die M-Relationship *Car-producedBy-CarManufacturer*, siehe folgende Seite.

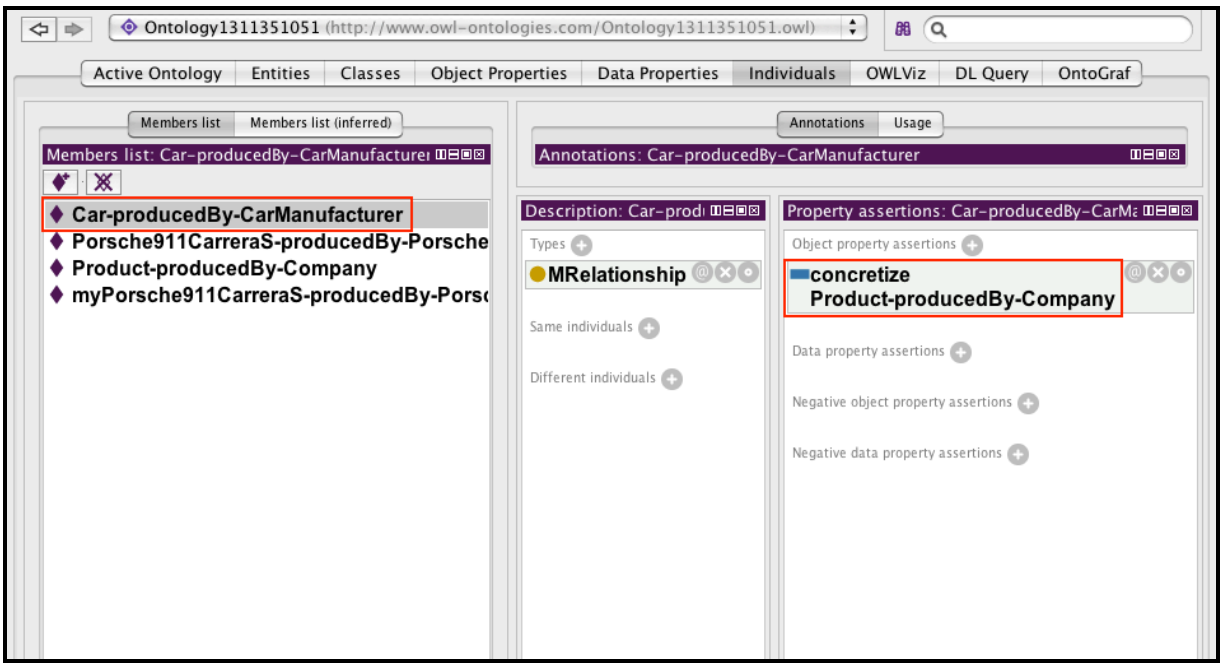

Abbildung 37: Konkretisierung von Product-producedBy-Company

3.3.3 Ermittlung und Zuweisung von source und target

In den Zeilen 20, 22 und 23 des Mapping Algorithmus erfolgt die Ermittlung und Zuweisung der Objekteigenschaften *source* und *target* und der darin enthaltenen Start- und Ziel-M-Objects zu der M-Relationship:

20: **for all** $r \in R$ **do**
22: **assert:** source([r],[s_r])
23: **assert:** target([r],[t_r])

Mein Vorgehen dabei ist schnell erklärt, da die Vorgehensweise beinahe ident zu jener aus Punkt 3.3.2 ist. Einziger Unterschied ist, dass *source* und *target* auch dann ermittelt und zugewiesen wurden, wenn *concretizeRelation* leer war. Während das Attribut der Objekteigenschaft *source* aus dem Slot *relationSource* ermittelt wurde, wurde das Attribut von *target* aus dem Slot *relationTarget* ausgelesen. Somit waren alle Objekte bekannt, welche zur Realisierung der Zeilen 22 und 23 benötigt wurden.

Die Zuweisung der Attribute konnte gleichermaßen vorgenommen werden wie bereits im vorangegangenen Punkt bei der Konkretisierung der M-Relationships beschrieben wurde. Als Objekteigenschaft wurde der Methode diesmal natürlich nicht

concretize, sondern *source* beziehungsweise *target* als Parameter übergeben. In Abbildung 38 wird gezeigt, welches Bild sich ergibt, nachdem *Car* als *source* und *CarManufacturer* als *target* der M-Relationship *Car-producedBy-CarManufacturer* zugewiesen wurden.

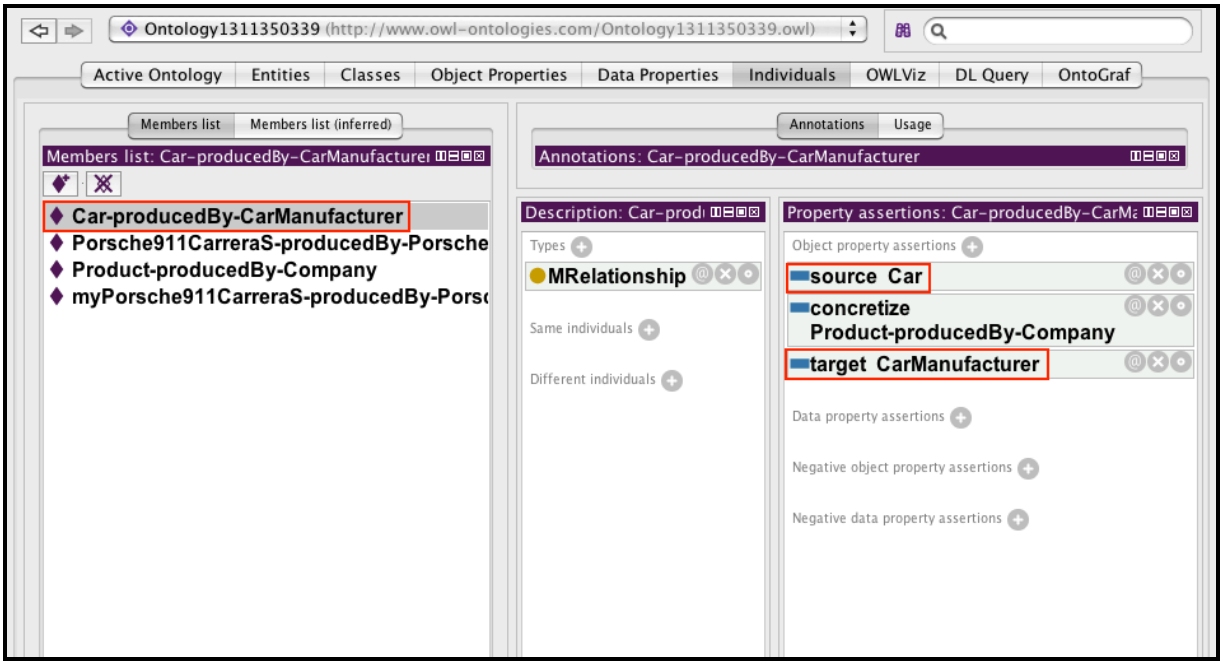

Abbildung 38: Zuweisung von source und target zu Car-producedBy-Company

3.3.4 Source- und Target-Level Konkretisierung

In den Zeilen 20 und 24 des Mapping Algorithmus erfolgt die Umsetzung des bereits fünften Integrity Constraints:

20: **for all** $r \in R$ **do**

24: **assert IC:** $\exists$concretize_t.{[r]} $\sqsubseteq$ ($\forall$source.($\exists$concretize_t.{[s_r]} $\sqcup$ {[s_r]}) $\sqcap$ $\forall$target.$\exists$concretize_t.{[t_r]}) $\sqcup$ ($\forall$source.$\exists$concretize_t.{[s_r]} $\sqcap$ $\forall$target.($\exists$concretize_t.{[t_r]} $\sqcup$ {[t_r]}))

Durch diesen IC werden Domain und Range von Konkretisierungen einer M-Relationship beschränkt. Beispielsweise haben alle direkten oder indirekten Konkretisierungen von *Product-producedBy-Company* eine direkte oder indirekte Konkretisierung von *Product*, oder *Product* selbst als *source* und eine direkte oder indirekte

Konkretisierung von *Company*, oder *Company* selbst als *target*. Entweder *Product* oder *Company* muss konkretisiert werden. [Neum09b]

Um diesen IC umzusetzen, wurde zunächst wiederum über alle M-Relationships iteriert und anschließend für jede M-Relationship folgende Schritte durchgeführt:

Schritt 1: Im ersten Schritt wurde zunächst mittels der M-Relationship eine *OWLObjectOneOf* Klasse erstellt. Aus dieser Klasse und *concretize_t* wurde anschließend eine *OWLObjectSomeValuesFrom* Klasse gebildet. Somit war der Subklassenteil (∃concretize_t.{[*r*]}) dieses IC umgesetzt.

Schritt 2: Als nächstes wurden die *OWLNamedIndividuals* der Objekteigenschaften *source* und *target* ermittelt. Mit diesen *OWLNamedIndividuals* wurden anschließend wiederum *OWLObjectOneOf* Klassen (owl*ObjOOSource*, *owlObjOOTarget*) erstellt.

Schritt 3: Aus einer Kombination von *concretize_t* und *owlObjOOSource* wurde in diesem Schritt zunächst eine *OWLObjectSomeValuesFrom* Klasse (*owlObjSVF1*) erstellt. Aus der eben erstellten Klasse und *owlObjOOSource* wurde anschließend eine *OWLObjectUnionOf* Klasse gebildet. In weiterer Folge wurde aus der Objekteigenschaft *source* und der eben erstellten *OWLObjectUnionOf* Klasse eine *OWLObjectAllValuesFrom* Klasse (*owlObjAVF1*) erstellt. Danach wurde mit *owlObjOOTarget* und *concretize_t* eine *OWLObjectSomeValuesFrom* Klasse (*owlObjSVF2*) erstellt, aus der anschließend in Kombination mit der Objekteigenschaft *target* eine weitere *OWLObjectAllValuesFrom* Klasse (*owlObjAVF2*) erstellt wurde. Als Schlusspunkt dieses Schritts wurde aus *owlObjAVF1* und *owlObjAVF2* eine *OWLObjectIntersectionOf* Klasse erstellt, womit (∀source.(∃concretize_t.{[s_r]} ⊔ {[s_r]}) ⊓ ∀target.∃concretize_t.{[t_r]}) umgesetzt war.

