universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

„Entwicklung eines kollaborativen Ontologieeditors auf Basis des semantischen Wikis Ylvi“

Verfasser

Wolfgang Philipp

Angestrebter akademischer Grad

Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, November 2008

Studienkennzahl lt. Studienblatt:	175
Studienrichtung lt. Studienblatt:	Wirtschaftsinformatik
Prüfer:	Univ. Prof. Dr. techn. Wolfgang Klas
Betreuer:	Dipl.-Ing. Niko Popitsch Mag. Bernhard Schandl

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die größte und zugleich auch wichtigste Herausforderung im Bereich des Semantischen Webs besteht in der formalen Beschreibung der Bedeutung von Information, sodass die Semantik dieser Information nicht nur für den Menschen, sondern auch für Maschinen interpretierbar wird. Menschen sind in der Lage, aus Symbolen oder Sequenzen von Symbolen, dem Kontext, in welchem diese Symbole gebraucht werden, und ihrem bereits vorhandenem Wissen die Bedeutung einer bestimmten Information abzuleiten. Damit auch für Maschinen die semantische Information lesbar und weiterverarbeitbar wird, muss, wie auch im realen Leben, ein Vokabular geschaffen werden, mit welchem Wissen repräsentiert werden kann. Diese Vokabulare werden mit Hilfe von Ontologien definiert. Ontologien sind explizite und formale Konzeptualisierungen eines bestimmten Wissensbereichs und beschreiben die elementaren Konzepte eines Wissensgebiets und deren Beziehungen zueinander [11, 12]. Sie sind die wichtigsten und grundlegendsten Elemente im Bereich des Semantischen Webs und sind Voraussetzung für die automatisierte Interpretation und Weiterverarbeitung von Daten [25].

Die Idee des Semantischen Webs geht bereits auf das Jahr 1999 zurück, als Tim Berners-Lee[1] seine Vision des zukünftigen World Wide Webs (WWW) wie folgt formulierte:

> *"I have a dream for the Web [in which machines] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize."* [3]

[1] Tim Berners-Lee gilt auch als Erfinder des traditionellen World Wide Webs.

Tatsächlich hat sich in diesem letzten Jahrzehnt einiges in diese Richtung entwickelt. Es wurde intensiv geforscht, standardisierte Sprachen wurden geschaffen und mittlerweile existiert auch eine Vielzahl von Applikationen und Anwendungsszenarien, welche sich der Technologie des Semantischen Webs bedienen. Zum aktuellen Zeitpunkt ist man allerdings noch sehr weit von oben genannter Zukunftsvision aus dem letzten Jahrtausend entfernt. Der Hauptgrund, warum sich das Semantische Web nur schleppend entwickelt und etabliert hat, ist der Mangel an bereits verfügbaren und geeigneten Ontologien. Dafür verantwortlich ist hauptsächlich die bisher übliche und traditionelle Vorgehensweise zur Entwicklung von Ontologien. [25, 29]

Da auch die Realität und die Struktur eines Wissensbereichs kompliziert sind, benötigt man für die Definition von Ontologien Fachexperten[2] des jeweiligen Wissensbereichs (domain experts), welche in der Lage sind, den jeweiligen Wissensbereich informal zu beschreiben. Doch auch die vorhandenen Technologien und Sprachen zur formalen Beschreibung von Ontologien sind komplex und nicht gerade einfach und schnell zu erlernen. Deshalb sind zusätzlich Knowledge Engineers vonnöten, welche dieses Wissen mit den verfügbaren Technologien aus dem Bereich des Semantischen Webs maschinenverständlich beschreiben. Die Erstellung einer Ontologie ist also eine komplexe Aufgabe, was bedeutet, dass auch die verwendeten Werkzeuge (Tools) zur Erstellung von Ontologien komplex sind und deshalb ist es für die Fachexperten oder potentiellen Benutzer von Ontologien meist schwierig bis unmöglich, diese selbst zu erstellen. Nachdem Ontologien erstellt worden sind, werden diese veröffentlicht und sind somit für andere Applikationen nutzbar. Dieser Ansatz zur Entwicklung von Ontologien bringt allerdings eine Menge Nachteile mit sich: [11, 25, 29]

- Die Entwicklung einer Ontologie ist aufgrund der nötigen Zusammenarbeit von Fachexperten und Knowledge Engineers von Natur aus ein kollaborativer Prozess. Der traditionelle Ansatz ist zwar auch kollaborativ, allerdings ist dabei nur eine geringe Menge von Personen in den Entwicklungsprozess involviert. Außerdem unterstützen übliche Werkzeuge diesen kollaborativen Charakter bei der Entwicklung von Ontologien in einem zu geringen Ausmaß, da diese Werkzeuge zumeist über keine verteilte Infrastruktur verfügen.

[2] Es wird ausdrücklich darauf hingewiesen, dass bei allen personenbezogenen Begriffen, die in dieser Arbeit verwendet werden, beide Geschlechter miteinbezogen sind. Aufgrund der besseren Lesbarkeit wird auf die Angabe beider geschlechterspezifischen Begriffe jedoch verzichtet.

- Bei der Modellierung von Wissen ist ein Knowledge Engineer auf eine detaillierte und ausführliche Beschreibung des Fachexperten angewiesen. Deshalb ist eine physische Zusammenkunft zwischen den beiden Expertengruppen so gut wie unverzichtbar. Somit werden die meisten Ontologien von einem kleinen Team zu einem bestimmten Zeitpunkt erstellt, was bedeutet, dass der traditionelle Ansatz zur Erstellung von Ontologien eher ein langsamer, unflexibler und nur wenig dynamischer Prozess ist. Viele Wissensbereiche sind aber durchaus dynamisch und häufig auftretenden Veränderungen unterworfen, was zur Folge hat, dass bereits existierende Ontologien oft nicht mehr am aktuellsten Stand der Dinge sind, weil wieder neue Erkenntnisse im jeweiligen Wissensbereich gewonnen werden konnten und die Ontologien in der Zwischenzeit nicht aktualisiert worden sind.

- Der traditionelle Ansatz ist ein top-down Prozess, im Zuge dessen eine Ontologie erst entwickelt und veröffentlicht wird und erst danach verwendet wird. Allerdings sind die tatsächlichen Daten auf Instanzebene, welche einen bestimmten Wissensbereich konkret beschreiben, im Vorhinein oft unbekannt oder nur sehr schwer voraus zu sagen. Außerdem ist es unmöglich, eine universal gültige Ontologie zu entwickeln, welche alle möglichen auftretenden Fälle in einem bestimmten Wissensbereich optimal beschreiben kann. Aus diesen Gründen steht man als Benutzer oft vor dem Problem, dass die existierenden Ontologien nicht den Anforderungen an die eigene Applikation genügen.

- Die Ontologien werden von einer kleinen Gruppe von Personen für eine große Anzahl von Benutzern geschaffen, wobei Menschen oft sehr stark differenzierende Auffassungen eines bestimmten Wissensbereichs haben können und daher auch viele verschiedene Anforderungen an eine Ontologie stellen. Wissen ist auch immer subjektiv und deshalb können die von einer kleinen Gruppe entwickelten Ontologien nicht all die subjektiven Auffassungen vieler verschiedener Benutzer abdecken.

- Die Semantik einer Ontologie ist mit einer formalen Sprache beschrieben. Damit ein Benutzer aber in der Lage ist, zu entscheiden, welche Ontologie am besten für seine Applikation geeignet ist, muss dieser die Bedeutung der Ontologie ebenso verstehen können. Da die in einer Ontologie definierten Begriffe zur Beschreibung von Ressourcen meist sehr wenig Aufschluss über die eigentliche

Semantik dieser Begriffe geben, sind auch informale und für Menschen verständliche Beschreibungen notwendig. Weiters bedarf es oft auch Einblick in den Diskurs, der zu einer bestimmten Ontologie geführt hat, um diese verstehen zu können, da die Entwicklung einer Ontologie ein evolutionärer Verfeinerungsprozess ist. Dies wird weder von den üblichen Werkzeugen noch von den standardisierten Beschreibungssprachen ausreichend unterstützt. Somit kann es zu Unterschieden zwischen der intendierten Semantik der Entwickler und der aufgefassten Bedeutung der Nutzer dieser Ontologien kommen.

- In vielen Fällen müsste eine Ontologie erweitert oder leicht angepasst werden, damit diese für einen bestimmten Benutzer optimal nutzbar wird. Dieser hat allerdings keinen Einfluss auf die Entwicklung und Evolution der Ontologien und kann seine Vorschläge, Wünsche und Anregungen auch nur sehr schwer einbringen.

Nun gilt es, neue Wege zu finden und vorhandene Infrastrukturen zu nutzen, um zumindest einige dieser genannten Probleme bei der Ontologieentwicklung zu eliminieren. Dabei sollte vor allem der inhärente kollaborative Charakter der Ontologieentwicklung besser unterstützt werden, da dies das Hauptproblem und die Ursache für nahezu alle anderen genannten Probleme ist.

1.2 Ansatz

Es wäre wünschenswert, die Ontologieentwicklung einer breiten Masse zugänglich zu machen, weil dies die Anzahl vorhandener Ontologien deutlich erhöhen könnte und die in Kapitel 1.1 genannten Probleme eventuell beseitigen würde. Um eine kollaborative Ontologieentwicklung zu ermöglichen, benötigt man eine verteilte Infrastruktur. Da das WWW für viele Menschen über alle Kontinente verteilt zugänglich ist, wäre dies ein geeignetes Netzwerk, welches sehr viele Fachexperten und Knowledge Engineers zusammenführen könnte, und zwar unabhängig von deren aktuellem Aufenthaltsort und der Zeit. Es existieren auch bereits geeignete Plattformen für eine gemeinschaftliche und evolutionäre Entwicklung von Ontologien im WWW, nämlich die immer populärer werdenden Wikis. Dabei handelt es sich um Ansammlungen von Webseiten, welche von den Benutzern nicht nur gelesen, sondern auch von diesen selbst erstellt und bearbeitet werden können [28]. Mit Hilfe dieser Plattformen wäre es also für jeden möglich, an der Ontologieentwicklung selbst mitzuwirken und seine eigenen Vorschläge bzw.

Anforderungen an eine Ontologie mit einzubringen. Das Potential solcher Plattformen zeigt das prominenteste Beispiel Wikipedia, das mittlerweile zur größten Enzyklopädie der Welt gewachsen ist. Durchgeführte wissenschaftliche Berechnungen zeigen, dass zirka 93 % der Artikel in Wikipedia einem ontologisch konsistenten Modell entsprechen, was bedeutet, dass eine breite Masse an Menschen in der Lage ist, gemeinschaftlich Wissensmodelle zu entwickeln und diese auch kontinuierlich zu verfeinern [12]. Ein Wiki ist also eine Wissensbasis, in welchem dieses Wissen in Form von Artikeln informal beschrieben ist.

Man stelle sich nun vor, es gäbe die Möglichkeit, die Artikel im Wiki so zu erweitern, dass man das in den Artikeln informal ausgedrückte Wissen formal in einer Ontologiebeschreibungssprache beschreiben könnte. Auf diese Weise könnten Knowledge Engineers die existierenden Artikel mit den Primitiven einer Ontologiebeschreibungssprache erweitern, sodass die Artikel auch in einer maschinenverständlichen Sprache vorliegen würden. Mit Hilfe von bereits verfügbaren Applikationen, sogenannten Inferenz-Tools, ist es möglich, aus konkreten Beschreibungen und der zugrunde liegenden Ontologie Schlussfolgerungen zu ziehen. Dies bedeutet, in weiterer Folge wäre es also auch mit solchen Inferenz-Tools möglich, aus dem in den Artikeln formulierten Wissen neues Wissen zu generieren.

Mit diesem kollaborativen Ansatz zur Ontologieentwicklung sollte es weniger häufig zu Differenzen zwischen der Auffassung der Entwickler und jener der Benutzer kommen, weil die semantische Beschreibung einer Ontologie auch in einer natürlichen Sprache gegeben ist. In Wikis gibt es üblicherweise für jeden Artikel auch Seiten, auf welchen, ähnlich wie in einem Forum, über den Inhalt des Artikels diskutiert wird. Darüber hinaus sind auch alle früheren Versionen eines Artikels verfügbar. Dies bedeutet, dass somit auch der soziale Diskurs, der zu einer bestimmten Ontologiebeschreibung geführt hat, zugänglich wäre, was zusätzlich hilft, die tatsächliche Bedeutung einer Ontologie zu verstehen. [11, 25, 29]

Auf diese Weise wird die Ontologieentwicklung viel flexibler und dynamischer, weil jeder die Möglichkeit hat, jederzeit an der Entwicklung mitzuwirken. Das erhoffte Resultat wären somit Ontologien am neuesten Stand, welche schnell auf Veränderungen in einem Wissensbereich reagieren können und kontinuierlich weiter verfeinert werden, bis sie den Anforderungen einer bestimmten Interessensgruppe entsprechen. Dadurch würde sich die Vielseitigkeit verfügbarer Ontologien erheblich erhöhen und man wäre

in der Lage, mehrere unterschiedliche Anforderungen von Benutzern zu erfüllen. Dies würde wiederum bedeuten, dass Ontologien häufiger verwendet werden würden und die Aktivität einer Community für Ontologieentwicklung steigen könnte, wenn die Benutzer die Möglichkeit haben, sich selbst an der Erstellung zu beteiligen. [11, 29]

Aufgrund des gemeinschaftlichen und öffentlichen Charakters von Wikis wäre also zu erwarten, dass im Laufe der Zeit immer mehr maschinenverständliche Ontologien in standardisierten Sprachen zur Verfügung stehen würden, sofern die Gemeinschaft des Wikis lebendig und aktiv ist.

Eine höhere verfügbare Anzahl von wiederverwendbaren und qualitativ hochwertigen Ontologien könnte wiederum ein Anreiz für Entwickler darstellen, diese Ontologien in ihren Applikationen zu verwenden. Dies würde neue Möglichkeiten für funktional mächtige, automatisierte und über weite Bereiche miteinander verbundene Applikationen schaffen. Diese neuen Möglichkeiten könnten der nötige Anstoß für die weitere Entwicklung und die endgültige Durchsetzung des Semantischen Webs sein.

Im Rahmen dieser Arbeit soll dieses innewohnende Potenzial eines Wikis zur Ansammlung von Wissen zur kollaborativen Erstellung von Ontologien genutzt werden. Konkret geht es dabei um die Erweiterung eines bereits existierenden semantischen Wikis namens Ylvi[3], sodass es möglich ist, die Artikeleinträge in diesem Wiki mit zusätzlicher Information zu versehen, um daraus formal beschriebene Wissensmodelle zu generieren. Die Sprache der Wahl zur Wissensrepräsentation ist Simple Knowledge Organization System[4] (SKOS, vgl. Kapitel 2.4 und 4.1.2), welche hauptsächlich zur Beschreibung von Thesauri und Klassifikationsschemen verwendet wird. Streng betrachtet ist SKOS selbst auch eine Ontologie, was bedeutet, dass die in SKOS formulierten Wissensmodelle eigentlich schon auf Instanzebene beschrieben werden. Da das SKOS-Modell aber mit Hilfe von Elementen aus mächtigeren Sprachen erweitert werden kann, ist es mit SKOS selbst wiederum möglich, andere Ontologien zu definieren. [13, 20]

Zum besseren Verständnis betrachte man Abbildung 1, welche die Funktionalität des geplanten Systems beschreibt. Hier sieht man mehrere Artikel, welche mit einem in Ylvi zur Verfügung gestellten Metamodell beschrieben sind. Dieses Metamodell beinhaltet alle Sprachkonstrukte, welche auch in SKOS verfügbar sind und ist somit völlig

---

3 http://www.informatik.univie.ac.at/ylvi

4 http://www.w3.org/2004/02/skos/

kompatibel zu SKOS. Artikel können mit diesem Metamodell als SKOS-Klassen typisiert werden, zum Beispiel als ein Objekt der SKOS-Klasse *Concept*. In diesem Beispiel stehen auch zwei Konzepte miteinander in hierarchischer Beziehung. *Fahrzeug* ist in diesem Fall ein allgemeineres Konzept als *Auto*. Umgekehrt könnte man ein *Auto* als ein Subkonzept von *Fahrzeug* betrachten. Die Gesamtheit aller Artikel und ihrer Beziehungen zueinander repräsentiert nun ein Wissensmodell.

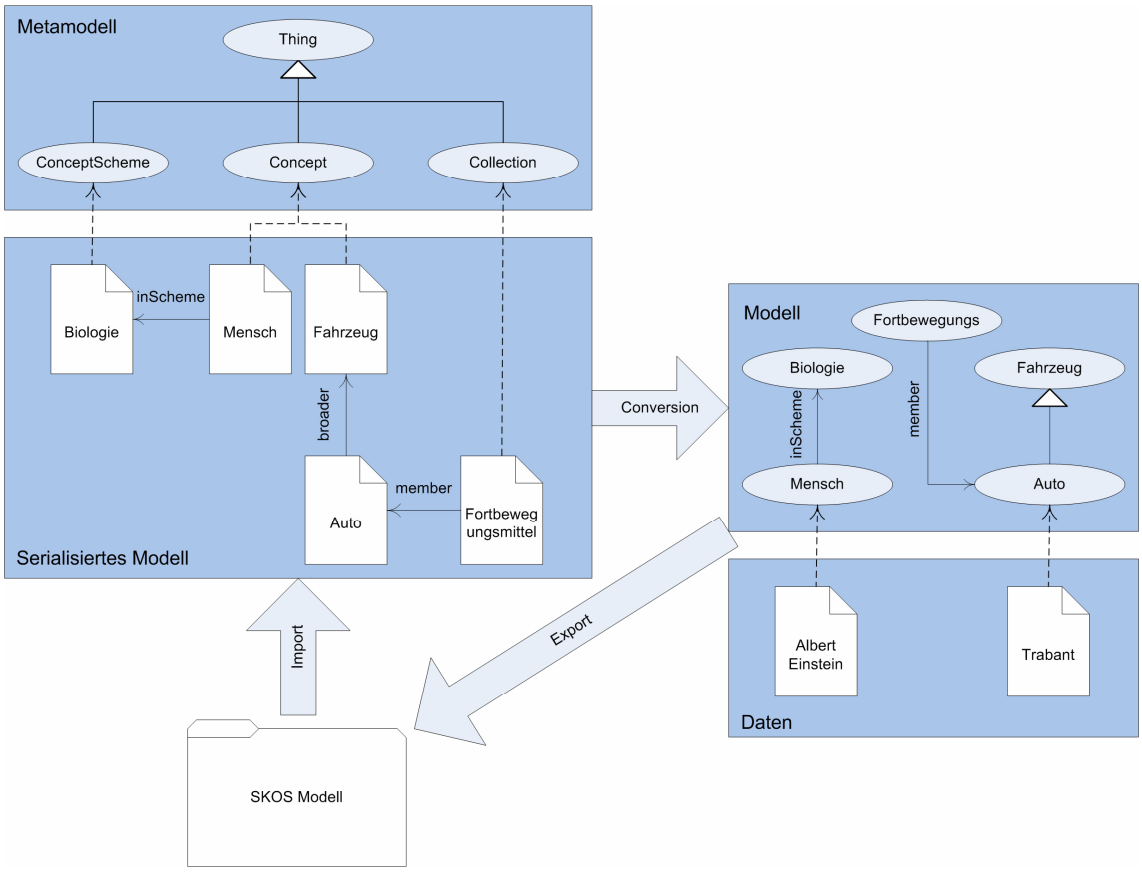

Abb. 1: Übersicht der Funktionalität des geplanten Systems.

Dieses mit den Artikeln ausgedrückte Wissensmodell kann nun in ein spezifisches Modell umgewandelt werden, welches dann wiederum verwendet werden kann, um andere Artikel als konkrete Instanzen eines bestimmten Konzepts zu typisieren. In diesem Beispiel wird nach der Umwandlung ein Artikel namens *Trabant* erstellt, welcher als *Auto* typisiert wird. Diese Typisierung verbessert die Suche in einem Wiki wesentlich, weil nicht nur nach lexikalischen Zeichenketten, sondern auch nach Typen gesucht werden kann. Beispielsweise könnte man alle Artikel auflisten lassen, welche als *Auto* typisiert wurden. Man könnte sich aber auch alle als *Fahrzeug* definierten Artikel anzeigen lassen. In dieser Menge wären nun nicht nur alle *Autos*, sondern auch alle *Fahrzeuge* und

alle Artikel, welche als Instanz eines Subkonzepts von *Fahrzeug* definiert wurden, also zum Beispiel auch alle als *Motorrad* typisierten Artikel.

Das System sollte nun in der Lage sein, dieses interne Wissensmodell in ein SKOS-Modell zu transformieren, sodass das im Wiki ausgedrückte Wissen auch in einer formalen Sprache beschrieben ist.

Umgekehrt soll es auch möglich sein, SKOS-Modelle von außen in Artikeleinträge des Wikis überzuleiten, sodass formal definiertes Wissen von außen auch ins semantische Wiki importiert werden kann.

1.3 Beitrag

Diese Arbeit beschäftigt sich mit Ontologien und den Problemen bei deren Entwicklung und stellt einen alternativen und kollaborativen Ansatz zur Ontologieentwicklung auf Basis eines semantischen Wikis vor.

Um den praktischen Teil dieser Arbeit verstehen zu können, werden zuerst die im Rahmen dieser Arbeit verwendeten Technologien aus dem Bereich des Semantischen Webs vorgestellt, wie zum Beispiel die standardisierten Sprachen Resource Description Framework (RDF), RDF Schema, Web Ontology Language (OWL) und Simple Knowledge Organization System (SKOS).

Weiters werden semantische Wikis und deren Unterschiede zu traditionellen Wikis beschrieben und die beiden Ansätze der traditionellen und der kollaborativen Ontologieentwicklung miteinander verglichen.

Im Rahmen dieser Arbeit wurde ein semantisches Wiki in seiner Funktionalität erweitert, sodass es möglich ist, innerhalb dieses semantischen Wikis mit den Sprachkonstrukten von SKOS Ontologien zu entwickeln (vgl. Kapitel 1.2).

Projekte, welche dieser Arbeit ähnlich sind, werden unter die Lupe genommen und mit dieser Arbeit verglichen.

Weiters wird die Implementation des Systems näher beschrieben, wobei besonders auf die dabei entstandenen Schwierigkeiten eingegangen wird. Um die Funktionalität des implementierten Systems besser verstehen zu können, wird diese anhand eines konkreten Beispiels verdeutlicht.

Zu guter Letzt folgen ein kurzer Ausblick, Lösungsvorschläge, wie die während der Implementation entstandenen und ungelösten Schwierigkeiten beseitigt werden könnten und weitere Ideen, wie das System in Zukunft hinsichtlich seiner Qualität und Benutzerfreundlichkeit verbessert werden könnte.

1.4 Übersicht

Im ersten Kapitel dieser Arbeit werden überblicksartig die Idee und der Sinn des im Rahmen dieser Arbeit implementierten Systems beschrieben. Kapitel 1.1 behandelt die aktuellen Probleme der traditionellen Ontologieentwicklung. Kapitel 1.2 stellt einen alternativen und kollaborativen Ansatz zur Ontologieentwicklung vor, welcher die davor genannten Probleme eventuell beseitigen könnte. Kapitel 1.3 und Kapitel 1.4 geben einen Überblick der in dieser Arbeit behandelten Themen.

Kapitel 2 beschäftigt sich mit den im Rahmen dieser Arbeit verwendeten Technologien des Semantischen Webs und stellt die für das weitere Verständnis dieser Arbeit relevanten Bereiche dieser Technologien vor. Zuerst wird erklärt, worum es sich bei Ontologien tatsächlich handelt (Kapitel 2.1). Danach folgen Einführungen in diverse standardisierte Sprachen, wie Resource Description Framework (RDF) und RDF Schema (Kapitel 2.2), Web Ontology Language (OWL, Kapitel 2.3) und Simple Knowledge Organization System (SKOS, Kapitel 2.4). Kapitel 2.5 befasst sich mit semantischen Wikis und deren Unterschieden zu traditionellen Wikis. Weiters wird das bereits existierende semantische Wiki Ylvi vorgestellt. Die traditionelle und bisher übliche Vorgehensweise zur Ontologieentwicklung und die Unterschiede zur bzw. Vorteile der kollaborativen Ontologieentwicklung werden in Kapitel 2.6 behandelt.

Kapitel 3 beschreibt einige Projekte, welche dieser Arbeit ähnlich sind, stellt diese Projekte dieser Arbeit gegenüber und erläutert die Unterschiede zu bzw. Gemeinsamkeiten mit dieser Arbeit.

Die Implementierungsdetails des im Rahmen dieser Arbeit entwickelten Systems werden in Kapitel 4 beschrieben. Dabei wird besonders auf die Integration des SKOS-Metamodells in das semantische Wiki Ylvi eingegangen und die damit verbundenen Probleme bzw. Schwierigkeiten und deren eventuellen Lösungen werden näher erläutert (Kapitel 4.1). Weiters werden die Implementation der anderen Systemkomponenten und die während dieser Implementation entstandenen Schwierigkeiten beschrieben (Kapitel

4.2 - 4.4). Ein konkretes Beispiel sollte in Kapitel 4.5 die Funktionalität des implementierten Systems verdeutlichen.

In Kapitel 5 wird noch einmal die gesamte Arbeit kurz zusammengefasst. Darauf folgt ein kurzer Ausblick, in welchem die möglichen Impulse dieser Arbeit auf die Entwicklung und Durchsetzung des Semantischen Webs beschrieben werden (Kapitel 5.1). In Kapitel 5.2 werden alternative Ideen für im System nicht optimal umgesetzte Lösungen vorgeschlagen. Weiters wird darauf eingegangen, wie das implementierte System in weiterer Zukunft bezüglich seiner Qualität und Benutzerfreundlichkeit verbessert werden könnte.

Das Literaturverzeichnis dieser Arbeit befindet sich in Kapitel 6.

2 Verwendete Technologien

In diesem Kapitel werden die Grundlagen der im Rahmen dieser Arbeit verwendeten Technologien etwas näher erläutert. Es werden hier nur jene Bereiche der Technologien behandelt, deren Wissen für das weitere Verständnis dieser Arbeit von Relevanz ist.

2.1 Ontologien

Ontologien sind als geteilte Auffassungen eines bestimmten Interessensbereichs zu verstehen. Sie beschreiben also eine Ansicht auf einen bestimmten Wissensbereich. Diese Ansicht wird als eine Menge von Konzepten, deren Definitionen und Beziehungen zueinander begriffen [11]. Ontologien können sich bezüglich ihrer Ausdruckskraft sehr stark voneinander unterscheiden. Es gibt sogenannte Lightweight- und Heavyweight-Ontologien. Lightweight-Ontologien beschreiben Konzepte, Eigenschaften von Konzepten und Beziehungen zwischen diesen Konzepten. Heavyweight-Ontologien sind Lightweight-Ontologien, welche um Axiome und Einschränkungen erweitert werden. Diese erhöhen die Ausdruckskraft und beschreiben die intendierte Semantik einer Ontologie etwas klarer und bestimmter [8].

Mit Hilfe der in den Kapiteln 2.2.7 und 2.3 beschriebenen Technologien ist es möglich, Ontologien zu erstellen. Es können Konzepte, die Eigenschaften dieser Konzepte, die hierarchischen und assoziativen Beziehungen zwischen den Konzepten definiert werden und auch, welche Konzepte welche Eigenschaften verwenden und welche Konzepte als Werte der Eigenschaften verwendet werden sollten. [31]

Nach Entwicklung einer Ontologie kann diese zur Repräsentation von Wissen verwendet werden. Zum Beispiel könnte eine Applikation eine Ontologie verwenden, um ihre Daten in einer maschinenverständlichen Sprache zu beschreiben. Dadurch ist es für andere Applikationen, welche dieselbe Ontologie benützen, möglich, die Daten der anderen Applikation in das eigene System zu integrieren, obwohl die beiden Applikationen unabhängig voneinander und nicht mit dem Hintergedanken entwickelt wurden, jemals miteinander zu kommunizieren. Aufgrund der formal beschriebenen Bedeutung innerhalb der Ontologie sind Applikationen auch in der Lage, selbständig Schlussfolgerungen zu ziehen. Als Beispiel betrachte man ein Property *liegtIn*, welches zwei geographi-

sche Regionen zueinander in Beziehung setzt, wobei sich die eine Region innerhalb der anderen befindet. Dieses Property wird als transitives Property definiert, was bedeutet, dass Statements der Form *A liegtIn B* und *B liegtIn C* ein Statement der Form *A liegtIn C* implizieren. Wenn also ein Statement formuliert ist, dass Wien in Österreich liegt und ein weiteres Statement ausdrückt, dass Österreich in Europa liegt, ist eine Applikation in der Lage, die Information abzuleiten, dass Wien eine europäische Stadt ist, obwohl diese Information nicht explizit vorhanden ist. [31]

Das Potenzial für Anwendungen, welche sich qualitativ gut entwickelter Ontologien bedienen, ist sehr groß. Man könnte intelligente elektronische Agenten entwickeln und diese mit dem Auftrag ins WWW schicken, nach bestimmten Informationen zu suchen. Zum Beispiel könnte man dem Agenten seine Präferenzen bezüglich eines Urlaubs mitteilen. Diese Präferenzen könnten sich auf viele verschiedene Dinge beziehen, zum Beispiel die persönliche Einstellung zu hygienischen Zuständen von Unterkünften, die Preisklasse, in Werten ausgedrückte Vorlieben für verschiedenste Interessen (Entspannung, Kulturreisen, Städtereisen), den geplanten Zeitraum, Dinge, welche unbedingt in der Region vorhanden sein müssen, bevorzugte Temperaturen usw. Der Agent könnte dann das WWW nach Informationen durchsuchen, dabei auch nötige Schlussfolgerungen aus den vorhandenen Informationen ziehen und würde dem Benutzer eine Liste mit allen möglichen Reisezielen inklusive vollständig geplanter Reiseroute mit Vorschlägen für Unterkünfte, Ausflüge, Flüge usw. präsentieren. Dies ist nur eines von vielen potentiellen Anwendungsszenarien für Applikationen, welche formal definierte Ontologien zum Austausch, zur Ansammlung und Weiterverarbeitung von Informationen verwenden.

2.2 Resource Description Framework (RDF)

2.2.1 Grundlagen

Das Resource Description Framework (RDF) [18] ist eine Recommendation des World Wide Web Consortiums (W3C)[5] und ist somit eine vom W3C offiziell standardisierte und empfohlene Sprache. Sie ist die Basistechnologie im Bereich des Semantischen Webs, mit welcher es möglich ist, Informationen über Ressourcen im WWW abzubil-

[5] http://www.w3.org/

den. Die Ressourcen beschränken sich in diesem Kontext nicht nur auf physische oder im WWW erreichbare Ressourcen, sondern repräsentieren im Prinzip alles, was man auch in der Realität beschreiben kann, wie zum Beispiel Personen, Produkte, Autos usw. Aber auch abstrakte Begriffe und Konzepte sind davon nicht ausgeschlossen.

RDF wird benötigt, wenn diese Informationen über eine Ressource nicht nur einem Menschen am Bildschirm präsentiert werden sollen, sondern automatisiert von einer Applikation weiterverarbeitet werden sollen. Die beschriebenen Ressourcen werden von einem Uniform Resource Identifier (URI) eindeutig identifiziert und werden mit Hilfe von Properties (Eigenschaften) und deren Werten beschrieben. Eine Beschreibung einer Ressource kann in einem Graphen repräsentiert werden, wie in Abbildung 2 zu sehen ist.

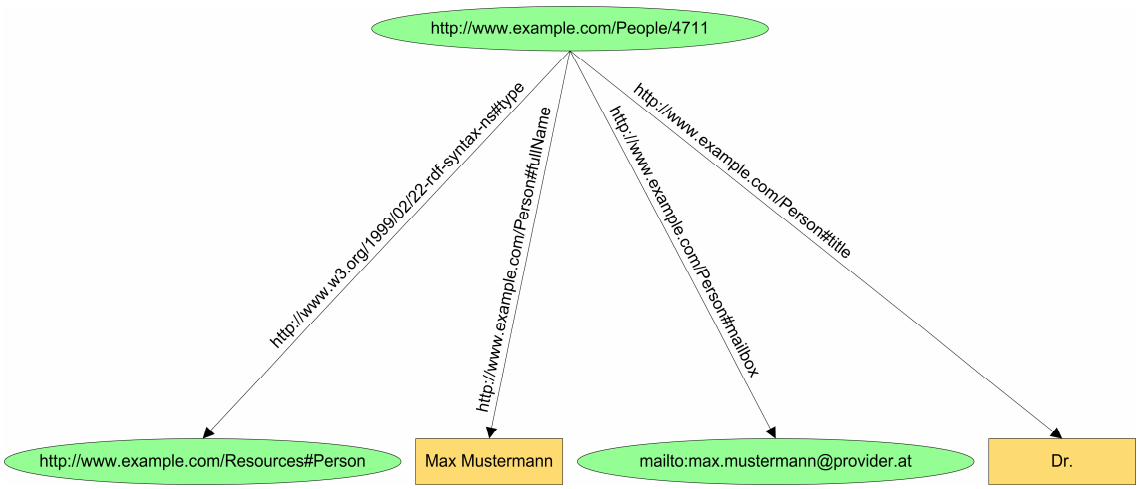

Abb. 2: Ein RDF Graph, welcher die Ressource Max Mustermann beschreibt.

Hier wird eine Ressource mit dem URI `http://www.example.com/People/4711` beschrieben. Dabei handelt es sich um eine *Person* namens *Max Mustermann* mit dem akademischen Titel *Dr.* und der Mailadresse `max.mustermann@provider.at`. Wie in dem Graph ersichtlich, werden in RDF URIs für folgende Elemente verwendet:

- Für Individuen, in diesem Beispiel Max Mustermann, identifiziert durch `http://www.example.com/People/4711`.
- Für Typen von Individuen, beschrieben mit dem Property *rdf:type* und in diesem Beispiel identifiziert durch `http://www.example.com/Resources#Person`.
- Properties von Ressourcen, wie zum Beispiel *mailbox*, identifiziert durch `http://www.example.com/Person#mailbox`.

- Werte von Properties, wie in diesem Beispiel die Mailadresse `max.mustermann@provider.at` für das Property *mailbox*. Werte von Properties können aber auch gewöhnliche Zeichenketten sein, wie *Max Mustermann* für das Property *fullName*.

Eine Ressource wird in RDF mit Hilfe von Statements beschrieben. Statements bestehen immer aus einem Subjekt (der zu beschreibenden Ressource), einem Prädikat (ein Property einer Ressource) und einem Objekt (dem Wert des Properties). Eine Beschreibung einer Ressource ist also ähnlich einem Satz in einer natürlichen Sprache.

Man betrachte zum Beispiel den Satz „**http://www.example.com/index.html** hat einen **Ersteller**, dessen Wert **Max Mustermann** ist“. In diesem doch etwas ungewöhnlich formulierten Satz wäre also `http://www.example.com/index.html` das Subjekt, *Ersteller* das Prädikat und *Max Mustermann* das Objekt. Damit die Bedeutung dieses Satzes auch für Maschinen interpretierbar wird, muss, wie bereits erwähnt, auch das Property eindeutig durch einen URI identifizierbar sein. Die Bedeutung dieses Satzes könnte also mit einem Statement ausgedrückt werden, welches folgende Elemente besitzt:

- Ein Subjekt `http://www.example.com/index.html`
- Ein Prädikat `http://purl.org/dc/elements/1.1/creator`
- Ein Objekt `http://www.example.com/People/4711`

Statements bestehen also immer aus 3 Elementen und werden deshalb auch als Tripel bezeichnet. RDF ist nicht an eine bestimmte Serialisierungsform gebunden, was bedeutet, dass ein RDF-Modell auf mehrere unterschiedliche Weisen dargestellt und serialisiert werden kann. Eine dieser maschinenverständlichen Serialisierungsformen, mit welchen RDF-Modelle beschrieben werden können, ist TURTLE[6]. Das davor beschriebene Tripel würde in TURTLE wie folgt dargestellt werden:

```
ex:index.html   dc:creator   expeople:4711 .
```

Damit nicht immer die vollständigen URIs ausgeschrieben werden müssen, können diesen sogenannte Präfixe vorangestellt werden. In diesem Beispiel müssten drei Präfixe definiert werden:

---

[6] TURTLE steht für Terse RDF Triple Language. http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/

- `ex` für `http://www.example.com/`
- `dc` für `http://purl.org/dc/elements/1.1/`
- `expeople` für `http://www.example.com/People/`

Diese URIs referenzieren auf Namensräume (Namespaces), in welchen das jeweilige Vokabular definiert ist. In weiterer Folge dieser Arbeit werden des Öfteren Präfixe für Namensräume verwendet, welche nicht explizit definiert werden.

2.2.2 Plain Literals

RDF Plain Literals [15] sind gewöhnliche Zeichenketten mit einem optionalen Language-Tag. Die Sprache sollte dabei gemäß der Empfehlung des RFC 3066[7] angegeben werden. Plain Literals werden verwendet, um natürliche Sprache innerhalb eines RDF-Modells auszudrücken. In dem Graph in Abbildung 2 sind *Max Mustermann* und *Dr*. Plain Literals ohne Language-Tag. Würde man zum Beispiel den Titel eines Films beschreiben, würde ein verwendeter Language-Tag die zusätzliche Information ausdrücken, in welcher Sprache der jeweilige Titel formuliert ist.

