STI – SEMANTIC TECHNOLOGY INSTITUTE

Universität Innsbruck
Semantic Technology Institute

Bakkalaureatsarbeit Implementierung eines OWL Imports in WSMO4J

Andreas Frankl
Matrikelnr.: 0125427
andreas.frankl@student.uibk.ac.at

betreut von Dr. Elena Simperl
und co-betreut von Nathalie Steinmetz

Innsbruck, 22. Februar 2008

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abstract

Diese Bakkalaureatsarbeit handelt vom Import von OWL-DL Ontologien (Web Ontology Language) nach WSML-Ontologien (Web Service Modeling Language). Zu diesem Zweck gibt es eine theoretische Einführung in das Semantic Web und einige zugehörige Technologien, nämlich OWL-DL, WSML-DL und die darunterliegende Beschreibungslogik. Damit der Import in beide Richtungen bijektiv funktioniert, werden die Schwierigkeiten des sogenannten Roundtripping genau behandelt. Basierend auf die existierende Transformation von WSML-DL nach OWL-DL wird das Mapping von OWL-DL nach WSML-DL diskutiert. Mit Hilfe der OWL API und WSMO4j wird der Import in Java implementiert.

Danksagung

Mein besonderer Dank gilt Nathalie Steinmetz, die mich bei dieser Arbeit betreut hat. Bei Unklarheiten war sie stets zur Stelle und konnte mit verständlichen Erklärungen alle Fragen beantworten. Nicht zuletzt durch die freundliche und engagierte Betreuung hat mir diese Arbeit viel Freude bereitet.
Bedanken möchte ich mich auch bei Mathias Ziegler, der die Arbeit Korrektur gelesen und nützliche Anmerkungen gemacht hat.

Kapitel 1

Einleitung

Im traditionellen World Wide Web werden statische Informationen gespeichert und Benutzer können Webseiten, die mit HTML aufgebaut sind, auf ihrem Browser betrachten. Die meisten Inhalte im Internet sind für Menschen aufbereitet, die von ihnen ohne Probleme verstanden und interpretiert werden können. Wenn ein Benutzer Informationen benötigt, wird mit Hilfe von Suchmaschinen danach gesucht. Solche Anwendungen verwenden statistische Verfahren, die Worthäufigkeiten und Wortnachbarschaften ermitteln, um Webseiten richtig einzuordnen. Auf diese Weise kann nur die Syntax berücksichtigt werden, und beispielsweise bei der Bildersuche im Internet stoßen diese herkömmlichen Methoden der Suchmaschinen schnell an ihre Grenzen. Deswegen möchte man Inhalte im Internet mit Semantik versehen, damit effektivere Suchformen verwendet werden können.

Nach einem Vorschlag von World-Wide-Web-Begründer Tim Berners Lee versucht man das Internet mit dem Semantic Web zu erweitern. Dadurch möchte man erreichen, dass das Internet nicht nur für Menschen, sondern auch für Maschinen lesbar wird. Ziel ist es, dass Softwareagenten autonom im Auftrag eines Users handeln und das Internet nach Informationen durchsuchen. Diese Agenten sollen miteinander kommunizieren und komplexe Aufgaben wie das Buchen einer Urlaubsreise mit der Auswahl eines Hotels und eines Fluges übernehmen können. Jegliche Sinnzusammenhänge bleiben dem automatisch gesteuerten Besucher bei statischen Webseiten verborgen, weil sie die Bedeutung von Links nicht verstehen.

Deswegen möchte man Beschreibungsstandards und Technologien entwickeln um Daten im Internet mit semantischen Metadaten auszuzeichnen. Metadaten sind maschinenlesbare Informationen über elektronische Ressourcen und andere Dinge[Lee et al., 2001]. Diese Initiative des Semantic Web hat das W3C Konsortium ins Leben gerufen. Es soll nicht nur Informationen darüber geben, wie Webseiten mit HTML auf dem Bildschirm dargestellt und wie sie syntaktisch mit XML aufgebaut werden, sondern es soll auch die Bedeutung der Metadaten

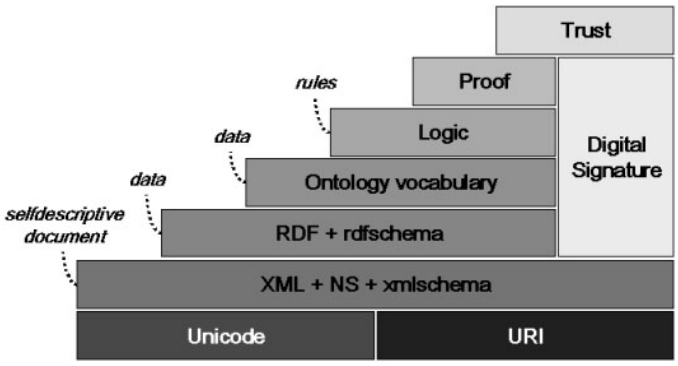

Abbildung 1.1: Semantic Web Schichten

vorhanden sein. Man möchte kein separates Internet konstruieren, sondern das Semantic Web soll eine Erweiterung des Bestehenden sein.

Damit tatsächlich die Semantik von Webseiten und ihren Inhalten von autonomen Softwareagenten herausgefunden werden können, muss man den Maschinen das Verstehen der gespeicherten Information ermöglichen, und es muss vorher bekannt sein, in welchen Beziehungen die Objekte zueinander stehen. Mittels Ontologien können Inhalte im Internet mit Metadaten verknüpft werden. Ontologien stammen aus der künstlichen Intelligenz und dienen zur Aufbereitung von Bedeutungen. Sie modellieren abgegrenzte Wissensgebiete, sogenannte Domänen, und strukturieren semantische Relationen hierarchisch, indem sie zum Beispiel Gruppen- und Untergruppenzugehörigkeiten definieren. Ontologien dienen als unterliegende Basistechnologie und werden im Kapitel 2.1 näher beschrieben.

Die Initiative des Semantic Web hat als Grundlage das Semantic Web Schichtenmodell [Koivunen and Miller, 2001], das aus aufeinander aufbauenden Schichten besteht. Für jede Schicht wurde vom W3C ein Standard definiert. In Abbildung 1.1 ist der Schichtenaufbau dargestellt.

Die URI (Unique Resource Identifiers)[Sun, 1998] als unterste Schicht dient zur Identifikation von Ressourcen. Die darüber liegende Schicht heißt XML + NS + xmlschema [Decker et al., 2000b] und gibt die Syntax und den Namensraum für die nächsten Schichten vor. Die Schicht RDF (Resource Description Framework) ist ein Modell und eine Syntax zur Beschreibung von Metadaten. Näheres zu diesem Thema wird im Abschnitt 2.3.1 erläutert. In der nächsten Schicht liegen Ontologiesprachen, wobei die Sprache OWL (Web Ontology Language) [**?**] zu einem W3C-Standard erhoben wurde. Diese Schicht wurde deswegen eingeführt, damit die Ausdruckstärke der RDF-Schicht erheblich erhöht werden kann. Die OWL versieht Inhalte mit einer Bedeutung, indem man Ontologien mit Klassen, Eigenschaften und den Beziehungen zueinander aufbaut.

Ontologien werden im Kapitel 2.1 näher besprochen. Mit der darunterliegenden Schicht RDF ist es nur möglich, Beziehungen zwischen Subjekten und Objekten zu beschreiben, zusätzliche Axiome zu definieren, ist nicht möglich. Zum Beispiel kann man nicht ausdrücken, dass ein Mensch entweder eine Frau oder ein Mann ist, und dass diese Varianten einander ausschließen. Im Kapitel 2.3.3 wird die Ontologiesprache OWL ausführlich vorgestellt und auf die zusätzlichen Beschreibungsmöglichkeiten eingegangen. Die Schichten Logic, das für die Definition von Regeln zuständig ist, und Proof, das diese Regeln auswertet, wurden noch nicht realisiert. Die oberste Schicht wird Trust genannt. Hier geht es darum, welche Aussage, die mit digitalen Signaturen signiert wird, vertrauenswürdig ist.

Um Ontologien für das Internet zu implementieren, gibt es neben dem W3C Standard OWL noch weitere Sprachen. Diese Bakkalaureatsarbeit befasst sich mit der Transformation von OWL in die Ontologiesprache WSML (Web Service Modelling Language). Dadurch soll sich dieser jüngeren Sprache ein breiteres Publikum erschließen, damit bereits bestehende OWL-Dokumente in WSML-Dokumente übertragen werden können. Beide Ontologiesprachen werden in Untersprachen strukturiert und die Transformation beschränkt sich auf OWL-DL nach WSML-DL. Diese Untersprachen basieren auf der logischen Beschreibungssprache Description Logic, aber mit unterschiedlichen Konstruktoren. Die zugrundeliegende Technologie wird im Kapitel 2.2 näher erläutert.

Die Description Logic ist eine Beschreibungssprache von Ontologien, und OWL bzw. WSML knüpfen an diese Logik an. Damit aber wirklich Semantik aus den beschriebenen Inhalten gelesen werden kann, gibt es verschiedene Schlussfolgerungssysteme. Diese sogenannten Reasoner basieren auf Wissensrepräsentationen und leiten neues Wissen ab. Zum Beispiel leitet man aus den Aussagen „Alle Griechen sind sterblich“ und „Sokrates ist ein Grieche“ ab, dass Sokrates sterblich ist. Je komplexer solche Aussagen sind, desto aufwändiger wird es für die Reasoner. Deswegen schließt man für das Internet einen Kompromiss zwischen Ausdruckmächtigkeit und Skalierbarkeit und beschränkt sich auf bestimmte Beschreibungskonstrukte, die im Kapitel Description Logics2.2 aufgezählt werden.

Diese Arbeit baut auf eine bereits implementierte Transformation in die Richtung von WSML nach OWL auf und konzentriert sich auf das sogenannte Roundtripping. Dies ist die Fähigkeit, Daten zu konvertieren und wieder zurück in das Original ohne Verluste oder andere Unterschiede zu transformieren. Im Kapitel 3.3 wird dieses Problem ausführlich diskutiert und eine Lösung präsentiert. Nachdem im Kapitel 2.3 die Sprache OWL und im Kapitel 2.4 die Sprache WSML vorgestellt wurden, wird im Kapitel Z eine Mappingtabelle, die für die Transformation von OWL-DL nach WSML-DL benötigt wird, präsentiert. Im Anschluss werden der implementierte Transformator und die verwendeten Programmierschnittstellen im Abschnitt 4 näher erläutert.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die zugrundeliegenden Technologien für den Import von OWL-Dokumenten vor. Zuerst werden Ontologien vorgestellt und Description Logics näher erläutert. Darauf aufbauend werden die Ontologiesprachen OWL und WSML beschrieben.

2.1 Ontologien

Ontologien stehen für eine Wissensrepräsentation eines formal definierten Systems von Begriffen und Relationen. Zusätzlich enthalten sie Regeln, um Schlussfolgerungen zu entwickeln und dass damit gleichzeitig ihre Gültigkeit gewährleistet bleibt. T. Gruber [Gruber, 1993] definiert Ontologien so: *„An ontology is an explicit specification of a conceptualization“*, frei übersetzt könnte dies bedeuten: *„Eine Ontologie ist eine eindeutige Beschreibung einer Konzeptualisierung.“* Ontologien könnte man für Softwareentwickler am besten mit UML-Klassendiagrammen in der objektorientierten Softwareentwicklung vergleichen. Dort werden einzelne Klassen mit ihren Attributen und die Beziehungen zueinander modelliert. Bei einer Ontologie werden stattdessen Begriffe mit ihren Eigenschaften und Relationen beschrieben. Zum Beispiel kann man die Ontologie von Menschen entwickeln. Menschen bestehen aus den Begriffen Frauen und Männer, die jeweils wieder Eltern haben, wobei es einen weiblichen Elternteil gibt, der eine Frau ist, und einen männlichen Elternteil, der ein Mann ist. Wenn Eltern Kinder haben, gibt es Mädchen und Buben, die wiederum den Begriffsgruppen Frauen und Männer angehören. Begriffe können in Hierarchien angeordnet werden, damit man sie genauer beschreiben und ihre Eigenschaften vererben kann. Menschen sind zum Beispiel Europäer, und Österreicher ist davon eine Unterklasse. In Ontologien gibt es Instanzen, die Objekte von Begriffen sind. Hans Bauer könnte eine Instanz des Begriffs Männer sein. Um Beziehungen zwischen Begriffen auszudrücken, gibt es in Ontologien sogenannte Relationen. Zum Beispiel kann man die Relation lebtIn konstruieren und sagen, dass der Mann Hans Bauer in dem Land Österreich lebt (Mann Hans Bauer leb-

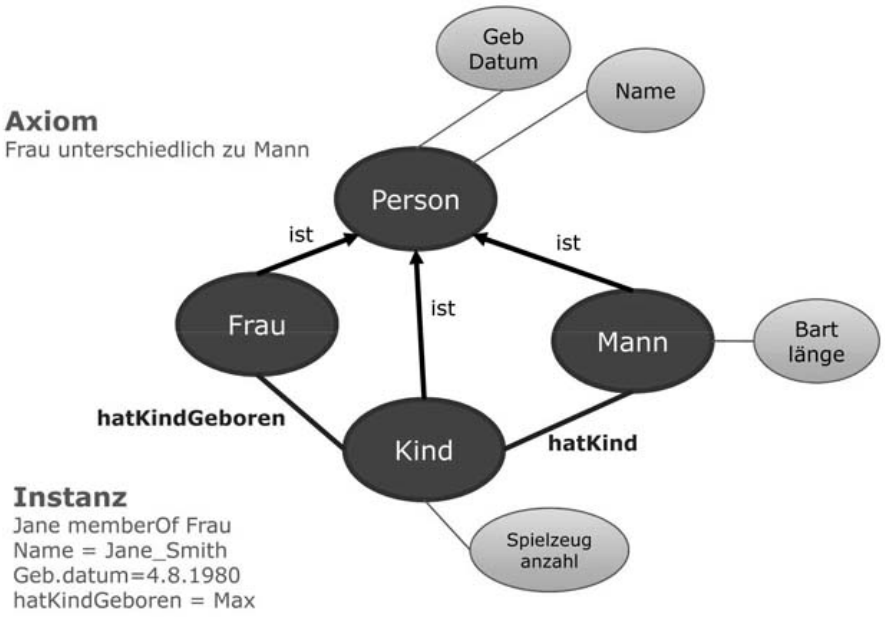

Abbildung 2.1: Ontologie

tIn Österreich). Auf diese Weise kann man die Ontologie immer mehr erweitern. In Abbildung 2.1 wurde eine Beispielontologie grafisch veranschaulicht.

