Computerlinguistik & Künstliche Intelligenz

„Technologien und Konzepte des Semantic Web“

Magisterarbeit
zur Erlangung des
Magister Artium

im Fachbereich
Sprach- und Literaturwissenschaften
der Universität Osnabrück

vorgelegt von: Marc Schilling
aus Osnabrück
am 03.05.2005

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Auf den zur Zeit rund 230 Millionen Internet-Hosts [Internet Domain Survey, 2004] befindet sich eine riesige Ansammlung an Informationen. Das ständig wachsende World Wide Web überlässt die inhaltliche Erschließung relevanter Dokumente der informationssuchenden Person. Die heutige HTML-basierte Hypertextstruktur ist nicht darauf ausgerichtet, Informationen bereitzuhalten, die von Computern "verstanden" und intelligent verarbeitet werden können.
Mit dieser Problematik beschäftigt sich der „WWW-Erfinder“ Tim Berners-Lee in seiner Vision vom Semantic Web. Das Web der nächsten Generation soll Daten mit semantischen Informationen anreichern und es Programmen erlauben, die intendierte Bedeutung der Informationen zu erfassen.

1.2 Zielsetzung und Vorgehen

Im Rahmen dieser Magisterarbeit werden die Technologien und Konzepte erläutert, welche die Basis für das Semantic Web bilden. Dazu wird im folgenden Kapitel zunächst das von Berners-Lee formulierte Beispielszenario der Vision geschildert. In Kapitel 1.4 wird der Weg zum Semantic Web vor dem Entstehungshintergrund des WWW beschrieben, die pragmatische Zielsetzung des Semantic Web benannt und anhand des Schichtenmodells werden die konkreten Technologien zur Realisierung diskutiert. Kapitel 2 führt die grundlegenden Begriffe Wissensrepräsentation, Semantische Netze, Metadaten und Ontologien im Kontext des Semantic Web ein. Einen Schwerpunkt dieser Arbeit bilden die vom W3C standardisierten Sprachen zur Repräsentation von Ontologien. Aus diesem Grund widmet sich Kapitel 3 einer ausführlichen Betrachtung der Sprachen RDF/S und OWL.

Im praktischen Teil der Arbeit wird in Kapitel 4 auf der Grundlage der erläuterten Technologien und unter Verwendung der Repräsentationssprache OWL untersucht, mit welchen semantischen Informationen eine postalische Anschrift

angereichert werden kann. Dazu wird ein Webformular entwickelt in das Adressdaten eingegeben werden können und durch ein Java-Servlet die Ontologie „Postalische Anschrift“ in Form einer Datei generiert. Das Ziel der prototypischen Implementierung besteht auf der einen Seite darin, die Beziehungen und Konzepte zwischen Adressdaten einer Anschrift durch eine Ontologie zu repräsentieren und auf der anderen Seite zu erörtern, welche Informationen sich durch ein Inferenzsystem aus der erzeugten Ontologie ableiten lassen.

1.3 Beispielszenario der Vision des Semantic Web

„Das Unterhaltungssystem spielt den Beatles-Song „ We Can Work It Out“, als das Telefon klingelt. Sobald Pete antwortet, wird die Lautstärke des Unterhaltungssystems verringert, indem das Telefon eine entsprechende Nachricht sendet. Am anderen Ende der Leitung ist seine Schwester Lucy, die ihm aus der Hausarztpraxis mitteilt, dass ihre Mutter einen Facharzt aufsuchen muss, um dort in mehreren physiotherapeutischen Sitzungen behandelt zu werden.

Um für die Mutter den Fahrdienst zum Facharzt mit Pete abzustimmen, instruiert Lucy den Semantic Web Agenten ihres Handheld. Dieser nimmt Kontakt mit dem Agenten des Hausarztes auf und erhält alle für die Behandlung relevanten Informationen. Nun ist Lucy’s Agent in der Lage alle, von der Krankenkasse zugelassenen Physiotherapiepraxen in einem Umkreis von 20 km herauszusuchen, die von einem vertrauenswürdigen Bewertungssystem mit exzellent oder sehr gut bewertet worden sind. Die Semantik der zu bewertenden Schlüsselwörter stellt das Semantic Web dem Agenten zur Verfügung. Mögliche Behandlungstermine werden über die Webseiten der Physiotherapeuten mit den Terminkalendern von Pete und Lucy abgeglichen.

Nach einigen Minuten präsentiert der Agent den beiden einen fertigen Plan. Da Pete diesen Behandlungsort zur Hauptverkehrszeit nur schwer erreichen kann, beauftragt er seinen Agenten mit einer restriktiveren Suche in Bezug auf Ort und Zeit. Unter Berücksichtigung dieser Constraints wird eine für alle Seiten akzeptable Lösung gefunden.“

Dieses von Tim Berners-Lee entwickelte Beispielszenario [Berners-Lee, 2001] ist mit den heutigen Techniken des World Wide Web nicht zu realisieren. Der Grund liegt im Design heutiger Webseiten, die Inhalte lediglich in einer für den Menschen lesbaren Form anzeigen. Es fehlt eine Struktur, welche die Bedeutung der Informationen definiert. Das Semantic Web erweitert das bestehende Web um diese Strukturinformationen, welche dann von Softwareagenten verarbeitet werden sollen.

1.4 Der Weg zum Semantic Web

1.4.1 Der ursprüngliche Entwurf des World Wide Web

Die Idee des World Wide Web wurde in den achtziger Jahren von Tim Berners-Lee am CERN-Forschungszentrum in Genf entwickelt und in „Information Management: A Proposal“ [Berners-Lee, 1989] beschrieben. Kernproblematik dieses Artikels ist der permanente Informationsverlust durch die Fluktuation von Mitarbeitern. Dadurch wurde die Frage aufgeworfen, wie das Informationssystem des Großforschungszentrums organisiert werden könnte, um diesen Verlust zu vermeiden. Nach Ansicht von Tim Berners-Lee kann ein hierarchisch organisiertes System das kaum leisten. Als Beispiel führt er das UUCP (Unix to Unix Copy)-News-System an: *„Typically, a discussion under one newsgroup will develope into a different topic, at which point it ought to be in a different part of the tree.”* Benötigt wird also ein Verweis, mit dem der Übergang zu diesem „*different part of the tree“*-Knoten möglich ist. Der Begriff für einen Verweis namens Hypertext wurde bereits in den fünfziger Jahren von Ted Nelson geprägt und beschreibt durch Menschen lesbare Informationen, die in nicht beschränkter Weise verknüpft sind.

Um Informationen wiederzufinden, deren Ablageort man nicht kennt, wird mit Hilfe von Schlüsselwörtern gesucht. Dieses Vorgehen wird von Tim Berners-Lee kritisch kommentiert: *„The usual problem with keywords, however, is that two people never chose the same keywords. The keywords then become useful only to people who already know the application well.”*

Die geschilderten Überlegungen münden in der Formulierung folgender Designprinzipien für das WWW:

- *Remote access across networks*: In einem verteilten System wie dem CERN ist ein Remote-Zugang über ein Client/Server-Modell essentiell.
- *Heterogenity*: Auf gespeicherte Daten muss von unterschiedlichen Systemen aus zugegriffen werden können.
- *Non-Centralisation*: Eine zentrale Kontrolle würde die angestrebte Verbreitung behindern.
- *Access to existing data*: Die Integration in bestehende Informationssysteme erhöht die Akzeptanz und Verbreitung.
- *Private links*: Es soll die Möglichkeit bestehen, Knoten und Verweise mit privaten Kommentaren zu versehen.
- *Data analysis*: Basierend auf Hypertextdaten soll mit Hilfe von typisierten Knoten und Verweisen eine Datenanalyse möglich sein.
- Live links: Hypertext-Verweise sollen grundsätzlich aktuelle Dokumente anzeigen.

1989 schlug Tim Berners-Lee das Hypertext Projekt vor. Aufgrund von Systemunabhängigkeit und Quelltextoffenheit entschied er sich für das Internetprotokoll TCP/IP als technische Basis und implementierte 1990 den ersten WWW-Server (httpd). Unter Verwendung des von ihm entwickelten Hypertext Transfer Protokolls (http) schrieb er den ersten WWW-Client. Dieser "what-you-see-is-what-you-get" Browser war auch als Editor konzipiert und sollte jeden User in die Lage versetzen, bestehende Dokumente mit Bemerkungen und zusätzlichen Links zu versehen. Dieser Kernforderung des „collaborative hypertext“ wurde im Laufe der Entwicklung nicht Rechnung getragen, da sich die Client-Entwicklung auf die Browserfunktionalität konzentrierte.
Im Dezember 1990 begann Tim Berners-Lee mit der Entwicklung der Auszeichnungssprache Hypertext Markup Language (HTML). In der ersten Beschreibung von HTML [Berners-Lee, 1990] wurden ausschließlich logische Elemente zur Auszeichnung von Text verwendet. Logische Textauszeichnungen haben Bedeutungen (z.B. „betont“) und der Browser entscheidet, wie

sie dargestellt werden (z.B. fett oder kursiv). Die HTML-Spezifikation 1.0 aus dem Jahr 1993 ließ zusätzlich physikalische Elemente zu. Physikalische Textauszeichnungen schreiben dem Browser direkt die Formatierung vor.
Die Vermengung von logischem und physikalischem Markup stellt eine Verletzung der von Tim Berners-Lee geforderten Trennung von Struktur und Darstellung dar.
Das WWW als universelles Informationssystem sollte keiner starren Hierarchie folgen, sondern vielmehr als System von Knoten und Verweisen verstanden werden. Typisierte Knoten repräsentieren Objekte jeglicher Art, während typisierte Verweise Beziehungen zwischen diesen abbilden. Tim Berners-Lee schlug in seinen ursprünglichen Überlegungen zum HTML-Design folgende Typisierungen vor.

Typisierte Knoten:

- People, Groups of people, Projects
- Software-, hardware objects
- Concepts
- Documents, Keywords

Strukturelle Verweise zwischen zwei Knoten:

- A depends on B
- A is part of B
- A made B
- A refers to B
- A uses B
- A is an example of B

Semantische Verweise:

- A is interested in B
- A approves B
- A refutes B

Die Verwendung semantischer Verweise beschreibt eine Art semantisches Netzwerk auf Hypertextbasis. Steht beispielsweise Knoten A für eine Person und Knoten B für ein Projekt, könnte die Relation „ A is interested in B“ dazu führen, dass die Person automatisch über Änderungen des Projektes informiert werden möchte.

Diese Art von semantischen Verweisen sind in der heutigen HTML-Spezifikation [W3C-HTML 4.01 Specification, 1999] nicht vorgesehen. Die Möglichkeit zur Typisierung beschränkt sich auf die Angaben einiger logischer Textauszeichnungen [Münz, 2001] von denen einige kurz erläutern werden.

```
<a href="../../../index.htm" rev="contents">Inhalt</a>
<a href="../naechste_datei.htm" rel="next">weiter</a>
<a href="../vorherige_datei.htm" rel="prev">zur&uuml;ck</a>
<a href="../index.htm" rev="chapter">zum Kapitelanfang</a>
```

Das Attribut *rel*= bestimmt die logische Vorwärtsbeziehung zum Verweisziel, *rev*= die logische Rückwertsbeziehung (*rel*= relation = Bezug, *rev*= reverse = Umkehr).

- *rel="contents"* bedeutet: Verweis zum Inhaltsverzeichnis
- *rev="next"* bedeutet: Rückverweis zur nächsten Datei
- *rel="prev"* bedeutet: Verweis zur vorherigen Datei
- *rev="chapter"* bedeutet: Rückverweis zum Kapitel

- *rel/rev="bookmark"* bedeutet: Verweis zu einem allgemeinen Orientierungspunkt
- *rel/rev="alternate"* bedeutet: Verweis zu einer Datei mit dem gleichen Inhalt wie dem der aktuellen, jedoch in einer anderen Dokumentversion

Betrachtet man die Entwicklung des World Wide Web, so kristallisieren sich drei Punkte heraus, die mit den Forderungen von Tim Berners-Lee beim Entwurf von HTML nicht in Einklang zu bringen sind:

- Die Forderung, vorhandene Fremddokumente mit Bemerkungen und zusätzlichen Links zu versehen („collaborative hypertext“), wurde nicht erfüllt.
- HTML ist keine reine Strukturbeschreibungssprache, da durch Einführung physikalischer Markup-Elemente, die Trennung von Form und Inhalt bzw. Darstellung und Struktur aufgehoben worden ist.
- Eine Typisierung von Knoten und Verweisen ist nicht möglich.

Insbesondere die fehlende Typisierung ist verantwortlich dafür, dass sich das Auffinden relevanter Informationen im heutigen WWW schwierig gestaltet.

1.4.2 Information Retrieval im heutigen WWW

Die derzeit zur Verfügung stehenden Suchmaschinen [Search Engine Watch, 2004] basieren im wesentlichen auf Methoden zur Volltextsuche. Eine Volltextsuchmaschine besteht aus Datensammlern, Robots oder Agenten genannt, Indizierungs- und Abfrageprogramm. Die ermittelten Daten werden vom Indizierer weiterverarbeitet und dann wird abhängig von den in den Webseiten enthaltenen Schlüsselwörtern ein Volltextindex aufgebaut. Ein Rankingalgorithmus priorisiert die indizierten Seiten, setzt sie also in Bezug zu den Suchbegriffen.
Die Ursache unzureichender Suchergebnisse liegt darin begründet, dass die in natürlicher Sprache vorliegenden Information im WWW zwar maschinenlesbar, aber nicht maschinenverständlich sind.

Unter dem Konzept der Maschinenverständlichkeit versteht [Berners-Lee, 1998]:
„The concept of machine-understandable documents does not imply some magical artificial intelligence which allows machines to comprehend human mumblings. It only indicates a machine's ability to solve a well-defined problem by performing well-defined operations on existing well-defined data. Instead of asking machines to understand people's language, it involves asking people to make the extra effort“

Ziel des Semantic Web ist es also nicht, die natürliche Sprache zu verstehen, sondern bestehende Webseiten mit Hilfe von Typisierungen, einheitlichen Beschreibungsmodellen und standardisierten Sprachen zu erweitern. Der enorme Mehraufwand bei der Bereitstellung wohldefinierter Daten soll durch Softwarewerkzeuge verringert werden. Gelingt dies, ergeben sich daraus neue Möglichkeiten der Informationsverarbeitung:

- Intelligente kontextsensitive Suchmaschinen statt Schlüsselwort-Suchmaschinen.
- Beantwortung von Fragen, statt Abfrage von Informationen.
- Verbesserter Austausch von Informationen und deren Bedeutung.
- An individuelle Anforderungen angepasste Informationen.

1.4.3 Die Architektur des Semantic Web

Die Architektur des Semantic Web wird im Artikel “Semantic Web Road map” [Berners-Lee, 1998a] beschrieben und basiert auf einem Schichtenmodell.

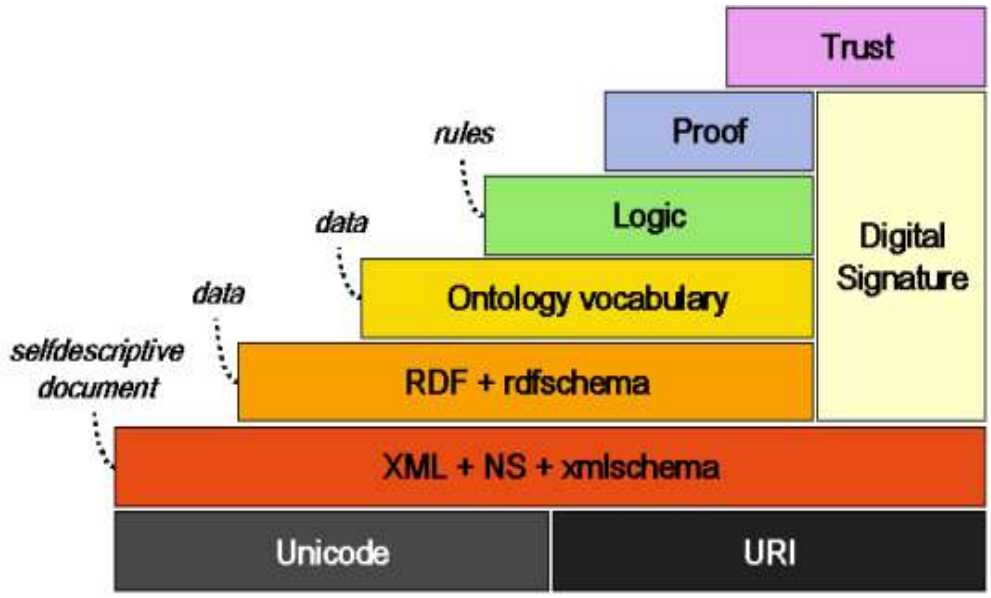


Abbildung 1: Schichtenmodell des Semantic Web, nach Berners-Lee and Eric Miller, entnommen [Koivunen, 2001]

Abbildung 1 zeigt die Technologien, welche zur Realisierung des Semantic Web eingesetzt werden sollen. Die Kommentare auf der linken Seite des Schichtenmodells geben die Funktionalität der einzelnen Ebenen an. Durch die

Realisation jeder einzelnen Schicht soll ein Mehrwert geschaffen werden und so eine inkrementelle Entwicklung unterstützt werden. Koordiniert wird diese Entwicklung durch die Arbeitsgruppe „Semantic Web Activity“ [W3C-Semantic Web Activity, 2001] des „World Wide Web Consortiums“ (W3C), dessen Direktor Tim Berners-Lee ist.

Auf der ersten Ebene des Schichtenmodells befindet sich der standardisierte Unicode-Zeichensatz [UNICODE]. Im Gegensatz zum älteren 8-Bit ASCII-Code, benötigt Unicode für die Darstellung eines Zeichens zwei Byte und kann dadurch fast alle Schriftsprachen der Welt mittels eines einzigen Zeichensatzes darstellen.

Die eindeutige Identifikation von Ressourcen wird mit Hilfe des Uniform Resource Identifier [W3C-URI, 2001] erreicht. Als Obermenge des bekannten Uniform Recource Locator erweitert er dessen Verweismöglichkeiten, um Begriffe aus der realen Welt (z.B. die ISBN-Nummer eines Buches) und um abstrakte Konzepte.

Durch den Einsatz der standardisierten Extensible Markup Language (XML) und deren Erweiterungen XML Schema und XML Namespaces soll die syntaktische Interoperabilität für das Semantic Web sichergestellt werden. Unter Interoperabilität versteht man die Fähigkeit unabhängiger, heterogener Systeme, möglichst nahtlos zusammen zu arbeiten, um Informationen auf effiziente und (wieder)verwertbare Art und Weise auszutauschen [net-lexikon, 2004].Die Aufgabe der höheren Ebenen des Schichtenmodells besteht darin, die semantische Interoperabilität für ein verteiltes heterogenes System, wie das Semantic Web, zur Verfügung stellen.

Auf der dritten Ebene wird dazu das Resource Description Framework (RDF) und die dazugehörige Schemasprache (RDFS) eingeführt. RDF stellt einen Repräsentationsformalismus für Metadaten zur Verfügung und legt damit fest, in welcher Form Aussagen über Ressourcen dargestellt werden. Die Form orientiert sich dabei an der Subjekt, Prädikat, Objekt Notation der natürlichen Sprache, wobei *S* die Eigenschaft *P* mit dem Wert *O* hat. Auch bei Subjekt, Prädikat und Objekt handelt es sich um Ressourcen, welche generell eindeutig per URI identifiziert sind. RDFS bietet die Möglichkeit Taxonomien aufzubauen und einfache semantische Restriktionen über Ressourcen festzulegen. Mit Ontologien wird auf der vierten Ebene das Ziel verfolgt, ein gemeinsames, wieder-

verwendbares und maschinenverständliches Vokabular zur Beschreibung von Ressourcen, sowie deren Beziehungen (Hierarchien, Eigenschaften und Relationen) untereinander, zu definieren.
Zur Beschreibung von Ontologien hat das W3C die Web Ontology Language (OWL) standardisiert. OWL setzt auf RDF/S auf und bietet die Möglichkeit logische Regeln zu formulieren, welche zusätzlich maschinell interpretierbare Beschreibungen von Ressourcen liefern. Die für eine maschinelle Interpretation nötigen Inferenzmechanismen sollen auf der Logik-Ebene dem Semantic Web hinzugefügt werden und nach [Studer et al., 2003] folgende Aufgabe erfüllen:
„By applying logical deduction, one can infer new knowledge from the information which is stated explicitly."
Auf der ontologischen Ebene müssen Ressourcen danach so eindeutig (formal) beschrieben werden, dass „neues Wissen" durch die Anwendung logischer Regeln abgeleitet werden kann. Wird beispielsweise ausgesagt, dass die Ressource *Person A* mit der Ressource *Person B* kooperiert und das die Relation *kooperiert* symmetrisch ist, können Inferenzmechanismen daraus die Aussage ableiten, dass auch *Person B* mit *Person A* kooperiert, obwohl dies lediglich von Person A spezifiziert wurde. Neben der Ableitung „neuen Wissens" besteht die Möglichkeit, die logische Konsistenz von Ontologien zu überprüfen. Grundlage dafür ist, dass die Ontologiesprache neben einer wohldefinierten Syntax eine auf Logik basierende formale Semantik zur Verfügung stellt, die es erlaubt effiziente Inferenzmechanismen einzusetzen.

Für die Ebenen Logik, Proof und Trust existieren zum jetzigen Zeitpunkt noch keine Standards, sondern lediglich grundsätzliche Überlegungen. Ein viel versprechender Ansatz zur formalen Spezifikation von Ontologien besteht in der Verwendung von OWL in der Variante DL (Description Logics). Dabei wird OWL auf Beschreibungslogiken abgebildet. Unter Beschreibungslogiken versteht man eine Familie von logikbasierten Wissensrepräsentationssprachen, welche seit Jahren intensiv erforscht werden. Beschreibungslogiken sind eine Teilmenge der Prädikatenlogik erster Stufe bei denen die Ausdruckskraft zugunsten effizienter Inferenzmechanismen eingeschränkt ist. Sie haben die Eigenschaft vollständig berechenbar und entscheidbar zu sein, was bedeutet,

das alles, was logisch folgt, auch in endlicher Zeit berechnet werden kann. Diese positive Eigenschaften haben dazu geführt, das innerhalb der letzten Dekade zahlreiche Inferenzsystem [DL Implementation Group] auf Basis von Beschreibungslogiken implementiert wurden. OWL DL wurde unter dem Gesichtspunkt entwickelt, die Möglichkeiten dieser Inferenzsysteme zu nutzen, also „neues Wissen“ (im Sinne des oben angeführten Beispiels), aufgrund von Regeln abzuleiten zu können.

In diesem Zusammenhang muss betont werden, dass durch das Semantic Web kein globales Inferenzsystem für das Web entwickelt werden soll. Dies wäre in einem verteilten heterogenen System, in dem jeder Aussagen und Regeln über Ressourcen formulieren kann, aufgrund der von Inferenzsystemen geforderten logischen Konsistenz, kaum zu realisieren.

Die grundsätzliche Überlegung besteht darin, auf der Logik-Ebene eine einheitliche Sprache zu entwickeln, welche in der Lage ist, die auf der ontologischen Ebene beschriebenen Aussagen und Regeln so zu repräsentieren, dass sie auf die internen Repräsentationen von Inferenzsystemen abgebildet werden können. [Berners-Lee, 2001a] bezeichnet diese Sprache als *“(...) Semantic Web's unifying language (the language that expresses logical inferences made using rules and information such as those specified by ontologies).”*

Hinter den Ebenen Proof und Trust steht die Überlegung, dass es für das Semantic Web notwendig ist, Aussagen zu verifizierten. Einerseits könnte eine Aussage als wahr angesehen werden, wenn sichergestellt ist, dass sie aus einer vertrauenswürdigen Quelle stammt. Dazu bietet sich der Einsatz von Public-Key-Kryptographie an, um die Herkunft der Aussagen und Regeln mit Hilfe digitaler Signaturen zu identifizieren. Auf dieser Basis soll ein dezentrales Netz von transitiven Vertrauensbeziehungen aufgebaut werde. Die Idee hinter dem „Web of trust“ ist: Wenn Person A den Aussagen von Person B vertraut und B wiederum Person C, dann sollte auch A in gewissem Grad C vertrauen. Diese Vertrauensbeziehungen könnten von Personen auf ihre Softwareagenten übertragen werden.

Werden Aussagen durch Inferenzsysteme abgeleitet, soll die Vertrauenswürdigkeit dadurch erreicht werden, dass Inferenzsysteme neben den neu abgeleiteten Aussagen zusätzliche Informationen über die Ableitung ausgeben, z.B. welche Regel angewendet wurden oder auf welchen Aussagen die Ablei-

tung basiert. Diese Informationen werden Usern oder zukünftigen Softwareagenten in der “unifying language” mit dem Ziel präsentiert, zu entscheiden, ob diesem “Beweis” zu trauen ist oder nicht. Wird ein “Beweis” mit einer eindeutigen URI versehen und digital signiert, könnte dies eine erneute Ableitung überflüssig machen.

Vergleicht man die im Kapitel 1.3 beschriebene Vision des Semantic Web mit der zum jetzigen Zeitpunkt standardisierten Sprache OWL, wird deutlich, dass noch reichlich Forschungs- und Entwicklungsarbeit zu leisten ist, bis Ansätze in dieser Vision die Realität umgesetzt werden können.

2 Grundlagen

Die von Tim Berners-Lee entwickelte Vision des Semantic Web hat zu einer Renaissance der Diskussion über Konzepte und Technologien geführt, die als Grundlagen für das Web der nächsten Generation angesehen werden können. In diesem Kapitel werden die Grundlagen vorgestellt, wobei mit den Unterschieden zwischen traditionellen, typischerweise zentralisierten, Wissensrepräsentationssystemen und einem verteilten System wie dem Semantic Web begonnen wird. Das Ziel von Berners-Lee besteht darin, existierende Webseiten um eine semantische Komponente auf Basis von Metadaten zu erweitern. Daher wird in Kapitel 2.2 untersucht, welche Anforderungen Metadaten erfüllen müssen, um diesem Ziel näher zu kommen. Die wörtliche deutsche Übersetzung des Begriffs Semantic Web, als „Semantisches Netz“, weist auf eine Verwandtschaft beider Begriffe hin. Der ursprüngliche Entwurf des World Wide Web (vgl. 1.4.1) betrachtete das WWW als eine Art Semantisches Netz auf Hypertextbasis. Da das Semantic Web als global verteilte Variante eines Semantischen Netzes angesehen werden kann, beschreibt Kapitel 2.3 Semantische Netze. Bei der Repräsentation von Wissen nehmen die abstrakten Konzepte von Ontologien eine zentrale Rolle ein. In Kapitel 2.4 werden zunächst die Formen und Einsatzmöglichkeiten von Ontologien benannt und anschließend die Möglichkeiten der Integration von verteilten Ontologien diskutiert.

2.1 Das Semantic Web als Wissensrepräsentationssystem

Der Terminus Wissensrepräsentation wird in der Künstlichen Intelligenz als die symbolische Darstellung von Wissen über einen Gegenstandsbereich definiert. Ein Wissensrepräsentationssystem benötigt für die Darstellung und Beschreibung von Ressourcen eine formale Struktur mit definierten Regeln innerhalb einer festgelegten Terminologie, also eine Syntax. Innerhalb dieser Struktur wird das Wissen um die Beziehungen zwischen den Ressourcen, also die Semantik, mit Hilfe von Metadaten repräsentiert. Das Semantic Web wird häufig als ein Wissensrepräsentationssystem für das WWW bezeichnet. Im folgenden werden die Eigenschaften traditioneller Wissensrepräsentationssysteme nach [Crawford, 1990] betrachtet und die Unterschiede zum Semantic Web aufgezeigt.

1. eine kompakte Syntax.
2. eine wohldefinierte Semantik, um präzise zu beschreiben, was repräsentiert werden soll.
3. eine ausreichende Ausdruckskraft, um menschliches Wissen zu repräsentieren.
4. es muss ein effizienter und nachvollziehbarer Inferenzmechanismus zum Einsatz kommen können.
5. es muss der Aufbau einer umfangreichen Wissensbasis möglich sein.

Zwischen den Punkten (3.) und (4.) besteht dabei ein Zielkonflikt, da es kaum möglich sein wird, beide Ziele gleichzeitig zu erreichen.
Traditionellen Systemen der Wissensrepräsentation liegt typischerweise eine zentrale Wissensbasis zu Grunde. Die Priorität liegt auf einem effizienten Inferenzmechanismus (4.), weshalb die Ausdruckskraft (3.) eingeschränkt werden muss. Mit dem Semantic Web wird das Designziel der Interoperabilität verfolgt, also der Austausch von Informationen zwischen verteilten Systemen. Um dieses Ziel zu erreichen, liegt die Priorität auf einer möglichst großen Ausdruckskraft (3.) und nicht auf einem (zentralen) effizienten Inferenzmechanismus (4.). In diesem fundamentalen Unterschied sieht Tim Berners-Lee eine gewisse Analogie zur Entwicklung des Hypertextsystems. Hätte man bei HTML

beispielsweise auf die Konsistenz von Links bestanden, wäre das Wachstum des WWW wohl kaum derart exponentiell verlaufen. Da für das Semantic Web eine ähnliche Entwicklung angestrebt wird, soll die Ausdruckskraft möglichst wenig eingeschränkt sein, um eine breite Basis für den Austausch von Informationen zwischen verteilten Systems zu schaffen. Dieses Designziel begründet Berners-Lee mit einem Beispiel menschlicher Kommunikation. Auf die Frage *„What is a hex-head bolt?“* lautet die Antwort *„A hex-head bolt is a type of machine bolt“*. Durch dieses Beispiel soll verdeutlicht werden, dass es eine Vielzahl an Informationen gibt, die durch Klassifizierung mit einer ausdrucksstarken Sprache wie z.B. RDF/S repräsentiert werden können. Trotzdem wird es innerhalb des globalen Semantic Web Subsysteme geben wird, welche die Ausdruckskraft der Repräsentationssprache so einschränken, dass effiziente Inferenzmechanismen eingesetzt werden können.

Nach dem Satz von Gödel, [Wikipedia, 2005] *„Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig.“,* können nicht sämtliche Informationen einer Repräsentation verarbeitet werden. Die Aufgabe von Inferenzmechanismen für das Semantic Web besteht also darin, die Möglichkeiten zur Verarbeitung von Informationen zu erweitern. Ein global konsistentes System kann und soll dadurch nicht verwirklicht werden.