2.2.3 Typed Literals

Typed Literals sind ähnlich zu verstehen wie Datentypen in Programmiersprachen. In RDF sind Typed Literals Zeichenketten, welche zusätzlich mit einem Datentyp versehen sind. Diese Information muss im RDF-Modell selbst codiert sein, damit Applikationen die Bedeutung eines Wertes richtig interpretieren können. Eine Applikation könnte zwar so programmiert sein, dass sie den Wert eines Properties als richtigen Datentypen interpretiert, jedoch wäre in diesem Fall die Semantik in der Applikation implementiert und dies widerspricht dem Grundziel von RDF, nämlich Daten automatisiert auszutauschen und weiterzuverarbeiten. Betrachten wir beispielsweise die Zeichenkette „10“. Diese Zeichenkette würde im Dezimalsystem die Zahl 10 bedeuten, in einem Zahlensystem mit Basis 2 (binäres Zahlensystem) hingegen würde diese Zeichenkette die Zahl 2 darstellen. Ebenso könnte damit eine Zeichenkette - bestehend aus den Ziffern 1 und 0 – gemeint sein. Plain Literals werden immer als Strings interpretiert, deswegen ist es wichtig, Werte zu typisieren, wenn diese Werte zur weiteren Berechnung verwendet

---

[7] RFC 3066 – Tags for the Identification of Languages, H. Alvestrand, IETF, Januar 2001. http://www.isi.edu/in-notes/rfc3066.txt

werden müssen. Zur eindeutigen Angabe eines bestimmten Datentyps wird dafür ein URI verwendet, welcher in den meisten Fällen auf einen der vordefinierten und standardisierten XML Schema Datentypen[8] verweist.

```
expeople:4711   exterms:alter   „29“^^xsd:integer .
```

In diesem Statement wird das Alter einer bestimmten Person ausgedrückt, wobei explizit angeben wird, dass der Wert für das Alter als Datentyp *Integer* interpretiert werden soll.

2.2.4 Das Property rdf:type

In RDF werden Ressourcen mit Hilfe des vordefinierten Properties *rdf:type* typisiert bzw. klassifiziert. Dieses Prinzip ähnelt jenem von Klassen und deren Objekten in objektorientierten Programmiersprachen. Wird eine Ressource mit einem *rdf:type* Property beschrieben, dann ist der Wert dieses Properties eine Kategorie (eine Klasse) von Dingen und das Subjekt, welches von diesem Property Gebrauch macht, ist eine Instanz dieser Kategorie (Klasse). Zum Beispiel könnte man die Klasse *Auto* mit dem URI `http://www.examples.com/Auto` definieren und ein konkretes Auto (z.B. Audi A4), das beschrieben wird, erhält als Wert des Properties *rdf:type* die Klasse *Auto* zugewiesen.

```
ex:AudiA4   rdf:type   ex:Auto .
```

Mit RDF selbst ist es jedoch nicht möglich, Klassen oder Properties zu definieren. Dafür benötigt man ausdruckskräftigere Sprachen wie RDF Schema (vgl. Kapitel 2.2.7) oder OWL (vgl. Kapitel 2.3).

2.2.5 RDF Containers

In RDF besteht auch die Möglichkeit, Ressourcen zu gruppieren. Dafür stehen drei vordefinierte Typen von Containern zur Verfügung, *rdf:Bag*, *rdf:Seq* und *rdf:Alt*. Ein *rdf:Bag* Container repräsentiert eine Gruppe von Ressourcen, wobei die Reihenfolge der Ressourcen in diesem Container nicht von Relevanz ist. In einem *rdf:Seq* Container hingegen spielt die Reihenfolge der Ressourcen sehr wohl eine Rolle und in einem

[8] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

rdf:Alt Container beschreiben die sich darin befindenden Ressourcen eine Auswahlmöglichkeit für eines der Elemente in diesem Container.

2.2.6 Die Klasse rdf:List

Die drei zuvor beschriebenen RDF Container haben alle jedoch eine Einschränkung, und zwar, dass keine Aussage darüber getroffen werden kann, ob noch andere Ressourcen im selben Container vorhanden sein könnten. Es kann zwar eine Aussage getroffen werden, dass sich die jeweiligen Ressourcen im Container befinden, aber nicht, dass dies auch wirklich alle Ressourcen innerhalb des Containers sind. Zum Beispiel könnte eine andere RDF-Beschreibung noch weitere Ressourcen als Elemente dieses Containers definieren. Eine tatsächlich geschlossene Gruppe von Ressourcen kann mit Hilfe des vordefinierten Typs *rdf:List* beschrieben werden. Eine *rdf:List*, welche 3 Ressourcen (*A*, *B* und *C*) beinhaltet, sieht in TURTLE folgendermaßen aus:

```
(<A>, <B>, <C>)
```

2.2.7 RDF Schema

RDF bietet eine einfache Möglichkeit, Ressourcen mit einem bestimmten Vokabular zu beschreiben. RDF Schema stellt eigene Sprachkonstrukte zur Verfügung, mit welchen ein solches Vokabular definiert werden kann. Somit kann definiert werden, welche Instanzen von bestimmten Klassen welche Properties verwenden sollten und welche Typen von Ressourcen bzw. Datentypen für die Werte der Properties verwendet werden sollten. Zum Beispiel können eigene Klassen von Ressourcen mit der RDF Schema Ressource *rdfs:Class* definiert werden, wie zum Beispiel:

```
ex:Fahrzeug   rdf:type   rdfs:Class .
```

Hier wird eine Klasse namens *Fahrzeug* definiert. Wird dann ein konkretes Fahrzeug beschrieben, könnte man es mit dem *rdf:type* Property als ein Objekt dieser Klasse instantiieren, indem man dem Property die Ressource *ex:Fahrzeug* zuweist. RDF Schema stellt auch ein Property *rdfs:subClassOf* zur Verfügung, mit welchem ausgedrückt werden kann, dass eine Klasse eine speziellere Klasse einer anderen ist, wie zum Beispiel:

```
ex:Kraftfahrzeug   rdfs:subClassOf   ex:Fahrzeug .
ex:Automobil       rdfs:subClassOf   ex:Kraftfahrzeug .
ex:Motorrad        rdfs:subClassOf   ex:Kraftfahrzeug .
```

Das zweite dieser Statements bedeutet, dass *Automobil* eine Subklasse von *Kraftfahrzeug* darstellt. Ein Auto ist also auch immer ein Kraftfahrzeug, dies gilt jedoch nicht im umgekehrten Fall, weil jedes Kraftfahrzeug nicht unbedingt ein Auto sein muss. In RDF können Klassen von mehreren Klassen abgeleitet werden. Im Gegensatz zu den meisten objektorientierten Programmiersprachen ist hier also eine Mehrfachvererbung erlaubt und auch sinnvoll.

Es können auch eigene Properties definiert werden, mit welchen der Charakter von Ressourcen beschrieben wird. Dafür stellt RDF die Klasse *rdf:Property* zur Verfügung, welche als Wert des *rdf:type* Properties verwendet wird, um ein Subjekt als Property zu kennzeichnen.

```
exterms:ps   rdf:type   rdf:Property .
```

Hier wird ein Property *ps* (Pferdestärke) definiert, welches die Pferdestärke eines Kraftfahrzeugs beschreiben soll. Mit Hilfe der RDF Schema Properties *rdfs:domain* und *rdfs:range* wird beschrieben, Instanzen welcher Klassen als Subjekt und Objekt des zu beschreibenden Properties verwendet werden sollen. Im folgenden Statement wird definiert, dass es sich bei Ressourcen, welche das Property *ps* verwenden, um Instanzen der Klasse *Kraftfahrzeug* handelt, welche eine Unterklasse der Klasse *Fahrzeug* ist:

```
exterms:ps   rdfs:domain   ex:Kraftfahrzeug .
```

rdfs:range hingegen definiert, um welche Ressourcen es sich bei der Beschreibung des Properties (also beim Objekt des Statements) handelt. In diesem Beispiel wäre dies der Datentyp *Integer*:

```
exterms:ps   rdfs:range   xsd:integer .
```

Als Range könnte allerdings auch eine Klasse definiert werden, sodass Werte eines Properties Instanzen einer bestimmten Klasse sein müssen. Ein Property, welches den Hersteller von Fahrzeugen beschreibt, hätte als Range zum Beispiel Instanzen der Klasse *Hersteller:*

```
exterms:hergestelltVon   rdf:type      rdf:Property .
exterms:hergestelltVon   rdfs:domain   ex:Fahrzeug .
exterms:hergestelltVon   rdfs:range    ex:Hersteller .
```

Die Semantik dieser Definitionen unterscheidet sich von jener Semantik in objektorientierten Programmiersprachen, in welchen ein Attribut einer bestimmten Klasse zugewiesen wird und auch nur von Objekten dieser Klasse verwendet werden darf. Mit den Properties *rdfs:domain* und *rdfs:range* wird allerdings nicht vorgeschrieben, dass nur

bestimmte Instanzen einer Klasse als Domain und Range eines Properties verwendet werden dürfen, denn theoretisch dürfen alle Ressourcen ein bestimmtes Property verwenden. Mit diesen Properties wird hingegen ausgedrückt, dass es sich bei allen Ressourcen, die als Domain oder Range eines Properties verwendet werden, um Instanzen einer bestimmten Klasse handelt, und zwar jener Klasse, welche als Domain bzw. Range des jeweiligen Properties definiert ist. Wird zum Beispiel das Property *hergestellt-Von* von einer bestimmten Ressource verwendet, dann ist diese Ressource implizit eine Instanz der Klasse *Fahrzeug*, selbst wenn die Ressource als eine Instanz einer anderen Klasse oder überhaupt nicht instantiiert wurde. Die Properties *rdfs:domain* und *rdfs:range* ermöglichen somit auch Schlussfolgerungen, weil die Information, dass es sich bei einer bestimmten Ressource um eine Instanz einer bestimmten Klasse handelt, implizit vorhanden ist, wenn eine Ressource ein bestimmtes Property verwendet oder als Objekt eines bestimmten Properties verwendet wird.

Die Definition von Domain und Range eines bestimmten Properties gilt nicht nur für Instanzen der angegebenen Klasse, sondern auch für Instanzen aller Subklassen der angegebenen Klasse. In diesem Fall wird zum Beispiel die Schlussfolgerung gezogen, dass es sich bei einer Ressource um eine Instanz der Klasse *Fahrzeug* handeln muss, wenn das Property *hergestelltVon* auch von einer Instanz der Klasse *Kraftfahrzeug* verwendet wird. Dies ist konsistent mit dem bisher definierten Modell, da *Kraftfahrzeug* als eine Unterklasse von *Fahrzeug* definiert ist.

Properties können, wie Klassen auch, von anderen Properties abgeleitet werden. Dies wird mit Hilfe des RDF Schema Properties *rdfs:subPropertyOf* ausgedrückt. Wurde z.B. schon ein Property namens *name* (zur Beschreibung eines Namens einer Person) definiert, könnte ein weiteres Property *kuenstlername* definiert werden, welches von *name* abgeleitet wird.

```
ex:Schauspieler          rdf:type             rdfs:Class .
exterms:name             rdf:type             rdf:Property .
exterms:name             rdfs:domain          ex:Schauspieler .
exterms:name             rdfs:range           xsd:string .
exterms:kuenstlername    rdf:type             rdf:Property .
exterms:kuenstlername    rdfs:subPropertyOf   exterms:name .
```

Ein Künstlername ist also eine spezialisierte Form eines Namens. Ein Künstlername ist immer zugleich auch ein Name, aber nicht jeder Name ist auch ein Künstlername. Abgeleitete Properties übernehmen auch bezüglich Domain und Range die Definitionen ihrer Superklassen.

Das RDF Schema Property *rdfs:label* [4] wird verwendet, um eine für Menschen lesbare Form des Namens einer Ressource darzustellen, weil in den meisten Fällen anhand des URIs keine Rückschlüsse auf den Namen einer Ressource gezogen werden können. Im zuvor beschriebenen Beispiel hätte man auch definieren können, dass *exterms:name* ein Subproperty von *rdfs:label* ist.

```
exterms:name   rdfs:subPropertyOf   rdfs:label .
```

rdfs:label kann von allen Ressourcen verwendet werden und erwartet als Range einen Literal:

```
ex:86294   rdfs:label   „Johnny Depp“ .
```

Mit Hilfe von RDF Schema können also Vokabulare definiert werden, welche dann zur Beschreibung von konkreten Ressourcen verwendet werden können. Zum Beispiel könnte man zuvor definiertes Vokabular an anderer Stelle wie folgt zur Beschreibung nutzen:

```
ex:VW          rdf:type                ex:Hersteller .
ex:Yamaha      rdf:type                ex:Hersteller .
ex:AudiA4      rdf:type                ex:Automobil .
ex:AudiA4      exterms:ps              “138”xsd:integer .
ex:AudiA4      exterms:hergestelltVon  ex:VW .
ex:YamahaYZF   rdf:type                ex:Motorrad .
ex:YamahaYZF   rdf:ps                  „152“xsd:integer .
ex:YamahaYZF   exterms:hergestelltVon  ex:Yamaha .
ex:BBardot     rdf:type                ex:Schauspieler .
ex:BBardot     exterms:kuenstlername   „Brigitte Bardot“xsd:string .
ex:BBardot     exterms:name            „Camille Javal“xsd:string .
```

Mit RDF und RDF Schema ist es jedoch nicht möglich, ausdrucksstärkere Ontologien abzubilden (vgl. Kapitel 2.3). Allerdings ist RDF erweiterbar und deshalb sehr flexibel in der Anwendung. Es existieren noch weitere Sprachen, welche auf RDF aufbauen und mit welchen ausdrucksstärkere Ontologien beschrieben werden können. Die wohl bekannteste und wichtigste von diesen wird im nächsten Kapitel behandelt.

2.3 Web Ontology Language (OWL)

Mit Hilfe von RDF Schema ist es möglich, Klassen und Properties und ihre hierarchischen Beziehungen zueinander zu definieren. Wenn komplexere Ontologien erstellt werden sollen, stoßen RDF und RDF Schema an ihre Grenzen. Es folgen einige Beispiele, welche in RDF Schema nicht ausgedrückt werden können:

- Es können keine Einschränkungen bezüglich der Kardinalität eines Properties ausgedrückt werden, z.B. dass ein bestimmtes Property höchstens einmal von einem bestimmten Subjekt verwendet werden darf.
- Es können keine Properties als transitive Properties definiert werden (vgl. Kapitel 2.3.4).
- Es kann nicht ausgedrückt werden, dass ein bestimmtes Property ein eindeutiger Bezeichner von Instanzen einer bestimmten Klasse ist.
- Es können keine neuen Klassen mit Hilfe von Mengenoperationen an bereits existierenden Klassen beschrieben werden. Zum Beispiel könnte man neue Klassen als die Vereinigung oder den Durchschnitt zweier Klassen definieren oder behaupten, dass zwei Klassen zueinander disjunkt sind.

Damit Applikationen auch selbständig schlussfolgern können und die Daten automatisiert und sinnvoll weiterverarbeiten können, müssen diese komplexeren Eigenschaften von Ontologien beschrieben werden können. Mit Hilfe der Web Ontology Language (OWL)[9] [19, 31] ist es möglich, die Semantik eines Vokabulars formal zu beschreiben, und zwar in einer weitaus ausdrucksstärkeren Form als es mit RDF Schema möglich ist. OWL ist, wie auch RDF und RDF Schema, eine Empfehlung (Recommendation) des W3Cs. In einer OWL Ontologie werden, wie in RDF Schema, die Klassen, deren Properties und die Instanzen der Klassen beschrieben, nur eben detaillierter und mit stärkerem Ausdruck.

OWL stellt drei verschieden ausdrucksstarke Versionen des Standards zur Verfügung:

- *OWL Lite*: Diese Version von OWL ist die ausdrucksschwächste von allen drei Versionen. OWL Lite eignet sich vor allem zur Erstellung von hierarchischen Klassifikationsschemen. Wenn allerdings spezifischere Einschränkungen im Klassifikationsschema definiert werden sollen, eignet sich OWL Lite nicht mehr unbedingt. Zum Beispiel ist es mit OWL Lite zwar möglich, Einschränkungen bezüglich der Kardinalität von Properties zu definieren, allerdings kann für ein Property nur entweder Kardinalität 0 oder Kardinalität 1 gewählt werden.
- *OWL DL*: Diese Version von OWL stellt alle im OWL-Vokabular vorhandenen Sprachkonstrukte zur Verfügung, was bedeutet, dass die volle Ausdruckskraft

von OWL ausgenützt werden kann. Es gibt aber dennoch ein paar Einschränkungen. Zum Beispiel darf eine Klasse nicht zugleich auch eine Instanz oder ein Property sein und ein Property darf auch nicht zugleich eine Klasse oder eine Instanz sein. Diese in OWL DL vorhandenen Einschränkungen stellen allerdings sicher, dass eindeutige Schlussfolgerungen aus den vorhandenen Beschreibungen gezogen werden können.

- *OWL Full*: Diese Version ist für Applikationen geeignet, deren Anforderungen die vollste Ausdruckskraft von OWL erfordern. Hier sind die Einschränkungen von OWL DL nicht mehr vorhanden. Zum Beispiel kann in OWL Full eine Klasse zugleich als eine Sammlung von Instanzen und als eine elementare Instanz definiert werden. Allerdings kann es vorkommen, dass eindeutige Schlussfolgerunen aus einer mit OWL Full definierten Ontologie nicht mehr möglich sind.

Welche von diesen Versionen für eine bestimmte Applikation am geeignetsten ist, muss im konkreten Einzelfall entschieden werden. Es sollte allerdings beachtet werden, dass die Komplexität der verschiedenen Versionen mit der steigenden Ausdruckskraft der verschiedenen Versionen korreliert. Demnach sollte die ausdrucksschwächste dieser drei Versionen gewählt werden, mit welcher es möglich ist, die Semantik der zu erstellenden Ontologie vollständig zu beschreiben. Es sollte also keine komplexere Version verwendet werden, wenn die intendierte Semantik auch mit einer ausdrucksschwächeren Version modellierbar ist.

In den folgenden Unterkapiteln werden jene Elemente von OWL näher erläutert, welche für das weitere Verständnis von SKOS relevant sind. Die dabei vorkommenden Beispiele sind nicht in TURTLE, sondern in einer anderen Serialisierungsform beschrieben, nämlich in RDF/XML[10]. Dabei handelt es sich um ein XML-Vokabular, mit welchem RDF-Modelle beschrieben werden können.

---

[9] http://www.w3.org/2004/OWL/

[10] http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/

2.3.1 Die Klasse owl:Class

Wie auch in RDF Schema können in OWL eigene Klassen definiert werden, wobei die Beziehungen zwischen allen Klassen in einer bestimmten Ontologie die Semantik dieser Ontologie beschreiben. Im folgenden Beispiel wird eine Klasse namens *Tier* definiert:

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

Zur Vererbung von Klassen wird auch in OWL das bereits in Kapitel 2.2.7 beschriebene RDF Schema Property *rdfs:subClassOf* verwendet. Zur Beschreibung eines konkreten Tiers könnte man nun eine Instanz definieren, wie zum Beispiel *Flipper* als eine Instanz der Klasse *Delphin*:

```
<owl:Class rdf:ID="Delphin">
  <rdfs:subClassOf rdf:resource=#Tier />
</owl:Class>

<Delphin rdf:ID="Flipper" />
```

2.3.2 Datatype Properties

Es gibt in OWL zwei Formen von Properties, nämlich Datatype Properties und Object Properties (vgl. Kapitel 2.3.3). Ein Datatype Property beschreibt eine Relation zwischen einer Instanz einer Klasse und einem RDF Typed Literal (vgl. Kapitel 2.2.3). Zum Beispiel könnte man ein Property namens *alter* definieren, dessen Wert als Datentyp *Integer* interpretiert werden sollte.

```
<owl:DatatypeProperty rdf:ID="alter">
   <rdfs:domain rdf:resource="#Tier" />
   <rdfs:range  rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>
```

&xsd; ist in diesem Fall ein sogenanntes Entity und steht für den URI `http://www.w3.org/2001/XMLSchema#`. Es können weitere Einschränkungen des Properties getroffen werden, zum Beispiel, dass jedes Tier höchstens einen Wert für sein Alter besitzen darf.

2.3.3 Object Properties

Ein ObjectProperty beschreibt eine Relation zwischen zwei Ressourcen. Das in Kapitel 2.1 bereits beschriebene Property namens *liegtIn* ist ein Beispiel für ein Object Property. Dieses Property setzt eine Instanz der Klasse *GeographischesObjekt* und eine Instanz der Klasse *Region* zueinander in Beziehung:

```
<owl:DatatypeProperty rdf:ID="liegtIn">
   <rdfs:domain rdf:resource="#GeographischesObjekt" />
   <rdfs:range  rdf:resource="#Region"/>
</owl:DatatypeProperty>
```

Nun könnte man z.B. ausdrücken, dass die Stadt Wien in Österreich liegt. Die Klasse *Stadt* ist dabei von der Klasse *GeographischesObjekt* abgeleitet, ebenso die Klasse *Region:*

```
<Region rdf:ID="Oesterreich" />
<Stadt rdf:ID="Wien">
   <liegtIn rdf:resource="#Oesterreich" />
</Stadt>
```

Auch ObjectProperties können wie DatatypeProperties bezüglich ihrer Kardinalität eingeschränkt werden.

2.3.4 Eigenschaften von Properties

Properties können in OWL auch als verschiedene Typen von Properties klassifiziert werden. Dies ist notwendig, damit es für Applikationen möglich wird, eigene Schlussfolgerungen aus den Ressourcenbeschreibungen zu ziehen.

Wird ein Property *P* als transitiv spezifiziert, dann gilt für die Ressourcen *X*, *Y* und *Z*:

P (X, Y) und P (Y, Z) impliziert P (X, Z).

Das im letzten Kapitel beschriebene ObjectProperty *liegtIn* ist bezüglich seiner Semantik transitiv. Folgendes Statement würde zur Definition des Properties hinzukommen:

```
<rdf:type rdf:resource="&owl;TransitiveProperty" />
```

&owl; ist hier wiederum ein Entity und steht für den URI `http://www.w3.org/2002/07/owl#`. Zum besseren Verständnis betrachte man folgendes Beispiel:

```
<Region rdf:ID="Europa" />

<Stadt rdf:ID="Wien">
   <liegtIn rdf:resource="#Oesterreich" />
</Stadt>

<Region rdf:ID="Oesterreich">
   <liegtIn rdf:resource="#Europa" />
</Region>
```

Da die Stadt Wien in Österreich liegt und Österreich ein europäisches Land ist, wird Wien von einer Applikation auch als europäische Stadt interpretiert, da *liegtIn* als transitives Property definiert ist.

Wird ein Property *P* als symmetrisch definiert, dann gelten für die Ressourcen *X* und *Y*:

P (X, Y) impliziert P (Y, X) und P (Y, X) impliziert P (X, Y).

Zum Beispiel könnte man ein Property *angrenzendeRegion* definieren:

```
<owl:ObjectProperty rdf:ID="angrenzendeRegion">
   <rdf:type rdf:resource="&owl;SymmetricProperty" />
   <rdfs:domain rdf:resource="#Region" />
   <rdfs:range rdf:resource="#Region" />
</owl:ObjectProperty>

<Region rdf:ID="Italien">
   <liegtIn rdf:resource="#Europa" />
</Region>

<Region rdf:ID="Oesterreich">
   <liegtIn rdf:resource="#Europa" />
   <angrenzendeRegion rdf:resource="#Italien" />
</Region>
```

Da das Property *angrenzendeRegion* als symmetrisch definiert ist, grenzt nicht nur Österreich an Italien, sondern auch implizit Italien an Österreich.

Wird ein Property *P* als funktional klassifiziert, dann gilt für die Ressourcen *X*, *Y* und *Z*:

P (X, Y) und P (X, Z) impliziert Y = Z.

In natürlicher Sprache bedeutet dies, dass das funktionale Property nur einen Wert annehmen darf, also nicht mehrere. In unseren Beispielen könnte das Property *alter* zur Beschreibung der Lebensjahre als funktionales Property definiert werden, weil Lebewesen nur einen Wert für ihr Alter besitzen können, nicht mehrere.

```
<owl:DatatypeProperty rdf:ID="alter">
   <rdf:type rdf:resource="&owl;FunctionalProperty" />
   <rdfs:domain rdf:resource="#Tier" />
   <rdfs:range  rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>
```

Wenn ein Property *P1* als *owl:inverseOf* von *P2* definiert wird, gelten für die Ressourcen *X* und *Y*:

P1 (X, Y) impliziert P2 (Y, X) und P2 (Y, X) impliziert P1 (X, Y).

Dies bedeutet, *P1* ist das Gegenteil von *P2* und umgekehrt. Zum Beispiel könnte ein Property *hergestelltVon* definiert werden, welches ein Produkt und einen Hersteller zu-

einander in Beziehung setzt. Für dieses Property würde die Klasse *Produkt* als Domain und die Klasse *Hersteller* als Range definiert werden. Es könnte aber auch ein Property *stelltHer* existieren, welches als Domain die Klasse *Hersteller* und als Range die Klasse *Produkt* erwartet:

```
<owl:ObjectProperty rdf:ID="hergestelltVon">
   <rdfs:domain rdf:resource="#Produkt" />
   <rdfs:range  rdf:resource="#Hersteller"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="stelltHer">
   <owl:inverseOf rdf:resource="#hergestelltVon" />
</owl:ObjectProperty>
```

2.3.5 Disjunkte Klassen

Klassen können als disjunkt definiert werden. Dies wird mit dem Konstruktor *owl:disjointWith* ausgedrückt und bedeutet, dass Instanzen der einen Klasse nicht auch zugleich Instanzen der anderen Klasse sein können, wie zum Beispiel:

```
<owl:Class rdf:ID="Saeugetier">
   <rdfs:subClassOf rdf:resource="#Tier" />
</owl:Class>

<owl:Class rdf:ID="Fisch">
   <rdfs:subClassOf rdf:resource="#Tier" />
   <owl:disjointWith rdf:resource="#Saeugetier" />
</owl:Class>
```

Dieses Beispiel beschreibt zwei Klassen, nämlich *Saeugetier* und *Fisch*, welche beide von der Klasse *Tier* abgeleitet sind. Da kein Tier existiert, das zugleich Säugetier und Fisch sein kann, werden diese beiden Mengen als disjunkt bezeichnet.

2.4 Simple Knowledge Organzization System (SKOS)

Wie bereits in Kapitel 1.2 erwähnt, soll das Metamodell der Sprache Simple Knowledge Organization System (SKOS) [13, 20, 21] in einem semantischen Wiki zur Verfügung gestellt werden, sodass es möglich ist, die Artikel innerhalb des Wikis mit den Sprachkonstrukten von SKOS zu erweitern. SKOS ist noch keine Recommendation des W3C, sondern befindet sich noch im Working Draft Status. Aber aufgrund der einfachen Anwendung ist SKOS dennoch ein beliebter und viel verwendeter Standard zum Austausch von Daten.

Vor allem in den Bibliotheks- und Informationswissenschaften werden Werkzeuge zur Organisation von großen Mengen von Objekten, wie zum Beispiel Büchern, benötigt. Diese Werkzeuge werden als Knowledge Organization Systems (KOS) bezeichnet. Es gibt verschiedene Typen von KOS, wie z.B. Thesauri, Klassifikationsschemen oder Taxonomien, welche jedoch alle auch viele Dinge gemeinsam haben. SKOS ist ein einheitliches Datenmodell zur Beschreibung von KOS, welche somit maschinenverständlich ausgedrückt werden können. Somit können die Daten eines KOS zwischen Anwendungen ausgetauscht und auch in einem einheitlichen und maschinenverständlichen Standard veröffentlicht werden. Dies macht vor allem Sinn, weil KOS zumeist auch in ähnlichen Anwendungen verwendet werden.

SKOS selbst ist streng betrachtet eigentlich keine Sprache zur Beschreibung von formalem Wissen. Es ist eigentlich selbst eine in OWL definierte Ontologie zur Beschreibung von KOS. Das bedeutet, die KOS werden eigentlich schon auf Instanzebene beschrieben, und zwar mit Hilfe von RDF-Tripeln. Dennoch ist SKOS flexibel, denn das Vokabular kann mit Hilfe von ausdrucksstärkeren Sprachen wie OWL erweitert werden. weswegen man mit SKOS trotzdem in der Lage ist, Lightweight-Ontologien zu entwickeln. Allerdings sind diese Ontologien wirklich sehr „leicht“ und eignen sich zum Beispiel kaum für Schlussfolgerungen aus vorhandenen Wissensrepräsentationen. Aber SKOS wurde auch nicht unbedingt dafür entwickelt. Ein weiteres Ziel des Semantischen Webs besteht nämlich in der besseren Organisation von unstrukturiert und informal beschriebener Information im WWW, sodass diese Information schneller und leichter auffindbar wird. RDF Schema und OWL sind Sprachen zur Beschreibung der Bedeutung von Information, aber um diese Technologien auch tatsächlich in einem mit Daten reich befüllten Netzwerk wie dem WWW anwenden zu können, müssen auch detaillierte Übersichten eines bestimmten Wissensbereichs erstellt werden, was in den meisten Fällen nicht automatisiert erfolgen kann. SKOS wurde also im eigentlichen Sinne erschaffen, um detaillierte Übersichten eines bestimmten Wissensbereichs zu erstellen, zum Beispiel in Form von hierarchisch geordneten Schlagwörtern.

In SKOS existieren vier Klassen zur Beschreibung von Knowledge Organization Systems, nämlich *Concept*, *ConceptScheme*, *Collection* und *OrderedCollection*. Instanzen dieser Klassen sind durch einen URI eindeutig bestimmt, womit sie innerhalb des WWW ohne die Möglichkeit einer Verwechslung referenziert werden können. Die Instanzen dieser SKOS-Klassen können mit Hilfe von vordefinierten Properties, also einem Vokabular, beschrieben werden.

In den folgenden Unterkapiteln werden alle in SKOS verfügbaren Klassen und deren Properties näher beschrieben. Die dabei verwendeten Beispiele sind in TURTLE formuliert.

2.4.1 Die Klasse skos:Concept

Diese Klasse repräsentiert ein Konzept. Ein Konzept ist der allgemeine Begriff für eine zu beschreibende konkrete Instanz, wie zum Beispiel *Auto* für *Audi A4* oder *Schauspieler* für *Johnny Depp*. Ein Konzept kann also auch als Begriff verstanden werden, wobei abstrakte Begriffe davon nicht ausgeschlossen sind.

Das SKOS-Modell ist, wie bereits erwähnt, eine in OWL definierte Ontologie und deshalb ist *skos:Concept* eine Instanz der Klasse *owl:Class*. Soll eine zu beschreibende Ressource ein SKOS-Konzept darstellen, muss diese Ressource als *skos:Concept* typisiert werden. Dies geschieht mit dem in Kapitel 2.2.4 beschriebenen Property *rdf:type*. Zum Beispiel könnte man ein SKOS-Konzept namens *Tier* definieren:

```
<Tier>   rdf:type   skos:Concept .
```

2.4.2 Konzeptschemen

Konzepte, die aus welchen Gründen auch immer inhaltlich in Zusammenhang stehen, können in Konzeptschemen eingeordnet und zusammengefasst werden. Konzeptschemen sind daher Ansammlungen von Konzepten und sind als eine Art Kategorie aufzufassen. Sie werden auch verwendet, um alle Konzepte innerhalb eines bestimmten KOS zusammenzufassen.

Die Klasse *skos:ConceptScheme* ist ebenfalls, wie alle SKOS-Klassen, eine Instanz der Klasse *owl:Class*. Wie SKOS-Konzepte auch, müssen Ressourcen, welche SKOS-Konzeptschemen darstellen sollen, als *skos:ConceptScheme* typisiert werden, wie im folgenden Beispiel die Ressource *Biologie*:

```
<Biologie>   rdf:type   skos:ConceptScheme .
```

Mit dem Property *skos:inScheme* ist es möglich, Konzepte in Konzeptschemen einzuordnen. *skos:inScheme* ist eine Instanz von *owl:ObjectProperty*, also ein Property, welches zwei Ressourcen zueinander in Beziehung setzt. Dieses Property hat die Klasse *skos:Concept* als Domain und die Klasse *skos:ConceptScheme* als Range definiert.

Wenn also beispielsweise das Konzept *Tier* dem Konzeptschema *Biologie* zugeordnet werden sollte, würde dies wie folgt formuliert werden:

```
<Tier>   skos:inScheme   <Biologie> .
```

Ein Konzept muss nicht unbedingt einem Konzeptschema zugeordnet werden, darf aber auch in unendlich vielen Konzeptschemen enthalten sein. Umgekehrt dürfen Konzeptschemen auch leer sein (wenn kein Konzept einem bestimmten Konzeptschema zugeordnet wird), aber auch unendlich viele Konzepte enthalten.

Das Property *skos:hasTopConcept* sagt aus, welche Konzepte in der hierarchischen Struktur innerhalb eines bestimmten Konzeptschemas ganz oben stehen, also Konzepte, für welche selbst keine Superkonzepte formuliert sind. Dieses Property ist ebenfalls eine Instanz der Klasse *owl:ObjectProperty*, was wiederum bedeutet, dass es zwei Ressourcen zueinander in Beziehung setzt. Statements, welche dieses Property als Prädikat verwenden, haben die Klasse *skos:ConceptScheme* als Domain und die Klasse *skos:Concept* als Range. Für das Konzept *Tier* sind keine Superkonzepte innerhalb des Konzeptschemas *Biologie* definiert:

```
<Biologie>   skos:hasTopConcept   <Tier> .
```

Allerdings existiert keine Integritätsbedingung, dass *Tier* auch wirklich keine Superkonzepte besitzen darf. Es wäre also möglich, ein weiteres Konzept namens *Lebewesen* als Superkonzept von *Tier* im Schema *Biologie* zu definieren. Dies wäre in SKOS dennoch valide. Dadurch wird gewährleistet, dass ein umfangreiches SKOS-Modell nach einer Änderung konsistent bleibt, ohne dass weitere Ressourcen des Modells aktualisiert werden müssen.

Das Property *skos:topConceptOf* ist als *owl:inverseOf* von *skos:hasTopConcept* definiert. Dieses Property wurde allerdings erst im fortgeschrittenen Stadium dieser Arbeit ins SKOS-Vokabular aufgenommen und wird daher in dieser Arbeit nicht mehr berücksichtigt (vgl. Kapitel 4.1.2)

In weiterer Folge dieser Arbeit werden alle für SKOS geltenden Integritätsbedingungen mit IB abgekürzt, fortlaufend nummeriert und ihre Erklärungen mit roter Farbe hinterlegt.

IB 1: *skos:ConceptScheme* und *skos:Concept* sind disjunkt. Dies bedeutet, eine Ressource kann nicht als Konzeptschema und Konzept zugleich definiert werden.

2.4.3 Lexical Labels

Da URIs in den meisten Fällen nicht sehr viel Aufschluss über den Namen einer Ressource geben, werden diese mit sogenannten Lexical Labels explizit benannt. Es existieren in SKOS drei verschiedene Arten von Labels, nämlich preferred Labels, alternative Labels und hidden Labels. *skos:prefLabel* bezeichnet den bevorzugten Namen einer Ressource, also jene Zeichenkette, die am ehesten die Bedeutung des Konzepts in einer natürlichen Sprache darstellt. *skos:altLabel* bezeichnet hingegen Synonyme des bevorzugten Labels, also alternative Ausdrücke, die ungefähr dieselbe Bedeutung wie jene des preferred Labels haben. Mit Hilfe des Properties *skos:hiddenLabel* können Zeichenketten angegeben werden, die bei der textbasierten Suche dazu verwendet werden, den gesuchten Ausdruck dennoch zu finden, wenn dem Benutzer bei der Eingabe des Suchbegriffs ein Rechtschreib- oder Tippfehler unterlaufen ist. Denn stimmt die falsche Schreibweise mit einem hidden Label überein, dann wird der gesuchte Begriff dennoch gefunden. Also werden vor allem falsche Schreibweisen eines Begriffs als hidden Label angegeben.

Alle Lexical Labels sind Instanzen der Klasse *owl:DatatypeProperty* und Subproperties von *rdfs:label* (vgl. Kapitel 2.2.7). Als Domain von Lexical Labels ist die Klasse *rdfs:Resource* definiert. Dies bedeutet, Instanzen aller in SKOS definierten Klassen können als Domain eines Statements, dessen Prädikat ein Lexical Label ist, verwendet werden. Als Range von Lexical Labels ist ein Plain Literal (vgl. Kapitel 2.2.2) definiert. Dies bedeutet, der Wert für ein Lexical Label wird durch eine beliebige Zeichenkette mit einem optionalen Language-Tag dargestellt. Der Language-Tag drückt dabei explizit die Sprache aus, in welcher die jeweilige Zeichenkette formuliert ist.

Für das Konzept *Tier* könnten folgende Lexical Labels definiert werden:

```
<Tier>
   skos:prefLabel     „Tier“@de ;
   skos:prefLabel     „animal“@en ;
   skos:altLabel      „Vieh“@de ;
   skos:hiddenLabel   „Tiehr“@de ;
   skos:hiddenLabel   „annimal“@en .
```

Dies ist eine Kurzschreibform von Statements in TURTLE. Solange ein Strichpunkt als Begrenzer eines Statements verwendet wird, gilt für die nächste Zeile dieselbe Ressource als Subjekt des Statements wie in der Zeile davor.

Die angegebene Sprache wird in TURTLE der eigentlichen Zeichenkette und einem folgenden @ angehängt. Die Angabe der Sprache ist jedoch nicht zwingend, sondern

optional. Da mehrere Sprachen angegeben werden können, beträgt die Kardinalität eines Lexical Labels für eine Ressource unendlich.