Schritt 4: In diesem Schritt konnte ich auf einige der im letzten Schritt erzeugten Klassen zurückgreifen, wodurch mir einiges an Arbeit erspart blieb. Aus *source* und *owlObjSVF1* wurde zunächst eine *OWLObjectAllValuesFrom* Klasse (*owlObjAVF3*) erzeugt, welche anschließend dazu diente in Verbindung mit *owlObjSVF2* eine *OWLObjectUnionOf* Klasse zu erstellen. Aus dieser neu erzeugten Klasse und *target* wurde danach wiederum eine *OWLObjectAllValuesFrom* Klasse (*owlObjAVF4*) er-

stellt. Als letztes wurde in diesem Schritt aus *owlObjAVF3* und *owlObjAVF4* eine weitere *OWLObjectIntersectionOf* Klasse erzeugt, wodurch (∀source.∃concretize_t.{[s_r]} ⊓ ∀target.(∃concretize_t.{[t_r]} ⊔ {[t_r]})) ebenfalls realisiert war.

Schritt 5: Abschließend wurde aus den in Schritt 3 und 4 erzeugten *OWLObjectIntersectionOf* Klassen eine *OWLObjectUnionOf* Klasse erstellt, welche mittels *SubClassOfAxiom* (inklusive Annotation Property) der in Schritt 1 erzeugten Subklasse als Superklasse zugewiesen wurde.

In Abbildung 39 werden die durch diesen Integrity Constraint erzeugten Axiome gezeigt, welche in Protégé OWL abermals bei den General Class Axioms angezeigt werden. Wie bereits bei den ICs zuvor habe ich auch in dieser Abbildung die bereits vor diesem Punkt erstellten ICs aus Gründen der besseren Leserlichkeit entfernt.

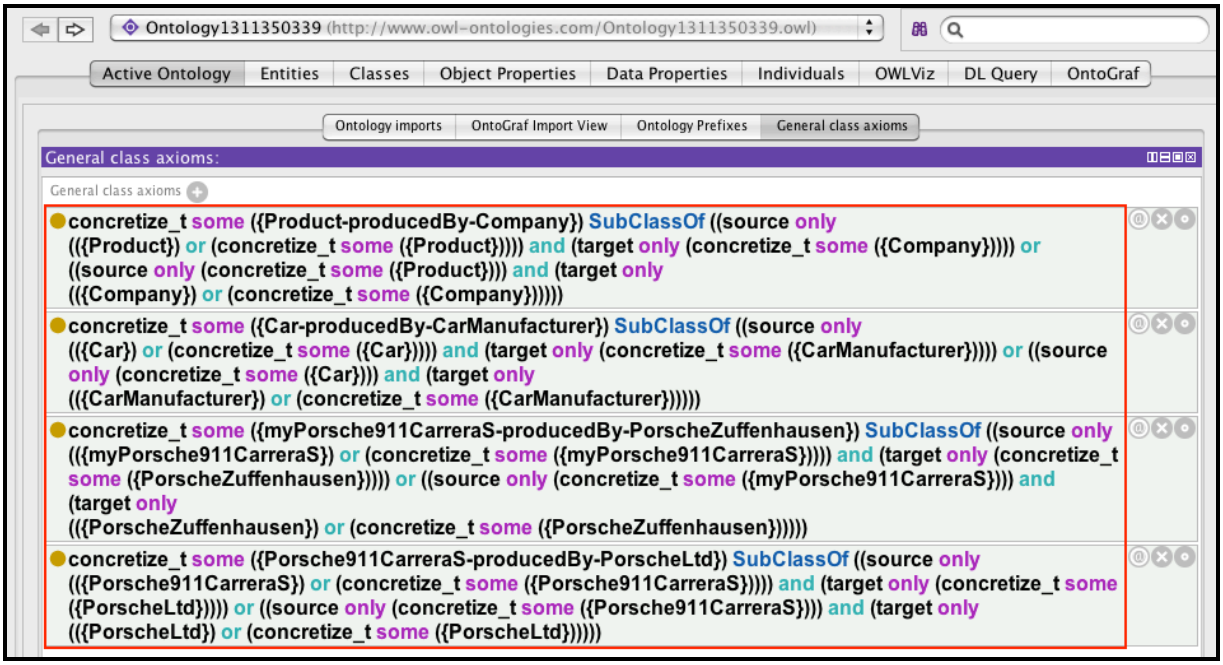

Abbildung 39: Durch IC (Zeile 24) erzeugte Axiome

3.3.5 Sicherstellung der Navigation entlang von M-Relationships

In den Zeilen 20, 25 und 26 wird der sechste und somit letzte IC des Mapping Algorithmus umgesetzt:

20: **for all** $r \in R$ **do**

25: **for all** $(I,I') \in C_r : I \neq \hat{I}_{sr} \vee I' \neq \hat{I}_{tr}$ **do**

26: **assert IC:** ∃concretize_t.{[*r*]} ⊓ (∃source.∃concretize_t.[*I*] ⊔ ∃target.∃concretize_t.[*I'*]) ⊑ ∃concretize_t.(∃concretize_t.{[*r*]} ⊓ ∃source.[*I*] ⊓ ∃target.[*I'*])

Durch diesen IC soll die sichere Navigation entlang von M-Relationships auf höher gelegenen Ebenen sichergestellt werden (mittels so genannter *Connection Levels*). Beispielsweise müssen alle M-Relationships, welche unter dem *Connection Level* (*Category, IndustrialSector*) liegen und eine Konkretisierung der M-Relationship *Product-producedBy-Company* darstellen, diese auch auf dem *Connection Level* (*Category, IndustrialSector*) konkretisieren. [Neum09b]

Bei der Realisierung dieses IC für Protégé OWL wurde wieder über alle M-Relationships iteriert und mit jedem M-Relationship folgende Schritte durchgeführt:

Schritt 1: Mit dem M-Relationship wurde zunächst eine *OWLObjectOneOf* Klasse erstellt und daraus anschließend in Verbindung mit *concretize_t* eine *OWLObjectSomeValuesFrom* Klasse. Als nächstes wurde die entsprechende Instanz in der Beispielrealisierung aus [Diwo11] ermittelt und die in den Slots *relationSource* und *relationTarget* enthaltenen Instanzen ausgelesen.

Schritt 2: In weiterer Folge wurden auch die im Slot *connectionLevel* dieser Instanz enthaltenen Level-Beziehungen (*Connection-Levels*) ausgelesen und diese mit einem Iterator durchlaufen. Eine Level-Beziehung besteht aus den beiden Slots *connectionSource* und *connectionTarget*. Also wurden aus jeder Level-Beziehung diese beiden Slots ausgelesen, wobei keiner dieser Slots einen Top-Level beinhalten durfte. Handelte es sich um keine Top-Level, wurden anschließend die entsprechenden Level (*levelSource*, *levelTarget*) im OWL Modell ermittelt.

Schritt 3: In diesem Schritt wurde zunächst aus *concretize_t* und *levelSource* eine *OWLObjectSomeValuesFrom* Klasse erstellt, aus welcher gleich darauf in Kombination mit *source* wiederum eine *OWLObjectSomeValuesFrom* Klasse gebildet werden konnte. Dasselbe wurde anschließend mit *levelTarget* durchgeführt. Die mit *le-*

velSource und *levelTarget* erstellten Klassen wurden in weiterer Folge in einer *OWLObjectUnionOf* Klasse zusammengefasst. Aus dieser Klasse und der in Schritt 1 erzeugten *OWLObjectSomeValuesFrom* Klasse wurde anschließend eine *OWLObjectIntersectionOf* Klasse gebildet, welche den Subklassenteil (∃concretize_t.{[*r*]} ⊓ (∃source.∃concretize_t.[*l*] ⊔ ∃target.∃concretize_t.[*l'*])) dieses IC darstellt.

Schritt 4: Als nächstes wurden aus der Kombination von *source* und *levelSource* beziehungsweise *target* und *levelTarget* zwei weitere *OWLObjectSomeValuesFrom* Klassen erstellt. Danach wurde aus diesen beiden Klassen und der in Schritt 1 erstellten *OWLObjectSomeValuesFrom* Klasse eine weitere *OWLObjectIntersectionOf* Klasse erstellt. Aus der Kombination von *concretize_t* und dieser Klasse wurde anschließend erneut eine *OWLObjectSomeValuesFrom* Klasse erstellt, welche gleichzeitig den Superklassenteil (∃concretize_t.(∃concretize_t.{[*r*]} ⊓ ∃source.[*l*] ⊓ ∃target.[*l'*])) dieses IC darstellt.

Schritt 5: Wie mittlerweile bereits von der Erstellung aller anderen ICs bekannt ist, wurde im letzten Schritt das Ergebnis aus Schritt 3 dem Ergebnis aus Schritt 4 mittels *OWLSubClassOfAxiom* (inklusive Annotation Property) als Subklasse zugewiesen. Die durch diesen IC erzeugten Axiome können Abbildung 40 entnommen werden.