2.2 Metadaten

Im allgemeinen werden Metadaten als Informationen über andere Daten bezeichnet und sollen im WWW zum besseren Auffinden von Dokumenten durch Suchmaschinen beitragen. Dazu werden im Dateikopf eines HTML-Dokuments Metadaten innerhalb des Standalone-Tags *meta* kodiert.

```
<head>
<meta name="author" content="Marc Schilling">
<meta name="keywords" content="Magisterarbeit, Semantic Web">
<meta name="description" content="Kurzbeschreibung des Dokuments.">
<meta name="date" content="2004-11-11T08:12:12+01:00">
</head>
```

Ein Meta-Tag enthält ein Attribut, beispielsweise die Eigenschaft *name=“author“* und dessen Wert *content=“Marc Schilling“*. Weitere verbreitete Attribute

sind Schlüsselwörter, eine Kurzbeschreibung, sowie das Erstellungsdatum. Problematisch ist, dass in der aktuellen HTML-Spezifikation weder mögliche Attribute noch deren Interpretation definiert sind. Daher ist nicht sichergestellt, ob und wie Suchmaschinen die Attribut-Wert-Paare interpretieren und welche Prioritäten sie den Attributen zuweisen. Darüber hinaus lassen sich Metadaten nur für das gesamte Dokument definieren. Es ist also nicht möglich, dass sich Metadaten auf Teile eines Dokumentes beziehen, was ihre Flexibilität einschränkt. Insgesamt führen die geschilderten Probleme dazu, dass der Einsatz von Metadaten wenig effektiv ist.
Im Kontext des Semantic Web werden Metadaten als maschinenverständliche Informationen über Web-Ressourcen oder andere Dinge bezeichnet [Berners-Lee, 1997]. Daraus ergeben sich folgende Anforderungen an Metadaten:

- Maschinenverständlichkeit setzt voraus, dass die Syntax, Semantik und Struktur von Metadaten standardisiert ist.
- Metadaten müssen unabhängig von bestimmten Anwendungsgebieten verarbeitet werden können.
- Metadaten sind selber Ressourcen, damit eindeutig identifizierbar und wiederverwendbar.
- es muss eindeutig sein, auf welche Ressourcen sich Metadaten beziehen.
- Metadaten können in jeder Ressource gespeichert werden, unabhängig davon, auf welche Ressource sie verweisen.
- durch Metadaten müssen auch Relationen zwischen Ressourcen ausgedrückt werden können.
- Metadaten müssen interoperabel sein.
- für Metadaten sollen Klassifizierungsschemata definiert und erweitert werden können.

Die genannten Anforderungen sollen durch das Resource Description Framework (RDF) und die Schemasprache (RDFS) erfüllt werden und dadurch die Grundlage für den Austausch und die maschinelle Verarbeitung von Metadaten gelegt werden.

2.3 Semantische Netze

Zur Darstellung semantischer Wissensbeziehungen empfiehlt sich das von Quillian [Quillian, 1967] eingeführte graphische Assoziationsmodell der Semantischen Netze. Innerhalb eines Semantischen Netzes wird ein Objekt (Konzept oder Begriff der realen Welt) als Knoten in einem gerichteten Graph dargestellt. Im Allgemeinen unterscheidet man zwischen

- generischen Knoten (Klassen)

und

- individuellen Knoten (Instanzen).

Die Beziehungen (binäre Relationen) zwischen den einzelnen Objekten werden über benannte Kanten repräsentiert. Das eigentliche Wissen über ein Objekt ergibt ich aus den Beziehungen zu anderen Objekten.
Der Kantenabstand zwischen verbundenen Objekten kann auf die Eigenschaft der semantischen Nähe hindeuten, wobei zwei benachbarte Objekte eine größere semantische Nähe haben, als wenn diese durch mehrere Zwischenknoten miteinander verbunden sind. Als Maßeinheit für die semantische Nähe kann nach [Collins & Quillian, 1969] die Reaktionszeit dienen, welche Menschen benötigen, um eine Aussage über ein Objekt als wahr zu erkennen. Aus der folgenden Tabelle geht hervor, dass die Objekte Kanarienvogel und Vogel eine größere semantische Nähe (semantic distance) aufweisen, als Kanarienvogel und Tier.

True sentences	Mean reaction time (ms)
A canary is a canary.	1.00
A canary is a bird.	1.17
A canary is an animal.	1.23

Tabelle 1: “Reaction times for sentences with differing semantic distances” [Collins & Quillian, 1969)]

Generell unterscheidet man folgende Kantenarten:

- strukturelle Beziehungen:
  - Die *is-a* Kante steht für eine Teilmengenbeziehung zwischen generischen Objekten und dient der Generalisierung bzw. Spezialisierung.

Mehrere solcher Kanten bilden eine hierarchisch gegliederte Klasseneinteilung mit Vererbungsmechanismen entlang der is-a Kanten und werden als Taxonomie bezeichnet.

- Eine *instance-of* Kante steht für eine Elementbeziehung zwischen individuellem und generischem Objekt, repräsentiert also ein Beispielexemplar einer Klasse.

- nicht strukturelle Beziehungen:
  - beliebige binäre Relationen

Möchte man drei- und mehrstellige Beziehungen abbilden, müssen neue Knoten eingefügt werden, welche die in Beziehung zu setzenden Knoten über geeignete Kanten verbinden oder mehrstellige Beziehungen müssen in mehrere zweistellige transformiert werden.

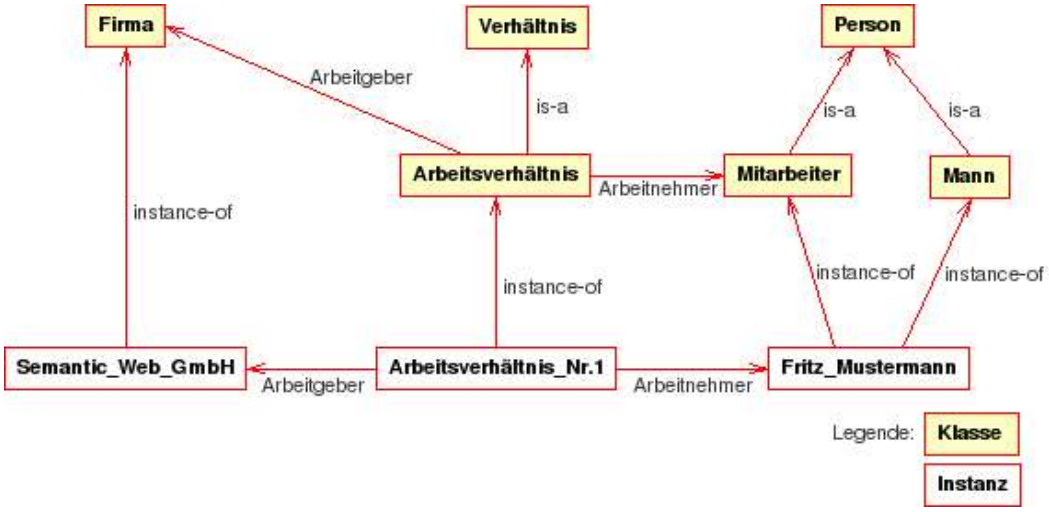
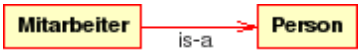


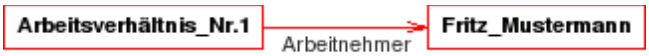
Abbildung 2: Beispiel eines Semantischen Netzes

Die Benennung von Knoten und Kanten mit sinntragenden Begriffen erschließt sich lediglich dem menschlichen Betrachter. Für eine maschinelle Verarbeitung muss die Semantik in einer geeigneten Form definiert werden, beispielsweise in Prädikatenlogik erster Stufe.



$(\forall x)$ [Mitarbeiter(x) $\rightarrow$ Person(x)]

Mitarbeiter(Fritz_Mustermann)



Arbeitnehmer(Arbeitsverhältnis_Nr.1,Fritz_Mustermann)

$(\forall x)\ [\text{Arbeitsverhältnis}(x) \rightarrow (\exists y)\ [\text{Mitarbeiter}(y) \,\&\, \text{Arbeitnehmer}(x,y)]\ ]$

Nach dieser Definition bringt das Semantische Netz den Sachverhalt zum Ausdruck, dass *Fritz_Mustermann*, als Instanz der Klasse Mann, eine Spezialisierung der Klasse Person ist. Es existiert ein A*rbeitsverhältnis_Nr.1* mit dem Arbeitnehmer *Fritz_Mustermann*, der ein Mitarbeiter ist und dem Arbeitgeber S*emantic_Web_GmbH* , die eine Firma ist.

Auch auf nicht explizit modelliertes Wissen kann mit Hilfe so genannter Anfragenetze [Reimer, 1991] inferiert werden. Durch die Transitivität der is-a Kante und die Inferenzregel, das eine zu einer Klasse gehörende Instanz auch zu allen übergeordneten Klassen gehört, kann die im folgenden Anfragenetz formulierte Frage, ob *Fritz_Mustermann* eine Person ist, mit „Ja“ beantwortet werden.
Um die Frage zu beantworten, welche Beziehung zwischen *Fritz_Mustermann* und der S*emantic_Web_GmbH* besteht, kommt das Verfahren der Aktivierungsausbreitung (spreading activation) zum Einsatz. Dabei wird derjenige Knoten aktiviert, der die beiden Knoten in Beziehung setzt.

Aus diesem Anfragenetz ergibt sich die Antwort, dass *Fritz_Mustermann* Arbeitnehmer des A*rbeitsverhältnis_Nr.1* ist und der Arbeitgeber die S*emantic_Web_GmbH*.
Ein Vorteil des graphischen Assoziationsmodells der Semantischen Netze liegt in der Übersichtlichkeit und Flexibilität. Neue Knoten und Kanten können nach

Bedarf definiert und so neues Wissen hinzugefügt werden. Auch auf nicht explizit modelliertes Wissen kann mit Hilfe so genannter Anfragenetze inferiert werden. Vererbungsmechanismen unterstützen einen sparsamen Umgang mit Speicherplatz, da das Wissen nur in der allgemeinsten Klasse gespeichert werden muss. Die Eigenschaft der semantischen Nähe führt zu einer Objektzentrierung, weil Aussagen über ein Konzept in dessen Nähe modelliert werden. Daraus resultiert normalerweise ein geringerer Suchaufwand, als bei logikbasierten Repräsentationsformaten, die dieses Strukturierungsmittel nicht aufweisen.

Prinzipiell ist die Darstellungsmächtigkeit Semantischer Netze mit der der Prädikatenlogik erster Stufe vergleichbar. Durch Partitionierung (vgl. [Reimer, 1991a]) lassen sich regelhafte Zusammenhänge und einschränkende Bedingungen mit Hilfe prädikatenlogischer Formeln repräsentieren. Die Darstellung von Quantoren, Negationen, Disjunktionen und Implikationen als Semantisches Netz erreicht aber schnell eine unübersichtliche Komplexität.

Ein Defizit semantischer Netze ist häufig das Fehlen einer formal exakt definierten Semantik. Die Frage, wie Knoten und Kanten interpretiert werden und welche Inferenzen daraus gezogen werden sollen, bleibt dadurch häufig unbeantwortet. Dies ist jedoch kein grundsätzliches Problem, da man die Semantik formal definieren kann.

2.4 Ontologien

Der Begriff Ontologie stammt ursprünglich aus der Philosophie und ist laut Duden Fremdwörterbuch definiert als: Lehre vom Sein, von den Ordnungs-, Begriffs- und Wesensbestimmungen des Seienden. Seit Anfang der neunziger Jahre sind Ontologien ein populäres Forschungsthema der Künstlichen Intelligenz, welches von der Arbeitsgruppe „Semantic Web Activity“ des W3C aufgegriffen wurde.
Eine der bekanntesten Definition des Begriffs stammt von [Gruber, 1993]):

„*To support the sharing and reuse of formally represented knowledge among AI systems, it is useful to define the common vocabulary in which shared knowledge is represented. A specification of a representational vocabulary for a shared domain of discourse – definitions of classes, relations, functions, and other objects – is called an ontology.*"

- Gemeinsam (*shared*) bedeutet, dass es sich bei einer Ontologie um eine Repräsentation von Wissen einer Domäne handelt, auf die sich eine Benutzergruppe geeinigt hat.
- Der Erstellungsaufwand von Ontologien soll durch Wiederverwendbarkeit (*reuse*) reduziert werden.
- Um die maschinelle Verarbeitung zu ermöglichen, müssen Ontologien formal (*formally*) definiert sein.

Der Grad der Formalisierung reicht von in natürlicher Sprache formulierten Ontologien, bis zu vollständig formal spezifizierten Ontologien. [McGuinness, 2003] illustriert die möglichen Formen von Ontologien in der folgenden Abbildung.

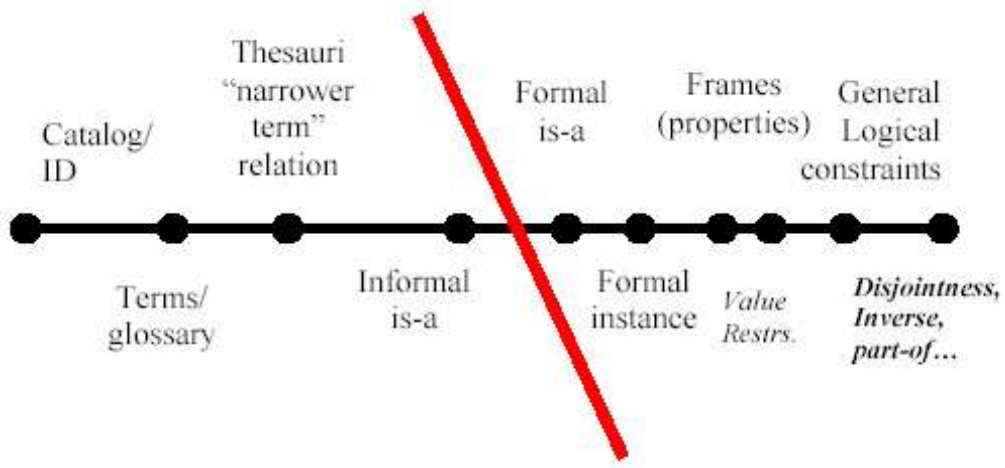


Abbildung 3: „An ontology spectrum"

Die einfachste Form von Ontologien stellen Kataloge und Wortverzeichnisse dar, die typischerweise in natürlicher Sprache spezifiziert werden. Da die Interpretation nicht eindeutig ist, sind sie für maschinelle Verarbeitung ungeeignet.

Thesauri bieten etwas erweiterte Möglichkeiten der Formalisierung, indem sie beispielsweise Synonym-Relationen bereitstellen. Da diese Relationen in den meisten Fällen jedoch nicht ausreichend hierarchisch strukturiert sind, wird eine maschinelle Verarbeitung erschwert.

Eine einfache Form hierarchischer Strukturierung findet man bei Web-Verzeichnissen, wie z.B. Yahoo. Problematisch ist, dass dort die Subklassen-relation is-a nicht formal als transitiv definiert ist (Informal is-a). Die fehlende strikt formale Definition der Vererbungsbeziehungen erschwert eine maschinelle Auswertung der Relationen erheblich.

Die bisher diskutierten Formen von Ontologien befinden sich auf der linken Seite der Diagonalen aus Abbildung 3. Für das Semantic Web sind jedoch nach [McGuinness, 2003a] nur Ontologieformen der rechten Seite geeignet, da sie folgende Mindestanforderungen erfüllen:

- “finite controlled (extensible) vocabulary”.
- „unambiguous interpretation of classes and term relationships".
- „strict hierarchical subclass relationships between classes“.

Semantic Web Ontologien müssen danach über eine strikt formale Definition der Vererbungsbeziehungen zwischen Klassen verfügen (Formal is-a), sowie zwischen Klassen und Instanzen (Formal instance). Es muss die Möglichkeit bestehen, Eigenschaften (properties) von Klassen zu definieren und deren Belegung mit Werten einzuschränken (Value Restrictions), was unter anderem mit Hilfe von Frames (vgl. [Reimer, 1991b]) geschehen kann. Die Ausdrucksmächtigkeit von Ontologien soll durch logische Relationen erweiterbar sein (General Logical constraints). Beispielsweise können Werte, die eine Eigenschaft annehmen können, mit Hilfe von:

- Disjunktionen (Klasse Mann und Frau sind disjunkt),
- Inversen Relationen (ist_mitarbeiter_von und hat_mitarbeiter),
- Part-of Relationen (ein Mitarbeiter ist Teil einer Firma)

bestimmt werden.

2.4.1 Einsatzmöglichkeiten von Ontologien

Durch die Verwendung von Ontologien lassen sich Anwendungen entwickeln, welche eine einfache Form der Konsistenzprüfung ermöglichen. Beispielsweise könnte man für eine Klasse Postleitzahl festlegen, dass sie in Deutschland nur positive fünfstellige Integerwerte annehmen können und dies überprüfen.

Ontologien können darüber hinaus zur Ergänzung oder Vervollständigung von Informationen beitragen. Wenn zum Beispiel eine Postanschrift benötigt wird, kann diese Information durch eine Ontologie dahingehend erweitert werden, dass eine Postanschrift aus mindestens einem Adressaten, einer Straßenangabe und/oder Postfach, einer Postleitzahl und einer Ortsangabe bestehen muss. Wurde in der Adresse zusätzlich ein Vorname angegeben, der eine eindeutige Geschlechtsbestimmung zulässt, erübrigt sich die Frage, ob die Anrede Herr oder Frau lauten soll.

Die Nutzung eines gemeinsamen Vokabulars durch Anwender und Applikationen bildet die Voraussetzung für Interoperabilität. Auf einen bestimmten Anwendungsbereich bezogene Domain-Ontologien, können über eine nahezu vollständige Definition ihrer Terme und den zwischen ihnen bestehenden Relationen verfügen. Über Äquivalenz-Relationen besteht dann die Möglichkeit, einzelne Terme exakt auf andere Terme abzubilden. Beispielsweise könnte eine Ontologie die Definition enthalten, dass ein *Semantic_Web_GmbH_Mitarbeiter* einer *Person* entspricht, bei der die Eigenschaft *Arbeitgeber* mit dem Individuum *Semantic_Web_GmbH* belegt ist. Durch diese Definition ist eine Anwendung, welche weder *Semantic_Web_GmbH_Mitarbeiter* noch *Mitarbeiter* versteht, jedoch die Terme *Person*, *Arbeitgeber* und *Semantic_Web_GmbH* kennt, in der Lage *Semantic_Web_Mitarbeiter* zu definieren.

Die Frage, welche Klassen und Individuen innerhalb einer Ontologie definiert sind und welche Relationen bestehen, könnte mit Hilfe einer datenbankähnlichen Abfrage beantwortet werden. Enthält z.B. eine Ontologie die Definition, dass ein Mitarbeiter maximal einen Arbeitgeber haben kann, lässt sich dies mit

Hilfe von Abfragen überprüfen. Eine solche Gültigkeitsprüfung würde unvollständige Informationen ermitteln, wenn die Eigenschaft Arbeitgeber keinen Wert hätte, oder bei mehr als einem Wert, die Definition als ungültig deklarieren.

Im Hinblick auf Suchoperationen verspricht der Einsatz von Ontologien Verbesserungen in folgenden Bereichen:

- Wenn eine Suchanfrage zu viele oder zu wenige Antworten generiert, kann eine Anwendung die Anfrage in Terme zerlegen und innerhalb einer Ontologie nach diesen Termen suchen. Kommen diese Terme vor, kann eine Spezialisierung oder Generalisierung vorgeschlagen werden, welche die Suche und deren Ergebnisse besser strukturiert und damit verfeinert.
- Der Einsatz von Ontologien erhöht die Vergleichbarkeit von Suchergebnissen. Beispielsweise könnte die Suche nach einem DVD-Player auf einer Ontologie basieren, die die Eigenschaften (Preis, Hersteller, abspielbare Formate, usw.) spezifiziert. Die Ausgabe des Suchergebnisses würde diese Eigenschaften und deren Werte für jeden gefundenen DVD-Player auflisten und dadurch die Vergleichbarkeit verbessern. Darüber hinaus ließen sich die Eigenschaften besonders hervorheben, welche für den Vergleich geeignet sind.
- Die verbesserte Strukturierung und Vergleichbarkeit ermöglicht Anwendern auf individuelle Bedürfnisse zugeschnittene Suchoperationen.

2.4.2 Integration von verteilten Ontologien

Ontologien werden in aller Regel unabhängig voneinander entwickelt. Die Integration verschiedener Ontologien zwecks gemeinsamer Nutzung und Wiederverwendung ist ein zentraler Aspekt des Semantic Web. Nach [Wiederhold, 1994] treten dabei vier Arten von Problemen auf:

- *Naming*: Verschiedene Terme beschreiben das gleiche Konzept (Synonym).
- *Scope*: Mehrere Ontologien beschreiben gemeinsame Konzepte, aber eine Ontologie weist einen höheren Spezialisierungsgrad auf.

- *Encoding*: Die gültigen Eigenschaftswerte sind unterschiedlich geschrieben oder verwenden sogar andere Maßeinheiten.
- *Context*: Ein Term beschreibt unterschiedliche Konzepte (z.B. ein Homonym).

[Heflin, 2001] beschreibt anhand von Beispielen, wie diese Probleme aufgelöst werden können. Um das Naming-Problem zwischen den Synonymen *Angestellter* vs. *Beschäftigter* zu lösen, könnte man die nachfolgende Verbindungsregel definieren:
$Ontologie_{Firma1}$*: Angestellter(x)* ↔ $Ontologie_{Firma2}$*: Beschäftigter(x)*

Die Scope-Problematik wird dadurch gelöst, dass ein spezielleres Konzept einer Ontologie auf das allgemeinere Konzept einer anderen Ontologie abgebildet wird. Wenn man weiß, dass jede *Frau* in der Ontologie *AF* eine *Person* in der Ontologie *FAA* ist, könnte folgende Verbindungsregel dies ausdrücken:
$Ontologie_{AF}$*: Frau(x)* → $Ontologie_{FAA}$*: Person(x)*

Das kompliziertere Encoding-Problem tritt auf, wenn beispielsweise ein Arbeitszeugnis die Beurteilung *hervorragend* und ein anderes die Schulnote *eins* vergibt. In diesem Fall könnten individuelle Regeln definiert werden:
$Ontologie_{Firma1}$*: Beurteilung(x, hervorragend)* ↔ $Ontologie_{Firma2}$*: Beurteilung (x, eins)*
Darüber hinaus muss mit unterschiedlichen Maßeinheiten (z.B. *Meter* vs. *Fuß*) umgegangen werden.
$Ontologie_{English}$*: Foot(x, l)* → $Ontologie_{Metric}$*: Meter(x,*(l, 0,3048))*
Funktionen dieser Art sind nicht unproblematisch, weil durch Rundungsfehler die Funktion bei Verwendung einer Biimplikation nicht mehr korrekt sein könnte.
Die Context-Problematik kann gelöst werden, indem generell angenommen wird, dass Terme aus verschiedenen Ontologien niemals das gleiche Konzept repräsentieren, es sei denn, dies ist explizit durch eine URI vorgegeben.

[Ding et al., 2002] differenziert folgende Möglichkeiten Ontologien zu kombinieren:

- Bei der Verschmelzung (*merging*) wird aus zwei oder mehr Ontologien eine neue generiert. In diesem Fall vereint und ersetzt die neue Ontologie die bestehenden, was unter Umständen Anpassungen und Erweiterungen zur Folge hat.
- Beim Verbinden/Ausrichten (*aligning*) werden Ontologien in eine gegenseitige Übereinstimmung gebracht. Sie bleiben in diesem Fall voneinander getrennt, jedoch muss mindestens eine Ontologie so angepasst werden, dass die Konzepte und das Vokabular mit den überlappenden Teilen der anderen Ontologien zusammenpassen. Dennoch können sie weiterhin jeweils verschiedene Teile einer Domain in unterschiedlichem Detaillierungsgrad beschreiben. In den meisten Fällen ist Aligning die Vorraussetzung für Merging.

- Bringt man verschiedene Ontologien in Beziehung zueinander (*relating*), wird spezifiziert, wie sie im logischen Sinne zusammenhängen. Dabei werden die ursprünglichen Ontologien nicht verändert, aber zusätzliche Axiome beschreiben die Relationen zwischen den Konzepten. Häufig können dadurch nur Teile kombiniert werden, denn größere Anpassungen würden Änderungen an den Ursprungsontologien notwendig machen.

Welche Kombinationsform gewählt wird, ist abhängig von der Frage, ob Ontologien verteilt oder unter einer zentralen Kontrolle entwickelt werden. Zentral entwickelte Ontologien tendieren zu einer stärkeren Integration als verteilte. Anwendungen des Semantic Web werden eher auf der Kombinationsform *Relating* basieren und die logischen Beziehungen zwischen Konzepten verschiedener Ontologien spezifizieren. Enthält beispielsweise eine Ontologie das Konzept "postal code" und eine andere das Konzept "zip code", könnten diese mit Hilfe der logischen Äquivalenzrelation in Beziehung gesetzt werden, ohne die ursprünglichen Ontologien zu verändern.

3 Sprachen des Semantic Web

Die Annotation bestehender HTML-Dokumente mit maschinenverständlichen Informationen wirft unweigerlich die Frage auf, in welcher Ontologie-Repräsentationssprachen diese kodiert werden sollen. Im Zentrum des Interesses stehen dabei die am 10. Februar 2004 vom World Wide Web Konsortium standardisierten Sprachen RDF/S und OWL [W3C-RDF and OWL Recommendation, 2004], durch welche die semantische Interoperabilität für das Semantic Web erreicht werden soll. Das W3C verfolgt mit der Standardisierung das Ziel, eine inkrementelle Entwicklung des Semantic Web zu unterstützten. Daher wurde darauf geachtet, dass die Aufwärtskompatibilität zwischen den Sprachen OWL, RDF/S und der Basistechnologie XML gewährleistet ist.

In Kapitel drei wird zunächst die XML-Sprachfamilie erläutert, da sie die Basis für syntaktische Interoperablität bildet. Danach erfolgt die Erläuterung der mächtigeren Formalismen RDF/S und OWL anhand von Beispielen. Am Ende des Kapitels wird kurz der Ontologieeditor Protégé vorgestellt, da er das zur Zeit interessanteste frei erhältliche Werkzeug für die Modellierung von Ontologien ist.

3.1 XML

Die Extensible Markup Language als Untermenge von SGML (Standard Generalized Markup Language) ist eine W3C Empfehlung [W3C-XML, 2004] und stellt eine Metasprache für Auszeichnungssprachen dar.
Im Unterschied zu HTML können mit XML eigene Markupbefehle und Attribute nach Bedarf definiert werden. Dokumentstrukturen werden dadurch in ihrer Komplexität an die zu repräsentierenden Informationen angepasst.
XML-Dokumente werden unter folgenden Bedingungen als wohl geformt bezeichnet:

- alle Elemente sind korrekt mit Start- und End-Tags geklammert.
- das Dokument enthält genau ein Wurzelelement.

Wohl geformte Dokumente dürfen jedoch unstrukturierten Freitext enthalten. Anwendungen, die XML-Dokumente verarbeiten, werden als XML-Prozessoren bezeichnet. Neben der Überprüfung auf Wohlgeformtheit, können sie Dokumente auch auf Validität prüfen. Gültig sind XML-Dokumente, wenn sie neben der Wohlgeformtheit noch zu einem assoziierten Schema uneingeschränkt konform sind, sie also keinen unstrukturierten Freitext enthalten. Anhand eines Schemas kann man daher die Gültigkeit eines XML- Dokumentes überprüfen.

3.1.1 XML Schema

Die Definition der Syntax, Struktur und Bedeutung der Tags erfolgt für jede Anwendung optional mit XML Schema [W3C-XML-Schema, 2001]. Ein Schema, auch Grammatik genannt, beschreibt eine bestimmte Art von Dokument vollständig und abschließend [Wielage, 2000]. Diese standardisierte Beschreibung ist insbesondere beim Datenaustausch sinnvoll.
Darüber hinaus werden die aus anderen Programmiersprachen bekannten primitiven Datentypen, wie z.B. string, decimal, float, double, boolean, time und date zur Verfügung gestellt. Es ist möglich, komplexe Datentypen unter Verwendung von Vererbungsmechanismen zu beschreiben und somit eine Typisierung für Elemente und Attribute zu erreichen.

3.1.2 XML Namespaces

XML ist als Container für strukturierte Daten konzipiert worden, wird also vor allem von Programmen zum Datenaustausch genutzt. Ein Problem tritt auf, falls ein Programm versucht, zwei XML-Dokumente zu verschmelzen, die jeweils Tags mit gleichem Namen aber unterschiedlicher Bedeutung haben. Um Mehrdeutigkeiten zu vermeiden, werden Namensräume angelegt, indem man den Tags eine URI als eindeutiges Namespace-Präfix zuweist.
Beispielsweise wird aus
<cnn:news>...</cnn:news> und <n24:news>...</n24:news>,
über das Anfügen des Attributs xmlns:prefix="URI":
<cnn:news xmlns:cnn="http://www.cnn.com/xmlns/">...</cnn:news>
<n24:news xmlns:n24="http://www.n24.de/">...</n24:news>

Auf der Basis von eindeutigen Namensräumen ist es möglich, ein definiertes Vokabular wieder zu verwenden.

3.1.3 Fazit

XML verfügt über einen hohen Standardisierungsgrad, stellt die Möglichkeit der Typisierung zur Verfügung, und die klare Struktur bietet einen guten Ausgangspunkt, um Dokumente inhaltlich zu erfassen. Zur Manipulation der XML-Dokumenten zugrunde liegenden Baumstruktur schlägt das W3C die Extensible Stylesheet Language Family (XSL) [W3C-XSL] vor. Mit ihr lassen sich XML-Dokumente über so genannte Stylesheets transformieren und formatieren. Der Datenaustausch kann nur automatisiert ablaufen, wenn die Anwendungen das gleiche XML Schema verwenden, welches die XML-Dokumente vollständig und abschließend beschreibt. Sollte das nicht der Fall sein, müssen Stylesheets entwickelt werden, welche die mitunter recht komplizierten Anpassungen der Datenstrukturen beschreiben.

Nach [Berners-Lee, 2001a] reicht die Ausdrucksmächtigkeit von XML und XML Schema allein nicht aus:

- „XML documents have no inherent meaning“

  und

- „XML Schemas talk about documents, not things

In der geforderten Semantic Web Sprache soll also jeder User einfache Aussagen über alles machen können und gleichzeitig die inhärente Bedeutung erfassbar sein. Da Aussagen über eine Ressource normalerweise zu keinem Zeitpunkt komplett beschrieben sind, stellt die mangelnde Flexibilität von XML ein Problem dar. Die wohl geformten XML-Dokumenten zugrunde liegende Baumstruktur macht es schwierig, Erweiterungen und Änderungen vorzunehmen, weil sich die Daten in den Blättern der einzelnen Äste befinden. Der Zugriff und die Speicherung der Daten ist bei einer Baumstruktur nur durch komplexe Traversierungen möglich. Werden mehrere Aussagen über voneinander unabhängige Ressourcen zusammengesetzt, entsteht ein gerichteter Graph.

Da ein solcher Graph nicht dem Datenmodell von XML entspricht und XML-Elemente nichts über ihre Bedeutung aussagen, entwickelte das W3C das Resource Description Framework (RDF).

3.2 Resource Description Framework (RDF)

Das Resource Description Framework wurde entwickelt, um die Definition und Wiederverwendung von Metadaten zu standardisieren und mit diesen Web-Ressourcen zu beschreiben.