IB 2: *skos:prefLabel*, *skos:altLabel* und *skos:hiddenLabel* sind paarweise disjunkt. Dies bedeutet, es dürfen für eine Ressource nicht dieselben Bezeichnungen in denselben Sprachen für unterschiedliche Labels formuliert werden. Dieselbe Zeichenkette in derselben Sprache darf also zum Beispiel nicht als preferred Label und als alternative Label definiert werden.

IB 3: Es darf nur ein *skos:prefLabel* pro Sprache für eine Ressource vergeben werden. Dies bedeutet, es dürfen unendlich viele prefLabels verwendet werden, solange keine Sprache doppelt oder mehrfach für preferred Labels verwendet wird.

2.4.4 Notations

SKOS-Konzepte können mit einer oder mehreren Notations versehen werden, welche zur eindeutigen Identifikation von Konzepten innerhalb eines bestimmten Konzeptschemas dienen. Diese können zum Beispiel verwendet werden, um den Zusammenhang zwischen einer digitalen Beschreibung und einer physisch tatsächlich existierenden Ressource, welche in einem anderen Identifikationssystem mit der angegebenen Notation kodiert ist, herzustellen.

Eine Notation ist ebenfalls eine Instanz der Klasse *owl:DatatypeProperty*. Als Objekt eines Statements, dessen Prädikat das Property *skos:notation* ist, sollte laut Empfehlung der SKOS-Referenz immer ein Typed Literal (vgl. Kapitel 2.2.3) verwendet werden. Zum Beispiel könnte man dem Konzept *Tier* eine Notation zuweisen.

```
<Tier>
   skos:notation   „Tier“^^http://www.w3.org/2001/XMLSchema#string .
```

Der Datentyp wird in TURTLE, getrennt durch zwei Zeichen („^^“), an die eigentliche Zeichenkette angehängt. Dabei kann es sich, wie in diesem Beispiel, um einen vordefinierten XML Schema Datentypen oder einen selbst definierten Datentypen handeln.

2.4.5 Documentation Properties

Documentation Properties werden verwendet, um zusätzliche Informationen über Ressourcen in natürlicher Sprache auszudrücken. Mit ihnen ist es also möglich, die Ressourcen informal zu dokumentieren. Der Domain von Documentation Properties ist die

Klasse *rdfs:Resource*, was bedeutet, dass alle in SKOS definierten Klassen mit diesen Documentation Properties dokumentiert werden können. SKOS stellt sieben verschiedene Documentation Properties zur Verfügung.

skos:note dient zur allgemeinen Dokumentation von Ressourcen und ist ein Superproperty von den restlichen sechs Documentation Properties.

Mit dem Property *skos:changeNote* können Änderungen an einer Ressource fest gehalten werden, um die Administration und Wartung der Ressourcen zu vereinfachen.

skos:definition beschreibt eine vollständige Erklärung der intendierten Semantik der jeweiligen Ressource.

Mit Hilfe des Properties *skos:editorialNote* können wichtige Hinweise angegeben werden, welche für das spätere Editieren der Ressource nützlich sein könnten, wie zum Beispiel noch nicht in der Beschreibung berücksichtigte Eigenschaften der Ressource. Sie dienen auch, ähnlich wie *changeNotes*, der besseren Wartbarkeit eines KOS, wenn die Datenmenge unübersichtlich groß wird.

skos:example gibt ein Beispiel für die Verwendung einer bestimmten Ressource, zum Beispiel in einem natürlichen Satz.

Das Property *skos:historyNote* hält signifikante Änderungen in der Bedeutung einer Ressource fest. Dies bezieht sich nicht auf Änderungen in der Beschreibung der Ressource, sondern auf Änderungen in ihrer Bedeutung selbst.

skos:scopeNote liefert, unter Umständen auch unvollständig, Informationen über die intendierte Bedeutung einer Ressource. Dies bezieht sich vor allem darauf, wie die Ressource zu verwenden ist.

In SKOS existieren drei verschiedene Muster zur Dokumentation von Ressourcen. Das erste Muster davon ist die Dokumentation als Plain Literal (vgl. Kapitel 2.2.2), was bedeutet, dass das Objekt des Statements ein Plain Literal ist:

```
<Tier>   skos:note   „Dies ist ein Tier-Konzept“ .
```

Die Dokumentation einer Ressource kann auch mit einer beliebigen RDF-Beschreibung (related resource description) erfolgen. Hier ein Beispiel in RDF/XML:

```
<skos:Concept rdf:about="http://www.example.org/concepts#laptops">
   <skos:changeNote rdf:parseType="Resource">
      <rdf:value>The preferred label for this concept changed from
      'lap top computers' to 'notebook computers' on 30 Jan 2008.
      </rdf:value>
```

```
      <dc:creator>
         <foaf:Person>
            <foaf:name>Max Mustermann</foaf:name>
            <foaf:mbox
               rdf:resource="mailto:max.mustermann@provider.at"/>
         </foaf:Person>
      </dc:creator>
      <dc:date>2008-01-30</dc:date>
   </skos:changeNote>
</skos:Concept>
```

Die dritte Möglichkeit zur Dokumentation besteht in der Referenz auf ein anderes Dokument, wie zum Beispiel:

```
<skos:definition rdf:resource="http://www.example.com/Tier.txt"/>
```

Alle Documentation Properties sind als Instanzen der Klasse *owl:ObjectProperty* definiert. Im Falle der Dokumentation als Plain Literal ist es eigentlich ein *DatatypeProperty*, aber *DatatypeProperty* ist in OWL eine Subklasse von *ObjectProperty*, womit die Definition gültig bleibt. Als Range der Documentation Properties ist die Klasse *rdfs:Resource* definiert. Da Plain Literals ebenfalls Ressourcen darstellen, ist auch diese Definition im Falle der Dokumentation als Plain Literal gültig. Jedes Documentation Property kann beliebig oft zur informalen Beschreibung einer Ressource verwendet werden.

Der folgende Ausschnitt einer Ressourcebeschreibung zeigt Beispiele für die Verwendung dieser Properties:

```
<Tier>
   skos:note            „Dies ist ein Tier-Konzept.“ ;
   skos:changeNote      „Konzept wurde von Konzeptschema ‚Lebewesen’
                        nach Konzeptschema ‚Biologie’ verschoben.“ ;
   skos:definition      „Tiere sind Lebewesen, welche…“ ;
   skos:editorialNote   „Bitte die hierarchischen Beziehungen dieses
                        Konzepts in anderen Konzepten entfernen, wenn
                        dieses Konzept entfernt wird.“ ;
   skos:example         „Streng betrachtet sind auch Menschen Tiere.“ ;
   skos:scopeNote       „Dieses Konzept wird für die Typisierung aller
                        Instanzen von Tieren verwendet.“ .
```

2.4.6 Semantic Relations

Semantic Relation Properties sind Relationen zwischen Konzepten, wobei die Semantik einer Relation durch den Typ des Semantic Relation Properties gegeben ist. SKOS unterscheidet dabei zwischen zwei grundlegenden Kategorien von Relationen, nämlich zwischen hierarchischen und assoziativen Relationen. In Abbildung 3 ist eine hierarchische Übersicht aller Semantic Relation Properties zu sehen.

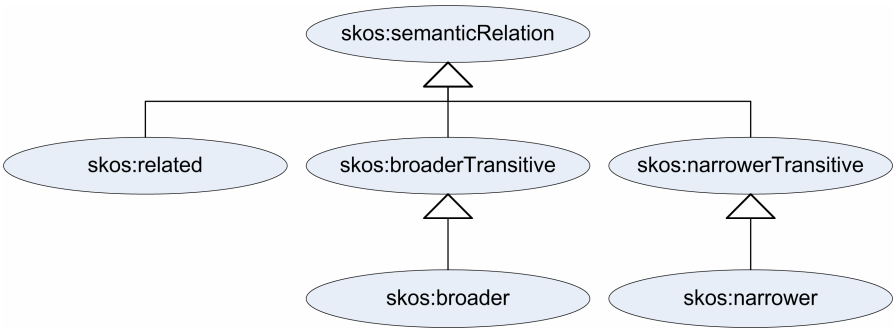

Abb. 3: Die subProperty-Beziehungen zwischen den Semantic Relation Properties in SKOS.

Das Property *skos:semanticRelation* ist das Superproperty aller anderen Semantic Relation Properties, wird aber nicht für die Formulierung von Statements verwendet. Wie bereits erwähnt, handelt es sich bei Semantic Relation Properties um Beziehungen zwischen zwei Konzepten, weshalb sowohl als Domain als auch als Range aller Semantic Relation Properties die Klasse *skos:Concept* definiert ist. Da es sich also um eine Beziehung zwischen zwei Ressourcen handelt, sind alle Properties dieses Typs Instanzen der Klasse *owl:ObjectProperty*.

Eine hierarchische Relation zwischen zwei Konzepten bedeutet, dass ein Konzept allgemeiner (*broader*) als das andere Konzept ist (*narrower*). So bedeutet zum Beispiel `<A> skos:broader <B>`, dass *B* ein allgemeineres Konzept als *A* ist, *A* ist demnach ein Subkonzept von *B*. `<C> narrower <D>` bedeutet, dass *D* ein spezielleres Konzept als *C* ist, demnach wäre *C* ein Superkonzept von *D*. Man könnte also in unserem Beispiel zwei neue Konzepte definieren und hierarchische Beziehungen zum bereits vorhanden Konzept *Tier* formulieren,

```
<Saeugetier>
   rdf:type       skos:Concept ;
   skos:broader   <Tier> .

<Lebewesen>
   rdf:type        skos:Concept ;
   skos:narrower   <Tier> .
```

Das Property *skos:narrower* ist als *owl:inverseOf* von *skos:broader* definiert, was bedeutet, dass eine Formulierung wie `<Tier> skos:narrower <Saeugetier>` redundant ist. Diese Information ist aufgrund dieser *owl:inverseOf*-Definition implizit vorhanden. Dies gilt auch im umgekehrten Fall, also das Statement `<Tier> skos:broader <Lebewesen>` wäre in diesem Fall ebenso überflüssig, was aber nicht bedeutet, dass es nicht trotzdem formuliert werden sollte.

skos:broader und *skos:narrower* sind weder als reflexive noch als irreflexive Properties definiert, was bedeutet, dass Statements der Form `<Tier> skos:broader <Tier>` in SKOS erlaubt sind.

Das Property *skos:related* beschreibt eine assoziative Relation zwischen zwei Konzepten, welche in beliebiger Bedeutung zueinander in Verbindung stehen, wobei allerdings keines der beiden Konzepte allgemeiner oder spezieller als das andere Konzept ist. Zum Beispiel könnte man ein Konzept namens *Tiger* und ein Konzept namens *Loewe* definieren, welche beide Subkonzepte des Konzepts *Saeugetier* darstellen. Sie stehen in der hierarchischen Struktur beide auf derselben Ebene und sind miteinander verwandt, weil es sich bei beiden Säugetieren um Katzen handelt.

```
<Tiger>
   rdf:type       skos:Concept ;
   skos:broader   <Saeugetier> .

<Loewe>
   rdf:type       skos:Concept ;
   skos:broader   <Saeugetier> ;
   skos:related   <Tiger> .
```

In diesem Beispiel wäre es eigentlich besser, noch ein Konzept namens *Katze* zu erstellen, welches von *Saeugetier* abgeleitet werden sollte und ein Superkonzept von *Tiger* und *Loewe* darstellen sollte, um die hierarchische Struktur zu verfeinern. Allerdings existieren keine bestimmten Vorgaben, wie detailliert eine hierarchische Struktur sein sollte.

skos:related ist eine Instanz von *owl:SymmetricProperty*. Dies bedeutet, ein Statement der Form `<Loewe> skos:related <Tiger>` impliziert ein Statement der Form `<Tiger> skos:related <Loewe>`. Wie die Properties zur Beschreibung von hierarchischen Beziehungen auch, ist *skos:related* weder als reflexives noch als irreflexives Property definiert. Dies bedeutet also, dass Staments der Form `<Tiger> skos:related <Tiger>` in SKOS valide sind.

Die Properties *skos:broader* und *skos:narrower* sind nicht als transitives Property definiert. Dies bedeutet, dass Statements der Form `<Tiger> skos:broader <Saeugetier>` und `<Saeugetier> skos:broader <Tier>` kein Statement der Form `<Tiger> skos:broader <Tier>` implizieren. Diese beiden Properties sind mit Absicht nicht transitiv definiert, weil KOS des Öfteren aufgrund der riesigen Datenmengen nicht streng hierarchisch strukturiert sind oder auch hierarchische Relationen zwischen Konzepten definiert sein können, welche bezüglich ihrer Semantik tatsächlich nicht transitiv

sein sollen. Für transitive hierarchische Beziehungen stellt SKOS die beiden Properties *skos:broaderTransitive* bzw. *skos:narrowerTransitive* zur Verfügung, welche auch tatsächlich als Instanzen von *owl:TransitiveProperty* definiert sind. *skos:broader* ist dabei ein Subproperty von *skos:broaderTransitive* und *skos:narrower* ein Subproperty von *skos:narrowerTransitive*. Diese beiden Properties sind eigentlich nicht für die Annotation auf Instanzebene gedacht und werden nur verwendet, um Schlussfolgerungen bezüglich indirekten hierarchischen Beziehungen zu ziehen. Zur Veranschaulichung betrachte man folgendes Beispiel:

```
<Tier>   rdf:type   skos:Concept .

<Saeugetier>
   rdf:type       skos:Concept ;
   skos:broader   <Tier> .

<Tiger>
   rdf:type       skos:Concept ;
   skos:broader   <Saeugetier> .
```

Da *skos:broader* nicht als transitives Property definiert ist, besteht hier keine hierarchische Beziehung zwischen dem Konzept *Tier* und *Tiger*, obwohl in diesem Fall sehr wohl eine vorhanden ist. Ein Reasoner (Inferenzmaschine), welcher diese indirekte hierarchische Beziehung aus den vorhandenen Statements ableiten sollte, verwendet nun anstelle des Properties *skos:broader* das Superproperty *skos:broaderTransitive*:

```
<Tiger>        skos:broaderTransitive   <Saeugetier> .
<Saeugetier>   skos:broaderTransitive   <Tier> .
```

Da *skos:broaderTransitive* als transitives Property definiert ist, ist es für einen Reasoner möglich, das gewünschte Statement abzuleiten:

```
<Tiger>   skos:broaderTransitive   <Tier> .
```

Wie *skos:narrower*, ist auch *skos:narrowerTransitive* als *owl:inverseOf* von *skos:broaderTransitive* definiert, was bedeutet, dass die davor formulierten Statements dasselbe ausdrücken wie folgende Statements:

```
<Tier>         skos:narrowerTransitive   <Saeugetier> .
<Saeugetier>   skos:narrowerTransitive   <Tiger> .
```

Dies würde wiederum implizieren:

```
<Tier>   skos:narrowerTransitive   <Tiger> .
```

Der Zusammenhang zwischen diesen transitiven und nicht transitiven Properties mag auf den ersten Blick verwirrend erscheinen, macht aber durchaus Sinn, weil dies bedeu-

tet, dass man je nach Fall selbst entscheiden kann, ob die Applikation *skos:broader* und *skos:narrower* als transitives oder nicht transitives Property interpretieren soll. Grundsätzlich gilt, dass die Beziehungen nicht transitiv sind, weil eben *skos:broader* und *skos:narrower* auch nicht transitiv sind und die transitiven Properties nicht für die Instanzebene gedacht sind. Für Fälle, in denen Schlussfolgerungen aus indirekten hierarchischen Beziehungen gezogen werden sollen, kann trotzdem *skos:broader* bzw. *skos:narrower* verwendet werden und ein Reasoner wäre mit Hilfe der Properties *skos:broaderTransitive* bzw. *skos:narrowerTransitive* dennoch in der Lage, indirekte Beziehungen zwischen den Konzepten abzuleiten.

Auch die transitiven Properties können in SKOS reflexiv verwendet werden. So sind hierarchische Zyklen wie im folgenden Beispiel erlaubt:

```
<Tiger>        skos:broader   <Saeugetier> .
<Saeugetier>   skos:broader   <Tier> .
<Tier>         skos:broader   <Tiger> .
```

Hier ist der reflexive Gebrauch von *skos:broaderTransitive* nicht offensichtlich. Wenn man allerdings das Property *skos:broader* durch das Property *skos:broaderTransitive* ersetzt und dann den transitiven Schluss zieht, erhält man ein Statement der Form `<Tiger> skos:broaderTransitive <Tiger>.`

IB 4: *skos:related* und *skos:broaderTransitive* sind disjunkt. Da *skos:related* ein symmetrisches Property ist und *skos:narrowerTransitive* die Inverse von *skos:broaderTransitive* ist, ist *skos:related* folglich auch mit *skos:narrowerTransitive* disjunkt. Klar ausgedrückt bedeutet dies, dass zwei Konzepte nicht sowohl in assoziativer als auch in hierarchischer Beziehung zueinander stehen dürfen. Dies bezieht sich sowohl auf unmittelbar als auch auf indirekte hierarchisch verwandte Konzepte.

Demnach sind folgende Statements nicht konform mit dem SKOS-Datenmodell:

```
<Tiger>
   skos:broader   <Saeugetier> ;
   skos:related   <Saeugetier> ;
```

Diese Integritätsbedingung gilt aber auch über mehrere hierarchische Ebenen. Somit wäre auch folgendes Statement nicht valide:

```
<Tiger>
   skos:broader   <Saeugetier> ;
   skos:related   <Tier> .

<Saeugetier>   skos:broader   <Tier> .
```

Aus diesen Statements folgt `<Tiger> skos:broaderTransitive <Tier>`. Da auch `<Tiger> skos:related <Tier>` formuliert ist, verstoßen diese Statements gegen IB 4. Aufgrund der Symmetrie von *skos:related* implizieren diese Statements auch `<Tier> skos:related <Tiger>`. Und da *skos:broaderTransitive* die Inverse von *skos:narrowerTransitive* ist, implizieren diese Statements auch `<Tier> skos:narrowerTransitive <Tiger>`, womit bewiesen ist, dass diese Integritätsbedingung auch für *skos:narrowerTransitive* gilt. Dies bezieht sich natürlich auch hier auf direkte und indirekte verwandte Konzepte der gesamten Hierarchie entlang.

2.4.7 Collections

SKOS Collections sind benannte und/oder geordnete Gruppen von Ressourcen. Ressourcen werden in Collections zusammengefasst, wenn sie aus welchen Gründen auch immer in Zusammenhang stehen und es Sinn macht, sie gemeinsam zu benennen oder zu ordnen. Es sind auch verschachtelte Collections möglich, was bedeutet, dass eine Collection weitere Collections enthalten kann. SKOS stellt zwei verschiedene Arten von Collections zur Verfügung, nämlich *Collections* und *OrderedCollections*. *OrderedCollections* sind geordnete Sammlungen und werden verwendet, wenn die Reihenfolge der Ressourcen für die Semantik der Beschreibung relevant ist. Beide Arten von Collections sind Instanzen der Klasse *owl:Class*. *skos:OrderedCollection* ist dabei eine Subklasse von *skos:Collection*.

Beide Typen von Collections haben ein eigenes Property zur Verfügung, um die Elemente in einer Collection zu definieren. *skos:Collection* verwendet dafür das Property *skos:member*, während *skos:OrderedCollection* das Property *skos:memberList* verwendet. Beide Properties setzen dabei Ressourcen zueinander in Beziehung, somit sind diese Properties Instanzen von *owl:ObjectProperty*. Der Range von *skos:member* ist in der SKOS-Referenz nicht angegeben, was bedeutet, dass eine beliebige Ressource in einer SKOS-Collection enthalten sein kann. Das Property *skos:memberList* hat hingegen einen Range definiert, und zwar die Klasse *rdf:List* (vgl. Kapitel 2.2.6). Auch in einer *rdf:List* können beliebige Ressourcen enthalten sein, also auch alle in SKOS definierten Klassen. *skos:memberList* ist eine Instanz von *owl:FunctionalProperty*, was bedeutet, dass der Wert des Properties nur einen Wert annehmen darf, nämlich jenen der *rdf:List*.

Eine Collection, um alle in einem KOS vorhandenen Tierkonzepte zu gruppieren, würde wie folgt aussehen:

```
<Tiere>
   rdf:type      skos:Collection ;
   skos:member   <Tiger> ;
   skos:member   <Loewe> ;
   skos:member   <Saeugetier> .
```

Dieselbe Sammlung in geordneter Reihenfolge würde stattdessen ganz anders aussehen, nämlich:

```
<Tiere>
   rdf:type          skos:OrderedCollection ;
   skos:memberList   (<Tiger> <Loewe> <Saeugetier>) .
```

Wie man erkennen kann, kommt nur ein Statement mit dem Property *skos:memberList* vor, obwohl drei Ressourcen in der Collection enthalten sind. Im Falle der ungeordneten Collection wird für jede Ressource in der Collection ein eigenes Statement formuliert.

IB 5: *skos:Collection* ist sowohl mit *skos:Concept* als auch mit *skos:ConceptScheme* disjunkt. Da Konzepte und Konzeptschemen auch disjunkt sind, bedeutet dies, dass keine Ressource als eine Instanz zweier verschiedener SKOS-Klassen definiert sein darf.

2.4.8 Mapping Properties

Mapping Properties beschreiben Relationen zwischen Konzepten aus verschiedenen Konzeptschemen. Somit lassen sich bereits vorhandene Konzepte aus fremden Konzeptschemen leicht ins eigene Konzeptschema integrieren.

Das Property *skos:mappingRelation* wird nicht zur Beschreibung von Ressourcen verwendet und stellt lediglich das Superproperty aller anderen Mapping Properties dar. Alle Mapping Properties sind wiederum Instanzen von *owl:ObjectProperty*. Mapping Properties haben, wie die Semantic Relation Properties auch, sowohl als Domain als auch als Range die Klasse *skos:Concept* definiert.

Das Property *skos:exactMatch* drückt aus, dass zwei Konzepte aus verschiedenen Konzeptschemen hinsichtlich ihrer Semantik identisch sind, also dasselbe beschreiben. Dabei sollten diese zumindest so ähnlich sein, dass sie in verschiedenen Applikationen austauschbar verwendet werden könnten. Zum Beispiel könnte in einem Konzeptschema ein Konzept namens *Wolf* und in einem anderen, wissenschaftlichen, Konzeptschema ein Konzept namens *Canis Lupus* vorhanden sein, wobei beide Konzepte bezüglich ihrer Semantik identisch sind. Das Property würde in diesem Fall wie folgt verwendet werden:

```
<Wolf>   skos:exactMatch   <Canis Lupus> .
```

Natürlich ist *skos:exactMatch* auch ein symmetrisches Property, weil wenn ein Konzept dasselbe beschreibt wie ein anderes, dann muss auch das andere dasselbe beschreiben wie das eine. Ein Statement der Form `<A> skos:exactMatch <B>` impliziert also ein Statement der Form `<B> skos:exactMatch <A>`.

Das Property *skos:closeMatch* wurde erst zu einem Zeitpunkt in das SKOS-Vokabular aufgenommen, als diese Arbeit bereits weit fortgeschritten war und wird aus diesem Grund in dieser Arbeit nicht mehr berücksichtigt. (vgl. Kapitel 4.1.2)

Mit den Properties *skos:broadMatch* und *skos:narrowMatch* werden hierarchische Beziehungen zwischen Konzepten aus verschiedenen Konzeptschemen ausgedrückt. Diese werden gleich verwendet wie die gewöhnlichen Properties zur Beschreibung von hierarchischen Beziehungen (vgl. Kapitel 2.4.6). *skos:broadMatch* ist ein Subproperty von *skos:broader* und *skos:narrowMatch* ist ein Subproperty von *skos:narrower*. Wie bei den hierarchischen semantischen Relationen ist auch hier *skos:narrowMatch* die Inverse von *skos:broadMatch*. Da diese Mapping Properties Subproperties der korrespondierenden Semantic Relation Properties und in weiterer Folge auch Subproperties der transitiven Properties sind, können auch diese Properties von einer konkreten Applikation als transitiv oder nicht transitiv interpretiert werden.

Diese Properties dürfen ebenfalls reflexiv verwendet werden, demnach ist ein Statement der Form `<A> skos:broadMatch <A>` im SKOS-Datenmodell erlaubt.

Wie bei den Semantic Relation Properties ist es auch mit *skos:broadMatch* und *skos:narrowMatch* nicht untersagt, hierarchische Zyklen zu bilden. Demnach sind `<A> skos:broadMatch <B>`, `<B> skos:broadMatch <C>` und `<C> skos:broadMatch <A>` valide SKOS-Statements.

skos:relatedMatch wird verwendet, um assoziative Beziehungen zwischen zwei Konzepten aus verschiedenen Konzeptschemen zu modellieren. Auch *skos:relatedMatch* ist ein Subproperty von *skos:related*. Somit ist auch *skos:relatedMatch* eine Instanz von *owl:SymmetricProperty*. Dies bedeutet, ein Statement der Form `<A> skos:relatedMatch <B>` impliziert auch ein Statement der Form `<B> skos:relatedMatch <A>`. Weil alle dieser drei Properties (*skos:broadMatch*, *skos:narrowMatch*, *skos:relatedMatch*) von ihren korrespondierenden Semantic Relation Properties abgeleitet werden, gilt auch für diese Properties die in Kapitel 2.4.6 beschriebene Integritätsbedingung 4. Dies bedeutet, folgende Statements wären in SKOS nicht valide:

```
<Tiger>
   skos:broadMatch     <Saeugetier> ;
   skos:relatedMatch   <Tier> .

<Saeugetier>   skos:broadMatch   <Tier> .
```

Auch *skos:relatedMatch* darf reflexiv verwendet werden, wie zum Beispiel im Statement `<A> skos:relatedMatch <A>`.

Da die Mapping Relation Properties von ihren korrespondierenden Semantic Relation Properties abgeleitet sind, besteht ein direkter Zusammenhang zwischen diesen beiden Gruppen von Properties. Abbildung 4 zeigt eine aktualisierte Version von Abbildung 3, in welcher die Subproperty-Beziehungen zwischen den Semantic Relation- und Mapping Properties zu sehen sind. In Abbildung 5 sind nur die Mapping Properties aufgelistet.

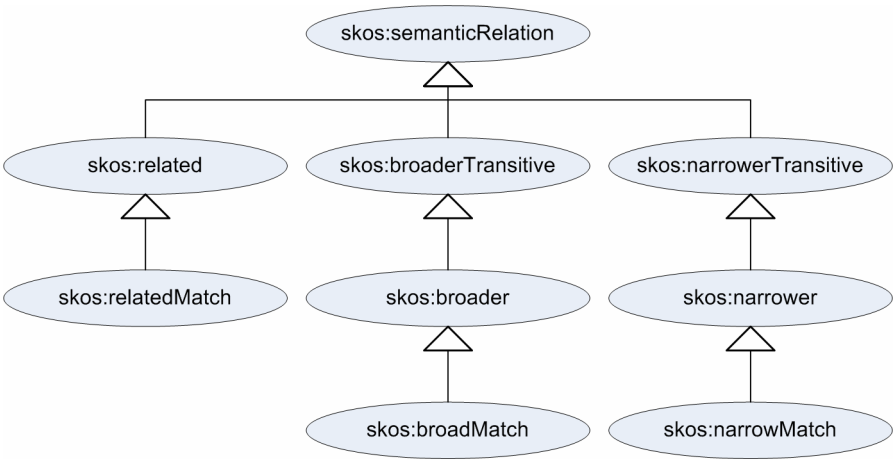

Abb. 4: Die subProperty-Beziehungen zwischen den Semantic Relation- und Mapping Properties in SKOS.

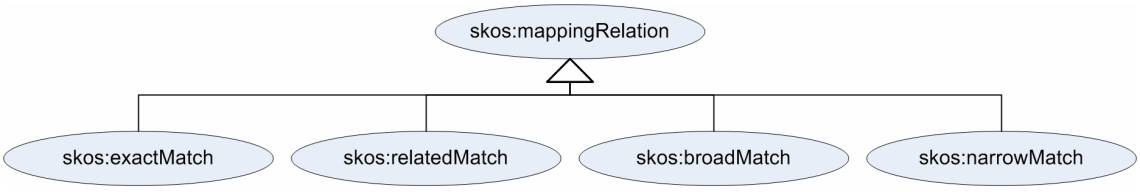

Abb. 5: Übersicht der Mapping Relation Properties in SKOS.

Weiters definiert SKOS keine Integritätsbedingung, dass auch wirklich nur Konzepte aus verschiedenen Konzeptschemen mit diesen Properties miteinander verbunden werden dürfen. Es ist also konsistent, in einem SKOS-Datenmodell zwei Konzepte aus demselben Schema mit einem Mapping Property zueinander in Beziehung zu setzen. Ebenso ist es auch umgekehrt erlaubt, die Semantic Relation Properties zu verwenden,

um Konzepte aus verschiedenen Konzeptschemen miteinander zu verbinden. Dies macht vor allem deswegen Sinn, weil Wissensmodelle oft kontinuierlich verändert werden. So könnte es zum Beispiel auch passieren, dass sich Konzepte erst in einem Konzeptschema befinden, später aber in einem völlig anderen. Würde es eine Integritätsbedingung dafür geben, dass die jeweiligen Typen von Properties richtig verwendet werden müssten, könnte es passieren, dass bereits bestehende Wissensmodelle nicht mehr valide wären, weil zum Beispiel erst ein Semantic Relation Property für die Formulierung einer Beziehung zu einem Konzept im selben Konzeptschema verwendet wurde, das Konzept später aber nicht mehr in diesem, sondern in einem anderen Konzeptschema enthalten ist. Bei der Definition von neuen SKOS-Modellen sollte man sich jedoch unbedingt an die Empfehlung halten, die Properties jeweils richtig zu verwenden.

2.4.9 Class- und Property Extension

Alle in SKOS vordefinierten Klassen und Properties können mit Hilfe der vordefinierten RDF Schema Properties *rdfs:subClassOf* und *rdfs:subPropertyOf* erweitert werden. Somit ist es möglich, eigene Properties bzw. Klassen zu definieren, welche von einem vorhandenen SKOS-Property bzw. von einer vorhandenen SKOS-Klasse abgeleitet werden und mit Hilfe von OWL-Sprachprimitiven weiter spezialisiert werden können. Auf diese Weise können eigene und ausdrucksstärkere Properties und Klassen definiert werden. Dies bedeutet, es ist möglich, das SKOS-Vokabular gemäß den Anforderungen an die eigene Applikation anzupassen.

Nehmen wir als Beispiel an, wir wollten wieder das bereits beschriebene Property namens *liegtIn* zur Beschreibung einer Beziehung zwischen zwei Regionen definieren, wobei eine Region innerhalb einer anderen Region liegen kann. Diese Beziehung zwischen zwei Regionen ist bezüglich ihrer Semantik transitiv, allerdings ist das in SKOS definierte Property zur Beschreibung von assoziativen Relationen *skos:related* nicht transitiv definiert. So könnte man ein eigenes, tatsächlich transitives Property erstellen:

```
<liegtIn>   rdf:type             owl:ObjectProperty ;
            rdfs:subPropertyOf   skos:related ;
            rdf:type             owl:TransitiveProperty .
```

Wie man hier sieht, wird ein Property definiert, welches von *skos:related* abgeleitet und explizit als transitives Property definiert wird. Somit hat man ein ausdrucksstärkeres Property definiert, welches die Eigenschaften von *skos:related* übernimmt und zudem noch ein transitives Property darstellt.

2.5 Semantische Wikis

2.5.1 Einführung

Traditionelle Wikis sind webbasierte Anwendungen, welche ein einfaches und kollaboratives Erstellen von Webseiten ermöglichen. Das Besondere an Wikis ist, dass jede Person mit einem Zugang zum WWW die Möglichkeit hat, aktiv an der Erstellung von Webseiten mitzuwirken. Wikis verwenden eine eigene, wenig komplexe Markup-Sprache zur Erstellung ihrer Seiten. Diese Webseiten können auch einfach miteinander verlinkt werden. Wikis unterstützen außerdem die Versionierung von Artikeln, indem Änderungen am Inhalt eines Artikels festgehalten werden. Weiters ist es möglich, textbasiert nach Artikeln im Wiki zu suchen.

Allerdings ist die Suche nach einer bestimmten Information oft recht schwierig. In den Artikeln ist Wissen zumeist in unstrukturierter Form enthalten, das eigentlich leicht zu strukturieren wäre, wie z.B. bestimmte Daten über Persönlichkeiten, wie Geburtstag und Geburtsort, vollständiger Name usw. In traditionellen Wikis existieren zwar manuell angelegte Seiten für eine bestimmte Art von Information, jedoch müssen diese auch manuell gewartet und aktualisiert werden, sobald eine neue Persönlichkeit in einem Artikel beschrieben wird. So ist es zum Beispiel dennoch möglich, sich alle Persönlichkeiten auflisten zu lassen, welche an einem bestimmten Tag Geburtstag haben. Allerdings lässt sich diese Auflistung bezüglich der Typen nicht weiter spezifizieren, sodass nur alle Persönlichkeiten mit einem bestimmten Geburtstag aufgelistet werden können, nicht aber nur Schauspieler. Außerdem existieren diese manuell angelegten Seiten nicht für jede erdenkliche Art von Information. So lässt sich zum Beispiel nur schwer nach allen Persönlichkeiten suchen, deren Körpergröße über 2 Meter beträgt oder deren Muttersprache Spanisch ist.

Semantische Wikis [28] erweitern traditionelle Wikis in dem Sinn, dass strukturierte Information explizit in Form von maschinenverständlichen Metadaten angegeben werden kann. Dies bedeutet, die Artikel in einem Wiki selbst und die Links zwischen diesen Artikeln können mit semantischen Annotationen versehen werden. Das Vokabular zur semantischen Annotation liegt dabei einer definierten Ontologie zugrunde, welche die zur Typisierung von Artikeln und Links zwischen den Artikeln verwendeten Klassen und Properties definiert. In vielen semantischen Wikis haben die Benutzer nicht nur die Möglichkeit, eventuell bereits im Wiki vordefinierte Ontologien zu verwenden, son-

dern auch selbst eigene Ontologien zu erstellen, welche dann in weiterer Folge im Wiki selbst für die semantische Beschreibung von Artikeln verwendet werden können. In diesem Fall profitiert also auch das semantische Wiki selbst von den darin entwickelten Ontologien, weil auf diese Weise vor allem die Effizienz der Suche nach Information in einem Wiki erheblich gesteigert wird, weil nicht nur nach in Artikeln vorkommende Zeichenketten gesucht werden kann, sondern eben auch nach bestimmten Typen von Artikeln oder speziellen Werten von Eigenschaften eines bestimmten Typs, wie zum Beispiel nach allen Schauspielern, welche am Neujahrstag Geburtstag haben.

2.5.2 Ylvi und METIS

Ylvi [26] ist ein konkretes und bereits existierendes semantisches und multimediales Wiki. Es erweitert die Funktionalität traditioneller Wikis um die Möglichkeit der Typisierung von Artikeln, Links zwischen Artikeln und Medienobjekten und ermöglicht somit nicht nur eine textbasierte, sondern auch eine semantische Suche nach Inhalt.

Ylvi basiert auf METIS [24], einem äußerst flexiblen und erweiterbaren Multimedia-Management-Framework zur flexiblen Erstellung von Multimediaanwendungen jeglicher Art. METIS kann durch sogenannte *Semantic Packs* erweitert werden, um eigene Datenmodelle erstellen zu können. So ist es zum Beispiel möglich, standardisierte Vokabulare zur Beschreibung von Ressourcen in METIS zu importieren. Dies ist auch ein Teil dieser Arbeit, nämlich die in der SKOS-Ontologie definierten Klassen und Properties in Form eines *Semantic Packs* in Ylvi zur Verfügung zu stellen, sodass die Artikel in Ylvi mit diesen Klassen und Properties annotiert werden können.

Weiters kann die Funktionalität von METIS durch sogenannte Plugins erweitert werden. Diese Plugins können auch innerhalb von Ylvi ausgeführt werden, was bedeutet, dass auch die Funktionalität von Ylvi durch Plugins erweitert werden kann. Diese Plugins können also sozusagen als Schnittstelle zwischen METIS und Ylvi verwendet werden. Auch im Rahmen dieser Arbeit wurden Plugins entwickelt, welche die Funktionalität von Ylvi erweitern (vgl. Abbildung 1, Kapitel 1.2).

Ylvi verwendet das Datenmodell von METIS zur Speicherung der verschiedenen Objekte innerhalb des semantischen Wikis. Abbildung 6 zeigt einen Überblick der einzelnen Komponenten innerhalb des Datenmodells und ihrer Beziehungen zueinander.

Einfache, also elementare und nicht zusammengesetzte Medienobjekte, werden mit Hilfe von sogenannten *SingleMediaObjects (SMO)* repräsentiert. Ein *SingleMediaObject*

stellt eine abstrakte und logische Repräsentation eines konkreten Medienobjekts dar. Konkrete Mediendateien werden dem *SMO* mit Hilfe von Objekten der Klasse *MediaInstance* (Medieninstanz) zugewiesen. Diese Medieninstanzen sind mit den tatsächlichen Daten des Medienobjekts über so genannte *MediaLocator*s verbunden. Die tatsächliche Quelle des Medienobjekts kann dabei das Dateisystem, ein Webserver oder eine Datenbank sein, denn mit Hilfe eines Objekts der Klasse *MediaLocator* können alle diese Datenquellen einheitlich angesprochen werden.