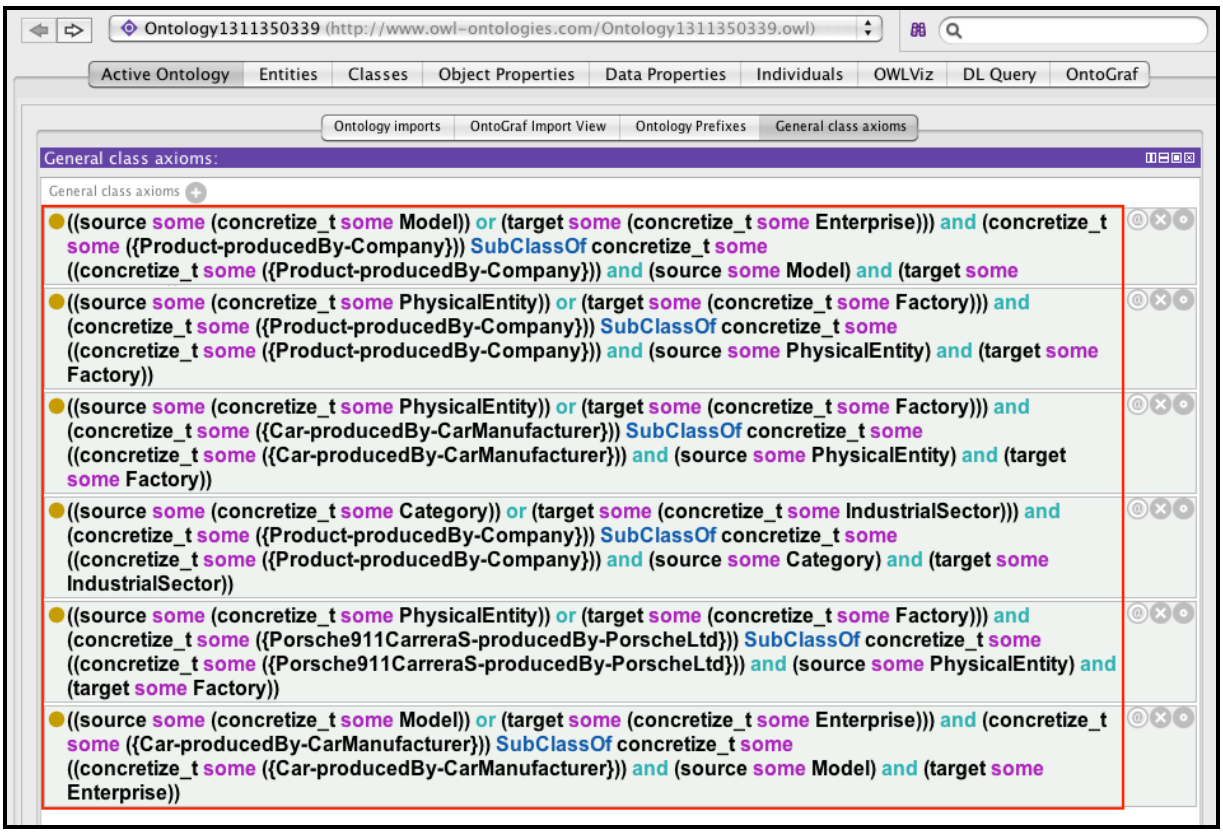

Abbildung 40: Durch IC (Zeile 26) erzeugte Axiome

3.3.6 Sicherstellung Unique Name Assumption

In den Zeilen 20 und 27 des Mapping Algorithmus wird das Mapping der M-Relationships nach OWL abgeschlossen, was gleichzeitig auch das Ende des Algorithmus bedeutet:

20: **for all** $r \in R$ **do**

27: **for all** $r' \in R : r \neq r'$ **do assert:** $[r] \not\approx [r']$

Wie bereits bei der Sicherstellung der Unique Name Assumption der M-Objects soll auch hier sichergestellt werden, dass es nicht möglich beziehungsweise erlaubt sein soll mehrere M-Relationships mit dem gleichen Namen zu versehen. Darüber hinaus wird sichergestellt, dass es sich bei allen M-Relationships um unterschiedliche Individuen handelt. Ich bin bei der Realisierung der Sicherstellung der Unique Name Assumption für die M-Relationships gleichermaßen vorgegangen wie bei den M-Objects (siehe Punkt 3.2.11), nur dass diesmal logischerweise nicht die M-Objects sondern alle M-Relationships durchlaufen wurden und wiederum mit der Methode getOWLDif-

ferentIndividualsAxiom(OWLNamedIndividual individual1, OWLNamedIndividual individual2) alle anderen M-Relationships als Gegensatz zu ihnen hinzugefügt wurden. Somit wurde sichergestellt, dass jede M-Relationship mit einem eindeutigen Namen versehen ist und sich zu allen anderen M-Relationships unterscheidet.

Die folgende Abbildung zeigt beispielhaft wie es in Protégé OWL darstellt wird, dass sich die M-Relationship *Car-producedBy-CarManufacturer* von den M-Relationships *Product-producedBy-Company*, *Porsche911CarreraS-producedBy-PorscheLtd* und *myPorsche911CarreraS-producedBy-PorscheZuffenhausen* unterscheidet.

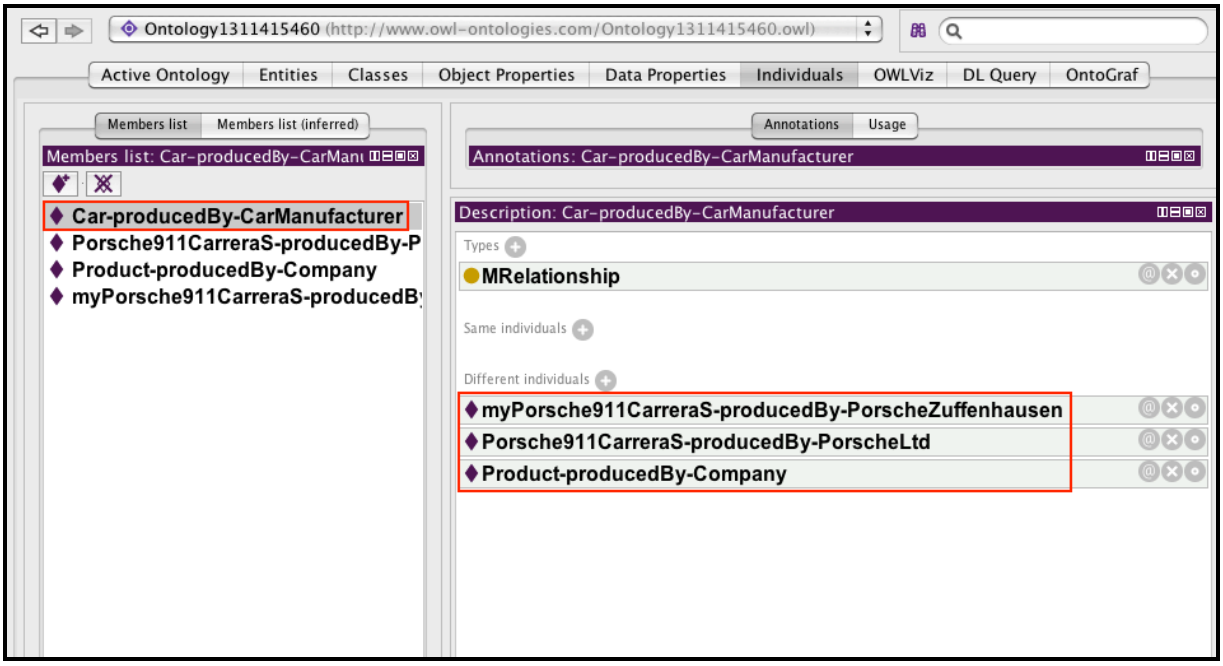

Abbildung 41: Unique Name Assumption Car-producedBy-CarManufacturer

4 Reasoning – Performance Studies

4.1 Testumgebung

In diesem Kapitel wird eine Gegenüberstellung verschiedener Reasoner im Hinblick auf ihr Antwortzeitverhalten und die von ihnen ermittelten Ergebnisse bezüglich der Umsetzung des Mapping Algorithmus aus Kapitel 3 vorgenommen. Ziel dabei ist es, den geeignetsten Reasoner für das Thema dieser Arbeit zu finden. Nähere Informationen zu den von mir gewählten Reasonern Fact++, HermiT und Pellet finden Sie im Anhang unter den Punkten A.1.3 - A.1.5. Getestet wurden diese in folgender Testumgebung:

Betriebssytem:	Mac OS X (Version 10.6.6)
Prozessortyp:	Intel Core i5
Prozessorgeschwindigkeit:	2.4 GHz
Anzahl der Prozessoren:	1
Gesamtzahl der Kerne:	2
L2-Cache (pro Kern):	256 KB
L3-Cache:	3 MB
Speicher:	4 GB 1067 MHz DDR3
Geschwindigkeit für Prozessorverbindungstyp:	4.8 GT/s
Boot-ROM-Version:	MBP61.0057.BOC

4.2 Beispielerweiterung

Da die in 2.4.2 angeführten M-Objects, Abstraktionsebenen und M-Relationships für die in diesem Kapitel durchgeführten Performance Studies nicht ausreichen, um aussagekräftige Ergebnisse über das Laufzeitverhalten der einzelnen Reasoner zu erhalten, musste ich eine Erweiterung des Beispiels aus [Neum09b] vornehmen. Das erweiterte Beispiel enthält 100 M-Objects, 9 Abstraktionsebenen und 40 M-

Relationships. Welche das konkret sind, können der folgenden Tabelle entnommen werden.

M-Objects	Product, Book, Car, Porsche911, HarryPotter4, Porsche911CarreraS, myCopyOfHP4, myPorsche911CarreraS, Company, CarManufacturer, PorscheLtd, PorscheZuffenhausen, Handy, Computer, Shoes, Motorbike, Movie, Furniture, HandyManufacturer, ComputerManufacturer, ShoeManufacturer, Producer, MotorbikeManufacturer, Publisher, FurnitureManufacturer, VWGolf, FiatPunto, MiniCooper, BMW3er, ThePelicanBrief, TheFirm, Misery, Shining, IT, Diabolus, Meteor, TheDaVinciCode, Illuminati, Hangover, Saw, PulpFiction, KillBill, SinCity, VWGolfGTI, FiatPuntoEvo, MiniCooperS, BMW3erCabrio, CopaMundial, PumaShadow, MercurialVapor, Integrity07Plus, Ducati848, KTM50SX, VW, Fiat, Mini, BMW, Adidas, Puma, Nike, Reebok, Ducati, KTM, Carlsen, Bertelsmann, BasteiLübbe, Heyne, myVWGolfGTI, myFiatPuntoEvo, myMiniCooperS, myBMW3erCabrio, myCopaMundial, myPumaShadow, myMercurialVapor, myIntegrity07Plus, myDucati848, myKTM50SX, myCopyOfTPB, myCopyOfTheFirm, myCopyOfMisery, myCopyOfShining, myCopyOfIT, myCopyOfDiabolus, myCopyOfMeteor, myCopyOfTDVC, myCopyOfIlluminati, myCopyOfHangover, myCopyOfSaw, myCopyOfPF, myCopyOfKB, myCopyOfSC, VWWolfsburg, FiatTurin, MiniCowley, BMWMünchen, AdidasHerzogenaurach, PumaNürnberg, NikeBremen, DucatiBorgoPanigale, KTMMattighofen

Abstraktions-ebenen	catalog, category, model, physical entity, brand, root, industrialSector, enterprise, factory