3.2.1 RDF-Datenmodell

Das Resource Description Framework spezifiziert ein domainunabhängiges, syntaxneutrales Datenmodell um RDF Ausdrücke zu repräsentieren. Syntaxneutralität bedeutet, dass zwei Ausdrücke äquivalent sind, wenn ihre Datenmodelle gleichbedeutend sind [W3C-RDF-Modell, 1999]. Mit Hilfe dieses Datenmodells werden einfache semantische Aussagen (Statements) über Ressourcen formuliert. Darüber hinaus ist es möglich, Aussagen über Aussagen zu machen, was als Reifikation bezeichnet wird. Das Datenmodell kann als RDF-Graph, als RDF-Triple und in RDF/XML repräsentiert werden.

RDF verfolgt folgende Designziele [W3C-RDF-Concepts, 2004]:

1. *having a simple data model*: Ein einfaches RDF-Datenmodell soll die maschinelle Verarbeitung und Manipulation erleichtern.
2. *having formal semantics and inference*: Auf Basis des RDF-Datenmodells soll es möglich sein, eine formale Semantik zu definieren, um aus RDF-Aussagen Schlussfolgerungen ziehen zu können.
3. *using an extensible URI-based vocabulary*: Das Vokabular ist vollständig erweiterbar, da in RDF alles durch URI´s eindeutig identifiziert werden kann.
4. *using an XML-based syntax*: Zur Serialisierung wird XML verwendet, um das RDF-Datenmodell damit zu kodieren und zwischen Anwendungen auszutauschen. Für die als RDF/XML bezeichnete Syntax ist es nicht erforderlich, dass sie "wohl geformt" ist, da RDF/XML nicht zur Validierung entwickelt wurde.

5. *supporting use of XML schema datatypes*: RDF verwendet die primitiven Datentypen von XML Schema, um den Datenaustausch zwischen RDF und anderen XML-Anwendungen zu unterstützen.
6. *allowing anyone to make statements about any resource*: Da das WWW offen, dezentral und dynamisch ist, muss jeder User Aussagen über alles machen können. Dies führt dazu, dass Aussagen unsinnig oder inkonsistent sein können. Beim Design von RDF-Anwendungen sollte dies berücksichtigt und unvollständige und inkonsistente Aussagen toleriert werden.

Nach dem RDF-Datenmodell ist eine Aussage ein Tripel bestehend aus Subjekt, Prädikat und Objekt und kann als gerichteter Graph mit Knoten und Kanten repräsentiert werden. Eine Menge von Tripeln wird als RDF-Graph bezeichnet, dessen Bedeutung sich aus der Konjunktion aller Tripel ergibt.

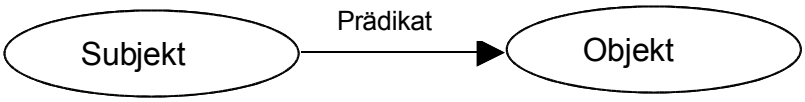


Abbildung 4: RDF-Graph eines Tripels

3.2.2 RDF Objekt Typen

Das Datenmodell des Recource Description Framework unterscheidet folgende drei Typen von Objekten:

- *Resource*: Alle mit RDF beschriebenen Objekte, die durch eine URI eindeutig gekennzeichnet sind, werden als Ressource bezeichnet. Dabei kann es sich um Web-Dokumente, Fragmente von Web-Dokumenten oder Objekte aus der realen Welt handeln.
- *Property* (Eigenschaft): Die Property wird auch als ein Prädikat bezeichnet und ist eine Charakteristik, ein Attribut oder eine Relation zur Beschreibung einer Ressource. Jede Property hat eine spezielle Bedeutung, definiert ihre erlaubten Werte, sowie den Typ von Ressourcen den sie beschreiben kann und die Relationen zu anderen Properties. Eine Property ist ebenfalls immer eine Ressource.

- *Statement* (Aussage): RDF stellte eine auf Aussagen basierende Beschreibungsmöglichkeit von Ressourcen zur Verfügung. Eine Aussage ist ein Tripel aus: Subjekt (Ressource), Prädikat (Property) und Objekt. Das Objekt ist der Wert der Eigenschaft des Subjekts und kann eine Ressource, ein Literal oder ein primitiver XML Schema-Datentyp sein.

3.2.3 RDF-Graph

Werden beispielsweise folgende Aussagen gemacht:

- es existiert eine Person, identifiziert durch "http://www.example.org/Person/Fritz_Mustermann",
- mit dem Vornamen "Fritz" und dem Nachnamen "Mustermann",
- sowie Angaben zur Strasse, Hausnummer, Ort und Postleitzahl,

dann sieht der RDF-Graph wie folgt aus:

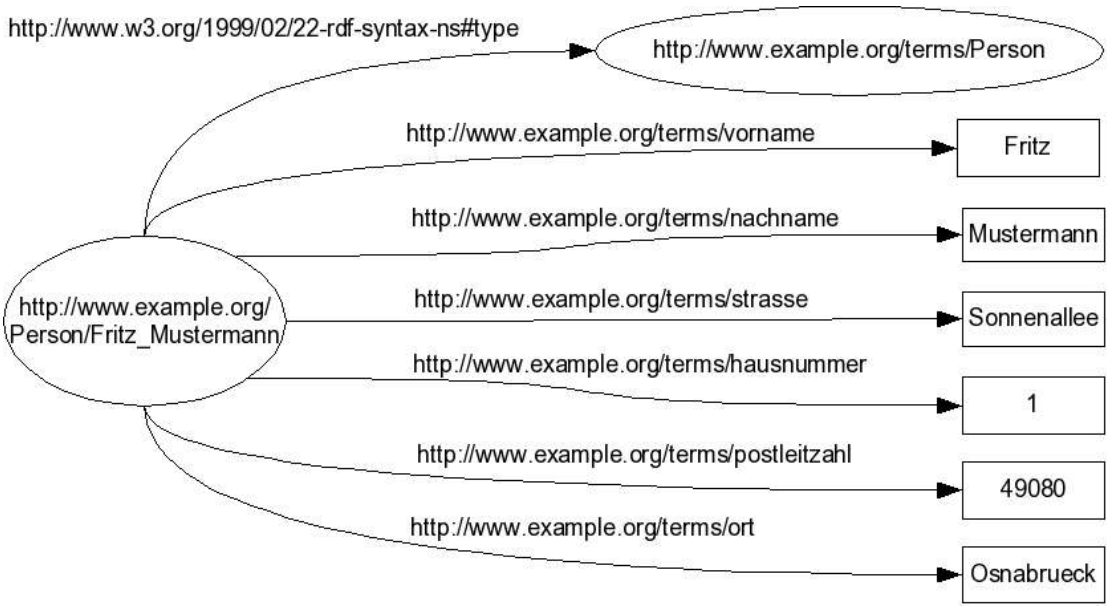


Abbildung 5: Repräsentation der Aussagen über "Fritz_Mustermann" als RDF-Graph

Knoten, welche eine URI-Referenz [W3C-RDF-URI-Reference, 2004] beschreiben, werden als Ellipsen dargestellt, während Knoten, die Literale beschreiben, als Rechtecke repräsentiert werden.

Die gerichtete Kante eines RDF-Graphen repräsentiert das Prädikat und identifiziert die Relation zwischen den Knoten, die es verbindet, mit Hilfe einer URI. Ein wichtiger Aspekt von RDF ist, dass direkt nur binäre Relationen repräsentiert werden können. Dies ist jedoch keine Einschränkung der Ausdrucksmächtigkeit, denn mehrstellige Relationen können als Sequenz von binären Relationen dargestellt werden.

3.2.4 Literale

Literale sind konstante Zeichenketten zur Beschreibung des Wertes einer Eigenschaft. Da sie im Gegensatz zu Ressourcen keine Identität besitzen, können sie nicht als Subjekt oder Prädikat einer Aussage fungieren, sondern nur als Objekt. Man unterscheidet zwischen zwei Arten von Literalen [W3C-RDF-Literals, 2004]:

- Ein *Plain Literal* ist eine "selbstbezeichnende" Zeichenkette mit optionalem "Language Tag".
- Ein *Typed Literal* ist eine Zeichenkette kombiniert mit einer URI, die den Datentyp angibt. RDF definiert selbst keine eigenen Datentypen, sondern verwendet die XML Schema Datentypen.

Alles was durch ein Literal repräsentiert wird, kann auch durch eine URI dargestellt werden. Die Verwendung von Literalen ist intuitiver, bringt jedoch den Nachteil mit sich, dass über Literale keine weiteren Aussagen gemacht werden können.

3.2.5 RDF-Triple

Die folgende Tabelle 2 zeigt die Repräsentation der Aussagen über "Fritz_Mustermann" als RDF-Triple.

Subjekt	Prädikat	Objekt
http://www.example.org/ Person/Fritz_Mustermann	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.example.org/ terms/Person
http://www.example.org /Person/Fritz_Mustermann	http://www.example.org/terms/ vorname	"Fritz"
http://www.example.org/ Person/Fritz_Mustermann	http://www.example.org/terms/ nachname	"Mustermann"

http://www.example.org/ Person/Fritz_Mustermann	http://www.example.org/terms/strasse	"Sonnenallee"
http://www.example.org/ Person/Fritz_Mustermann	http://www.example.org/terms/ hausnummer	"1"
http://www.example.org/ Person/Fritz_Mustermann	http://www.example.org/terms/ postleitzahl	"49080"
http://www.example.org/ Person/Fritz_Mustermann	http://www.example.org/terms/ort	"Osnabrueck"

Tabelle 2: Aussagen über "Fritz_Mustermann" als RDF-Triple.

3.2.6 RDF/XML Syntax

Im folgenden werden die Aussagen über "Fritz_Mustermann" in RDF/XML Syntax repräsentiert. Die Zeilennummerierung dient der Erläuterung.

```
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:      xmlns:exterms="http://www.example.org/terms/">
4:
5:      <exterms:Person rdf:about="http://www.example.org/Person/Fritz_Mustermann">
6:              <exterms:vorname>Fritz</exterms:vorname>
7:              <exterms:nachname>Mustermann</exterms:nachname>
8:              <exterms:strasse>Sonnenallee</exterms:strasse>
9:              <exterms:hausnummer>1</exterms:hausnummer>
10 :            <exterms:postleitzahl>49080</exterms:postleitzahl>
11:             <exterms:ort>Osnabrueck</exterms:ort>
12:     </exterms:Person>
13:
14: </rdf:RDF>
```

Beispiel 1: Aussagen über "Fritz_Mustermann" in RDF/XML Syntax

Die erste Zeile gibt an, dass es sich um ein XML-Dokument in der Version 1.0 handelt. Nach dem RDF-Start-Tag *<rdf:RDF* in Zeile zwei wird der XML-Namensraum xmlns deklariert. Alle Tags, die mit dem Präfix rdf: beginnen, sind Teil des Namensraums, der durch die URI *http://www.w3.org/1999/02/22-rdf-syntax-ns#* identifiziert wird. Ebenso verhält es sich mit dem Präfix exterms:, welches für Terme eines Vokabulars benutzt wird, das von einer Beispielorganisation namens *example.org* definiert wurde. Der RDF-Start-Tag wird am Ende von Zeile 3 mit dem Zeichen ">" geschlossen und der RDF-End-Tag befindet sich in Zeile 14.

Die eigentlichen Aussagen werden in den Zeilen 5-12 gemacht. Zeile 5 gibt hinter dem *rdf:about* Tag das Subjekt der Aussage als *http://www.example.org/Person/Fritz_Mustermann* an. Der Typ dieses Subjekts ist wiederum eine Aussage mit dem impliziten Prädikat *http://www.w3.org/1999/02/22-rdf-syntax-ns#type* und dem Objekt *http://www.example.org/terms/Person*.

Die absolute URI-Referenz *http://www.w3.org/1999/02/22-rdf-syntax-ns#type* ist nach dem Muster URI-Basis#Fragment aufgebaut. Das Fragment, in diesem Fall type, wird als so genannter *Fragment Identifier* [W3C-RDF-Fragment Identifiers, 2004] bezeichnet und gibt die relative Position zur Basis des Dokumentes an. Innerhalb der *exterms:Person* Start- und End-Tags werden in den Zeilen 6-11 die Prädikate *vorname*, *nachname*, usw. und ihre jeweiligen Objekte als *Plain Literals* angegeben.

3.2.7 Abkürzung und Organisation von URI-Referenzen

Die RDF/XML Syntax kann in abgekürzter Form notiert werden. Das Beispiel 1 verwendet in Zeile 5 die abgekürzte Form

```
5:    <exterms:Person rdf:about="http://www.example.org/Person/Fritz_Mustermann">
<exterms:vorname>Fritz</exterms:vorname>
[...]
</exterms:Person>
```

und ist äquivalent mit folgender Notation:

```
5a:   <rdf:Description rdf:about="http://www.example.org/Person/Fritz_Mustermann">
<rdf:type rdf:resource="http://www.example.org/terms/Person"/>
<exterms:vorname>Fritz</exterms:vorname>
[...]
</rdf:Description>
```

Hierbei wurde das *rdf:Description* Element, welches explizit den Typ des Subjekts als Ressource angibt, durch *exterms:Person* ersetzt. Daher impliziert die in Zeile 5 verwendete abgekürzte Form, dass *Fritz_Mustermann* von Typ *Person* ist.

Zur Abkürzung von URI-Referenzen muss die URI-Basis bekannt sein. Sie kann entweder die URI des RDF/XML Dokumentes sein oder explizit durch den

xml:base [W3C-XML-Base, 2001] Tag angegeben werden. Die URI-Basis kann von folgenden RDF/XML Attributen verwendet werden, da ihre Werte URI-Referenzen sind.

- rdf:resource
- rdf:about
- rdf:ID
- rdf:datatype

Beispiel 2 illustriert anhand eines Beispiel die Abkürzung von URI-Referenzen. Die Beispielorganisation example.org hat für ihre Mitarbeiter in Zeile 5 die URI-Basis durch xml:base auf *http://www.example.org/Mitarbeiter* festgelegt.

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:      xmlns:exterms="http://www.example.org/terms/"
5:      xml:base="http://www.example.org/Mitarbeiter">
6:
7:      <exterms:Person rdf:ID="Fritz_Mustermann">
8:              <exterms:vorname rdf:datatype="&xsd;string">Fritz</exterms:vorname>
9:              <exterms:nachname rdf:datatype="&xsd;string">Mustermann
                </exterms:nachname>
10:             <exterms:strasse rdf:datatype="&xsd;string">Sonnenallee</exterms:strasse>
11:             <exterms:hausnummer rdf:datatype="&xsd;integer">1</exterms:hausnummer>
12:             <exterms:postleitzahl rdf:datatype="&xsd;integer">49080</exterms:postleitzahl>
13:             <exterms:ort rdf:datatype="&xsd;string">Osnabrueck</exterms:ort>
14:     </exterms:Person>
15:
16: </rdf:RDF>
```

Beispiel 2: Abkürzung von URI-Referenzen

Da in diesem Beispiel *Typed Literals* verwendet werden, muss deren Datentyp durch rdf:datatype angegeben werden. Die *DOCTYPE* Deklaration in Zeile 2 definiert dazu die XML-Entität *xsd*. Der String *xsd* repräsentiert dabei die URI-Referenz des Namensraums für XML Schema Datentypen. Durch diese Deklaration ist es in den Zeilen 8-13 möglich, über die Entitätsreferenz *&xsd* auf die

vollständige URI-Referenz zuzugreifen, ohne sie jedes Mal angeben zu müssen.

In Zeile 7 wird das Subjekt der Aussage mit Hilfe von *rdf:ID="Fritz_Mustermann"* als *Fragment Identifier* angegeben. Alternativ hätte man auch *rdf:about="#Fritz_Mustermann"* verwenden können, denn beide Formen stellen einen Abkürzungsmechanismus für die gleiche vollständige URI-Referenz *http://www.example.org/Mitarbeiter/#Fritz_Mustermann* dar. Der Unterschied besteht lediglich darin, dass der Name hinter *rdf:ID* einzigartig in Bezug auf die durch *xml:base* definierte URI-Basis sein muss, bzw. einzigartig in Bezug auf das Dokument, wenn *xml:base* nicht definiert ist. *rdf:ID* erleichtert daher die Überprüfung bei der Zuweisung von Namen für Subjekt- oder Objekt-Ressourcen.

Ein gutes Hilfsmittel zur Überprüfung der RDF/XML-Syntax ist der [W3C-RDF-Validation-Service, 2003], welcher zusätzlich die RDF-Triple und den Graphen ausgibt. Validiert man das Beispiel 2, ohne die *xml:base* aus Zeile 5 anzugeben, wird durch die vollständige URI-Referenz *http://www.w3.org/RDF/Validator/run/1091191785533#Fritz_Mustermann* deutlich, dass dann *rdf:ID="Fritz_Mustermann"* relativ zur URI-Basis des Dokumentes ("beim W3C") interpretiert wird.

Zu beachten ist, dass RDF keinerlei Annahmen über Beziehungen zwischen zwei Ressourcen macht, nur weil ihre URI-Referenzen die gleiche Basis haben oder irgendwie ähnlich sind.

Ein grundlegendes Designziel von RDF ist, dass jeder Aussagen über Ressourcen machen oder erweitern kann. Dies kann durch die Modifikation des RDF Dokumentes erfolgen, indem die Ressource ursprünglich beschrieben wurde oder durch ein neues Dokument, welches auf die ursprüngliche Ressource verweist. Im folgenden Beispiel wird auf der Webseite von *www.example-web-service-geschlechtsbestimmung.org* die zusätzliche Aussage über den in Beispiel 2 beschriebenen *Fritz_Mustermann* gemacht, dass dieser das Geschlecht *maennlich* hat.

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:      xmlns:exterms="http://www.example-web-service-geschlechtsbestimmung.org/terms/">
5:
6:      <rdf:Description rdf:about="http://www.example.org/Mitarbeiter/#Fritz_Mustermann">
7:              <exterms:geschlecht rdf:datatype="&xsd;string">maennlich</exterms:geschlecht>
8:      </rdf:Description>
9: </rdf:RDF>
```

Beispiel 3: Zusätzliche Aussagen über an anderer Stelle beschriebene Ressourcen

3.2.8 Blank Nodes

Neben Knoten, die mit einer URI-Referenz beschriftet sind, sieht die RDF-Spezifikation *anonyme* Knoten, so genannte Blank Nodes [W3C-RDF/XML-Syntax, 2004] vor, welche folgende Eigenschaften haben:

- Blank Nodes können verwendet werden, wenn es unzweckmäßig ist, eine URI oder ein Literal zu vergeben.
- Da Blank Nodes über keine URI-Referenz verfügen, sind sie außerhalb von Dokumenten, in denen sie definiert wurden, unsichtbar.
- Ein Blank Node kann innerhalb eines RDF/XML Dokuments eindeutig durch einen *blank node identifier* definiert werden. Dadurch kann auf ihn von verschiedenen Stellen innerhalb des Dokuments verwiesen werden.
- Anonyme Knoten können nur als Subjekt oder Objekt innerhalb einer RDF Repräsentation erscheinen.

Blank Nodes bieten eine Möglichkeit, Aussagen über Knoten zu machen, die keine URI-Referenz haben, aber über ihre Relationen zu anderen Ressourcen definiert sind.
Möchte man beispielsweise den Graphen aus Abbildung 5 um die Aussage erweitern, dass die Angaben zu Strasse, Hausnummer, Postleitzahl und Ort die Adresse von Fritz Mustermann beschreiben, muss ein zusätzlicher Zwischenknoten für die Adresse eingefügt werden. Dieser neue Knoten re-

präsentiert das Konzept der Adresse, welches aus den Relationen (Prädikaten) *strasse*, *hausnummer*, usw. und ihren jeweiligen Werten besteht.

Dabei sollte man beachten, dass die Person Fritz Mustermann und seine Adresse nicht das gleiche sind und dies bei der Repräsentation berücksichtigen. Wird beispielsweise für die Adresse keine URI angegeben, kann die Adresse als Blank Node repräsentiert werden. Die Bedeutung eines Blank Node kann als "*there is a resource*" [W3C-RDF-Primer, 2004] beschrieben werden. Abbildung 6 zeigt den RDF-Graph, bei dem die Adresse als Blank Node modelliert wurde, und Beispiel 4 die entsprechende XML/RDF Syntax .

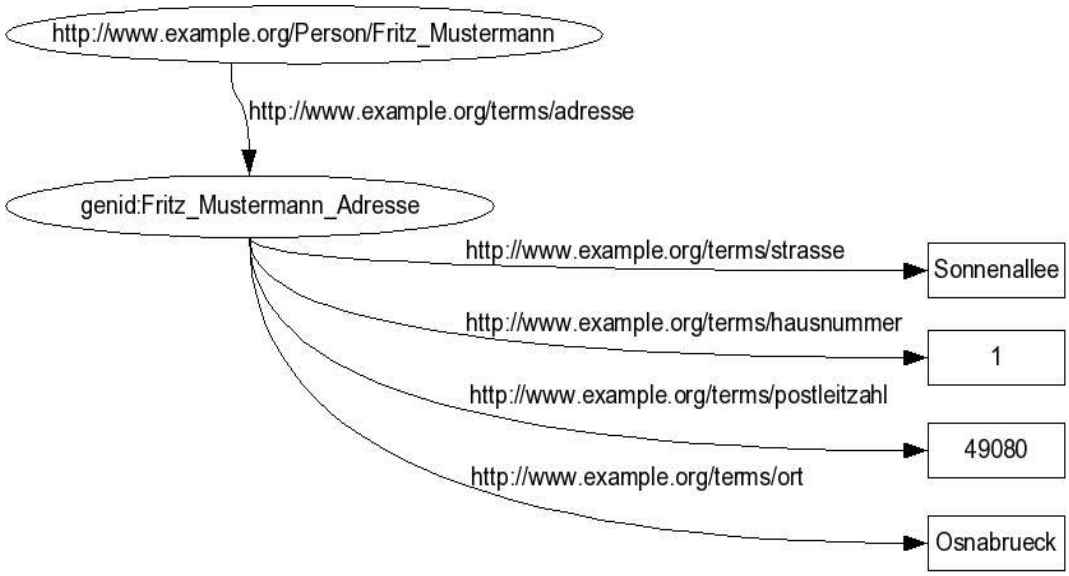


Abbildung 6: RDF-Graph mit einem Blank Node

Wie aus dem RDF-Graph ersichtlich ist, handelt es sich bei der Repräsentation der Beziehung zwischen *Fritz_Mustermann* und den Literalen, die seine Adresse repräsentieren, um eine 5-stellige Beziehung. Der Blank Node bieten eine Möglichkeit mehrstellige Relationen als Sequenz von binären Relationen darzustellen.

```
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:      xmlns:exterms="http://www.example.org/terms/">
4:
```

```
5:    <rdf:Description rdf:about="http://www.example.org/Person/Fritz_Mustermann">
6:          <exterms:adresse rdf:nodeID="Fritz_Mustermann_Adresse"/>
7:    </rdf:Description>
8:
9:    <rdf:Description rdf:nodeID="Fritz_Mustermann_Adresse">
10:         <exterms:strasse>Sonnenallee</exterms:strasse>
11:         <exterms:hausnummer>1</exterms:hausnummer>
12:         <exterms:postleitzahl>49080</exterms:postleitzahl>
13:         <exterms:ort>Osnabrueck</exterms:ort>
14:   </rdf:Description>
15:
16: </rdf:RDF>
```

Beispiel 4: RDF/XML Syntax mit einem Blank Node

Der *blank node identifier* "*Fritz_Mustermann_Adresse*" wird innerhalb der RDF/XML Syntax als Wert des *rdf:nodeID* Tag angegeben, welcher nicht mit *rdf:ID* verwechselt werden sollte. In Zeile 6 ist die als Blank Node modellierte Adresse das Objekt des Prädikats *exterms:adresse*, dessen Subjekt *Fritz_Mustermann* ist. Zeile 9 modelliert den Blank Node als Subjekt verschiedener anderer Aussagen.

3.2.9 RDF Container

Zur Gruppierung von Ressourcen (inklusive Blank Nodes) oder Literalen stellt RDF Container [W3C-RDF-Primer, 2004a] zur Verfügung. Die RDF Syntax definiert drei Typen von Containern, die sich in der Art der Gruppierung ihrer Listen-Elemente unterscheiden.

- *rdf:Bag* repräsentiert eine Gruppe von Ressourcen oder Literalen, bei der Elemente mehrfach vorkommen können und deren Reihenfolge ohne Bedeutung ist.
- *rdf:Seq* (Sequence) unterscheidet sich von rdf:Bag dadurch, dass die Reihenfolge der Elemente von Bedeutung ist.
- *rdf:Alt* (Alternative) repräsentiert eine Alternative zwischen Elementen des Containers, wobei die Reihenfolge keine Rolle spielt.

Da Container selber Ressourcen sind, wird ihr Typ über die *rdf:type* Property mit den Werten *rdf:Bag*, *rdf:Seq* oder *rdf:Alt* festgelegt. Container können Blank Nodes oder Ressourcen mit URI-Referenz sein und kennzeichnen die Gruppe ihrer Elemente als ganzes.

Die einzelnen Elemente eines Containers werden mit Hilfe der *container membership property* nach dem Muster *rdf:_n* für *n=1,2,3,…* definiert, wobei die Reihenfolge der Listenelemente nur bei *rdf:Seq* von Bedeutung ist.

Die Aussage “Das Geschlecht der Person könnte entweder weiblich oder maennlich sein”, kann als RDF-Graph und in RDF/XML folgendermaßen ausgedrückt werden.

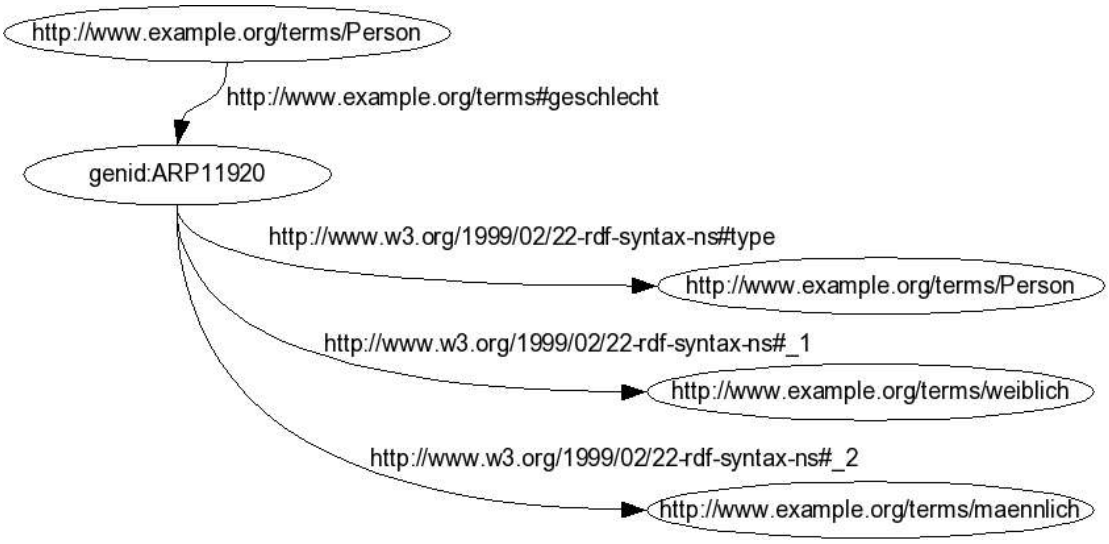


Abbildung 7: Beispiel für einen rdf:Alt Container als RDF-Graph

```
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:      xmlns:exterms="http://www.example.org/terms">
4:
5:      <rdf:Description rdf:about="http://www.example.org/terms/Person">
6:              <exterms:geschlecht>
7:                      <rdf:Alt>
8:                              <rdf:li rdf:resource="http://www.example.org/terms/weiblich"/>
9:                              <rdf:li rdf:resource="http://www.example.org/terms/maennlich"/>
10:                     </rdf:Alt>
11:             </exterms:geschlecht>
12:     </rdf:Description>
13: </rdf:RDF>
```

Beispiel 5: Alt Container in RDF/XML Syntax

In diesem Beispiel wurde in Zeile 8 und 9 nicht jedes Listenelement mit *rdf:_n* durchnummeriert, sondern statt dessen *rdf:li* (List Item) verwendet. Beim Generieren des entsprechenden RDF-Graphen entstehen daraus die *container membership properties rdf:_1* und *rdf:_2* (vgl. Abbildung 7), deren Werte, die Ressourcen *weiblich* und *maennlich* sind.
In Zeile 7 wird der *rdf:Alt* Container als Blank Node modelliert und stellt das Subjekt für jedes der beiden Listenelemente dar. Da sich *rdf:Alt* innerhalb der *exterms:geschlecht* Property-Tags befindet, handelt es sich hier um ein weiteres Beispiel einer verkürzten Notation (vgl. Beispiel 2), wobei in diesem Beispiel *rdf:Description* und *rdf:type* durch *rdf:Alt* ersetzt worden sind. Der Blank Node vom Typ Alt-Container ist daher auch der Wert der Property *exterms:geschlecht* des Subjekts Person.

Mit der verwendeten RDF Repräsentation kann nicht ausgedrückt werden, dass es neben *weiblich* und *maennlich* keine weiteren Listenelemente innerhalb des Alt-Containers gibt. Um Container zu beschreiben, die nur die spezifizierten Elemente enthalten, stellt RDF Collections [W3C-RDF-Primer, 2004b] zu Verfügung.
Collections gruppieren Elemente in Form einer Listenstruktur. Das Vokabular bietet dazu den vordefinierten Typ *rdf:List*, die Properties *rdf:first* und *rdf:rest*, sowie die Ressource *rdf:nil*. Durch die vordefinierten Properties first und rest kann, wie bei der Programmiersprache Lisp, die Struktur traversiert werden. Jedes Listenelement ist das Objekt der *rdf:first* Property mit einer Ressource von Typ List als Subjekt. Diese List-Ressource verbindet die restlichen Listen-Elemente mit Hilfe der *rdf:rest* Property. Ist das Ende der Liste erreicht, wird der rdf:rest Property die Ressource *rdf:nil* als Objekt zugewiesen, wobei *rdf:nil* die leere Liste repräsentiert.

Um den Aufbau von Listenstrukturen zu vereinfachen, kann alternativ zum oben beschriebenen Vokabular das Attribut *rdf:parseType="Collection"* eingesetzt werden. Die Listenstruktur wird dabei mit Hilfe der innerhalb des *rdf:parseType="Collection*" Attributes eingebetteten Listenelementen aufgebaut.

Der Konjunktiv innerhalb der Beispielaussage “Das Geschlecht der Person könnte entweder *weiblich* oder *maennlich* sein” soll auf einen wichtigen Aspekt der RDF Repräsentation aus Beispiel 5 und Abbildung 7 hinweisen. Zu beachten ist, dass RDF lediglich vordefinierte Typen und Properties zur Beschreibung von Containern zur Verfügung stellt. Die Bedeutung, welche mit diesen Containern assoziiert wird, z.B. dass die Container-Elemente weiblich und maennlich Alternativen darstellen, ist lediglich beabsichtigt. Aus diesem Grund müssen Anwendungen entwickelt werden, welche sich gemäß der von RDF beabsichtigten Bedeutung verhalten.

