[0]

Tutorium

Aufbau von Wissensdomänen anhand einer semantischen Datenbank

Studiengang:	Informatik
Autoren:	Sven Osterwalder[1], Mira Günzburger[2]
Betreuer:	Prof. Dr. Jürgen Eckerle[3]
Experte:	Leclerc Jean-Marie
Datum:	27. Dezember 2014

[0] https://openclipart.org/detail/17566/cartoon-owl-by-lemmling

[1] mira.guenzburger@students.bfh.ch
[2] sven.osterwalder@students.bfh.ch
[3] juergen.eckerle@bfh.ch

Versionen

Version	*Datum*	*Status*	*Bemerkungen*
0.1	03.10.2014	Entwurf	Initiale Erstellung des Dokuments
0.2	03. - 31.10.2014	Entwurf	Theoretische Grundlagen
0.3	31.10.2014	Entwurf	Leitfaden ent

Inhaltsverzeichnis

1 Einleitung

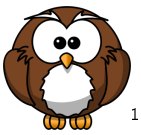
Die relationale Datenspeicherung ist ein klassicher Ansatz zur Wissensabbildung. Häufig wird die relationale Datenspeicherung in der objektorientierten Programmierung verwendet. Experten aus einer Fachrichtung sind dadurch fähig, die Daten zu interpretieren und Schlüsse zu ziehen. In der heutigen Zeit möchte man Zugang zum Wissen auch ohne die Hilfe von Experten haben.

Daher möchten wir eine neue, eher wenig bekannte Art der Wissensabbildung vorstellen, nämlich die Wissensmodellierung (Knowledge Engineering) bzw. Expertensysteme. Diese berücksichtigt das Abbilden von Objekteigenschaften und -Verhalten und kann mithilfe von Schlussfolgerungen die Rolle des Experten einnehmen. Sie ist auch Teilgebiet der künstlichen Intelligenz.

Grundsätzlich geht es beim knowledge engineering darum, nicht nur Informationen, sondern auch Wissen abzubilden. Es handelt sich um eine Wissensrepräsentation, welche zusätzliche Verarbeitungsmöglichkeiten aufweist zum Beispiel logisches Schlussfolgern oder Beweisführungen. Ein wichtiger Aspekt ist die Darstellung in einer von Maschinen lesbaren Form. [1]

Ein wichtiges Einsatzgebiet dieser Form der Wissensabbildung ist das semantische Web. Auch bei diesem geht es in erster Linie darum, Informationen mit ihrer Bedeutung (der sogenannten Semantik) zu verbinden.

Das nachfolgenden Dokument zeigt, wie Wissen in einem wissensbasierten System abgebildet und genutzt werden kann. Anhand unseres Beispieles eines Reiseplaners wollen wir zeigen, welche Schritte eine Wissensmodellierung beinhaltet. Weiter zeigen wir formale Aspekte für die Wissensmodellierung auf, wie zum Beispiel verschiedene Sprachen. Aufgrund unserer Erfahrung geben wir praktische Tipps für die direkte Umsetzung.



Als Symbol führen wir die Eule ein. Im jeweils danebenstehenden Abschnitt sind praktische Hinweise zur aktuellen Thematik aufgeführt.

Das Symbol Elefant dient um Beispiele zu geben, wie ein Thema in der Praxis umgesetzt werden kann. Das Erscheinen dieses Symboles (am Ende eines Kapitels) zeigt das entsprechende Beispiel im nebenstehenden Text.

---

[1] https://openclipart.org/detail/17566/cartoon-owl-by-lemmling
[2] https://openclipart.org/detail/17810/-by--17810

1.1 Wissen

Ein knowledge engineer betreibt Wissenakquise mithilfe von Experten aus der entsprechenden Fachwelt. Dafür müssen Wissensbestände eines Sachgebietes aufgebaut werden.
Die Kommunikation von Wissensbeständen ist ebenso wichtig wie ihr Vorhandensein. In der Fachwelt hängt die Wissenskommunikation sehr von der Anerkennung der Wissensmodellierung ab.

Je nach Situation ist eine andere Form der Wissensabbildung geeignet. Wenn man sich für eine bestimmte Wissensabbildung entschieden hat, muss diese eine spezielle Aufgabe erfüllen. Hierbei muss der Nutzen der Abbildung im Mittelpunkt stehen.

Eine Information kann für eine bestimmte Problemstellung unumgänglich, für eine andere aber nutzlos sein. Sucht jemand bei der Planung einer Reise zum Beispiel ein familienfreundliches Hotel zur Übernachtung, so ist die Historie der Besitzer dabei nicht relevant. Die Information, dass ein Hotel familienfreundlich ist, jedoch schon.

Es gibt nicht "das richtige" Vorgehen beim Sammeln von Wissen. Es ist sinnvoll das Wissen von groben Begriffen bis hin zu detaillierten Informationen einzuteilen.

Das nachfolgende Kapitel zeigt einen der vielen Formalismen zum Sammeln und Nutzen von Wissen in Form eines Expertensystemes auf.