Ontologien werden zum Beispiel in E-Commerce verwendet, wobei die Produktpalette in einer Ontologie strukturiert ist und so die Kommunikation zwischen Käufer und Verkäufer unterstützt wird. Die Firma DELL-Computers setzt beispielsweise Ontologien bei der PC-Konfiguration ein. Suchengines wie HotBot setzen Ontologien ein, damit Webseiten vom Benutzer leichter gefunden werden können. Und viele andere Webservices verwenden Ontologien um Vorgänge zu strukturieren.

2.2 Description Logics

Description Logics (DLs) ist eine Familie von Sprachen zur Wissensrepräsentation (Knowledge represenation KR) um die Welt, wie zum Beispiel Ontologien, zu definieren, indem man Konzepte (concepts) angibt. Konzepte mit ihren Attributen verwendet man, um Eigenschaften von Objekten und Individuen, die Instanzen von Konzepten sind, zu spezifizieren (world description). DLs sind mit formaler und logikbasierender Semantik ausgestattet. Die Beschreibungslogik ist in Sub- und Super-Konzepte (Subsumtion) in Form von Hierarchien strukturiert. Unter anderem ermöglicht dies logische Schlüsse aus den gegebe-

nen Informationen zu ziehen (Reasoning). [Baader et al., 2002]
Ein Beispiel für eine logische Schlussfolgerung wird hier mit der Rolle motherOf konstruiert, das hierarchisch unter parentOf liegt. Sei Anna die Mutter von Felix, so ist der logische Schluss, dass Anna auch Elternteil von Felix ist.

Ein wichtiges Gebiet der DLs ist die Komplexität von entscheidbaren Schlussfolgerungssystemen [Spalthoff and Hitzler, 2005]. Wenn eine DL sehr ausdrucksstark ist, führt die Komplexität zu Problemen beim Ziehen logischer Schlüsse. Haben die DLs weniger Ausdruckmöglichkeiten, dann können effizientere Schlussfolgerungssysteme unterstützt werden. Die Ausdruckmächtigkeit wird durch die Anzahl von Konstruktoren zur Erstellung von komplexen Descriptions bestimmt. Elementare Descriptions sind atomare Konzepte und atomare Rollen. Komplexe Descriptions werden aus diesen induktiv gebildet. In abstrakter Notation benennt man atomare Konzepte mit A, B und atomare Rollen mit R. C und D stehen für Konzept Descriptions. Im Folgenden wird die Description Language $\mathcal{AL}$ (Attributive Language) mit den in Abbildung 2.2 aufgelisteten Konstruktoren verwendet. Alle anderen Sprachen sind eine Erweiterung von $\mathcal{AL}$.

Konzept Descriptions müssen folgende Regeln beachten:
C;D →

A | (atomares Konzept)

$\top$ | (universales Konzept)

$\bot$ | (speziellstes Konzept)

$\neg$A | (atomare Negation)

C $\sqcap$ D | (Konzeptkonjunktion)

$\forall$R.C | (universelle Quantifikation)

$\exists$R.$\top$ | (beschränkte existentielle Quantifikation).

Die Negation von A bedeutet „alles außer A“ und die Konjunktion von C und D „sowohl C als auch D“. Die universelle Quantifikation steht für „alle x, deren alle Rollen R mit einem y vom Typ C sind“. Die beschränkte existentielle Quantifikation kann mit „alle x, die eine Beziehung vom Typ R haben“ übersetzt werden.

Um die Sprache $\mathcal{AL}$ zu verdeutlichen, wird ein Beispiel vorgestellt, das aus [Baader and Nutt, 2003] entnommen wurde. Angenommen Person und Female sind atomare Konzepte, dann ist Person $\sqcap$ Female und Person $\sqcap$ $\neg$Female Personen, die weiblich sind und Personen, die nicht weiblich sind. hasChild sei eine atomare Rolle. Damit kann man die Konzepte Person $\sqcap$ $\exists$hasChild.$\top$ und Person $\sqcap$ $\forall$hasChild.Female konstruieren, die Personen beschreiben, die Kinder haben und Personen, dessen Kinder weiblich sind. Wenn man das speziellste Konzept $\bot$ verwendet, kann man auch Personen beschreiben, die kein Kind haben mit

Konstruktor	Syntax	Semantik		
Atomares Konzept	A	$A^I \subseteq \Delta^I$		
Atomare Rolle	R	$R^I \subseteq \Delta^I \times \Delta^I$		
Transitive Rolle	$R \in R_+$	$R^I = (R^I)^+$	$\mathcal{AL}$	
Konzeptkonjunktion	$C \sqcap D$	$C^I \cap D^I$		
Allquantor	$\forall R.C$	$\{a \in \Delta^I \mid \forall b.<a,b> \in R^I \Rightarrow b \in C^I\}$		$\mathcal{S}$
Negation	$\neg C$	$\Delta^I \setminus C^I$	$\mathcal{C}$	
Existenzquantor	$\exists R.C$	$\{a \in \Delta^I \mid \exists b.<a,b> \in R^I \wedge b \in C^I\}$	$\mathcal{E}$	
Konzeptdisjunktion	$C \sqcup D$	$C^I \cup D^I$	$\mathcal{U}$	
Rollenhierarchie	$R \sqsubseteq S$	$R^I \subseteq S^I$	$\mathcal{H}$	
Abgeschlossene Klassen	$\{i_1,\ldots,i_n\}$		$\mathcal{O}$	
Inverse Rollen	R^-	$\{<a,b> \mid <b,a> \in R^I\}$	$\mathcal{I}$	
Zahlenrestriktion	$\geq nR$	$\{a \mid \#\{b.<a,b> \in R^I\} \geq n\}$	$\mathcal{N}$	
	$\leq nR$	$\{a \mid \#\{b.<a,b> \in R^I\} \leq n\}$		
Qualifizierte	$\geq nR.C$	$\{a \mid \#\{b.<a,b> \in R^I \wedge b \in C^I\} \geq n\}$	$\mathcal{Q}$	
Zahlenrestriktion	$\leq nR.C$	$\{a \mid \#\{b.<a,b> \in R^I \wedge b \in C^I\} \leq n\}$		

Abbildung 2.2: DL Konstruktoren

Person $\sqcap \forall$hasChild.$\bot$.

Um eine formale Semantik zu erstellen, werden Interpretationen $\mathcal{I}$, die als Domäne eine nichtleere Menge Δ^I hat und einer InterpretationsFunktion, die jedem atomaren Konzept eine Teilmenge $\mathcal{A}^I$ von Δ^I zuordnet, eingeführt. Die Semantik wird folgendermaßen formuliert:

$$\top^I = \Delta^I$$

$$\bot^I =$$

$$\neg(A)^I = \Delta^I \quad A^I$$

$$(C \sqcap D)^I = C^I \cap D^I$$

$$(\forall R.C)^I = \{a \in \Delta^I | \forall b. < a, b > \in R^I \Rightarrow b \in C^I\}$$

$$(\exists R.\bot)^I = \{a \in \Delta^I | \exists b. < a, b > \in R^I\}$$

Zwei Konzepte C, D seien äquivalent und man schreibt C $\equiv$ D wenn $C^I = D^I$ für alle Interpretationen $\mathcal{I}$. Demnach ist zum Beispiel $\forall$hasChild.Female $\sqcap$ $\forall$hasChild.Student äquivalent zu $\forall$hasChild.(Female $\sqcap$ Student).

$\mathcal{AL}$ wurde im Laufe der Zeit mit Konstruktoren erweitert um die Ausdrucksmächtigkeit zu vergrößern. Tabelle ttttttt beinhaltet weitere Konstruktoren, die auch für die Ontologiesprachen OWL und WSML relevant sind.

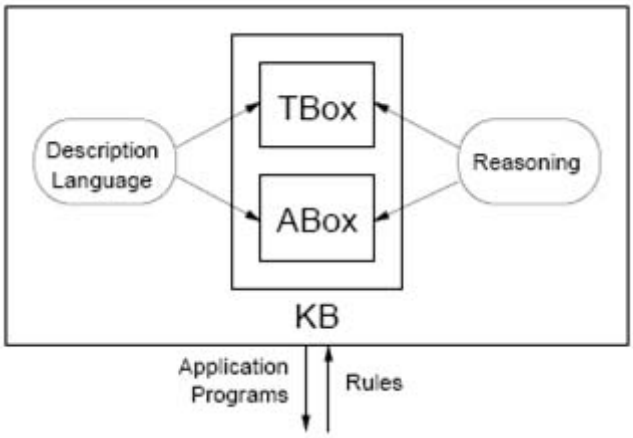

Abbildung 2.3: Architektur der DL Wissensrepräsentation

Knowledgebase Description Logics verwenden ein System um über den Inhalt der Knowledge Base (KB) zu schlussfolgern und diesen zu manipulieren. Dieses System besteht aus der sogenannten TBox und der ABox. Eine TBox beinhaltet die Terminologie der DL, also das Vokabular. Das Vokabular enthält Konzepte (Klassen), die Individuen kennzeichnen und unäre Prädikatensymbole sind. Zudem enthält das Vokabular Rollen, die binäre Prädikatensymbole sind und die Beziehungen zwischen Individuen kennzeichnen. Die TBox kann dazu verwendet werden, komplexe Descriptions, die aus Konzepten und Rollen bestehen, zu benennen. Eine der wichtigsten Inferenzen in TBoxen ist die Subsumierung und die Erfüllbarkeit von Konzepten. Die ABox umfasst Assertionen, die Behauptungen auf Individuen aufstellen, und enthält damit das Wissen von tatsächlich existierenden Dingen und Rollen. In Abbildung 2.3 wird die Architektur der Knowledgebase dargestellt.

Hierzu ein Beispiel:

TBox: Woman $\equiv$ Person $\sqcap$ Female

ABox: Woman(Jane)

//Woman ist in der TBox als weibliche Person definiert. In der ABox wird das Konzept Woman mit Jane instanziert.

TBox: Mother $\equiv$ Woman $\sqcap$ hasChild.Person

ABox: <Jane, Johnny> : hasChild

//Mother wird als Woman definiert, die mindestens ein Kind hat. Instanziert

wird die Rolle hasChild mit Jane als Mother, die Johnny als Kind hat.

2.3 OWL

OWL ist eine formale Sprache um Ontologien im Semantic Web darzustellen. OWL ist eine Weiterentwicklung von DAML+OIL und hat seine Wurzeln in RDF. Im Folgenden werden RDF und DAML+OIL kurz vorgestellt.

2.3.1 RDF

RDF ist eine Sprache um Metadaten von Webinhalten beschreiben zu können [Decker et al., 2000b]. Sie besteht aus RDF-Tripeln von Ressourcen, Eigenschaftselementen und Objekten. Eine Ressource steht für ein Subjekt, wie zum Beispiel eine komplette Webseite oder Teile davon. Aber auch im Internet nicht visualisierte Dinge, wie Bücher, Autos, Computer, oder andere Dinge werden als Ressourcen behandelt. Damit so ein Subjekt eine eindeutige Bezeichnung besitzt, kann es mit einer URI identifiziert werden. Die Eigenschaftselemente bzw. Prädikate haben die Aufgabe Subjekte zu beschreiben und die Verbindung mit den Objekten herzustellen. Objekte stehen für den Wert eines Prädikats. Ein Objekt kann in der einfachsten Form ein Literal oder wieder eine Ressource sein. Folgendes Beispiel wird das RDF-Schema verdeutlichen:

```
(http://www.example.com, dc:creator, Hans Bauer)
//Die Webseite http://www.example.com hat Hans Bauer als Autor.
```

in RDF/XML wird dieses Beispiel folgendermaßen definiert:

```
<rdf:Description rdf:about="http://www.example.com">
       <dc:creator>Hans Bauer</dc:creator>
</rdf:Description>
```

Doch ist man mit der Sprache sehr eingeschränkt und kann hauptsächlich Metadaten damit ausdrücken, die nur einen kleinen Teil der Informationen von einer Webseite, darstellen. Sehr deutlich werden die Nachteile, wenn man aufwendigere Ausdrucke formen möchte. Zum Beispiel kann man bei dem Prädikat „hatKind“ nicht festlegen, dass ein Mensch nur einen anderen Menschen und keine Katze als Kind haben kann. Man sollte festlegen können, welche Gruppen von Subjekten für bestimmte Prädikate in Frage kommen können. Weiters ist es mit RDF nicht möglich, bei Ressourcen oder Objekte gewisse Klassen unterschiedlich zueinander anzugeben. Dies möchte man zum Beispiel mit Frauen und Männer ausdrücken, die einander ausschließen. Es gibt bei RDF auch keine Möglichkeit, Klassen zu kombinieren oder deren Kardinalität anzugeben. Dies ist nur ein Teil der Defizite von RDF bei der Realisierung von Semantic Web und dem Versuch, explizit logische Schlüsse aus semantischen Angaben zu ziehen. Deswegen hat man mit DAML+OIL eine Semantic Web Sprache entwickelt, um die Ausdruckmächtigkeit zu erhöhen.

2.3.2 DAML+OIL

DAML [Pagels, 2000] ist eine Sprache um Ontologien zu beschreiben. Entwickelt wurde sie ab 2000 von einer Forschungseinrichtung des US-amerikanischen Verteidigungsministeriums und steht für DARPA (**D**efense **A**dvanced Research Projects Agency) **M**arkup **L**anguage. OIL [Decker et al., 2000a] ist hingegen eine Entwicklung aus Europa und wird mit Ontology Inference Layer übersetzt. Sie ist im Zuge des E-Commerce-Hypes in den 1990er entstanden. DAML+OIL [Connolly et al., 2001] ist ein Zusammenschluss dieser beiden ähnlichen Entwicklungen und bietet im Vergleich zu RDF differenziertere Ausdruckmöglichkeiten. DAML+OIL ist keine komplette Neuentwicklung, sondern baut auf RDF auf, damit bestehende RDF-Daten möglichst problemlos weiterverwendet werden können. Da OWL die Weiterentwicklung der Sprache DAML+OIL ist und deshalb starke Ähnlichkeiten aufweist, wird DAML+OIL nicht näher beschrieben und im nächsten großen Abschnitt OWL vorgestellt.