3.2.10 Reification

Bisher wurde gezeigt, wie man Aussagen über beliebige Ressourcen machen kann. Da Aussagen selber Ressourcen sind, bietet RDF die Möglichkeit, Aussagen über Aussagen zu machen, was als *Reification* (Vergegenständlichung) bezeichnet wird. Das RDF *Reification* Vokabular besteht aus dem Typ rdf:Statement und den RDF Properties *subject*, *predicate* und *object*. Häufig wird Reification dazu verwendet, um Herkunftsinformationen über Aussagen aufzuzeichnen. Möchte die Organisation example.org beispielsweise notieren, dass die Geschlechtsbestimmung von *Fritz_Mustermann* durch *example-web-service-geschlechtsbestimmung.org* vorgenommen wurde, könnte dies in RDF/XML folgendermaßen aussehen.

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:      xmlns:dc="http://purl.org/dc/elements/1.1/"
5:              xmlns:exterms="http://www.example.org/terms/"
6:              xml:base="http://www.example.org/Mitarbeiter">
7:
8:      <rdf:Description rdf:ID="Fritz_Mustermann">
9:              <exterms:geschlecht rdf:datatype="&xsd;string">maennlich</exterms:geschlecht>
10:     </rdf:Description>
11:
12:     <rdf:Statement rdf:about="#reification_statement">
13:             <rdf:subject
                 rdf:resource="http://www.example.org/Mitarbeiter#Fritz_Mustermann"/>
```

```
14:          <rdf:predicate rdf:resource="http://www.example.org/terms/geschlecht"/>
15:          <rdf:object rdf:datatype="&xsd;string">maennlich</rdf:object>
16:
17:    <dc:creator
             rdf:resource="http://www.example-web-service-geschlechtsbestimmung.org"/>
18:    </rdf:Statement>
19:
20: </rdf:RDF>
```

Beispiel 6: Beispiel für Reification in RDF/XML

Zeile 4 der RDF/XML Syntax führt über das Präfix *dc* das Prädikat „*creator*“ aus dem Namensraum des Dublin Core Element Set [Dublin-Core, 2004] ein. Der Dublin Core Element Set stellt eine der bekanntesten Anwendungen von RDF dar und besteht aus einem Vokabular von 15 typischen Metadatenelementen (*title*, *creator*, *subject*, *keywords*, ...) zur inhaltlichen und formalen Beschreibung von Ressourcen.

3.3 RDF Schema (RDFS)

RDF Schema [W3C-RDF-Schema, 2004] wird als "RDF´s vocabulary description language" bezeichnet und erweitert RDF um die Möglichkeit ein Typsystem aufzubauen. Dazu definiert RDFS Klassen und Properties, mit deren Hilfe Gruppen von verwandten Ressourcen und deren Beziehungen zueinander klassifiziert werden können.

Da RDFS domainneutral entwickelt wurde, stellt es kein Vokabular für applikationsspezifische Klassen (wie z.B. eine Klasse *exterms:Person* und die Property *exterms:nachname*) zu Verfügung, sondern bietet die Möglichkeit, die sinnvolle Verwendung und Kombination von Klassen und Eigenschaften mit Hilfe vordefinierter Sprachkonstrukte zu beschreiben. Zum Aufbau einer Taxonomie orientiert sich RDF Schema an den Formalismen zur Darstellung Semantischer Netze (vgl. 2.3) und führt die folgenden Klassen, Properties und Constraints ein.

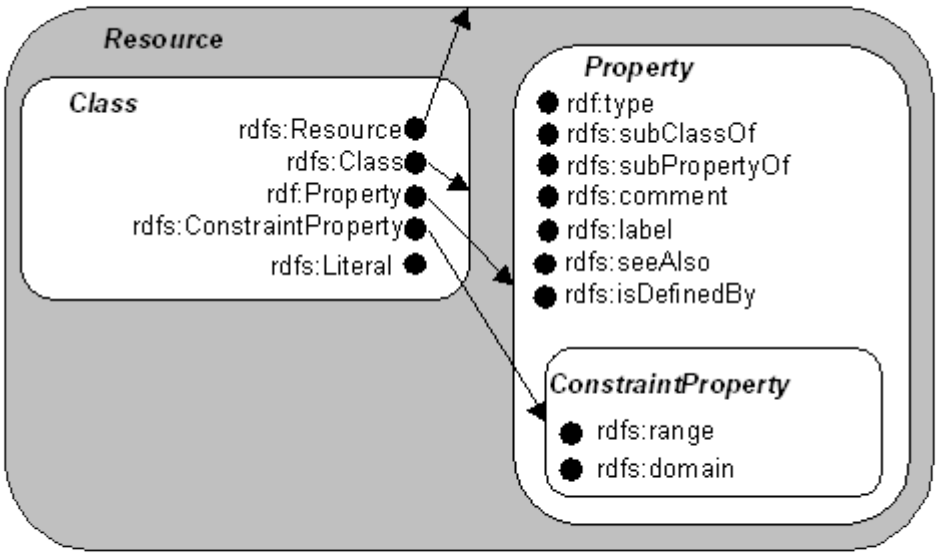


Abbildung 8: Klassen, Ressourcen und Properties in
RDF Schema [W3C-RDF-Schema, 2000]

In Abbildung 8 werden Klassen durch abgerundete Rechtecke dargestellt. Wird eine Klasse von einem abgerundeten Rechteck umschlossen, ist sie Unterklasse dieser Oberklasse. Ressourcen sind durch Punkte gekennzeichnet, die über Pfeile mit der Klasse verbunden sind, in der sie definiert werden. Befindet sich eine Ressource innerhalb einer Klasse, dann ist die Ressource vom Typ der umgebenden Klasse und es existiert entweder explizit oder implizit eine *rdf:type* Property.

RDF Schema Klassen:

- *rdfs:Resource*: Alle Dinge, welche mittels RDF Ausdrücken beschrieben werden können, werden als Ressource bezeichnet und können Instanzen einer oder mehrerer Klassen sein.
- *rdfs:Class*: Ressourcen können zur besseren Kategorisierung in Klassen eingeteilt werden. *rdfs:Class* ist die Oberklasse aller in RDF definierten Klassen und dabei immer auch eine Instanz von rdfs:Class.
- *rdf:Property*: Die Klasse *rdf:Property* ist die Oberklasse aller Ressourcen, die Properties (Relationen zwischen Subjekt- und Objekt-Ressourcen), darstellen.

Abbildung 9 zeigt die Klassenhierarchie in einer Graphenrepräsentation bestehend aus Knoten und gerichteten Kanten. Die *rdf:type* Kante verweist auf eine Instanzbeziehung zwischen einer Ressource und einer Klasse. Bemerkenswert ist die rekursive Beziehung zwischen *rdfs:Resource* und *rdfs:Class*. Die Klassendefinition *rdfs:Class* ist auf der einen Seite selber eine Ressource, also eine Unterklasse von *rdfs:Resource*, auf der anderen Seite ist rdfs:Resource eine Instanz von *rdfs:Class*.

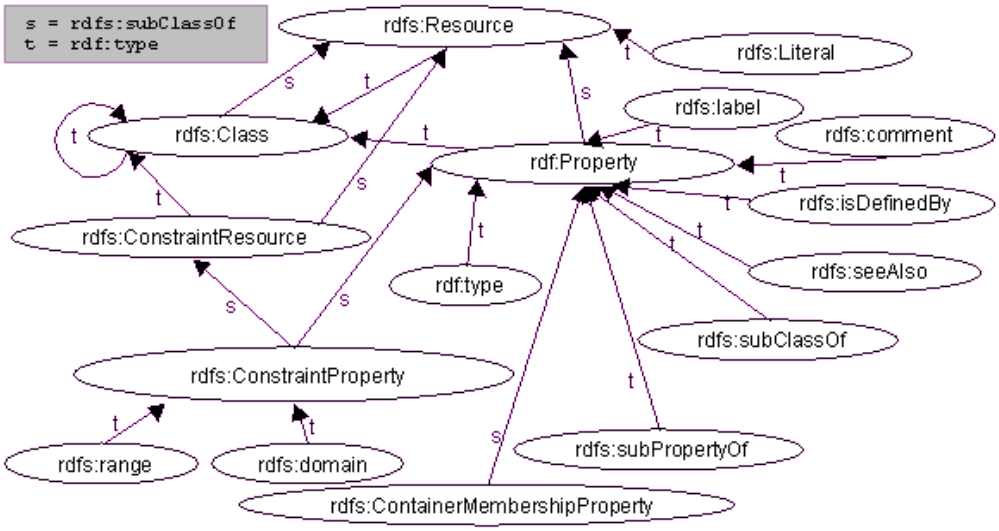


Abbildung 9: Klassenhierarchie von RDF Schema [W3C-RDF-Schema, 2000a]

RDF Schema Properties:

- *rdf:type*: Gibt an, dass eine Ressource Instanz einer Klasse ist. Mögliche Wertebereiche von *rdf:type* sind Instanzen von *rdfs:Class*.
- *rdfs:subClassOf:* Diese Eigenschaft beschreibt eine transitive Vererbungsbeziehung zwischen zwei Klassen. Es ist möglich, dass eine Klasse Unterklasse mehrerer anderer Klassen ist (Mehrfachklassifikation), wobei jedoch kein Zyklus entstehen darf.
- *rdfs:subPropertyOf*: Mit Hilfe von *rdfs:subPropertyOf* wird eine transitive Vererbungsbeziehung zwischen zwei Properties beschrieben. Eine Untereigenschaft stellt dabei eine Spezialisierung einer oder mehrerer anderer Eigenschaften dar, wobei kein Zyklus entstehen darf.
- *rdfs:comment* und *rdfs:label*: Die beiden Properties versehen eine Ressource mit einem für den Menschen lesbaren Kommentar und Namen.

Die verwendete Sprache wird mit einem optionalen Language Tag [RFC-3066, 2001] angezeigt.

- *rdfs:seeAlso*: Diese Eigenschaft ordnet einer Ressource eine andere Ressource zu, welche zusätzliche Informationen bereitstellt.
- *isDefinedBy*: Diese Property verweist auf eine Ressource, in der die beschriebene Ressource definiert wird. Bei *isDefinedBy* handelt es sich um eine Subproperty von *rdfs:seeAlso*.

Constraint Properties:

- *rdfs:range*: Zur Einschränkung des Wertebereiche einer Property werden durch *rdfs:range* die Klassen angegeben, welche als Werte dieser Property erlaubt sind.
- *rdfs:domain*: Zur Festlegung des Definitionsbereichs einer Property werden durch *rdfs:domain* die Klassen angegeben, in denen diese Property verwendet werden kann.

Der Umgang mit Constraint Properties wird anhand der folgenden Beispiele verdeutlicht. Soll angegeben werden, dass die Property *arbeitet_für* Werte annehmen kann, die Instanzen der Klasse *Firma* sind, sieht der Auszug aus der RDF Syntax folgendermaßen aus:

```
1: <rdfs:Class rdf:ID="Firma"/>
2: <rdf:Property rdf:ID="arbeitet_für">
3:      <rdfs:range rdf:resource="#Firma"/>
4: </rdf:Property>
```

Beispiel 7: Auszug aus einem RDF Schema mit rdfs:range Property

Sollte zwischen Zeile 2 und 4 mehr als eine Range-Property angegeben werden, ist zu beachten, dass die Werte der *arbeitet_für* Property Ressourcen sind, die Instanzen aller als Range angegebenen Klassen sein müssen. Entsprechendes gilt für die Domain-Property, welche im folgenden Beispiel hinzugefügt wurde. Dadurch wird ausgesagt, dass die Eigenschaft *arbeitet_für* ein Objekt vom Typ *Firma* und ein Subjekt vom Typ *Mitarbeiter* hat.

```
1: <rdfs:Class rdf:ID="Mitarbeiter"/>
2: <rdfs:Class rdf:ID="Firma"/>
3: <rdf:Property rdf:ID="arbeitet_für">
4:      <rdfs:domain rdf:resource="#Mitarbeiter"/>
5:      <rdfs:range rdf:resource="#Firma"/>
6: </rdf:Property>
```

Beispiel 8: Auszug aus einem RDF Schema mit rdfs:range und rdfs:domain Property

3.3.1 Vergleich der Typsysteme von RDF Schema und objektorientierten Programmiersprachen

Das Typsystem von RDF Schema weist eine gewisse Ähnlichkeit mit Typsystemen objektorientierter Programmiersprachen, wie z.B. Java, auf. Trotzdem gibt es bedeutende Unterschiede, welche in diesem Abschnitt diskutiert werden.

Eine objektorientierte Programmiersprache würde eine Klasse *Mitarbeiter* definieren, welche über ein Attribut *arbeitet_für* verfügt, dessen Wert vom Typ *Firma* ist. Das Attribut ist dabei Teil der Klassenbeschreibung und lässt sich auch nur auf Instanzen der Klasse *Mitarbeiter* anwenden. Enthält eine andere Klasse ein Attribut gleichen Namens, werden beide als unterschiedliche Attribute angesehen. Im Gegensatz dazu, werde RDF Schema Klassen unabhängig von Eigenschaften (Attributen) definiert. Die Beschreibung der Eigenschaften erfolgt in separaten Aussagen (vgl. Beispiel 8), wobei die Angabe, auf welche Klasse die Property angewendet werden kann, optional ist. Wird der Definitionsbereich nicht durch *rdfs:domain* eingeschränkt, ist er per default global.

Aus diesem Grund spricht man bei RDF Schema von einem eigenschaftszentrierten (property-centric) und bei objektorientierten Programmiersprachen von einem klassenzentrierten (class-centric) Modellierungsansatz. Der eigenschaftszentrierte Ansatz bietet auf der einen Seite den Vorteil, dass Eigenschaften leicht neu definiert und erweitert werden können, was insbesondere in einem offenen und dezentralen System von Bedeutung ist. Auf der anderen Seite muss die sinnvolle Kombination von Klassen und Eigenschaften, der so genannte Argumentrahmen, festgelegt werden. Dies geschieht durch die Angabe des Definitions- und Wertebereichs einer Property und ist nicht unproblematisch. So ist es beispielsweise mit Hilfe von RDF Schema nicht möglich,

den Wertebereich einer Property in Abhängigkeit von ihrem Definitionsbereich festzulegen. Daher ist die folgende Aussage mit RDF Schema nicht zu realisieren:
„Wenn die Property *arbeitet_für* zur Beschreibung einer Ressource der Klasse *Mitarbeiter* verwendet wird, dann ist der Wertebereich der Property eine Ressource der Klasse *Firma*, wenn hingegen *arbeitet_für* zur Beschreibung einer Ressource der Klasse *Auftragnehmer* eingesetzt wird, dann ist der Wertebereich der Property eine Ressource der Klasse *Auftraggeber*".
Die verschiedenen Modellierungsansätze wirken sich auch auf die Vererbung von Eigenschaften zwischen Unter- und Oberklassen aus. Eine in Java definierte Unterklasse erbt alle Eigenschaften der Oberklasse. Im Gegensatz dazu werden in RDF Schema, durch die Verwendung der *subClassOf* Property, keine Eigenschaften zwischen Unter- und Oberklasse vererbt.
Darüber hinaus kann eine in Java definierte Unterklasse nur von einer Oberklasse abgeleitet werden, da es eine Mehrfachvererbung im Sinne von C++ in Java nicht gibt [Middendorf, 1999]. Im Unterschied dazu kann eine Klasse in RDF Schema mehrere Oberklassen haben.

3.3.2 Interpretation der Typdeklarationen von RDF Schema und objektorientierten Programmiersprachen

Die im vorangegangenen Abschnitt aufgezeigten Unterschiede der Typsysteme beeinflussen die Interpretation ihrer Typdeklarationen. Wird beispielsweise in einer objektorientierten Programmiersprache eine Klasse *Mitarbeiter* mit dem Attribut *arbeitet_für* und einem Wert vom Typ *Firma* deklariert, handelt es sich dabei um eine Gruppe von Constraints. Dies bedeutet, dass die Interpretation vorgeschrieben ist, es also keine Möglichkeit gibt, eine Instanz der Klasse *Mitarbeiter* zu erzeugen, welcher das Attribut fehlt oder dessen Wert nicht vom Typ *Firma* ist. Im Gegensatz dazu wird durch die Deklaration des Definitions- und Wertebereichs bei RDF Schema nicht festgelegt, wie dieser von Anwendungen zu interpretieren ist.
Beschreibt ein Schema beispielsweise die Aussage, dass eine *arbeitet_für* Property über eine *rdfs:range* der Klasse *Firma* verfügt, hängt die Interpretation von der Anwendung ab.

- Eine Anwendung verwendet das Schema, um sicherzustellen, dass jede *arbeitet_für* Property ausschließlich Werte annehmen kann, die Instanzen der Klasse *Firma* sind. In diesem Fall wird die Deklaration, wie in objektorientierten Programmiersprachen, als Constraint interpretiert.
- In einer anderen Anwendung ist die Property *arbeitet_für* ohne die Angabe des Wertebereichs deklariert. Das Schema wird zur Gewinnung zusätzlicher Informationen herangezogen und gefolgert, dass die Ressource eine Instanz der Klasse *Firma* sein muss.
- Eine weitere Anwendung hat den Wertebereich der Property auf Instanzen der Klasse *Auftraggeber* festgelegt und verwendet das Schema, um auf eine mögliche Inkonsistenz hinzuweisen.

Dieses Beispiel soll einen fundamentalen Unterschied zwischen einer Programmiersprache und RDF Schema verdeutlichen. Eine Programmiersprache hat in jedem Fall eine formale Semantik, zumindest wenn sie kompiliert und dadurch verifiziert werden konnte. RDFS stellt lediglich in begrenztem Umfang Informationen über die Interpretation von RDF Aussagen zur Verfügung. Die Frage, wie diese zu interpretieren sind, ob z.B. eine Inkonsistenz auftritt, muss durch Anwendungen beantwortet werden.

3.3.3 Fazit

Auf der Basis von RDF Schema lassen sich Klassen und Eigenschaften definieren und in eine Hierarchie einordnen. Außerdem können über Constraint-Properties semantische Restriktionen für die Verwendung und Kombination von Klassen und Eigenschaften formuliert werden. Mit RDF Schema lassen sich also einfache Ontologien beschreiben. Der wichtigste Faktor, den RDF/S für eine maschinelle Interpretation von Ressourcenbeschreibungen zur Verfügung stellt, ist die Möglichkeit der Klassifikation. Durch die Transitivität der *subClassOf* Eigenschaft lassen sich Konzepte verfeinern und teilweise wiederverwenden. Da RDFS Mehrfachklassifikation unterstützt, existieren unter Umständen verschiedene Alternativen, um maschinell relevante Informationen zu extrahieren.

Die Einordnung von RDF/S innerhalb des Schichtenmodell verdeutlicht, dass es sich um eine Basistechnologie zum Erreichen der semantischen Interoperabilität im Sinne eines „kleinsten gemeinsamen Nenners“ handelt. Deshalb reichen die von RDF Schema zur Verfügung gestellten Sprachkonstrukte nicht aus, um das Potential von Ontologien auszuschöpfen. Es lassen sich die folgenden Defizite identifizieren, welche für eine maschinelle Interpretation von Ressourcenbeschreibungen von Bedeutung sind.

- Es besteht keine Möglichkeit die Kardinalität von Eigenschaften auszudrücken, weshalb beispielsweise nicht repräsentiert werden kann, dass eine Person exakt einen biologischen Vater hat.
- Es ist mit Hilfe von RDF Schema nicht möglich, die Kardinalität einer Property in Abhängigkeit von ihrem Definitionsbereich festzulegen. Beispielsweise wäre es sinnvoll, wenn die Property *hat_spieler* in der Klasse *Fußballmannschaft* den Wert *elf* annehmen könnte und in einer Klasse *Basketballmannschaft* den Wert *fünf*.
- Es kann nicht festgelegt werden, dass zwei unterschiedliche Klassen mit unterschiedlichen URI´s ein und dieselbe Klasse beschreiben. (Entsprechendes gilt für Instanzen).
- RDFS fehlt die Möglichkeit, neue Klassen auf Basis der grundlegenden Mengenoperationen (Vereinigung-, Schnitt-, und Komplementmenge) zu beschreiben oder auszudrücken, dass zwei Klassen disjunkt sind.
- Obwohl man mit dem RDF Datenmodell Aussagen über Aussagen formulieren kann, was mit der Prädikatenlogik erster Stufe nicht möglich ist, fehlt die aus der Prädikatenlogik bekannte Existenz- und Allquantifizierung. Daher kann keine Implikation ausgedrückt, also der Wertebereich einer Property nicht in Abhängigkeit von ihrem Definitionsbereich festgelegt werden.

Die oben aufgeführten Defizite werden durch die Web Ontology Language beseitigt, indem zusätzliche vordefinierte Sprachkonstrukte eingeführt werden.

3.4 Web Ontology Language (OWL)

Historisch betrachtet ist OWL eine Weiterentwicklung der Sprache DAML+OIL [W3C-DAML+OIL]. Dabei steht DAML für die „DARPA Agent Markup Language“, welche aus einem Projekt des amerikanischen Verteidigungsministeriums entstanden ist und OIL für “Ontology Inference Layer”, einem Projekt der Europäischen Union. OWL ist das Ergebnis der Bemühungen, eine einheitliche Web-Ontologiesprache zu entwickeln.
Grundsätzlich ist das Datenmodell von OWL syntaxneutral. Es hat sich jedoch die RDF/XML Syntax zum Austausch zwischen verteilten Systemen etabliert.

Die Motivation bei der Entwicklung von OWL [W3C-OWL-Language-Guide, 2004] bestand auf der einen Seite darin, eine Sprache mit möglichst hoher Ausdruckskraft zur Verfügung zu stellen und auf der anderen Seite, effiziente Inferenzmechanismen nutzen zu können. Diesem Zielkonflikt wird dadurch begegnet, dass die Sprache OWL in drei Varianten vorliegt und es Anwendern dadurch möglich ist, die für ihre Bedürfnisse geeignete Variante auszuwählen.

3.4.1 OWL Varianten

Die Web Ontology Language besteht aus *OWL Lite*, *OWL DL* und *OWL Full*, welche eine zunehmende Aussagekraft zur Verfügung stellen.

- *OWL Lite* stellt eine Untermenge der Sprachkonstrukte von OWL Full zur Verfügung und ist als Migrationspfad für Thesauri und Taxonomien gedacht. Im Vordergrund steht die Möglichkeit, Klassenhierarchien und einfache Restriktionen zu repräsentieren. Neben der Anzahl der Sprachkonstrukte ist auch deren Anwendung eingeschränkt, weshalb beispielsweise die Kardinalität einer Property nur die Werte Null oder Eins annimmt.

- *OWL DL* (*Description Logics*) stellt sämtliche Sprachkonstrukte von OWL Full zur Verfügung, wobei deren Anwendung, im Hinblick auf die Möglichkeit effiziente Inferenzmechanismen einsetzen zu können, eingeschränkt ist. Beispielsweise darf eine Klasse nicht gleichzeitig eine Instanz oder eine

Property einer anderen Klasse sein. Mit anderen Worten, die Typen müssen klar voneinander getrennt werden. Wie der Name OWL DL andeutet, lassen sich Ontologien auf Beschreibungslogiken abbilden und mit Hilfe von Inferenzsystemen, wie z.B. RACER, maschinell interpretieren. Aus diesem Grund verfügt OWL DL über eine formale Semantik und ist vollständig entscheidbar.

- *OWL Full* stellt die gleichen Sprachkonstrukte wie OWL DL zur Verfügung. Der Unterschied besteht jedoch darin, dass es keinerlei Einschränkungen für deren Anwendung gibt. Es existiert keine klare Trennung zwischen den Typen (Klasse, Instanz und Property), weshalb es beispielsweise möglich ist, dass eine Klasse gleichzeitig eine Instanz von sich selber ist. Darüber hinaus ist es in OWL Full erlaubt, die vordefinierten Sprachkonstrukte (z.B. *subClassOf*) zu erweitern und dadurch deren Bedeutung zu verändern. Die Variante OWL Full verfügt also über eine mit RDF vergleichbare maximale Ausdruckskraft, was den Nachteil mit sich bringt, dass sie im Gegensatz zu OWL DL nicht vollständig entscheidbar ist.

In Bezug auf die Zugehörigkeit einer Ontologie zu einer der drei Varianten und der Gültigkeit von Schlussfolgerungen, gelten die folgenden Beziehungen:

- Jede legale OWL Lite Ontologie ist eine legale OWL DL Ontologie.
- Jede legale OWL DL Ontologie ist eine legale OWL Full Ontologie.
- Jede gültige OWL Lite Folgerung ist eine gültige OWL DL Folgerung.
- Jede gültige OWL DL Folgerung ist eine gültige OWL Full Folgerung.

3.4.2 OWL Sprachkonstrukte

Neben den bereits aufgeführten Sprachkonstrukten von RDF/S werden in diesem Abschnitt die durch OWL zusätzlich eingeführten Sprachkonstrukte erläutert.

3.4.2.1 **Charakteristiken von Eigenschaften**

Im folgenden werden die durch OWL eingeführten Charakteristiken von Objekt-Eigenschaften vorgestellt. Object-Properties sind binäre Relationen zwischen Instanzen von zwei Klassen, wobei zwischen den Begriffen Instanz und Individuum in OWL nicht unterschieden wird. Mit Charakteristiken wird das Ziel verfolgt, Eigenschaften genauer zu spezifizieren und logische Ableitungen vornehmen zu können.

TransitiveProperty: Wenn eine Property P als transitiv gekennzeichnet ist und die Paare (x,y) und (y,z) Instanzen der Eigenschaft P sind, dann kann abgleitet werden, dass auch das Paar (x,z) eine Instanz von P ist.

SymmetricProperty: Ist das Paar (x,y) Instanz einer symmetrischen Property P, dann ist auch das Paar (y,x) Instanz dieser Eigenschaft.

FunctionalProperty: Wenn eine Property P als funktional gekennzeichnet ist und die Paare (x,y) und (x,z) Instanzen der Eigenschaft P sind, wird dadurch impliziert, dass y gleich z ist. Mit anderen Worten legt eine funktionale Property eine minimale Kardinalität von Null und eine maximale Kardinalität von eins für eine Eigenschaft fest. Die folgende Beispiel besagt, dass eine Frau höchstens einen Ehemann haben kann, was nicht bedeutet, dass jede Frau einen Mann als Ehemann haben muss.

```
<owl:ObjectProperty rdf:ID="hat_ehemann">
        <rdf:type rdf:resource="&owl;FunctionalProperty" />
        <rdfs:domain rdf:resource="#Frau" />
        <rdfs:range rdf:resource="#Mann" />
</owl:ObjectProperty>
```

inverseOf: Wird eine Eigenschaft P1 als inverse Relation zu der Eigenschaft P2 definiert und festgelegt, dass x mit y durch P1 verknüpft ist, dann ist y mit x durch P2 verknüpft. Da dies auch umgekehrt gilt, ist *inverseOf* eine *SymmetricProperty*. Wird beispielsweise die Eigenschaft *ist_ehemann_von* als inverse Relation zu der Eigenschaft *hat_ehemann* definiert und festgelegt, dass Fritz

ist_ehemann_von Maria gilt, kann daraus Maria *hat_ehemann* Fritz abgeleitet werden.

InverseFunctionalProperty: Die Eigenschaft invers funktional ist die Umkehrung einer FunctionalProperty, weshalb die Kardinalität ihrer Eigenschaft maximal eins sein darf. Das durch *rdfs:range* angegebene Objekt der Property ist mit dem eindeutigen Primärschlüssel einer Datenbank vergleichbar. Bei der *InverseFunctionalProperty* wird also durch das Objekt einer Aussage das Subjekt eindeutig festgelegt.

```
<owl:InverseFunctionalProperty rdf:ID="biologische_Vater_von">
        <rdfs:domain rdf:resource="#Mann"/>x
        <rdfs:range rdf:resource="#Mensch"/>y
</owl:InverseFunctionalProperty>
```

Dieses Beispiel repräsentiert den Sachverhalt, dass jedem Menschen nur eine eindeutige Instanz der Klasse Mann als biologischer Vater zugewiesen werden kann. Besitzen zwei verschiedene Instanzen der Klasse Mann die Eigenschaft biologischer Vater eines Menschen zu sein, kann abgeleitet werden, dass es sich bei beiden Instanzen um den selben Mann handeln muss.

Bei den Eigenschaften *FunctionalProperty* und *InverseFunctionalProperty* handelt es sich um global geltende Einschränkungen der Kardinalität. Es ist daher unerheblich, welchen Klassen die Eigenschaften zugewiesen werden, denn die Einschränkungen müssen, im Gegensatz zu den im kommenden Abschnitt vorgestellten Property Restriktionen, überall gelten.

3.4.2.2 **Einschränkungen von Eigenschaften**

Eine Propery Restriktion stellt eine spezielle Art von Klassenbeschreibung dar. Durch sie kann angegeben werden, wie Eigenschaften durch Instanzen einer Klasse verwendet werden können. Die Einschränkung wird dabei innerhalb von *owl:Restriction*-Elementen formuliert und die *owl:onProperty* zeigt an, um welche Eigenschaft es geht. Man unterscheidet zwischen Einschränkungen des

lokalen Wertebereichs und der lokalen Kardinalität, also welche und wie viele Werte eingesetzt werden können.

Um anzugeben welche Eigenschaften durch Instanzen einer Klasse eingesetzt werden können, stehen die beiden Sprachkonstrukte *allValuesFrom* und *someValuesFrom* zur Verfügung.
Das folgende Beispiel aus der OWL-Referenz zeigt die Verwendung der *allValuesFrom* Restriktion.

```
<owl:Class rdf:ID="Wein">
        <rdfs:subClassOf>
                <owl:Restriction>
                        <owl:onProperty rdf:resource="#hat_Hersteller" />
                        <owl:allValuesFrom rdf:resource="#Weingut" />
                </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>
```

Die Klasse Wein hat die Eigenschaft *hat_Hersteller*, deren Wertebereich durch *allValuesFrom* auf die Klasse Weingut eingeschränkt ist. Es wird ausgesagt, dass für alle Weine, die einen Hersteller haben, dieser ein Weingut sein muss. Daraus lässt sich jedoch nicht ableiten, das die Property *hat_Hersteller* mindestens einen Wert vom Typ Weingut haben muss, was die Semantik des Wortes *allValues* nahe legen könnte.
Ersetzt man im obigen Beispiel *allValuesFrom* durch die Restriktion *someValuesFrom*, wird ausgesagt, dass alle Weine, mindestens einen Hersteller haben müssen, welcher ein Weingut ist, es aber Hersteller geben kann, die nicht vom Typ Weingut sind. Daraus lässt sich ableiten, das die Property *hat_Hersteller* mindestens einen Wert vom Typ Weingut haben muss, aber nicht, dass alle Hersteller vom Typ Weingut sein müssen. Mit anderen Worten, es brauchen nicht alle Werte einer Eigenschaft Instanzen der gleichen Klasse sein, um diese Restriktion zu erfüllen.

Durch die *hasValue* Restriktion kann eine Klasse dadurch definiert werden, dass eine Property mindestens einen bestimmten Wert hat. Bei dem Wert

handelt es sich im Gegensatz zu *allValuesFrom* und *someValuesFrom* um eine Instanz und keine Klassen. Im folgenden Beispiel wird definiert, dass ein *News-Service* ein Service ist, bei dem der Wert seiner *wird_realisiert_durch* Eigenschaft ein *Network_News_Transfer_Protocol* sein muss, welches Instanz der Klasse *News-Service* ist.