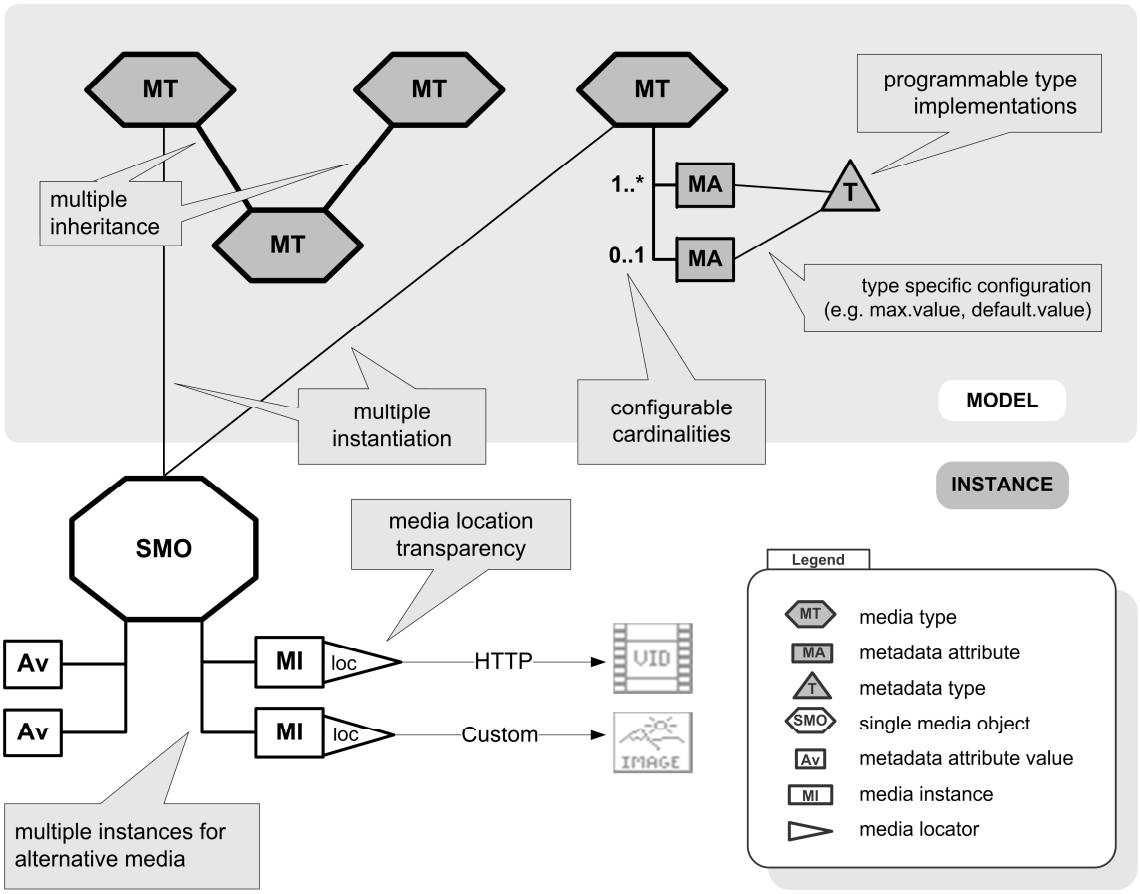

Abb. 6: Die Komponenten des METIS-Datenmodells und ihre Beziehungen zueinander.

Quelle: [24]

ComplexMediaObjects (CMO) unterscheiden sich nicht wesentlich von *SMOs*, außer dass sie zusammengesetzte Medienobjekte sind, also zum Beispiel ein Video mit einem Audio. *ComplexMediaObjects* können *SMOs* und auch andere *CMOs* enthalten

Um Medienobjekte semantisch klassifizieren zu können, müssen erst verschiedene Kategorien von *MediaTypes* (Medientypen) vorhanden sein. *MediaTypes* können weiter spezialisiert werden, sodass sich hierarchische Strukturen innerhalb der Medientypen modellieren lassen. Dabei ist auch eine Mehrfachvererbung möglich, was bedeutet, dass

ein Medientyp von mehreren anderen Medientypen abgeleitet werden kann, sodass der abgeleitete Typ einen spezielleren Typ von mehr als einem Medientypen darstellt. Ebenso können Medienobjekte mit mehreren *MediaTypes* typisiert werden Dies ist wichtig, damit die Aussagekraft des Modells erhalten bleibt, weil eine konkrete Ressource auch in der Realität zwei verschiedene Typen einer Ressource repräsentieren könnte. Um eine Analogie zum realen Leben herzustellen, könnte man sich vorstellen, dass ein konkreter Mensch als Europäer und als Schauspieler klassifiziert werden soll, weil es sich bei dem konkreten Menschen um einen europäischen Schauspieler handelt und sowohl der Medientyp *Schauspieler* als auch der Medientyp *Europäer* vorhanden sind. Medientypen sind also nicht wirklich auf Medien beschränkt, sondern können alle möglichen Dinge repräsentieren, welche auch in der Realität beschrieben werden können.

MediaTypes können weiters in sogenannte *MediaTypeCategories* (Medientypkategorien) eingeordnet und zusammengefasst werden, wenn sie aus irgendwelchen Gründen inhaltlich in Zusammenhang stehen oder gruppiert werden sollten.

Medientypen können *MetadataAttributes* (Metadatenattribute) besitzen, welche die Metadaten einer Medieninstanz des jeweiligen Typs mit konkreten Werten (*MetadataAttributeValues*) beschreiben. Metadatenattribute sind eigenständige Objekte und sind deshalb nicht an einen bestimmten Medientypen gebunden. Sie können daher mehreren Medientypen zugewiesen werden und sind auch in abgeleiteten Medientypen verwendbar. Sie werden also an speziellere Medientypen der Hierarchie entlang vererbt. Auch die Kardinalität von *MetadataAttributes* kann vorgeschrieben werden, welche ausdrückt, wie viele Werte ein bestimmter Medientyp für ein bestimmtes Attribut haben darf bzw. muss. Metadatenattribute sind ebenfalls typisiert, was bedeutet, dass ihr Wert eine Instanz eines bestimmten Metadatentyps (*MetadataTypes*) ist. Dies können einfache Datentypen, wie zum Beispiel *Integer* oder *String*, aber auch komplexere, selbst definierte Typen sein.

In METIS ist es auch möglich, Beziehungen zwischen Medienobjekten zu definieren. Diese Beziehungen werden mit Hilfe von binären und gerichteten Assoziationen (*Associations*) dargestellt, deren Semantik in sogenannten Assoziationstypen (*AssocTypes*) definiert ist. Assoziationstypen können ebenfalls, wie Medientypen, vererbt und hierarchisch organisiert werden. Dabei erben die spezielleren Assoziationstypen die Eigenschaften der allgemeineren Assoziationstypen.

Wie bereits erwähnt, verwendet das semantische Wiki Ylvi das Datenmodell von METIS, was bedeutet, dass es auch seinen flexiblen Charakter und dessen Eigenschaften nützen kann. Abbildung 7 zeigt eine Übersicht der Eigenschaften und Funktionen von Ylvi.

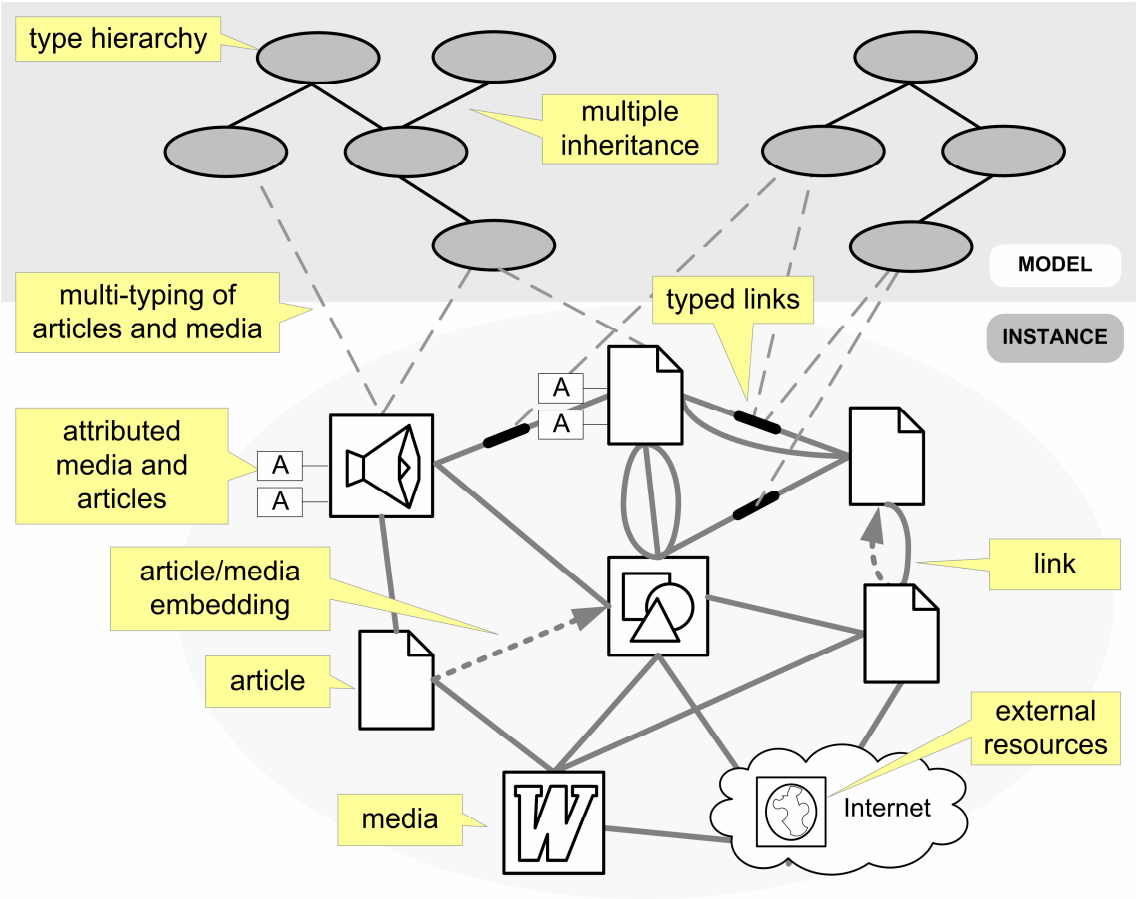

Abb. 7: Die wesentlichen Eigenschaften und Funktionen des semantischen Wikis Ylvi.

Quelle: [26]

Folgend werden die Eigenschaften von Ylvi zusammengefasst:

- *Typhierarchie*: Wie bereits beschrieben, können Medientypen hierarchisch organisiert werden, wobei die Eigenschaften von den allgemeineren Typen an die spezielleren Typen vererbt werden.
- *Multi-Typing*: Artikel in Ylvi können Instanzen von mehreren verschiedenen Typen darstellen.
- *Attribute von Typen*: Attribute können Medientypen zugewiesen werden. In Artikeln eines bestimmten Typs können in weiterer Folge diese Attribute verwendet werden, um den jeweiligen Artikel zu beschreiben.

- *Typisierte Links:* Die Links zwischen Artikeln sind typisiert und verwenden ein Vokabular aus einem in einer konkreten Ylvi-Instanz definierten *Semantic Pack*. Dies erhöht die Möglichkeiten der semantischen Suche, weil auch nach Linktypen gesucht werden kann. Es ist auch möglich, externe Ressourcen mit typisierten Links zu referenzieren.

Ylvi verwendet eine eigene Markup-Sprache zur Annotation von Typen, Attributen und Links. In Abbildung 8 sind der Quellcode eines Artikels und der korrespondierende Text als angezeigter Artikel im Wiki zu sehen.

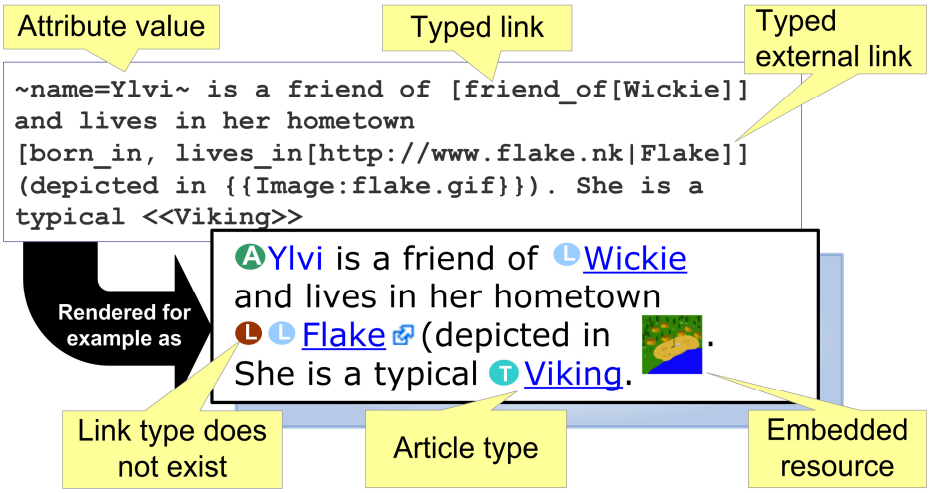

Abb. 8: Der Quellcode eines Artikels in Ylvi und der daraus resultierende angezeigte Text des Artikels im Wiki.

Quelle: [26]

Wie man an dieser Abbildung erkennen kann, wird der visualisierte Text eines Artikels von den semantischen Annotationen kaum beeinflusst. Auf diese Weise ist die informale Beschreibung eines Artikels immer noch sehr gut lesbar, was sehr wichtig ist, damit die intendierte Semantik eines Artikels auch von einem menschlichen Benutzer verstanden werden kann.

Typen werden von doppelt angeführten und eckigen Spitzklammern umschlossen, also `<<Typ>>`. In diesem konkreten Beispiel wird die durch den Artikel beschriebene Ressource, nämlich Wickie's weiblicher Freund namens Ylvi aus der bekannten Zeichentrickserie, als `<<Viking>>` (Wikinger) typisiert. Dafür muss ein *MediaType* namens *Viking* bereits definiert sein, für welchen auch Attribute und Relationen definiert sein können. Mit diesem *MediaType* kann eine konkrete Instanz eines Wikingers, wie zum Beispiel Ylvi, beschrieben werden. Wikinger haben für gewöhnlich, wie alle anderen

Menschen auch, einen Namen und können auch mit anderen Menschen, vermutlich ebenso Wikingern, befreundet sein. Dafür könnte man für den Medientyp *Viking* ein Attribut namens *name* und eine Assoziation namens *friend_of* definieren. Dieses Attribut und diese Assoziation werden auch, wie in Abbildung 8 ersichtlich, zur Beschreibung von Ylvi verwendet. Attribute werden von zwei ~ eingeschlossen. Innerhalb dieser Zeichen folgt erst der Attributname und danach der Wert des Attributs, also `~Attributname=Attributwert~`. In diesem Beispiel wäre dies `~name=Ylvi~`. Assoziationen, auch als Links bezeichnet, werden in Ylvi mit eckigen Klammern annotiert, wobei nach der ersten Klammer die Assoziationstypen folgen und danach in einer weiteren, verschachtelten Klammer das Ziel des Links angegeben wird, also `[Linktyp [Linkziel]]`. In diesem konkreten Beispiel wäre dies also `[friend_of [Wickie]]`.

Im Hinblick auf das Ziel dieser Arbeit, nämlich die Integration des SKOS-Metamodells in Ylvi, bedeutet dies, dass ein Metamodell geschaffen werden muss. Das heißt, es müssen Medientypen für die in SKOS definierten Klassen (Konzepte, Konzeptschemen, Collections), Attribute für die Beschreibung von Instanzen dieser Klassen und Assoziationstypen für die Modellierung von semantischen Beziehungen zwischen Instanzen dieser Klassen definiert werden.

Artikel in Ylvi werden als *ComplexMediaObjects* dargestellt, weil diese mehrere verschiedene Arten von Medien beinhalten können. *CMOs* können als ein bestimmter Medientyp klassifiziert werden, was bedeutet, dass auch Artikel klassifiziert werden können. Dies bedeutet, dass Artikel beispielsweise als Medientyp *skos:Concept* typisiert werden können und in diesen Artikeln auch typisierte Links aus dem SKOS-Vokabular zu anderen Artikeln definiert werden können.

Zum Beispiel könnte ein Artikel namens *Auto* erstellt werden, welcher als *Concept* typisiert wird. Dieses Konzept steht für kein konkretes Auto, sondern für den allgemeinen Begriff *Auto*. Dabei können auch Beziehungen zu anderen Artikeln definiert werden, zum Beispiel eine hierarchische Beziehung zum Artikel *Fahrzeug*, welcher ebenso als *Concept* typisiert wird. Diese Beziehung sagt aus, dass *Auto* ein Subkonzept von *Fahrzeug* ist, also jede Instanz eines Autos auch ein Fahrzeug ist. Damit lässt sich mit Hilfe des im *Semantic Pack* definierten Metamodells ein Wissensmodell beschreiben, dass dann gespeichert werden kann. Bei dieser Speicherung werden zum Beispiel alle als *Concept* typisierten Artikel in Medientypen umgewandelt. Als Resultat können Artikel in weiterer Folge als diese neu erstellten Typen klassifiziert werden, zum Beispiel,

wenn ein Artikel für ein konkretes Auto erstellt wird, wie für den Audi A4. METIS speichert auch die hierarchischen Beziehungen zwischen Medientypen und aufgrund der im Konzept *Auto* definierten hierarchischen Beziehung zum Konzept *Fahrzeug*, ist Ylvi in der Lage, die Information abzuleiten, dass der Audi A4 ein Fahrzeug ist. Sucht man nun zum Beispiel nach allen Instanzen des Typs *Fahrzeug,* würden alle Artikel aufgelistet werden, welche als *Fahrzeug*, als *Auto*, aber auch als *Motorrad* oder *Fahrrad* typisiert wurden, weil alle diese Konzepte Subkonzepte von *Fahrzeug* darstellen.

2.6 Ontologieentwicklung

2.6.1 Einführung

Unter Ontologieentwicklung (Ontology Engineering) [22, 23] versteht man sämtliche Aktivitäten, welche den Entwicklungsprozess und den Lebenszyklus einer Ontologie betreffen sowie die während der Ontologieentwicklung eingesetzten Methoden, Werkzeuge und Sprachen.

Grundsätzlich unterscheidet man zwischen drei verschiedenen Arten von Ontologien:

- *Representation Ontologies*: Diese Ontologien definieren Sprachprimitive, welche zur Erstellung von Ontologien verwendet werden können. Sie stellen also sozusagen Meta-Ontologien dar. OWL ist zum Beispiel eine *Representation Ontology*, weniger streng betrachtet ebenso das in Ylvi zur Verfügung gestellte Metamodell zur Beschreibung von SKOS-Ressourcen.
- *General* oder *upper-level ontologies*: Diese Ontologien beinhalten sehr allgemeine Begriffe, also keine tief spezialisierten Konzepte. Deswegen sind diese Ontologien auch von einem bestimmten Problem oder einem bestimmten Wissensbereich unabhängig und können somit in den verschiedensten Applikationen oder Wissensbereichen verwendet werden.
- *Domain Ontologies*: Dies sind spezifische Ontologien, welche einen bestimmten Wissensbereich beschreiben. Sie sind in den meisten Fällen an eine bestimmte Applikation angepasst und aus diesem Grund auch nicht in verschiedenen Applikationen, also universal, verwendbar. Sie beschreiben also ein subjektives Vokabular aus einem bestimmten Bereich für eine bestimmte Applikation, wie zum Beispiel ein Vokabular zur Beschreibung von Autos, welches von einem

konkreten Autohersteller selbst definiert wird und seinen spezifischen Bedürfnissen zugeschnitten ist. Dieses Vokabular definiert die Konzepte des jeweiligen Wissensbereichs und deren Beziehungen zueinander.

Die Entwicklung einer Ontologie kann in fünf Phasen eingeteilt werden. Jede dieser fünf Phasen besteht aus mehreren Aktivitäten.

1. *Spezifikation*: Hier werden der Zweck (warum soll diese Ontologie entwickelt werden?) und Anwendungsbereich (wie und von wem soll diese Ontologie verwendet werden?) der zu entwickelnden Ontologie identifiziert.
2. *Konzeptualisierung*: Hier soll die zu erstellende Ontologie in einem konzeptualisierten Modell beschrieben werden, sodass sie den in der ersten Phase definierten Anforderungen entspricht.
3. *Formalisierung*: Hier wird das in der vorhergehenden Phase beschriebene konzeptualisierte Modell in eine formale Form transformiert. allerdings noch nicht in die endgültige Form.
4. *Implementation*: Das formale Modell aus Phase 3 wird in eine Ontologiebeschreibungssprache übersetzt.
5. *Wartung*: Die Ontologie wird auf einem aktuellen Stand gehalten, kontinuierlich verfeinert und deren fehlerhaften Definitionen werden korrigiert.

Die Entwicklung einer Ontologie entspricht dabei einem „evolving prototype life cycle“-Modell, was bedeutet, dass auf jede beliebige Phase eine beliebige andere Phase folgen kann. Solange die Evaluation kein gewünschtes Resultat bringt oder die in der Spezifikation definierten Anforderungen nicht erfüllt sind, wird der Prototyp kontinuierlich verbessert.

Es existieren noch andere Aktivitäten, welche während allen beschriebenen Phasen, also während des gesamten Lebenszyklus einer Ontologie, ausgeführt werden:

- *Wissensaneignung*: Wissen des jeweiligen Wissensbereichs kann zum Beispiel durch Interviews mit Fachexperten oder durch Lektüre von Fachliteratur erworben werden.
- *Evaluierung:* Bewertung der Qualität einer Ontologie.

- *Dokumentation:* Dokumentation der in einer Ontologie enthaltenen Konzepte und Begriffe ist wichtig, um die Wiederverwendung und Wartung zu erleichtern. Außerdem sollte die Ontologie auch in einer menschenverständlichen Form beschrieben sein, damit die Semantik dieser Ontologie auch für Menschen verständlich ist.

Es existieren einige verschiedene bekannte Methoden zur Entwicklung von Ontologien, welche jeweils für bestimmte Typen von Ontologien besser oder schlechter geeignet sind. Dies bedeutet, es existiert keine Methode, die in allen Fällen am besten geeignet ist. Es muss jeweils im konkreten Fall entschieden werden, welche Methode für die Entwicklung einer bestimmten Ontologie am geeignetsten ist. Eine Ontologie kann entweder völlig neu entwickelt werden oder aus anderen Ontologien zusammengesetzt sein. Die bekanntesten und gängigsten Methoden zur Entwicklung von Ontologien sind:

- TOVE Methode [10]
- ENTERPRISE Methode [32]
- METHONTOLOGY [9]

Bei der Wiederverwendung von Ontologien wird zwischen Fusion und Komposition unterschieden. Fusion bedeutet, dass eine Ontologie eines bestimmten Wissensbereichs mindestens aus zwei Ontologien aus demselben Wissensbereich zusammengesetzt ist. Komposition bedeutet hingegen, dass eine Ontologie aus zwei oder mehreren Ontologien aus einem anderen Wissensbereich zusammengesetzt ist.

2.6.2 Kollaborative Ontologieentwicklung

Der wesentliche Unterschied zwischen traditioneller und kollaborativer Ontologieentwicklung [11, 25, 29] besteht in der Unterstützung des kollaborativen Prozesses der Ontologieentwicklung durch die zugrunde liegende Infrastruktur. Im Falle der traditionellen Ontologieentwicklung werden Ontologien von einem kleinen, aus Knowledge Engineers und Fachexperten bestehenden Team für eine große Anzahl von potentiellen Benutzern erstellt. Die Ontologien werden zumeist zu einem bestimmten Zeitpunkt erstellt, und zwar dann, wenn eine Zusammenkunft der beiden Expertengruppen stattfindet. Die Ontologieentwicklung ist also ein inhärent kollaborativer Prozess, weil die Zusammenarbeit verschiedener Expertengruppen notwendig ist. Dieser kollaborative Charakter der Ontologieentwicklung wird von der traditionellen Vorgehensweise nicht aus-

reichend unterstützt, weil die dafür verwendeten Werkzeuge zumeist über keine verteilte Infrastruktur verfügen. Somit ist ein hoher Kommunikationsaufwand mit der Entwicklung einer Ontologie verbunden, was bedeutet, dass die Ontologieentwicklung sowohl zeitlichen als auch lokalen Einschränkungen unterworfen ist. Wie schon in Kapitel 2.6 beschrieben, ist der Prozess der Ontologieentwicklung aber auch ein sehr dynamischer Prozess, der eine kontinuierliche Verbesserung und Wartung der Ontologien erfordert. Wegen den eben genannten lokalen und zeitlichen Einschränkungen, wird daher auch dieser dynamische Charakter der Ontologieentwicklung vom traditionellen Ansatz in einem zu geringen Ausmaß unterstützt. Die dadurch entstehenden Probleme wurden bereits in Kapitel 1.1 ausführlich beschrieben.

Bezüglich der Phasen der Ontologieentwicklung unterscheiden sich die traditionelle und kollaborative Ontologieentwicklung eigentlich kaum, jede Phase und deren Aktivitäten sind auch bei der kollaborativen Vorgehensweise vorhanden. Auch bei der kollaborativen Entwicklung von Ontologien kann einer beliebigen Phase jede andere beliebige Phase folgen, wobei aber erwähnt werden sollte, dass im kollaborativen Fall zu erwarten wäre, dass die zeitlichen Phasen nicht mehr so scharf voneinander getrennt sind und sich eher überlappen. Wenn die Anzahl der in den Entwicklungsprozess involvierten Personen hoch ist, ist weiters davon auszugehen, dass die verschiedenen Phasen nicht sequentiell, sondern parallel ausgeführt werden, weil verschiedene Benutzer zur gleichen Zeit Aktivitäten aus verschiedenen Phasen der Ontologieentwicklung ausführen.

Als Beispiel einer Infrastruktur zur Entwicklung von Ontologien betrachte man, wie in dieser Arbeit, ein semantisches Wiki. Die Spezifikation einer Ontologie könnte auf jenen Seiten veröffentlicht werden, welche in einem Wiki für die Diskussion eines bestimmten Artikels zur Verfügung stehen. Die Konzeptualisierung übernehmen in diesem Fall die Benutzer, welche die Semantik eines Konzepts innerhalb eines Artikels informal beschreiben. Die Formalisierung erfolgt danach durch die Knowledge Engineers, welche den informalen Text mit den semantischen Annotationen erweitern, sodass die Semantik eines Konzepts auch formal beschrieben ist. Die Implementation der Ontologie wird vom System selbst übernommen, welches das in den Artikeln formal definierte Wissen in eine standardisierte Ontologiebeschreibungssprache exportiert. Aufgrund der zeitlich und örtlich unabhängigen Infrastruktur eines Wikis ist die Wartung einer Ontologie praktisch jederzeit möglich, was bedeutet, dass der Prozess zur kontinuierlichen Verfeinerung einer Ontologie wesentlich beschleunigt werden sollte. Dies hat wiederum eine Beschleunigung des gesamten Prozesses zur Ontologieentwick-

lung zur Folge, was zu einer höheren Anzahl verfügbarer Ontologien führen könnte. Wikis eignen sich auch sehr gut zur Dokumentation von Ontologien, weil die Elemente einer Ontologie mit Hilfe des informalen Textes in einem Artikel selbst dokumentiert sind. Die Ontologien werden auch laufend evaluiert, weil diese innerhalb des Wikis zur Beschreibung von weiteren Artikeln verwendet werden können und somit Schwachpunkte einer Ontologie sofort aufgedeckt werden.

Der kollaborative Ansatz zur Entwicklung von Ontologien unterstützt also den dynamischen Charakter des Prozesses zur Ontologieentwicklung wesentlich besser als die traditionelle Methode, weil dies ein allgegenwärtiger Prozess ist, welcher ständig aktiv ist, sofern auch die Community im Wiki lebendig ist.

Auf diese Weise könnte die Ontologieentwicklung einer breiten Masse zugänglich gemacht werden, was bedeutet, dass Entwickler und Benutzer von Ontologien nicht mehr unbedingt disjunkte Mengen darstellen müssen, wie es in der traditionellen Vorgehensweise zumeist der Fall ist. In einem Wiki hat jeder Benutzer die Möglichkeit, an der Ontologieentwicklung mitzuwirken, was bedeutet, dass die Benutzer der Ontologien auch Einfluss auf die Evolution einer Ontologie nehmen können und somit ihre eigenen Anforderungen und Wünsche in die Ontologieentwicklung mit einfließen lassen können. Dies bedeutet, es kommt weniger häufig zu Diskrepanzen zwischen der intendierten Semantik der Entwickler einer Ontologie und deren Benutzer, was bedeutet, dass die Ontologien für die Benutzer brauchbarer werden.

Semantische Wikis unterstützen diesen kollaborativen und dynamischen Charakter der Ontologieentwicklung, weil sie über eine verteilte Infrastruktur verfügen, deren Nutzung weder zeitlichen noch örtlichen Einschränkungen ausgesetzt ist. Die jüngste Vergangenheit zeigt auch, dass eine Tendenz zur kollaborativen Ontologieentwicklung auf Basis von semantischen Wikis zu beobachten ist. Einige Projekte dieser Art werden im nächsten Kapitel vorgestellt.

3 Ähnliche Projekte

Da SKOS noch keine Recommendation (Empfehlung) des W3Cs ist und daher immer wieder Änderungen unterworfen ist, existieren erst sehr wenige SKOS-Editoren. Einer davon wurde von Simon Jupp mit der Unterstützung von Sealife Project[11] und CO-ODE[12] entwickelt. Dieser Editor[13] ist ein Plugin für Protégé[14], eine Open Source Applikation zum Editieren von Ontologien. Allerdings ist Protégé 4.0 zum aktuellen Zeitpunkt wieder in den Beta-Status zurückgekehrt. Weil dieses Plugin für SKOS für Protégé 4.0 entwickelt wurde, ist anzunehmen, dass die in Vielzahl aufgetretenen Fehler beim Testen des Editors mit der aktuellen Beta-Version von Protégé 4.0 zusammenhängen. Wie auch immer, dieser Editor fördert wohl eher weniger den kollaborativen Charakter der Ontologieentwicklung, weil die Anwendung über keine verteilte Infrastruktur verfügt.

Der ThManager[15] [17] ist ein Open Source Tool zum Erstellen, Editieren und Visualisieren von SKOS-Beschreibungen. Weiters ist es möglich, SKOS-Beschreibungen in den Editor zu importieren bzw. im Editor erstellte Beschreibungen in SKOS zu exportieren. Dieser Editor unterscheidet sich jedoch von dem im Rahmen dieser Arbeit implementierten System, da der ThManager ebenfalls über keine verteilte Infrastruktur verfügt. Außerdem handelt es sich dabei um einen reinen Ontologieeditor und nicht um einen in einem semantischen Wiki verpackten Ontologieeditor, wie es in dieser Arbeit der Fall ist.

Das W3C stellt ein Online-Tool zur Validierung von SKOS-Beschreibungen[16] zur Verfügung. Die Gemeinsamkeit mit dem in dieser Arbeit vorgestellten System besteht darin, dass auch in dieser Arbeit eine Validierung der Artikel vorgenommen wird und entsprechende Fehlermeldungen bei vorliegenden Inkonsistenzen ausgegeben werden. Ansonsten existieren allerdings keine Gemeinsamkeiten, weil sich dieses Tool zur Validie-

---

11 http://www.biotec.tu-dresden.de/sealife/
12 http://www.co-ode.org/
13 http://code.google.com/p/skoseditor/
14 http://protege.stanford.edu/
15 http://thmanager.sourceforge.net/
16 http://www.w3.org/2004/02/skos/validation

rung nicht zur Erstellung von Ontologien bzw. SKOS-Beschreibungen eignet. Allerdings kann dieses Tool sehr gut zum Testen von vorhandenen SKOS-Modellen eingesetzt werden. Es wurde zum Beispiel auch dafür verwendet, um zu überprüfen, ob der im Rahmen dieser Arbeit implementierte SKOS-Editor auch tatsächlich valide SKOS-Beschreibungen exportiert.

Kollaborative SKOS-Editoren scheinen offensichtlich noch gar nicht zu existieren, was einen weiteren Anreiz zur Durchführung dieser Arbeit dargestellt hat.

Aber in der jüngsten Vergangenheit ist im Bereich des Semantischen Webs viel Zeit in die Erforschung von alternativen Möglichkeiten zur Ontologieentwicklung investiert worden und semantische Wikis spielen dabei aufgrund ihres kollaborativen und leicht zugänglichen Charakters eine große Rolle. Es existieren einige ähnliche Projekte, welche semantische Wikis als Plattformen zur kollaborativen Entwicklung von Ontologien nutzen.

Wiki@nt [2] ist ein Editor zur Erstellung von komplexen und logisch konsistenten Ontologien. Das Metamodell zur Erstellung von Ontologien ist sehr formal definiert, ähnlich dem Metamodell von OWL. Die Anwendung wird auch von einem Wiki unterstützt, allerdings werden in den Artikeln nur Axiome in einer eigenen Markup-Sprache gespeichert. Außerdem ist es in Wiki@nt nicht möglich, Ressourcen mit den erstellten Ontologien auf Instanzebene zu beschreiben. Dies bedeutet, die entwickelten Ontologien können nicht innerhalb des Systems verwendet werden und es existieren auch keine Instanzdaten, welche eine kontinuierliche Verfeinerung und Anpassung einer Ontologie erleichtern und beschleunigen würden, weil die Anforderungen an und die Schwächen von bereits existierenden Ontologien mit Hilfe von verfügbaren Instanzdaten sofort sichtbar wären. Wiki@nt unterstützt also nicht die Verwaltung von Modell und den Instanzdaten im selben System, wie es in Ylvi jedoch der Fall ist. Der Editor scheint bezüglich Wissensmodellierung auf eine fortgeschrittene Zielgruppe ausgerichtet zu sein und ist für unerfahrene Benutzer wohl zu komplex, was bedeutet, dass viele potentielle Nutzer von Ontologien keine Möglichkeit haben, ihre eigenen Anforderungen und Ideen mit einzubringen.

myOntology[17] [30] hingegen nutzt die Infrastruktur eines Wikis tatsächlich zur kollaborativen Erstellung von Lightweight-Ontologien. Dies bedeutet, die Klassen und Proper-

[17] http://www.myontology.org/

ties einer Ontologie werden tatsächlich in den Artikeln des Wikis definiert. Dies ist also ein ähnlicher Ansatz wie jener in dieser Arbeit, unterscheidet sich aber doch wesentlich in zwei Punkten. Zum einen ist myOntology ebenfalls ein reiner Ontologieeditor, welcher nicht das Vermischen von formal und informal beschriebenem Inhalt erlaubt. myOntology versucht stattdessen, die informale Semantik einer Ressource mit Hilfe von Referenzen auf Multimediadateien, welche die jeweilige Ressource beschreiben, zu vermitteln. In Ylvi besteht jedoch die Möglichkeit, bereits im Wiki existierendes und durch Text informal beschriebenes Wissen mit Hilfe von semantischen Annotationen zu erweitern, sodass dieses Wissen auch formal beschrieben ist. Zum anderen gibt es in myOntology, wie auch in Wiki@nt, keine Möglichkeit, die entwickelten Ontologien auf Instanzebene zu verwenden, sodass auch hier keine Instanzdaten zur Verfügung stehen. Die Zielgruppe lässt sich auch hier eher auf fortgeschrittene Knowledge Engineers eingrenzen.

OntoWiki[18] [1] ist ebenfalls ein webbasiertes Tool zum kollaborativen Erstellen von Wissensmodellen. Es speichert durchgeführte Änderungen an Ressourcenbeschreibungen und unterstützt somit Einblick in die Evolution eines Wissensmodells. Dabei kann man sich auch an Diskussionen zur Entwicklung eines Modells beteiligen oder vorhandene Diskussionen einsehen, womit auch der Diskurs, der zu einer Wissensrepräsentation geführt hat, sichtbar ist. Dies erleichtert das Verständnis der intendierten Semantik. Weiters bietet OntoWiki ein Bewertungssystem für bereits existierenden Inhalt, was die Suche nach qualitativ hochwertigen Wissensmodellen erleichtern sollte. Dieses Tool unterscheidet sich allerdings von dem im Rahmen dieser Arbeit implementierten System dadurch, dass hier der informale Inhalt eines Artikels nicht im Vordergrund steht. In OntoWiki ist also ein Vermischen von informalem Text mit semantischen Annotationen ebenso nicht möglich.

Der Ansatz von IkeWiki[19] [27] ist dem Ansatz dieser Arbeit sehr ähnlich, weil auch hier der Charakter eines traditionellen Wikis erhalten geblieben ist und die semantischen Annotationen so transparent wie möglich gehalten werden, damit die Lesbarkeit des Artikels nicht negativ beeinflusst wird. Auch hier ist es möglich, existierenden informalen Text innerhalb von Artikeln mit Hilfe von semantischen Annotationen zu erweitern. Die Ontologien werden innerhalb des Systems nicht nur erstellt, sondern auch auf In-

[18] http://ontowiki.net/Projects/OntoWiki
[19] http://ikewiki.salzburgresearch.at/

stanzebene verwendet. Da in IkeWiki die Erstellung von OWL-Ontologien möglich ist, unterstützt die Applikation auch Schlussfolgerungen und Ableitungen aus bereits vorhandenem Wissen.

Platypus Wiki[20] [6] ist dieser Arbeit ebenfalls sehr ähnlich. Mit Hilfe eines Wikis ist es mit dieser Anwendung möglich, Ontologien mit den Sprachprimitiven von RDF Schema und OWL zu entwickeln. Allerdings sind in Platypus Wiki die formale und die informale Beschreibung einer Ressource voneinander getrennt. Dies bedeutet, auch hier ist ein Vermischen von semantischen Annotationen und informalem Text nicht möglich. Weiters legt die Applikation seinen Schwerpunkt eher auf die Erstellung von RDF-Beschreibungen auf Instanzebene.