M-Relationships	Product-producedBy-Company, Car-producedBy-CarManufacturer, Porsche911CarreraS-producedBy-PorscheLtd, myPorsche911CarreraS-producedBy-PorscheZuffenhausen, Motorbike-producedBy-MotorbikeManufacturer, Book-producedBy-Publisher, Handy-producedBy-HandyManufacturer, Computer-producedBy-ComputerManufacturer, Shoes-producedBy-ShoeManufacturer, Movie-producedBy-Producer, Furniture-producedBy-FurnitureManufacturer, HarryPotter4-producedBy-Carlsen, ThePelicanBrief-producedBy-Bertelsmann, TheFirm-producedBy-Bertelsmann,

M-Relationships	Misery-producedBy-BasteiLübbe, Shining-producedBy-BasteiLübbe, Diabolus-producedBy-BasteiLübbe, Meteor-producedBy-BasteiLübbe, IT-producedBy-Heyne, TheDaVinciCode-producedBy-BasteiLübbe, Illuminati-producedBy-BasteiLübbe, VWGolfGTI-producedBy-VW, FiatPuntoEvo-producedBy-Fiat, MiniCooperS-producedBy-Mini, BMW3erCabrio-producedBy-BMW, CopaMundial-producedBy-Adidas, PumaShadow-producedBy-Puma, MercurialVapor-producedBy-Nike, Integrity07Plus-producedBy-Reebok Ducati848-producedBy-Ducati, KTM50SX-producedBy-KTM, myVWGolfGTI-producedBy-VWWolfsburg, myFiatPuntoEvo-producedBy-FiatTurin, myMiniCooperS-producedBy-MiniCowley, myBMW3erCabrio-producedBy-BMWMünchen, myCopaMundial-producedBy-AdidasHerzogenaurach, myPumaShadow-producedBy-PumaNürnberg, myMercurialVapor-producedBy-NikeBremen, myDucati848-producedBy-DucatiBorgoPanigale, myKTM50SX-producedBy-KTMMattighofen

Tabelle 4: M-Objects, M-Levels und M-Realtionships des erweiterten Beispiels

4.3 Reasoning Ergebnisse

In diesem Punkt wird getestet, ob die Reasoner die umgesetzten Assertionen und Integrity Constraints wie gewünscht interpretieren, oder ob sie doch voneinander abweichen. Zuerst erfolgt ein Test, ob die Reasoner die transitive Hülle von *concretize* in Bezug auf die M-Objects bzw. M-Relationships richtig ableiten. Anschließend werden die Reasoner im Hinblick darauf getestet, ob sie erkennen, wenn einer der formulierten ICs verletzt wird. Abschließend werden noch Tests in Bezug auf die Vererbung von Eigenschaftswerten, der Unique Name Assumption und den Disjoint der M-Levels durchgeführt. Die Tests beziehen sich dabei allesamt auf die in Tabelle 2 enthaltenen M-Objects, M-Relationships und M-Levels.

4.3.1 Test der Transitivität von concretize

Wie bereits bekannt ist, kann ein Objekt auf mehreren Abstraktionsebenen konkretisiert und beschrieben werden, wodurch sich eine Konkretisierungshierarchie ergibt. Die transitive Eigenschaft von *concretize* (*concretize_t*) umfasst dabei alle Objekte, welche durch ein bestimmtes Objekt direkt oder indirekt konkretisiert werden. Besonders gut geeignet für den Test der Transitivität sind die beiden M-Objects *myCopyOfHP4* und *myPorsche911CarreraS* bzw. die M-Relationship *myPorsche911CarreraS-producedBy-PorscheZuffenhausen*, da diese auf der untersten Ebene liegen und somit die meisten Objekte direkt oder indirekt konkretisieren. Als Ergebnis müssten die Reasoner hier folgende M-Objects bzw. M-Relationships zurückliefern:

myCopyOfHP4	{Product, Book, HarryPotter4}
myPorsche911CarreraS	{Product, Car, Porsche911, Porsche911CarreraS}
myPorsche911CarreraS-producedBy-PorscheZuffenhausen	{Product-producedBy-Company, Car-producedBy-CarManufacturer, Porsche911CarreraS-producedBy-PorscheLtd}

Den folgenden Abbildungen können die gelieferten Reasoning Ergebnisse entnommen werden. Es sei vorweg gesagt, dass alle dieselben erwarteten M-Objects und M-Relationships ermitteln konnten, daher werden die Ergebnisse hier auch nicht für jeden Reasoner einzeln angeführt. Dieser Test wurde somit von jedem Reasoner bestanden. Wie Sie beispielsweise in Abbildung 42 auf der rechten Seite sehen können, konkretisiert das M-Object *myPorsche911CarreraS* direkt das M-Object *Porsche911CarreraS*, während es indirekt die M-Objects *Porsche911*, *Car* und *Product* konkretisiert.

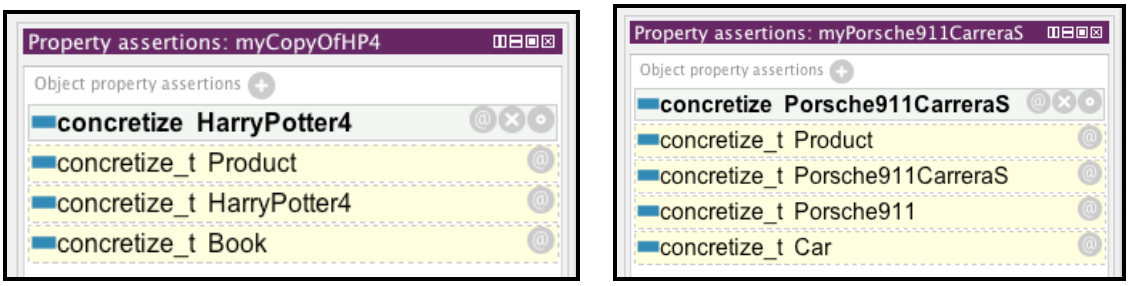

Abbildung 42: Ergebnis des Tests der Transitivität (M-Objects)

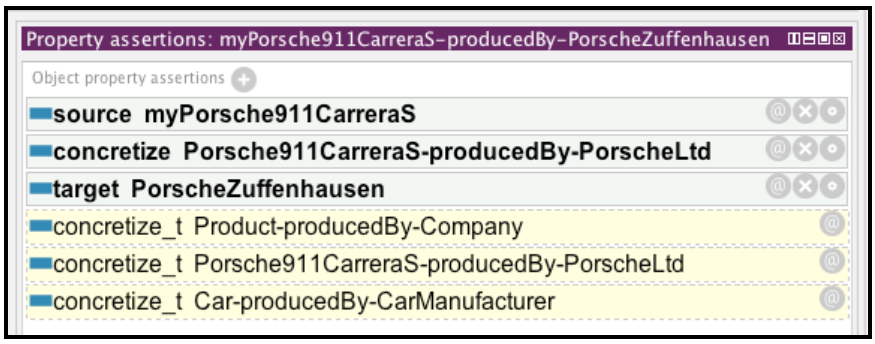

Abbildung 43: Ergebnis des Tests der Transitivität (M-Relationships)

4.3.2 Test der Integrity Constraints

Nun geht es an das Testen der in Kapitel 3 realisierten Integrity Constraints. Starten werde ich mit jenem aus Zeile 9 (Attributschema). Wie bereits bekannt ist soll dieser IC sicherstellen, dass auf einem bestimmten Level eines M-Objects eine Dateneigenschaft nur genau einen definierten Datentyp und nur genau einen Wert haben darf. Als Beispiel wird hier beim M-Object *Car* auf dem Level *Category* die *taxrate* auf Double geändert (der IC verlangt Integer). Anschließend wird versucht *taxrate* ein zweites Mal hinzuzufügen, diesmal jedoch mit einem anderen Wert. Die folgenden beiden Abbildungen zeigen die von den Reasonern gelieferten Ergebnsse:

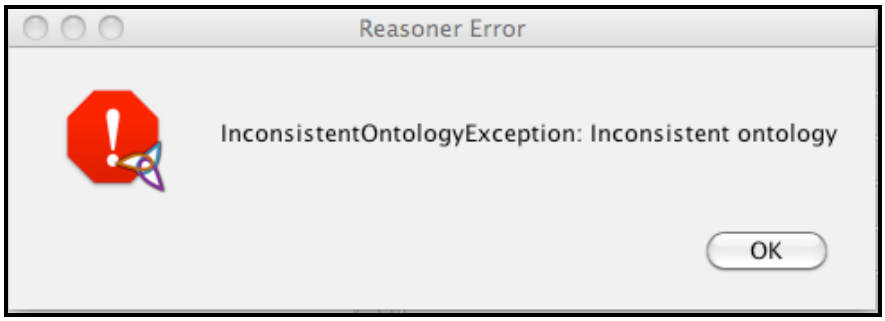

Abbildung 44: Fehlermeldung von Fact++ und HermiT

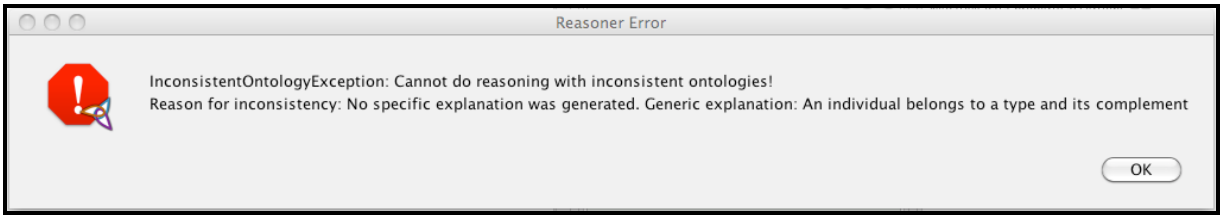

Abbildung 45: Fehlermeldung von Pellet

Wie in den beiden Abbildungen zu sehen ist, erkennen alle drei Reasoner, dass die Ontologie aufgrund der Änderungen inkonsistent geworden ist. Sie haben also den entsprechenden IC richtig interpretiert. Sie unterscheiden sich lediglich dahingehend, dass die Fehlermeldung von Pellet im Vergleich zu Fact++ und HermiT etwas anders aussieht. Das Ergebnis ist in jedem Fall bei allen drei das gleiche, wodurch sie alle diesen Test bestanden haben.