```
<owl:Class rdf:ID="News-Service">
        <rdfs:subClassOf rdf:resource="#Service"/>
        <rdfs:subClassOf>
                <owl:Restriction>
                        <owl:onProperty rdf:resource="#wird_realisiert_durch"/>
                        <owl:hasValue rdf:resource="#Network_News_Transfer_Protocol"/>
                </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>
```

In der Variante OWL Lite kann die *hasValue* Restriktion nicht eingesetzt werden.

Um die Kardinalität von Eigenschaften einzuschränken, stehen die Sprachkonstrukte *minCardinality*, *maxCardinality* und *Cardinality* zur Verfügung. Das folgende Beispiel repräsentiert den Sachverhalt, dass jeder Weinjahrgang nur genau ein Lesejahr hat.

```
<owl:Class rdf:ID="Weinjahrgang">
        <rdfs:subClassOf>
        <owl:Restriction>
                <owl:onProperty rdf:resource="#hat_ein_Weinlese_Jahr"/>
                <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
                </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>
```

Die Festlegung der Kardinalität auf einen numerischen Wert ist eine verkürzte Schreibweise für ein Intervall bestehend aus unterer (*minCardinality*) und oberer (*maxCardinality*) Grenze. Wie bereits erwähnt, können in OWL Lite keine beliebigen positiven Werte für die Kardinalität angegeben werden, sondern nur

0 und 1. Dadurch kann ausgedrückt werden, dass Instanzen der Klasse „genau einen“, „mindestens einen“ und „nicht mehr als einen“ Wert für die Eigenschaft haben müssen.

3.4.2.3 **Komplexe Klassen**

Neben der Möglichkeit, Klassen durch Propery Restriktionen zu beschreiben, stellt OWL die grundlegenden Mengenoperatoren *intersectionOf*, *unionOf* und *complementOf* zur Verfügung. Darüber hinaus können die Instanzen einer Klasse explizit aufgezählt werden (*oneOf*). Außerdem ist die Aussage möglich, dass die Instanz einer Klasse nicht gleichzeitig Instanz einer anderen Klasse sein kann (*disjointWith*). Da diese Sprachkonstrukte in der Variante OWL Lite gar nicht oder nur eingeschränkt zur Verfügung stehen, sind die folgenden Beispiele nur mit den Varianten DL und Full zu realisieren.

Das Beispiel definiert die Klasse Weißwein als Schnittmenge, wodurch abgeleitet werden kann, dass wenn etwas die Farbe weiß hat und ein Wein ist, es eine Instanz der Klasse Weißwein sein muss.

```
<owl:Class rdf:ID="Weißwein">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Wein" />
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hatFarbe" />
            <owl:hasValue rdf:resource="#Weiß" />
        </owl:Restriction>
    </owl:intersectionOf>
</owl:Class>
```

Durch den *unionOf* Operator kann eine Klasse als Vereinigungsmenge von anderen Klassen repräsentiert werden.

```
<owl:Class rdf:about ="#Person">
    <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about ="#Mann"/>
        <owl:Class rdf:about ="#Frau"/>
    </owl:unionOf>
</owl:Class>
```

Die Komplementmenge repräsentiert alle Instanzen einer Klasse, welche nicht zu einer bestimmten Klasse gehören.

```
<owl:Class rdf:ID="NichtFranzösischerWein">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Wein"/>
        <owl:Class>
            <owl:complementOf>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="#angebaut_in" />
                    <owl:hasValue rdf:resource="#Frankreich" />
                </owl:Restriction>
            </owl:complementOf>
        </owl:Class>
    </owl:intersectionOf>
</owl:Class>
```

Das Beispiel definiert die Klasse *NichtFranzösischerWein* als die Schnittmenge von Weinen mit der Menge aller Dinge, welche nicht in Frankreich angebaut werden.

Sollen die Instanzen einer Klasse nicht durch Mengenoperationen beschrieben werden, können mit Hilfe von *oneOf* die Instanzen aufgezählt werden.

```
<owl:Class rdf:ID="Weinfarbe">
    <rdfs:subClassOf rdf:resource="#WeinBeschreibung"/>
    <owl:oneOf rdf:parseType="Collection">
        <owl:Weinfarbe rdf:about="#Weiß"/>
        <owl:Weinfarbe rdf:about="#Rose"/>
        <owl:Weinfarbe rdf:about="#Rot"/>
    </owl:oneOf>
</owl:Class>
```

Neben den aufgezählten Instanzen kann es keine weiteren gültigen Weinfarben geben.
Der Operator *disjointWith* repräsentiert die Unterschiedlichkeit einer Menge von Klassen. Dadurch lässt sich festlegen, dass eine Instanz einer Klasse nicht gleichzeitig Instanz einer anderen Klassen sein kann.

```
<owl:Class rdf:ID="Teigwaren">
        <rdfs:subClassOf rdf:resource="#Essbare_Dinge"/>
        <owl:disjointWith rdf:resource="#Fleisch"/>
        <owl:disjointWith rdf:resource="#Fisch"/>
        <owl:disjointWith rdf:resource="#Früchte"/>
</owl:Class>
```

In diesem Beispiel wird ausgesagt, dass *Teigwaren* disjunkt mit allen anderen Klassen sind. Diese Definition besagt jedoch nicht, dass die Klassen Fleisch, Fisch und Früchte disjunkt sind. Um dies zu erreichen, muss der *disjointWith* Operator für jedes dieser Klassenpaare definiert werden.

3.4.2.4 **Mapping von Ontologien**

Das Potenzial von Ontologien lässt sich erst durch die gemeinsame Nutzung und Wiederverwendung von verteilt entwickelten Ontologien erschließen.
Zum Importieren einer anderen Ontologie wird das Sprachkonstrukt *owl:imports* mit dem Argument *rdf:resource* im Header einer OWL Datei deklariert. Im Gegensatz zur Deklaration von Namespaces (*xmlns:*), mit denen nur auf Namen aus anderen Ontologien referenziert werden kann, inkludiert *owl:imports* die gesamte Menge von Aussagen aus einer anderen Ontologie.

Zentraler Aspekt bei der Abbildung einer Ontologie auf eine andere ist die Integration von Klassen und Eigenschaften. Dazu stellt OWL Sprachkonstrukte zur Verfügung, welche die Äquivalenz zwischen Klassen und Properties, sowie zwischen Instanzen ausdrücken können.

Durch die Property *equivalentClass* kann angegeben werden, dass zwei Klassen exakt dieselben Instanzen besitzen.

```
<owl:Class rdf:ID="Angestellter">
        <owl:equivalentClass rdf:resource="Beschäftigter"/>
</owl:Class>
```

Das folgende Beispiel zeigt die Verwendung von *equivalentClass* in Zusammenhang mit einer Restriction.

```
<owl:Class rdf:ID="Made_in_Germany">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#made_in" />
            <owl:someValuesFrom rdf:resource="#Germany" />
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Die Klasse *Made_in_Germany* beschreibt exakt die Dinge, welche in Deutschland gefertigt wurden. An dieser Stelle soll auf die Unterschiede bei der Verwendung der Relationen *equivalentClass* und *subclass* eingegangen werden [W3C-OWL-Language-Guide, 2004a]. Wird im obigen Beispiel anstelle von *equivalentClass subclass* eingesetzt, handelt es sich um eine „hinreichende aber nicht notwendige“ Bedingung. Dinge, welche in Deutschland gefertigt wurden, müssen dann nicht notwendigerweise Instanzen der Klasse *Made_in_Germany* sein. Im Gegensatz dazu handelt es sich bei *equivalentClass* um eine „notwendig und hinreichende Bedingung“. Dinge, welche in Deutschland gefertigt wurden, müssen Instanzen der Klasse *Made_in_Germany* sein.

Relation	Implikation
rdfs:subclass („hinreichend aber nicht notwendig“)	Made_in_Germany (x) impliziert made_in(x,y) und Germany y)
owl:equivalentClass („notwendig und hinreichend“)	Made_in_Germany(x) impliziert made_in(x,y) und Germany(y) made_in(x,y) und Germany(y) impliziert Made_in_Germany(x)

Tabelle 3: Die Begriffe "notwendig" und "hinreichend" und deren Implikationen

Durch die *equivalentProperty* kann angegeben werden, dass zwei Properties äquivalent sind, wodurch sich beispielsweise synonyme Properties repräsentieren lassen.

Das Sprachkonstrukt *sameA*s kann zur Gleichsetzung von zwei Instanzen eingesetzt werden, die in unterschiedlichen Ontologien definiert wurden.

```
<Person rdf:ID="Amtierender_Bundespräsident"/>
<Person rdf:ID="Horst_Köhler">
    <owl:sameAs rdf:resource="#Amtierender_Bundespräsident"/>
</Person>
```

Wie das Beispiel zeigt, können in OWL verschiedene Namen auf die gleiche Instanz verweisen. Um das Gegenteil auszudrücken, also dass zwei Instanzen unterschiedlich voneinander sind, wird *differentFrom* verwendet.

Soll definiert werden, dass eine Menge von Instanzen paarweise unterschiedlich ist, stellt *AllDifferent* in Verbindung mit *distinctMembers* dazu einen komfortablen Mechanismus zur Verfügung.

```
<owl:AllDifferent>
    <owl:distinctMembers rdf:parseType="Collection">
        <Jahreszeit rdf:about="#Frühling">
        <Jahreszeit rdf:about="#Sommer">
        <Jahreszeit rdf:about="#Herbst">
        <Jahreszeit rdf:about="#Winter">
    </owl:distinctMembers>
</owl:AllDifferent>
```

3.5 Der Ontologieeditor Protégé 3.0

Protégé [Protégé, 2005] ist ein frei erhältlicher Open Source Ontologieeditor auf der Basis von Java und daher unabhängig von der Plattform. Hauptvorteil gegenüber anderen Editoren, wie beispielsweise [OilEd, 2003], ist seine Plugin Architektur. Daher existieren zahlreiche Erweiterungen zum Erstellen und Verarbeiten von Ontologien in verschiedenen Repräsentationssprachen (RDF/S, DAML+OIL, OWL). Eine Wissensbasis besteht aus Klassen, Properties (Slots), Individuen, sowie logischen Regeln und Restriktionen. Zur Verarbeitung besteht die Möglichkeit der Visualisierung und in begrenztem Umfang auch der Konvertierung zwischen Repräsentationssprachen. Darüber hinaus können In-

formationen durch SQL-ähnliche Suchfunktionen abgefragt und auf Basis des integrierbaren Inferenzsystems RACER abgeleitet werden.
Der Ontologieeditor Protégé hat im Laufe seiner Entwicklung deutlich an Stabilität und Funktionsumfang gewonnen, was ihn insgesamt zu einem interessanten Werkzeug zur Modellierung von Ontologien macht.

4 Prototypische Implementierung

Nachdem in den bisherigen Kapiteln die grundlegenden Technologien und Sprachen des Semantic Web vorgestellt wurden, soll im praktischen Teil der Arbeit die Generierung der Beispielontologie „Postalische Anschrift“ diskutiert werden.

Gegenstand des Beispiels sind Adressdaten einer Postanschrift, welche im WWW häufig innerhalb eines HTML-Dokumentes kodiert werden. Da es sich bei diesen Daten meist um eine beziehungslose Ansammlung von Zeichenketten handelt, besteht das Ziel der prototypischen Implementierung darin, die Beziehungen und Konzepte zwischen Adressdaten einer Anschrift durch eine Ontologie zu repräsentieren. Dazu müssen die Daten zunächst in das folgende Webformular eingegeben werden.

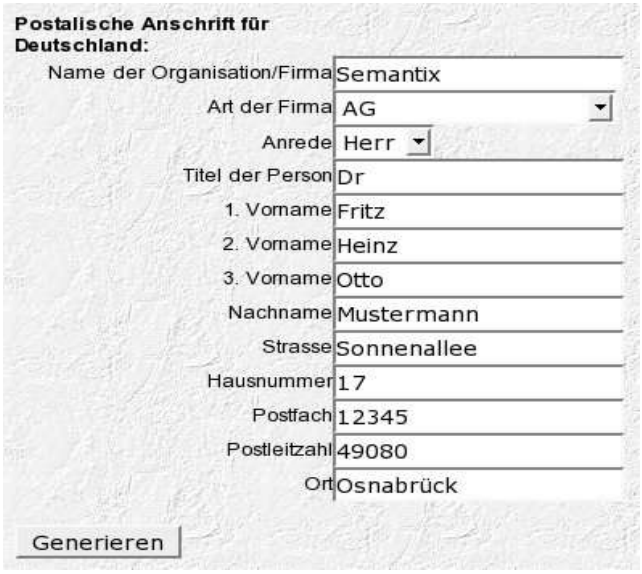


Abbildung 10: Webformular zur Eingabe der Adressdaten unter:
http://marc-schilling.de:8080/post_ont_servlet/Servlet1.shtml

Betrachtet man die Domäne einer deutschen Postanschrift, dann ist der Adressat entweder eine Organisation, eine Firma oder eine Person, was anhand der Eingaben durch ein Servlet entschieden werden kann. Wird beispielsweise ein „Name der Organisation/Firma“ eingegeben und keine „Art der Firma“, handelt es sich beim Adressaten um eine Organisation, da jede Firma eine Gesell-

schaftsform (z.B. „AG“) haben muss. Eine Person wird anhand ihres Nachnamens identifiziert und kann in Kombination mit einer Organisation/Firma angegeben werden. Dies wird als „zu Händen“ interpretiert, wodurch es sich bei der Person um einen Mitarbeiter handeln muss.
Nach der Eingabe der Daten prüft das Servlet, ob es sich um eine formal gültige Postanschrift handelt und kodiert die entsprechende Ontologie. Dazu ist es notwendig eine plausible Kombination von Feldern mit korrekten Datentypen einzugeben. Beispielsweise führt die Eingabe einer Hausnummer ohne Straßenangabe oder eines Nachnamen mit Zahlen zu einer Fehlermeldung. Neben der formalen Prüfung wird eine datenbankgestützte Geschlechtsbestimmung anhand des Vornamens und ein Abgleich der Postleitzahl mit dem Ort vorgenommen. Die folgende Abbildung 11 zeigt das Resultat der Formulareingabe. Anhand des ersten Vornamens konnte das Geschlecht als männlich bestimmt werden. Da auch der eingegebene Ort mit den möglichen Postleitzahlen übereinstimmt, ist das Formular korrekt ausgefüllt.

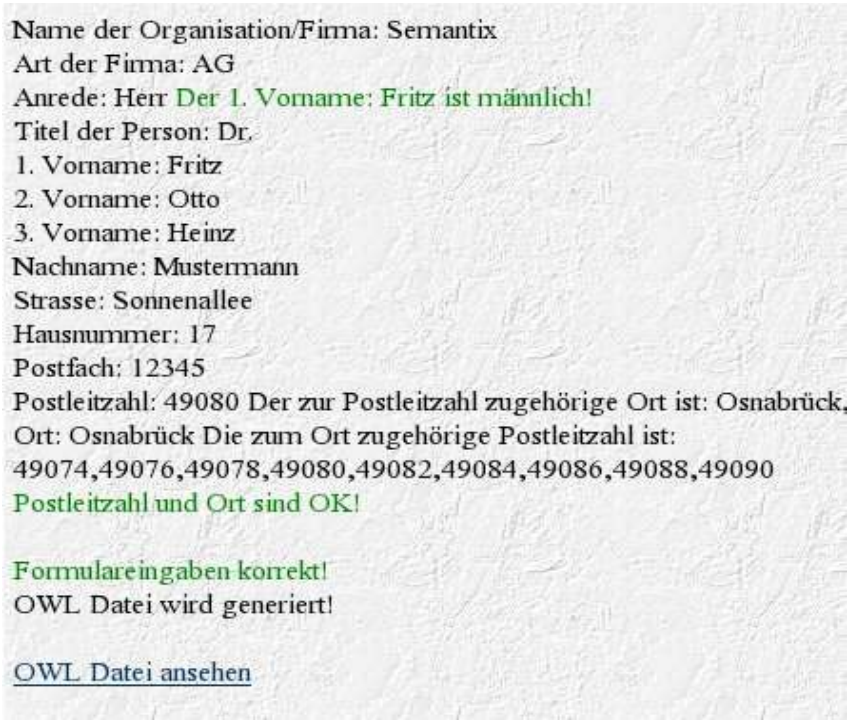


Abbildung 11: Resultat der im Webformular gemachten Eingaben

Aus den überprüften Daten und der gewonnenen Information, dass der erste Vorname männlich ist, wird eine Ontologie in Form einer OWL Datei mit dem

Namen „*post_ont.owl*“ erzeugt. Um die innerhalb der Datei repräsentierten Beziehungen und Konzepte, sowie deren eingegebene Instanzen abzufragen, muss sie durch ein Inferenzsystem geladen werden.

4.1 Verwendete Technologien

Die Implementierung basiert auf der objektorientierten und plattformunabhängigen Java Technologie. Da die Anwendung serverseitig ablaufen soll, kommt ein Java Servlet zum Einsatz, welches die im Webformular eingegebenen Daten verarbeitet und daraus die Ontologie erzeugt. Als Servlet-Container fungiert der Jakarta Tomcat Applikationsserver in der Version 4.1.18. Tomcat ist eine Entwicklung des Apache Projekts und erweitert die Funktionalität des gleichnamigen Webservers um die Möglichkeit, Java Anwendungen im Web zur Verfügung zu stellen. Bei der Programmierung des Servlets kam die Entwicklungsumgebung Borland JBuilder 5 Professional zum Einsatz. Um eine Verbindung des Servlets mit der Open Source Datenbank MySQL herzustellen, wird der „Java-Database-Connectivity“ Treiber verwendet. Dadurch kann das Servlet mit standardisierten SQL Anfragen auf die Datenbank zugreifen.

4.2 Eingesetzte Datenbanken

Die Geschlechtsbestimmung anhand des Vornamens basiert auf zwei Textdateien [Textarchiv 7] mit jeweils männlichen und weiblichen Vornamen, welche als Datensätze in eine MySQL Datenbank integriert wurden. Da es sich dabei um internationale Vornamen handelt, kann in einigen Fällen (z.B. Jean) eine exakte Bestimmung des Geschlechts nicht vorgenommen werden.
Zur Kontrolle der eingegebenen Postleitzahl und des Ortes wurde auf die Datenbank [OpenGeoDB] (Version 0.1.1) des gleichnamigen Projektes zurückgegriffen. Im Mittelpunkt des Projektes steht der Aufbau einer möglichst vollständigen Datenbank mit Postleitzahlen, Orten und Geokoordinaten für den deutschsprachigen Raum. Bei der Überprüfung der Eingabe musste beachtet werden, dass eine Postleitzahl verschiedenen Orten zugeordnet sein kann, was auch umgekehrt gilt.

4.3 Modellierung der Ontologie

Aufgrund der im Vergleich zu RDF/S erweiterten Sprachkonstrukte und Inferenzmöglichkeiten wird als Repräsentationssprache OWL in der Variante DL eingesetzt. Die folgende Beschreibung der durch das Servlet generierten Ontologie basiert auf einem vollständig ausgefüllten Eingabeformular (vgl. Abbildung 10). Mit Hilfe des [OWL-Ontology Validator, 2003] wurde überprüft, ob sie gültig im Sinne der Variante DL ist. Zunächst werden im Header der Ontologiedatei die Namensräume definiert.

```
1:    ?xml version="1.0"?>
2:    <rdf:RDF
3:          xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:          xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
6:          xmlns:owl="http://www.w3.org/2002/07/owl#"
7:          xmlns="http://marc-schilling.de/sw/post_ont.owl#"
8:          xml:base="http://marc-schilling.de/sw/post_ont.owl">
9:          <owl:Ontology rdf:about=""/>
```

In Zeile 8 wird ein Wert für die URI-Basis und damit der Name der Ontologie festgelegt, wodurch auf die Elemente der Ontologie über die in Zeile 7 angegebene Definition des Namensraums zugegriffen werden kann. Hätte man keine URI-Basis festgelegt, sondern den Namen innerhalb des *owl:Ontology* Tags in Zeile 9 angegeben, wäre die URI-Basis implizit der Speicherort der Ontologiedatei gewesen, was den späteren Zugriff erschwert hätte.

Danach werden drei Instanzen der Klasse *Postalische Anschrift* erzeugt.

```
<owl:Class rdf:ID="Postalische_Anschrift"/>

<Postalische_Anschrift rdf:ID="Anschrift_Firma"/>
<Postalische_Anschrift rdf:ID="Anschrift_Organisation"/>
<Postalische_Anschrift rdf:ID="Anschrift_Person"/>
```

Die Entscheidung, eine Instanz *Anschrift_Firma* zu erzeugen und keine Unterklasse von *Postalischer_Anschrift*, wurde getroffen, weil die Instanz später le-

diglich als Wert der Eigenschaft *ist_adressat* verwendet wird. Eine Modellierung als Unterklasse wäre nur dann sinnvoll gewesen, wenn dadurch zusätzliche Eigenschaften oder Einschränkungen eingeführt worden wären, über welche die Oberklasse nicht verfügt [Noy, 2001].

Durch die Syntax *rdf:ID="ist_adressat"* wird in Zeile 2 der Name der Eigenschaftsdefinition angegeben. Innerhalb der Ontologie könnte nun auf die Eigenschaft durch *rdf:resource="#ist_adressat"* referenziert werden. Die Erweiterung der Eigenschaftsdefinition geschieht über eine Referenz der Form *rdf:about"#ist_adressat"* in den Zeilen 3 bis 5. Dort wird mit Hilfe der *rdfs:domain* Property festgelegt, dass jede Instanz, welche die Eigenschaft verwendet, Instanz der in Zeile 1 definierten Klasse *Adressat* sein muss. In der generierten Ontologie wird *ist_adressat* später einer Instanz der Klasse Firma zugewiesen.

```
1:  <owl:Class rdf:ID="Adressat"/>
2:  <owl:ObjectProperty rdf:ID="ist_adressat"/>

3:  <owl:ObjectProperty rdf:about="#ist_adressat">
4:      <rdfs:domain rdf:resource="#Adressat"/>
5:  </owl:ObjectProperty>
```

Nach [Horridge, 2004] ist beim Einsatz der *rdfs:domain* Property in OWL ein wichtiger Punkt zu beachten. Ein Inferenzsystem prüft nicht, ob eine Instanz, welche die Eigenschaft verwendet, Instanz von *Adressat* ist, sondern leitet dies ab. Insbesondere bei umfangreichen Ontologien muss dieses Verhalten bei der Wahl des Kontextes der Domain- und Range-Property berücksichtigt werden.

Aus den eingegebenen Adressdaten wird nun eine einfache Klassenhierarchie aufgebaut. Das allgemeinste Konzept *Organisation* wird dabei durch die Klasse *Firma* spezialisiert und diese wiederum durch *Art_Firma*.

```
<owl:Class rdf:ID="Organisation"/>
<owl:Class rdf:ID="Firma">
    <rdfs:subClassOf rdf:resource="#Organisation"/>
</owl:Class>
```

```
<owl:Class rdf:ID="Art_Firma">
        <rdfs:subClassOf rdf:resource="#Firma"/>
</owl:Class>
```

Die Klasse *Art_Firma* wird hier als Unterklasse modelliert, da sie die Menge der verschieden Arten von Unternehmungen repräsentiert.

```
<owl:Class rdf:ID="Strasse"/>
<owl:Class rdf:ID="Postfach"/>
<owl:Class rdf:ID="Hausnummer"/>
<owl:Class rdf:ID="Postleitzahl"/>
<owl:Class rdf:ID="Ort"/>

<owl:ObjectProperty rdf:ID="hat_art_firma"/>
<owl:ObjectProperty rdf:ID="hat_strasse"/>
<owl:ObjectProperty rdf:ID="hat_postfach"/>
<owl:ObjectProperty rdf:ID="hat_hausnummer"/>
<owl:ObjectProperty rdf:ID="hat_postleitzahl"/>
<owl:ObjectProperty rdf:ID="hat_ort"/>
```

Neben den oben definierten Klassen und Eigenschaften wird die inverse Object-Property *hat_mitarbeiter* benötigt, um die eingegebene Firma mit der Person in Beziehung zu setzen.

```
<owl:ObjectProperty rdf:ID="hat_mitarbeiter">
        <owl:inverseOf>
                <owl:ObjectProperty rdf:ID="ist_mitarbeiter"/>
        </owl:inverseOf>
</owl:ObjectProperty>
```

Da es sich bei *Semantix* um ein individuelles Einzelelement der Klasse *Firma* handelt, wird dieses als Instanz modelliert und angegeben, über welche Eigenschaften sie verfügt.

```
<Firma rdf:ID="Semantix">
        <ist_adressat>
                <Postalische_Anschrift rdf:about="#Anschrift_Firma"/>
        </ist_adressat>
        <hat_art_firma>
```

```
            <Art_Firma rdf:ID="AG"/>
        </hat_art_firma>
        <!-- Innerhalb der generierten Ontologie werden an dieser Stelle Angaben zu
        Strasse, Ort, usw. auf die gleiche Weise definiert. -->
        <hat_mitarbeiter>
            <Mitarbeiter rdf:ID="Mustermann"/>
        </hat_mitarbeiter>
</Firma>
```

Durch die Verwendung der Property *ist_adressat* kann abgeleitet werden, dass *Semantix* eine Instanz der Klasse *Adressat* ist und über den Wert *Anschrift_Firma* verfügt. Die Deklaration der *AG* als *Art_Firma* führt aufgrund der Unterklassenbeziehung zu *Firma* dazu, dass auch *AG* eine Instanz von *Firma* ist. Danach wird *Mustermann* als Instanz der Klasse *Mitarbeiter* definiert und über die Relation *hat_mitarbeiter* in Verbindung mit *Semantix* gesetzt. Die eigentliche Deklaration der Klasse *Mitarbeiter* als Unterklasse von *Person* wird in den folgenden vier Zeilen vorgenommen.

```
<owl:Class rdf:ID="Person"/>
<owl:Class rdf:ID="Mitarbeiter">
        <rdfs:subClassOf rdf:resource="#Person"/>
</owl:Class>
```

Die beiden Klassen *Geschlecht* und *Anrede* werden durch die direkte Aufzählung all ihrer möglichen Instanzen spezifiziert. Obwohl es die Übersetzung des Operators *oneOf* als „eines von“ nahe legt, leitet ein Inferenzsystem keine Fehler ab, wenn *Herr* und *Frau* gleichzeitig als Wert einer Eigenschaft Verwendung finden. Später werden die Instanzen der beiden Klassen zur Klassifizierung einer männlichen oder weiblichen Person eingesetzt und es wird sichergestellt, dass eine Person nicht „beide Anreden“ oder eine falsche Kombination von Anrede und Geschlecht haben kann.

```
<owl:Class rdf:ID="Geschlecht">
        <owl:oneOf rdf:parseType="Collection">
            <Geschlecht rdf:ID="Maennlich"/>
            <Geschlecht rdf:ID="Weiblich"/>
        </owl:oneOf>
</owl:Class>
```

```
<owl:Class rdf:ID="Anrede">
	<owl:oneOf rdf:parseType="Collection">
		<Anrede rdf:ID="Herr"/>
		<Anrede rdf:ID="Frau"/>
	</owl:oneOf>
</owl:Class>
```

Eine Geschlechtsbestimmung ist aufgrund der zugrunde liegenden Datensätze nur möglich, wenn ein einzelner und kein durch einen Bindestrich getrennter Vorname, wie beispielsweise Heinz-Otto, eingegeben wurde. Aus diesem Grund werden drei Klassen und Eigenschaften für Vornamen definiert.

```
<owl:Class rdf:ID="Titel"/>
<owl:Class rdf:ID="Vorname1"/>
<owl:Class rdf:ID="Vorname2"/>
<owl:Class rdf:ID="Vorname3"/>

<owl:ObjectProperty rdf:ID="hat_anrede"/>
<owl:ObjectProperty rdf:ID="hat_titel"/>
<owl:ObjectProperty rdf:ID="hat_vorname1"/>
<owl:ObjectProperty rdf:ID="vorname1_hat_geschlecht"/>
<owl:ObjectProperty rdf:ID="hat_vorname2"/>
<owl:ObjectProperty rdf:ID="hat_vorname3"/>

<Mitarbeiter rdf:about="#Mustermann">
	<ist_mitarbeiter>
		<Firma rdf:about="#Semantix"/>
	</ist_mitarbeiter>
</Mitarbeiter>
```

Nachdem die Inverse der Object-Property und deren Wert angeben wurde, erfolgt die Definition der Instanz der Klasse Person. Das anhand des ersten Vornamens ermittelte Geschlecht *Maennlich* ist der Wert der Property *vorname1_hat_geschlecht*.

```
<Person rdf:about="#Mustermann">
	<hat_anrede>
		<Anrede rdf:ID="Herr"/>
```

```
        </hat_anrede>
        <hat_vorname1>
                <Vorname1 rdf:ID="Fritz"/>
        </hat_vorname1>
        <vorname1_hat_geschlecht>
                <Geschlecht rdf:about="#Maennlich"/>
        </vorname1_hat_geschlecht>
        <!-- Innerhalb der generierten Ontologie werden an dieser Stelle Angaben zu
        Titel und weiteren Vornamen auf die gleiche Weise definiert. -->
</Person>
```

Die Klassifizierung einer Instanz von Person als männlich oder weiblich basiert auf dem eingegebenen Merkmal Anrede und dem ermittelten Geschlecht. Für das hier diskutierte vollständig ausgefüllte Eingabeformular sind beide Merkmale vorhanden. Da dies nicht vorausgesetzt werden kann, müssen innerhalb der Klasse *Maennliche_Person* folgende Möglichkeiten berücksichtigt werden.

- Es wurde kein erster Vorname eingegeben oder das Geschlecht war nicht (eindeutig) ermittelbar, weshalb der Person die Eigenschaft *vorname1_hat_geschlecht* und damit der Wert fehlt. Wenn in diesem Fall die Anrede *Herr* existiert, wird die Person trotzdem als eine männliche Person klassifiziert.
- Das Geschlecht konnte anhand des Vornamens ermittelt werden, aber es fehlt die Anrede, was ebenfalls zu einer Klassifikation als männliche Person führt.
- Wenn weder Anrede noch Geschlecht vorhanden sind, kann nicht klassifiziert werden.