Semantic MediaWiki[21] [16] ist eine Erweiterung von MediaWiki[22], jener Wiki-Engine, der auch Wikipedia zu Grunde liegt. Auch mit dieser Anwendung ist es möglich, den informalen Text eines Wiki-Artikels mit semantischen Annotationen zu erweitern. Allerdings ist eine Ontologieentwicklung im eigentlichen Sinne nicht möglich. Es muss kein Vokabular definiert werden, um Properties verwenden zu können. Sobald ein neues Property verwendet wird, wird dieses im System erstellt. Dessen Semantik wird informal auf einer eigenen Wikiseite beschrieben, welche auch editierbar ist, aber eben nur in natürlicher Sprache. Neben bereits vordefinierten Properties des Systems selbst, existieren schon sehr viele benutzerdefinierte Properties, welche man auch wieder verwenden sollte. Dieser Ansatz ähnelt eher einem Tagging-System und bietet eigentlich keine Möglichkeit zur Ontologieentwicklung. Es können allerdings externe Vokabulare und Ontologien importiert werden, wie zum Beispiel gängige Standards zur Beschreibung von Ressourcen. Der Import von Ontologien befindet sich allerdings erst im Beta-Status und ist in Version 1.0 nicht verfügbar.

SweetWiki[23] [5] konzentriert sich unter anderem auf die bessere Organisation von großen Datenmengen in traditionellen Wikis. Es stellt eine vordefinierte Ontologie zur Verfügung, welche die elementaren Konzepte eines Wikis beschreibt, um Metadaten über das Wiki und ihre Artikel selbst halten zu können. Damit sollten einige Probleme von Wikis mit großen Datenmengen eliminiert werden, wie zum Beispiel veraltete Naviga-

---

20 http://platypuswiki.sourceforge.net/

21 http://semantic-mediawiki.org/wiki/Semantic_MediaWiki

22 http://www.mediawiki.org/wiki/MediaWiki/de

23 http://sweetwiki.inria.fr/wiki/data/Main/MainHome.jsp

tionspfade oder eine aufgrund des Volumens unmögliche Wartung der existierenden Artikel. Darüber hinaus kann auch in SweetWiki der Text eines Artikels semantisch annotiert werden. Dabei ähnelt dieses System ebenso eher einem Tagging-System. Benutzer können bei der Erstellung von Wiki-Artikeln beliebige Annotationen wählen, wobei bereits existierende Annotationen mittels Auto-Vervollständigen-Funktion vorgeschlagen werden. Die Benutzer können neu verwendete Annotationen optional mit Konzepten aus vorhandenen Ontologien verknüpfen, um deren Semantik zu beschreiben. Admins und erfahrene Knowledge-Engineers kontrollieren die dadurch entstandenen „Folksonomies“ und passen diese gegebenenfalls an, wobei der gewählte Ausdruck des Benutzers für die Annotation unverändert bleibt.

Kaukolu[24] [14] ist dieser Arbeit hinsichtlich des Ansatzes wieder etwas ähnlicher. Allerdings werden in dieser Applikation RDF-Ressourcen erstellt, um bestimmte Teile des Textes in einem Artikel, z.B. einen Absatz, zu annotieren. Hier wird ein Artikel nicht als die zugleich beschriebene Ressource aufgefasst, sondern als ein Artikel, der mit mehreren unterschiedlichen RDF-Ressourcen beschrieben werden kann.

Außerdem existieren noch einige andere kollaborative Ontologieeditoren, welche allerdings kein semantisches Wiki als die zugrunde liegende Infrastruktur nutzen und auch Anwendungen, die ihren Schwerpunkt auf der Generierung von Ontologien aus Tagging-Systemen haben, welche aber in dieser Arbeit nicht näher beschrieben werden.

[24] http://kaukoluwiki.opendfki.de/

4 Implementation

Wie bereits näher erläutert, wurde im Rahmen dieser Arbeit ein System implementiert, welches das bestehende semantische Wiki Ylvi erweitert, sodass es möglich ist, die Artikel im Wiki mit SKOS-Sprachelementen zu annotieren.

Dafür wurde zuerst ein Metamodell erstellt, welches die in SKOS definierten Klassen und Properties innerhalb des Wikis zur Verfügung stellt. Allerdings ist SKOS im Vergleich zu OWL keine Sprache, mit welcher selbst Metamodelle zur Beschreibung von Ressourcen erstellt werden können. SKOS ist eigentlich selbst schon ein konkretes Modell, welches mit einem Metamodell (in diesem Fall OWL) definiert wurde, was bedeutet, dass die Ressourcen bereits auf Instanzebene beschrieben werden. Allerdings können mit SKOS hierarchische Beziehungen zwischen Konzepten beschrieben werden, welche dann auch für Artikel gelten, die Instanzen eines bestimmten Konzepts darstellen. Als Beispiel betrachte man, wie schon in Kapitel 1.2 und 2.5.2, zwei Konzepte, welche miteinander in hierarchischer Beziehung stehen, nämlich *Fahrzeug* und *Auto*, wobei *Auto* ein Subkonzept von *Fahrzeug* ist. Die hierarchische Struktur zwischen den Konzepten bleibt in einem internen Datenmodell (vgl. Kapitel 2.5.2) erhalten. Möchte man nun nach allen Artikeln suchen, welche als *Fahrzeug* typisiert wurden, würde man folglich auch alle Autos erhalten, obwohl im Quellcode des Artikels für ein konkretes Auto die Information nicht explizit angegeben wurde, dass es sich dabei auch um ein Fahrzeug handelt, sondern eben nur um ein Auto. Streng betrachtet eignet sich SKOS wohl eher weniger zur Ontologieentwicklung, aber es bietet ein Metamodell, um dem Wiki selbst mehr Semantik zu verleihen, weil dadurch zum Beispiel nach Typen gesucht werden kann, und dies auch über mehrere hierarchische Ebenen. Für das semantische Wiki stellt SKOS also dennoch ein Metamodell dar, mit welchem die Artikel in diesem Wiki formal beschrieben werden können.

Die Benutzer haben dann die Möglichkeit, mit Hilfe des *Conversion Plugins* die als SKOS-Klassen typisierten Artikel in METIS-Objekte umzuwandeln, sodass die definierten Typen für die Klassifizierung anderer Artikel auf Instanzebene zur Verfügung stehen. Dabei werden die definierten Artikel hinsichtlich des SKOS-Datenmodells auf ihre Validität überprüft. Invalide Artikel werden nicht in Objekte umgewandelt. Dies

gilt auch für Artikel, welche für sich selbst betrachtet zwar valide sind, aber in ihren Beschreibungen auf invalide Artikel verweisen.

Nach der Validierung und Konvertierung ins interne Datenmodell besteht die Möglichkeit, mit Hilfe des *Export Plugins* das zugrunde liegende SKOS-Modell aus Ylvi heraus zu exportieren. Dieses kann dann beispielsweise von einer anderen Ylvi-Instanz importiert werden.

Als dritte Komponente wurde ein *Import Plugin* implementiert, mit welchem es möglich ist, bereits bestehende SKOS-Modelle ins semantische Wiki zu übernehmen. Dabei kann es sich sowohl um extern vorliegende Modelle als auch um Modelle aus einer anderen oder derselben Ylvi-Instanz handeln.

Abbildung 9 zeigt eine Übersicht der beiden Schichten Ylvi und METIS und wie diese beiden Schichten durch die jeweiligen Plugins miteinander verbunden sind.

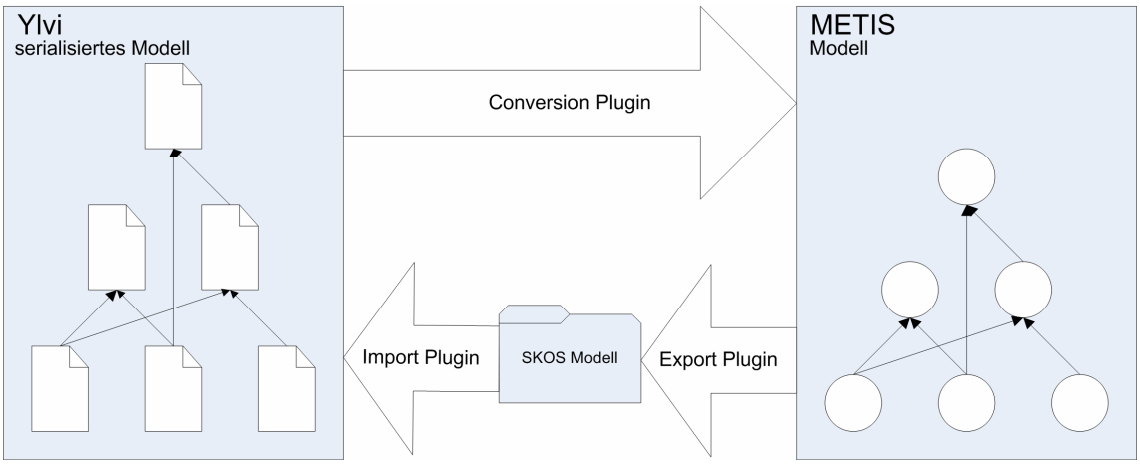

Abb. 9: Die beiden Schichten Ylvi und METIS, welche über die implementierten Plugins miteinander verbunden sind.

Das *Conversion Plugin* liest die Artikel in Ylvi aus und erzeugt korrespondierende Objekte innerhalb der METIS-Schicht. Das *Export Plugin* liest das interne Repräsentationsmodell aus und erstellt daraus ein korrespondierendes SKOS-Modell. Das *Import Plugin* importiert ein SKOS-Modell und erstellt die jeweiligen Artikel innerhalb der Ylvi-Schicht.

4.1 Ylvi / SKOS Mapping

Wie bereits beschrieben, musste zuerst das SKOS-Modell in Ylvi integriert werden, damit das SKOS-Vokabular zur Beschreibung der Artikel genutzt werden kann. Dafür

wurde ein so genanntes *Semantic Pack* erstellt [24]. In diesem *Semantic Pack* wurden für alle Klassen und Properties, welche in SKOS definiert sind, korrespondierende METIS-Objekte (vgl. Kapitel 2.5.2) definiert.

In weiterer Folge dieses Abschnitts werden „Design Goals“ mit DG abgekürzt und mit grüner Farbe gekennzeichnet. Nicht optimal umgesetzte Lösungen („Limitations“) werden mit L abgekürzt und mit blauer Farbe kenntlich gemacht.

4.1.1 Design Goals

Erst wurden einige „Design Goals“ (DG) definiert, welche vom System so gut wie möglich unterstützt werden sollten und als ein Wegweiser für Entscheidungsfragen dienten, welche im Zuge der Implementation auftraten.

DG 1 (Einfachheit):
Das System sollte sowohl für fortgeschrittene Knowledge Engineers als auch für unerfahrene Benutzer nutzbar sein. Dies bedeutet, es sollte wie in einem traditionellen Wiki möglich sein, informalen Text in einem Artikel zu verfassen. Auf diese Weise ist es für unerfahrene Benutzer möglich, an der Wissensmodellierung aktiv teilzunehmen, indem sie die Semantik einer Ressource informal beschreiben. Somit übernehmen hinsichtlich der Wissensmodellierung unerfahrene Benutzer sozusagen die Rolle der Fachexperten, wie es auch in einem traditionellen Wiki der Fall ist. Benutzer, welche die Beschreibung von Ressourcen mit SKOS beherrschen, annotieren danach den Text in den Artikeln mit semantischen Ausdrücken aus dem SKOS-Vokabular. Das System ist also eher auf eine breite Zielgruppe ausgerichtet und sollte dementsprechend leicht in der Anwendung und intuitiv nutzbar sein.

DG 2 (Charakter eines traditionellen Wikis):
Damit das Wiki auch für unerfahrene Benutzer leicht zu verwenden ist, ist es wichtig, die formalen semantischen Annotationen so transparent wie möglich zu halten, weil sich zusätzliche Information am Bildschirm negativ auf die Lesbarkeit eines Artikels auswirkt, vor allem dann, wenn der Leser mit diesen semantischen Informationen recht wenig anzufangen weiß. Der Charakter eines traditionellen Wikis und die informale Beschreibung einer Ressource sollten also im Vordergrund stehen. Das System sollte damit eher weniger einem klassischen Ontologieeditor gleichen.

DG 3 (Kompatibilität zu SKOS):
Alle Ausdrücke des SKOS-Vokabulars sollten im semantischen Wiki für die Beschreibung von Ressourcen zur Verfügung stehen Das SKOS-Modell sollte dabei so gut wie möglich abgebildet werden.

DG 4 (Validität):
Es sollen nur valide und SKOS-konforme Modelle aus den Artikeln erstellt werden. Um dies zu gewährleisten, muss eine Validierung der Artikel durchgeführt werden.

DG 5 (geringer Informationsverlust):
Es sollte versucht werden, so wenig Information wie möglich zu verlieren. Dies bezieht sich sowohl auf den Inhalt eines Artikels als auch auf das von den Artikeln gebildete SKOS-Modell. Zum Beispiel sollen bei einem Export der Artikel aus einer Ylvi-Instanz und bei einem darauffolgenden Import der aus der einen Instanz exportierten Artikel in eine andere Instanz die Artikel möglichst unverändert bleiben. Dies bezieht sich eben nicht nur auf die in SKOS formulierten Statements, sondern auch auf die informale Beschreibung eines Artikels. Umgekehrt soll bei einem Import eines extern vorliegenden SKOS-Modells und darauffolgenden Export aus Ylvi heraus, das SKOS-Modell so weit wie möglich unverändert bleiben.

4.1.2 SKOS-Metamodell

Damit die Artikel in Ylvi mit den Sprachprimitiven des SKOS-Vokabulars beschrieben werden können, müssen entsprechende METIS-Objekte zur Verfügung stehen, welche die einzelnen Komponenten des SKOS-Modells darstellen. In diesem Kapitel wird beschrieben, welche METIS-Objekte zur Repräsentation der einzelnen Elemente in SKOS verwendet werden. Zum besseren Verständnis werden in diesem Kapitel auch Beispiele angeführt, die demonstrieren sollen, wie die semantischen Annotationen im Quellcode eines Artikels verwendet werden. Diese Beispiele beziehen sich auf die in Kapitel 2.4 erläuterten Beispiele zur Erklärung des SKOS-Modells und übersetzen diese SKOS-Beschreibungen in den Quellcode eines Artikels.

Zum Beispiel wurde in Kapitel 2.4.1 ein Konzept namens *Tier* definiert. In Ylvi würde man zur Definition eines solchen Konzepts einen Artikel namens *Tier* erstellen und diesen als Konzept wie folgt typisieren:

```
<<Concept>>
```

Damit ein Artikel als *Concept* typisiert werden kann, muss bereits ein existierender Typ namens *Concept* vorhanden sein. Dieser Typ wird in METIS als *MediaType* repräsentiert, die SKOS-Klasse *Concept* im Metamodell entspricht also einem *MediaType* im internen Datenmodell. Artikel, welche als *Concept* typisiert wurden, werden bei der Konvertierung ins interne Modell selbst wieder in *MediaTypes* umgewandelt, was bedeutet, dass die dadurch neu geschaffenen *MediaTypes* wiederum zur Klassifikation weiterer Artikel genutzt werden können.

Wie jede in RDF beschriebene Ressource muss das Konzept auch eindeutig über einen URI identifizierbar sein. Dies entspricht einem Attribut des *MediaTypes Concept*, also einem *MetadataAttribute* im Datenmodell von METIS. Der URI wird als *Configuration Property*[25] des neu erstellten *MediaTypes*, im Beispiel also des *MediaTypes Tier*, gespeichert. Um DG 1 (*Einfachheit*) zu unterstützen, muss der URI nicht zwingend angegeben werden. Wenn jedoch explizit ein URI angegeben werden soll, dann erfolgt die Annotation im Artikel für das Konzept *Tier* wie folgt:

```
~URI=http://www.skosylvi.com/Tier~
```

Dies ist nur ein Beispiel für einen URI und ist frei erfunden. Es kann ein beliebiger URI gewählt werden. Das Attribut URI kann in Ylvi für alle SKOS-Klassen, nicht nur für Konzepte, verwendet werden, da, wie bereits erwähnt, alle RDF-Ressourcen eindeutig identifizierbar sein müssen.

Damit Artikel im Wiki als *ConceptScheme* (vgl. Kapitel 2.4.2) typisiert werden können, muss im vordefinierten Metamodell ein *MediaType* namens *ConceptScheme* zur Verfügung stehen. Bei der Konvertierung in das interne Modell werden als Konzeptschema klassifizierte Artikel allerdings in ein Objekt der METIS-Klasse *MediaTypeCategory* umgewandelt. Diese Klasse entspricht am ehesten der Auffassung eines Konzeptschemas, weil es möglich ist, *MediaTypes* zu *MediaTypeCategories* zuzuordnen. Und auch bei der Navigation im Wiki selbst, hat man auf diese Weise alle Instanzen von Konzepten eines bestimmten Konzeptschemas zusammengefasst, wie in Abbildung 10 zu sehen ist.

---

[25] Für Objekte der Klasse *MediaType* können in METIS beliebig viele *Configuration Properties* erstellt werden. *Configuration Properties* bestehen dabei aus einem Namen und dem dazugehörigen Wert, also ähnlich einem Attribut mit dazugehörigem Attributwert. Für ein *Configuration Property* kann ein beliebiger Name gewählt werden.

Abb. 10: Navigationsmenü für die typenbasierte Suche in Ylvi. Die Worte in Großbuchstaben repräsentieren die Konzeptschemen (MediaTypeCategories), in welchen verschiedene Konzepte (MediaTypes) enthalten sind. Wählt man in diesem Menü ein Konzept aus, werden alle Artikel, welche eine Instanz des ausgewählten Konzepts darstellen, aufgelistet.

In Kapitel 2.4.2 wurde ein Konzeptschema namens *Biologie* definiert. In Ylvi müsste man dafür einen Artikel namens *Biologie* erstellen und ihn als *ConceptScheme* typisieren:

```
<<ConceptScheme>>
```

L 1: *MediaTypeCategories* besitzen kein *Configuration Property* wie *MediaTypes*, in welchem man den URI eines bestimmten Konzeptschemas speichern könnte. Dies ist allerdings notwendig, denn beim Import einer im Wiki bereits vorhandenen Ressource muss diese anhand des URIs wiedererkannt werden, da es sich aufgrund der Eindeutigkeit eines URIs um dieselbe Ressource handeln muss. Als Lösung wird der URI in das

description-Feld des *MediaTypeCategory*-Objekts geschrieben, welches normalerweise für eine kurze Beschreibung der jeweiligen *MediaTypeCategory* zur Verfügung steht.

Das Property *skos:inScheme*, welches Konzepte in bestimmte Konzeptschemen einordnet, ist eine Instanz der Klasse *owl:ObjectProperty*, was bedeutet, dass zwei Ressourcen miteinander in Beziehung gesetzt werden. Im METIS-Datenmodell wird diese Beziehung daher mit Hilfe eines *AssocTypes* dargestellt. In Ylvi müsste der Artikel *Tier* um folgende semantische Annotation erweitert werden:

```
[inScheme [Biologie]]
```

L 2: In METIS ist es nicht möglich, einen *MediaType* in mehrere *MediaTypeCategories* einzuordnen. Somit werden Artikel eines bestimmten Konzepts in Ylvi auch nur in einer Kategorie angezeigt, wenn das jeweilige Konzept in mehr als ein Konzeptschema eingeordnet wurde. Allerdings betrifft dies nur das interne Repräsentationsmodell. Beim Export werden dennoch alle Statements dieser Form berücksichtigt, weil die im internen Modell nicht berücksichtigten Statements aus dem Quellcode des Artikels ausgelesen werden können.

Das Property *skos:hasTopConcept* ist ebenfalls eine Instanz der Klasse *owl:ObjectProperty*, was bedeutet, dass dieses Property im METIS-Datenmodell wiederum als *AssocType* repräsentiert wird. In Ylvi müsste innerhalb des Artikels *Biologie* folgendes Statement formuliert werden:

```
[hasTopConcept [Tier]]
```

L 3: Das Property *skos:topConceptOf* ist als *owl:inverseOf* von *skos:hasTopConcept* definiert. Dieses Property wurde allerdings erst im fortgeschrittenen Stadium dieser Arbeit ins SKOS-Vokabular aufgenommen und wird daher im implementierten System nicht berücksichtigt. Es ist in der SKOS-Referenz allerdings ohnehin eine Warnung dafür ausgegeben, dass es in Zukunft unter Umständen wieder verändert oder gänzlich entfernt wird.

Concept und *ConceptScheme* sind disjunkte Klassen (vgl. IB 1, Kapitel 2.4.2). Für Ylvi bedeutet dies, dass kein Artikel als *Concept* und *ConceptScheme* zugleich typisiert werden darf.

Ein Lexical Label (vgl. Kapitel 2.4.3) entspricht im METIS-Datenmodell einem *MetadataAttribute* eines *MediaTypes*. Da in METIS kein Datentyp für Zeichenketten in ver-

schiedenen Sprachen zur Verfügung steht, wird ein gewöhnlicher String als *Metadata-Type* definiert.

Zur Annotation von Lexical Labels wird in Ylvi dieselbe Syntax wie in TURTLE verwendet. Der Text des Artikels für das Konzept *Tier* könnte demnach um folgende Statements erweitert werden:

```
~prefLabel=Tier@de~
~prefLabel=animal@en~
~altLabel=Vieh@de~
~hiddenLabel=Tiehr@de~
~hiddenLabel=annimal@en~
```

Es sei zu beachten, dass bei der Angabe der Zeichenkette die Anführungszeichen in der Ylvi-Syntax weggelassen werden.

L 4: Die Angabe einer Sprache mit einem @ und darauf folgenden Sprachcode ist nicht optimal, weil dies gegen Design Goal 2 (*Charakter eines traditionellen Wikis*) verstößt. Design Goal 1 schreibt vor, dass die semantischen Annotationen nicht wie im davor beschriebenen Beispiel isoliert im Artikel stehen sollen, sondern mit dem vorhandenen informalen Text verknüpft werden sollen. Wenn im Text das Wort Tier vorkommt, was sehr wahrscheinlich ist, könnte man dieses Wort als *prefLabel* definieren, allerdings würde dann im Text des Wikis Tier@de aufscheinen, was die Lesbarkeit des Artikels einschränkt und auch die Transparenz im negativen Sinne beeinflusst.

Notations (vgl. Kapitel 2.4.4) werden im internen Datenmodell ebenfalls als *Metadata-Attribute* repräsentiert. Dieses Attribut darf allerdings nicht von allen SKOS-Klassen, sondern nur vom *MediaType Concept* verwendet werden. Da in METIS kein generischer Datentyp zur Verfügung steht, wird aufgrund seiner Kompatibilität und Konvertierungseigenschaften zu den anderen Datentypen der Datentyp *String* als *MetadataType* definiert und der jeweilige Datentyp wird, wie in TURTLE, an die tatsächliche Notation angehängt. Allerdings ist es in Ylvi nicht nötig, den Namespace des jeweiligen XML Schema Datentyps anzugeben. Der Artikel *Tier* könnte also folgende Notation definieren:

```
~notation=Tier^^string~
```

L 5: Wird eine Notation als *String* repräsentiert, wird der dabei an den String angehängte Datentyp nicht tatsächlich als solcher interpretiert. Aber DG 5 (*geringer Informationsverlust*) wird dennoch unterstützt, weil die Information, um welchen Datentypen es sich handelt, sowohl beim Export als auch beim Import erhalten bleibt.

L 6: Eigentlich können auch selbst definierte Datentypen als Datentyp für eine Notation verwendet werden. In der ersten Version dieses Systems werden allerdings nur die vordefinierten Datentypen von XML Schema unterstützt.

L 7: Das Anfügen der Zeichenkette ^^ und dem zu repräsentierenden Datentypen ist wiederum bezüglich DG 2 (*Charakter eines traditionellen Wikis*) nicht optimal, weil dies die Lesbarkeit eines Artikels negativ beeinflusst. (vgl. L 4).

Wie in Kapitel 2.4.5 bereits beschrieben, stellt SKOS drei verschiedene Muster zur Dokumentation von Ressourcen zur Verfügung, nämlich Dokumentation als Plain Literal, als RDF-Beschreibung und als Referenz auf ein Dokument.

L 8: In der ersten Version des im Rahmen dieser Arbeit implementierten SKOS-Editors wird nur Dokumentation als Plain Literal unterstützt. Die anderen beiden Muster zur Dokumentation von Ressourcen werden nicht berücksichtigt. Für den Import bedeutet dies, dass Statements dieser Art ignoriert werden.

Da im implementierten System nur die Dokumentation einer Ressource als Plain Literal unterstützt wird, werden Documentation Properties als *MetadataAttribute* eines *MediaTypes* dargestellt. Der gewählte *MetadataType* für dieses Attribut ist ein *String*. In Ylvi könnte man den Artikel *Tier* um folgende Statements erweitern:

```
~skos:note=Dies ist ein Tier-Konzept.~
~skos:changeNote=Konzept wurde von Konzeptschema ‚Lebewesen' nach Kon-
zeptschema ‚Biologie' verschoben.~
~skos:definition=Tiere sind Lebewesen, welche…~
~skos:editorialNote=Bitte die hierarchischen Beziehungen dieses Kon-
zepts in anderen Konzepten entfernen, wenn dieses Konzept entfernt
wird.~
~skos:example=Streng betrachtet sind auch Menschen Tiere.~
~skos:scopeNote=Dieses Konzept wird für die Typisierung aller Instan-
zen von Tieren verwendet.~
```

L 9: Auch bei den Documentation Properties ist die Angabe einer Sprache mit einem @ und darauf folgenden Sprachcode nicht optimal. (vgl. L 4 und L7).

Alle Semantic Relation Properties (vgl. Kapitel 2.4.6) sind Instanzen der Klasse *owl:ObjectProperty*. Somit werden auch diese Properties als *AssocTypes* im internen Datenmodell von METIS dargestellt. Das Property *skos:semanticRelation* wird nicht für die Formulierung von Statements verwendet und aus diesem Grund auch nicht im Metamodell zur Verfügung gestellt.

Für unser Beispiel könnte man nun zwei neue Artikel erstellen und hierarchische Beziehungen zum bereits vorhandenen Konzept *Tier* definieren. Der Artikel *Saeugetier* wird wie folgt formuliert:

```
<<Concept>>
[broader [Tier]]
```

Der Artikel für das Konzept *Lebewesen* würde ähnlich aussehen:

```
<<Concept>>
[narrower [Tier]]
```

L 10: In SKOS dürfen die Properties zur Beschreibung von hierarchischen Beziehungen auch reflexiv verwendet werden, demnach sind Statements der Form `<Tier> skos:broader <Tier>` in SKOS erlaubt. Solche Beziehungen können im METIS-Datenmodell jedoch nicht abgebildet werden. Da im Fall des semantischen Wikis solche Relationen allerdings sowieso vermieden werden sollten, werden Statements dieser Form in Ylvi unterbunden. Auf diese Weise werden die Benutzer sozusagen dazu gezwungen, strenge Hierarchiestrukturen zwischen den Konzepten beizubehalten.

Es werden in Ylvi zwei weitere Artikel erstellt, nämlich für das Konzept *Tiger* und für das Konzept *Loewe*. Diese beiden Konzepte sind Subkonzepte des Konzepts *Saeugetier* und sind verwandt, weil es sich bei beiden Tieren um Katzen handelt. Der Artikel *Tiger* würde folgende Statements enthalten:

```
<<Concept>>
[broader [Saeugetier]]
[related [Loewe]]
```

Der Artikel für das Konzept *Loewe* würde wiederum ähnlich aussehen:

```
<<Concept>>
[broader [Saeugetier]]
[related [Tiger]]
```

L 11: Es ist zum aktuellen Zeitpunkt im METIS-Datenmodell nicht möglich, assoziative Relationen zwischen *MediaTypes* abzubilden, dies ist nur für hierarchische Relationen möglich. Dies bedeutet, in Artikeln definierte assoziative Relationen sind nach der Konvertierung ins interne Modell in diesem nicht vorhanden. Allerdings sind diese Relationen immer noch in dem Text eines Artikels selbst vorhanden, was bedeutet, dass für den Export nicht nur das interne Modell herangezogen werden muss, sondern auch Statements aus dem Quellcode eines Artikels ausgelesen werden müssen.

L 12: Auch *skos:related* darf in SKOS reflexiv verwendet werden, demnach ist ein Statement der Form `<Tiger> skos:related <Tiger>` valide. In Ylvi können *AssocTypes* zwar auch reflexiv verwendet werden, jedoch wird eine in einem Artikel formulierte reflexive Beziehung nicht in der Übersicht aller Beziehungen eines Artikels dargestellt. Diese Limitation bezieht sich allerdings nur auf die Visualisierung der Beziehung.

Wie bereits im Kapitel 2.4.6 näher beschrieben, sind die Properties *skos:broader* und *skos:narrower* grundsätzlich nicht als transitive Properties definiert. Es steht aber einer Applikation frei, diese beiden Properties trotzdem als transitive Properties zu interpretieren. In Ylvi sind indirekte hierarchische Beziehungen sehr wohl von Bedeutung, weil diese die typenbasierte Suche im Wiki wesentlich verbessern. Wird zum Beispiel nach allen Artikeln gesucht, welche ein konkretes *Tier* darstellen, dann wären auch alle als *Tiger* typisierten Artikel in der gefundenen Menge enthalten, weil ein Tiger natürlich auch ein Tier ist. Diese verbesserte Suche ist, wie bereits erwähnt, eine der Eigenschaften, welche ein semantisches Wiki von einem traditionellen Wiki unterscheidet. Aus diesem Grund werden die Properties *broader* und *narrower* im implementierten System als transitive Properties interpretiert.

Um diese transitive Eigenschaft auch für Schlussfolgerungen nutzen zu können, stellt SKOS die beiden Properties *skos:broaderTransitive* und *skos:narrowerTransitive* zur Verfügung, welche auch tatsächlich als transitive Properties definiert sind. Da diese beiden Properties aber ohnehin nicht für die explizite Beschreibung von Ressourcen gedacht sind, werden diese Properties auch nicht im Metamodell zur Verfügung gestellt.

L 13: Hierarchische Zyklen sind in SKOS erlaubt, allerdings sind diese mit dem METIS-Datenmodell nicht abbildbar. Da im semantischen Wiki solche Statements ohnehin sinnlos wären und auch vermieden werden sollten, wird bei der Validierung der Artikel überprüft, ob hierarchische Zyklen vorhanden sind. Wenn dem so ist, werden alle Artikel, welche bei der Bildung eines hierarchischen Zyklus beteiligt sind, nicht in korrespondierende *MediaTypes* umgewandelt.

Um Artikel in Ylvi als *Collection* (vgl. Kapitel 2.4.7) typisieren zu können, muss im Metamodell ein *MediaType* namens *Collection* vorhanden sein. Die Properties zur Beschreibung der Elemente einer *Collection* (*skos:member* und *skos:memberList*) sind Instanzen der Klasse *owl:ObjectProperty*. Somit werden auch diese Properties im METIS-Datenmodell als *AssocTypes* repräsentiert. Der Artikel *Tiere*, der eine nicht geord-

nete *Collection* beschreibt, welche alle in einem KOS definierten Tierkonzepte beinhaltet, würde in Ylvi wie folgt aussehen:

```
<<Collection>>
[member [Tiger]]
[member [Loewe]]
[member [Saeugetier]]
```

Es wurde entschieden, in der ersten Version des implementierten Systems keinen eigenen *MediaType* für eine *OrderedCollection* zur Verfügung zu stellen. Stattdessen wird ein Artikel, der eine geordnete *Collection* beschreiben soll, genau gleich beschrieben wie im nicht geordneten Fall, außer dass die folgende Attributdefinition hinzugefügt wird, welche die *Collection* als eine geordnete Sammlung kennzeichnet:

```
~ordered=true~
```

Es wurde ebenso entschieden, auch nur einen *AssocType* für beide *Collections* zur Verfügung zu stellen, demnach wird *member* also auch für die Beschreibung der Elemente einer *OrderedCollection* verwendet. Die Bedeutung der Reihenfolge ergibt sich aus der Reihenfolge, in welcher die Ressourcen tatsächlich im Artikel definiert sind.

L 14: Die Lösung der Attributangabe, wenn es sich um eine geordnete *Collection* handelt, ist nicht optimal, weil sie gegen Design Goal 2 (*Charakter eines traditionellen Wikis*) verstößt. Dies verringert zudem die Lesbarkeit eines Artikels. (vgl. L 4, L 7 und L 9)

skos:Collection ist sowohl mit *skos:Concept* als auch mit *skos:ConceptScheme* disjunkt. Da *Concepts* und *ConceptSchemes* ebenso disjunkt sind, bedeutet dies, dass keine Ressource als eine Instanz zweier verschiedener SKOS-Klassen definiert sein darf. Für Ylvi bedeutet dies, dass kein Artikel als Instanz mehrerer unterschiedlicher SKOS-Klassen zugleich typisiert sein darf. Ein Artikel darf also entweder ein *Concept*, ein *ConceptScheme* oder eine *Collection* darstellen, nicht aber mehrere dieser Ressourcen gleichzeitig.

Alle Mapping Properties in SKOS (vgl. Kapitel 2.4.8) sind Instanzen der Klasse *owl:ObjectProperty*, was bedeutet, dass auch diese Properties als *AssocType* im Metamodell definiert sind.

Da das Property *skos:mappingRelation* lediglich das Superproperty aller anderen Mapping Properties darstellt und nicht zur Beschreibung von Ressourcen verwendet wird, wird dieses Property auch nicht im Metamodell zur Verfügung gestellt.

Das in Kapitel 2.4.8 beschriebene Beispiel zur Verwendung des Properties *skos:exactMatch*, drückt aus, dass zwei Konzepte (*Wolf* und *Canis Lupus*) aus verschiedenen Konzeptschemen identisch sind. Der Artikel für das Konzept *Wolf* würde in Ylvi wie folgt aussehen:

```
<<Concept>>
[exactMatch [Canis Lupus]]
```

Die Properties zur Beschreibung von hierarchischen Beziehungen zwischen Konzepten aus verschiedenen Konzeptschemen (*skos:broadMatch* und *skos:narrowMatch*) werden analog zu den gewöhnlichen Properties zur Beschreibung von hierarchischen Beziehungen verwendet. Auch diese beiden Properties werden, wie die Semantic Relation Properties zur Beschreibung von hierarchischen Beziehungen, vom implementierten System als transitive Properties interpretiert.

L 15: Wie bereits in L 10 beschrieben, dürfen auch die hierarchischen Mapping Properties reflexiv verwendet werden, somit sind Statements der Form `<A> skos:broadMatch <A>` im SKOS-Datenmodell erlaubt. Dies kann im METIS-Datenmodell nicht abgebildet werden, ist aber ohnehin nicht wünschenswert.

L 16: Wie bei den Semantic Relation Properties ist es auch mit *skos:broadMatch* und *skos:narrowMatch* nicht untersagt, hierarchische Zyklen zu bilden. Demnach sind `<A> skos:broadMatch <B>`, `<B> skos:broadMatch <C>` und `<C> skos:broadMatch <A>` valide SKOS-Statements. Wie bereits in L 13 beschrieben, sind hierarchische Zyklen in METIS allerdings nicht abbildbar.

Auch das Property *skos:relatedMatch* wird gleich verwendet wie sein Superproperty *skos:related*.

L 17: Assoziative Relationen zwischen *MediaTypes* sind nach der Konvertierung ins interne Repräsentationsmodell in diesem nicht vorhanden. Die Relationen bleiben aber in serialisierter Form im Quellcode des Artikels erhalten. (vgl. L 11)

L 18: Auch *skos:relatedMatch* darf reflexiv verwendet werden, demnach ist ein Statement der Form `<A> skos:relatedMatch <A>` in SKOS erlaubt. Wie bereits in L 12 beschrieben, können solche Relationen zwar definiert werden, sind jedoch nicht in der Übersicht aller definierten Relationen eines Artikels aufgelistet.

L19: Das Property *skos:closeMatch* wurde erst zu einem Zeitpunkt in das SKOS-Vokabular aufgenommen, als diese Arbeit bereits weit fortgeschritten war und wird aus diesem Grund in dieser Arbeit nicht berücksichtigt.

L 20: Wie in Kapitel 2.4.9 bereits beschrieben, können in SKOS eigene Klassen und Properties definiert werden, welche von vordefinierten SKOS-Elementen abgeleitet werden. Diese können dann mit Hilfe von RDF Schema- oder OWL-Statements semantisch erweitert werden. Um diese Funktionalität unterstützen zu können, müsste auch RDF Schema und OWL als Metamodell ins System integriert werden, was einer vollständigen Integration dieser beiden Sprachen entsprochen hätte. Aus diesem Grund wird Class- und Property Extension in der ersten Version dieses Systems nicht unterstützt.

Abbildung 11 fasst noch einmal das gesamte für Ylvi erstellte SKOS-Metamodell im Überblick zusammen.