Als nächstes erfolgt der Test des IC aus Zeile 12, welcher die Level-Hierarchie festlegt. So wird beispielsweise festgelegt, dass der Level *Category* des M-Objects *Product* den übergeordneten Level von *Model* darstellt und *Model* wiederum den übergeordneten Level von *Physical Entity*. Würde nun also z.B. das Modell dahingehend abgeändert werden, dass *myCopyOfHP4* nicht mehr *HarryPotter4* sondern *Book* direkt konkretisiert und *HarryPotter4* nun eine direkte Konkretisierung von *myCopyOfHP4* darstellt, dann müssten die Reasoner die Verletzung des ICs erkennen, da sich die Level-Hierarchie nun dahingehend abgeändert hat, dass *Physical Entity* nun keinen Sub-Level von *Model* mehr darstellt, sondern umgekehrt.

Wie bereits eingangs zur Diplomarbeit erwähnt, sind die Reasoner im Moment noch nicht in der Lage, Integrity Constraints von Standardaxiomen zu unterscheiden und interpretieren sie daher unter Open World Assumption. Daher gehen sie davon aus, obwohl aufgrund der Änderung *Physical Entity* nun keine Sub-Ebene von *Model* mehr darstellt, dass jedes M-Object mit dem Level *Physical Entity* ein M-Object mit dem Level *Model* konkretisiert, auch wenn das nicht der Fall ist. Daher konnten alle drei Reasoner den Umstand auch nicht erkennen, dass die Ontologie aufgrund der Änderung nun inkonsistent geworden ist und den Test daher auch nicht bestehen. Also konnte mit dem Test des IC aus Zeile 14 (Festlegen erlaubter Level-Eigenschaften) fortgefahren werden.

Dieser IC stellt sicher, dass eine bestimmte Dateneigenschaft nur auf einem bestimmten Level eines M-Objects (oder dessen Konkretisierungen) definiert werden darf. Nehmen Sie als Beispiel die Dateneigenschaft *taxRate* des M-Objects *Product*. Durch den IC wurde festgelegt, dass *taxRate* auf dem Level *Category* definiert werden muss, d.h. dass die Reasoner eine Verletzung des ICs feststellen müssten, wenn *taxRate* beispielsweise auf der *Model*-Ebene definiert wird. Zu diesem Zweck

habe ich auf der *Category*-Ebene bei den beiden M-Objects *Car* und *Book* (welche beide Konkretisierungen von *Product* darstellen) die *taxRate* entfernt und stattdessen auf der *Model*-Ebene bei den beiden M-Objects *HarryPotter4* und *Porsche911CarreraS* mit dem Wert 20 hinzugefügt. Die dadurch resultierende Verletzung dieses IC konnten alle drei Reasoner erkennen, was wiederum mit der Fehlermeldung aus Abbildung 44 bzw. Abbildung 45 zum Ausdruck gebracht wird. Dieser IC wurde somit ebenfalls von allen drei Reasonern richtig interpretiert.

Hier wird aber bereits die große Schwäche der Reasoner offensichtlich. Obwohl sie korrekterweise die Fehlermeldung geben, dass die Ontologie inkonsistent geworden ist, geben sie keinen Hinweis darauf (oder zumindest nur sehr mangelhaft), warum das so ist. In diesem Fall war es jetzt klar, warum sie inkonsistent geworden ist, weil ich nach jeder Änderung ein Reasoning durchgeführt habe. Würde man jedoch mehrere Änderungen am Modell durchführen und anschließend ein Reasoning durchführen, würde es durchaus mühsam werden, den Fehler zu erkennen und zu korrigieren, was alles andere als ein angenehmes Arbeiten mit Protégé OWL ermöglicht.

Wie ebenfalls bereits bekannt ist stellt der IC aus Zeile 16 sicher, dass ein bestimmter Level zu einem bestimmten M-Object (und dessen Konkretisierungen) gehört. Daraus kann also geschlossen werden, dass es die Reasoner erkennen müssten, wenn beispielsweise auf dem Level *Category* des M-Objects *Product* ein neues M-Object namens *PC* eingeführt wird, dass es sich dabei um eine Konkretisierung von *Product* handeln muss, ohne zuvor angegeben zu haben, dass *PC* eine Konkretisierung von *Product* darstellt. In Abbildung 46 sehen Sie, wie die Reasoner diesen Fall interpretieren.

Alle Reasoner kamen zum richtigen Ergebnis, nämlich dass *PC* eine Konkretisierung von *Product* darstellt, obwohl dies zuvor nicht explizit angegeben wurde. Daher kann auch für diesen Test festgehalten werden, dass er von allen drei Reasonern bestanden wurde. Somit waren die Tests der Integrity Constraints die M-Objects betreffend abgeschlossen, womit die Tests der ICs die M-Relationships betreffend gestartet werden konnten.

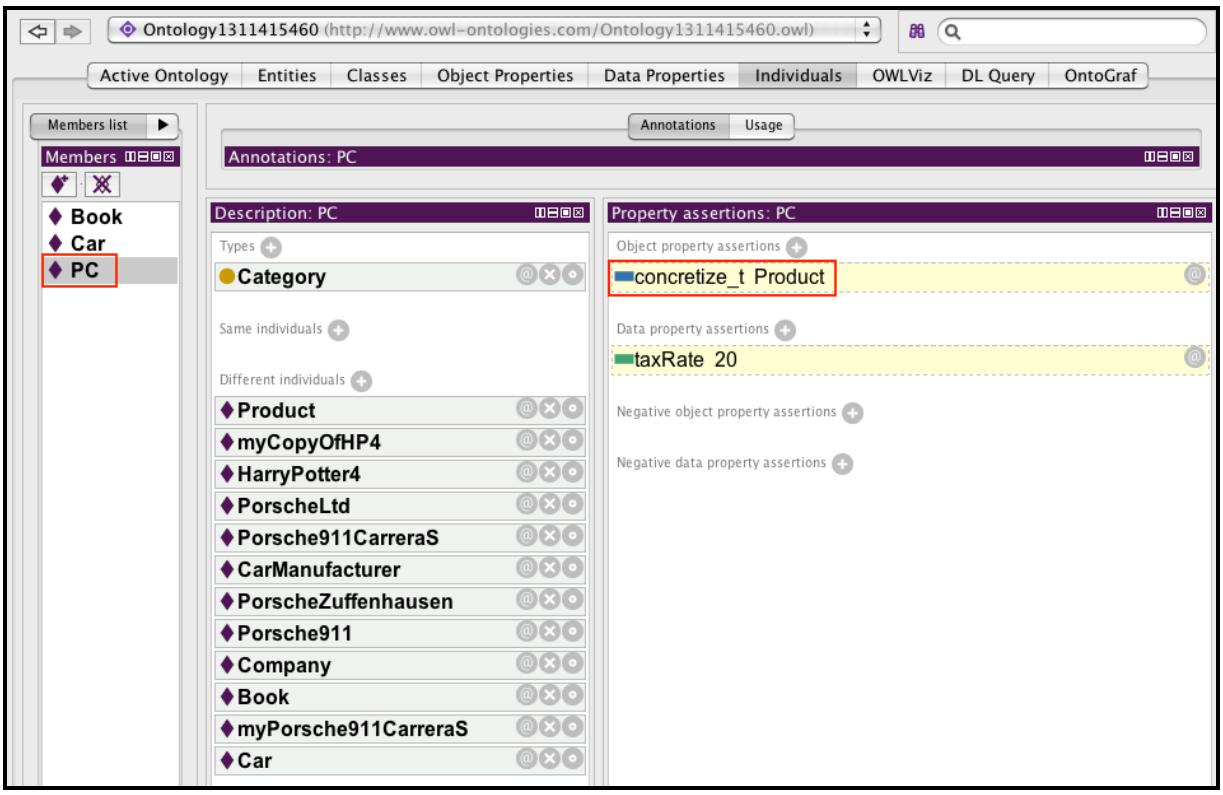

Abbildung 46: Testergebnis IC Zeile 16

Die Tests der Integrity Constraints aus den Zeilen 24 und 26 stellten sich ähnlich schwierig dar wie jener Test des ICs aus Zeile 12. Wobei die Ursache dafür wieder darin zu finden ist, dass die ICs von den Reasonern nicht als solche interpretiert werden konnten, sondern unter Open World Assumption als Standardaxiome behandelt wurden. Daher war es den Reasonern auch nicht möglich, eventuelle Verletzungen dieser ICs richtig zu interpretieren, wodurch diese Tests erwartungsgemäß von keinem der drei Reasoner bestanden werden konnten.

4.3.3 Test der Vererbung von Eigenschaftswerten

In Zeile 10 des Mapping Algorithmus wird mittels Assertion festgelegt, dass Konkretisierungen eines M-Objects den Wert einer Dateneigenschaft erben, wenn diese Dateneigenschaft auf einem Level des übergeordneten M-Objects definiert wurde, der nicht dem Top-Level entspricht. Wenn also nun beispielsweise beim M-Object *Product* auf dem Level *Category* die Eigenschaft *taxRate* mit dem Wert 20 belegt wird, dann müssen die beiden Konkretisierungen *Book* und *Car* diesen Wert erben. Das heißt also mit anderen Worten, dass alle Konkretisierungen von *Product* den Wert 20 für *taxRate* übernehmen müssen. Würde nun *taxRate* beispielsweise bei *Book* von

20 auf den Wert 25 erhöht werden, müssten die Reasoner erkennen, dass eine Verletzung der Ontologie vorliegt.

Nachdem ich die eben erwähnten Schritte durchgeführt hatte, kamen alle Reasoner zu demselben richtigen Ergebnis, nämlich dass es sich nun um eine inkonsistente Ontologie handelt (siehe Abbildung 44 und Abbildung 45), womit dieser Test von allen bestanden wurde.