2.3.3 OWL Beschreibung

Die Web Ontology Language (OWL) [Bechhofer et al., 2004c] ist wahrscheinlich die meistgebräuchlichste formale Sprache um Ontologien im Semantic Web darzustellen. Sie ist mittlerweile seit Februar 2004 ein World Wide Web Consortium (W3C) Standard. Die Sprache verwendet das Schema von RDF um Klassen und Eigenschaften zu beschreiben und fügt wichtige Erweiterungen hinzu.

Eine OWL Ontologie ist ein RDF-Graph, der aus einer Menge von RDF-Tripeln besteht und wird in verschiedenen Syntaxen beschrieben:

- RDF/XML Syntax ist auch in RDF im Einsatz.
  zum Beispiel:

  ```
  <rdf:Description rdf:about="#Tree">
     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  </rdf:Description>
  ```

- Abstract Syntax, die für Menschen viel leichter zu lesen ist.
  Dasselbe Beispiel in RDF ist in der Abstract Syntax folgendermaßen übersetzt:

  ```
  Class(Tree partial)
  ```

- XML Presentation Syntax ist ein Dialekt der Abstract Syntax.
  zum Beispiel:

  ```
  <owlx:Class owlx:name="Tree" owlx:complete="false"/>
  ```

Im Folgendem werden Beispiele in der populären und einfacher zu lesenden Abstract Syntax angegeben.

2.3.4 OWL Untersprachen

In OWL gibt es drei Untersprachen, die für unterschiedliche Zwecke entwickelt worden sind und aufsteigend mächtig sind. OWL Lite ist für Nutzer designt worden, die erst beginnen, sich mit OWL auseinanderzusetzen, und diese können die Version von OWL einfach implementieren. OWL DL (DL steht für Description Logic) wurde entwickelt, um die existierende Description Logic zu unterstützen und die Konstrukte in OWL umzusetzen. Außerdem sollte es die Eigenschaften für die vorhandenen Schlussfolgerungssysteme (Reasoning) von DL übernehmen. OWL Full stellt die komplette OWL Sprache dar. Gewisse Einschränkungen von Lite oder DL gibt es bei OWL Full nicht.

OWL Full und OWL DL unterstützen dieselben OWL Sprachkonstrukte. Der Unterschied liegt darin, dass man in der Nutzung von RDF-Features unterschiedlich eingeschränkt ist. In OWL Full können Sprachkonstrukte frei vermischt werden. Es gibt hier keine strikte Trennung von Klassen, Eigenschaften, Instanzen und Datentypen. Man kann zum Beispiel den Namen einer Klasse auch als Instanz verwenden. Weiters gibt es in OWL Full keinen Unterschied von ObjectProperties und DatatypeProperties, den es in OWL DL und OWL Lite sehr wohl gibt. Die Sprache bietet die maximale Kompatibilität an RDF und ist damit die erste Adresse, wenn RDF-Nutzer mit OWL starten wollen.

OWL DL ist eine Untersprache von OWL, die gewisse Einschränkungen beinhaltet. Die Sprachkonstrukte müssen getrennt voneinander gehalten werden. Zum Beispiel darf der Name einer Klasse nicht gleichzeitig der Name einer Instanz sein. Eigenschaften für Objekte (ObjectProperties) und Eigenschaften für Datentypen (DatatypeProperties) sind in OWL DL unterschiedlich, deswegen können die Konstrukte, inverseOf, inverse functional, symmetric und transitive nicht für DatatypeProperties spezifiziert werden. Kardinalitätseinschränkungen dürfen bei keinen transitiven oder inversen Eigenschaften vorkommen. Wenn Klassen oder Eigenschaften sich auf andere Klassen oder Eigenschaften beziehen, müssen diese extra angegeben werden. Wenn zum Beispiel in einer Klassendeklaration eine Superklasse angegeben wird, muss diese Superklasse extra definiert werden. In OWL DL muss man sich an Richtlinien halten, die auf der Description Logic basieren, was aber den großen Vorteil hat, dass schon entwickelte Schlussfolgerungssysteme (OWL reasoner) unterstützt werden.

OWL Lite ist eine Untersprache von OWL DL und unterstützt nur eine Teilmenge an Sprachkonstrukten von OWL. Die Konstrukte oneOf, unionOf, complementOf, hasValue, disjointWith und DataRange werden nicht unterstützt. Es gibt noch einige andere Einschränkungen, die auf [Bechhofer et al., 2004a] nachgelesen werden können.

In OWL gibt es keine Reihenfolge der Konstrukte. Es spielt keine Rolle, ob am Anfang des Dokuments der Ontologie Header steht und ob Klassen vor Eigenschaften oder Instanzen deklariert werden. Für den Menschen ist es aber

oft leichter nachzuvollziehen, wenn der Header zu Beginn deklariert wird und die restlichen Konstrukte in einer gewissen Anordnung stehen. OWL unterstützt das Prinzip der offenen Welt. Das heißt, dass Beschreibungen von Ontologien nicht auf ein einzelnes Dokument beschränkt bleiben, sondern dass andere Ontologien leicht importiert werden können. Im Folgenden werden die Sprachkonstrukte von OWL DL näher besprochen.

2.3.5 OWL DL

Bevor man OWL Ausdrücke verwenden kann sollte man den Namensraum angeben, damit die Objekte der Ontologie eindeutig sind und besser lesbar werden. Das Standard OWL Vokabular gibt es bei http://www.w3.org/2002/07/owl.

2.3.5.1 Namensraum

Ein Namensraum dient zur eindeutigen Zuordnung von Objekten. Dabei fungiert der Namensraum als abstrakter Container, der einen zusätzlichen Bezeichner an jeden Namen anheftet. Dieser zusätzliche Bezeichner führt dazu, dass keine Namenskonflikte auftreten. Wenn es zum Beispiel in zwei Ontologien Objekte mit demselben Namen gibt, tritt wegen des unterschiedlichen Namensraums kein Namenskonflikt auf, und man kann diese beiden unterscheiden. Im Internet ist dies extrem wichtig, da sonst durch die Vernetzung Konflikte durch gleiche Bezeichnungen oft unvermeidbar sind. Ein typisches Beispiel einer Namensraum-Deklaration in OWL ist:

```
<rdf:RDF
    xmlns     ="http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"
    xml:base  ="http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"
    xmlns:food="http://www.w3.org/TR/2004/REC-owl-guide-20040210/food#"
    xmlns:owl ="http://www.w3.org/2002/07/owl#"
    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/2001/XMLSchema#">
```

Die erste Deklaration ist der Default Namensraum für die Ontologie. Die zweite Zeile deklariert die sogenannte Basis, die den Namensraum angibt, wenn keine andere URI angegeben ist. Die dritte Deklaration identifiziert den Namensraum von der unterstützten food-Ontologie. Die vierte ist eine Konvention um das OWL Vokabular einzuführen. OWL baut auf Konstrukte von RDF, RDFS und XML auf, die in den nächsten Deklarationen bestimmt werden.

2.3.5.2 Ontology Header

Nachdem der Namensraum definiert worden ist, werden die Informationen der Ontologie wie Name, Versionskontrolle, Import von anderen Ontologien oder Kommentare usw. deklariert.
Beispiel eines Headers:

```
Ontology( <http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine>
 Annotation(owl:imports <http://www.w3.org/TR/
 /2004/REC-owl-guide-20040210/food>)
 Annotation(rdfs:comment "An example OWL ontology")
 Annotation(rdfs:label "Wine Ontology")
```

Wenn in RDF/XML Syntax eine Ontologie spezifiziert wird, nämlich mit <owl:Ontology rdf:about=““>, und das Attribut about leer bleibt, dann wird der Name des Dokuments angenommen; oder wenn in xml:base in der Namespace Deklaration angegeben, dann beschreibt diese URI das Attribut about. Anschließend kann man Kommentare hinzufügen, wie mit priorVersion, das eine Versionsinformation beinhaltet. Für weitere Informationen über die Versionskontrolle sei auf [Bechhofer et al., 2004b] verwiesen. Mit dem Kommentar owl:imports kann man eine weitere Ontologie importieren. Die Annotation rdfs:comment verwendet man um die Ontologie oder andere Elemente zu kommentieren. Eine weitere Annotation ist rdfs:label, mit dem man die Ontologie kennzeichnen kann. Es können noch weitere Kommentare hinzugefügt werden, die dem Dublin Core Standard [S. Weibel and Wolf, 1998], wie Title, Creator und Date entsprechen.

2.3.5.3 Klassen

In OWL gibt es Klassen, die Ressourcen mit gleichen Eigenschaften gruppieren. Zu einer Klasse kann es eine Menge an Individuen geben, die auch ohne eine angegebene Klasse, von einer sogenannten anonymen Klasse, instanziert werden können. Es gibt sechs verschiedene Möglichkeiten Klassen zu deklarieren:

1. über eine URI Referenz als Klassen Bezeichner (class identifier)
2. eine Aufzählung (enumeration) von Instanzen
3. eine Eigenschaften Restriktion (property restriction)
4. ein Durchschnitt (intersection)
5. eine Vereinigung (union)
6. das Komplement (complement) von zwei oder mehr Klassen

Eine Typ 1 Klassendeklaration owl:Class, die eine Unterklasse von rdfs:Class ist, ist zum Beispiel:

```
Class (Human partial)
```

wobei Human, der Klassenname, eine URI ist. Diese URI ist wie in diesem Beispiel sehr kurz, weil angenommen wird, dass der Default Namespace deklariert wurde. Sonst müsste man als Klassen-ID zum Beispiel „http://www.w3.org/TR/2004/REC-owl-guide-20040210/life#Human“ angeben. Nach partial kann eine Superklasse definiert werden, wovon Human eine Unterklasse ist.
Zum Beispiel:

```
Class (Human partial Creature)
```

Hier wird deklariert dass Human eine Unterklasse von Creature ist. Man kann auch Unterklassen von anonymen Klassen, die in den nächsten Abschnitten vorgestellt werden, bilden und diese beliebig schachteln.

Ähnlich verhält es sich mit dem Schlüsselwort complete. Hier wird aber ausgedrückt, dass complete eine äquivalente Restriktion angibt. Als Restriktion kann man wie mit partial eine weitere Klasse oder andere Einschränkungen, die später diskutiert werden, angeben und beliebig schachteln.
Zum Beispiel:

```
Class (Human complete restriction)
```

Eine Typ-2 -Klassendeklaration konstruiert man mit **oneOf**, das verwendet wird um eine Liste von Instanzen einer Klasse anzugeben. Die folgende anonyme Klasse, die mit EquivalentClasses definiert wird, sagt aus, dass ihre Instanz a, b, c oder d sein kann.
Zum Beispiel:

```
EquivalentClasses(oneOf (a b c d))
```

Wenn man ausdrücken möchte, dass die Instanzen zweier Klassen keine gemeinsamen Individuen haben, verwendet man das Schlüsselwort **DisjointClasses**. Ein typisches Beispiel ist die Definition des Geschlechts, wenn man bestimmen möchte, dass man nicht gleichzeitig Mann oder Frau sein kann.
Zum Beispiel:

```
DisjointClasses(Woman Man)
```

Über eine Typ 3 **property restriction** kann man auch eine Klasse deklarieren. Es beschreibt eine anonyme Klasse mit Individuen, die dieselbe Restriktion haben. In folgendem Beispiel wird eine anonyme Klasse deklariert, die von der Eigenschaft hasParents eingeschränkt ist.
Zum Beispiel:

```
EquivalentClasses(restriction (hasParents Restriction))
```

Eine Einschränkung kann man mit **allValuesFrom** bilden und definieren, dass alle Instanzen, die die Eigenschaft hasParents betreffen, von einem bestimmten Wert sein sollen. In diesem Fall ist der Wert die Klasse Mensch.
Zum Beispiel:

```
EquivalentClasses (restriction (hasParents allValuesFrom (Human)))
```

someValuesFrom ist eine weitere Einschränkung und legt fest, dass eine Eigenschaft mindestens einen angegebenen Wert haben soll. Das folgende Beispiel definiert eine Klasse von Individuen, die mindestens einen Arzt als Elternteil haben sollen.
Zum Beispiel:

```
EquivalentClasses (restriction (hasParents someValuesFrom (Doctor)))
```

Mit **value** kann man auch eine anonyme Klasse definieren, wobei als Wert ein Individuum oder ein Datentyp angegeben sein kann. Folgende anonyme Klasse an Individuen hat Hans Bauer als Elternteil.
Zum Beispiel:

```
EquivalentClasses (restriction (hasParents value (HansBauer)))
```

In OWL kann man eine Kardinalität angeben, was zur Folge hat, dass Instanzen einer Klasse eine gewisse Anzahl an Werten besitzen soll. Mit **maxCardinality** bestimmt man die maximale Anzahl und im Beispiel wird definiert, dass man höchstens zwei Elternteile haben kann.
Zum Beispiel:

```
EquivalentClasses (restriction (hasParents maxCardinality (2)))
```

minCardinality bewirkt genau das Gegenteil von maxCardinality und man kann zum Beispiel ausdrücken, dass man mindestens zwei Elternteile haben muss:

```
EquivalentClasses (restriction (hasParents minCardinality (2)))
```

Das nächste OWL Konstrukt **cardinality** bewirkt, dass Instanzen einer Klasse genau N an Werten besitzen sollen. Diese Einschränkung ist eigentlich redundant, da man mit dem Paar minCardinality und maxCardinality jede Kardinalität ausdrücken kann.
Zum Beispiel:

```
EquivalentClasses (restriction (hasParents Cardinality (2)))
```

Typ 4-6 intersection, union und complement
Die drei Typen der Klassendeklaration Durchschnitt, Vereinigung und Komplement repräsentieren die Konstruktoren, die in Description Logic verwendet werden und stehen für AND, OR und NOT. Diese Sprachkonstrukte unterstützen die Möglichkeit Klassen zu schachteln.
Zum Beispiel:

```
EquivalentClasses (intersectionOf (
   oneOf(a b)
   oneOf(b c)))
```

Dieses Beispiel von **intersectionOf** drückt aus, dass es eine anonyme Klasse gibt, die den Durchschnitt von Aufzählungen bildet. In diesem Fall wäre das Ergebnis die Instanz b.

```
EquivalentClasses (unionOf (
   oneOf(a b)
   oneOf(b c)))
```

Mit **unionOf** wird eine Vereinigung gebildet und in dem Beispiel bewirkt es, dass die zwei vereinigten Aufzählungen das Ergebnis a, b und c liefert.