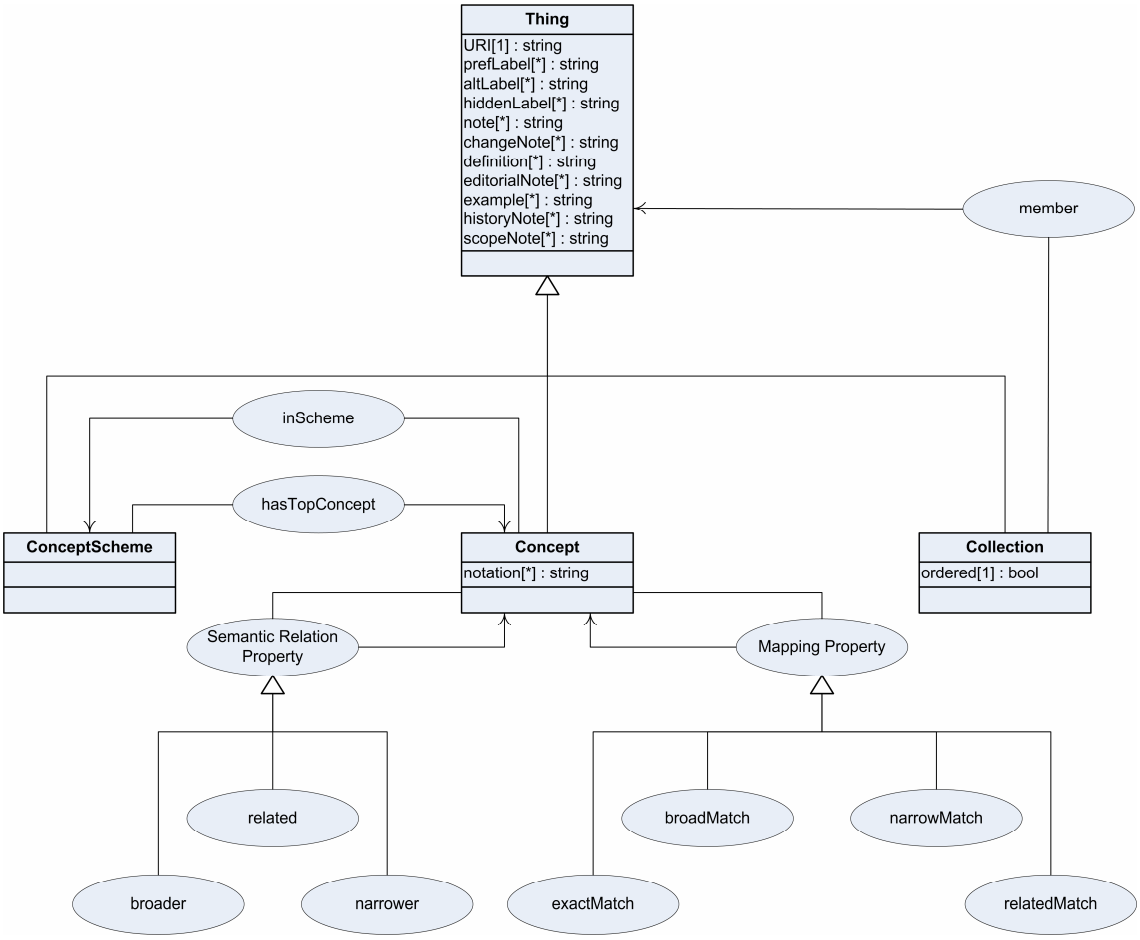

Abb. 11: Das gesamte in Ylvi zur Verfügung gestellte SKOS-Metamodell.

SKOS-Klassen (*MediaTypes*) werden in dieser Abbildung mit Rechtecken repräsentiert. Die für die jeweiligen Klassen definierten Attribute stehen innerhalb des jeweiligen Rechtecks, wobei der Datentyp und die Kardinalität für das jeweilige Attribut angegeben sind. Beziehungen zwischen diesen Klassen (*AssocTypes*) werden als Ellipsen dargestellt, wobei hier auf die Angabe der Kardinalität verzichtet wird, weil diese für alle definierten *AssocTypes* ohnehin * (0 bis unendlich) beträgt.

4.2 Conversion Plugin

4.2.1 Einführung

Das *Conversion Plugin* wandelt die als SKOS-Klassen typisierten Artikel des Wikis in die entsprechenden Objekte des METIS-Datenmodells um. Wie in Abbildung 1 ersichtlich, handelt es sich dabei um die Umwandlung von Artikeln, welche mit dem zur Verfügung gestellten Metamodell beschrieben werden, zu Objekten (zum Beispiel *MediaTypes*), welche dann in weiterer Folge zur semantischen Beschreibung von anderen Artikeln genutzt werden können.

4.2.2 Implementierungsdetails

Vom später beschriebenen *Export Plugin* sollen nur konsistente SKOS-Modelle exportiert werden. Damit das *Export Plugin* von einem konsistenten Modell ausgehen kann, ist es wichtig, dass auch nur SKOS-konforme Artikel in METIS-Objekte umgewandelt werden. Aus diesem Grund werden im *Conversion Plugin* zuerst alle Artikel validiert.

Dabei wird für jeden Artikel überprüft, ob die im Artikel verwendeten Properties auch wirklich von dieser Klasse verwendet werden dürfen, ob also der Domain des jeweiligen Properties richtig verwendet wird. Verwendet ein Konzept zum Beispiel das Property *hasTopConcept*, wird eine entsprechende Warnung für dieses Konzept ausgegeben. Dennoch werden Artikel, welche Statements mit nicht für sie gedachten Properties formulieren, in Objekte umgewandelt. Das jeweilige Statement wird in diesem Fall einfach ignoriert. Anders verhält es sich bei der Überprüfung, ob der Range von bestimmten Properties richtig verwendet wird. Wie in Kapitel 2.4.6 bereits beschrieben, dürfen Semantic Relation Properties nur verwendet werden, um Konzepte miteinander in Beziehung zu setzen. Wenn im Artikel *Tiger* eine *broader*-Beziehung zu dem Artikel *Biologie* definiert ist und es sich bei dem Artikel *Biologie* um ein Konzeptschema handelt,

dann wird der Artikel, welcher dieses Statement beinhaltet, nicht in ein METIS-Objekt, in diesem Fall in einen *MediaType*, umgewandelt. Zusätzlich müssen alle in Kapitel 2.4 beschriebenen Integritätsbedingungen für die jeweiligen SKOS-Klassen überprüft werden.

Alle Artikel, welche einem SKOS-konformen Modell entsprechen, werden in METIS-Objekte umgewandelt, allerdings nur, wenn in keinem ihrer Statements auf einen anderen, invaliden Artikel verwiesen wird. Wenn also ein Artikel in einem seiner Statements auf einen Artikel verweist, der selbst nicht valide ist, dann darf der referenzierende Artikel ebenso nicht in ein Objekt umgewandelt werden. Dies ist notwendig, weil die aus den Artikeln erstellten Objekte im internen Repräsentationsmodell miteinander verknüpft sind. Zum Beispiel werden hierarchische Beziehungen zwischen Konzepten im Modell festgehalten. Für das Statement `<Tiger> broader <Saeugetier>` würde jenem *MediaType*-Objekt, welches das Konzept *Tiger* repräsentiert, das *MediaType*-Objekt für das Konzept *Saeugetier* als Superklasse zugewiesen werden. Wäre der Artikel für Säugetier allerdings nicht konsistent bezüglich des SKOS-Modells, würde kein entsprechendes Objekt existieren und die semantische Beziehung zwischen diesen beiden Konzepten könnte nicht im internen Repräsentationsmodell festgehalten werden. Deswegen werden bei der Validierung alle für sich validen Artikel zwischengespeichert. Ebenso werden für jeden Artikel die in diesem Artikel referenzierten Artikel festgehalten. Danach wird für jeden validen Artikel überprüft, ob jeder seiner referenzierten Artikel ebenso valide ist. Referenziert ein Artikel auf einen invaliden Artikel, darf er selbst nicht mehr in ein Objekt umgewandelt werden und wird deswegen aus der Liste der validen Artikel entfernt. Dieser Vorgang wird solange wiederholt, bis bei einem Durchlauf aller übrig gebliebenen validen Artikel kein Artikel mehr aus der Liste entfernt werden muss. Nach dieser Überprüfung ist sichergestellt, dass sich nur mehr valide Artikel in dieser Liste befinden, welche auch nur auf andere valide Artikel in ihren Statements verweisen.

Alle in dieser Liste übrig gebliebenen Artikel können nun in METIS-Objekte umgewandelt werden. Dabei erhält das aus einem Artikel erstellte Objekt den Namen des korrespondierenden Artikels als eigenen Namen.

L 21: Problematisch wird dies, wenn ein Objekt bereits vorhanden ist, welches denselben Namen hat wie der umzuwandelnde Artikel. Dies kann in zwei Fällen auftreten.

Erstens könnten zwei Artikel mit demselben Namen vorhanden sein[26]. Zweitens könnte ein Artikel bereits in ein Objekt umgewandelt und daraufhin gelöscht worden sein. Würde nun wieder ein neuer Artikel mit diesem Namen erstellt werden, ist im Modell immer noch das vom anderen Artikel erstellte Objekt vorhanden. In beiden dieser Fälle wird an den Namen des zu erstellenden Objekts ein „_copy“ angehängt, damit es vom anderen Objekt unterschieden werden kann. Dies gilt für alle im Metamodell definierten *MediaTypes*, also für *Concept*, *ConceptScheme* und *Collection*.

Zuerst werden alle Artikel, welche als *ConceptScheme* typisiert sind, in Objekte der Klasse *MediaTypeCategory* umgewandelt. Dass diese als erstes erstellt werden, macht vor allem deswegen Sinn, weil einem Konzept, welches durch einen *MediaType* dargestellt wird, bereits eine *MediaTypeCategory* zugewiesen werden kann, wenn im Artikel des Konzepts eine *inScheme*-Beziehung zu einem *ConceptScheme* formuliert wurde. Damit aus der erstellten *MediaTypeCategory* der korrespondierende Artikel ermittelt werden kann, muss die eindeutige interne ID für einen Artikel bei der *MediaTypeCategory* hinterlegt werden.

L 22: Da, wie bereits in L 1 beschrieben, *MediaTypeCategories* kein *Configuration Property* besitzen, wird die ID für den Artikel im *description*-Feld des jeweiligen Objekts gespeichert.

Danach werden alle Artikel, welche Konzepte repräsentieren, in *MediaTypes* umgewandelt. Hier werden auch hierarchisch definierte Beziehungen zwischen *MediaType*-Objekten im Modell gespeichert. Deshalb muss man erst alle Konzepte in *MediaTypes* umwandeln und dann noch mal durchlaufen, um die hierarchisch verwandten Konzepte den jeweiligen Konzepten zuweisen zu können. Bei einmaligem Durchlauf könnte es nämlich dazu kommen, dass einem *MediaType* ein hierarchisch verwandter *MediaType* zugewiesen wird, welcher noch gar nicht existiert, was einen Programmabbruch zur Folge hätte.

---

[26] In Ylvi ist es möglich, einen Artikel zugleich als eine Instanz des Datenmodells (z.B. als ein Individuum) und als eine Instanz des Metamodells (z.B. als ein *Concept*) zu typisieren. Dies macht durchaus Sinn. Als Beispiel betrachte man den Begriff *Free Jazz*. Damit könnte die Musikrichtung gemeint sein (als *Concept*) aber auch das gleichnamige Album von Ornette Coleman (als Instanz des Konzepts *Album*). Damit diese beiden verschiedenen Dinge auch voneinander getrennt beschrieben werden können (also in verschiedenen Artikeln), ist es notwendig, dass zwei Artikel innerhalb von Ylvi denselben Namen haben können. [25]

Zu guter Letzt werden die als *Collection* typisierten Artikel in *MediaTypes* umgewandelt. Nun könnten die erstellten *MediaTypes* bereits verwendet werden, um andere Artikel im semantischen Wiki zu typisieren. Wie bei *MediaTypeCategories*, wird auch bei *MediaTypes* (also *Concepts* und *Collections*) die interne ID des Artikels hinterlegt, hier jedoch nicht im *description*-Feld, sondern in einem *Configuration Property*.

4.3 Export Plugin

4.3.1 Einführung

Nach der Umwandlung haben die Benutzer die Möglichkeit, das durch die Artikel beschriebene SKOS-Modell durch das Klicken eines Buttons im System zu speichern. Dafür wird ein *SMO* erstellt, welches das aktuelle SKOS-Modell in einer bestimmten RDF-Serialisierungsform enthält.

Dieses im *SMO* enthaltene, serialisierte Modell soll dann in weiterer Folge exportiert werden können. Dies bedeutet, das Modell soll von den Benutzern in einer bestimmten Serialisierungsform lokal auf ihren Rechnern gespeichert werden können. Die Serialisierungsform sollte dabei nicht vom System vorgegeben werden, sondern soll mittels Content Negotiation ermittelt werden. Dies bedeutet, dass die Präferenzen des Benutzers für eine bestimmte Serialisierungsform im Header des Requests codiert sind, diese von der Anwendung ausgelesen werden und je nach vorliegender Präferenz das Modell in einer bestimmten Serialisierungsform exportiert wird.

So wäre es möglich, ein Modell aus der einen Instanz zu exportieren, um es danach wieder in eine völlig andere Instanz zu importieren. Somit könnte man auch gut überprüfen, ob bzw. wie weit DG 5 (*geringer Informationsverlust*) hält, also ob ein Informationsverlust bei Export und darauffolgenden Import zu beobachten ist oder nicht.

L 23: Man hat sich dann aber dazu entschlossen, in der ersten Version des Systems das erstellte SKOS-Modell nicht mittels Content Negotiation zum Download anzubieten, weil diese Funktionalität nicht wirklich in die eigentliche Anforderung des Systems fällt. Stattdessen wird vorerst nur mal das *SMO* erstellt, welches das SKOS-Modell in serialisierter Form enthält. Es ist aber von einer Ylvi-Instanz aus möglich, auf ein *SMO* einer anderen Instanz zuzugreifen. Dies bedeutet, es können auch auf diese Weise Modelle aus fremden Instanzen in die eigene Instanz importiert werden, was im Grunde

genommen mit einem Export einer Instanz und einem darauffolgenden Import einer anderen Instanz gleichzusetzen ist.

4.3.2 Implementierungsdetails

Die Erstellung des *SMO* geschieht mit Absicht nicht automatisch, weil es sein könnte, dass nicht alle Artikel valide waren und die Benutzer erst die invaliden Statements in den Artikeln korrigieren möchten, bevor sie erneut die Umwandlung durchführen. Würde automatisch das SKOS-Modell nach der Umwandlung im System gespeichert werden, könnte ein gültiges, bereits existierendes Modell überschrieben werden. Deswegen ist es nach der Validierung und Umwandlung auch wichtig, die Benutzer in Form von Erfolgs- oder Fehlermeldungen für jeden Artikel über das gerade eben erstellte Modell zu informieren. Dabei sollten die jeweiligen Meldungen genaue Auskunft darüber geben, welche Statements in welchen Artikeln nicht valide sind, damit die fehlerhaften Aussagen in den Artikeln schnell korrigiert werden können.

Weil die Erstellung des SKOS-Modells aber nicht automatisch nach der Umwandlung ausgeführt wird, könnte es zu dem Fall kommen, dass aus den Artikeln die korrespondierenden Objekte erstellt werden, der Benutzer dann jedoch eine gewisse Zeit verstreichen lässt, bis er dann das serialisierte SKOS-Modell erstellt. So könnte es passieren, dass zum Zeitpunkt der Umwandlung alle Artikel valide sind und somit auch alle Artikel im Wissensmodell enthalten wären, zum Zeitpunkt der Erstellung des SKOS-Modells zumindest ein Artikel allerdings nicht mehr valide ist, weil ein anderer Benutzer diesen in der Zwischenzeit verändert hat und zwar so, dass der Artikel inkonsistent wurde. Dies würde bedeuten, dass ein invalides SKOS-Modell gespeichert werden könnte, was allerdings unter keinen Umständen erwünscht ist, weil dies gegen DG 4 (*Validität*) verstößt.

Deswegen werden alle in einem Artikel beschriebenen Statements bereits bei der Validierung des *Conversion Plugins* (vgl. Kapitel 4.2) zwischengespeichert, um zu gewährleisten, dass die Daten zum Zeitpunkt der Umwandlung auch wirklich dieselben sind wie jene zum Zeitpunkt der Erstellung des SKOS-Modells.

L 24: Es könnte aber dennoch passieren, dass Artikel schon während der Umwandlung von anderen Benutzern modifiziert werden, da die Umwandlung bei einer großen Menge von Artikeln doch etwas Zeit in Anspruch nimmt. Allerdings ist hier die Wahrscheinlichkeit, dass so ein Fall eintritt, wesentlich geringer als im vorher beschriebenen

Fall, weil es sich hier um Zeitspannen im Sekundenbereich handelt, während die Zeitspanne zwischen Umwandlung und Erstellung des SKOS-Modells theoretisch unendlich sein könnte.

Die aus dem Wiki exportierten Ressourcen erhalten dabei jenen URI, welcher von den Benutzern in den Artikeln angegebenen wurde. Diese Angabe ist jedoch nicht zwingend vorgeschrieben, deswegen muss in Fällen, in denen der URI nicht explizit angegeben wurde, ein URI erstellt werden. Dieser URI setzt sich aus dem URL für die laufende Instanz gefolgt von einem Slash und dem Namen des korrespondierenden Objekts in METIS zusammen, also z.B. `http://www.skosylvi.com/Tiger`. Damit bei einem möglichen späteren Import überprüft werden kann, ob die Ressource bereits im lokalen SKOS-Modell enthalten ist, muss auch der URI beim entsprechenden METIS-Objekt hinterlegt werden.

L 25: Auch hier gilt wieder die Einschränkung, dass dieser URI bei *MediaTypeCategories* in das *description*-Feld geschrieben werden muss (vgl. L 1 und L 22).

L 26: Zu Problemen könnte es kommen, wenn mehr als zwei Artikel den gleichen Namen haben, weil dann auch zumindest zwei METIS-Objekte vorhanden sind, welche denselben Namen haben. Sind zum Beispiel drei Artikel namens *Lebewesen* im Wiki vorhanden, werden bei der Konvertierung der Artikel drei METIS-Objekte erstellt, eines namens *Lebewesen*, und zwei namens *Lebewesen_copy*. Dies würde bedeuten, es wären im exportierten Modell zwei Ressourcen mit demselben URI vorhanden, was einen Konflikt darstellen würde. Allerdings bezieht sich diese Limitation eher auf einen Anwendungsfehler, weil für Artikel mit gleichen Namen von den Benutzern selbst URIs vergeben werden sollten, um diese unterscheiden zu können.

Bei einem Import soll ein Artikel für jede Ressource im importierten Modell erstellt werden. Ein Artikel muss einen Namen erhalten und dafür wird der Wert des *prefLabels* der jeweiligen Ressource in der im System eingestellten Standardsprache herangezogen. Ist im Modell kein *prefLabel* in der Standardsprache definiert, wird der URI der Ressource als Artikelname übernommen.

L 27: Dies könnte bei einem Import von extern beschriebenen Modellen problematisch werden, vor allem dann, wenn dieses Modell in einer anderen Sprache beschrieben ist. In diesem Fall könnte es vorkommen, dass überhaupt keine *prefLabels* in der Standard-

sprache des Systems formuliert sind. In solchen Fällen würde ein Artikel mit dem URI als Namen erstellt werden, was alles andere als eine optimale Lösung ist.

L 28: Der Name eines Artikels ist in Ylvi auf hundert Zeichen beschränkt. Wenn ein *prefLabel* in der Standardsprache definiert ist, sollte dies zu keinen Problemen führen, da der Wert des *prefLabels* mit an Sicherheit grenzender Wahrscheinlichkeit aus weniger als hundert Zeichen bestehen wird. Falls der URI als Artikelname übernommen wird, könnte dieser jedoch mehr als 100 Zeichen enthalten. Deswegen muss der ermittelte Namen für einen zu erstellenden Artikel gegebenenfalls gekürzt werden.

L 29: Damit nach einem Export einer Ylvi-Instanz und einem darauffolgenden Import des exportierten Modells in eine andere Instanz der Artikelname rekonstruiert werden kann, muss beim Export ein zusätzliches Statement zur Beschreibung der Ressource hinzugefügt werden, wenn von den Benutzern kein *prefLabel* in der eingestellten Standardsprache im Artikel angegeben wurde. Somit wird der Name des korrespondierenden METIS-Objekts als *prefLabel* mit exportiert, damit dieser beim Import wiederhergestellt werden kann. Somit stimmt das in den Artikeln beschriebene SKOS-Modell nicht mehr ganz mit dem exportierten Modell überein, dennoch ist dies unbedingt notwendig, weil ansonsten L 27 (URI als Artikelname) zum Tragen kommen würde, was wesentlich gravierender ist als diese Limitation. Außerdem wird bei einem darauffolgenden Import dieses zusätzliche Statement nicht wieder in den Text des Artikels aufgenommen. Dies bedeutet, dass die Artikel nach einem Export und darauffolgenden Import diesbezüglich wieder identisch sind.

L 30: Zu Problemen kann es außerdem kommen, wenn ein Benutzer nicht den tatsächlichen Namen des Artikels als *prefLabel* in der Standardsprache wählt. Zum Beispiel könnte ein Benutzer im Artikel *Tier* ein *prefLabel* namens `Tiehr@de` formulieren. Bei einem Export dieses Artikels und darauffolgenden Import in eine andere Instanz, würde in dieser anderen Instanz ein Artikel namens *Tiehr* erstellt werden. Wird das exportierte Modell wieder in dieselbe Instanz importiert und lässt sich der jeweilige Artikel anhand der URI nicht mehr identifizieren, wären zwei Artikel im System vorhanden, welche dasselbe beschreiben, nämlich einer mit dem Namen *Tier* und einer namens *Tiehr*. Dies verstößt gegen DG 5 (*geringer Informationsverlust*), ist aber bei strenger Betrachtung eher ein Anwendungsfehler als ein Problem innerhalb des Systems selbst.

DG 2 (*Charakter eines traditionellen Wikis)* sagt aus, dass der Charakter eines traditionellen Wikis im System im Vordergrund stehen soll. Das bedeutet auch, dass es wichtig ist, informalen Text in einem Artikel zu erhalten. Deswegen müssen beim Export nicht nur die in SKOS formulierten Statements berücksichtigt werden, sondern auch der informale Text, in welchem diese Statements gebraucht werden. Ansonsten wäre diese informale Beschreibung nach einem darauffolgenden Import nicht mehr vorhanden. Ebenso problematisch wäre es bei einem Import einer in Ylvi bereits bestehenden Ressource. In diesem Fall würde kein neuer Artikel erstellt, sondern der bereits bestehende Artikel überschrieben werden. Würde man beim Export den informalen Inhalt nicht berücksichtigen, würde der informale Inhalt des bereits in Ylvi vorhandenen Artikels unwiderruflich gelöscht werden. Aus diesem Grund wird der gesamte Quellcode des Artikels als Wert eines *skos:definition*-Properties exportiert. Um dieses Property mit dem Quellcode des Artikels von tatsächlichen *skos:definition*-Properties unterscheiden zu können, wird im exportierten Modell die Zeichenkette „ArticleSource:“ vor dem eigentlichen Quellcode eingefügt. Zwar wird beim Import der Quellcode des bereits vorhandenen Artikels bei dieser Vorgehensweise ebenfalls überschrieben, allerdings ist es auf diese Weise möglich, den informalen Inhalt eines Artikels zum Zeitpunkt des Exports wiederherzustellen.

4.4 Import Plugin

4.4.1 Einführung

Das *Import Plugin* importiert extern vorliegende SKOS-Modelle und erstellt entsprechende Artikel für jede im Modell beschriebene Ressource. Dabei sollen die Benutzer einen URI in ein Textfeld eingeben können, von welchem ein SKOS-Modell referenziert wird. Dabei wird angenommen, dass nur valide SKOS-Modelle importiert werden. Dies bedeutet, in diesem Fall wird keine Validierung des zu importierenden Modells vorgenommen. Validiert werden die beim Import erstellten Artikel ohnehin, wenn versucht wird, diese in entsprechende METIS-Objekte umzuwandeln. Es ist möglich, der laufenden Instanz sowohl völlig unbekannte Ressourcen als auch im Wiki bereits vorhandene Ressourcen zu importieren.

4.4.2 Implementierungsdetails

Zuerst wird für jede im importierten Modell beschriebene Ressource überprüft, ob diese auch im lokalen SKOS-Modell enthalten ist, sofern ein SKOS-Modell in der lokalen Instanz überhaupt vorhanden ist. Wenn die zu importierende Ressource im Modell gefunden wird, ist man in der Lage, aufgrund des URIs das jeweilige Objekt rauszufiltern, da der URI beim Objekt hinterlegt ist. Ebenso ist die interne ID für einen Artikel hinterlegt, womit man den entsprechenden Artikel eindeutig identifizieren kann. Es könnte aber auch passieren, dass eine Ressource im lokalen Modell vorhanden ist, der zugehörige Artikel mittlerweile aber nicht mehr existiert.

Es könnte aber auch der umgekehrte Fall eintreten, also die Ressource im lokalen Modell nicht mehr vorhanden ist, der Artikel aber immer noch im System existiert. Dieser Fall tritt dann ein, wenn ein Artikel bei einer Erstellung eines lokalen Modells valide war und deshalb auch im Modell enthalten ist, der Artikel jedoch in weiterer Folge inkonsistent wurde und mittlerweile schon ein aktuelleres SKOS-Modell im System gespeichert wurde, worin der jeweilige Artikel nicht mehr enthalten ist.

Deshalb müssen alle Artikel durchgesucht und überprüft werden, ob einer von ihnen einen mit der Ressource übereinstimmenden URI definiert hat, wenn die Ressource nicht im lokalen Modell enthalten ist oder der zugehörige Artikel nicht mehr gefunden wurde.

L 31: Wenn eine Ressource im lokalen Modell nicht mehr enthalten ist, der korrespondierende Artikel aber noch im Wiki vorhanden ist und im Artikel selbst kein URI definiert wurde, kann der entsprechende Artikel nicht mehr identifiziert werden, weil weder ein entsprechendes METIS-Objekt vorhanden ist, in welchem die interne ID des Artikels gespeichert ist noch der Artikel anhand der URI identifiziert werden kann. In dieser Situation würde ein zweiter Artikel mit gleichen Namen erstellt werden, was nicht wünschenswert ist. Allerdings handelt es sich hier wiederum eher um einen Anwendungsfehler, weil die Benutzer nach Modifizierung eines Artikels eigentlich das SKOS-Modell im System erneut speichern sollten. Es sollte von den Benutzern auch generell darauf geachtet werden, dass alle Artikel im System konsistent formuliert sind.

Wird ein Artikel im Wiki identifiziert, hat man im Prinzip zwei verschiedene Ressourcen mit identischen URIs zur Verfügung, was eigentlich einen Konflikt darstellt, weil zwei Ressourcen mit demselben URI eigentlich auch dieselbe Ressource sein sollten. Somit wird der aktuelle im System vorhandene Artikel mit der Beschreibung aus dem

importierten Modell überschrieben. Wenn kein Artikel identifiziert werden konnte, wird ein neuer erstellt, welcher, wie bereits erwähnt, den Wert des *prefLabels* in der Standardsprache als Namen erhält.

Weiters muss bei jeder Ressource überprüft werden, ob ein *definition*-Property vorhanden ist, das den Quellcode eines Artikels als Wert beinhaltet. Wie bereits in Kapitel 4.3.2 erwähnt, wird der Quellcode eines Artikels ebenfalls exportiert, damit dieser beim Import wieder rekonstruiert werden kann. Ist kein Quellcode eines Artikels angegeben, werden einfach die im zu importierenden Modell formulierten Statements in den Artikel geschrieben. Ist ein Quellcode hingegen vorhanden, so muss dieser im zu erstellenden oder zu überschreibenden Artikel wiederhergestellt werden. In diesem Fall wäre es jedoch möglich, dass das Modell nach einem Export außerhalb verändert wird, was Inkonsistenzen zwischen den tatsächlichen Statements und dem Quellcode des Artikels im zu importierenden Modell zur Folge haben könnte. Zum Beispiel wäre es möglich, dass Statements zu einer Ressourcenbeschreibung hinzugefügt werden. Diese Statements sind zwar im eigentlichen Modell vorhanden, nicht aber in jenem *definition*-Property, welches den Quellcode eines Artikels enthält. In diesem Fall werden die Statements einfach am Ende des Quellcodes angehängt. Problematischer wird es, wenn der umgekehrte Fall eintritt, also ein Statement zwischen den Zeitpunkten des Exports und Imports aus dem eigentlichen Modell entfernt wird. In diesem Fall ist das Statement im Quellcode des Artikels immer noch enthalten, allerdings nicht mehr im eigentlichen Modell. Damit keine Inkonsistenzen entstehen, darf dieses Statement nicht in den Quellcode des erstellten oder überschriebenen Artikels übernommen werden. Weil die Statements zumeist jedoch innerhalb von Sätzen formuliert sind, kann man diese nicht einfach aus dem Quellcode löschen, weil in diesem Fall ein Wort aus einem Satz entfernt werden würde und somit die Bedeutung des ursprünglichen Inhalts verändert wird. Zum Beispiel könnte ein kurzer Ausschnitt des Quellcodes für den Artikel *Tiger* wie folgt aussehen:

```
Der ~prefLabel=Tiger~ ist ein in Asien verbreitetes [broader [Saeuge-
tier]]…
```

Wenn nun das *skos:broader* Statement aus dem exportierten Modell entfernt wird, darf es auch beim Import im zu erstellenden bzw. zu überschreibenden Artikel nicht berücksichtigt werden. Wie aus dem Beispiel ersichtlich, würde aber ein jegliches Löschen des Statements den Satz verändern. Zum Beispiel würde in diesem Satz das Wort *Saeugetier* fehlen. Dies wird unterbunden, indem zwar der Link erhalten bleibt, aber der Link-

typ aus dem Artikeltext entfernt wird. Im davor genannten Beispiel würde folgender Quellcode übrig bleiben:

```
Der ~prefLabel=Tiger~ ist ein in Asien verbreitetes [[Saeugetier]]…
```

Nun besteht zwar noch ein Link zum anderen Artikel, allerdings ist dies ein ganz gewöhnlicher Link ohne jegliche Bedeutung, wie man Links aus traditionellen Wikis kennt. Würde in diesem Beispiel das *skos:prefLabel*-Property aus dem Modell entfernt werden, würde man den gesamten Ausdruck (`~prefLabel=Tiger~`) durch den Attributwert ersetzen, sodass nur mehr der Ausdruck `Tiger` übrig bleiben würde. Auf diese Weise ist es möglich, den gesamten Inhalt eines Artikels, also auch den informalen Text, wiederherzustellen, ohne dass Inkonsistenzen in oder zwischen den Artikeln entstehen können. Inkonsistenzen können nur in Fällen entstehen, in welchen das zu importierende Modell selbst inkonsistent ist und wie bereits erwähnt, wird beim Import davon ausgegangen, dass nur valide SKOS-Modelle importiert werden.

L 32: Problematisch wird dies bei geordneten *Collections*. Man betrachte als Beispiel einen Artikel, welcher eine *OrderedCollection* repräsentiert. Diese Collection beinhaltet vier Ressourcen, nämlich *A*, *B*, *C* und *D*, in jener Reihenfolge, in welcher sie gerade eben erwähnt wurden. Diese *Collection* wird exportiert und außerhalb von Ylvi mit einem Editor verändert, sodass beim Import zwischen den Ressourcen *B* und *C* noch eine andere Ressource *E* in der *Collection* enthalten ist. Im Normalfall würde man dieses Statement einfach am Ende des Quellcodes hinzufügen, im Falle einer *OrderedCollection* ist dies jedoch nicht möglich, da dadurch die Reihenfolge der Ressourcen im Artikel nicht mehr mit der Reihenfolge in dem zu importierenden SKOS-Modell übereinstimmen würde. Da die Reihenfolge der enthaltenen Ressourcen bei einer *OrderedCollection* jedoch von Relevanz ist, würde somit auch nicht die tatsächliche Semantik dieser *Collection* importiert werden. Das neu zur *OrderedCollection* hinzugekommene Element kann auch nicht an der richtigen Stelle eingefügt werden, weil davon auszugehen ist, dass die Elemente der *Collection* im Artikel innerhalb von Sätzen formuliert sind. Würde man das neu dazugekommene Statement z.B. direkt nach *B* einfügen, würde dieses Statement mitten im Satz stehen, mit welchem die Ressource *B* beschrieben wird. Ebenso verhält es sich, wenn man das Statement direkt vor *C* einfügen wollte, weil auch hier das *member*-Statement für die Ressource *C* mitten in einem Satz vorkommen könnte und man somit den ursprünglichen informalen Inhalt des Artikels durcheinanderwirbeln würde. Um dies zu vermeiden, werden in einem solchen Fall alle Linktypen von

member-Links entfernt, sodass nur mehr gewöhnliche Links im Quellcode vorhanden sind und die gesamte *OrderedCollection* am Ende des Quellcodes hinzugefügt, damit auch die formalen Statements in den Artikel übernommen werden. Dies wird allerdings nur durchgeführt, wenn neue Ressourcen zwischen den ursprünglichen Ressourcen eingefügt worden sind oder sich die Reihenfolge der ursprünglichen Ressourcen geändert hat. Wenn Ressourcen am Ende einer *OrderedCollection* hinzugefügt werden, ist dies irrelevant, weil diese Statements am Ende des Quellcodes hinzugefügt werden können, ohne die Reihenfolge der Ressourcen oder den informalen Inhalt durcheinanderzubringen. Muss der Inhalt eines Artikels auf diese Weise modifiziert werden, werden die Benutzer beim Importbericht darüber informiert, dass sich der Inhalt eines bestimmten Artikels geändert hat, damit dieser gegebenenfalls sofort aktualisiert werden kann.

4.5 Beispiel

Anhand eines konkreten Beispiels sollte die Funktionalität des implementierten Systems verdeutlicht werden. In diesem Beispiel werden hauptsächlich Konzepte aus dem Tierreich beschrieben, weil diese sich sehr gut für die hierarchische Klassifizierung eignen. Diese Artikel zeigen nur die SKOS-relevanten Statements, allerdings sollten in der Praxis die SKOS-relevanten Statements innerhalb des informalen Textes definiert sein, also innerhalb von Sätzen. Um das implementierte System vollständig verstehen zu können, ist es also wichtig, davon auszugehen, dass die SKOS-Statements nicht isoliert im Artikel stehen, sondern selbst Teil des informalen Textes sind, wie bereits in Kapitel 4.4.2 beschrieben wurde.

Abbildung 12 zeigt alle in diesem Beispiel erstellten Ressourcen und deren Beziehungen zueinander. Die Vierecke, welche die verschiedenen Konzepte beinhalten, repräsentieren Konzeptschemen. Ellipsen stellen die einzelnen Konzepte dar und die gerichteten Pfeile zwischen diesen Ellipsen beschreiben die Beziehungen der Konzepte zueinander. Die im folgenden Beispiel erstellten *Collections* sind aus Gründen der besseren Übersichtlichkeit nicht in der Abbildung enthalten. Außerdem sind in der Abbildung nicht alle in den Artikeln formulierten Beziehungen sichtbar. So verzichtet man z.B. auf redundante *narrower*-Beziehungen, wenn die Semantik der hierarchischen Beziehung zwischen zwei Konzepten bereits über eine *broader*-Beziehung ausgedrückt ist.

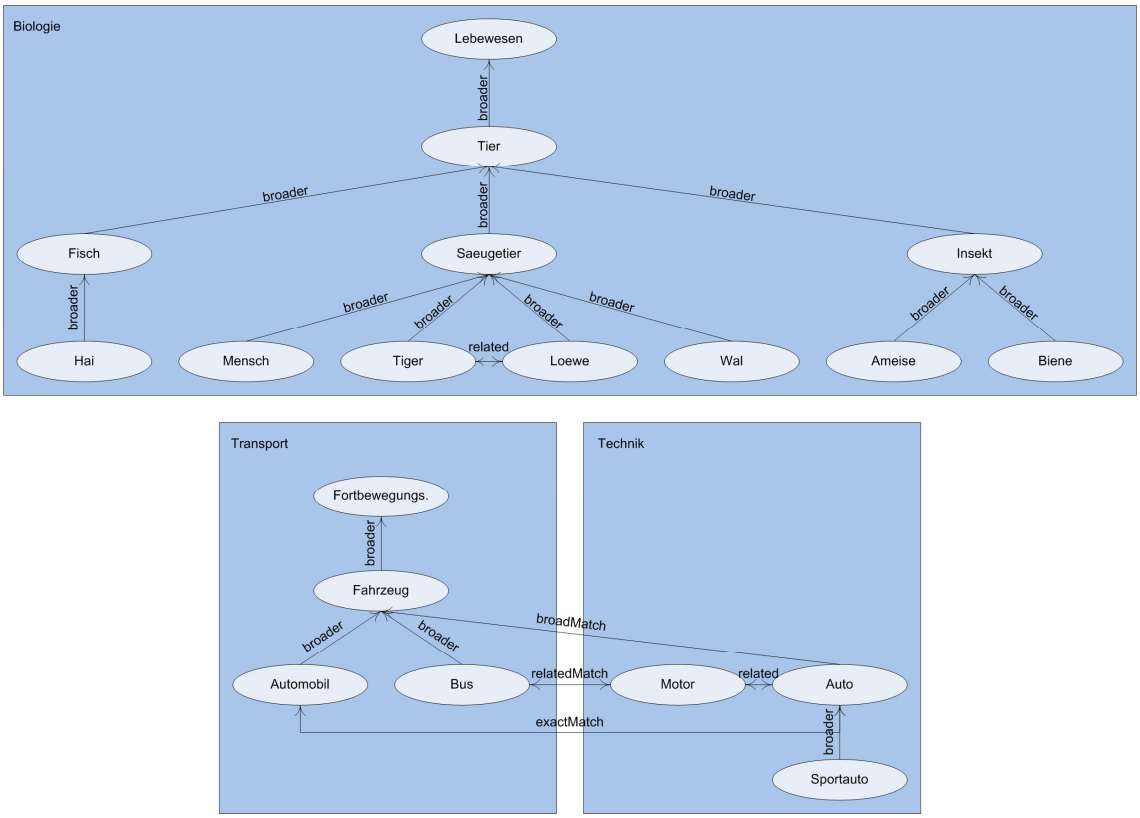

Abb. 12: Die einzelnen Konzepte des in diesem Kapitel erläuterten Beispiels und deren Beziehungen zueinander.