4.3.4 Test der Disjunktheit von M-Levels

In Zeile 17 des Mapping Algorithmus werden alle Levels zu allen anderen Levels disjunkt gesetzt, was bedeutet, dass es kein M-Object gibt, dass Mitglied von mehr als einem Level ist. Wie in Abbildung 33 zu sehen ist, sind *Category*, *Model*, *Enterprise*, *Brand*, *IndustrialSector*, *PhysicalEntity*, *Factory* und *Root* die Disjoint-Klassen (Disjoint-Levels) des Levels *Catalog*. Würde man nun versuchen, die beiden Klassen *Catalog* und *Category* als äquivalente Levels (Klassen) zu verwenden, müssten die Reasoner erkennen, dass es sich bei den beiden um unterschiedliche Levels handelt und daher eine Verletzung vorliegt. Um diesen Umstand zu testen habe ich dem Level *Catalog Category* als äquivalente Klasse hinzugefügt. Während Fact++ und HermiT wiederum dieselbe Fehlermeldung erzeugten, wie bereits in den Punkten zuvor, wurde bei Pellet der Fehlermeldung folgender Hinweis hinzugefügt:

> *Reason for inconsistency: Individual http://www.owl-ontologies.com/Ontology1296769430.owl#Book is forced to belong to class http://www.owl-ontologies.com/Ontology1296769430.owl#Category and its complement*

Wie Sie dem Hinweis entnehmen können, ist diese Fehlermeldung um einiges aussagekräftiger als jene der beiden anderen Reasoner. Es wird sofort ersichtlich, dass die Ontologie aufgrund dessen verletzt ist, weil das M-Object *Book* eindeutig zum Level *Category* gehören muss, was durch die hinzugefügte Äquivalenz von *Catalog* und *Category* nun nicht mehr gegeben ist. Obwohl alle Reasoner die Inkonsistenz der Ontologie erkennen konnten, weißt Pellet hier aufgrund der zusätzlichen Information einen kleinen Pluspunkt auf.

4.3.5 Test der Unique Name Assumption

In den Zeilen 18 bzw. 27 des Mapping Algorithmus erfolgt die Sicherstellung der Unique Name Assumption. Durch diese soll sichergestellt werden, dass sich jedes M-Object bzw. jede M-Relationship (diese werden beide, wie bereits bekannt ist, in Protégé OWL als Individuen dargestellt) von allen anderen unterscheidet. Die Unique Name Assumption verhält sich also analog zur Disjunktheit der einzelnen Levels. Würde man nun also versuchen, dem M-Object *Car Book* als gleiches M-Object bzw. der M-Relationship *Car-producedBy-CarManufacturer Porsche911CarreraS-producedBy-PorscheLtd* als gleiche M-Relationship hinzuzufügen, müssten die Reasoner die Inkonsistenz der Ontologie erkennen. Diesen Sachverhalt konnten auch alle Reasoner richtig interpretieren, wobei Pellet diesmal eine gute Erklärung für die inkonsistente Ontologie geben konnte:

> *Reason for inconsistency: No specific explanation was generated. Generic explanation: An individual is sameAs and differentFrom another individual at the same time*

Aufgrund dieser Erklärung, die Pellet hier versucht zu geben, wird ersichtlich, dass versucht wurde, zwei Individuen gleichzeitig unterschiedlich und gleich zu behandeln, was zumindest für die Suche nach der entsprechenden Verletzung eine Hilfe darstellt. Andererseits wäre es auch hier nicht schlecht zu wissen, um welche Individuen es sich dabei konkret handelt. Im Vergleich zu Fact++ und HermiT versucht Pellet hier zumindest überhaupt einen Hinweis auf den Fehler zur Verfügung zu stellen, was wiederum einen kleinen Pluspunkt für diesen Reasoner darstellt.

4.3.6 Resümee

Leider konnte sich aufgrund der Reasoning Ergebnisse keiner der getesteten Reasoner als sonderlich geeignet oder ungeeignet herauskristallisieren, da sie alle zu denselben Ergebnissen gelangten. Lediglich Pellet konnte sich einen ganz kleinen Vorteil verschaffen, indem zumindest versucht wurde eine Erklärung für eine inkonsistente Ontologie zu geben. Aber das alleine reicht nicht aus, um Pellet nun als geeignetsten Reasoner im Bereich der M-Objects und M-Relationships vorzuschlagen.

Es kann jedoch festgehalten werden, obwohl diese Reasoner im Moment noch nicht in der Lage sind Integrity Constraints richtig zu interpretieren, dass sie trotz Open World Assumption die Hälfte der ICs (3 von 6) richtig interpretieren konnten. Hier bleibt abzuwarten ob der in [Moti07] entwickelte Ansatz auch noch in einem Reasoner umgesetzt wird. In der folgenden Tabelle werden abschließend noch einmal die gelieferten Reasoning Ergebnisse (in Bezug auf erfolgreich bzw. nicht erfolgreich) von Fact++, Pellet und HermiT gegenübergestellt, die wie bereits gesagt, leider nicht sehr aufschlussreich in Bezug auf einen geeigneten Reasoner sind.

	Pellet	**Fact++**	**HermiT**
Transitivität von concretize	✔	✔	✔
IC Zeile 9	✔	✔	✔
IC Zeile 12			
IC Zeile 14	✔	✔	✔
IC Zeile 16	✔	✔	✔
IC Zeile 24			
IC Zeile 26			
Vererbung von Eigenschaftswerten	✔	✔	✔
Disjunktheit von M-Levels	✔	✔	✔
Unique Name Assumption	✔	✔	✔

Tabelle 5: Gegenüberstellung der Reasoning Ergebnisse

4.4 Gegenüberstellung des Laufzeitverhaltens

Da alle Reasoner OWL richtig interpretieren konnten und es daher in Punkt 4.3 nicht gelungen ist, einen geeigneten für die Thematik der M-Objects und M-Relationships zu ermitteln, erhoffte ich mir zumindest aufgrund des Antwortzeitverhaltens der einzelnen Reasoner in Bezug auf eine unterschiedliche Anzahl von Individuen eine vor-

sichtige Aussage darüber treffen zu können, welcher sich am ehesten für diese Thematik eignen würde. Das Ergebnis kann der folgenden Tabelle entnommen werden, wobei ich zunächst die Antwortzeit bei 16 Individuen (Beispiel aus [Neum09b], M-Objects und M-Relationships zusammengefasst) gemessen habe und die Anzahl anschließend auf 35, 66, 100 bzw. 140 Individuen erhöht habe. Um welche Individuen es sich dabei konkret handelt, wurde bereits in Punkt 4.2 angeführt.

	Pellet	**Fact++**	**HermiT**
12 M-Objects 4 M-Relationships	751 ms	46 ms	141 ms
25 M-Objects 10 M-Relationships	1045 ms	62 ms	327 ms
50 M-Objects 16 M-Relationships	2371 ms	125 ms	562 ms
75 M-Objects 25 M-Relaionships	3269 ms	140 ms	1217 ms
100 M-Objects 40 M-Relationships	4025 ms	390 ms	2496 ms

Tabelle 6: Laufzeitverhalten der Reasoner bei zunehmender Individuenanzahl

Wie zu sehen ist, konnte sich hier Fact++ im Vergleich zu HermiT und Pellet deutlich durchsetzen. Während der Unterschied zu HermiT zu Beginn noch ziemlich gering ist (95 Millisekunden bei 16 Individuen), wächst die Differenz zwischen den beiden im Laufe der Zeit auf ein Vielfaches an (2106 Millisekunden bei 140 Individuen).

Pellet ist hier stark abgeschlagen. Beträgt der Unterschied zu Beginn bei 16 Individuen bereits 610 Millisekunden im Vergleich zu HermiT, so liegt die Differenz zu Fact++ hier sogar bereits bei 705 Millisekunden. Am Ende antwortet Pellet sogar um 3635 Millisekunden langsamer als Fact++, wobei der Unterschied im Vergleich zu HermiT hier mit 1529 Millisekunden schon fast als gering zu bezeichnen ist. Das liegt auch daran, dass sich die Antwortzeiten von HermiT bei jeder Erhöhung der Anzahl der Individuen verdoppelt, während sich bei Pellet die Antwortzeiten stetig zwischen

einer halben Sekunde und Sekunde verlängern. In Abbildung 47 ist gut ersichtlich, wie sich die beiden Kurven von Pellet und HermiT nach und nach annähern, während die Kurve von Fact++ noch weit darunter liegt.