ComplementOf ist das Analogon des logischen NOT in der Description Logic. Das folgende Beispiel definiert eine anonyme Klasse mit der Eigenschaft, dass die Instanzen keine Instanz der Klasse Mensch ist:

```
EquivalentClasses (complementOf (Human))
```

2.3.5.4 Properties

In OWL konstruiert man Eigenschaften und Relationen mit Properties, wobei man zwei Arten von Properties unterscheidet, ObjectProperties und DatatypeProperties. Bei ObjectProperties weisen Individuen auf Individuen und bei DatatypeProperties weisen Individuen auf Datentypen. Die einfachste Form Properties auszudrücken wird in folgendem Beispiel angegeben und bedeutet, dass die Werte der Eigenschaft hasParents Individuen sind:

```
ObjectProperty(hasParents)
```

Wie es bei Klassen Unterklassen gibt, so gibt es auch bei Object und DatatypeProperties SubProperties. Zum Beispiel ist hasMother eine SubProperty von hasParents:

```
ObjectProperty(hasMother subPropertyOf hasParents)
```

Bei Eigenschaften gibt es eine Domäne, worauf sich die Eigenschaft bezieht und einen Bereich, wohin die Eigenschaft verweist. Domänen und Bereiche können deklarierte Klassen und anonyme Klassen sein. Bei DatatypeProperties ist der Bereich ein Datentyp.
Zum Beispiel:

```
ObjectProperty (hasParents
   domain (Mensch)
   range (Mensch))

DatatypeProperty (hasWeight
   domain (Mensch)
   range (xsd:integer))
```

Mit **inverseOf** kann man bei Properties die inverse Eigenschaft definieren. P1 und P2 seien Eigenschaften und x die Domäne und y der Bereich. Wenn gilt $P1(x, y) = P2(y, x)$, dann ist P2 die inverse Eigenschaft von P1.
Zum Beispiel:

```
ObjectProperty (hasChild inverseOf (hasParents))
```

Um auszudrücken, dass eine Eigenschaft nur einen Wert in dem Bereich haben darf, gibt es in OWL das Schlüsselwort Functional. $P(x, y)$ und $P(x, z) \rightarrow$

$y = z$ Das folgende Beispiel definiert die ObjectProperty Husband und gibt an, dass eine Frau nur einen Mann als Gatten haben kann:
Zum Beispiel:

```
ObjectProperty (biologicalMotherOf InverseFunctional
   domain (Woman)
   range (Human))
```

Das Schlüsselwort Transitive bewirkt, dass eine Eigenschaft P gilt: $P(x, y)$ und $P(y, z) \rightarrow P(x, z)$
Zum Beispiel:

```
ObjectProperty (localisedIn Transitive
   domain (Region)
   range (Region))
```

In OWL kann man mit dem Schlüsselwort **Symmetric** symmetrische Eigenschaften $P(x, y) = P(y, x)$ konstruieren. Im folgenden Beispiel sagt man aus, dass ein Mensch mit einem anderen Menschen befreundet ist und umgekehrt:

```
ObjectProperty (FriendOf Symmetric
   domain (Human)
   range (Human))
```

2.3.5.5 Individuen

OWL vertritt den Ansatz der OWA (Open World Assumption) im Gegensatz zu der Closed World Assumption, das heißt der Annahme, dass die Wissensbasis alle Individuen enthält. Die Existenz von weiteren Individuen ist bei der offenen Welt möglich, sofern sie nicht explizit ausgeschlossen wird.
In diesem Kapitel werden zwei Typen von Individuen diskutiert.

- Individuen, die Mitglieder (Instanz) von Klasse sind und ihre Eigenschaftswerte haben
- Individuen mit ihrer individuellen Identität

Das folgende Konstrukt gibt ein Beispiel für Individuen, die Mitglied einer Klasse sind:

```
Individual(Tosca type(Opera)
  value(hasComposer GiacomoPuccini)
  value(premierePlace Roma))
```

Somit ist GiacomoPuccini ein Individuum das Mitglied von dem Individuum Tosca, das eine Instanz von der Klasse Opera ist.

In folgendem Beispiel ist Hans Bauer vom Typ Mensch und seine Mutter ist HildeBauer.

```
Individual (HansBauer
   type (Human)
   value (hasMother HildeBauer))
```

Den zweiten Typ von Individuen gibt es deshalb, weil man im Internet nicht zwingend zwischen unterschiedliche Namen unterscheidet. Deswegen muss man in OWL explizit Mitglieder von Klassen bestimmen, die gleich oder unterschiedlich sind, auch wenn sie unterschiedlich heißen. Mit **SameIndividual** legt man fest, dass Individuen auf dasselbe Ding zeigen sollen.
Zum Beispiel:

```
SameIndividual (HermannMaier Herminator)
```

Genau das Gegenteil bewirkt man mit **DifferentIndividuals**. Hier wird explizit gesagt, dass Instanzen unterschiedlich zu anderen sein sollen.
Zum Beispiel:

```
DifferentIndividuals(Herminator Terminator Rambo)
```

2.3.6 OWL-DL und SHOIN(D)

OWL-DL baut auf die Description Logic SHOIN auf [Horrocks et al., 2003]. SHOIN ist eine Erweiterung von AL mit dem Komplement C und beinhaltet weitere Konstruktoren. Damit ist es zusätzlich möglich, Gleichheit und Ungleichheit zwischen Individuen, abgeschlossene Klassen, Zahlenrestriktionen, Subrollen und Rollenäquivalenz, Inverse und transitive Rollen und Datentypen darzustellen.

- $\mathcal{ALC}$: Attribute Language with Complement
- $\mathcal{S}$: ALC + Rollentransitivität
- $\mathcal{H}$: Subrollenbeziehung
- $\mathcal{O}$: abgeschlossene Klassen
- $\mathcal{I}$: inverse Rollen
- $\mathcal{N}$: Zahlenrestriktionen $\leq$n R etc.
- $(\mathcal{D})$: Datentypen

Folgende Tabelle vergleicht die OWL-DL Konstruktoren und Axiome mit der DL Syntax und gibt entsprechende Beispiele an:

OWL Ausdruck	DL Syntax	Beispiel
intersectionOf	$C_1 \sqcap ... \sqcap C_n$	Human $\sqcap$ Male
unionOf	$C_1 \sqcup ... \sqcup C_n$	Doctor $\sqcup$ Lawyer
complementOf	$\neg C$	$\neg$Male
oneOf	$x_1 \sqcup ... \sqcup x_n$	{john} $\sqcup$ {mary}
allValuesFrom	$\forall P.C$	$\forall$hasChild.Doctor
someValuesFrom	$\exists P.C$	$\exists$hasChild.Lawyer

maxCardinality	$\leq nP$	$\leq$1hasChild
minCardinality	$\geq nP$	$\geq$2hasChild
subClassOf	$C_1 \sqsubseteq C_2$	Human $\sqsubseteq$ Animal $\sqcap$ Biped
equivalentClass	$C_1 \equiv C_2$	Man $\equiv$ Human $\sqcap$ Male
disjointWith	$C_1 \sqsubseteq \neg C_2$	Male $\sqsubseteq \neg$Female
sameIndividualAs	$x_1 \equiv x_2$	{HermannMaier} $\equiv$ {Herminator}
differentFrom	$x_1 \sqsubseteq \neg x_2$	{john} $\sqsubseteq \neg$ {peter}
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter $\sqsubseteq$ hasChild
equivalentProperty	$P_1 \equiv P_2$	cost $\equiv$ price
inverseOf	$P_1 \equiv P_2^-$	hasChild $\equiv$ hasParent$^-$
transitiveProperty	$P^+ \sqsubseteq P$	ancestor$^+ \sqsubseteq$ ancestor
functionalProperty	$\top \sqsubseteq \leq 1P$	$\top \sqsubseteq \leq$ 1hasMother
inverseFunctionalProperty	$\top \sqsubseteq \leq 1P$	$\top \sqsubseteq \leq$ 1hasSSN$^-$

Tabelle 2.1: OWL Ausdrücke und DL Syntax

2.4 WSML

Die Web Service Modeling Language [de Bruijn et al., 2005] ist wie OWL eine Ontologiesprache und umfasst formale Syntaxen und Semantik für das Meta-Modell Web Service Modeling Ontology (WSMO). Damit ist es möglich Ontologien, Semantic Web Services, Goals und Mediatoren zu beschreiben. Ontologien wurden im Kapitel 2.1 ausführlich beschrieben. Web Services stellen den Zugang zu bestimmte Funktionalitäten zur Verfügung. Goals sind Zielsetzungen des Clients beim Aufruf von Web Services. Mediatoren sind Vermittler, die für die Interoperabilität zwischen WSMO-Elementen verantwortlich sind. WSML basiert auf Description Logics, Prädikatenlogik der ersten Stufe und Logische Programmierung, die für Semantik im Internet verwendet werden. Anders als bei OWL gibt es für WSML unterschiedlich komplexe Schlussfolgerungssysteme und man hat dafür WSML-Varianten, nämlich WSML-Core, WSML-DL, WSML-Flight, WSML-Rule und WSML-Full, entwickelt. Der Grad der Ausdruckmächtigkeit einer Sprachvariante führt zu einfacheren bzw. komplexeren oder besser skalierenden bzw. nicht so gut skalierenden logischen Formalismen.

WSML-Core ist eine Kombination von Description-Logics und Horn Logik (ohne Funktionssymbole und Gleichheit) und wurde mit Datentypen erweitert.

WSML-DL ist eine Erweiterung von WSML-Core um die Description Logics SHIQ und deckt OWL-DL ab, der fast vollständig in WSML implementierbar ist. Diese WSML-Variante ist mit OWL-DL kompatibel.

WSML-Flight erweitert WSML-Core in die Richtung der Logischen Programmierung.

WSML-Rule erweitert WSML-Flight um Funktionssymbole und der Unabhängigkeit von Variablen in logischen Ausdrücken.

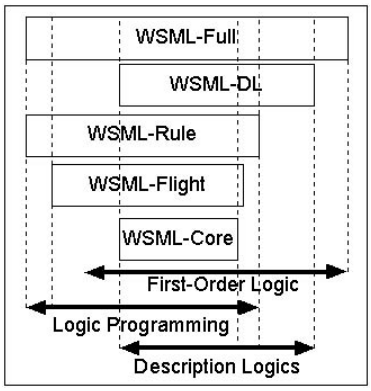

Abbildung 2.4: WSML Varianten

WSML-Full vereint die WSML-Varianten für Description Logics und der Logischen Programmierung.

Abbildung 2.4 veranschaulicht die Ausdruckmächtigkeit der WSML Varianten. Im Folgenden wird WSML-DL vorgestellt, da in dieser Arbeit das Hauptaugenmerk auf die Transformation von Ontologien von OWL-DL nach WSML-DL liegt.

2.4.1 WSML-DL Syntax

Jede WSML-Spezifikation beginnt mit dem Schlüsselwort wsmlVariant mit einem anschließenden Bezeichner als IRI, die für eine WSML-Variante steht. Bezeichner werden im Kapitel 2.4.1.2 näher beschrieben. Für die Variante WSML-DL definiert man:

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-dl"
```

2.4.1.1 Namensraum

Nachdem man die WSML-Variante bestimmt hat, kann man einen optionalen Block für den Namensraum angeben, der mit dem namespace Schlüsselwort beginnt und von eine oder mehrere Namespace-Referenzen, die in einer geschwungenen Klammer begrenzt sind, gefolgt wird. Ein Namensraum kann als Teil einer IRI gesehen werden und macht die Objekte in einer Ontologie eindeutig. Nebenbei erspart es dem Ersteller Schreibarbeit und trägt zur Übersichtlichkeit des Dokuments bei. Namensräume wurden im Abschnitt von OWL im Kapitel 2.3.5.1 näher beschrieben. Bezeichner wie eine IRI werden im nächsten Abschnitt 2.4.1.2 erläutert. Das nächste Beispiel beinhaltet mehrere Namespace-Referenzen:

```
namespace {_"http://www.example.org/ontologies/example#",
```

```
dc _"http://purl.org/dc/elements/1.1#",
foaf _"http://xmlns.com/foaf/0.1/",
wsml _"http://www.wsmo.org/wsml-syntax#",
loc _"http://www.wsmo.org/ontologies/location#",
oo _"http://example.org/ooMediator#"}
```

2.4.1.2 Bezeichner

Bezeichner beginnen in WSML stets mit einem Unterstrich und werden dann mit Anführungszeichen eingefasst. Drei Arten zur Identifikation von Objekten stehen zur Verfügung:

Eine IRI (Internationalized Resource Identifier) [Duerst and Suignard, 2005] ist eine Möglichkeit von WSML ein Bezeichner zu sein und ist die internationale Form der Uniform Resource Identifier (URI). Damit ist es möglich Objekte (beispielsweise beginnt eine IRI im Internet mit „http://“) zu identifizieren.
Zum Beispiel:

```
_"http://www.example.org/ontologies/example#"
```

Ein weiterer Bezeichner ist die Menge der Datentypen, die in WSML in der Form
_datatype(value) angegeben werden. datatype steht für einen Datentyp und value für einen gewissen Wert. Hierbei gibt es Abkürzungen für Integer-Zahlen, wo man bloß die Zahl angeben muss, und für Strings, wo man Zeichen nur innerhalb von Anführungszeichen schreiben kann.
Zum Beispiel:

```
_date(1900,2,14)
```

Die dritte Möglichkeit Objekte zu identifizieren ist in einer anonymen Form und wird verwendet, wenn keine konkrete Angabe eines Bezeichners nötig ist und man von einem anderen anonymen Bezeichner unterscheiden möchte. In WSML werden zwei Arten in nummerierte (_#1, _#2, . . .) und unnummerierte IDs (_#) unterschieden. Mehr dazu kann man unter [de Bruijn Holger Lausen et al., 2005] nachlesen. Um von OWL-DL nach WSML-DL zu transformieren werden anonyme Bezeichner nicht weiter benötigt.

2.4.1.3 Header

Nach dem empfohlenen Namensraum-Block folgt der Header der WSML-Spezifikation, der einen Block von nicht funktionellen Eigenschaften, importierte Ontologien und die verwendeten Mediatoren enthalten kann. Die sogenannten nonFunctionalProperties können im Header als auch bei jedem anderen WSML-Element angegeben werden und dienen dazu zusätzliche Informationen, wie Version und Autor, zu vermerken. Sie beginnen mit nonFunctionalProperties und enden mit endNonFunctionalProperties oder in der kürzeren Form nfp und endnfp. Dazwischen schreibt man Eigenschaften wie wsml#version,

wsml#accuracy, wsml#financial, usw. und die empfohlenen Attribute von Dublin Core [S. Weibel and Wolf, 1998]. Zusätzlich kann der Nutzer eigene Attribute hinzufügen.
Zum Beispiel:

```
nonFunctionalProperties
    dc#title hasValue "WSML example ontology"
    dc#subject hasValue "family"
    dc#description hasValue "fragments of a family ontology to
                             provide WSML examples"
    dc#date hasValue _date("2007-12-24")
    dc#format hasValue "text/html"
    dc#language hasValue "en-US"
    dc#rights hasValue _"http://www.deri.org/privacy.html"
    wsml#version hasValue "$Revision: 1.191 $"
endNonFunctionalProperties
```

Dieser Block der nonFunctionalProperties fließt nicht in den logischen Part der Sprache ein und bleibt für den Reasoner demnach völlig unberücksichtigt. Zugriff erlangt man nur durch eine API.

Weiters kann man im Header mit importsOntology und einer Liste von IRIs Ontologien importieren. Dieses Konstrukt ist vergleichbar mit dem OWL-Konstrukt owl:imports.
Zum Beispiel:

```
importsOntology {_"http://www.wsmo.org/ontologies/location",
    _"http://xmlns.com/foaf/0.1"}
```

Als drittes Header-Element kann man mit usesMediator Mediatoren angeben. Zum Beispiel:

```
usesMediator _"http://example.org/ooMediator"
```

Mediatoren werden nicht weiter besprochen, da dies für den Import von Ontologien von OWL-DL nach WSML-DL keine Verwendung findet.
Die Ontologie Spezifikation unterteilt man in eine konzeptuelle und eine logische Syntax.

2.4.1.4 Konzeptuelle Syntax

Eine WSML Ontologie-Spezifikation beginnt mit dem Schlüsselwort ontology und einer IRI für die Bezeichnung der Ontologie. Wenn keine IRI angegeben ist, dient der Ort der Ontologie zur Identifikation.
Zum Beispiel:

```
ontology family
```

Eine Ontologie besteht aus Klassen, in WSML werden sie Konzepte genannt, aus Relationen, aus Instanzen, aus Relationsinstanzen und aus Axiomen. In den folgenden Abschnitten werden diese WSML-Elemente vorgestellt.

Konzepte Um Ressourcen mit gleichen Eigenschaften zusammenzufassen, gibt es in WSML Konzepte, die mit concept und einer optionalen Bezeichnung definiert werden. Daraufhin kann man eine oder mehrere Superkonzepte mit subConceptOf angeben. Das Konzept erbt dann die Attribute der übergeordneten Konzepte.
Zum Beispiel:

```
concept Human subConceptOf {Primate, LegalAgent}
  nonFunctionalProperties
    dc#description hasValue "concept of a human being"
  endNonFunctionalProperties
  hasMother impliesType Human
  hasWeight ofType _float

//Hier wird das Konzept Human mit zwei Attributen und einer
//Beschreibung definiert. Außerdem wird spezifiziert, dass
//Primate und LegalAgent Superkonzepte von Human sind.
```

WSML erlaubt zwei Arten von Attributdefinitionen, nämlich beschränkte Definitionen mit dem Schlüsselwort ofType und abgeleitete Definitionen mit impliesType. Bei dem Beispiel Human werden zwei Attribute definiert, hasMother als abgeleitetes Attribut, das wiederum auf das Konzept Human zeigt und das beschränkte hasWeight, das auf den Datentyp _float zeigt.

Relationen Mit Relationen kann man in WSML Beziehungen zwischen Klassen und Datentypen herstellen. Sie werden mit dem Schlüsselwort relation gefolgt von einer Bezeichnung definiert. Sie können mit dem Schlüsselwort subRelationOf als Subrelation einer anderen Relation definiert werden. WSML erlaubt nur binäre Relationen und die Parameter einer Relation sind strikt geordnet.
Zum Beispiel:

```
relation hasBirthdateOfHuman (impliesType Human, ofType _date)
     subRelationOf hasBirthdate

//Hier wird die Relation hasBirthdateOfHuman definiert, die das
//Konzept Human mit dem Datentyp _date verbindet. Zusätzlich wird
//hasBirthdate als Superrelation festegelegt.
```

Instanzen von Konzepte Instanzen sind Objekte von Konzepten und werden mit dem Schlüsselwort instance optional gefolgt von einem Namen definiert. Anschließend kann man mit dem Schlüsselwort memberOf eine Anzahl an Konzepte angeben, die instanziert werden. Nach einem optionalen nonFunctionalProperties-Block kann man mit hasValue die expliziten Werte der Attribute von den instanzierten Konzepten definieren. Instanzen können sowohl in Ontologien als auch in privaten Datenbanken, die auf die Ontologie zeigt, spezifiziert werden.
Zum Beispiel:

```
instance Jack memberOf {Human}
   hasMother hasValue Jane
   hasWeight hasValue _float(82,4)
```

Instanzen von Relationen So wie man Konzepte instanziert, kann man auch Relationen instanzieren. Mit dem Schlüsselwort relationinstance wird eine Instanz von einer Relation begonnen. Anschließend müssen beide Werte einer Relation in WSML-DL in Klammern angegeben werden.
Zum Beispiel:

```
relationinstance hasBirthdateOfHuman ( Jack, _date(1982,2,11) )
```

Axiome Axiome gibt es in WSML um zusätzliche Informationen von Konzepten und Attributen beschreiben zu können. Auf diese Weise kann man zum Beispiel Subkonzepte und Attributdefinitionen neu spezifizieren. Ein Axiom beginnt mit dem Schlüsselwort axiom mit einer Bezeichnung und dem Schlüsselwort definedBy. Anschließend fügt man einen logischen Ausdruck hinzu, der mit einem Punkt beendet wird. Logische Ausdrücke werden im nächsten Abschnitt näher vorgestellt.
Zum Beispiel beschreibt folgendes Axiom die Äquivalenz der Klasse Human und die Vereinigung von Primate und LegalAgent:

```
axiom axiomOfHuman
   definedBy
      ?x memberOf Human equivalent
          (?x memberOf Primate and
           ?x memberOf LegalAgent).
```

2.4.1.5 Logische Ausdrücke in WSML-DL

In den logischen Ausdrücken gibt es eine neue Art von Identifikatoren, nämlich Variablen, die für Konzepte, Attribute, Instanzen, Relationsargumente und Attributswerten stehen. Variablen starten mit einem Fragezeichen und einer Anzahl an alphanumerischen Zeichen.

Atomare Moleküle In der logischen Syntax gibt es zwei Arten von Molekülen. a-Moleküle sind in der Form ?x memberOf C und b-Moleküle sind in der Form ?x[propid hasValue ?y]. Folgende Moleküle, hier Elemente genannt, sind demnach möglich, die logisch verknüpft werden können:

- C1 subConceptOf C2
- ?x memberOf C1
- ?x [propid ofType datatype]
- ?x [propid impliesType C]

- ?x [propid hasValue ?y]

?x und ?y stehen für Variablen, C1 und C2 stehen für Konzepte, datatype steht für einen Datentyp und propid steht für den Bezeichner eines Attributs. Mit subConceptOf kann man das Superkonzept eines Konzepts angeben. Mit memberOf kann man das Konzept (den Typ) einer Variable definieren. In WSML-DL zeigt man mit ofType auf einen Datentyp und mit impliesType auf ein Konzept. Das Schlüsselwort hasValue zeigt auf eine Variable.
Zum Beispiel:

```
//Woman ist ein Subkonzept vom Konzept Person
Woman subConceptOf Person

//die Instanz Jane ist vom Typ Woman
Jane memberOf Woman

//Die Eigenschaft hasWeight zeigt auf den Datentyp Integer
?x[hasWeight ofType integer]

//die Eigenschaft hasMother zeigt auf das Konzept Woman
?x[hasMother impliesType Woman]

//die Eigenschaft hasColor zeigt auf die Instanz White
?x[hasColor hasValue White ]
```

WSML-DL Descriptions In WSML gibt es eine Reihe an Ausdrücken mit denen man Elemente logisch verknüpft, wobei Elemente auch für sich alleine valide sind. Auf diese Weise kann man logische Ausdrücke auf vielfache Weise verschachteln. Solche verknüpften Elemente nennt man in WSML-DL Descriptions. E1 und E2 seien Elemente und ?x1,...,?xn seien Variablen. Folgende logische Verknüpfungen stehen zur Auswahl:

- E1 and E2
- E1 or E2
- E1 implies E2
- E1 impliedBy E2
- E1 equivalent E2
- neg(E1)
- forall ?x1,...,?xn (E1)
- exists ?x1,...,?xn (E1)

and und or, bewirken den Durchschnitt und die Vereinigung von zwei Elementen. Mit implies und impliedBy erreicht man eine rechte und eine linke Implikation und mit equivalent eine duale Implikation. Mit neg kann man einen Ausdruck negieren und mit den Schlüsselworten forall bzw exists kann man den Allquantor bzw den Existenzquantor bilden.
Zum Beispiel:

```
//Mit equivalent werden zwei Ausdrücke auf Gleichheit verknüpft
//und mit or bildet man die Vereinigung von Woman und Man
?x memberOf Human
  equivalent
  ?x memberOf Woman or ?x memberOf Man .

//Man und Woman schließen einander aus. Wenn ein Objekt vom
//Konzept Man ist folgt daraus, dass dieses Objekt nicht vom
//Typ Woman ist.
?x memberOf Man
  implies
  neg (?x memberOf Woman) .