Als erstes wird ein Konzeptschema namens *Biologie* definiert, in welchem die später definierten Konzepte für die verschiedenen Tierarten enthalten sind. Dieses Konzeptschema formuliert weiters das Konzept *Lebewesen* als das höchste in der Hierarchie stehende Konzept von allen im Konzeptschema enthaltenen Konzepten.

```
<<ConceptScheme>>
~URI=http://www.example.com/Biologie~
[hasTopConcept [Lebewesen]]
```

In der ersten Zeile wird der Artikel als *ConceptScheme* typisiert. Danach folgt die Angabe des URIs, welche nicht zwingend, sondern optional ist. In der letzten Zeile wird die Beziehung zum Konzept *Lebewesen* ausgedrückt, welches eines von mehreren möglichen Konzepten ist, das kein Subkonzept eines anderen Konzepts im selben Konzeptschema darstellt. Der Artikel für das Konzept *Lebewesen* könnte zum Beispiel folgendermaßen aussehen:

```
<<Concept>>
~note=Dies ist ein Lebewesen-Konzept.~
[inScheme [Biologie]]
```

Hier wird der Artikel wiederum typisiert, diesmal jedoch als *Concept*. In der Beschreibung dieses Artikels ist kein URI explizit angegeben. In der zweiten Zeile ist ein Do-

cumentation Property definiert, und zwar das Property *note*. Danach ist die Assoziation formuliert, dass sich das Konzept *Lebewesen* im Konzeptschema *Biologie* befindet.

Nun wird ein Konzept namens *Tier* erstellt, welches ein Subkonzept von *Lebewesen* ist:

```
<<Concept>>
~prefLabel=Tier@de~
~altLabel=Vieh@de~
[broader [Lebewesen]]
[inScheme [http://www.example.com/Biologie]]
```

Hier ist auch explizit ein *prefLabel* für das Konzept vergeben worden, außerdem auch noch ein alternativer Ausdruck für das Konzept, nämlich Vieh. In der vierten Zeile ist die Definition der hierarchischen Beziehung zwischen den Konzepten *Tier* und *Lebewesen* formuliert. In diesem Fall ist *Tier* ein Subkonzept von *Lebewesen*. Dies bedeutet, dass jedes Tier auch ein Lebewesen ist, aber nicht umgekehrt, denn nicht jedes Lebewesen ist auch ein Tier, wie zum Beispiel eine Pflanze. Im letzten Statement wird wieder eine Beziehung zum Konzeptschema *Biologie* definiert, diesmal wird allerdings der URI des Konzeptschemas *Biologie* als Objekt des Statements verwendet. Diese Angabe ist genauso gültig wie die Angabe des Artikelnamens und ist äquivalent zu `[inScheme [Biologie]]`. Ein URI als Objekt eines Statements kann in Ylvi zum Beispiel verwendet werden, um zwei Artikel mit demselben Namen voneinander zu unterscheiden. Wäre in diesem Fall ein zweiter Artikel namens *Biologie* definiert, wäre das Statement `[inScheme [Biologie]]` nicht eindeutig. Durch die Angabe des URIs wird das Ziel des Links eindeutig bestimmt. Allerdings sollte von den Benutzern generell vermieden werden, mehrere Artikel mit demselben Namen zu erstellen. Auch die Angabe des URIs als Objekt eines Statements sollte vermieden werden, wenn die Angabe nicht zur eindeutigen Identifikation eines Artikels notwendig ist. Eine weitere sinnvolle Verwendung von URIs als Objekte von Statements wäre der Fall, in dem der URI auch tatsächlich im informalen Text des Artikels vorkommen sollte, der Name des zu referenzierenden Artikels jedoch nicht.

Als nächstes wird ein Artikel erstellt, der das Konzept *Fisch* beschreiben soll, welches wiederum ein Subkonzept von *Tier* ist:

```
<<Concept>>
~prefLabel=Fisch@de~
~hiddenLabel=Fish@de~
~hiddenLabel=Fiesch@de~
~hiddenLabel=Visch@de~
[inScheme [id:79]]
[broader [Tier]]
```

In diesem Artikel werden auch zwei *hiddenLabels* definiert, welche verwendet werden, um eine Ressource in einem KOS dennoch auffinden zu können, falls ihr Name bei einer textbasierten Suche falsch geschrieben wurde. Hier sind drei mögliche falsche Schreibweisen von Fisch angegeben, nämlich Fish, Fiesch und Visch. Auch dieses Konzept wird wieder dem Konzeptschema *Biologie* zugeordnet, allerdings wird in diesem Fall die dritte Möglichkeit zur Angabe eines Linkziels verwendet, nämlich eine interne ID. In der lokalen Instanz von Ylvi ist 79 die interne ID des *CMOs*, welches den Artikel *Biologie* repräsentiert. Auch diese Möglichkeit zur Angabe eines Ziels wurde dafür entwickelt, um zwei Artikel mit demselben Namen voneinander unterscheiden zu können.

L 33: Zwei Artikel, welche dieselbe Ressource repräsentieren sollten, besitzen in zwei unterschiedlichen Instanzen mit an Sicherheit grenzender Wahrscheinlichkeit nicht dieselbe ID. Somit könnte die tatsächliche Ressource als Ziel eines Links nicht rekonstruiert werden, wenn die Artikel aus einer Instanz exportiert und in eine andere Instanz importiert werden. Aus diesem Grund werden Ziele für Links, für welche interne IDs verwendet werden, im exportierten Quellcode des Artikels durch den URI der jeweiligen Ressource im exportierten Modell ersetzt und gekennzeichnet. Bei einem darauffolgenden Import in eine andere Instanz kann eine durch einen URI ersetzte ID mit Hilfe der Kennzeichnung wieder identifiziert werden und durch die jeweilige ID innerhalb der importierenden Instanz wieder ersetzt werden, sodass die Angabe des Linkziels als interne ID im Quelltext des Artikels wieder rekonstruiert werden kann. Die Angabe von internen IDs sollte von den Benutzern allerdings gänzlich vermieden werden, weil zur eindeutigen Identifikation im Falle von mehreren Artikeln mit demselben Namen auch die URIs der jeweiligen Ressourcen verwendet werden können, welche somit in mehreren unterschiedlichen Instanzen eindeutig identifiziert werden könnten. Die Angabe von internen IDs war allerdings schon eine implementierte Funktionalität von Ylvi, bevor die Anwendung mit dem hier beschriebenen SKOS-Editor erweitert wurde, sodass interne IDs auch im implementierten System genutzt werden können. Im Falle des entwickelten SKOS-Editors sind interne IDs daher eigentlich überflüssig.

Nun wird noch ein Konzept namens *Insekt* erstellt, welches ebenfalls ein Subkonzept von *Tier* darstellt:

```
<<Concept>>
~URI=http://www.example.com/Insekt~
~notation=Insekt~
```

```
[inScheme [Biologie]]
[broader [Tier]]
```

Hier wird auch eine Notation für dieses Konzept definiert. Um DG 1 (*Einfachheit*) zu unterstützen, ist eine Angabe eines Datentyps in Ylvi nicht unbedingt erforderlich, in SKOS ist diese Angabe jedoch zwingend. Wird kein Datentyp explizit von den Benutzern angeben, wird beim Export des SKOS-Modells automatisch der Datentyp *String* für die jeweilige Notation definiert. Bei einem darauffolgenden Import wird der Datentyp der Notation im Quellcode des Artikels jedoch wieder weggelassen, um DG 5 (*geringer Informationsverlust*) zu unterstützen.

Es wird noch ein Subkonzept von *Tier* definiert, nämlich das Konzept *Saeugetier*:

```
<<Concept>>
~URI=http://www.example.com/Saeugetier~
~example=Auch ein Mensch ist ein Säugetier.~
[inScheme [Biologie]]
[broader [Tier]]
[narrower [Mensch]]
```

In diesem Konzept ist auch ein Documentation Property definiert, und zwar das Property *example*. Außerdem ist eine *narrower*-Beziehung zum Konzept *Mensch* angegeben, welche aussagt, dass das Konzept *Saeugetier* ein Superkonzept von *Mensch* ist, was bedeutet, dass jeder Mensch auch ein Säugetier ist. Dieses Statement ist eigentlich für die Semantik des gesamten Wissensmodells redundant, wenn im Artikel für das Konzept *Mensch* eine *broader*-Beziehung zu *Saeugetier* definiert ist. In den davor beschriebenen Artikeln wurde nie ein *narrower*-Statement formuliert, dieses ist aber implizit vorhanden, weil es das inverse Property von *broader* darstellt. In der Praxis sollte man sowohl die *broader*-Statements als auch die inversen *narrower*-Statements in den entsprechenden Konzepten formulieren, damit die Semantik in beiden Konzepten sofort ersichtlich ist. Wäre nur eine der beiden Beziehungen formuliert, so wäre die Semantik nur in einem der beiden Konzepte sichtbar. In dieser Arbeit wird jedoch in vielen Fällen auf die Formulierung beider Statements verzichtet, weil dies für das Verständnis des Beispiels auch nicht notwendig ist.

Ein Subkonzept von *Saeugetier*, nämlich das Konzept *Mensch*, könnte wie folgt aussehen:

```
<<Concept>>
~definition=Der Mensch ist eine intelligente und hoch entwickelte Le-
bensform, welche den Planeten Erde bewohnt.~
~prefLabel=Mensch@de~
~altLabel=homo sapiens@la~
```

```
[inScheme [Biologie]]
[broader [Saeugetier]]
```

Hier ist eine Beschreibung des Konzepts in einem Documentation Property gegeben, nämlich im Property *definition*. Außerdem werden ein *prefLabel* und ein *altLabel* definiert, nämlich Mensch in deutscher Sprache und der korrespondierende lateinische Ausdruck als Alternative.

Der Artikel *Ameise* repräsentiert wiederum ein Konzept, welches ein Subkonzept von *Insekt* darstellt:

```
<<Concept>>
[inScheme [Biologie]]
[broader [Insekt]]
```

Es wird noch ein Artikel für ein anderes Insekt erstellt, nämlich für das Konzept *Biene*:

```
<<Concept>>
~prefLabel=Biene@de~
[inScheme [Biologie]]
[broader [Insekt]]
[related [Ameise]]
```

Wie aus der Artikelbeschreibung ersichtlich, ist auch *Biene* ein Subkonzept von *Insekt*. Ameisen und Bienen sind aber noch spezifischer miteinander verwandt, weil beide Tierarten zu einer bestimmten Ordnung der Insekten gehören, nämlich zu den Hautflüglern. Diese Beziehung wird, wie im Quellcode des Artikels ersichtlich, mit einem *related*-Statement formuliert. Da *related* ein symmetrisches Property ist, bedeutet dies implizit, dass auch Ameisen mit Bienen verwandt sind. Dies bedeutet, ein Statement der Form `[related [Biene]]` im Artikel *Ameise* wäre, wie bei *broader*- und *narrower*-Beziehungen, redundante Information bezüglich der Semantik.

Nachfolgend wird ein weiterer Artikel erstellt, der das Konzept *Hai* beschreiben soll, welches ein Subkonzept von *Fisch* darstellt, da Haie bekannterweise auch Fische sind:

```
<<Concept>>
~URI=http://www.example.com/Hai~
~prefLabel=Hai~
~notation=185^^int~
[inScheme [Biologie]]
[broader [Fisch]]
```

In dieser Artikelbeschreibung wird ein *prefLabel* ohne Language-Tag angegeben. Dieser muss nämlich nicht verpflichtend angegeben werden. Außerdem ist wieder eine Notation definiert, diesmal mit Datentyp *Integer* und dem Wert 185. Dies könnte zum Beispiel bedeuten, dass die Ressource bereits in irgendeiner Form in einem anderen System

vorhanden ist und in diesem anderen System mit der Zahl 185 eindeutig identifiziert wird.

Nun wird ein Konzept *Loewe* erstellt, welches ein Subkonzept von *Saeugetier* darstellt:

```
<<Concept>>
~URI=http://www.example.com/Loewe~
~changeNote=Am 30. August verändert, broader-Beziehung zu Tier wurde
durch broader-Beziehung zu Saeugetier ersetzt.~
[inScheme [Biologie]]
[broader [Saeugetier]]
[related [Tiger]]
```

In diesem Artikel wird ein weiteres Documentation Property verwendet, nämlich das Property *changeNote*. Weil Löwen und Tiger Katzen und somit miteinander verwandte Tiere sind, besteht außerdem eine assoziative Beziehung zum Konzept *Tiger*, welches im folgenden Artikel beschrieben wird:

```
<<Concept>>
[inScheme [Biologie]]
[broader [Saeugetier]]
[related [Loewe]]
```

Auch hier ist die assoziative Beziehung zum Konzept *Loewe* explizit angegeben. Aber auch diese Beziehung wäre ohnehin implizit im Wissensmodell vorhanden, weil *related* als symmetrisches Property definiert ist.

Es wird noch ein Artikel *Wal* erstellt, der ein Konzept repräsentiert, welches ein Subkonzept von *Saeugetier* darstellt:

```
<<Concept>>
~prefLabel=Wal@de~
~prefLabel=whale@en~
[inScheme [Biologie]]
[broader [Saeugetier]]
[narrower [Delphin]]
```

In diesem Artikel sind zwei *prefLabels* definiert, eines in Deutsch und eines in Englisch. Außerdem ist hier wieder explizit eine *narrower*-Beziehung zum Konzept *Delphin* angegeben, welches wie folgt beschrieben wird:

```
<<Concept>>
~prefLabel=dolphin@en~
~prefLabel=Delphin~
[inScheme [Biologie]]
[broader [Wal]]
```

Dieses Konzept für einen *Delphin* ist ein Subkonzept von *Wal*, was bedeutet, dass jeder Delphin auch ein Wal ist.

Es wird auch ein Artikel namens *Tiere* erstellt, der eine *Collection* repräsentieren soll, welche alle soeben definierten Konzepte beinhaltet:

```
<<Collection>>
~URI=http://www.example.com/Tiere~
[member [Tier]]
[member [Fisch]]
[member [Insekt]]
[member [Saeugetier]]
[member [Ameise]]
[member [Biene]]
[member [Hai]]
[member [Loewe]]
[member [Tiger]]
[member [Wal]]
[member [Delphin]]
```

Dies ist eine ungeordnete *Collection*, da kein Attribut *ordered* mit dem Attributwert `true` angegeben wurde. Dies bedeutet, in diesem Fall spielt die Reihenfolge der Konzepte keine Rolle für die Semantik dieser *Collection*.

Nun werden noch zwei andere Konzeptschemen definiert, nämlich die Konzeptschemen *Technik* und *Transport*. Diese Konzeptschemen beinhalten beide hauptsächlich Konzepte, welche verschiedene Arten eines Fahrzeugs repräsentieren. Nachfolgend wird das Konzeptschema *Technik* beschrieben:

```
<<ConceptScheme>>
~URI=http://www.example.com/Technik~
[hasTopConcept [Auto]]
```

Wie aus dem Artikel ersichtlich, ist in diesem Konzeptschema *Auto* eines der möglichen Konzepte, welche in der Hierarchiekette ganz oben stehen.

Das Konzeptschema *Transport* sieht dabei ganz ähnlich aus:

```
<<ConceptScheme>>
[hasTopConcept [Fortbewegungsmittel]]
```

In diesem Konzeptschema ist eine *hasTopConcept*-Beziehung zum Konzept *Fortbewegungsmittel* definiert.

Nun wird ein Konzept namens *Motor* erstellt, welches im Konzeptschema *Technik* enthalten ist:

```
<<Concept>>
~URI=http://www.example.com/Motor~
~prefLabel=Motor@de~
[inScheme [Technik]]
```

Der Artikel, welcher das Konzept für ein *Fortbewegungsmittel* repräsentiert, wird wie folgt formuliert:

```
<<Concept>>
~URI=http://www.example.com/Fortbewegungsmittel~
~prefLabel=Fortbewegungsmittel@de~
[inScheme [Transport]]
[narrower [Fahrzeug]]
```

Dieses Konzept ist im Konzeptschema *Transport* enthalten und ist ein Superkonzept des Konzepts *Fahrzeug*, welches folgendermaßen beschrieben wird:

```
<<Concept>>
~URI=http://www.example.com/Fahrzeug~
~prefLabel=Fahrzeug@de~
~prefLabel=Foazeig@de-AT~
[inScheme [Transport]]
[broader [Fortbewegungsmittel]]
[narrowMatch [Auto]]
```

In diesem Artikel ist auch ein *prefLabel* im österreichischen Dialekt angegeben. Es können also auch regionale Ausprägungen von Sprachen als Language-Tag angegeben werden. Die *broader*-Beziehung zum Konzept *Fortbewegungsmittel* ist auch hier explizit angegeben, obwohl diese Information bereits durch das *narrower*-Statement im Konzept *Fortbewegungsmittel* ausgedrückt ist. In dieser Beschreibung wird auch ein Mapping Property verwendet, nämlich das Property *narrowMatch*, welches eine hierarchische Beziehung zum Konzept *Auto* darstellt. Das Konzept *Auto* befindet sich allerdings in einem anderen Konzeptschema, weswegen das Property *narrowMatch* verwendet wird. Der Artikel für das Konzept *Auto* ist wie folgt beschrieben:

```
<<Concept>>
~URI=http://www.example.com/Auto~
~prefLabel=Auto@de~
[inScheme [Technik]]
[broadMatch [Fahrzeug]]
[narrower [Sportauto]]
[related [Motor]]
```

Auch hier ist wieder explizit die *broadMatch*-Beziehung zu *Fahrzeug*, welches in einem anderen Konzeptschema enthalten ist, definiert, was bedeutet, dass jedes Auto auch ein Fahrzeug ist. Weil jedes Auto von einem Motor angetrieben wird, ist in der letzten Zeile des Artikels eine assoziative Beziehung zum Konzept *Motor* definiert. Weiters ist *Auto* ein Superkonzept von *Sportauto*, was wiederum bedeutet, dass jedes Sportauto auch ein Auto ist, nicht aber umgekehrt. Der Artikel für das Konzept *Sportauto* ist nachfolgend beschrieben:

```
<<Concept>>
~URI=http://www.example.com/Sportauto~
~prefLabel=Sportauto@de~
~editorialNote=Bitte direkt hierarchisch verwandte Konzepte aktuali-
sieren, falls dieses Konzept gelöscht wird!~
[inScheme [Technik]]
[broader [Auto]]
```

Hier wird wiederum ein Documentation Property verwendet, und zwar das Property *editorialNote*, welches zur Beschreibung wichtiger Hinweise für Editoren dieses Konzepts dient.

Der nachfolgende Artikel beschreibt einen weiteren Typ eines Fahrzeugs, und zwar das Konzept *Bus*, welches, wie Auto, ein Subkonzept von *Fahrzeug* darstellt und im Konzeptschema *Transport* enthalten ist:

```
<<Concept>>
~prefLabel=Bus@de~
[inScheme [Transport]]
[broader [Fahrzeug]]
[relatedMatch [Motor]]
```

Die letzte Zeile dieses Artikels definiert wiederum eine assoziative Beziehung zum Konzept *Motor*, weil auch jeder Bus von einem Motor angetrieben wird. Allerdings befindet sich das Konzept *Bus* nicht im selben Konzeptschema wie das Konzept *Motor*, deshalb wird in diesem Fall das Property *relatedMatch* verwendet.

Nun wird noch ein Artikel für das Konzept *Automobil* erstellt, welches eigentlich dasselbe beschreiben soll wie das Konzept *Auto*, sich aber in einem anderen Konzeptschema, nämlich in *Transport*, befindet:

```
<<Concept>>
~prefLabel=Automobil@de~
~prefLabel=automobile@en~
[inScheme [Transport]]
[exactMatch [Auto]]
```

Zu guter Letzt wird wieder eine *Collection* namens *Autos* erstellt, welche verschiedene Typen von Autos beinhaltet.

```
<<Collection>>
~URI=http://www.examples.com/Autos~
~ordered=true~
[member [Auto]]
[member [Sportauto]]
```

Hier handelt es sich um eine geordnete *Collection*, weil diese mit Hilfe des Attributs *ordered* und dem Wert `true` als eine solche definiert wird. Diese *Collection* enthält nur

zwei Konzepte, nämlich *Auto* und *Sportauto*, wobei die Reihenfolge dieser Konzepte für die Semantik der *Collection* relevant ist.

Nun haben die Benutzer die Möglichkeit, all diese Artikel mit Hilfe des in Kapitel 4.2 beschriebenen *Conversion Plugins* in korrespondierende METIS-Objekte umzuwandeln. Abbildung 13 zeigt einen Screenshot der METIS-Konsole.

Abb. 13: Die MediaTypeCategory Biologie und die in ihr enthaltenen MediaTypes, welche hauptsächlich verschiedene Konzepte von Tierarten repräsentieren.

Hier sieht man eine *MediaTypeCategory* namens *Biologie*, welche den Artikel für das Konzeptschema *Biologie* repräsentiert. In dieser Kategorie stehen 13 *MediaTypes*, welche jene Konzepte darstellen, die in das Konzeptschema *Biologie* eingeordnet wurden, also die verschiedenen Tierarten.

In Abbildung 14 sind die restlichen erstellten METIS-Objekte ersichtlich. In der Kategorie *SKOS Collections* stehen zwei *MediaTypes*, nämlich *Autos* und *Tiere*, welche die beiden davor definierten *Collections* repräsentieren. Es wurden noch zwei weitere *MediaTypeCategories* aus den jeweiligen als *ConceptScheme* typisierten Artikeln erstellt, nämlich *Technik* und *Transport*. In beiden Kategorien stehen noch weitere *MediaTypes*, welche die korrespondierenden Konzepte darstellen.

Diese *MediaTypes* können nun selbst verwendet werden, um Artikel zu typisieren, wenn konkrete Instanzen von Konzepten beschrieben werden sollen. In unserem Fall werden also hauptsächlich Artikel erstellt, welche konkrete Instanzen von verschiedenen Tierarten darstellen. In den folgenden Beispielen handelt es sich dabei um berühmt gewordene Tiere, hauptsächlich aus TV- oder Kinoproduktionen. Zum Beispiel erstellen wir einen Artikel namens *Ferdy die Ameise*, welcher die gleichnamige Zeichentrickserie beschreibt. Dabei handelt es sich weniger streng betrachtet auch um eine Ameise, also wird der Artikel als *Ameise* typisiert.

```
<<Ameise>>
Ferdy die Ameise ist eine Zeichentrickserie…
```

Wie man sieht, wird der durch die Konvertierung erstellte *MediaType Ameise* zur weiteren Typisierung analog zu den bereits vordefinierten SKOS-Klassen *Concept*, *ConceptScheme* und *Collection* verwendet. Außerdem werden folgende Artikel auf dieselbe Weise erstellt: *Biene Maja*, welche logischerweise als *Biene* typisiert wird, ebenso aus „Biene Maja“ *Flip* (Heuschrecke) als *Insekt*, *Flipper* als *Delphin*, *Nemo* als *Fisch*, *Der weiße Hai* als *Hai*, *Simba* aus „König der Löwen“ als *Loewe*, *Albert Einstein* als *Mensch*, *Shir Khan* aus dem „Dschungelbuch“ als *Tiger*, *Moby Dick* als *Wal* und *Lassie* als *Saeugetier*, weil kein eigenes Konzept *Hund* zur Verfügung steht. Weiters werden Artikel für Instanzen von Konzepten aus den anderen Konzeptschemen erstellt, wie *Airbus A380* als *Fortbewegungsmittel* (da kein eigener *MediaType Flugzeug* vorhanden ist), *Kettler Scorpion* (Fahrrad) als *Fahrzeug*, *Trabant* als *Auto*, *Porsche 911* als *Sportauto* und *Temsa Safari* als eine Instanz des Konzepts *Bus*.

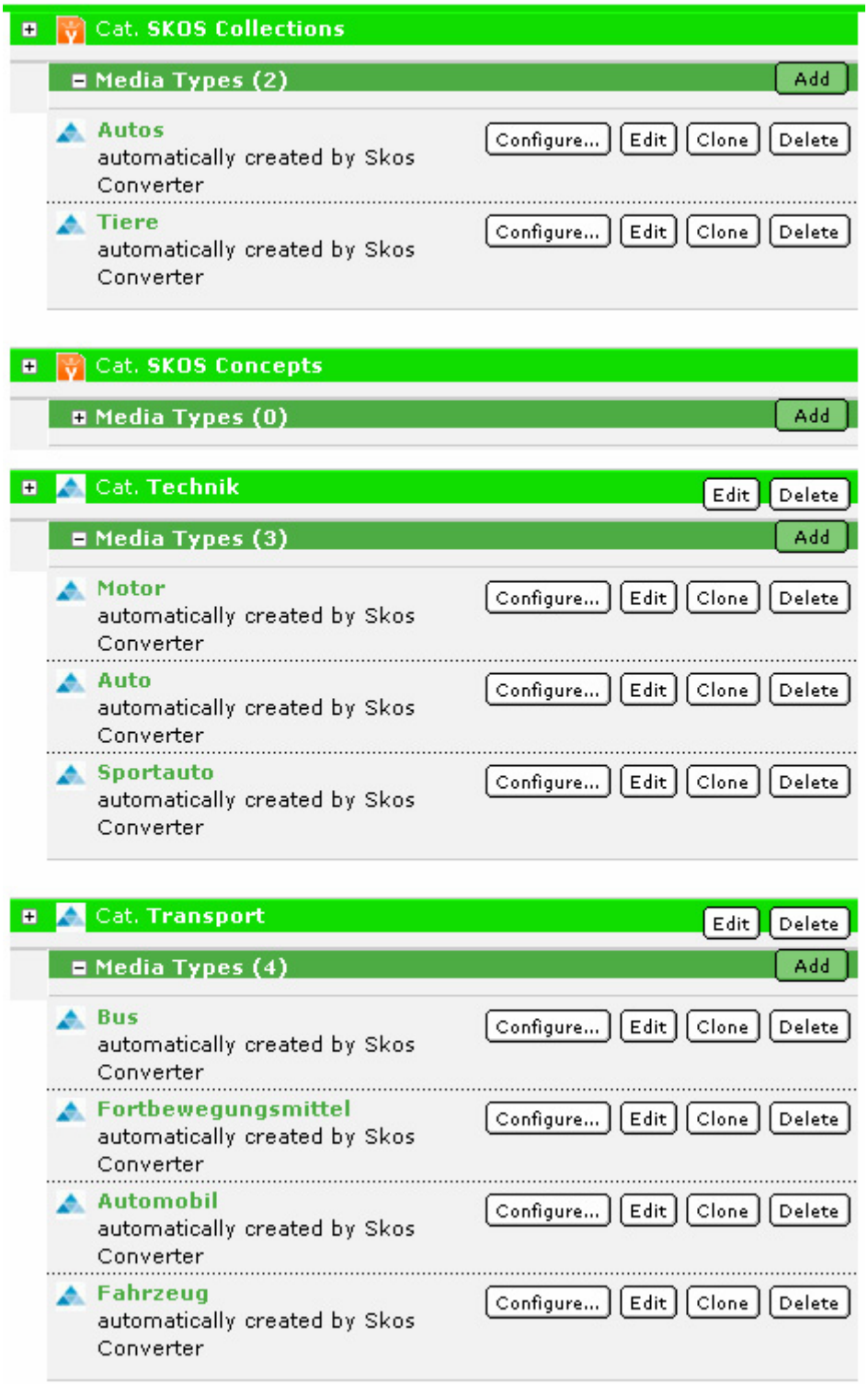

Abb. 14: Die restlichen erstellten METIS-Objekte (Die beiden Collections, welche als MediaTypes repräsentiert werden, die beiden MediaTypeCategories Technik und Transport und ihre zugehörigen MediaTypes).

Abbildung 15 zeigt die typenbasierte Suche im semantischen Wiki Ylvi. Das Navigationsmenü ganz links zeigt alle Instanzen, welche Säugetiere sind. Da ist natürlich auch *Albert Einstein* vertreten, weil auch Menschen streng betrachtet Säugetiere sind. Das nächste Menu zeigt die Suche nach allen Insekten, daneben sind alle Instanzen des

Konzepts *Fisch* aufgelistet und das letzte Navigationsmenü zeigt die Suche nach allen Walen. Wie aus Abbildung 15 ersichtlich, werden auch alle Artikel, welche konkrete Instanzen von Konzepten darstellen, angezeigt, selbst wenn sie nicht explizit als jenes Tier typisiert wurden, nach welchem gesucht wurde, sondern eine direkte oder indirekte hierarchische Beziehung zur gesuchten Tierart besitzen. So wird zum Beispiel nicht nur *Nemo* aufgelistet, wenn man nach Instanzen des Konzepts *Fisch* sucht, sondern auch *Der Weiße Hai*, weil Haie eben auch Fische sind.

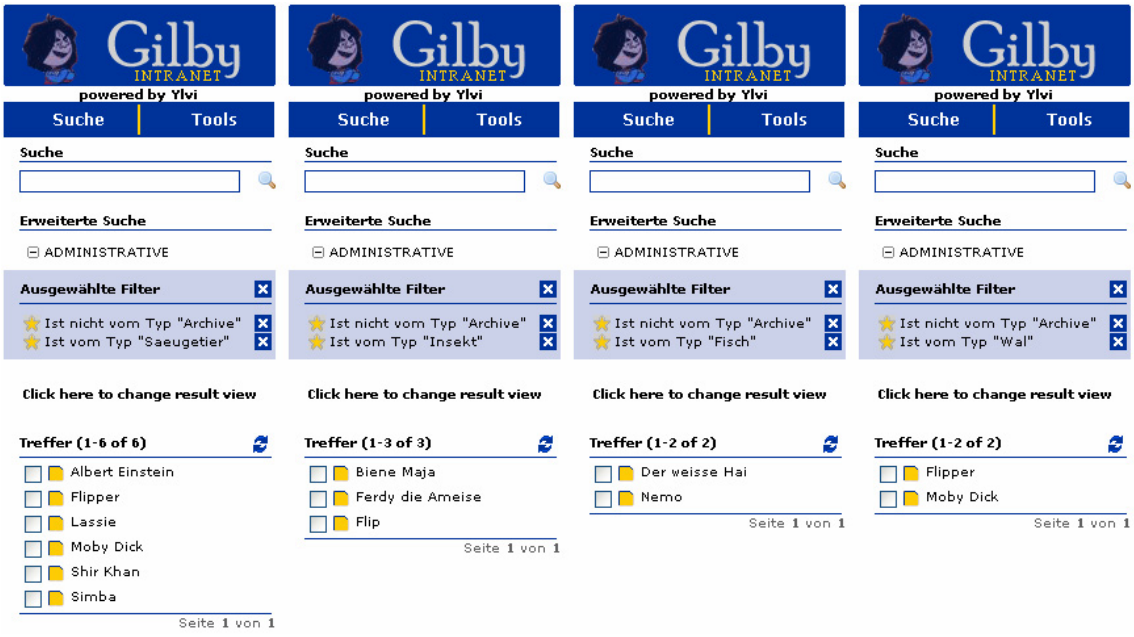

Abb. 15: Die typenbasierte Suche im semantischen Wiki Ylvi. Hier wird nach Instanzen verschiedener Tierarten gesucht.

Ähnlich verhält es sich bei den Artikeln, welche Instanzen von Konzepten aus den anderen Konzeptschemen darstellen. Abbildung 16 soll dies wiederum verdeutlichen. Das erste Navigationsmenü zeigt die Suche nach allen Instanzen des Typs *Fortbewegungsmittel*, das nächste Menü die Instanzen, welche vom Typ *Fahrzeug* sind, dann sind Instanzen des Konzepts *Auto* aufgelistet und die letzte Abbildung zeigt die Suche nach Sportautos. Wie in den Abbildungen ersichtlich ist, reduziert sich die Liste der Artikel, je weiter man die Suche nach Typen die Hierarchieleiter entlang spezialisiert.

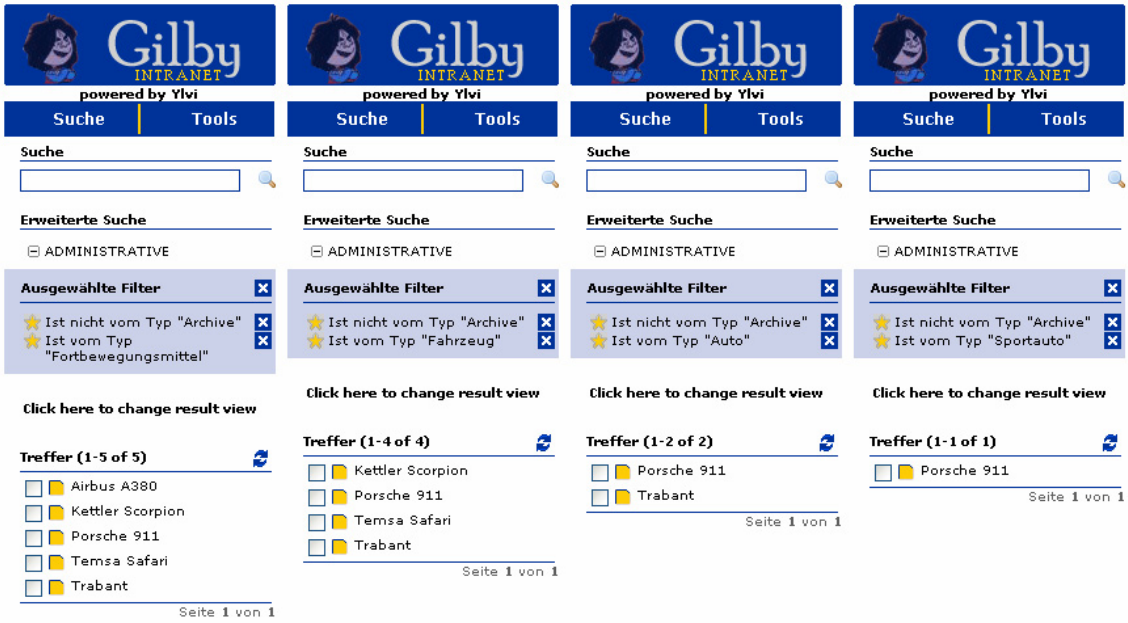

Abb. 16: Die typenbasierte Suche im semantischen Wiki Ylvi. Hier wird nach Instanzen verschiedener Fortbewegungsmittel gesucht.