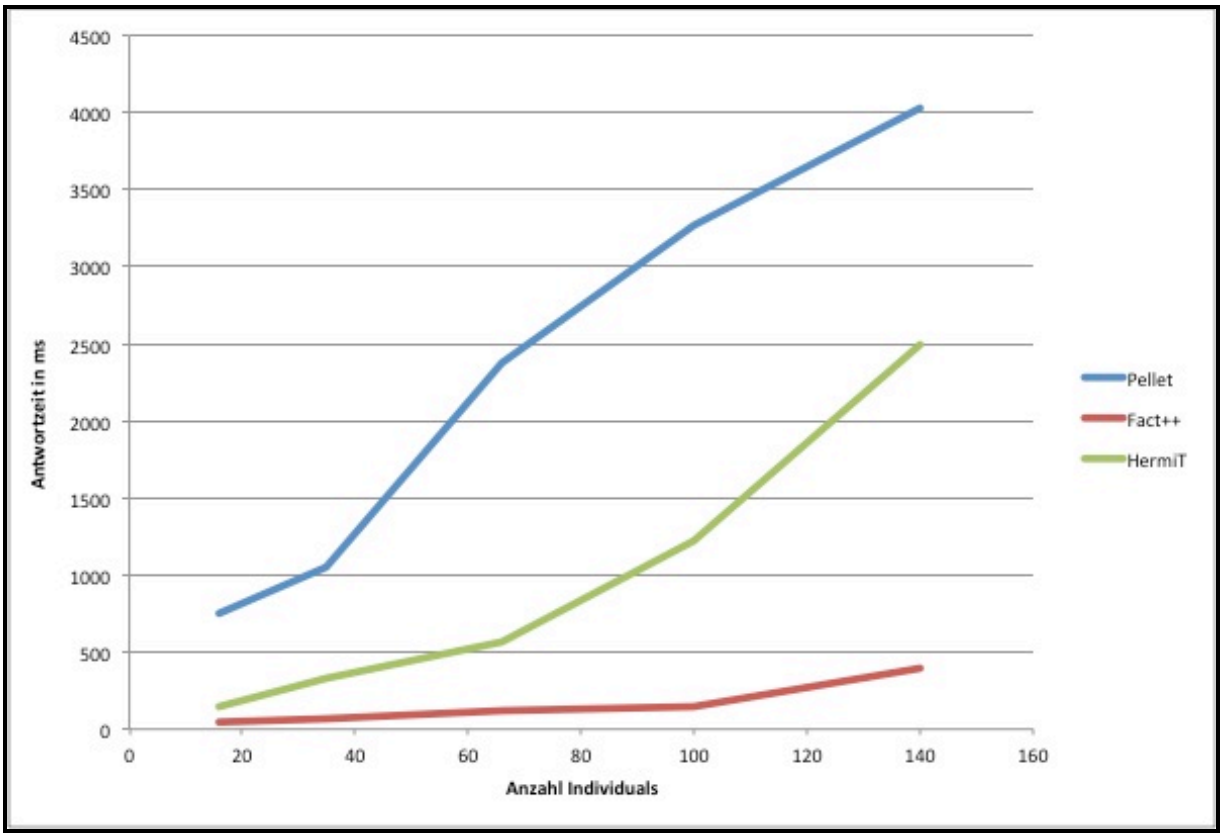

Abbildung 47: Antwortzeitverhalten einzelner Reasoner

Würde man nun aus den eben gewonnenen Erkenntnissen auf einen geeigneten Reasoner schließen wollen, würde die Wahl bestimmt auf Fact++ fallen. Diese Aussage wäre jedoch im Hinblick auf die Reasoning Ergebnisse verfälscht und ist daher mit großem Vorbehalt zu betrachten. Interessant ist in diesem Zusammenhang noch die Tatsache, dass HermiT mit den Antwortzeiten von Fact++ nicht mithalten konnte, wird von diesem Reasoner doch behauptet (jedoch von der Information Systems Group der Universität Oxford selbst), dass er in sehr vielen Fällen schneller sei als alle anderen Reasoner.

5 Zusammenfassung

In dieser Arbeit wurde eine kurze Einführung in die zentralen Bestandteile von Descripton Logics gegeben und auf die unterschiedlichen Versionen der Webontologiesprache OWL eingegangen. Ein weiterer Punkt beschäftigte sich damit, wie Integrity Constraints in OWL abgebildet werden können. M-Objects und M-Relationships wurden in den Grundlagen intensiv behandelt und deren Mapping nach OWL veranschaulicht und erläutert. Auch Protégé wurde genauer betrachtet, vor allem die zentralen Bestandteile von Protégé OWL und Protégé Frames. Wie M-Objects und M-Relationships in Protégé Frames abgebildet werden können, wurde ebenfalls in den Grundlagen gezeigt.

Im Zentrum dieser Arbeit stand die Realisierung des Mapping Algorithmus aus [Neum09b]. Es wurde gezeigt, wie die einzelnen Zeilen dieses Algorithmus umgesetzt und in das zuvor erstellte Export-Plugin eingebettet wurden. Es wurde anhand von Screenshots gezeigt, wie sich die M-Objects, M-Levels, M-Relationships und OWL Objekt- und Dateneigenschaften nach durchgeführtem Mapping in Protégé OWL darstellten. Die im Mapping Algorithmus enthaltenen Integrity Constraints wurden dabei als Standardaxiome (General Class Axioms) umgesetzt und mittels Annotation Property als ICs gekennzeichnet.

In den anschließenden Reasoning – Performance Studies wurden die Reasoner Pellet, Fact++ und HermiT in Bezug auf die von ihnen abgeleiteten Ergebnisse und ihr Antwortzeitverhalten bei wachsender Anzahl von Individuen betrachtet. Da diese Reasoner zurzeit noch nicht in der Lage sind, zwischen standardmäßigen Axiomen und jenen Axiomen zu unterscheiden, welche als Integrity Constraints interpretiert werden sollen, schlug ein Teil dieser Tests fehl. Auch die von den Reasonern erzeugten Fehlermeldungen bei erkannten Constraint-Verletzungen waren zum Großteil unbrauchbar. In diesem Zusammenhang bleibt abzuwarten, ob um ICs erweitertes OWL von OWL-Reasonern künftig unterstützt wird. Hinsichtlich des Antwortzeitverhaltens konnte sich jedenfalls Fact++ deutlich gegenüber HermiT und Pellet durchsetzen.

Literaturverzeichnis

[Amat07] d'Amato, C., Fanizzi, N.: *Ontologies: An Introduction.* Università degli Studi di Bari - Dipartimento di Informatica. Bari, Italien (2007)

[Berg08] Berger, S., Eichinger, C.: *Ontologien am Beispiel von Description Logics (OWL-DL)*. Johannes Kepler Universität Linz – Institut für Wirtschaftsinformatik – Data & Knowledge Engineering. Linz, Österreich (2008)

[Clar09a] Clark&Parsia: *Pellet: OWL 2 Reasoner for Java.* http://clarkparsia.com/pellet/ 22. April 2010

[Clar09b] Clark&Parsia: *Pellet Features.* http://clarkparsia.com/pellet/features 22. April 2010

[Diwo11] Diwold, A.: *Multi-Level-Modellierung: Implementierung eines Protégé-Plugins (Arbeitstitel).* Johannes Kepler Universität Linz – Institut für Wirtschaftsinformatik – Data & Knowledge Engineering. Linz, Österreich (2011)

[Ecli10] Eclipse: *About the Eclipse Foundation.* http://www.eclipse.org/org/ 22. April 2010

[Fact07] The University of Manchester: *Fact++.* http://owl.man.ac.uk/factplusplus/ 22. April 2010

[Fact09] The University of Manchester: *Fact++.* http://owl.cs.manchester.ac.uk/fact++/ 22. April 2010

[Herm10] Information Systems Group: *Hermit OWL Reasoner – The new kid on the OWL block.* http://hermit-reasoner.com/ 22. April 2010

[Horr05a] Horrocks, I.: *Ontology Reasoning: the Why and the How*. Presentation at Workshop in honor of Frank Wolter. Liverpool, England (2005)

[Horr05b] Horridge, M.: *The Manchester OWL Syntax Editor Guide*. The University of Manchester – Department of Computer Science. Manchester, England (2005)

[Horr07a] Horridge, M., Glimm, B., Sattler, U.: *Hybrid Logics and Ontology Languages*. Electronic Notes in Theoretical Computer Science (ENTCS), Volume 174. Amsterdam, Niederlande. (2007)

[Horr07b] Horridge, M., Jupp, S., Moulton, G., Rector, A., Stevens, R., Wroe, C.: *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools (Edition 1.1)*. The University Of Manchester – Department of Computer Science. Manchester, England. (2007)

[Knub04a] Knublauch, H., Fergerson, R.W., Noy, N.F., Musen, M.A.: *The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications.* Proceedings of the 3rd International Semantic Web Conference (ISWC 2004). Hiroshima, Japan. (2004)

[Knub04b] Knublauch, H., Dameron, O., Musen, M.A.: *Weaving the Biomedical Semantic Web with the Protégé OWL Plugin.* Proceedings of the 1st International Workshop on Formal Biomedical Knowledge Representation (KR-MED 2004). Whistler, Kanada. (2004)

[Knub04c] Knublauch, H., Musen, M.A.: *Editing Description Logic Ontologies with the Protégé OWL Plugin.* Proceedings of the 2004 International Workshop on Description Logics (DL2004). Whistler, Kanada. (2004)

[Moti07] Motik, B., Horrocks, I., Sattler, U.: *Bridging the gap between OWL and relational databases.* Proceedings of the 16th International World Wide Web Conference (WWW 2007). Calgary, Kanada (2007)

[Nard03] Nardi, D., Brachman, R.J.: *An Introduction to Description Logics.* In: The Description Logic Handbook. Baader et al. Cambridge University Press. Cambridge, England (2003)

[Neum09a] Neumayr, B., Grün, K., Schrefl, M.: *Multi-level domain modeling with m-objects and m-relationships.* Proceedings of the 6th Asia-Pacific Conference on Conceptual Modeling (APCCM 2009). Wellington, Neuseeland (2009)

[Neum09b] Neumayr, B., Schrefl, M.: *Multi-Level Conceptual Modeling and OWL.* Proceedings of the Joint International Workshop on Metamodels, Ontologies, Semantic Technologies and Information Systems for the Semantic Web (MOST-ONISW 2009). Gramado, Brasilien (2009)

[Neum10a] Neumayr, B., Schrefl, M., Thalheim, B.: *Hetero-Homogeneous Hierarchies in Data Warehouses.* Proceedings of the 7th Asia-Pacific Conference on Conceptual Modelling (APCCM 2010). Brisbane, Australien (2010)

[Neum10b] Neumayr, B.: *Multi-Level Modeling with M-Objects und M-Relationships.* Johannes Kepler Universität Linz – Institut für Wirtschaftsinformatik – Data & Knowledge Engineering. Linz, Österreich (2010)

[Neum11] Neumayr, B., Schrefl, M., Thalheim, B.: *Modeling Techniques for Multi-Level Abstraction.* In: Delcambre, L., Kaschek, R., eds.: The Evolution of Conceptual Modeling. Lecture Notes in Computer Science (LNCS) Volume 6520. Springer Verlag. Berlin, Deutschland (2011)

[Noy00] Noy, N.F., Fergerson, R.W., Musen, M.A.: *The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility.* Proceedings of the 2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW 2000). Juan-les-Pins, Frankreich (2000)

[Prot09] Stanford Center for Biomedical Informatics Research: *Choosing between Versions of Protégé.* http://protegewiki.stanford.edu/index.php/Protege4Migration 13. April 2010

[Prot10a] Stanford Center for Biomedical Informatics Research: *What is Protégé?* http://protege.stanford.edu/overview/ 12. April 2010

[Prot10b] Stanford Center for Biomedical Informatics Research: *An introduction to developing plug-ins.* http://protege.stanford.edu/doc/pdk/plugins/overview.html 28. April 2010

[Prot10c] Stanford Center for Biomedical Informatics Research: *How to write an export plug-in.* http://protege.stanford.edu/doc/pdk/plugins/export_plugin.html 28. April 2010

[Prot11] Stanford Center for Biomedical Informatics Research: *What is Protégé-OWL?* http://protege.stanford.edu/overview/protege-owl.html 24. Juli 2011

[Siri07] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: *Pellet: A practical OWL-DL reasoner*. Journal of Web Semantics (JWS) Volume 5. Amsterdam, Niederlande (2007)