//Dieses Beispiel bedeutet, dass jeder Mensch einen Vater hat,
//der ein Mensch ist und jeder Vater ist ein Mensch.
?x memberOf Human
  implies
  exists ?y ( ?x[hasFather hasValue ?y]  and ?y memberOf Human )
  and
  forall ?y ( ?x[hasFather hasValue ?y] implies ?y memberOf Human ) .
```

2.4.2 WSML-DL und SHIQ(D)

WSML-DL basiert auf der Description Logic SHIQ. Damit ist es möglich Gleichheit und Ungleichheit zwischen Individuen, qualifizierende Zahlenrestriktionen, Subrollen und Rollenäquivalenz, Inverse und transitive Rollen und Datentypen darzustellen.

- $\mathcal{ALC}$: Attribute Language with Complement
- $\mathcal{S}$: ALC + Rollentransitivität
- $\mathcal{H}$: Subrollenbeziehung
- $\mathcal{I}$: inverse Rollen
- $\mathcal{Q}$: Qualifizierende Zahlenrestriktionen $\leq$n R.C etc.
- $(\mathcal{D})$: Datentypen

OWL DL	**DL Syntax**	**WSML-DL**
c1 subClassOf c2	$c1 \subseteq c2$	c1 subConceptOf c2
lexpr unionOf rexpr	$lexpr \cup rexpr$	lexpr or rexpr
lexpr intersectionOf rexpr	$lexpr \cap rexpr$	lexpr and rexpr
complementOf expr	$\neg expr$	neg expr
allValuesFrom expr	$\forall R.expr$	forall Y expr
someValuesFrom	$\exists R.expr$	exists Y expr
X type id	$X : id$	X memberOf id
property	$< X1, X2 >: id$	X1[id hasValue X2]
subPropertyOf	$id1 \subseteq$ ∀id2.id3	id1[id2 impliesType id3]
equivalentClass(c1, c2)	$c1 \equiv c2$	c1 :=: c2

Tabelle 2.2: OWL DL, DL Syntax und WSML-DL

Kapitel 3

Mapping

Damit man OWL-DL Dokumente in WSML-DL Ontologien transformieren kann, wurde ein Mapping entwickelt. Mapping bedeutet die Abbildung von Daten in andere Daten. Die Regeln für den Import von OWL-DL nach WSML-DL werden im nächsten Abschnitt vorgestellt. Anhand der konstruierten Tabelle kann man OWL-DL Konstrukte in die Sprache WSML-DL abbilden.

3.1 Mapping OWL-DL → WSML-DL

In OWL- DL gibt es eine Anzahl an Konstrukte, die verschachtelt werden können. Deswegen werden im Folgenden diese Konstrukte Element genannt. Element steht entweder für eine Klasse Class oder einer anonymen Klasse. Eine anonyme Klasse kann eine PropertyRestriction, Intersection, Union oder Complement sein. In WSML-DL ist es ähnlich. Hier ist ein Element ein Konzept concept oder eine WSML-DL-Description, die in Kapitel 2.4.1.5 vorgestellt wurden.

OWL-DL	**WSML-DL**
Namespace	
Namespace (id = nid)	**namespace** id _"nid"
Ontology Header	
Ontology (oid)	**ontology** _"oid"
Annotation (annotID "text")	**nfp** annotID **hasValue** "text" **endnfp**
RDF-XML Syntax: <owl:imports rdf:resource="oid"/> *Abstract Syntax:* **Annotation** (owl:imports oid)	**nfp** http://owl2wsmltransformation/Import **hasValue** "oid" **endnfp**
Annotation (http://owl2wsmltransformation/Import oid)	**importsOntology** _"oid "
Klassen	
Class (classid **partial**)	**concept** classid

Class (classid1 **partial** Element1 ... Elementn)	**concept** classid1 **subConceptOf** Element1 ... Elementn
equivalent - complete	
Class (classid **complete** Element1 ... Elementn)	?x **memberOf** classid **equivalent** Element1 **equivalent** ... **equivalent** Elementn
disjoint	
DisjointClasses(classid1 . . . classidn)	?x **memberOf** classid1 **implies** **neg** (?x **memberOf** classid2) **and** . . . **and** **neg** (?x **memberOf** classidn)
PropertyRestriction	
allValuesFrom: alle	
restriction (propid **allValuesFrom** (classid1)) **ObjectProperty**(propid **range**(classid2)	concept classid1 propid impliesType classid2 **nfp** annotID **hasValue** "einfache Klasse ohne sub" **endnfp**
restriction (propid **allValuesFrom** (Element))	**forall** ?x (?y[propid **hasValue** ?x] **implies** Element)
someValuesFrom: mind. 1	
restriction (propid **someValuesFrom** (Element)	**exists** ?x (?y[propid **hasValue** ?x] **and** Element))
value	
restriction (propid **value** (Element))	?x[propid **hasValue** Element]
Kardinalität	
restriction (propid **minCardinality** (n))) **ObjectProperty**(propid **range**(classid)	**exists** ?x1, .. ?xn(?x[propid **hasValue** ?x1] **and** . . . **and** ?x[propid **hasValue** ?xn] **and** ?x1 **memberOf** classid . . . ?xn **memberOf** classid **and** **neg**(?x1 = ?x2)) **and** . . . **and** **neg**(?xn-1 = ?xn))

restriction (propid **maxCardinality** (n))) **ObjectProperty**(propid **range**(classid)	**forAll** ?x1, .. ?xn(?x[propid **hasValue** ?x1] **and** . . . **and** ?x[propid **hasValue** ?xn] **implies** (?x1 = ?x2) **or** . . . **or** (?xn-1 = ?xn) **and** ?x1 **memberOf** classid . . . ?xn **memberOf** classid
restriction (propid **cardinality** (n))) **ObjectProperty**(propid **range**(classid)	**exists** ?x1, .. ?xn(?x[propid **hasValue** ?x1] **and** . . . **and** ?x[propid **hasValue** ?xn] **and** ?x1 **memberOf** classid . . . ?xn **memberOf** classid **and** **neg**(?x1 = ?x2)) **and** . . . **and** **neg**(?xn-1 = ?xn)) **and** **forAll** ?x1, .. ?xn(?x[propid **hasValue** ?x1] **and** . . . **and** ?x[propid **hasValue** ?xn] implies (?x1 = ?x2)) **or** . . . **or** (?xn-1 = ?xn))
Intersection	
intersectionOf (Element . . . Element)	Element **and** . . . **and** Element
Union	
unionOf (Element . . . Element))	Element **or** . . . **or** Element
Complement	
complementOf (Element)	**neg** Element

Properties	
ObjectProperty (propid **domain** (classid1) **range** (classid2))	**concept** classid1 propid **impliesType** classid2 *oder* [1] **relation** propid (**impliesType** classid1, **impliesType** classid2) **nfp** annotID **hasValue** "relation" **endnfp**
DatatypeProperty (dataID **domain** (classid) **range** (xsd:Datatype))	**concept** classid dataID **ofType** Datatype *oder* [1] **relation** dataID (**impliesType** classid, **ofType** Datatype) **nfp** annotID **hasValue** "relation" **endnfp**
mit super **ObjectProperty** (propid1 **super** (propid2) **domain** (classid1) **range** (classid2))	**concept** classid1 propid1 **impliesType** classid2 axiom axClass1 **defindedBy** ?x[propid1 **hasValue** ?y] **impliedBy** ?x [propid2 **hasValue** ?y] *oder* [1] **relation** propid1 (**impliesType** classid1, **impliesType** classid2) **subrelationOf** propid2 **nfp** annotID **hasValue** "relation" **endnfp**
SubPropertyOf (propid1 propid2)	?x[propid1 **hasValue** ?y] impliedBy ?x [propid2 **hasValue** ?y] *oder* [1] **relation** propid1 **subrelationOf** propid2 **nfp** annotID **hasValue** "relation" **endnfp**
Transitive: P(x,y) und P(y,z) → P(x,z)	
ObjectProperty (propid **Transitive** **domain** (classid1) **range** (classid2))	?x[propid hasValue ?z] impliedBy ?x[propid **hasValue** ?y] **and** ?y[propid **hasValue** ?z]
Symmetric: P(x,y)=P(y,x)	
ObjectProperty (propid **Symmetric** **domain** (classid1) **range** (classid2))	?x[propid **hasValue** ?y] **impliedBy** ?y[propid **hasValue** ?x]

[1] Erklärung in Kapitel Roundtripping 3.3

Functional: P(x,y) und P(x,z) → y=z (maxCard=1)	
ObjectProperty (propid **Functional** **domain** (classid1) **range** (classid2))	**forall** ?x, ?y, ?z(?x[propid **hasValue** ?y] **and** ?x[propid **hasValue** ?z] **implies** (?y = ?z)) **and** Element
Inverse: P1(x,y)=P2(y,x)	
ObjectProperty (propid1 **inverseOf** (propid2))	?x[propid1 **hasValue** ?y] **impliedBy** ?y[propid2 **hasValue** ?x]
InverseFunctional: P(y,x) und P(z,x) → y=z	
ObjectProperty (propid1 **InverseFunctional** **domain** (classid1))	**forall** ?x, ?y, ?z(?y[propid **hasValue** ?x] **and** **range**(classid2) ?z[propid **hasValue** ?x] **implies** (?y = ?z)) **and** Element
EquivalentProperties(propid1 , .. propidn)	?x[propid1 **hasValue** ?y] **equivalent** ?x[propid2 **hasValue** ?y] **equivalent** . . . ?x[propidn **hasValue** ?y]
Individual	
Individual (indid)	**instance** indid
Individual (indid **type** (classid))	**instance** indid **memberOf** classid
Individual (indid **type** (classid) **value** (propid1 propValue) . . . **value** (propidN propValue))	**instance** indid **memberOf** classid propid **hasValue** propValue . . . propidN **hasValue** propValue
ObjectProperty(propid) Individual1 . . . Individualn	**relationInstance** propid (Individual1, . . . , Individualn) nfp annotid hasValue „relationInstance“ endnfp *oder* [1] **instance** indid1 propid **hasValue** indid2 **instance** indid2
same as	
SameIndividual (indid1 . . . indidn)	indid1 = indid2 **and** . . . **and** indid1 = indidn
DifferentIndividuals(indid1 . . . indidn)	indid1 = **neg**(indid2) **and** . . . **and** indid1 = **neg**(indidn)

[1] Erklärung in Kapitel Roundtripping 3.3

Listen existieren in wsml-dl nicht	
EnumeratedClass (classid indid1 indid2 ... indidn	**nfp** http://owl2wsmltransformation/Enumeration1 **hasValue** "indid1" http://owl2wsmltransformation/Enumeration2 **hasValue** "indid2" ... http://owl2wsmltransformation/EnumerationN **hasValue** "indid3" **endnfp**

Tabelle 3.1: Mappingtabelle

Mapping Beispiele Damit die Mapping-Tabelle besser nachvollziehbar ist, wird im Folgenden eine Tabelle mit Beispielen zu allen Konstrukten angeführt. Die OWL-DL Dokumente sind frei erfunden und wurden mit der im Zuge dieser Bakkalaureatsarbeit implementierten Importfunktion transformiert.

OWL-DL	WSML-DL
Namespace	
Namespace (xmlns = http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#)	**namespace** xmlns _" http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"
Ontology Header	
Ontology (http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine)	**ontology** _" http://www.w3.org/TR/2004/ REC-owl-guide-20040210/wine"
Annotation (rdfs:comment "this is an example of an ontology")	**nfp** rdfs:comment **hasValue** " this is an example of an ontology" **endnfp**
Klassen	
Class (Human **partial**)	**concept** Human
Class (Human **partial** Creature Animal)	**concept** Human **subConceptOf** Creature Animal
equivalent – complete	
Class (Human **complete** **unionOf** (Woman Man)	**concept** Human **axiom** _" axiomOfHuman" **definedBy** ?x **memberOf** Human **equivalent** ?x **memberOf** Woman **or** ?x **memberOf** Man
Disjoint	
DisjointClasses(Pasta Meat Fruit)	?x **memberOf** Pasta **implies** **neg** (?x **memberOf** Meat) **and** **neg** (?x **memberOf** Fruit)
PropertyRestriction	
allValuesFrom: alle	
Class(Wine **partial** PotableLiquid restriction(hasMaker allValuesFrom(Winery)))	**concept** Wine **subConceptOf** PotableLiquid **axiom** _" axiomOfWine" **definedBy** ?x **memberOf** Wine **implies** **forall** ?x (?y[hasMaker **hasValue** ?x] **implies** ?x **memberOf** Winery) .

Class (SimpleClass **partial** restriction(anyProperty allValuesFrom(AnyClass)))	**concept** SimpleClass **nfp** annotID **hasValue** “simple“ **endnfp** prop **impliesType** AnyClass
someValuesFrom: mind. 1	
Class(TexasThings **complete** restriction(locatedIn someValuesFrom(TexasRegion)))	**concept** TexasThings **axiom** _” axiomOfTexasThings” **definedBy** ?x **memberOf** TexasThings **equivalent** (**exists** ?x (?y[locatedIn **hasValue** ?x] **and** ?x **memberOf** TexasRegion)).
Value	
Class(WhiteWine **complete** restriction(hasColor value (White)))	**concept** WhiteWine **axiom** _”axiomOf WhiteWine” **definedBy** ?x **memberOf** WhiteWine **equivalent** (?x[hasColor **hasValue** White]).
Kardinalität	
Class(Person **complete** restriction(hasParent minCardinality(2)))	**axiom** _“ axiomOfPerson“ **definedBy** ?x **memberOf** _“Person“ **equivalent** (**exists** ?x1,?x2(?x[_¨hasParent“ **hasValue** ?x1] **and** ?x[_¨hasParent“ **hasValue** ?x2] **and** ?x1 **memberOf** _“ Person“ **and** ?x2 **memberOf** _“ Person“ **and** **neg** (?x1 = ?x2))).
Class(QuadBike **complete** restriction(hasWheels maxCardinality(4)))	**axiom** _“axiomOfQuadBike“ **definedBy** ?x **memberOf** _“QuadBike“ **equivalent** (**forall** ?x1,?x2,?x3,?x4(?x[_“hasWheels“ **hasValue** ?x1] **and** ?x[_“hasWheels“ **hasValue** ?x2] **and** ?x[_“hasWheels“ **hasValue** ?x3] **and** ?x[_“hasWheels“ **hasValue** ?x4] **implies** ?x1 = ?x2 **or** ?x1 = ?x3 **or** ?x1 = ?x4 **or** ?x2 = ?x3 **or** ?x2 = ?x4 **or** ?x3 = ?x4) **and** ?x1 **memberOf** _“QuadBike“ **and** ?x2 **memberOf** _“QuadBike“ **and** ?x3 **memberOf** _“QuadBike“ **and** ?x4 **memberOf** _“QuadBike“).

Class(Butcher **complete** restriction(hasKnifes Cardinality(3)))	**axiom** _"axiomOfButcher" **definedBy** ?x **memberOf** _"Butcher" **equivalent** (**exists** ?x1,?x2,?x3(?x[_"hasKnife" **hasValue** ?x1] **and** ?x[_"hasKnife" **hasValue** ?x2] **and** ?x[_"hasKnife" **hasValue** ?x3] **and** ?x1 **memberOf** _"Butcher" **and** ?x2 **memberOf** _"Butcher" **and** ?x3 **memberOf** _"Butcher" **and** **neg** (?x1 = ?x2) **and** **neg** (?x1 = ?x3) **and** **neg** (?x2 = ?x3)) **and** **forall** ?x1,?x2,?x3(?x[_"hasKnife" **hasValue** ?x1] **and** ?x[_"hasKnife" **hasValue** ?x2] **and** ?x[_"hasKnife" **hasValue** ?x3] **implies** ?x1 = ?x2 **or** ?x1 = ?x3 **or** ?x2 = ?x3)).
Intersection	
Class(WhiteWine **complete** intersectionOf(Wine White))	**concept** WhiteWine **axiom** _"axiomOf WhiteWine" **definedBy** ?x **memberOf** WhiteWine **equivalent** (?x **memberOf** Wine **and** ?x **memberOf** White).
Union	
Class (Human **complete** **unionOf** (Woman Man))	**concept** Human **axiom** _"axiomOfHuman" **definedBy** ?x **memberOf** Human **equivalent** ?x **memberOf** Woman **or** ?x **memberOf** Man
Complement	
Class(NonConsThing **complete** complementOf(ConsumableThing))	**concept** NonConsThing **axiom** _"axiomOfNonConsThing" **definedBy** ?x **memberOf** _"NonConsThing" **equivalent** (**neg** (?x **memberOf** _"ConsThing")).
Properties	
ObjectProperty (hasMother **domain** (Person) **range** (Mother))	**concept** Person hasMother **impliesType** Mother $oder$[1] **relation** hasMother (**impliesType** Person, **impliesType** Mother) **nfp** annotID **hasValue** "relation" **endnfp**

[1]Erklärung in Kapitel Roundtripping 3.3

DatatypeProperty (hasWeight **domain** (Person) **range** (xsd:integer))	**concept** Person hasWeight **ofType** _integer $oder^1$ **relation** hasWeight (**impliesType** Person, **ofType** _integer) **nfp** annotID **hasValue** "relation" **endnfp**
$mit\ super$ **ObjectProperty** (hasMother **super** (hasParent) **domain** (Person) **range** (Person))	**concept** Person hasMother **impliesType** Person **axiom** axiomOfHasMother **definedBy** ?x[hasMother **hasValue** ?y] **impliedBy** ?x [hasParent **hasValue** ?y] $oder^1$ **relation** hasMother (**subrelationOf** hasParent, **impliesType** Person, **impliesType** Person) **nfp** annotID **hasValue** "relation" **endnfp**
SubPropertyOf (hasMother hasParent)	?x[hasMother **hasValue** ?y] **impliedBy** ?x [hasParent **hasValue** ?y] $oder^1$ **relation** propid1 **subrelationOf** propid2 **nfp** annotID **hasValue** "relation" **endnfp**
Transitive: P(x,y) und P(y,z) → P(x,z)	
ObjectProperty (locatedIn **Transitive** **domain** (Thing) **range** (Region))	?x[locatedIn **hasValue** ?z] **impliedBy** ?x[locatedIn **hasValue** ?y] **and** ?y[locatedIn **hasValue** ?z]
Symmetric: P(x,y)=P(y,x)	
ObjectProperty (adjacentRegion **Symmetric** **domain** (Region)) **range** (Region))	?x[adjacentRegion **hasValue** ?y] **impliedBy** ?y[adjacentRegion **hasValue** ?x]
Functional: P(x,y) und P(x,z) → y=z (maxCard=1)	
ObjectProperty (hasVintageYear **Functional** **domain** (Vintage) **range** (VintageYear))	**forall** ?y, ?z(?x[hasVintageYear **hasValue** ?y] **and** ?x[hasVintageYear **hasValue** ?z] **implies** (?y = ?z)) **and** ?y **memberOf** Vintage **and** ?z **memberOf** Vintage
Inverse: P1(x,y)=P2(y,x)	
ObjectProperty (hasMother **inverseOf** (hasChild))	?x[hasMother **hasValue** ?y] **impliedBy** ?y[hasChild **hasValue** ?x]

InverseFunctional: P(y,x) und P(z,x) → y=z	
ObjectProperty (producesWine **InverseFunctional**)	**forall** ?y, ?z(?y[producesWine **hasValue** ?x] **and** ?z[producesWine **hasValue** ?x] **implies** (?y = ?z))
EquivalentProperties(hasAge , hatAlter)	?x[hasAge **hasValue** ?y] **equivalent** ?x[hatAlter **hasValue** ?y]
Individual	
Individual (Valkyrie)	**instance** Valkyrie
Individual type (Tosca (opera))	**instance** Tosca **memberOf** opera
Individual (Tosca type (opera) value (hasComposer GiacomoPuccini) value (premiereDate xsd:date(1900-01-14)))	**instance** indid **memberOf** opera hasComposer **hasValue** "GiacomoPuccini" premiereDate **hasValue** _date(1900,2,14)
ObjectProperty (hasDriver) **Individual** (NikiLauda) **Individual** (McLaren1984 type (Car) value (hasDriver NikiLauda))	**instance** McLaren1984 **memberOf** Car hasDriver **hasValue** NikiLauda **instance** NikiLauda *oder* [1] **relationInstance** hasDriver (McLaren1984, NikiLauda) nfp annotid hasValue „relationInstance" endnfp **instance** McLaren1984 **memberOf** Car **instance** NikiLauda
same as	
SameIndividual (HermannMaier Herminator)	**axiom** _"axiomOfHermannMaier" **definedBy** _"HermannMaier" = _"Herminator". **axiom** _"axiomOfHerminator "**definedBy** _"Herminator" = _"HermannMaier".
DifferentIndividuals(CarX BikeZ)	**axiom** _"axiomOfCarX" **definedBy** **neg** (_"CarX" = _" BikeZ") . **axiom** _"axiomOfBikeZ" **definedBy** **neg** (_"BikeX" = _" CarZ") .
Listen existieren in WSML-DL nicht List	
Class(WineColor **partial** WineDescriptor) **EnumeratedClass**(WineColor Red Rose White)	**concept** WineColor **subConceptOf** WineDescriptor nfp _"http://owl2wsmltransformation/Enumeration1" **hasValue** "Red" _"http://owl2wsmltransformation/Enumeration2" **hasValue** "Rose" _"http://owl2wsmltransformation/Enumeration3" **hasValue** "White" **endnfp** **instance** Red **memberOf** Thing **instance** Rose **memberOf** Thing **instance** White **memberOf** Thing

Tabelle 3.2: Tabelle mit Mappingbeispielen

[1] Erklärung in Kapitel Roundtripping 3.3

3.2 Einschränkungen

Die Schwierigkeit beim Mapping von OWL-DL nach WSML-DL ist die Tatsache, dass beide Ontologiesprachen auf verschiedene Description Logics basieren. OWL-DL unterstützt die Konstruktoren SHOIN(D) und WSML-DL die Konstruktoren SHIQ(D).

WSML-DL akzeptiert im Gegensatz zu OWL-DL den Konstruktor O nicht, mit dem man abgeschlossene Klassen erstellen kann. In OWL kann man mit oneOf Aufzählungen von Instanzen bilden, die in WSML nicht möglich sind. Zum Beispiel kann man damit einen der Wochentage ausdrücken:

```
oneOf(Montag, Dienstag, Mittwoch, Donnerstag,
      Freitag, Samstag, Sonntag)
```

Einen weiteren Unterschied gibt es bei den Zahlenrestriktionen. OWL-DL unterstützt Zahlenrestriktionen mit dem Konstruktor N; WSML-DL hingegen konstruiert qualifizierende Zahlenrestriktionen mit Q. In OWL kann man zum Beispiel nicht sagen, dass man genau zwei Elternteile hat und dass diese jeweils eine Frau und ein Mann sind. In WSML ist dies sehr wohl möglich und man kann den Wertebereich festlegen. Dieser Unterschied ist aber für den Import von OWL-Ontologien insofern kein Problem, wenn man keine unterschiedlichen Wertebereiche angibt.

WSML-DL mit Wertebereich	**exists** ?x1,?x2(?x[_"hasParent" **hasValue** ?x1] **and** ?x[_"hasParent" **hasValue** ?x2] **and** ?x1 **memberOf** _"Mother" **and** ?x2 **memberOf** _"Father" **and** neg(?x1 = ?x2)) **and** **forall** ?x1,?x2(?x[_"hasParent" **hasValue** ?x1] **and** ?x[_"hasParent" **hasValue** ?x2] **implies** ?x1 = ?x2) and ?x1 **memberOf** _"Mother" and ?x2 **memberOf** _"Father").
OWL-DL ohne Wertebereich	**Class**(Person **complete** **restriction**(hasParent **Cardinality**(2)))

Tabelle 3.3: Wertebereiche

Abgesehen von den unterschiedlich unterstützten Konstruktoren, gibt es in OWL-DL viel mehr Datentypen als in WSML-DL. Für die Implementierung wurden ähnliche WSML-Datentypen gewählt. Folgende Tabelle führt das Mapping der Datentypen an:

OWL Datentypen	**WSML Datentypen**
string	WSML_STRING
normalizedString	WSML_STRING
token	WSML_STRING

language	WSML_STRING
NMTOKEN	WSML_STRING
Name	WSML_STRING
NCName	WSML_STRING
boolean	WSML_BOOLEAN
decimal	WSML_DECIMAL
float	WSML_FLOAT
double	WSML_DOUBLE
integer	WSML_INTEGER
positiveInteger	WSML_INTEGER
nonPositiveInteger	WSML_INTEGER
negativeInteger	WSML_INTEGER
nonNegativeInteger	WSML_INTEGER
int	WSML_INTEGER
long	WSML_INTEGER
short	WSML_INTEGER
byte	WSML_INTEGER
unsignedLong	WSML_INTEGER
unsignedInt	WSML_INTEGER
unsignedShort	WSML_INTEGER
unsignedByte	WSML_INTEGER
dateTime	WSML_DATETIME
time	WSML_TIME
date	WSML_DATE
gYearMonth	WSML_GYEARMONTH
gYear	WSML_GYEAR
gMonthDay	WSML_GMONTHDAY
gDay	WSML_GDAY
gMonth	WSML_GMONTH
hexBinary	WSML_HEXBINARY
base64Binary	WSML_HEXBINARY
anyURI	WSML_IRI

Tabelle 3.4: Datentypen

3.3 Roundtripping

Diese Arbeit baut auf eine bereits implementierte Transformation, von WSML-DL nach OWL-DL, auf und versucht auf das Roundtripping Rücksicht zu nehmen. Dies ist die Fähigkeit, Daten zu konvertieren und wieder zurück in das Original ohne Verluste oder andere Unterschiede zu transformieren. Zum Beispiel drückt man in OWL-DL Eigenschaften mit Properties aus. In WSML kann man hingegen Eigenschaften als Attribute bei Konzepten oder als Relationen definieren. Da beide Arten zu einem validen Ergebnis führen, werden Eigenschaften in der Implementierung standardmäßig in Attribute transformiert. Wenn bei einer Property der Kommentar http://owl2wsmltransformation/annotation/Relation vorhanden ist, wird eine Relation gebildet. Wenn dieser Kommentar nicht existiert, wird die Property als Attribut implementiert (vorausgesetzt die Domäne ist bekannt). Im Gegensatz dazu werden solche Relationen mit einer NonFunctionalProperty versehen, damit wieder die Transformation von WSML nach

OWL darauf zurückgreifen kann. Folgende Tabelle beschäftigt sich mit dem Roundtripping, worauf die Implementierung Rücksicht genommen hat.

OWL-DL	WSML-DL
ObjectProperty (propid domain (classid1) range (classid2))	concept classid1 propid impliesType classid2
ObjectProperty (propid **comment("http://owl2wsmltransformation/Relation")** domain (classid1) range (classid2))	relation propid (impliesType classid1, impliesType classid2) **nfp** **annotID hasValue** **"http://owl2wsmltransformation/Relation"** **endnfp**
DatatypeProperty (dataID domain (classid) range (xsd:Datatype))	concept classid dataID ofType Datatype
DatatypeProperty (dataID **comment("http://owl2wsmltransformation/Relation")** domain (classid) range (xsd:Datatype))	relation dataID (impliesType classid, ofType Datatype) **nfp** **annotID hasValue** **"http://owl2wsmltransformation/Relation"** **endnfp**
ObjectProperty (propid1 super (propid2) domain (classid1) range (classid2))	concept classid1 propid1 impliesType classid2 axiom axClass1 definedBy ?x[propid1 hasValue ?y] impliedBy ?x [propid2 hasValue ?y]
ObjectProperty (propid1 **comment("http://owl2wsmltransformation/Relation")** super (propid2) domain (classid1) range (classid2))	relation propid1 (subrelationOf propid2, impliesType classid1, impliesType classid2) **nfp** **annotID hasValue** **"http://owl2wsmltransformation/Relation"** **endnfp**
ObjectProperty(propid) Individual1 Individual2	instance indid1 propid hasValue indid2 instance indid2
ObjectProperty(propid) **comment("http://owl2wsmltransformation/RelationInstance")** Individual1 Individual2	relationInstance propid (Individual1, Individual2)
EnumeratedClass (classid indid1 indid2 ... indidn))	nfp http://owl2wsmltransformation/Enumeration1 hasValue "indid1" http://owl2wsmltransformation/Enumeration2 hasValue "indid2" ... http://owl2wsmltransformation/EnumerationN hasValue "indidn" endnfp

Tabelle 3.5: Roundtripping Regeln

Kapitel 4

Implementierung

Im Zuge dieser Bakkalaureatsarbeit wurde eine Importfunktion von OWL-DL Dokumenten für das Paket WSMO4J entwickelt. Deswegen wurde ein Mapping zwischen OWL-DL Konstrukten und WSML-DL Elementen erstellt, damit anhand im Mapping definierten Regeln OWL-DL Ontologien auf WSML-DL Ontologien abgebildet werden können.
Damit man auf die einzelnen Elemente einer OWL-Ontologie zugreifen kann, wurde für die Implementierung die Schnittstelle OWL API verwendet. WSML-Konstrukte wurden für die Transformation mit dem Paket WSMO4J erstellt. Diese Schnittstellen werden in den nächsten zwei Abschnitten vorgestellt.

4.1 OWL API Version 1.1

[1]

Mitglieder des WonderWeb Konsortiums haben eine API [WonderWeb, 2004] für OWL entwickelt, die Entwicklern einen Zugang zu den Datenstrukturen von OWL-Ontologien bereitstellt. Die API enthält Pakete, die verschiedene Bereiche für das Arbeiten mit OWL Ontologien abdecken.

model Das model-Paket bietet einen lesenden Zugriff auf die OWL Ontologien an. Es werden Methoden angeboten, die die definierten Klassen, ihre Eigenschaften, Axiome usw. zurückgeben. Wie bei einem objekt-orientierten Design üblich, werden Interfaces angeboten, die dem Benutzer ermöglichen alternative Implementierungen zu verwenden.

change Ein kritischer Punkt bei der Implementierung ist das Problem die reelle Welt in Ontologien abzubilden, da die Domänen sich ständig ändern, neue Klassen entstehen und die Bedeutung von Klassen wechseln. Das change-Paket erweitert das model-Paket um die Manipulation der Strukturen. Zum Beispiel kann man Elemente hinzufügen oder löschen, Definitionen ändern usw.

[1] http://owlapi.sourceforge.net/

Parser Der Parser bildet die Ontologie nach einem Import von RDF/XML Syntax oder Abstract Syntax in eine interne Repräsentation ab.

Serializer Hier wird eine concrete Syntax von der internen Datenrepräsentation produziert.

inference Dieses Paket umfasst eine detaillierte Beschreibung der formalen Semantik der Sprache und ermöglicht den Zugriff auf die OWL DL Schlussfolgerungssysteme.

Validierung WonderWeb hat zur API auch einen Validator entwickelt, der im Web unter http://www.mygrid.org.uk/OWL/Validator frei zugänglich ist. Dieser überprüft OWL-RDF Dokumente auf ihre syntaktische Richtigkeit und gibt an, um welche OWL-Variante es sich handelt. Außerdem werden RDF/XML Dokumente in die Abstract Syntax transformiert.

4.2 WSMO4J Version 0.6.1

[2]

WSMO4J ist eine API, die auf das Modell Web Service Modeling Ontology (WSMO) basiert, und eine Referenz-Implementierung um Semantic Web Service-Anwendungen zu erstellen.

Modell Mit WSMO4J kann man Ontologien, Web Services, Ziele und Mediatoren in konzeptueller Syntax erstellen und logische Ausdrücke entwickeln. In diesem Paket hat man sowohl Zugriff auf die WSMO-Elemente als auch die Möglichkeit sie zu ändern, erstellen oder hinzuzufügen.

Validator Der WSML Validator überprüft WSML Dokumente auf syntaktische Fehler und gibt die WSML Variante an. Dieses Werkzeug ist auch online zu verwenden unter
http://tools.deri.org/wsml/validator/v1.2/

Parser Serializer In diesem Paket werden die Funktionen für den Import und den Export von WSML Dokumenten verwaltet. Dabei ist es möglich WSML Sprachvarianten in andere Varianten zu exportieren und das Format in WSML oder WSML-XML zu ändern.

4.3 Architektur

Das Programm, das die Transformation erledigt, wurde in Java implementiert und verwendet die Schnittstellen OWL API und WSMO4J. Diese Pakete wurden

[2] http://wsmo4j.sourceforge.net/

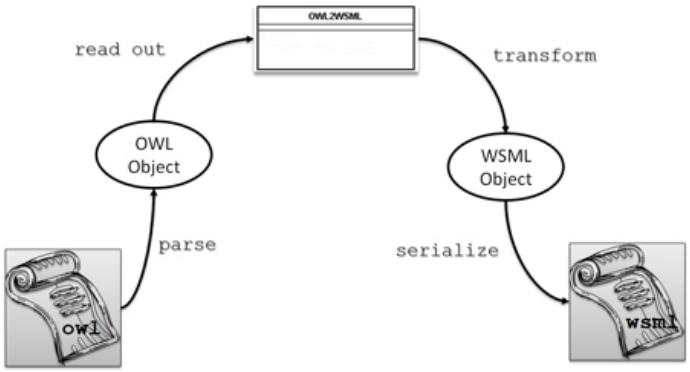

Abbildung 4.1: Architektur

in den vorigen Kapiteln vorgestellt. Der Import von OWL Ontologien benötigt einen Zeichenstrom von einer OWL Ontologie in OWLRDF-Syntax, damit der von OWLParser.class implementierte Parser OWL-Objekte daraus bilden kann. Im Paket WSMO4J werden noch andere Parser für den Import und Export von Dokumente zur Verfügung gestellt, die alle das gleiche Interface implementieren. Durch den Import von OWL-Ontologien, der im Rahmen dieser Bakkalaureatsarbeit implementiert wurde, wurde die Schnittstelle verbessert, da eine vorhandene Importfunktion verschachtelte Elemente nicht unterstützt hat und auf das Roundtripping mit Bezug auf WSML-DL nach OWL-DL keine Rücksicht genommen hat. Durch diese Möglichkeit kann man nun ein größeres Publikum für die Ontologiesprache WSML erreichen, das nun die zahlreich vorhandenen OWL-Ontologien importieren und die Vorteile von WSML nutzen kann.

Abbildung 4.1 stellt die Vorgangsweise des Imports dar. Mit der OWL-API werden die einzelnen Elemente sukzessive ausgelesen und die Klasse OWL2WSML, die im Rahmen dieser Arbeit implementiert wurde, vollzieht die Transformation in WSML-Objekte. Die erhaltene WSML-Ontologie wird in ein TopEntity-Array, das Ontologien, Semantic Web Services, Ziele und Mediatoren verwaltet, eingefügt.

4.4 Umsetzung

Für die Implementierung des Imports von OWL-DL nach WSML-DL für WSMO4j wurde die Klasse OWLParser erstellt, die das Interface Parser implementiert und sich so einheitlich in die Reihe von anderen Parsern des Pakets WSMO4J einfügt. Die implementierten Methoden rufen die neu erstellte Klasse OWL2WSML mit der Methode generateOntology() auf und übergeben ein Dokument einer OWL-Ontologie in OWL/RDF Syntax der Klasse. Dieses Dokument wird mit der Klasse OWLRDFParser in eine interne Struktur geparst. Von der Methode generateOntology() wird eine in WSML-DL transformierte Ontologie zurückgege-