In der durch das Servlet generierten Ontologie befindet sich neben der hier aufgeführten Klasse *Maennliche_Person* auch das Pendant *Weibliche_Person*, welche beide Unterklassen von *Person* sind.

```
1: <owl:Class rdf:ID="Maennliche_Person">
2: <rdfs:subClassOf rdf:resource="#Person"/>
3: <owl:equivalentClass>
4: <owl:Class>
```

```
5:      <owl:unionOf rdf:parseType="Collection">
6:              <owl:Restriction>
7:                      <owl:hasValue rdf:resource="#Herr"/>
8:                      <owl:onProperty>
9:                              <owl:FunctionalProperty rdf:about="#hat_anrede"/>
10:                     </owl:onProperty>
11:             </owl:Restriction>
12:             <owl:Restriction>
13:                     <owl:hasValue rdf:resource="#Maennlich"/>
14:                     <owl:onProperty>
15:                             <owl:FunctionalProperty rdf:about="#vorname1_hat_geschlecht"/>
16:                     </owl:onProperty>
17:             </owl:Restriction>
18:     </owl:unionOf>
19: </owl:Class>
20: </owl:equivalentClass>
21: </owl:Class>
```

Die Klasse wird in den Zeilen 3 und 4 als äquivalent zu einer anonymen Klasse, einem so genannten Blank Nodes (vgl. 3.2.8), bestehend aus einer Vereinigungsmenge definiert. Innerhalb des *unionOf* Operators sind zwei Restriktionen formuliert, von denen mindestens eine gelten muss, damit etwas Instanz dieser Klasse ist. Durch die Klassendefinition wird jedoch nicht gefordert, dass es sich dabei um eine Instanz der Klasse Person handeln muss.
In den Zeilen 9 und 15 wird über die Eigenschaften *hat_anrede* und *vorname1_hat_geschlecht* ausgesagt, dass sie als funktionale Properties maximal einen Wert besitzen dürfen, welcher in den Zeilen 7 und 13 durch *hasValue* angegeben wird.
Aufgrund dieser Deklaration ist ein Inferenzsystem in der Lage einen Fehler anzuzeigen, wenn in der Ontologie irgendeiner Instanz „beide Anreden“ oder „beide Geschlechter“ zugewiesen werden würde. Also beispielsweise durch die folgende Definition:

```
<owl:Class rdf:ID="XYZ"/>
<XYZ rdf:ID="Instanz_von_XYZ">
        <hat_anrede>
                <Anrede rdf:ID="Herr"/>
        </hat_anrede>
```

```
        <hat_anrede>
                <Anrede rdf:ID="Frau"/>
        </hat_anrede>
</XYZ>
```

Damit ein Inferenzsystem auch einen Fehler anzeigen kann, wenn eine falsche Kombination von Anrede und Geschlecht definiert würde, muss die Klasse *Maennliche_Person* als disjunkt zu ihrem Pendant *Weibliche_Person* deklariert werden.

```
owl:Class rdf:about="#Maennliche_Person">
        <owl:disjointWith rdf:resource="#Weibliche_Person"/>
</owl:Class>
```

Aufgrund dieser Deklaration kann ein Inferenzsystem einen Fehler anzeigen, wenn beispielsweise folgendes definiert würde:

```
<owl:Class rdf:ID="XYZ"/>
<XYZ rdf:ID="Instanz_von_XYZ">
        <hat_anrede>
                <Anrede rdf:ID="Herr"/>
        </hat_anrede>
        <vorname1_hat_geschlecht>
                <Geschlecht rdf:about="#Weiblich"/>
        </vorname1_hat_geschlecht>
</XYZ>
```

4.4 Inferenzsysteme

Zur Verarbeitung von OWL Ontologien auf der Grundlage von Beschreibungslogiken stehen zur Zeit die beiden Inferenzsysteme [FaCT] (*Fast Classification of Terminologies*) und [RACER] (*Renamed ABox and Concept Expression Reasoner*) zur Verfügung. Daher muss zunächst die Frage beantwortet werden, welches der beiden Systeme zur Verarbeitung der durch das Servlet generierten Ontologiedatei „*post_ont.owl“* geeigneter ist.

Die Terminologie von Beschreibungslogiken definiert eine Wissensbasis als ein Paar bestehend aus T-Box und A-Box [Möller, 2003]. Eine T-Box re-

präsentiert dabei die Menge von Konzepten und Relationen, während die A-Box das Wissen über die Instanzen und deren Relationen beschreibt. Beide Systeme bieten die Möglichkeit des T-Box-Reasoning, wodurch beispielsweise Unter- und Oberklassenbeziehungen abgeleitet werden können, jedoch ist RACER das einzige System, welches zusätzlich Inferenzmechanismen für die A-Box zur Verfügung stellt. Da innerhalb der generierten Ontologie zahlreiche Instanzen verwendet werden, kommt als Inferenzsystem nur RACER in Frage.

4.5 Information Retrieval mit RACER

Das Inferenzsystem ist in Form einer Client-Serverarchitektur realisiert. Als Serversoftware wird die Version 1.7.23 eingesetzt , welche für die Betriebssysteme Linux, Windows und Mac OS X unter der [RACER-Download-Page] frei zur Verfügung steht. Der Zugriff auf die Serverkomponente erfolgt durch das JAVA-Applet RICE (Racer Interactive Client Environment). Um die im folgenden beschriebenen Informationen ableiten zu können, muss auf einem lokalen Rechner sowohl die Serverkomponente als auch der Client installiert sein. Die Ontologiedatei muss dann durch RICE geladen werden.

Den gestellten Anfragen liegt die aufgrund des vollständig ausgefüllten Eingabeformulars generierte Ontologiedatei zugrunde, welche sich unter "*http://marc-schilling.de/sw/post_ont.owl*" befindet. Für die komplette Dokumentation der von RACER bereitgestellten Befehle, wird auf den „User's Guide and Reference“ [Haarslev, 2003] verwiesen.

4.5.1 T-Box-Retrieval

Alle atomaren Konzepte lassen sich durch die Anfrage (*all-atomic-concepts*) ausgeben, wodurch alle in der Ontologie befindliche Klassen angezeigt werden. Die Frage, ob es unerfüllbare Konzepte in der aktuellen T-Box gibt, wird mit Hilfe von (*tbox-coherent-p*) beantwortet. Für die T-Box in der Ontologie namens „DEFAULT“ lautet die Antwort „True“, weshalb keine inkonsistenten atomaren Konzepte existieren, da andernfalls die Antwort „NIL“ lauten würde.

In der Racer-Terminologie wird eine Taxonomie als klassifizierte T-Box bezeichnet, was mit (*tbox-classified?*) abgefragt werden kann. Die Ausgabe der Taxonomie erfolgt durch (*taxonomy*).

Unter- und Oberklassenbeziehungen lassen sich durch (*concept-descendants* |...|) und (*concept-ancestors* |...|) in Verbindung mit der URI-Referenz in Erfahrung bringen. Die Frage und Antwort nach allen Unterklassen von Person lautet:

```
(concept-descendants |http://marc-schilling.de/sw/post_ont.owl#Person|)
```

RACER Replies:

```
((*BOTTOM* BOTTOM)
    (|http://marc-schilling.de/sw/post_ont.owl#Weibliche_Person|)
    (|http://marc-schilling.de/sw/post_ont.owl#Maennliche_Person|)
    (|http://marc-schilling.de/sw/post_ont.owl#Mitarbeiter|))
```

Neben der Möglichkeit abzufragen, ob Klassen disjunkt (*concept-disjoint?* |...| |...|) sind, was bei männlichen und weiblichen Personen der Fall ist, kann auch nach Subsumtionsbeziehungen gefragt werden.

Die Klassendefinition von *Maennliche_Person* basiert auf der Existenz von zwei Property-Werten in Verbindung mit einer Vereinigungsmenge und sieht folgender Maßen aus:

```
(get-concept-definition |http://marc-schilling.de/sw/post_ont.owl#Maennliche_Person|)
```

RACER Replies:

```
(OR
    (SOME |http://marc-schilling.de/sw/post_ont.owl#hat_anrede|
          |http://marc-schilling.de/sw/post_ont.owl#Herr|)
    (SOME |http://marc-schilling.de/sw/post_ont.owl#vorname1_hat_geschlecht|
          |http://marc-schilling.de/sw/post_ont.owl#Maennlich|))
```

4.5.2 A-Box-Retrieval

Alle innerhalb der Ontologie vorhanden Instanzen lassen sich durch (*all-individuals*) ausgeben. Um sämtliche Informationen zu einer Instanz abzufragen, also zu welchen Klassen sie gehört und welche Werte ihre Eigenschaften

haben, wird (*describe-individual* |...|) in Verbindung mit der URI-Referenz verwendet.

Um komplexere Anfragen zu realisieren, stellt RACER die in Abbildung 12 gezeigten Operatoren zur Verfügung, wobei der Allquantor (*all R C*) noch nicht direkt unterstützt wird.

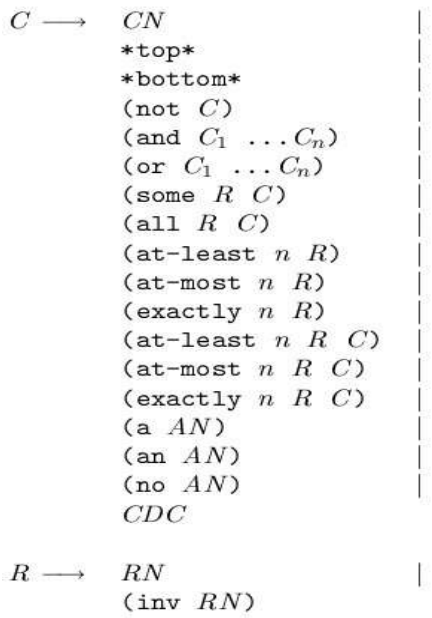


Abbildung 12: „An overview over all concept- and role-building operators„ entnommen: „RACER User's Guide and Reference“

Sollen beispielsweise alle Mitarbeiter einer Firma angezeigt werden, kann dies unter Verwendung des Existenzquantors (*some R C*) durch folgende Anfrage erfolgen:

(concept-instances (some |http://marc-schilling.de/sw/post_ont.owl#ist_mitarbeiter| |http://marc-schilling.de/sw/post_ont.owl#Firma|))

RACER Replies:

(|http://marc-schilling.de/sw/post_ont.owl#Mustermann|)

Neben der Möglichkeit die Instanzen einer Klassen abzufragen, für welche eine bestimmte Eigenschaft existiert, hätten auch einfach alle Instanzen der Klasse Mitarbeiter abgefragt werden können:

(concept-instances |http://marc-schilling.de/sw/post_ont.owl#Mitarbeiter|)

Durch den Operator (individual-fillers) wird die Instanz ermittelt, welche mit einer anderen Instanz über eine Eigenschaft verbunden ist, wobei in der RACER-Terminologie Eigenschaften als Rollen bezeichnet werden.

(individual-fillers |http://marc-schilling.de/sw/post_ont.owl#Mustermann|
|http://marc-schilling.de/sw/post_ont.owl#ist_mitarbeiter|)

RACER Replies:

(|http://marc-schilling.de/sw/post_ont.owl#Semantix|)

Abbildung 13 zeigt die Instanzen, welche durch die Rolle *ist_adressat* verbunden sind.

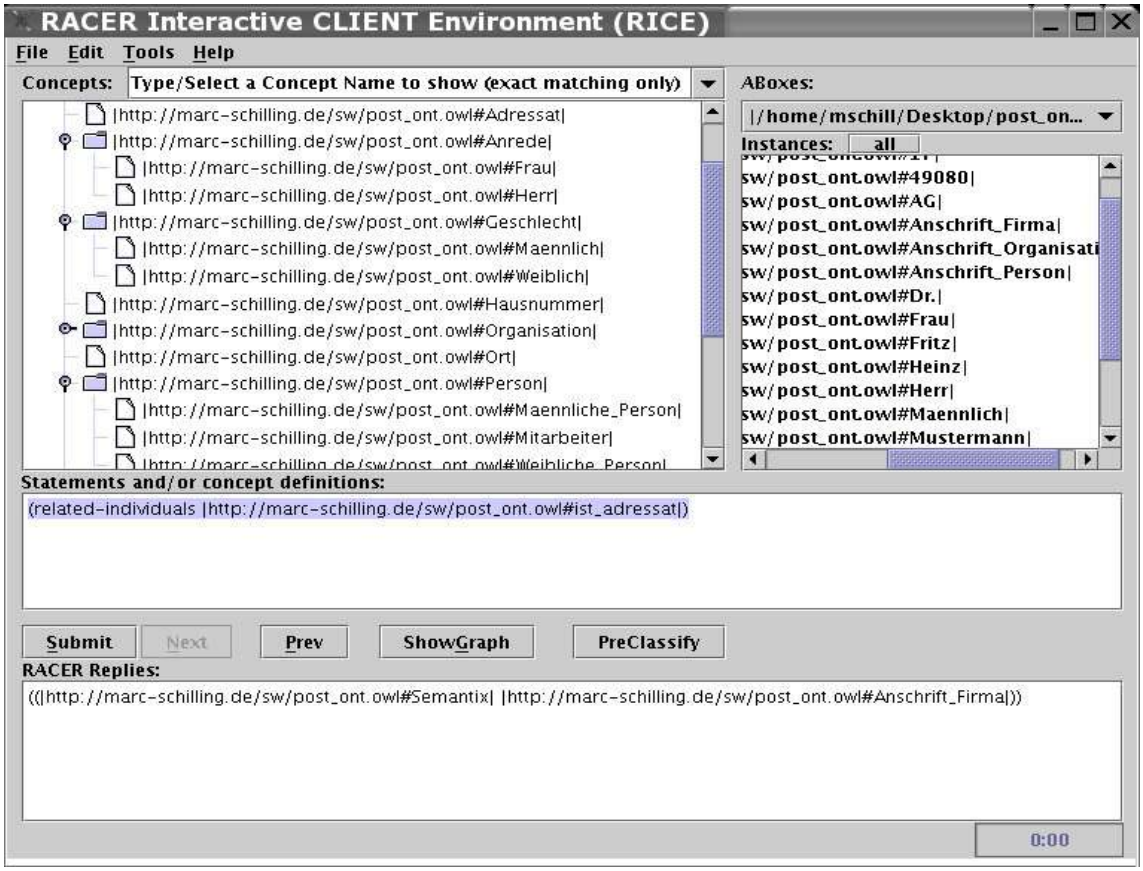


Abbildung 13: Das Abfrage-Interface von RICE

Mit Hilfe von (*related-individuals*) kann anhand der Eigenschaft das Subjekt und Objekt eines Tripel angezeigt werden.

(related-individuals |http://marc-schilling.de/sw/post_ont.owl#ist_mitarbeiter|)

RACER Replies:

((|http://marc-schilling.de/sw/post_ont.owl#Mustermann|
|http://marc-schilling.de/sw/post_ont.owl#Semantix|))

Werden alle direkten und abgeleiteten Konzepte der Instanz *Mustermann* durch den Operator (*individual-types*) abgefragt, besteht das Ergebnis aus den Klassen *Mitarbeiter*, *Person* und *Maennliche_Person*. Alternativ kann durch (*individual-direct-types*) nach den speziellsten Konzepten gefragt werden, wobei das Ergebnis dann nur aus den Klassen *Mitarbeiter* und *Maennliche_Person* besteht.

Einen Überblick über alle in der A-Box vorgenommenen Rollenzuweisungen und die dadurch in Beziehung gesetzten Instanzen erhält man mit Hilfe von (*all-role-assertions*).

Die folgende Abbildung zeigt die A-Box der generierten Ontologie mit Instanzen und ihren farbig dargestellten Beziehungen zu anderen Instanzen.

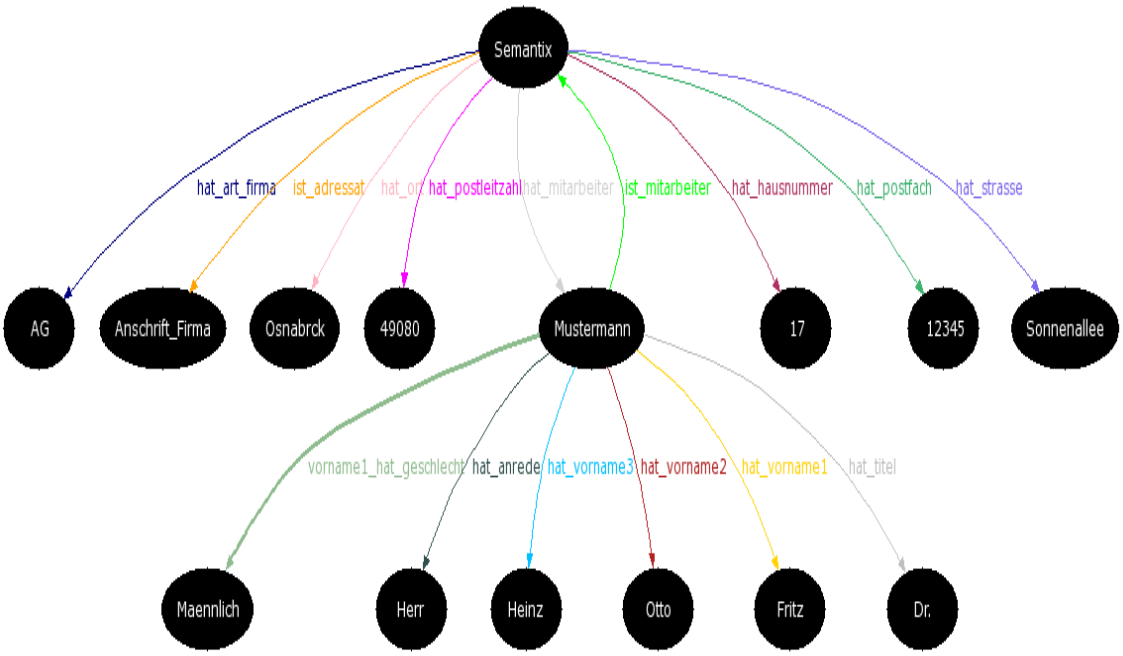


Abbildung 14: Graphische Darstellung der A-Box durch RICE

4.6 Bewertung der Prototypischen Implementierung

Das Ziel der prototypischen Implementierung bestand darin, die Beziehungen und Konzepte zwischen den eingegebenen Adressdaten einer postalischen Anschrift durch eine Ontologie zu repräsentieren. Um dieses Ziel zu erreichen, wurden die eingegebenen Daten als Instanzen von Klassen modelliert und die Beziehungen zwischen den Instanzen durch Relationen angegeben, wodurch sich die in den Kapiteln 4.5.1 und 4.5.2 beschriebenen Informationen ableiten lassen. Da als Repräsentationssprache OWL in der Variante DL verwendet wurde, verfügt die Ontologie über eine formale Semantik.

Insgesamt repräsentiert die generierte Ontologie nur einen Teil des Wissens, welches mit einer Domäne „Postalische Anschrift“ in Verbindung gebracht wird. Ein grundsätzliches Problem in der Domäne besteht beispielsweise darin, dass eine Unterscheidung, ob es sich entweder um eine Organisation, eine Firma oder eine Person handelt, nur anhand von nicht vorhandenen Eigenschaften (z.B. *hat_art_firma*) getroffen werden kann. Aufgrund der sich daraus ergebenden Probleme bei der Modellierung der Ontologie wurde die Unterscheidung durch das Servlet getroffen und anhand der Eigenschaft *ist_adressat* kodiert.
Eine Überprüfung der korrekten Datentypen geschieht ebenfalls durch das Servlet und wird nicht innerhalb der Ontologie durch *rdf:datatype* angegeben, was z.B. für die Eigenschaft *hat_postleitzahl* vom Typ positiver Integerwert sinnvoll gewesen wäre. Der Grund dafür liegt zum einen in der von der Variante OWL-DL geforderten Typentrennung. Danach ist eine Eigenschaft entweder Object-Property, verbindet also eine Instanz mit einer anderen oder Datatype-Property, welche eine Instanz mit einem XML-Schema Datentyp in Relation setzt. Da in der Ontologie die Eigenschaften als Object-Properties modelliert wurden, konnten die Datentypen nicht angegeben werden, ohne das der OWL-Validator die Ontologie als OWL Full eingestuft hätte. Zum anderen werden Datatype-Properties von heutigen Inferenzsystemen noch nicht unterstützt.
Die geforderte Typentrennung beeinflusst darüber hinaus die ohnehin komplexe Frage, ob ein Begriff als Klasse oder Instanz modelliert werden soll. Bei-

spielsweise wurde der Begriff Frau als Instanz der Klasse Anrede definiert. Dadurch ist es innerhalb der DL-Ontologie unmöglich, eine Klasse Frau zu modellieren, um eine weibliche Person zu beschreiben.

Soll das Wissen einer Domäne durch eine Ontologie repräsentiert werden, erfordert dies eine beträchtliche Einarbeitungszeit in die Sprachen RDF/S und OWL. Insbesondere die im Vergleich zu objektorientierten Programmiersprachen unterschiedlichen Typsysteme und deren Interpretation (vgl. 3.3.1, 3.3.2) waren dabei gewöhnungsbedürftig.

Aufgrund der geschilderten Modellierungsprobleme gestaltet sich die prinzipiell von einer Ontologie geforderte Wiederverwendbarkeit als schwierig, weshalb auf eine unter diesem Aspekt sinnvolle Trennung zwischen A-Box und T-Box verzichtet wurde. Der generierte Code kann jedoch problemlos in ein HTML-Dokument integriert werden und würde einem menschlichen Betrachter in aktuellen Browsern nicht angezeigt.

5 Zusammenfassung und Ausblick

Obwohl Tim Berners-Lee im Artikel „The Semantic Web“ dass Wort Vision nicht verwendet, wurde das darin geschilderte Beispielszenario in Kapitel 1.3 als solche bezeichnet. Die Intention des Artikel bestand wohl eher darin, dem Begriff Semantic Web eine höhere Popularität zu verschaffen, als eine realistische Einschätzung über die sich daraus ergebenden Möglichkeiten zu vermitteln.
Betrachtet man den ursprünglichen Entwurf des WWW, stellt das Semantic Web eine konsequente Weiterentwicklung der von Berners-Lee bereits 1989 formulierten Überlegungen dar. Als Direktor des W3C wird er diese Entwicklung sicherlich weiter vorantreiben, wobei der Erfolg von der breiten Akzeptanz einheitlicher Beschreibungsmodelle und standardisierter Sprachen abhängt.

Für die unteren Ebenen des Schichtenmodells existieren bereits erste erfolgreiche Anwendung. Beispielsweise kann hier auf die RDF Site Summary [W3C-

RSS] verwiesen werden, welche sich stark wachsender Beliebtheit erfreut und auf Technologien des Semantic Web basiert. Durch RDF/S wird insgesamt die Basis zur Verfügung gestellt, um Informationsinseln strukturiert miteinander zu vernetzen. Mit OWL steht bereits heute eine Ontologiesprache zur Verfügung, welche eine wohldefinierte Syntax und eine auf Logik basierende formale Semantik bereitstellt.
Für die Ebenen Logik, Proof und Trust wurde in Kapitel 1.4.3 konstatiert, dass die Technologien und Konzepte das Stadium akademischer Forschung noch nicht wirklich verlassen haben. Relativ unklar bleibt beispielsweise die Rolle der so genannten “unifying language” und damit der Einsatz von Inferenzmechanismen innerhalb eines verteilten Netzes. Eine besondere Herausforderung stellt dabei die in Kapitel 2.4.2 diskutierte Integration von verteilten Ontologien unter den Aspekten Ableitbarkeit, Überprüfbarkeit und Vertrauenswürdigkeit von Informationen dar.

Aufgrund der Komplexität der Technologien und des nicht zu unterschätzenden Aufwands, welcher selbst für die Modellierung eines kleinen Weltausschnitts notwendig ist, wird sich das Semantic Web nicht derart exponentiell entwickeln, wie es das WWW getan hat. Betrachtet man die Forschungsaktivitäten seit 1998, spricht jedoch einiges dafür, dass der von Tim Berners-Lee propagierte Netzwerkeffekt eintreten wird und semantische Technologien weiter Einzug in andere Bereiche der Informationstechnik (Suchmaschinen, Datenbanken) halten werden.

6 Anhang

6.1 Inhalt der CD

- Die Magisterarbeit im PDF-Format.
- Die Quellen des Literaturverzeichnisses, sofern diese aus dem Netz stammen.
- Die in der Arbeit diskutierte Ontologie in Form der Datei „post_ont.owl“.
- Der Quellcode des Servlets, mit dem die Ontologiedatei generiert wurde.
- Die Software des Inferenzsystems RACER in der Version 1.7.23, sowie der als Java-Applikation entwickelte Client RICE, welche im akademischen Umfeld frei verwendet werden darf.

6.2 Literaturverzeichnis

[Internet Domain Survey, 2004] ”Internet Domain Survey, Jan 2004”,
Internet Systems Consortium,
http://www.isc.org/index.pl?/ops/ds, besucht 21.04.2004

[Berners-Lee, 2001] Tim Berners-Lee, James Hendler und Ora Lassila:“The Semantic Web“, Scientific American, Mai 2001,
http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html,
besucht 08.04.2003

[Berners-Lee, 1989] Tim Berners-Lee: “Information Management: A Proposal”, CERN, März 1989,
http://www.w3.org/History/1989/proposal.html, besucht 21.04.2004

[Berners-Lee, 1990] Tim Berners-Lee: „Original design issues“ ff.,
http://www.w3.org/DesignIssues/, besucht 21.04.2004

[W3C-HTML 4.01 Specification, 1999] “HTML 4.01 Specification - W3C Recommendation 24 December 1999”,
http://www.w3.org/TR/html401/, besucht 21.04.2004

[Münz, 2001] Stefan Münz: „SELFHTML, HTML-Dateien selbst erstellen“,, Version 8.0,
http://de.selfhtml.org/html/verweise/typisierte.htm, besucht 21.04.2004

[Search Engine Watch, 2004],
http://www.searchenginewatch.com/links, besucht 21.04.2004

[Berners-Lee, 1998] Tim Bernere-Lee: „What the Semantic Web can represent“,
http://www.w3.org/DesignIssues/RDFnot.html, besucht 21.04.2004

[Berners-Lee, 1998a] Tim Berners-Lee: “A roadmap to the Semantic eb” (Sept 98)” ff.,
http://www.w3.org/DesignIssues/, besucht 21.04.200

[Koivunen, 2001] Marja-Riitta Koivunen, Eric Miller: “Proceedings of the Semantic Web Kick-off Seminar”, Finland, Nov 2, 2001,
http://www.w3.org/2001/12/semweb-fin/w3csw, besucht 21.04.2004

[W3C-Semantic Web Activity, 2001], Semantic Web Activity Arbeitsgruppe,
http://www.w3.org/2001/sw/, besucht 13.05.2004

[UNICODE],
http://www.unicode.org, besucht 28.01.2004

[W3C-URI, 2001] URI Planning Interest Group, W3C/IETF: „URIs, URLs, and URNs: Clarifications and Recommendations 1.0“,
http://www.w3.org/TR/uri-clarification/, besucht 21.04.2004

[net-lexikon, 2004],
http://www.lexikon-definition.de/Interoperabilitaet.html, Stand 30.01.2004

[Studer et al., 2003] Rudi Studer, Andreas Hotho, Gerd Stumme, Raphael Volz: „Semantic Web – State of the Art and Future Directions“, Künstliche Intelligenz, Heft 3/03, S. 5 ff.

[DL Implementation Group] Peter F. Patel-Schneider: „DL Implementation Group (DIG)“,
http://dl.kr.org/dig/, besucht, 22.05.2004

[Berners-Lee, 2001a] Tim Berners-Lee, James Hendler und Ora Lassila:“The Semantic Web“, Scientific American, Mai 2001,
http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html, besucht 08.04.2003

[Crawford, 1990] J. M. Crawford, Benjamin Kuipers: „ALL: Formalizing Access Limited Reasoning „ Department of Computer Sciences, The University of Texas At Austin, 25 May 1990,
ftp://ftp.cs.utexas.edu/pub/qsim/papers/Crawford+Kuipers-sowa-91.ps.gz, besucht 13.05.2204

[Wikipedia, 2005],
http://de.wikipedia.org/wiki/G%C3%B6delscher_Unvollst%C3%A4ndigkeitssatz,
besucht 04.01.2005

[Berners-Lee, 1997] Tim Berners-Lee: “Axioms of the Web Architecture: Metadata”,
http://www.w3.org/DesignIssues/Metadata.html, besucht 02.04.2004

[Quillian, 1967] M. R. Quillian: „Word concepts: A theory and simulation of some basic semantic capabilities“. Behavioral Science, 12:410 430, 1967.

[Collins & Quillian, 1969] Allan Collins, M. R. Quillian, M. Ross: "Retrieval time from semantic memory", in: Journal of Verbal Learning and Verbal Behavior, Bd. 8, 1969, S. 240-247

[Reimer, 1991] Ulrich Reimer: „Einführung in die Wissensrepräsentation“,Leitfäden der angewandten Informatik, B.G. Teubner Stuttgart 1991, Seite 124 ff.

[Gruber, 1993] Thomas R. Gruber: „A Translation Approach to Portable Ontology Specifications”, Academic Press, Stanford University, Juni 1993,
http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html, besucht 28.04.2004

[McGuinness, 2003] Deborah L. McGuinness: „Ontologies Come of Age“, in „Spinning The Semantic Web“, Edited by Dieter Fensel, James Hendler, Herry Lieberman, and Wolfgang Wahlster, The MIT Press Cambrige, Massachusetts, Seite 175 ff.

[McGuinness, 2003a] Deborah L. McGuinness: „Ontologies Come of Age“, in „Spinning The Semantic Web“, Edited by Dieter Fensel, James Hendler, Herry Lieberman, and Wolfgang Wahlster, The MIT Press Cambrige, Massachusetts, Seite 178 ff.

[Reimer, 1991b] Ulrich Reimer: „Einführung in die Wissensrepräsentation“,Leitfäden der angewandten Informatik, B.G. Teubner Stuttgart 1991, Seite 159 ff.

[Wiederhold, 1994] Gio Wiederhold: "An algebra for ontology composition" in "Proceedings of the 1994 Monterey Workshop on Formal Methods", pages 56–62. U.S. Naval
Postgraduate School, July 1994.