Nach der Umwandlung der Artikel in die jeweiligen METIS-Objekte haben die Benutzer die Möglichkeit, das aktuelle SKOS-Modell in serialisierter Form zu speichern. Dieses Modell kann dann von anderen Ylvi-Instanzen importiert werden. Ein Ausschnitt aus der aus den Artikeln generierten Ressourcenbeschreibung würde in TURTLE wie folgt aussehen:

```
@prefix skos:  <http://www.w3.org/2008/05/skos#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://www.example.com/Biologie>
   rdf:type skos: ConceptScheme ;
   skos:definition "ArticleSour-
ce:\n<<ConceptScheme>>\n~URI=http://www.example.com/Biologie~\n[hasTop
Concept [Lebewesen]]" ;
   skos:hasTopConcept <http://www.skosylvi.com/Lebewesen> ;
   skos:prefLabel "Biologie"@de .

<http://www.skosylvi.com/Lebewesen>
   rdf:type skos:Concept ;
   skos:definition "ArticleSource:\n<<Concept>>\n~note=Dies ist ein
Lebewesen-Konzept.~\n[inScheme [Biologie]]" ;
   skos:inScheme <http://www.example.com/Biologie> ;
   skos:note "Dies ist ein Lebewesen-Konzept." ;
   skos:prefLabel "Lebewesen"@de .

<http://www.skosylvi.com/Tier>
   rdf:type skos:Concept ;
   skos:altLabel "Vieh"@de ;
   skos:broader <http://www.skosylvi.com/Lebewesen> ;
   skos:definition "ArticleSour-
ce<<Concept>>\n~prefLabel=Tier@de~\n~altLabel=Vieh@de~\n[broader [Le-
bewesen]]\n[inScheme [http://www.example.com/Biologie]]" ;
```

```
   skos:inScheme <http://www.example.com/Biologie> ;
   skos:prefLabel "Tier"@de .

<http://www.skosylvi.com/Fisch>
   rdf:type skos:Concept ;
   skos:broader <http://www.skosylvi.com/Tier> ;
   skos:definition "ArticleSour-
ce:\n<<Concept>>\n~prefLabel=Fisch@de~\n~hiddenLabel=Fish@de~\n~hidden
Label=Fiesch@de~\n~hiddenLabel=Visch@de~\n[inScheme
[*IDhttp://www.example.com/Biologie*ID]]\n[broader [Tier]]" ;
   skos:hiddenLabel "Visch"@de , "Fish"@de , "Fiesch"@de ;
   skos:inScheme <http://www.example.com/Biologie> ;
   skos:prefLabel "Fisch"@de .

<http://www.example.com/Insekt>
   rdf:type skos:Concept ;
   skos:broader <http://www.skosylvi.com/Tier> ;
   skos:definition "ArticleSour-
ce:\n<<Concept>>\n~URI=http://www.example.com/Insekt~\n~notation=Insek
t~\n[inScheme [Biologie]]\n[broader [Tier]]" ;
   skos:inScheme <http://www.example.com/Biologie> ;
   skos:notation "Insekt"^^<http://www.w3.org/2001/XMLSchema#string> ;
   skos:prefLabel "Insekt"@de .

<http://www.example.com/Saeugetier>
   rdf:type skos:Concept ;
   skos:broader <http://www.skosylvi.com/Tier> ;
   skos:definition "ArticleSour-
ce:\n<<Concept>>\n~URI=http://www.example.com/Saeugetier~\n~example=Au
ch ein Mensch ist ein Saeugetier.~\n[inScheme [Biologie]]\n[broader
[Tier]]\n[narrower [Mensch]]" ;
   skos:example "Auch ein Mensch ist ein Saeugetier." ;
   skos:inScheme <http://www.example.com/Biologie> ;
   skos:narrower <http://www.skosylvi.com/Mensch> ;
   skos:prefLabel "Saeugetier"@de .

<http://www.example.com/Tiere>
   rdf:type skos:Collection ;
   skos:definition "ArticleSour-
ce:\n<<Collection>>\n~URI=http://www.example.com/Tiere~\n[member
[Tier]]\n[member [Fisch]]\n[member [Insekt]]\n[member [Saeuge-
tier]]\n[member [Ameise]]\n[member [Biene]]\n[member [Hai]]\n[member
[Loewe]]\n[member [Tiger]]\n[member [Wal]]\n[member [Delphin]]" ;
   skos:member <http://www.skosylvi.com/Tier> ,
<http://www.skosylvi.com/Tiger> , <http://www.skosylvi.com/Wal> ,
<http://www.example.com/Loewe> , <http://www.skosylvi.com/Biene> ,
<http://www.skosylvi.com/Fisch> , <http://www.example.com/Hai> ,
<http://www.example.com/Saeugetier> , <http://www.example.com/Insekt>
, <http://www.skosylvi.com/Ameise> , <http://www.skosylvi.com/Delphin>
;
   skos:prefLabel "Tiere"@de .

<http://www.example.com/Autos>
   rdf:type skos:OrderedCollection ;
   skos:definition "ArticleSour-
ce:\n<<Collection>>\n~URI=http://www.example.com/Autos~\n~ordered=true
~\n[member [Auto]]\n[member [Sportauto]]" ;
   skos:memberList (<http://www.example.com/Sportauto>
<http://www.example.com/Auto>);
   skos:prefLabel "Autos"@de .
```

Als erstes wird das Konzeptschema *Biologie* beschrieben. Wie ersichtlich, wurde für diese Ressource von den Benutzern ein URI explizit angegeben, nämlich `http://www.example.com/Biologie`. Da kein *prefLabel* explizit im Artikel angegeben wurde, wird dieses eingefügt, um den Artikelnamen bei einem darauffolgenden Import wiederherstellen zu können. Außerdem wird auch der Quellcode des Artikels, wie dieser im Wiki steht, als *definition*-Property exportiert, um ihn ebenfalls bei einem Import wiederherstellen zu können. Am Ende der RDF-Beschreibung sind die beiden Listen beschrieben, zuerst die ungeordnete Liste namens *Tiere* und anschließend eine *OrderedCollection* mit dem Namen *Autos*. Derselbe Ausschnitt sieht in RDF/XML folgendermaßen aus:

```
<rdf:RDF
   xmlns:skos=http://www.w3.org/2008/05/skos#
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >

<rdf:Description rdf:about="http://www.example.com/Biologie">
   <skos:definition>ArticleSource:
      &lt;&lt;ConceptScheme&gt;&gt;
      ~URI=http://www.example.com/Biologie~
      [hasTopConcept [Lebewesen]]
   </skos:definition>
   <skos:hasTopConcept
      rdf:resource="http://www.skosylvi.com/Lebewesen"/>
   <skos:prefLabel xml:lang="de">Biologie</skos:prefLabel>
   <rdf:type
      rdf:resource="http://www.w3.org/2008/05/skos#ConceptScheme"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.skosylvi.com/Lebewesen">
   <skos:definition>ArticleSource:
      &lt;&lt;Concept&gt;&gt;
      ~note=Dies ist ein Lebewesen-Konzept.~
      [inScheme [Biologie]]
   </skos:definition>
   <skos:note>Dies ist ein Lebewesen-Konzept.</skos:note>
   <skos:inScheme rdf:resource="http://www.example.com/Biologie"/>
   <skos:prefLabel xml:lang="de">Lebewesen</skos:prefLabel>
   <rdf:type rdf:resource="http://www.w3.org/2008/05/skos#Concept"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.skosylvi.com/Tier">
   <skos:definition>ArticleSource:
      &lt;&lt;Concept&gt;&gt;
      ~prefLabel=Tier@de~
      ~altLabel=Vieh@de~
      [broader [Lebewesen]]
      [inScheme [http://www.example.com/Biologie]]
   </skos:definition>
   <skos:inScheme rdf:resource="http://www.example.com/Biologie"/>
   <skos:altLabel xml:lang="de">Vieh</skos:altLabel>
   <skos:broader rdf:resource="http://www.skosylvi.com/Lebewesen"/>
   <skos:prefLabel xml:lang="de">Tier</skos:prefLabel>
   <rdf:type rdf:resource="http://www.w3.org/2008/05/skos#Concept"/>
</rdf:Description>
```

```
<rdf:Description rdf:about="http://www.skosylvi.com/Fisch">
   <skos:definition>ArticleSource:
      &lt;&lt;Concept&gt;&gt;
      ~prefLabel=Fisch@de~
      ~hiddenLabel=Fish@de~
      ~hiddenLabel=Fiesch@de~
      ~hiddenLabel=Visch@de~
      [inScheme [*IDhttp://www.example.com/Biologie*ID]]
      [broader [Tier]]
   </skos:definition>
   <skos:inScheme rdf:resource="http://www.example.com/Biologie"/>
   <skos:hiddenLabel xml:lang="de">Fish</skos:hiddenLabel>
   <skos:hiddenLabel xml:lang="de">Fiesch</skos:hiddenLabel>
   <skos:hiddenLabel xml:lang="de">Visch</skos:hiddenLabel>
   <skos:broader rdf:resource="http://www.skosylvi.com/Tier"/>
   <skos:prefLabel xml:lang="de">Fisch</skos:prefLabel>
   <rdf:type rdf:resource="http://www.w3.org/2008/05/skos#Concept"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.example.com/Insekt">
   <skos:definition>ArticleSource:
      &lt;&lt;Concept&gt;&gt;
      ~URI=http://www.example.com/Insekt~
      ~notation=Insekt~
      [inScheme [Biologie]]
      [broader [Tier]]
   </skos:definition>
   <skos:notation
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Insekt
   </skos:notation>
   <skos:inScheme rdf:resource="http://www.example.com/Biologie"/>
   <skos:broader rdf:resource="http://www.skosylvi.com/Tier"/>
   <skos:prefLabel xml:lang="de">Insekt</skos:prefLabel>
   <rdf:type rdf:resource="http://www.w3.org/2008/05/skos#Concept"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.example.com/Saeugetier">
   <skos:definition>ArticleSource:
      &lt;&lt;Concept&gt;&gt;
      ~URI=http://www.example.com/Saeugetier~
      ~example=Auch ein Mensch ist ein Saeugetier.~
      [inScheme [Biologie]]
      [broader [Tier]]
      [narrower [Mensch]]
   </skos:definition>
   <skos:narrower rdf:resource="http://www.skosylvi.com/Mensch"/>
   <skos:example>Auch ein Mensch ist ein Saeugetier.</skos:example>
   <skos:inScheme rdf:resource="http://www.example.com/Biologie"/>
   <skos:broader rdf:resource="http://www.skosylvi.com/Tier"/>
   <skos:prefLabel xml:lang="de">Saeugetier</skos:prefLabel>
   <rdf:type rdf:resource="http://www.w3.org/2008/05/skos#Concept"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.example.com/Tiere">
   <skos:prefLabel xml:lang="de">Tiere</skos:prefLabel>
   <skos:member rdf:resource="http://www.skosylvi.com/Tier"/>
   <skos:member rdf:resource="http://www.skosylvi.com/Delphin"/>
   <skos:member rdf:resource="http://www.skosylvi.com/Ameise"/>
   <rdf:type
      rdf:resource="http://www.w3.org/2008/05/skos#Collection"/>
   <skos:member rdf:resource="http://www.example.com/Saeugetier"/>
   <skos:member rdf:resource="http://www.example.com/Insekt"/>
```

```
   <skos:member rdf:resource="http://www.example.com/Loewe"/>
   <skos:member rdf:resource="http://www.skosylvi.com/Wal"/>
   <skos:member rdf:resource="http://www.skosylvi.com/Biene"/>
   <skos:member rdf:resource="http://www.skosylvi.com/Fisch"/>
   <skos:member rdf:resource="http://www.example.com/Hai"/>
   <skos:member rdf:resource="http://www.skosylvi.com/Tiger"/>
   <skos:definition>ArticleSource:
      &lt;&lt;Collection&gt;&gt;
      ~URI=http://www.example.com/Tiere~
      [member [Tier]]
      [member [Fisch]]
      [member [Insekt]]
      [member [Saeugetier]]
      [member [Ameise]]
      [member [Biene]]
      [member [Hai]]
      [member [Loewe]]
      [member [Tiger]]
      [member [Wal]]
      [member [Delphin]]
   </skos:definition>
</rdf:Description>

<rdf:Description rdf:about="http://www.example.com/Autos">
   <skos:definition>ArticleSource:
      &lt;&lt;Collection&gt;&gt;
      ~URI=http://www.example.com/Autos~
      ~ordered=true~
      [member [Auto]]
      [member [Sportauto]]
   </skos:definition>
   <skos:memberList rdf:nodeID="A0"/>
   <rdf:type
     rdf:resource="http://www.w3.org/2008/05/skos#OrderedCollection"/>
   <skos:prefLabel xml:lang="de">Autos</skos:prefLabel>
</rdf:Description>

<rdf:Description rdf:nodeID="A0">
   <rdf:rest rdf:nodeID="A1"/>
   <rdf:first rdf:resource="http://www.example.com/Sportauto"/>
</rdf:Description>

<rdf:Description rdf:nodeID="A1">
   <rdf:rest
      rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
   <rdf:first rdf:resource="http://www.example.com/Auto"/>
</rdf:Description>
```

Dieses aus den Artikeln generierte RDF-Modell kann nun von anderen Instanzen importiert werden. Bei diesem Import werden im semantischen Wiki neue Artikel erstellt bzw. alte Artikel überschrieben und der als *skos:definition* exportierte Quellcode des Artikels wiederhergestellt. Wird das Modell in der Zeitspanne zwischen Export und Import verändert, muss der Quellcode des Artikels gegebenenfalls modifiziert werden, um mögliche Inkonsistenzen zu verhindern.

5 Schlussbetrachtung

5.1 Fazit

Domain-Ontologien sind ein wesentlicher Bestandteil des Semantischen Webs und Voraussetzung für die automatisierte Interpretation und Weiterverarbeitung von Daten. Zum aktuellen Zeitpunkt ist die Anzahl verfügbarer und verwendbarer Ontologien im Bereich des Semantischen Webs jedoch relativ gering. Darüber hinaus sind für viele Wissensbereiche überhaupt noch keine Ontologien vorhanden. Eine der Hauptursachen hierfür sind wohl die bisher angewendeten Methoden bei der Ontologieentwicklung, welche diesen von Natur aus dynamischen und kollaborativen Prozess nicht ausreichend unterstützen. Traditionelle Tools zur Ontologieentwicklung sind meist sehr komplex, sodass man zur Erstellung von Ontologien einerseits Fachexperten des jeweiligen Wissensbereichs und andererseits Experten im Bereich des Knowledge Engineering benötigt.

In dieser Arbeit wurde ein alternativer Ansatz vorgestellt, welcher sich der verteilten Infrastruktur eines semantischen Wikis zur kollaborativen Erstellung von Ontologien bedient. Traditionelle Wiki-Anwendungen, wie zum Beispiel Wikipedia, beweisen, dass eine Community in der Lage ist, Wissensbereiche konsistent zu beschreiben und dieses Wissen auch kontinuierlich zu verfeinern, sodass das im Wiki ausgedrückte Wissen auch immer auf einem aktuellen Stand bleibt. Das im Rahmen dieser Arbeit entwickelte System unterscheidet sich für einen gewöhnlichen Benutzer von außen betrachtet kaum von einem traditionellen Wiki. Alle Benutzer können ihr eigenes Wissen mit einbringen und dieses Wissen in Artikeln informal beschreiben. Dieses informal beschriebene Wissen kann daraufhin von erfahrenen Knowledge Engineers mit Hilfe von semantischen Annotationen formalisiert werden, sodass dieses Wissen nicht nur informal, sondern auch formal, also maschinenverständlich, beschrieben ist. Das Besondere daran ist, dass Fachexperten und Knowledge Engineers zusammengeführt werden, ohne dabei irgendwelchen physischen oder zeitlichen Einschränkungen ausgesetzt zu sein. Dieser Ansatz ist also ein ständig aktiver und allgegenwärtiger Prozess, während die traditionelle Ontologieentwicklung eher einem statischen und zeitpunktorientierten Prozess ähnelt.

Der wesentliche Unterschied zu ähnlichen bereits existierenden Projekten liegt darin, dass hier ein Wiki nicht nur als Infrastruktur eines reinen Ontologieeditors verwendet wird, sondern das Wiki selbst im Vordergrund steht. In diesem System wird also nicht nur formales Wissen beschrieben, sondern informales Wissen mit formalem Wissen vermischt. Es profitiert also nicht nur die Ontologieentwicklung von der verteilten Infrastruktur eines Wikis, sondern auch das Wiki profitiert von den definierten Ontologien, weil diese Ontologien direkt im Wiki selbst verwendet werden können, um andere Artikel semantisch zu beschreiben, was die Qualität des Wikis, vor allem hinsichtlich der Suche, deutlich verbessert. Man kann diese Wechselwirkung sozusagen als Symbiose betrachten, in welcher beide zusammengeführten Systeme (das semantische Wiki und der Ontologieeditor) voneinander profitieren.

Wie die Geschichte des traditionellen Webs zeigt, erlangt ein bestimmter Wissensbereich im Web oft erst dann eine große Informationsdichte, wenn eine große Community, also eine Vielzahl von Personen, an der Gestaltung dieses Wissensbereichs selbst beteiligt ist. Es spricht im Grunde genommen nichts dagegen, dass es sich mit der Ontologieentwicklung im Bereich des Semantischen Webs nicht ähnlich verhalten könnte.

Aufgrund des kollaborativen, dynamischen und allgegenwärtigen Charakters eines Wikis ist zu erwarten, dass aktuellere und, weil die Ontologien von den Benutzern selbst erstellt werden, den Anforderungen der Benutzer auch tatsächlich entsprechende Ontologien entwickelt werden. Diese positive Entwicklung könnte als Folge haben, dass die Ontologien auch verstärkt eingesetzt werden, weil diese dadurch für die Benutzer brauchbarer sind als von Fachexperten und Knowledge Engineers entwickelte Ontologien, auf deren Evolution ein Benutzer keinen Einfluss nehmen kann. Eine höhere Anzahl verfügbarer und qualitativ hochwertiger Ontologien könnte die Entwicklung von mächtigen, verteilten und automatisierten Applikationen im Bereich des Semantischen Webs fördern, womit das wahre Potential von Anwendungen des Semantischen Webs auch eine breitere Masse erreichen würde und somit ein Bewusstsein für das Semantische Web in den Köpfen der Benutzer schaffen könnte. Dies könnte wiederum ein Anreiz für die Ontologieentwicklung darstellen, die Anzahl und die Qualität von verfügbaren Ontologien weiter zu erhöhen, um noch mächtigere und bezüglich ihrer Funktionalität weit miteinander verbundene Applikationen zu ermöglichen, wie zum Beispiel intelligente Agenten. All diese Impulse könnten die Entwicklung und die Durchsetzung des Semantischen Webs beschleunigen, sodass Tim Berners-Lee Zukunftsvision aus dem Jahre 1999 vielleicht bald Wirklichkeit wäre.

5.2 Weitere Ideen

SKOS eignet sich streng betrachtet nur zur Erstellung von wenig ausdrucksstarken Ontologien. Im Grunde genommen können in diesem entwickelten SKOS-Editor nur Typen und deren Beziehungen zueinander definiert werden, nicht jedoch eigene Attribute oder Assoziationen. Diesbezüglich ist man auf die vordefinierten Attribute und Assoziationen des im semantischen Wiki zur Verfügung gestellten SKOS-Metamodells beschränkt.

Aber die Entwicklung und Integration dieses SKOS-Editors ins semantische Wiki könnte nach einer erfolgreichen Evaluierung des Systems beweisen, dass es auch möglich ist, komplexere und ausdrucksstärkere Ontologiesprachen, wie zum Beispiel OWL, in das semantische Wiki zu integrieren, da das interne Datenmodell sowohl die Erstellung von Attributen als auch von Assoziationen unterstützt. Eine mögliche nächste Herausforderung wäre also die Integration von OWL ins semantische Wiki, sodass auch etwas ausdrucksstärkere Ontologien entwickelt werden können, mit welchen es auch möglich ist, aus bereits vorhandenem Wissen neues Wissen zu generieren. Die Integration von OWL ins Wiki würde auch Property- und Class Extension in SKOS ermöglichen, was die im Kapitel 4.1.2 beschriebene Limitation 20 beseitigen würde, weil man mit Hilfe von OWL auch SKOS um ausdrucksstärkere Klassen und Properties erweitern könnte.

Generell sei erwähnt, dass die bisher beschriebene Syntax zur Beschreibung von Artikeln durch benutzerfreundliche Interfaces ersetzt werden sollte, wobei es aber möglich sein sollte, die bisher beschriebene Syntax weiterhin zu verwenden. Zum Beispiel wäre es möglich, Pull-Down-Menüs nach einer Auswahl eines bestimmten Properties mit den für dieses Property gültigen Ressourcen dynamisch zu befüllen.

Dieser entwickelte kollaborative SKOS-Editor ist ein erster Prototyp, der noch nicht vollständig kompatibel zu SKOS und völlig ausgereift ist, was man auch an der Vielzahl der in dieser Arbeit beschriebenen Limitations erkennen kann. Aus diesem Grund sollte man in weiterer Folge danach trachten, diese Limitations zu beseitigen, um die Applikation zu verbessern. Bei den meisten Limitations handelt es sich allerdings um Einschränkungen bezüglich des zugrunde liegenden Datenmodells in METIS, auf welches bei der Entwicklung des Editors kein Einfluss genommen werden konnte. Dies bedeutet also, um jene Limitations, die das interne Datenmodell als Ursache haben, zu beseitigen, müsste auch das Datenmodell in METIS modifiziert werden. Allerdings wä-

re dies ein sehr umfangreicher Einschnitt und hätte womöglich auch zur Folge, dass bereits existierende Applikationen auf Basis von METIS erneuert werden müssten.

Dennoch werden nachfolgend ein paar Ideen beschrieben, wie man einige dieser vorhandenen Limitations beseitigen könnte, um die Qualität der Applikation zu erhöhen. Zum Beispiel könnte man die Klasse für *MediaTypeCategories* verändern, sodass man ihnen, wie auch *MediaTypes*, ein *Configuration Property* zuweisen kann, in welchem der URI und die interne ID des Artikels gespeichert werden könnten (L 1, L 22 und L 25). Außerdem müsste es möglich sein, einen *MediaType* in mehrere *MediaTypeCategories* einordnen zu können, um die in SKOS erlaubte mehrfache Zuordnung eines Konzepts zu einem Konzeptschema zu unterstützen (L 2).

L 4 und L 9 (Angabe der Sprache bei Lexical Labels bzw. Documentation Properties) könnte man lösen, indem man einen Sprachdatentyp in METIS integriert, wobei die Auswahl einer Sprache wiederum mit einem benutzerfreundlichen Interface erfolgen sollte, die Ausgabe der Sprache allerdings unterbunden werden sollte, damit die Lesbarkeit des Artikels nicht negativ beeinflusst wird. Dies gilt auch für L 7 (angehängter Datentyp bei Notations). Auch hier sollte die Auswahl eines Datentyps zwar möglich sein, dieser sollte aber nicht in der Anzeige des Artikels sichtbar sein. Damit der Datentyp einer Notation auch als jeweiliger Datentyp interpretiert werden kann, könnte man einen generischen Datentypen anlegen oder aber auch im Metamodell für jeden Datentypen ein eigenes Attribut definieren. Man könnte alternativ aber auch das METIS-Datenmodell modifizieren, sodass es möglich wäre, ein verstecktes Attribut für ein Attribut selbst anzugeben, wie es auch schon bei Assoziationen möglich ist. Auf diese Weise wäre es ebenso möglich, die Anzeige der Sprache bzw. des Datentyps zu unterbinden.

Um die Documentation Properties vollständig zu unterstützen, müsste auch Dokumentation als RDF-Beschreibung und Dokumentation als Dokumentreferenz in Ylvi möglich sein (L 8). Die Dokumentation als Ressourcebeschreibung könnte man im Metamodell auch als *MetadataAttribute* realisieren, wobei der Wert des Attributs eine RDF-Beschreibung wäre. Dieses Dokumentationsmuster muss jedoch auf irgendeine Weise von der Dokumentation als Plain Literal unterschieden werden, weil diese auch unterschiedlich in RDF formuliert werden. Bei einer Dokumentation mittels Ressourcenbeschreibung wird die Beschreibung nämlich in einem *BlankNode* formuliert. Bei einer Dokumentation als Referenz auf ein Dokument muss jedoch auf eine Ressource refe-

renziert werden, was bedeutet, dass dafür ein *AssocType* zur Verfügung gestellt werden muss. Somit würden Documentation Properties im Metamodell sowohl als *Metadata-Attributes* als auch als *AssocTypes* zur Verfügung stehen.

Um die Semantic Relation- und Mapping Properties vollständig zu unterstützen, müsste man wiederum das interne Datenmodell von METIS verändern. Die meisten dieser Limitations beziehen sich aber ohnehin auf die Möglichkeit der Formulierung von relativ sinnlosen Statements in SKOS, welche mit dem Datenmodell von METIS nicht umgesetzt werden können, wobei in Ylvi Statements solcher Form ohnehin unerwünscht sind. Aus diesem Grund werden L 10, L 12 und L 13 bei den Semantic Relation Properties bzw. L 15, L 16 und L 18 bei den Mapping Properties vernachlässigt. Tatsächliche und nicht unwesentliche Limitations stellen jedoch L 11 bei den Semantic Relation- und L 17 bei den Mapping Properties dar, welche beschreiben, dass assoziative Relationen zwischen Ressourcen im internen Datenmodell nicht festgehalten werden können. Dafür müsste das interne Datenmodell erweitert werden, sodass auch assoziative Beziehungen im internen Modell abgebildet werden können.

Für Collections wäre es wohl sinnvoller, im Metamodell einen eigenen *MediaType* namens *OrderedCollection* zur Verfügung zu stellen anstatt die *Collection* mit Hilfe eines Attributs als geordnet zu kennzeichnen. (L 14). Auf diese Weise würde sich im Falle einer *OrderedCollection* nicht die Lesbarkeit des Artikels verringern.

Um das in L 21 beschriebene Problem (zwei Artikel mit demselben Namen) zu unterbinden, existieren bereits jetzt Mechanismen, die eine Erstellung eines neuen Artikels mit einem Namen, der bereits von einem anderen Artikel verwendet wird, verhindern sollen. Dabei wird bei der Erstellung des Artikels überprüft, ob ein Artikel mit dem entsprechenden Namen bereits existiert. Falls ein Artikel bereits vorhanden ist, werden die Benutzer darauf aufmerksam gemacht und haben die Möglichkeit, den vorhandenen Artikel zu editieren oder dennoch einen neuen Artikel mit gleichem Namen anzulegen. Alternativ könnte man das Erstellen von Artikel mit einem bereits vorhandenem Namen generell verbieten, indem der Benutzer gezwungen wird, in so einem Fall einfach einen verfügbaren Namen zu wählen.

Weiters sollte die im Kapitel 4.3 beschriebene und eigentlich geplante Funktionalität des *Exporter Plugins* (wie Content Negotiation und Möglichkeit zum einfachen Download des SKOS-Modells) implementiert werden. Vor allem die Funktionalität bezüglich

des Downloads ist von besonderer Bedeutung, damit das System auch wirklich in der Praxis eingesetzt werden kann (L 23).

Um das Problem der möglichen Aktualisierung von Artikeln während der Umwandlung der Artikel in korrespondierende METIS-Objekte (racing conditions) zu beseitigen, könnte man generell die Schreibrechte für Artikel während der Umwandlung deaktivieren, sodass diese für die Zeit der Umwandlung nicht verändert werden können (L 24).

Um das beschriebene Problem in L 26 zu beseitigen (zwei Ressourcen mit demselben URI im exportierten Modell, wenn mehr als zwei Artikel denselben Namen haben), könnte man den URI für Artikel, für welche kein URI explizit angegeben wurde, anders zusammensetzen. Man könnte den URL der jeweiligen Instanz verwenden (wie z.B. `http://www.skosylvi.com/`) und daran nicht den Namen des korrespondierenden METIS-Objekts, sondern den Namen des jeweiligen Artikels anhängen. Sollte der Fall auftreten, dass mehrere Artikel denselben Namen besitzen, könnte man den daraus resultierenden URI mit einer laufenden Nummer erweitern, sodass die URIs für die verschiedenen Artikel eindeutig sind.

6 Literaturverzeichnis

1. Sören Auer, Sebastian Dietzold, Thomas Riechert. OntoWiki – A Tool for Social, Semantic Collaboration. In: *Proceedings of 5th International Semantic Web Conference*, Seiten 736 – 749, Athens, GA, USA, 2006.

2. Jie Bao, Vasant Honavar. Collaborative Ontology Building with Wiki@nt – A multi-agent based ontology building environment. In: *ISWC Workshop on Evaluation of Ontology-based Tools (EON)*, Seiten 37-46, Hiroshima, Japan, 2004.

3. Tim Berners-Lee, Mark Fischetti. Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web. Seite 157 - 158, HarperBusiness, 2000.

4. Dan Brickley, R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004. URL http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

5. Michel Buffa, Fabien Gandon, Guillaume Eriteo, Peter Sander, Catherine Faron. SweetWiki: A semantic wiki. In: *Web Semantics: Science, Services and Agents on the World Wide Web*, Vol. 6, Nr. 1, Seiten 84 -97, Februar 2008.

6. Stefano Emilio Campanini, Paolo Castagna, Roberto Tazzoli. Platypus Wiki: a Semantic Wiki Wiki Web. In: *Proceedings of the 1st Italian Semantic Web Workshop Semantic Web Applications and Perspectives (SWAP)*, Ancona, Italien, 2004.

7. Matteo Cristani, Roberta Cuel. A Comprehensive Guideline for Building a Domain Ontology from Scratch. In: *I-Know: International Conference on Knowledge Management*, 2004.

8. Andreas Ekelhart, Stefan Fenz, Markus Klemen, Edgar Weippl. Security Ontologies: Improving Quantitative Risk Analysis. In: *40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, Seite 156ff, 2007.

9. Mariano Fernández, Asunción Gómez-Pérez, Natalia Juristo. METHONTOLOGY: From Ontological Art Towards Ontological Engineering. In: *Proceed-*

ings of the AAAI97 Spring Symposium Series on Ontological Engineering, Seiten 33 – 40, 1997.

10. Mark S. Fox. The TOVE Project Towards a Common-Sense Model of the Enterprise. In: *Lecture Notes In Computer Science*, Vol. 604; *Proceedings of the 5th international conference on Industrial and engineering applications of artificial intelligence and expert systems*, Seiten 25 – 34, 1992.

11. Martin Hepp, Daniel Bachlechner, Katharina Siorpaes. OntoWiki: Community-Driven Ontology Engineering and Ontology Usage based on Wikis. In: *Proceedings of the 2006 international symposium on Wikis*, Seiten 143 – 144, Odense, Dänemark, 2006.

12. Martin Hepp, Katharina Siorpaes, Daniel Bachlechner. Harvesting Wiki Consensus: Using Wikipedia Entries as Vocabulary for Knowledge Management. In: *IEEE Internet Computing, Vol. 11, No. 5,* Seiten 54 – 65, September – Oktober 2007.

13. Antoine Isaac, Ed Summers. SKOS Simple Knowledge Organisation System Primer, W3C Working Draft 21 February 2008. URL http://www.w3.org/TR/2008/WD-skos-primer-20080221/

14. Malte Kiesel. Kaukolu: Hub of the Semantic Corporate Intranet. In: *Proceedings of the First Workshop on Semantic Wikis – From Wiki to Semantics*, Seiten 31 – 42, Budva, Montenegro, Juni 2006.

15. Graham Klyne, Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. URL http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

16. Markus Krötzsch, Denny Vrandečić, Max Völkel, Heiko Haller, Rudi Studer. Semantic Wikipedia. In: *Journal of Web Semantics*, Vol. 5, Seiten 251 – 261, September 2007.

17. Javier Lacasta, Javier Nogueras-Iso, Francisco Javier Lopez-Pellicer, Pedro Rafael Muro-Medrano, Francisco Javier Zarazaga-Soria. ThManager: An Open Source Tool for creating and visualizing SKOS. In: *Information Technology and Libraries*, September 2007.

18. Frank Manola, Eric Miller. RDF Primer, W3C Recommendation 10 February 2004. URL http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

19. Deborah L. McGuinness, Frank van Harmelen. OWL Web Ontology Language Overview, W3C Recommendation 10 February 2004. URL http://www.w3.org/TR/2004/REC-owl-features-20040210/

20. Alistair Miles, Sean Bechhofer. SKOS Simple Knowledge Organisation System Reference, W3C Working Draft 25 January 2008. URL http://www.w3.org/TR/2008/WD-skos-reference-20080125/

21. Alistair Miles, Dan Brickley. SKOS Core Guide, W3C Working Draft 10 May 2005. URL http://www.w3.org/TR/2005/WD-swbp-skos-core-guide-20050510/

22. Elena Paslaru Bontas Simperl, Christoph Tempich. Ontology Engineering: A Reality Check. In: *5th International Conference on Ontologies, Databases, and Applications of Semantics (ODBASE2006)*, Montpellier, Frankreich, 2006.

23. Helena Sofia Pinto, João P. Martins. Ontologies: How can They be Built?. In: *Knowledge and Information Systems*, Vol. 6, Nr. 4, Seiten 441 – 464, 2004.

24. Niko Popitsch, Ross King, Gerd Utz Westermann. METIS: A Flexible Foundation for the Unified Management of Multimedia Assets. In: *Multimedia Tools and Applications*, Vol. 33, Nr. 3, Seiten 325 – 349, 2007.

25. Niko Popitsch. OntoYlvi: Collaborative Ontology Building with a Semantic Wiki. Unveröffentlichtes Manuskript, 2008.

26. Niko Popitsch, Bernhard Schandl, Stefan Leitich, Wolfgang Jochum, Arash Amiri. Ylvi – Multimedia-izing the Semantic Wiki. In: *1st Workshop „SemWiki2006 – From Wiki to Semantics"*, Budva, Montenegro, 2006.

27. Sebastian Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. In: *1st International Workshop on Semantic Technologies in Collaborative Applications STICA 06*, Manchester, Großbritannien, Juni 2006.

28. Sebastian Schaffert, François Bry, Joachim Baumeister, Malte Kiesel. Semantic Wikis. In: *IEEE Software*, Vol. 25, Nr. 4, Seiten 8 – 11, Juli / August 2008.

29. Katharina Siorpaes. Lightweight Community-Driven Ontology Evolution. In: *Proceedings of the 6th International Semantic Web Conference (ISWC 2007)*, Busan, Südkorea, 2007.

30. Katharina Siorpaes, Martin Hepp. myOntology: The Marriage of Ontology Engineering and Collective Intelligence. In: *Proceedings of the Workshop Bridging the Gap between Semantic Web and Web 2.0 (ESWC 2007)*, Innsbruck, Österreich, Juni 2007.

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

32. Mike Uschold, Martin King, Stuart Moralee, Yannis Zorgios. The Enterprise Ontology. In: *The Knowledge Engineering Review*, Vol. 13, Seiten 31 – 89, 1998.

Anhang

Kurzfassung

Ontologien sind die wichtigsten Bestandteile des Semantischen Webs und Grundvoraussetzung für die automatisierte Interpretation und Weiterverarbeitung von Information. Zum aktuellen Zeitpunkt besteht jedoch ein Mangel an verfügbaren und qualitativ hochwertigen Ontologien, was einer der Gründe für die langsame Entwicklung des Semantischen Webs ist. Die Gründe hierfür liegen zum Teil in der traditionellen Vorgehensweise bei der Ontologieentwicklung, welche diesen von Natur aus kollaborativen und dynamischen Prozess nicht ausreichend unterstützt. In dieser Arbeit wird ein alternativer Ansatz zur Ontologieentwicklung vorgestellt, welcher die verteilte Infrastruktur eines semantischen Wikis nützt, um den kollaborativen Charakter dieses Prozesses besser zu unterstützen. In diesem System soll es möglich sein, die informale Beschreibung eines Wiki-Artikels so zu erweitern, dass auch eine formale Beschreibung des Artikels vorliegt, sodass das in den Artikeln formulierte Wissen in eine standardisierte Beschreibungssprache exportiert werden kann. Auf diese Weise soll die Ontologieentwicklung einer breiten Masse zugänglich gemacht werden, sodass jeder Benutzer die Möglichkeit hat, aktiv an der Entwicklung mitzuwirken und seine eigenen Wünsche und Anregungen einzubringen. Weil das Internet ein über den gesamten Planeten viel genutztes und jederzeit erreichbares Netzwerk darstellt, wäre die Ontologieentwicklung auf diese Weise keinen örtlichen oder zeitlichen Einschränkungen ausgesetzt, was den gesamten Prozess der Entwicklung und Wartung einer Ontologie beschleunigen könnte. Wenn die Community eines Wikis lebendig und aktiv ist, ist davon auszugehen, dass die Anzahl verfügbarer und brauchbarer Ontologien deutlich zunimmt. Eine höhere verfügbare Anzahl qualitativ hochwertiger Ontologien wiederum könnte die Entwicklung und Durchsetzung des Semantischen Webs beschleunigen, weil dadurch hinsichtlich ihrer Funktionalität immer mächtigere automatisierte Applikationen möglich wären. Das Potenzial dieser mächtigen Applikationen könnte wiederum ein Bewusstsein für das Semantische Web in den Köpfen der Nutzer schaffen, was wiederum einen Anreiz darstellen könnte, die Anzahl verfügbarer Ontologien weiter zu erhöhen.

Abstract

Ontologies are the most important parts within the Semantic Web and necessary for the automated interpretation and processing of information. At this point in time there is however a lack of existing and reusable ontologies, which is one of the reasons the Semantic Web has developed quite slowly so far. One of the main reasons for this lack of ontologies is the traditional approach to ontology engineering. Ontology engineering is inherently a collaborative and dynamic process and these characteristics of this process are not supported by the traditional approach in a satisfiying manner. In this thesis, we present an alternative approach to ontology engineering which makes use of the distributed environment of a semantic wiki to better support this collaborative character. Within this wiki, users should be able to formalize informal explanations of articles so that it is possible to generate a formalized description of an ontology in a standardized language for defining ontologies. This way, even unexperienced users should be able to take part in the process of ontology engineering which could lead to a higher amount of persons involved in the engineering process and consequently to a higher amount of existing and reusable ontologies. The World Wide Web is permanently accessible and is used by a lot of people all over the planet. This means, ontology engineering within the WWW would no longer be exposed to local and temporal restrictions which could lead to an acceleration of the process of ontology engineering. Furthermore, if the community within the wiki is highly active, this could lead to a higher amount of ontologies, which meet the user's requirements. This in turn, could have a considerable impact to the development of the Semantic Web because a large amount of reusable ontologies could allow for powerful distributed applications, which interact with each other in an automated manner using these ontologies. These powerful applications could make the full potential of the semantic web visible to a large amount of users which in turn could be an incentive to ontology engineers to increase the amount of existing and reusable ontologies.

Lebenslauf

Persönliche Informationen

- Name: Wolfgang Philipp
- Familienstand: ledig
- Nationalität: österreichisch
- Geburtstag: 14. Oktober 1979
- Geburtsort: Klagenfurt

Ausbildung

1986 - 1990 Volksschule III Ferlach

1990 - 1995 Bundesgymnasium Lerchenfeldstraße Klagenfurt

1995 - 2000 Bundeshandelsakademie II Klagenfurt
Matura
- im Juni 2000 mit ausgezeichnetem Erfolg bestanden

2000 - 2005 Studium Wirtschaftsinformatik Universität Wien
1. Studienabschnitt
- abgeschlossen im Juni 2003

01 / 2007 Arbeiter-SamariterBund-Österreich Wien
Ausbildung zum Rettungssanitäter
- abgeschlossen Februar 2006

2007 - 2008 Studium Wirtschaftsinformatik Universität Wien
2. Studienabschnitt
- abgeschlossen im November 2008

Berufserfahrung

08 / 1999 Volksbank Kärnten Süd Ferlach
Ferienarbeit

07 - 08 / 2000 Audi AG Ingolstadt
Ferienarbeit

09 / 2001 Kraschitzer Ges.m.b.H. Klagenfurt
Erstellung der Firmenwebseite

07 – 08 / 2002 Kraschitzer Ges.m.b.H. Klagenfurt
Sekretär

08 / 2003 VIE Vienna International Airport Wien
Mitarbeiter in Gepäckzentrale

08 / 2004 Telekom Austria Wien
Erweiterung eines Content Management Systems

seit 10 / 2007 runIT EDV-DienstleistungsGmbH Wien
Software Developer

Zivildienst	01 / 2006 – 10 / 2006, absolviert beim Arbeiter-Samariterbund-Österreich in Wien.