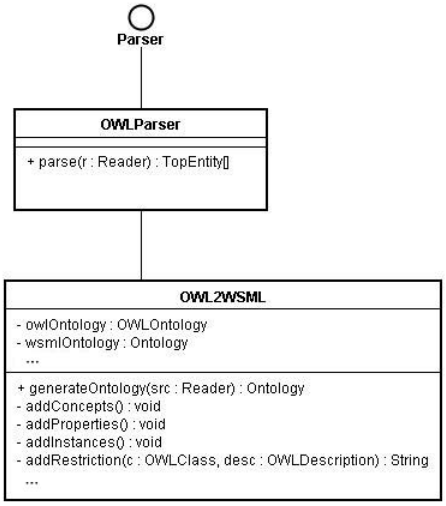

Abbildung 4.2: vereinfachtes Klassendiagramm

ben, die in ein TopEntity-Feld eingefügt wird. Ein TopEntity-Feld verwaltet das Meta-Modell Web Service Modeling Ontology (WSMO) mit Ontologien, Semantic Web Services, Ziele und Mediatoren. Abbildung 4.2 stellt ein vereinfachtes Klassendiagramm der implementierten Umsetzung dar.

In der Klasse OWL2WSML werden ausgehend von der Methode generateOntology() weitere private Methoden, die mit der OWL API die Konstrukte herauslesen, aufgerufen. addConcepts() liest alle OWL-Klassen aus, transformiert diese und erstellt mit der Schnittstelle WSMO4J WSML-Konzepte. Da OWL verschachtelte Strukturen mit Vereinigungen, Durchschnitten usw. mit anonymen Klassen erlaubt, wurde eine rekursive Methode addRestrictions() implementiert. Die Konstrukte werden, mit Ausnahme der einfachen Klassen, mit logischen Ausdrücken in Zeichenketten beschrieben. Diese Zeichenketten werden der Methode defineAxiom() übergeben, die Axiome für WSML-Elemente definiert. Am Ende der Methode addConcepts() wird die Methode addDisjointClasses() aufgerufen, die die explizit unterschiedlichen Klassen abarbeitet.

Nachdem die Konzepte abgearbeitet wurden, werden die Eigenschaften, die mit der OWL-API ausgelesen werden, mit der Methode addProperties() nach der Reihe transformiert und mit WSMO4J WSML-Elemente konstruiert. Hier wird besonderer Wert auf das Roundtripping gelegt, da Eigenschaften je nach Annotation als Attribut eines Konzepts oder als Relation übersetzt werden. Hier wird von einer zuvor implementierten Transformation von OWL-DL nach WSML-DL vorausgesetzt, dass entsprechende Kommentar-Annotationen gesetzt wurden. Wenn eine Eigenschaft eine Kommentar-Annotation
“http://owl2wsmltransformation/annotation/Relation“ enthält, wird eine Re-

lation, ansonsten ein Attribut des Konzepts der Domäne gebildet. Wenn keine Domäne vorhanden ist, muss eine Relation konstruiert werden. Weitere solche Besonderheiten wurden im Kapitel 3.2 erläutert. OWLDataProperties sind in OWL für die Eigenschaften von Datentypen zuständig. Da in OWL eine größere Anzahl an Datentypen als in WSML verfügbar ist, werden in WSML einige mit der aufrufenden Methode getDataType() in ähnliche Datentypen übersetzt. In OWL gibt es zum Beispiel Long und Double, die beide in WSML nach Double übersetzt werden, weil es in WSML den Datentyp Long nicht gibt. Weitere Ausnahmen wurden im Kapitel Einschränkungen erläutert. Zum Schluss der Methode addProperties() werden die Axiome auf Äquivalenz und SupProperties überprüft und dementsprechend nach WSML übersetzt.

Nachdem die Eigenschaften abgearbeitet wurden, werden die Instanzen in der Methode addInstances() mit den Schnittstellen OWL API und WSMO4J transformiert. Hier werden die Kommentar-Annotationen getestet, ob diese „http://owl2wsmltransformation/annotation/RelationInstance“ enthält. Wenn dies der Fall ist, wird eine Relationsinstanz erstellt. Bei den Datentypen gibt es wie bei den Eigenschaften wieder dieselben Ausnahmen, und es wird mit der Methode getDataValue() abgearbeitet. Nachdem die Individuen erstellt wurden, wird mit der Methode addSameAndDifferent() auf Gleichheit und Ungleichheit überprüft und mit logischen Axiomen übersetzt. Bei allen drei Konstrukten, nämlich Konzepte, Eigenschaften und Individuen, werden die NonFunctionalProperties mit der Methode addNFP() von den OWL-Annotationen gebildet.

4.5 Tests und Evaluierung

Die implementierte Importfunktion von OWL-Ontologien wurde mit JUnit mit der Klasse OWL2WSMLTests getestet. Dafür wurden verschiedene OWL-Dokumente getestet, die jeweils unterschiedliche Konstrukte abdecken. Die Evaluierung, indem man die transformierten WSML-Ontologien wieder zurück transformiert, konnte nicht durchgeführt werden, da erst die jetzt erstellte Implementierung auf das Roundtripping Rücksicht genommen und dafür Regeln aufgestellt hat. Der Import von WSML-Dokumenten nach OWL-Dokumenten hat diese Roundtripping-Regeln noch nicht implementiert. Dies ist aber für die Zukunft angedacht.

Kapitel 5

Zusammenfassung

Diese Bakkalaureatsarbeit handelt vom Import von OWL-Ontologien in der Untersprache OWL-DL nach WSML-DL innerhalb von WSMO4J. Nach einer grundlegenden Einführung in die Thematik der Ontologien und der logischen Beschreibungssprache Description Logic wurde die Ontologiesprache OWL für das Semantic Web vorgestellt. Der nächste Abschnitt hat sich mit der Vorstellung der Ontologiesprache WSML beschäftigt. Nach einer Einführung in diese Sprache wurde eine Mapping-Tabelle mit den Regeln für den Import von OWL-Ontologien aufgestellt. Die Ontologiesprachen eignen sich gut für eine Transformation, da die Sprachen OWL-DL und WSML-DL auf die Description Logic basieren. In der Implementierung resultierten die Schwierigkeiten auf die unterschiedlichen Beschreibungsformen. Bei der Transformation der Datentypen mussten Abstriche gemacht werden, da gewisse Datentypen zusammengefasst wurden. Ob diese große Anzahl an OWL-Datentypen sinnvoll ist, sei dahingestellt.

Das Ziel der Arbeit war der Import von OWL-Dokumenten, das eine Zusatzfunktion für das Paket WSMO4J darstellen soll, damit bereits mit dem W3C Standard OWL erstellte Ontologien auch für WSML nutzbar sind. Dadurch möchte man ein breiteres Publikum für WSML ansprechen, um es populärer zu machen.

Semantic Web Beschreibungsstandards existieren schon seit einigen Jahren, doch haben sich semantische Modelle in der breiten Masse noch nicht durchsetzen können. In der Theorie lassen sich solche Systeme recht einfach implementieren, in der Praxis hingegen stößt man aber auf zahlreiche Probleme durch die Dynamik und Veränderlichkeit der gesamten Sprache. Zurzeit führen die Techniken des Web 2.0, des sogenannten „Mitmach-Web“ zu großem Erfolg. Anstatt eine vollständige Ontologie zu erstellen, werden hier die User beim Web 2.0 aufgefordert, Dinge lediglich mit Schlagwörtern zu versehen (taggen). Durch die sprichwörtliche „Weisheit der Vielen“ wird das Problem der passenden Bezeichnung ziemlich pragmatisch gelöst. Solcherart entstandenen Tag-Wolken

sind doch sehr weit von vollständigen Ontologien entfernt. Wahrscheinlich werden sich diese zwei Systeme des Taggens und der semantischen Beschreibung einander immer mehr annähern, und das Semantic Web wird irgendwann zur Wirklichkeit.

Literaturverzeichnis

[Baader et al., 2002] Baader, F., Horrocks, I., and Sattler, U. (2002). Description logics for the semantic web.

[Baader and Nutt, 2003] Baader, F. and Nutt, W. (2003). Basic description logics.

[Bechhofer et al., 2004a] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004a). Owl web ontology language reference, 8.3 owl lite. Available from: http://www.w3.org/TR/owl-ref/#OWLLite.

[Bechhofer et al., 2004b] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004b). Owl web ontology language reference, 8.3 owl lite. Available from: http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OntologyVersioning.

[Bechhofer et al., 2004c] Bechhofer, S., van Harmelen, F., Hendler, J., Ian Horrocks, D. L. M., Patel-Schneider, P. F., and Stein, L. A. (2004c). Owl web ontology language reference. Available from: http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[Connolly et al., 2001] Connolly, D., van Harmelen, F., Horrocks, I., L., D., McGuinness, Patel-Schneider, P. F., and Stein, L. A. (2001). Daml+oil (march 2001) reference description. Available from: http://www.w3.org/TR/daml+oil-reference.

[de Bruijn et al., 2005] de Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L., Kifer, M., and Fensel, D. (2005). D16.1v0.21 the web service modeling language wsml. Available from: http://www.wsmo.org/TR/d16/d16.1/v0.21/.

[de Bruijn Holger Lausen et al., 2005] de Bruijn Holger Lausen, J., Krummenacher, R., Axel Polleres, L. P., Kifer, M., and Fensel, D. (2005). D16.1v0.21 the web service modeling language wsml, 2.1.2 identifiers in wsml. Available from: http://www.wsmo.org/TR/d16/d16.1/v0.21/#sec:wsml-identifiers.

[Decker et al., 2000a] Decker, S., Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D., Patel-Schneider, P. F., Staab, S., and Stein, L. (2000a). Description of oil. Available from: http://www.ontoknowledge.org/oil/.

[Decker et al., 2000b] Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M. C. A., Broekstra, J., Erdmann, M., and Horrocks, I. (2000b). The semantic web: The roles of XML and RDF. *IEEE Internet Computing*, 4(5):63–74.

[Duerst and Suignard, 2005] Duerst, M. and Suignard, M. (2005). Internationalized resource identifiers (iris). Available from: ftp://ftp.rfc-editor.org/in-notes/rfc3987.txt.

[Gruber, 1993] Gruber, T. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(1):199–220.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P., and van Harmelen, F. (2003). From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26.

[Koivunen and Miller, 2001] Koivunen, M.-R. and Miller, E. (2001). W3c semantic web activity. Available from: http://www.w3.org/2001/12/semweb-fin/w3csw.

[Lee et al., 2001] Lee, T. B., Connolly, D., Hawke, S., Herman, I., Prud'hommeaux, E., and Swick, R. (2001). W3c semantic web activity. Available from: http://www.w3.org/2001/sw/.

[Pagels, 2000] Pagels, M. (2000). The darpa agent markup language homepage. Available from: http://www.daml.org/.

[S. Weibel and Wolf, 1998] S. Weibel, J. Kunze, C. L. and Wolf, M. (1998). Rfc 2413 - dublin core metadata for resource discovery. Available from: http://rfc.net/rfc2413.html.

[Smith et al., 2004] Smith, M. K., Welty, C., and McGuinness, D. L. (2004). Owl web ontology language guide. Available from: http://www.w3.org/TR/owl-guide/.

[Spalthoff and Hitzler, 2005] Spalthoff, A. and Hitzler, D. P. (2005). Description logics - beschreibungslogiken. Available from: www.aifb.uni-karlsruhe.de/Lehre/Sommer2005/SemTech/stuff/DL.doc.

[Sun, 1998] Sun, S. X. (1998). Internationalization of the handle system — A persistent global name service.

[WonderWeb, 2004] WonderWeb (2004). Owl api documentation. Available from: http://www.chengtao.name/semanticweb/owl-api-javadoc/index.html.