[Heflin, 2001] Jeffrey Douglas Heflin, " TOWARDS THE SEMANTIC WEB: KNOWLEDGE REPRESENTATION IN A DYNAMIC, DISTRIBUTED ENVIRONMENT", Dissertation, University of Maryland, College Park, 2001,
http://www.cse.lehigh.edu/~heflin/pubs/heflin-thesis-orig.pdf, besucht 22.04.2004

[Ding et al., 2002] Ying Ding, Dieter Fensel, Michel Klein, and Borys Omelayenko: "The Semantic Web: Yet Another Hip?", Data & Knowledge Engineering, Volume 41, Issues 2-3, Pages 205-227, June 2002,
http://citeseer.ist.psu.edu/ding02semantic.html, besucht 02.07.2004

[W3C-RDF and OWL Recommendation, 2004] "World Wide Web Consortium Issues RDF and OWL Recommendations",
http://www.w3.org/2004/01/sws-pressrelease, besucht30.06.2004

[W3C-XML, 2004] Tim Bray , Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau: “Extensible Markup Language (XML) 1.0 (Third Edition) - W3C Recommendation", 04 February 2004,
http://www.w3.org/TR/REC-xml/, besucht 08.07.2004

[Wielage, 2000] Gunter Wielage, Oliver Pott: „XML new technology“, Markt+Technik, Paderborn 2000

[W3C-XSL] Liam Quin: “The Extensible Stylesheet Language Family (XSL)“, 17.06.2004,
http://www.w3.org/Style/XSL/, besucht 21.04.2004

[Berners-Lee, 2001a] Tim Berners-Lee: “Are we done yet?“, Tenth International World Wide Web Conference, Opening Keynote, Hong Kong, May 2001,
http://www.w3.org/2001/Talks/0501-tbl/Overview.html, besucht 21.04.2004

[W3C-RDF-Modell, 1999] Ora Lassila, Ralph R. Swick: "Resource Description Framework (RDF) Model and Syntax Specification", W3C Recommendation 22 February 1999,
http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, besucht 26.01.2004

[W3C-RDF-Concepts, 2004] Graham Klyne, Jeremy J. Carroll, Brian McBride: „Resource Description Framework (RDF): Concepts and Abstract Syntax", W3C Recommendation 10 February 2004,
http://www.w3.org/TR/rdf-concepts/, besucht 26.01.2004

[W3C-RDF-URI-Reference, 2004] Graham Klyne, Jeremy J. Carroll, Brian McBride: „Resource Description Framework (RDF): Concepts and Abstract Syntax", W3C Recommendation 10 February 2004,
http://www.w3.org/TR/rdf-concepts/#dfn-URI-reference, besucht 26.01.2004

[W3C-RDF-Literals, 2004], Graham Klyne, Jeremy J. Carroll, Brian McBride: „Resource Description Framework (RDF): Concepts and Abstract Syntax", W3C Recommendation 10 February 2004,
http://www.w3.org/TR/rdf-concepts/#section-Literals, besucht 26.01.2004

[W3C-RDF-Fragment Identifiers, 2004] Graham Klyne, Jeremy J. Carroll, Brian McBride: „Resource Description Framework (RDF): Concepts and Abstract Syntax", W3C Recommendation 10 February 2004,
http://www.w3.org/TR/rdf-concepts/#section-fragID, besucht 26.01.2004

[W3C-XML-Base, 2001], Jonathan Marsh: ”XML Base”, W3C Recommendation 27 June 2001,
http://www.w3.org/TR/xmlbase/, besucht 04.08.2004

[W3C-RDF-Validation-Service, 2003], Eric Prud'hommeaux and Ryan Lee: „W3C RDF Validation Service“,
http://www.w3.org/RDF/Validator/, besucht 04.08.2004

[[W3C-RDF/XML-Syntax, 2004], Dave Beckett, Brian McBride: "RDF/XML Syntax Specification (Revised)", W3C Recommendation 10 February 2004,
http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/#section-Syntax-blank-nodes, besucht 08.07.2004

[W3C-RDF-Primer, 2004] Frank Manola, Eric Miller, Brian McBride: "RDF Primer"
W3C Recommendation 10 February 2004,
http://www.w3.org/TR/rdf-primer/#structuredproperties, besucht 08.07. 2004

[W3C-RDF-Primer, 2004a] Frank Manola, Eric Miller, Brian McBride: "RDF Primer"
W3C Recommendation 10 February 2004,
http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#containers, besucht 08.07.2004

[W3C-RDF-Primer, 2004b] Frank Manola, Eric Miller, Brian McBride: "RDF Primer"
W3C Recommendation 10 February 2004,
http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#collections, besucht 08.07.2004

[Dublin-Core, 2004] Dublin Core Metadata Initiative,
http://dublincore.org, besucht 08.07.2004

[W3C-RDF-Schema, 2004] Dan Brickley, R.V. Guha, Brian McBride: "RDF Vocabulary Description Language 1.0: RDF Schema", W3C Recommendation 10 February 2004,
http://www.w3.org/TR/2004/REC-rdf-schema-20040210/, besucht 08.07.2004

[W3C-RDF-Schema, 2000] Dan Brickley, R.V. Guha: "Resource Description Framework (RDF) Schema Specification 1.0", W3C Candidate Recommendation 27 March 2000, http://www.w3.org/TR/2000/CR-rdf-schema-20000327/#s2.1.1, besucht 08.07.2004

[W3C-RDF-Schema, 2000a] Dan Brickley, R.V. Guha: "Resource Description Framework (RDF) Schema Specification 1.0", W3C Candidate Recommendation 27 March 2000, http://www.w3.org/TR/2000/CR-rdf-schema-20000327, besucht 08.07.2004

[RFC-3066, 2001] H. Alvestrand: „RFC 3066 - Tags for the Identification of Languages“ http://www.isi.edu/in-notes/rfc3066.txt, besucht 29.11.2004

[Middendorf, 1999] Stefan Middendorf, Reiner Singer: „Java Programmierhandbuch und Referenz für die Java-2-Plattform “, dpunkt Verlag 1999, Seite 59ff

[DAML+OIL, 2001] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein: „DAML+OIL (March 2001) Reference Description“ http://www.w3.org/TR/daml+oil-reference, besucht 17.01.2005

[W3C-OWL-Language-Guide, 2004] Michael K. Smith, Chris Welty, Deborah L. McGuinness: „OWL Web Ontology Language Guide“, W3C Recommendation 10 February 2004, http://www.w3.org/TR/2004/REC-owl-guide-20040210/, besucht 19.01.2005

[W3C-OWL-Language-Guide, 2004a] Michael K. Smith, Chris Welty, Deborah L. McGuinness: „OWL Web Ontology Language Guide“, W3C Recommendation 10 February 2004, http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OntologyMapping, besucht 19.01.2005

[Protégé, 2005], „The Protégé Ontology Editor and Knowledge Acquisition System“, http://protege.stanford.edu/, besucht 21.01.2005

[OilEd, 2003], http://oiled.man.ac.uk/, besucht 11.04.2004

[Textarchiv 7], http://www.ta7.de/txt/listen/, besucht 16.04.2004

[OpenGeoDB] Manuel Hoppe, Thomas Mack: „OpenGeoDB - freie Geokoordinaten-Datenbank“, http://opengeodb.sourceforge.net/, besucht 16.04.2004

[OWL-Ontology Validator, 2003] Sean Bechhofer, Raphael Volz: „OWL Ontology Validator“ as part of the EU IST Project WonderWeb,
http://phoebus.cs.man.ac.uk:9999/OWL/Validator, besucht 26.01.2005

[Noy, 2001] Natalya F. Noy, Deborah L. McGuinness: „Abstract: Ontology Development 101: A Guide to Creating Your First Ontology“, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001,
http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html, besucht 02.09.2004

[Horridge, 2004] Matthew Horridge: „A Practical Guide To Building OWL Ontologies With The Protégé-OWL Plugin“, Edition 1.0, June 13, 2004,
http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf, besucht 09.02.2005

[FaCT] Ian Horrocks: „The FaCT System, 2003“,
http://www.cs.man.ac.uk/~horrocks/FaCT/, besucht 17.03.2005

[RACER] Volker Haarslev, Ralf Möller, Michael Wessel: „Racer System Description“
http://www.sts.tu-harburg.de/~r.f.moeller/racer/, besucht 17.03.2005

[Möller, 2003]Ralf Möller, Volker Haarslev: „Description Logics for the Semantic Web: Racer as a Basis for Building Agent Systems“, Künstliche Intelligenz, Heft 3/03, S. 10 ff

[RACER-Download-Page] Volker Haarslev, Ralf Möller, Michael Wessel: „The RACER System Download Page“,
http://www.sts.tu-harburg.de/~r.f.moeller/racer/download.html, besucht 17.03.2005

[Haarslev, 2003] Volker Haarslev and Ralf Möller: „RACER User's Guide and Reference Manual Version 1.7.7“,
http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-manual-1-7-19.pdf, besucht 19.03.2005

[W3C-RSS 1.0] Gabe Beged-Dov, Dan Brickley, Rael Dornfest, Ian Davis, ff:
„RDF Site Summary (RSS) 1.0“,
http://web.resource.org/rss/1.0/spec, besucht 26.03.2005

6.3 Abbildungs- und Tabellenverzeichnis

6.4 Die Ontologiedatei post_ont.owl

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://marc-schilling.de/sw/post_ont.owl#"
  xml:base="http://marc-schilling.de/sw/post_ont.owl">
  <owl:Ontology rdf:about=""/>

  <owl:Class rdf:ID="Postalische_Anschrift"/>
  <Postalische_Anschrift rdf:ID="Anschrift_Firma"/>
  <Postalische_Anschrift rdf:ID="Anschrift_Organisation"/>
  <Postalische_Anschrift rdf:ID="Anschrift_Person"/>

  <owl:Class rdf:ID="Adressat"/>
  <owl:ObjectProperty rdf:ID="ist_adressat"/>
  <owl:ObjectProperty rdf:about="#ist_adressat">
    <rdfs:domain rdf:resource="#Adressat"/>
  </owl:ObjectProperty>

  <owl:Class rdf:ID="Organisation"/>
  <owl:Class rdf:ID="Firma">
    <rdfs:subClassOf rdf:resource="#Organisation"/>
  </owl:Class>

  <owl:Class rdf:ID="Art_Firma">
    <rdfs:subClassOf rdf:resource="#Firma"/>
  </owl:Class>

  <owl:Class rdf:ID="Strasse"/>
  <owl:Class rdf:ID="Postfach"/>
  <owl:Class rdf:ID="Hausnummer"/>
  <owl:Class rdf:ID="Postleitzahl"/>
  <owl:Class rdf:ID="Ort"/>

  <owl:ObjectProperty rdf:ID="hat_art_firma"/>
  <owl:ObjectProperty rdf:ID="hat_strasse"/>
  <owl:ObjectProperty rdf:ID="hat_postfach"/>
  <owl:ObjectProperty rdf:ID="hat_hausnummer"/>
  <owl:ObjectProperty rdf:ID="hat_postleitzahl"/>
  <owl:ObjectProperty rdf:ID="hat_ort"/>

  <owl:ObjectProperty rdf:ID="hat_mitarbeiter">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="ist_mitarbeiter"/>
    </owl:inverseOf>
  </owl:ObjectProperty>

  <Firma rdf:ID="Semantix">
    <ist_adressat>
      <Postalische_Anschrift rdf:about="#Anschrift_Firma"/>
    </ist_adressat>
    <hat_art_firma>
      <Art_Firma rdf:ID="AG"/>
    </hat_art_firma>
    <hat_strasse>
```

```
    <Strasse rdf:ID="Sonnenallee"/>
  </hat_strasse>
  <hat_postfach>
    <Postfach rdf:ID="12345"/>
  </hat_postfach>
  <hat_hausnummer>
    <Hausnummer rdf:ID="17"/>
  </hat_hausnummer>
  <hat_postleitzahl>
    <Postleitzahl rdf:ID="49080"/>
  </hat_postleitzahl>
  <hat_ort>
    <Ort rdf:ID="Osnabr?ck"/>
  </hat_ort>
  <hat_mitarbeiter>
    <Mitarbeiter rdf:ID="Mustermann"/>
  </hat_mitarbeiter>
</Firma>

<owl:Class rdf:ID="Person"/>
<owl:Class rdf:ID="Mitarbeiter">
  <rdfs:subClassOf rdf:resource="#Person"/>
</owl:Class>

<owl:Class rdf:ID="Geschlecht">
  <owl:oneOf rdf:parseType="Collection">
    <Geschlecht rdf:ID="Maennlich"/>
    <Geschlecht rdf:ID="Weiblich"/>
  </owl:oneOf>
</owl:Class>

<owl:Class rdf:ID="Anrede">
  <owl:oneOf rdf:parseType="Collection">
    <Anrede rdf:ID="Herr"/>
    <Anrede rdf:ID="Frau"/>
  </owl:oneOf>
</owl:Class>

<owl:Class rdf:ID="Titel"/>
<owl:Class rdf:ID="Vorname1"/>
<owl:Class rdf:ID="Vorname2"/>
<owl:Class rdf:ID="Vorname3"/>

<owl:ObjectProperty rdf:ID="hat_anrede"/>
<owl:ObjectProperty rdf:ID="hat_titel"/>
<owl:ObjectProperty rdf:ID="hat_vorname1"/>
<owl:ObjectProperty rdf:ID="vorname1_hat_geschlecht"/>
<owl:ObjectProperty rdf:ID="hat_vorname2"/>
<owl:ObjectProperty rdf:ID="hat_vorname3"/>

<Mitarbeiter rdf:about="#Mustermann">
  <ist_mitarbeiter>
    <Firma rdf:about="#Semantix"/>
  </ist_mitarbeiter>
</Mitarbeiter>

<Person rdf:about="#Mustermann">
  <hat_anrede>
    <Anrede rdf:ID="Herr"/>
  </hat_anrede>
```

```
  <hat_titel>
    <Titel rdf:ID="Dr."/>
  </hat_titel>
  <hat_vorname1>
    <Vorname1 rdf:ID="Fritz"/>
  </hat_vorname1>
  <vorname1_hat_geschlecht>
    <Geschlecht rdf:about="#Maennlich"/>
  </vorname1_hat_geschlecht>
  <hat_vorname2>
    <Vorname2 rdf:ID="Otto"/>
  </hat_vorname2>
  <hat_vorname3>
    <Vorname3 rdf:ID="Heinz"/>
  </hat_vorname3>
</Person>

<owl:Class rdf:ID="Maennliche_Person">
<rdfs:subClassOf rdf:resource="#Person"/>
<owl:equivalentClass>
<owl:Class>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:hasValue rdf:resource="#Herr"/>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#hat_anrede"/>
      </owl:onProperty>
    </owl:Restriction>
    <owl:Restriction>
      <owl:hasValue rdf:resource="#Maennlich"/>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#vorname1_hat_geschlecht"/>
      </owl:onProperty>
    </owl:Restriction>
  </owl:unionOf>
  </owl:Class>
</owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Weibliche_Person">
<rdfs:subClassOf rdf:resource="#Person"/>
<owl:equivalentClass>
<owl:Class>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:hasValue rdf:resource="#Frau"/>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#hat_anrede"/>
      </owl:onProperty>
    </owl:Restriction>
    <owl:Restriction>
      <owl:hasValue rdf:resource="#Weiblich"/>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#vorname1_hat_geschlecht"/>
      </owl:onProperty>
    </owl:Restriction>
  </owl:unionOf>
  </owl:Class>
</owl:equivalentClass>
</owl:Class>
```

```
  <owl:Class rdf:about="#Maennliche_Person">
    <owl:disjointWith rdf:resource="#Weibliche_Person"/>
  </owl:Class>

</rdf:RDF>
```

6.5 Quellcode des Servlets

```
package gf_servlet;
import java.lang.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.*;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author unascribed
 * @version 1.0
 */

public class Servlet1 extends HttpServlet {
  static final private String CONTENT_TYPE = "text/html";
  private Connection geschlechtdb, opengeodb;

  String zeile, newline=System.getProperty("line.separator");
  FileReader fr;
  FileWriter fw;
  BufferedReader br;
  BufferedWriter bw;

   static String anrede, name_orga_firma, art_firma, titel, vorname1, vorname2, vorname3, nachname1, strasse,
hausnummer, postfach, postleitzahl, ort, owl_file;

  static String vorname1_hat_geschlecht ="", check_anrede_string ="", check_art_firma_string="", check_vorname1,
check_vorname2, check_vorname3, check_nachname1, check_postleitzahl ="", check_ort="";
  static int check_anrede, check_art_firma;

    //Initialize global variables
  public void init() throws ServletException {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      geschlechtdb = DriverManager.getConnection("jdbc:mysql:/127.0.0.1/geschlecht", "LOGIN", "PASSWORT");
      opengeodb = DriverManager.getConnection("jdbc:mysql:/127.0.0.1/opengeodb", "LOGIN", "PASSWORT");
      owl_file = getServletConfig().getServletContext().getRealPath("/") + "post_ont.owl";
    } catch (Exception ex) { ex.printStackTrace(); }
  }

  //Process the HTTP Get request
   public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOExcep-
tion {

  }
  //Process the HTTP Post request
   public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOEx-
ception {
    int row_ort=0;
    boolean eingabe_ok = true;

    anrede = request.getParameter("anrede");
    name_orga_firma = request.getParameter("name_orga_firma");
    art_firma = request.getParameter("art_firma");
    titel = request.getParameter("titel");
    vorname1 = request.getParameter("vorname1");
    vorname2 = request.getParameter("vorname2");
    vorname3 = request.getParameter("vorname3");
    nachname1 = request.getParameter("nachname1");
    strasse = request.getParameter("strasse");
    hausnummer = request.getParameter("hausnummer");
    postfach = request.getParameter("postfach");
    postleitzahl = request.getParameter("plz");
    ort = request.getParameter("ort");

    boolean ist_gueltig = true;
```

```
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Technologien und Konzepte des Semantic Web</title></head>");
     out.println("<body bgcolor=\"#cfdaea\" link=\"#00325a\" vlink=\"#00325a\" alink=\"#00325a\"  background=\"back-
grnd.gif\" leftmargin=\"5\" topmargin=\"5\" marginwidth = \"5\" marginheight=\"5\">");


    out.println("Name der Organisation/Firma: ");
    out.println(name_orga_firma);
    out.println("<br>");

    out.println("Art der Firma: ");
    try {
      check_art_firma=Integer.parseInt(Servlet1.art_firma);
    }
    catch (NumberFormatException e2) {
    }
    if (check_art_firma ==0) check_art_firma_string="";
    if (check_art_firma ==1) check_art_firma_string="Einzelunternehmung";
    if (check_art_firma ==2) check_art_firma_string="GmbH";
    if (check_art_firma ==3) check_art_firma_string="GmbH_und_Co._KG";
    if (check_art_firma ==4) check_art_firma_string="AG";
    if (check_art_firma ==5) check_art_firma_string="KG";
    if (check_art_firma ==6) check_art_firma_string="GbR";
    if (check_art_firma ==7) check_art_firma_string="OHG";
    if (check_art_firma ==8) check_art_firma_string="KGaA";
    if (check_art_firma ==9) check_art_firma_string="Genossenschaft";
    out.println(check_art_firma_string);
    out.println("<br>");

    out.println("Anrede: ");
    try {
      check_anrede=Integer.parseInt(Servlet1.anrede);
    }
    catch (NumberFormatException e1) {
    }

    if (check_anrede ==0) {
      check_anrede_string="Keine Anrede";
    }
    if (check_anrede ==1) {
      check_anrede_string="Herr";
    }
    if (check_anrede ==2) {
      check_anrede_string="Frau";
    }
    out.println(check_anrede_string);
    vorname1_hat_geschlecht ="";
    int geschlecht = geschlechtsbestimmung (vorname1);

    if (geschlecht ==1) {
      vorname1_hat_geschlecht ="Maennlich";
    }
    if (geschlecht ==2) {
      vorname1_hat_geschlecht ="Weiblich";
    }

    if (vorname1.equals("")) {
    }
    else {
      if (geschlecht ==0) {
        out.println("Fr den 1. Vorname: "+vorname1+" ist kein Datensatz vorhanden!");
      }
      if ( (geschlecht ==1) & (check_anrede==0) ) {
        out.println("<font color=\"green\">");
        out.println("Der 1. Vorname: "+vorname1+" ist m   nlich!");
        out.println("</font>");

      }
      if ( (geschlecht ==2) & (check_anrede==0) ) {
        out.println("<font color=\"green\">");
        out.println("Der 1. Vorname: "+vorname1+" ist weiblich!");
```

```
    out.println("</font>");

  }
  if ( (geschlecht ==1) & (check_anrede==1) ) {
    out.println("<font color=\"green\">");
    out.println("Der 1. Vorname: "+vorname1+" ist m   nlich!");
    out.println("</font>");

  }
  if ( (geschlecht ==1) & (check_anrede==2) ) {
    out.println("<font color=\"red\">");
    out.println("Der 1. Vorname: "+vorname1+" ist m   nlich!");
    out.println("</font>");
    eingabe_ok = false;
  }
  if ( (geschlecht==2) & (check_anrede==2) ) {
    out.println("<font color=\"green\">");
    out.println("Der 1. Vorname: "+vorname1+" ist weiblich!");
    out.println("</font>");

  }
  if ( (geschlecht==2) & (check_anrede==1) ) {
    out.println("<font color=\"red\">");
    out.println("Der 1. Vorname: "+vorname1+" ist weiblich!");
    out.println("</font>");
    eingabe_ok = false;
  }

   if (geschlecht==3) {
     out.println("<font color=\"green\">");
     out.println("Der 1. Vornamen: "+vorname1+" kann sowohl weiblich als auch maennlich sein!");
     out.println("</font>");
   }

}
out.println("<br>");

out.println("Titel der Person: ");
out.println(titel);
out.println("<br>");

out.println("1. Vorname: ");
out.println(vorname1);
if (eingabe_ohne_space (vorname1) == true) {
  out.println("<br>");
}
else {
  out.println("<font color=\"red\">");
  out.println("Der 1. Vorname enth   t ungltige Zeichen!<br>" );
  out.println("</font>");
  eingabe_ok = false;
}

out.println("2. Vorname: ");
out.println(vorname2);
if (eingabe_ohne_space (vorname2) == true) {
  out.println("<br>");
}
else {
  out.println("<font color=\"red\">");
  out.println("Der 2. Vorname enth   t ungltige Zeichen!<br>" );
  out.println("</font>");
  eingabe_ok = false;
}

out.println("3. Vorname: ");
out.println(vorname3);
if (eingabe_ohne_space (vorname3) == true) {
  out.println("<br>");
}
else {
  out.println("<font color=\"red\">");
  out.println("Der 3. Vorname enth   t ungltige Zeichen!<br>" );
  out.println("</font>");
  eingabe_ok = false;
```

```
}

out.println("Nachname: ");
out.println(nachname1);
if (eingabe_mit_space (nachname1) == true) {
  out.println("<br>");
}
else {
  out.println("<font color=\"red\">");
  out.println("Der Nachname entht    ungltige Zeichen!<br>" );
  out.println("</font>");
  eingabe_ok = false;
}
out.println("Strasse: ");
out.println(strasse);
if (eingabe_mit_space (strasse) == true) {
  out.println("<br>");
}
else {
  out.println("<font color=\"red\">");
  out.println("Der Strassenname enth   t ungltige Zeichen!<br>" );
  out.println("</font>");
  eingabe_ok = false;
}

out.println("Hausnummer: ");
out.println(hausnummer);
if ( (strasse.equals("")) & (!hausnummer.equals("")) ) {
  out.println("<font color=\"red\">");
  out.println("Wenn eine Hausnummer angegeben wird, mu   auch ein Strassenname angegeben werden!<br>" );
  out.println("</font>");
  eingabe_ok = false;
}
else {
out.println("<br>");
}

out.println("Postfach: ");
out.println(postfach);
if (eingabe_ist_digit (postfach) == true) {
  out.println("<br>");
}
else {
  out.println("<font color=\"red\">");
  out.println("Das Postfach enth   t ungltige Zeichen!<br>" );
  out.println("</font>");
  eingabe_ok = false;
}

out.println("Postleitzahl: ");
out.println(postleitzahl);
if ( !postleitzahl.equals("") && postleitzahl.length() == 5 && eingabe_ist_digit (postleitzahl) ) {
 check_ort = ortsbestimmung(postleitzahl);
  if (check_ort !="") {
    out.println("Der zur Postleitzahl zugehige    Ort ist: "+check_ort);
  }
  else {
    out.println("<font color=\"red\">");
    out.println("Der zur Postleitzahl zugehige    Ort konnte nicht ermittelt werden!" );
    out.println("</font>");
    eingabe_ok = false;
  }
}
else {
  out.println("<font color=\"red\">");
  out.println("Die Postleitzahl ist ungltig!" );
  out.println("</font>");
  eingabe_ok = false;
}
out.println("<br>");

out.println("Ort: ");
out.println(ort);
if (eingabe_mit_space (ort) == true) {
  check_postleitzahl = plzbestimmung (ort);
```

```
      if (check_ort !="") {
        out.println("Die zum Ort zugeh   ige Postleitzahl ist: "+check_postleitzahl);
      }
      else {
        out.println("<font color=\"red\">");
        out.println("Die zum Ort zugeh   ige Postleitzahl konnte nicht ermittelt werden!" );
        out.println("</font>");
        eingabe_ok = false;
      }
    }
    else {
      out.println("<font color=\"red\">");
      out.println("Der Ortsname enth   t ungltige Zeichen!" );
      out.println("</font>");
      eingabe_ok = false;
    }


    if (postleitzahl.equals("") & ort.equals("")) {
       out.println("<font color=\"red\">");
       out.println("Es wurde keine Postleitzahl und/oder Ort eingegeben!");
       out.println("</font>");
       eingabe_ok = false;
     }
     out.println("<br>");

      StringTokenizer toki_plz;
      String postleitzahl_ausDB;
      boolean plz_gleich_plz_aus_DB = false;
      toki_plz=new StringTokenizer(check_postleitzahl,",");
      while (toki_plz.hasMoreTokens()) {
        postleitzahl_ausDB=toki_plz.nextToken();
        if (postleitzahl.equals(postleitzahl_ausDB) ) {
         plz_gleich_plz_aus_DB = true;
         break;
        }
      }
      if (plz_gleich_plz_aus_DB) {
        out.println("<font color=\"green\">");
        out.println("Postleitzahl und Ort sind OK!");
        out.println("</font>");
      }
      else {
        out.println("<font color=\"red\">");
        out.println("Postleitzahl und Ort sind unterschiedlich!");
        out.println("</font>");
        eingabe_ok = false;
      }
      out.println("<br>");
      out.println("</font>");

//**************************************


    check_vorname1=vorname1;
    check_vorname2=vorname2;
    check_vorname3=vorname3;
    check_nachname1=nachname1;
    check_ort=ort;


      if ((!name_orga_firma.equals("")) | (!nachname1.equals("")))
        {
      }
      else {
        out.println("<font color=\"red\">");
        out.println("Geben Sie den Namen der Organisation/Firma und/oder den Nachnamen ein!<br>" );
        out.println("</font>");
        eingabe_ok = false;
      }



       if (  ( (check_anrede>0) | (!titel.equals("")) | (!vorname1.equals("")) | (!vorname2.equals("")) | (!vorname3.equals
("""))) & (nachname1.equals(""))  )
```

```
    {
      out.println("<font color=\"red\">");
       out.println("Wenn Sie Anrede, Titel oder Vornamen einer Person eingeben, mussen Sie auch den Nachnamen
eingeben!<br>" );
      out.println("</font>");
      eingabe_ok = false;
    }

    if ( (!check_art_firma_string.equals("")) & (name_orga_firma.equals("")) )
    {
      out.println("<font color=\"red\">");
      out.println("Wenn Sie die Art der Firma eingeben, mussen Sie auch den Firmennamen eingeben!<br>" );
      out.println("</font>");
      eingabe_ok = false;
    }


    if ((!strasse.equals("")) | (!postfach.equals(""))) {
    }
    else {
      out.println("<font color=\"red\">");
      out.println("Geben Sie die Strasse und/oder das Postfach ein!<br>" );
      out.println("</font>");
      eingabe_ok = false;
    }

    if ((vorname1.equals("")) & (!vorname2.equals(""))) {
      out.println("<font color=\"red\">");
      out.println("Wenn Sie einen 2. Vornamen eingeben, mussen Sie auch den 1. Vornamen eingeben!<br>" );
      out.println("</font>");
      eingabe_ok = false;
    }
    if ((vorname1.equals("")) & (!vorname3.equals(""))) {
      out.println("<font color=\"red\">");
      out.println("Wenn Sie einen 3. Vornamen eingeben, mussen Sie auch den 1. Vornamen eingeben!<br>" );
      out.println("</font>");
      eingabe_ok = false;
    }


    out.println("<br>");
    if (eingabe_ok == false) {
      out.println("<font color=\"red\">");
      out.println("Das Formular ist nicht korrekt ausgefllt!");
      out.println("</font>");
    }
    else {
      out.println("<font color=\"green\">");
      out.println("Formulareingaben korrekt!<br>");
      out.println("</font>");
      out.println("OWL Datei wird generiert!<br>");
      out.println("<br>");

      out.println("<a href=\"post_ont.owl\" target=\"_blank\">OWL Datei ansehen</a><br>");

      out.println("</body></html>");

  try {

    File f = new File(owl_file);
    fw = new FileWriter(f);
    bw = new BufferedWriter(fw);



  bw.write("<?xml version=\"1.0\"?>"+newline);
  bw.write("<rdf:RDF"+newline);
  bw.write("  xmlns:rdf =\"http://www.w3.org/1999/02/22-rdf-syntax-ns#\""+newline);
  bw.write("  xmlns:rdfs=\"http://www.w3.org/2000/01/rdf-schema#\""+newline);
  bw.write("  xmlns:xsd =\"http://www.w3.org/2000/10/XMLSchema#\""+newline);
  bw.write("  xmlns:owl=\"http://www.w3.org/2002/07/owl#\""+newline);

  bw.write("  xmlns=\"http://marc-schilling.de/sw/post_ont.owl#\""+newline);
  bw.write("  xml:base=\"http://marc-schilling.de/sw/post_ont.owl\">"+newline);
```

```
  bw.write("  <owl:Ontology rdf:about=\"\"/>"+newline+newline);

  bw.write("  <owl:Class rdf:ID=\"Postalische_Anschrift\"/>"+newline);

  bw.write("  <Postalische_Anschrift rdf:ID=\"Anschrift_Firma\"/>"+newline);
  bw.write("  <Postalische_Anschrift rdf:ID=\"Anschrift_Organisation\"/>"+newline);
  bw.write("  <Postalische_Anschrift rdf:ID=\"Anschrift_Person\"/>"+newline+newline);

  bw.write("  <owl:Class rdf:ID=\"Adressat\"/>"+newline);

  bw.write("  <owl:ObjectProperty rdf:ID=\"ist_adressat\"/>"+newline);
  bw.write("  <owl:ObjectProperty rdf:about=\"#ist_adressat\">"+newline);
  bw.write("    <rdfs:domain rdf:resource=\"#Adressat\"/>"+newline);
  bw.write("  </owl:ObjectProperty>"+newline+newline);


  //Firma
 if (  (!name_orga_firma.equals("")) & (!check_art_firma_string.equals(""))  )
  {
    bw.write("  <owl:Class rdf:ID=\"Organisation\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Firma\">"+newline);
    bw.write("    <rdfs:subClassOf rdf:resource=\"#Organisation\"/>"+newline);
    bw.write("  </owl:Class>"+newline+newline);

    bw.write("  <owl:Class rdf:ID=\"Art_Firma\">"+newline);
    bw.write("    <rdfs:subClassOf rdf:resource=\"#Firma\"/>"+newline);
    bw.write("  </owl:Class>"+newline+newline);

    bw.write("  <owl:Class rdf:ID=\"Strasse\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Postfach\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Hausnummer\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Postleitzahl\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Ort\"/>"+newline+newline);

    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_art_firma\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_strasse\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_postfach\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_hausnummer\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_postleitzahl\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_ort\"/>"+newline+newline);

    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_mitarbeiter\">"+newline);
    bw.write("    <owl:inverseOf>"+newline);
    bw.write("      <owl:ObjectProperty rdf:ID=\"ist_mitarbeiter\"/>"+newline);
    bw.write("    </owl:inverseOf>"+newline);
    bw.write("  </owl:ObjectProperty>"+newline+newline);

    bw.write("  <Firma rdf:ID=\""+name_orga_firma+"\">"+newline);

    bw.write("    <ist_adressat>"+newline);
    bw.write("      <Postalische_Anschrift rdf:about=\"#Anschrift_Firma\"/>"+newline);
    bw.write("    </ist_adressat>"+newline);

    bw.write("    <hat_art_firma>"+newline);
    bw.write("      <Art_Firma rdf:ID=\""+check_art_firma_string+"\"/>"+newline);
    bw.write("    </hat_art_firma>"+newline);

    if (!strasse.equals("")) {

      bw.write("    <hat_strasse>"+newline);
      bw.write("      <Strasse rdf:ID=\""+strasse+"\"/>"+newline);
      bw.write("    </hat_strasse>"+newline);
    }

    if (!postfach.equals("")) {

      bw.write("    <hat_postfach>"+newline);
      bw.write("      <Postfach rdf:ID=\""+postfach+"\"/>"+newline);
      bw.write("    </hat_postfach>"+newline);
    }
    if (!hausnummer.equals("")) {

      bw.write("    <hat_hausnummer>"+newline);
      bw.write("      <Hausnummer rdf:ID=\""+hausnummer+"\"/>"+newline);
      bw.write("    </hat_hausnummer>"+newline);
```