[W3C09a] W3C OWL Working Group: *OWL 2 Web Ontology Language: Document Overview.* W3C Recommendation. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/ 10. Februar 2011

[W3C09b] W3C OWL Working Group: *OWL 2 Web Ontology Language: New Features and Rationale*. W3C Recommendation. http://www.w3.org/TR/2009/REC-owl2-new-features-20091027/ 10. Februar 2011

[W3C09c] W3C OWL Working Group: *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax.* W3C Recommendation. http://www.w3.org/TR/owl2-syntax/ 10. Februar 2011

A Anhang

A.1 Verwendete Programmversionen

Im Rahmen dieser Arbeit kamen folgende Programmversionen zum Einsatz:

- Protégé 3.4.1
- Protégé 4.1 Beta
- Pellet 2.1.2
- HermiT 1.3.3
- Fact++ 1.5.1
- Java Runtime Environment 1.6.0_07
- Entwicklungsumgebung Eclipse 3.3.2

A.1.1 Protégé 3.4.1

Unter http://protege.stanford.edu/download/registered.html können Protégé 3.4.1 sowie die Nachfolgeversion 4.1 Beta bezogen werden. Unter derselben Adresse steht auch der Source Code der einzelnen Versionen zur Verfügung. Da in Punkt 2.6 bereits auf die zentralen Bestandteile von Protégé Frames (Klassen, Instanzen, Slots, Facets, Formulare / Masken) und Protégé OWL (Individuen, Eigenschaften, Klassen) eingegangen wurde, wird in diesem Punkt lediglich auf die wesentlichen Eigenschaften von Protégé 3.4.1 eingegangen, zu sehen in Tabelle 7 auf der nächsten Seite.

Eigenschaften	
Frames	Unterstützung durch den Protégé Frames Editor
OWL	Unterstützung: • OWL 1 • OWL und RDF(S) • Zugriff auf OWL und RDF(S) Files mittels der Protégé OWL API • SPARQL • SWRL mittels SWRLTab • Metamodellierung (erlaubt OWL full) • Pellet Reasoner (direkte Verbindung) • DIG Reasoner (Verbindung via http DIG Interface) • Konfigurationsspeicherung in Protégé Project Files • OWL Imports mittels Repository-Mechanismen
Plugins	Plugin-Framework für Slot-Widgets, Tab-Widgets, Projekte, Backends, Imports und Exports Plugins werden nicht nur von Stanford selbst, sondern auch von der großen Protégé Gemeinschaft entwickelt, daher stehen bereits jede Menge unterschiedlicher Plugins zur Verfügung.
User Interface	• mittels Tab- und Slot-Widgets bzw. durch Zugriff auf das Metamodell konfigurierbar • mittels des „fill-in-the-blank-Stils“ können den Eigenschaften von Individuen Werte zugewiesen werden
Multi-user	Mittels der Client-Server Version von Protégé wird das Bearbeiten ein und derselben Ontologie durch mehrere Benutzer unterstützt.
Datenbank Speichermodell	Mittels des JDBC Datenbank Backends können Ontologien in einer Datenbank abgelegt werden.

Tabelle 7: Eigenschaften von Protégé 3.4.1 [Prot09]

A.1.2 Protégé 4.1 Beta

Ich werde in diesem Punkt lediglich auf die Eigenschaften dieser Version bzw. ihre Unterschiede im Vergleich zu Protégé 3.4.1 eingehen. Diese können der folgenden Tabelle entnommen werden:

Eigenschaften	
Frames	Derzeit noch keine Unterstützung von Protégé Frames
OWL	Unterstützung: • OWL 2 • lediglich OWL-Framework • Zugriff auf OWL Files mittels der Manchester OWL API • derzeit noch kein SPARQL Support • SWRL mittels Basiseditor und Pellet als Reasoner • OWL full • Fact++, HermiT, Pellet und zahlreiche andere Reasoner (direkte Verbindung) • globale Konfiguration • OWL Imports entweder aus User-Repositories oder dem Web
Plugins	Hier gilt das gleiche wie für Protégé 3.4.1, mit dem Unterschied, dass das Plugin-Framework nun auf OSGi Technologie basiert.
User Interface	• mittels Plugins konfigurierbar • den Eigenschaften von Individuen werden Werte in Form von Axiomen zugewiesen • Menü und „Drag and Drop“ Elemente
Multi-user	Derzeit noch keine Unterstützung
Datenbank Speichermodell	Derzeit noch keine Unterstützung

Tabelle 8: Eigenschaften von Protégé 4.1 Beta [Prot09]

A.1.3 Pellet 2.1.2

Reasoner wie Pellet stellen eine Kernkomponente von auf Ontologien basierenden Datenmanagementanwendungen dar. [Clar09a] Pellet wurde in Java implementiert, ist Open Source und war der erste OWL Reasoner der OWL DL vollständig unterstützte. [Siri07] Diverse Erweiterungen folgten, sodass die aktuellste Version mittlerweile OWL 2 unterstützt.

Pellet stellt folgende Features zur Verfügung: [Clar09b]

- Standard Reasoning Services
- Multiple Interfaces des Reasoners
- Verbindendes SPARQL-DL Query Answering
- Datentyp-Reasoning
- Unterstützung der SWRL Regeln
- Analyse und Verbesserung von Ontologien
- Ontologie-Debugging
- Inkrementelles Reasoning

Während Pellet in Protégé 3.4.1 bereits standardmäßig vorinstalliert ist, muss es in Protégé 4.1 Beta zusätzlich installiert werden, was jedoch aufgrund der verfügbaren Update-Funktion von Protégé einfach durchgeführt werden kann.

A.1.4 Fact++ 1.5.1

Der zweite Reasoner, den ich in meiner Arbeit verwendet habe ist Fact++ in der Version 1.5.1, welcher eine Erweiterung des OWL DL Reasoners Fact darstellt. Er verwendet zwar die gleichen Algorithmen wie sein Vorgänger, jedoch hat er eine andere interne Architektur beziehungsweise wurde er um einige Features erweitert. Fact++ wurde in C++ umgesetzt um einerseits die Portabilität zu maximieren und andererseits ein möglichst effizientes Software-Tool zur Verfügung stellen zu können. [Fact07]

Bei Fact++ handelt es sich ebenfalls um eine Open Source Software die unter der GPL/LGPL Lizenz verbreitet wird. Ursprünglich entwickelt wurde er von Dmitry Tsarkov und Ian Horrocks im Zuge des WonderWeb Projekts. Er ist ein „tableaux"-basierter Reasoner, der sowohl OWL 1 als auch OWL 2 unterstützt. Er kann sowohl als Standalone-DIG-Reasoner als auch als Back-end-Reasoner für OWL API basierende Applikationen verwendet werden. Während Fact++ in der 3.4.1 Version von Protégé noch zusätzlich installiert werden musste, wird er in der 4.1 Betaversion bereits als standardmäßiger Reasoner mitgeliefert. [Fact09]

A.1.5 HermiT 1.3.3

Der dritte und damit letzte von mir verwendete Reasoner ist HermiT, der ebenfalls OWL als Sprache verwendet und ebenfalls die beiden Versionen 1 und 2 unterstützt. Das Programm wurde von der Information Systems Group der Universität Oxford entwickelt und unter der LGPL veröffentlicht. HermiT 1.3.3 stellt dabei die derzeit aktuellste Version dar. Die Version ist mit Java 1.5 kompatibel, verwendet die OWL API 3 und ist ebenfalls Open Source. [Herm10]

Wird dem Programm ein OWL File zur Interpretierung übergeben ist es beispielsweise in der Lage die Ontologie auf ihre Konsistenz zu prüfen beziehungsweise Klassenbeziehungen zu erkennen. Laut Entwickler stellt dieser Reasoner eine absolute Neuheit dar, da er aufgrund des verwendeten Algorithmus imstande ist eine Vielzahl unterschiedlicher Ontologien binnen kürzester Zeit zu klassifizieren. Außerdem verwendet er so genanntes „core blocking". Damit wird erreicht, dass die vom Reasoner benötigte Speicherkapazität auf das nötigste reduziert wird. HermiT ist in der Betaversion von Protégé 4.1 bereits standardmäßig implementiert. [Herm10]

A.1.6 Entwicklungsumgebung Eclipse 3.3.2

„Eclipse ist eine offene Entwicklungsplattform, welche erweiterbare Frameworks, Werkzeuge und Laufzeitumgebungen für die Entwicklung, Verteilung und Verwaltung von Software über den gesamten Lebenszyklus hinweg, umfasst." [Ecli10]

Die Entwicklungsumgebung Eclipse wurde ursprünglich von Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft und Webgain im November 2001 ins Leben gerufen. Im Februar 2004 fand eine Neustrukturierung statt. Seit dem handelt es sich beim Unternehmen Eclipse um eine unabhängige Non-Profit-Organisation mit einer großen Open Source Community im Hintergrund. Dieser Gemeinschaft werden von Eclipse grundsätzlich folgende vier Leistungen angeboten: [Ecli10]

- **IT Infrastruktur:** Sie wird von der Eclipse Unternehmung für die Open Source Gemeinschaft verwaltet und beinhaltet Webseiten, Downloadseiten, Newsgruppen, CVS/SVN, Bugzilla Datenbanken und vieles mehr.

- **IP Management:** Eclipse ermöglicht es Softwareherstellern aufgrund der Open Source Technologie ihre eigenen kommerziellen Softwareprodukte herzustellen. Dies ist möglich, da alle Eclipse Projekte unter der EPL lizenziert sind.

- **Entwicklungsunterstützung:** Eclipse unterstützt ihre Anwender dabei, qualitativ hochwertige Software zu entwickeln, indem es dabei hilft, den Eclipse Entwicklungsprozess zu implementieren. Dieser Prozess umfasst unter anderem Reviews, um die Interaktion zwischen den Projekten beziehungsweise der gesamten Community möglichst konsistent zu halten.

- **Ecosystem Entwicklung:** Ermöglicht es der Open Source Community beispielsweise andere auf Eclipse basierende Open Source Projekte beziehungsweise auf Eclipse basierende kommerzielle Produkte einzubinden.

Eclipse unterstützt unter Anderen folgende Programmiersprachen:

- Java / Java EE
- C / C++
- PHP