```
}

bw.write("    <hat_postleitzahl>"+newline);
bw.write("      <Postleitzahl rdf:ID=\""+postleitzahl+"\"/>"+newline);
bw.write("    </hat_postleitzahl>"+newline);

bw.write("    <hat_ort>"+newline);
bw.write("      <Ort rdf:ID=\""+ort+"\"/>"+newline);
bw.write("    </hat_ort>"+newline);

if (!nachname1.equals("")) {
  bw.write("    <hat_mitarbeiter>"+newline);
  bw.write("      <Mitarbeiter rdf:ID=\""+nachname1+"\"/>"+newline);
  bw.write("    </hat_mitarbeiter>"+newline);

  bw.write("  </Firma>"+newline+newline);

  bw.write("  <owl:Class rdf:ID=\"Person\"/>"+newline);

  bw.write("  <owl:Class rdf:ID=\"Mitarbeiter\">"+newline);
  bw.write("    <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
  bw.write("  </owl:Class>"+newline+newline);


  bw.write("  <owl:Class rdf:ID=\"Geschlecht\">"+newline);
  bw.write("    <owl:oneOf rdf:parseType=\"Collection\">"+newline);
  bw.write("      <Geschlecht rdf:ID=\"Maennlich\"/>"+newline);
  bw.write("      <Geschlecht rdf:ID=\"Weiblich\"/>"+newline);
  bw.write("    </owl:oneOf>"+newline);
  bw.write("  </owl:Class>"+newline+newline);

  bw.write("  <owl:Class rdf:ID=\"Anrede\">"+newline);
  bw.write("    <owl:oneOf rdf:parseType=\"Collection\">"+newline);
  bw.write("      <Anrede rdf:ID=\"Herr\"/>"+newline);
  bw.write("      <Anrede rdf:ID=\"Frau\"/>"+newline);
  bw.write("    </owl:oneOf>"+newline);
  bw.write("  </owl:Class>"+newline+newline);

  bw.write("  <owl:Class rdf:ID=\"Titel\"/>"+newline);
  bw.write("  <owl:Class rdf:ID=\"Vorname1\"/>"+newline);
  bw.write("  <owl:Class rdf:ID=\"Vorname2\"/>"+newline);
  bw.write("  <owl:Class rdf:ID=\"Vorname3\"/>"+newline+newline);

  bw.write("  <owl:ObjectProperty rdf:ID=\"hat_anrede\"/>"+newline);
  bw.write("  <owl:ObjectProperty rdf:ID=\"hat_titel\"/>"+newline);
  bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname1\"/>"+newline);
  bw.write("  <owl:ObjectProperty rdf:ID=\"vorname1_hat_geschlecht\"/>"+newline);
  bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname2\"/>"+newline);
  bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname3\"/>"+newline+newline);


  bw.write("  <Mitarbeiter rdf:about=\"#"+nachname1+"\">"+newline);
  bw.write("    <ist_mitarbeiter>"+newline);
  bw.write("      <Firma rdf:about=\"#"+name_orga_firma+"\"/>"+newline);
  bw.write("    </ist_mitarbeiter>"+newline);
  bw.write("  </Mitarbeiter>"+newline+newline);

  bw.write("  <Person rdf:about=\"#"+nachname1+"\">"+newline);

  if (!check_anrede_string.equals("Keine Anrede")) {

    bw.write("    <hat_anrede>"+newline);
    bw.write("      <Anrede rdf:ID=\""+check_anrede_string+"\"/>"+newline);
    bw.write("    </hat_anrede>"+newline);
  }
  if (!titel.equals("")) {

    bw.write("    <hat_titel>"+newline);
    bw.write("      <Titel rdf:ID=\""+titel+"\"/>"+newline);
    bw.write("    </hat_titel>"+newline);
  }
  if (!vorname1.equals("")) {

    bw.write("    <hat_vorname1>"+newline);
    bw.write("      <Vorname1 rdf:ID=\""+vorname1+"\"/>"+newline);
```

```
  bw.write("    </hat_vorname1>"+newline);
 }
 if (!vorname1_hat_geschlecht.equals("")) {

  bw.write("    <vorname1_hat_geschlecht>"+newline);
  bw.write("      <Geschlecht rdf:about=\"#"+vorname1_hat_geschlecht+"\"/>"+newline);
  bw.write("    </vorname1_hat_geschlecht>"+newline);
 }
 if (!vorname2.equals("")) {

  bw.write("    <hat_vorname2>"+newline);
  bw.write("      <Vorname2 rdf:ID=\""+vorname2+"\"/>"+newline);
  bw.write("    </hat_vorname2>"+newline);
 }
 if (!vorname3.equals("")) {

  bw.write("    <hat_vorname3>"+newline);
  bw.write("      <Vorname3 rdf:ID=\""+vorname3+"\"/>"+newline);
  bw.write("    </hat_vorname3>"+newline);
 }

 bw.write("  </Person>"+newline+newline);

 if (  (!check_anrede_string.equals("Keine Anrede")) | (!vorname1_hat_geschlecht.equals(""))  ) {

  bw.write("  <owl:Class rdf:ID=\"Maennliche_Person\">"+newline);
  bw.write("  <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
  bw.write("  <owl:equivalentClass>"+newline);
  bw.write("  <owl:Class>"+newline);
  bw.write("    <owl:unionOf rdf:parseType=\"Collection\">"+newline);
  bw.write("      <owl:Restriction>"+newline);
  bw.write("        <owl:hasValue rdf:resource=\"#Herr\"/>"+newline);
  bw.write("        <owl:onProperty>"+newline);
  bw.write("          <owl:FunctionalProperty rdf:about=\"#hat_anrede\"/>"+newline);
  bw.write("        </owl:onProperty>"+newline);
  bw.write("      </owl:Restriction>"+newline);
  bw.write("      <owl:Restriction>"+newline);
  bw.write("        <owl:hasValue rdf:resource=\"#Maennlich\"/>"+newline);
  bw.write("        <owl:onProperty>"+newline);
  bw.write("          <owl:FunctionalProperty rdf:about=\"#vorname1_hat_geschlecht\"/>"+newline);
  bw.write("        </owl:onProperty>"+newline);
  bw.write("      </owl:Restriction>"+newline);
  bw.write("    </owl:unionOf>"+newline);
  bw.write("    </owl:Class>"+newline);
  bw.write("  </owl:equivalentClass>"+newline);
  bw.write("  </owl:Class>"+newline+newline);

  bw.write("  <owl:Class rdf:ID=\"Weibliche_Person\">"+newline);
  bw.write("  <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
  bw.write("  <owl:equivalentClass>"+newline);
  bw.write("  <owl:Class>"+newline);
  bw.write("    <owl:unionOf rdf:parseType=\"Collection\">"+newline);
  bw.write("      <owl:Restriction>"+newline);
  bw.write("        <owl:hasValue rdf:resource=\"#Frau\"/>"+newline);
  bw.write("        <owl:onProperty>"+newline);
  bw.write("          <owl:FunctionalProperty rdf:about=\"#hat_anrede\"/>"+newline);
  bw.write("        </owl:onProperty>"+newline);
  bw.write("      </owl:Restriction>"+newline);
  bw.write("      <owl:Restriction>"+newline);
  bw.write("        <owl:hasValue rdf:resource=\"#Weiblich\"/>"+newline);
  bw.write("        <owl:onProperty>"+newline);
  bw.write("          <owl:FunctionalProperty rdf:about=\"#vorname1_hat_geschlecht\"/>"+newline);
  bw.write("        </owl:onProperty>"+newline);
  bw.write("      </owl:Restriction>"+newline);
  bw.write("    </owl:unionOf>"+newline);
  bw.write("    </owl:Class>"+newline);
  bw.write("  </owl:equivalentClass>"+newline);
  bw.write("  </owl:Class>"+newline+newline);

  bw.write("  <owl:Class rdf:about=\"#Maennliche_Person\">"+newline);
  bw.write("    <owl:disjointWith rdf:resource=\"#Weibliche_Person\"/>"+newline);
  bw.write("  </owl:Class>"+newline+newline);

 }
}
```

```
    else {
    bw.write("  </Firma>"+newline+newline);
    }

  }

//Organisation
if (  (!name_orga_firma.equals("")) & (check_art_firma_string.equals(""))  )
  {
    bw.write("  <owl:Class rdf:ID=\"Organisation\"/>"+newline+newline);

    bw.write("  <owl:Class rdf:ID=\"Strasse\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Postfach\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Hausnummer\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Postleitzahl\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Ort\"/>"+newline+newline);

    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_strasse\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_postfach\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_hausnummer\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_postleitzahl\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_ort\"/>"+newline+newline);

    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_mitarbeiter\">"+newline);
    bw.write("    <owl:inverseOf>"+newline);
    bw.write("      <owl:ObjectProperty rdf:ID=\"ist_mitarbeiter\"/>"+newline);
    bw.write("    </owl:inverseOf>"+newline);
    bw.write("  </owl:ObjectProperty>"+newline+newline);

    bw.write("  <Organisation rdf:ID=\""+name_orga_firma+"\">"+newline);

    bw.write("    <ist_adressat>"+newline);
    bw.write("      <Postalische_Anschrift rdf:about=\"#Anschrift_Organisation\"/>"+newline);
    bw.write("    </ist_adressat>"+newline);


    if (!strasse.equals("")) {

      bw.write("    <hat_strasse>"+newline);
      bw.write("      <Strasse rdf:ID=\""+strasse+"\"/>"+newline);
      bw.write("    </hat_strasse>"+newline);
    }

    if (!postfach.equals("")) {

      bw.write("    <hat_postfach>"+newline);
      bw.write("      <Postfach rdf:ID=\""+postfach+"\"/>"+newline);
      bw.write("    </hat_postfach>"+newline);
    }
    if (!hausnummer.equals("")) {

      bw.write("    <hat_hausnummer>"+newline);
      bw.write("      <Hausnummer rdf:ID=\""+hausnummer+"\"/>"+newline);
      bw.write("    </hat_hausnummer>"+newline);
    }

    bw.write("    <hat_postleitzahl>"+newline);
    bw.write("      <Postleitzahl rdf:ID=\""+postleitzahl+"\"/>"+newline);
    bw.write("    </hat_postleitzahl>"+newline);


    bw.write("    <hat_ort>"+newline);
    bw.write("      <Ort rdf:ID=\""+ort+"\"/>"+newline);
    bw.write("    </hat_ort>"+newline);

    if (!nachname1.equals("")) {
      bw.write("    <hat_mitarbeiter>"+newline);
      bw.write("      <Mitarbeiter rdf:ID=\""+nachname1+"\"/>"+newline);
      bw.write("    </hat_mitarbeiter>"+newline);

      bw.write("  </Organisation>"+newline+newline);

      bw.write("  <owl:Class rdf:ID=\"Person\"/>"+newline);

      bw.write("  <owl:Class rdf:ID=\"Mitarbeiter\">"+newline);
```

```
bw.write("    <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
bw.write("  </owl:Class>"+newline+newline);



bw.write("  <owl:Class rdf:ID=\"Geschlecht\">"+newline);
bw.write("    <owl:oneOf rdf:parseType=\"Collection\">"+newline);
bw.write("      <Geschlecht rdf:ID=\"Maennlich\"/>"+newline);
bw.write("      <Geschlecht rdf:ID=\"Weiblich\"/>"+newline);
bw.write("    </owl:oneOf>"+newline);
bw.write("  </owl:Class>"+newline+newline);

bw.write("  <owl:Class rdf:ID=\"Anrede\">"+newline);
bw.write("    <owl:oneOf rdf:parseType=\"Collection\">"+newline);
bw.write("      <Anrede rdf:ID=\"Herr\"/>"+newline);
bw.write("      <Anrede rdf:ID=\"Frau\"/>"+newline);
bw.write("    </owl:oneOf>"+newline);
bw.write("  </owl:Class>"+newline+newline);

bw.write("  <owl:Class rdf:ID=\"Titel\"/>"+newline);
bw.write("  <owl:Class rdf:ID=\"Vorname1\"/>"+newline);
bw.write("  <owl:Class rdf:ID=\"Vorname2\"/>"+newline);
bw.write("  <owl:Class rdf:ID=\"Vorname3\"/>"+newline+newline);

bw.write("  <owl:ObjectProperty rdf:ID=\"hat_anrede\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_titel\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname1\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"vorname1_hat_geschlecht\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname2\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname3\"/>"+newline+newline);


bw.write("  <Mitarbeiter rdf:ID=\""+nachname1+"\">"+newline);
bw.write("    <ist_mitarbeiter>"+newline);
bw.write("      <Organisation rdf:about=\"#"+name_orga_firma+"\"/>"+newline);
bw.write("    </ist_mitarbeiter>"+newline);
bw.write("  </Mitarbeiter>"+newline+newline);

bw.write("  <Person rdf:about=\"#"+nachname1+"\">"+newline);

if (!check_anrede_string.equals("Keine Anrede")) {

  bw.write("    <hat_anrede>"+newline);
  bw.write("      <Anrede rdf:ID=\""+check_anrede_string+"\"/>"+newline);
  bw.write("    </hat_anrede>"+newline);
}
if (!titel.equals("")) {

  bw.write("    <hat_titel>"+newline);
  bw.write("      <Titel rdf:ID=\""+titel+"\"/>"+newline);
  bw.write("    </hat_titel>"+newline);
}
if (!vorname1.equals("")) {

  bw.write("    <hat_vorname1>"+newline);
  bw.write("      <Vorname1 rdf:ID=\""+vorname1+"\"/>"+newline);
  bw.write("    </hat_vorname1>"+newline);
}
if (!vorname1_hat_geschlecht.equals("")) {

  bw.write("    <vorname1_hat_geschlecht>"+newline);
  bw.write("      <Geschlecht rdf:about=\"#"+vorname1_hat_geschlecht+"\"/>"+newline);
  bw.write("    </vorname1_hat_geschlecht>"+newline);
}
if (!vorname2.equals("")) {

  bw.write("    <hat_vorname2>"+newline);
  bw.write("      <Vorname2 rdf:ID=\""+vorname2+"\"/>"+newline);
  bw.write("    </hat_vorname2>"+newline);
}
if (!vorname3.equals("")) {

  bw.write("    <hat_vorname3>"+newline);
  bw.write("      <Vorname3 rdf:ID=\""+vorname3+"\"/>"+newline);
  bw.write("    </hat_vorname3>"+newline);
```

```
    }

    bw.write("  </Person>"+newline+newline);

    if (  (!check_anrede_string.equals("Keine Anrede")) | (!vorname1_hat_geschlecht.equals(""))  ) {

      bw.write("  <owl:Class rdf:ID=\"Maennliche_Person\">"+newline);
      bw.write("  <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
      bw.write("  <owl:equivalentClass>"+newline);
      bw.write("  <owl:Class>"+newline);
      bw.write("    <owl:unionOf rdf:parseType=\"Collection\">"+newline);
      bw.write("      <owl:Restriction>"+newline);
      bw.write("        <owl:hasValue rdf:resource=\"#Herr\"/>"+newline);
      bw.write("        <owl:onProperty>"+newline);
      bw.write("          <owl:FunctionalProperty rdf:about=\"#hat_anrede\"/>"+newline);
      bw.write("        </owl:onProperty>"+newline);
      bw.write("      </owl:Restriction>"+newline);
      bw.write("      <owl:Restriction>"+newline);
      bw.write("        <owl:hasValue rdf:resource=\"#Maennlich\"/>"+newline);
      bw.write("        <owl:onProperty>"+newline);
      bw.write("          <owl:FunctionalProperty rdf:about=\"#vorname1_hat_geschlecht\"/>"+newline);
      bw.write("        </owl:onProperty>"+newline);
      bw.write("      </owl:Restriction>"+newline);
      bw.write("    </owl:unionOf>"+newline);
      bw.write("    </owl:Class>"+newline);
      bw.write("  </owl:equivalentClass>"+newline);
      bw.write("  </owl:Class>"+newline+newline);

      bw.write("  <owl:Class rdf:ID=\"Weibliche_Person\">"+newline);
      bw.write("  <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
      bw.write("  <owl:equivalentClass>"+newline);
      bw.write("  <owl:Class>"+newline);
      bw.write("    <owl:unionOf rdf:parseType=\"Collection\">"+newline);
      bw.write("      <owl:Restriction>"+newline);
      bw.write("        <owl:hasValue rdf:resource=\"#Frau\"/>"+newline);
      bw.write("        <owl:onProperty>"+newline);
      bw.write("          <owl:FunctionalProperty rdf:about=\"#hat_anrede\"/>"+newline);
      bw.write("        </owl:onProperty>"+newline);
      bw.write("      </owl:Restriction>"+newline);
      bw.write("      <owl:Restriction>"+newline);
      bw.write("        <owl:hasValue rdf:resource=\"#Weiblich\"/>"+newline);
      bw.write("        <owl:onProperty>"+newline);
      bw.write("          <owl:FunctionalProperty rdf:about=\"#vorname1_hat_geschlecht\"/>"+newline);
      bw.write("        </owl:onProperty>"+newline);
      bw.write("      </owl:Restriction>"+newline);
      bw.write("    </owl:unionOf>"+newline);
      bw.write("    </owl:Class>"+newline);
      bw.write("  </owl:equivalentClass>"+newline);
      bw.write("  </owl:Class>"+newline+newline);

      bw.write("  <owl:Class rdf:about=\"#Maennliche_Person\">"+newline);
      bw.write("    <owl:disjointWith rdf:resource=\"#Weibliche_Person\"/>"+newline);
      bw.write("  </owl:Class>"+newline+newline);

    }
   }
   else {
   bw.write("  </Organisation>"+newline+newline);
   }

  }

 //Person
 if (  (name_orga_firma.equals("")) & (check_art_firma_string.equals("")) & (!nachname1.equals("")) )
  {
    bw.write("  <owl:Class rdf:ID=\"Person\"/>"+newline+newline);

    bw.write("  <owl:Class rdf:ID=\"Strasse\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Postfach\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Hausnummer\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Postleitzahl\"/>"+newline);
    bw.write("  <owl:Class rdf:ID=\"Ort\"/>"+newline+newline);

    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_strasse\"/>"+newline);
    bw.write("  <owl:ObjectProperty rdf:ID=\"hat_postfach\"/>"+newline);
```

```
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_hausnummer\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_postleitzahl\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_ort\"/>"+newline+newline);

bw.write("  <owl:Class rdf:ID=\"Geschlecht\">"+newline);
bw.write("    <owl:oneOf rdf:parseType=\"Collection\">"+newline);
bw.write("      <Geschlecht rdf:ID=\"Maennlich\"/>"+newline);
bw.write("      <Geschlecht rdf:ID=\"Weiblich\"/>"+newline);
bw.write("    </owl:oneOf>"+newline);
bw.write("  </owl:Class>"+newline+newline);

bw.write("  <owl:Class rdf:ID=\"Anrede\">"+newline);
bw.write("    <owl:oneOf rdf:parseType=\"Collection\">"+newline);
bw.write("      <Anrede rdf:ID=\"Herr\"/>"+newline);
bw.write("      <Anrede rdf:ID=\"Frau\"/>"+newline);
bw.write("    </owl:oneOf>"+newline);
bw.write("  </owl:Class>"+newline+newline);

bw.write("  <owl:Class rdf:ID=\"Titel\"/>"+newline);
bw.write("  <owl:Class rdf:ID=\"Vorname1\"/>"+newline);
bw.write("  <owl:Class rdf:ID=\"Vorname2\"/>"+newline);
bw.write("  <owl:Class rdf:ID=\"Vorname3\"/>"+newline+newline);

bw.write("  <owl:ObjectProperty rdf:ID=\"hat_anrede\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_titel\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname1\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"vorname1_hat_geschlecht\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname2\"/>"+newline);
bw.write("  <owl:ObjectProperty rdf:ID=\"hat_vorname3\"/>"+newline+newline);


bw.write("  <Person rdf:ID=\""+nachname1+"\">"+newline);

bw.write("    <ist_adressat>"+newline);
bw.write("      <Postalische_Anschrift rdf:about=\"#Anschrift_Person\"/>"+newline);
bw.write("    </ist_adressat>"+newline);

if (!check_anrede_string.equals("Keine Anrede")) {

  bw.write("    <hat_anrede>"+newline);
  bw.write("      <Anrede rdf:ID=\""+check_anrede_string+"\"/>"+newline);
  bw.write("    </hat_anrede>"+newline);
}
if (!titel.equals("")) {

  bw.write("    <hat_titel>"+newline);
  bw.write("      <Titel rdf:ID=\""+titel+"\"/>"+newline);
  bw.write("    </hat_titel>"+newline);
}
if (!vorname1.equals("")) {

  bw.write("    <hat_vorname1>"+newline);
  bw.write("      <Vorname1 rdf:ID=\""+vorname1+"\"/>"+newline);
  bw.write("    </hat_vorname1>"+newline);
}
if (!vorname1_hat_geschlecht.equals("")) {

  bw.write("    <vorname1_hat_geschlecht>"+newline);
  bw.write("      <Geschlecht rdf:about=\"#"+vorname1_hat_geschlecht+"\"/>"+newline);
  bw.write("    </vorname1_hat_geschlecht>"+newline);
}
if (!vorname2.equals("")) {

  bw.write("    <hat_vorname2>"+newline);
  bw.write("      <Vorname2 rdf:ID=\""+vorname2+"\"/>"+newline);
  bw.write("    </hat_vorname2>"+newline);
}
if (!vorname3.equals("")) {

  bw.write("    <hat_vorname3>"+newline);
  bw.write("      <Vorname3 rdf:ID=\""+vorname3+"\"/>"+newline);
  bw.write("    </hat_vorname3>"+newline);
}

if (!strasse.equals("")) {
```

```
  bw.write("    <hat_strasse>"+newline);
  bw.write("      <Strasse rdf:ID=\""+strasse+"\"/>"+newline);
  bw.write("    </hat_strasse>"+newline);
}

if (!postfach.equals("")) {

  bw.write("    <hat_postfach>"+newline);
  bw.write("      <Postfach rdf:ID=\""+postfach+"\"/>"+newline);
  bw.write("    </hat_postfach>"+newline);
}
if (!hausnummer.equals("")) {

  bw.write("    <hat_hausnummer>"+newline);
  bw.write("      <Hausnummer rdf:ID=\""+hausnummer+"\"/>"+newline);
  bw.write("    </hat_hausnummer>"+newline);
}

  bw.write("    <hat_postleitzahl>"+newline);
  bw.write("      <Postleitzahl rdf:ID=\""+postleitzahl+"\"/>"+newline);
  bw.write("    </hat_postleitzahl>"+newline);


  bw.write("    <hat_ort>"+newline);
  bw.write("      <Ort rdf:ID=\""+ort+"\"/>"+newline);
  bw.write("    </hat_ort>"+newline);

  bw.write("  </Person>"+newline+newline);


  if (  (!check_anrede_string.equals("Keine Anrede")) | (!vorname1_hat_geschlecht.equals(""))  ) {

    bw.write("  <owl:Class rdf:ID=\"Maennliche_Person\">"+newline);
    bw.write("  <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
    bw.write("  <owl:equivalentClass>"+newline);
    bw.write("  <owl:Class>"+newline);
    bw.write("    <owl:unionOf rdf:parseType=\"Collection\">"+newline);
    bw.write("      <owl:Restriction>"+newline);
    bw.write("        <owl:hasValue rdf:resource=\"#Herr\"/>"+newline);
    bw.write("        <owl:onProperty>"+newline);
    bw.write("          <owl:FunctionalProperty rdf:about=\"#hat_anrede\"/>"+newline);
    bw.write("        </owl:onProperty>"+newline);
    bw.write("      </owl:Restriction>"+newline);
    bw.write("      <owl:Restriction>"+newline);
    bw.write("        <owl:hasValue rdf:resource=\"#Maennlich\"/>"+newline);
    bw.write("        <owl:onProperty>"+newline);
    bw.write("          <owl:FunctionalProperty rdf:about=\"#vorname1_hat_geschlecht\"/>"+newline);
    bw.write("        </owl:onProperty>"+newline);
    bw.write("      </owl:Restriction>"+newline);
    bw.write("    </owl:unionOf>"+newline);
    bw.write("    </owl:Class>"+newline);
    bw.write("  </owl:equivalentClass>"+newline);
    bw.write("  </owl:Class>"+newline+newline);

    bw.write("  <owl:Class rdf:ID=\"Weibliche_Person\">"+newline);
    bw.write("  <rdfs:subClassOf rdf:resource=\"#Person\"/>"+newline);
    bw.write("  <owl:equivalentClass>"+newline);
    bw.write("  <owl:Class>"+newline);
    bw.write("    <owl:unionOf rdf:parseType=\"Collection\">"+newline);
    bw.write("      <owl:Restriction>"+newline);
    bw.write("        <owl:hasValue rdf:resource=\"#Frau\"/>"+newline);
    bw.write("        <owl:onProperty>"+newline);
    bw.write("          <owl:FunctionalProperty rdf:about=\"#hat_anrede\"/>"+newline);
    bw.write("        </owl:onProperty>"+newline);
    bw.write("      </owl:Restriction>"+newline);
    bw.write("      <owl:Restriction>"+newline);
    bw.write("        <owl:hasValue rdf:resource=\"#Weiblich\"/>"+newline);
    bw.write("        <owl:onProperty>"+newline);
    bw.write("          <owl:FunctionalProperty rdf:about=\"#vorname1_hat_geschlecht\"/>"+newline);
    bw.write("        </owl:onProperty>"+newline);
    bw.write("      </owl:Restriction>"+newline);
    bw.write("    </owl:unionOf>"+newline);
    bw.write("    </owl:Class>"+newline);
    bw.write("  </owl:equivalentClass>"+newline);
```

```
        bw.write("  </owl:Class>"+newline+newline);

        bw.write("  <owl:Class rdf:about=\"#Maennliche_Person\">"+newline);
        bw.write("    <owl:disjointWith rdf:resource=\"#Weibliche_Person\"/>"+newline);
        bw.write("  </owl:Class>"+newline+newline);

      }
  }


bw.write("</rdf:RDF>"+newline);

    bw.close();
}
catch (IOException ioe) {
System.out.println("Dateizugriff fehlgeschlagen!");
}
      }
}


  public void destroy() {
  }

  public boolean eingabe_ist_digit (String eingabe) {
    StringBuffer sb_ist_digit = new StringBuffer(eingabe);
    boolean gueltig = true;
    int sb_laenge_ist_digit=sb_ist_digit.length();
    for (int i = 0; i < sb_laenge_ist_digit; i++) {
      char ch=sb_ist_digit.charAt(i);
      if (gueltig == Character.isDigit(ch)) ;
      else {gueltig = false; break;}
    }
    return gueltig;
  }


  public boolean eingabe_ohne_space (String eingabe) {
    StringBuffer sb_ohne_space = new StringBuffer(eingabe);
    boolean gueltig = true;
    int sb_laenge_ohne_space=sb_ohne_space.length();
    for (int i = 0; i < sb_laenge_ohne_space; i++) {
      char ch=sb_ohne_space.charAt(i);
      if (gueltig == Character.isLetter(ch)) ;
      else {gueltig = false; break;}
    }
    return gueltig;
  }

  public boolean eingabe_mit_space (String eingabe) {
    StringBuffer sb_mit_space = new StringBuffer(eingabe);
    int sb_laenge_mit_space=sb_mit_space.length();
    int position_ist_space = 0;
    boolean gueltig = true;


    for (int j = 0; j < sb_laenge_mit_space; j++) {
      char ch=sb_mit_space.charAt(j);
      if (gueltig == Character.isWhitespace(sb_mit_space.charAt(0))) {gueltig = false; break;}
      if (gueltig == Character.isWhitespace(sb_mit_space.charAt(sb_laenge_mit_space-1))) {gueltig = false; break;}

      if (Character.isLetter(ch)) position_ist_space=0;
      if (Character.isWhitespace(ch)) position_ist_space+=1;

      if ( ((gueltig == Character.isLetter(ch)) | (gueltig == Character.isWhitespace(ch))) & position_ist_space <2  );
      else {gueltig = false; break;}
    }
    return gueltig;

  }

  public int geschlechtsbestimmung (String vorname) {
    int row_mann=0, row_frau=0, row_ort=0;
    int antwort=0;
    try {
```

```
        String query_mann = "SELECT * FROM MANN WHERE mann=\'"+vorname+"\'";
        Statement stmt = geschlechtdb.createStatement();
        ResultSet rs_mann = stmt.executeQuery(query_mann);
        while (rs_mann.next()) {
          antwort=1;
          row_mann++;
        }

        String query_frau = "SELECT * FROM FRAU WHERE frau=\'"+vorname1+"\'";

        ResultSet rs_frau = stmt.executeQuery(query_frau);
        while (rs_frau.next()) {
          antwort=2;
          row_frau++;
        }
      } catch (Exception ex) {System.out.println(ex.getMessage()); }


      if (row_mann==0 & row_frau==0) {
        antwort=0;
      }
      if (row_mann>0 & row_frau>0) {
        antwort=3;
      }
      return antwort;
    }

    public String ortsbestimmung (String plz) {
      int row_plz=0;
      String ortsbestimmung_ort="";

      try {
        //SELECT * FROM geodb_locations WHERE plz LIKE '%49080%';
        String query_plz = "SELECT * FROM geodb_locations WHERE plz LIKE\'%"+plz+"%\'";
        Statement stmt = opengeodb.createStatement();
        ResultSet rs_plz = stmt.executeQuery(query_plz);
        while (rs_plz.next()) {
          ortsbestimmung_ort += rs_plz.getString("ort")+",";
          row_plz++;
        }
       } catch (Exception ex) {System.out.println(ex.getMessage()); }
      return ortsbestimmung_ort;
    }

    public String plzbestimmung (String ort) {
      int row_ort=0;
      String plzbestimmung_plz="";
        try {
          String query_ort = "SELECT * FROM geodb_locations WHERE ort=\'"+ort+"\'";
          Statement stmt = opengeodb.createStatement();
          ResultSet rs_ort = stmt.executeQuery(query_ort);
          while (rs_ort.next()) {
            plzbestimmung_plz = rs_ort.getString("plz");
            row_ort++;
          }

        } catch (Exception ex) {System.out.println(ex.getMessage()); }
      return plzbestimmung_plz;
    }

  }
```

Erklärung

Hiermit versichere ich, die Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet zu haben. Die Magisterarbeit wurde noch keiner Prüfungsbehörde in gleicher oder anderer Form vorgelegt.

Osnabrück, den 03.05.2005 _____