2 Expertensysteme

Bei Expertensystemen handelt es sich um Systeme zur Wissenrepräsentation, welche in einem sehr eingeschränkten (Teil-) Gebiet die Leistung eines menschlichen Experten ersetzen oder diese sogar übertreffen können.Entwicklung und Erfolg von Expertensystemen führten zu einem standardisierten Prozess der Wissensdarstellung und zu Ontologien. Dies vereinfachte die Entwicklung von Expertensystemen in neuen, unerforschten Themengebieten erheblich. [2, S. 257]

2.1 Komponenten

Um ein Expertensystem aufbauen zu können, muss zuerst das Wissen über einen Problembereich formalisiert werden. [3, S. 23]

Die dazu benötigten Komponenten sind:

- Eine Wissensdatenbank, welche die Fakten des Problembereiches in formaler Sprache enthält
- Ein Verarbeitungsmechanismus zum automatischen Ziehen von Schlüssen (Inferenz-Maschine)

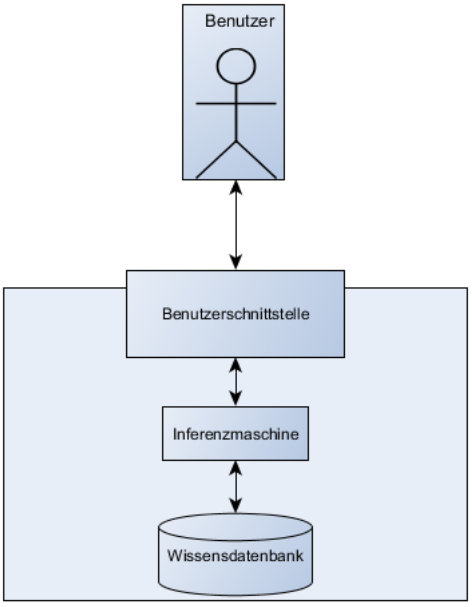


Abbildung 2.1: Aufbau eines Expertensystems.[1]

[1] [3, S. 23]

2.2 Problemlösung

Bei der Problemlösung, wird typischerweise in folgenden Schritten vorgegangen:

- Charakterisierung der Problemdomäne
- Symbolische Repräsentation der Objekte
- Eingabe des Wissens in den Computer
- Stellen von Fragen
- Interpretation der Antworten

Für die symbolische Repräsentation der Objekte ist es notwendig, eine geeignete Sprache zu wählen. Diese kann beispielsweise aus mathematischen Relationen oder Logik bestehen, oder eine Programmiersprache selbst sein.

In der Informatik wird das Wissen über ein Problem üblicherweise direkt mittels Lösungsalgorithmen programmiert. Bei der künstlichen Intelligenz hingegen, wird das Wissen von der Verarbeitungskomponente getrennt dargestellt. Dies hat den Vorteil, dass die Wissensbasis jederzeit ausgewechselt, die Verarbeitungskomponente jedoch bestehen bleiben kann. Ein Programm in Form eines Expertensystems kann somit also für unterschiedliche Anwendungen verwendet werden. (vgl. [3, S. 28 - 30])

Eine Problemlösung geht immer von dem vorhandenen, expliziten Wissen aus, z.B. in Form von Fakten.
Aus dem expliziten kann implizites Wissen gewonnen werden, es können damit Aussagen impliziert werden. Die Aufgabe der Verarbeitungskomponente besteht darin, das implizite Wissen abzuleiten. (vgl. [3, S. 30 - 31]) Die Sprache der Wissensrepräsentation und die zugehörige Verarbeitungskomponente müssen gewissen Kriterien erfüllen, Details siehe [3, S. 31].

2.3 Wissensarten

Um Wissen abzubilden, benützt der Mensch mehrere Arten:

- Relationales Wissen
- Vererbung von Eigenschaften
- Prozedurales Wissen
- Logisches Wissen

2.3.1 Relationales Wissen

Relationales Wissen wiederspiegelt einfache Beziehungen zwischen Objekten. Ein Nachteil am relationalen Wissen ist, dass nur Fakten abgebildet werden können, nicht aber logischen Abhängigkeiten.

```
Abenteuerreisen sind Reisen.
```

Listing 2.1: Einfaches Beispiel von relationalem Wissen.

2.3.2 Vererbung von Eigenschaften

Bei der Vererbung von Eigenschaften geht es um die Weitergabe von Eigenschaften einer Oberklasse an eine Unterklasse.

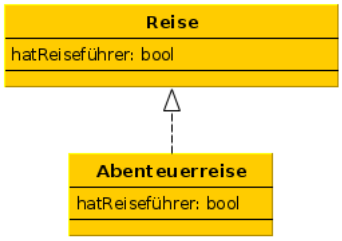


Abbildung 2.2: Einfaches Beispiel der Vererbung von Eigenschaften.[2]

Eine Reise hat einen Reiseführer. Definiert man eine Abenteuer-Reise ist auch eine Reise, ist es für den Menschen klar, dass eine Abenteuer-Reise auch einen Reiseführer haben kann. Beim Computer hingegen, muss die Unterklasse "Abenteuer-Reise" zuerst die Eigenschaft der Möglichkeit eines Reiseführers von der Oberklasse erben.

2.3.3 Prozedurales Wissen

Bei prozeduralem Wissen handelt es sich um ein Wissen, welches in bestimmten Situationen bestimmte Aktionen vorschreibt. Man kann dies auch als Folge von Aktionen auffassen. So zum Beispiel das Aufschliessen einer Türe: Man steckt den (passenden) Schlüssel in das Schlüsselloch, dreht diesen, entsichert damit das Schloss, drückt die Türfalle nach unten und öffnet die Türe.

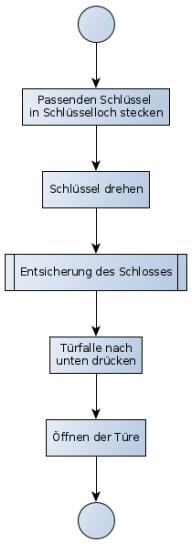


Abbildung 2.3: Einfaches Beispiel von logischem Wissen.[3]

[2] Eigene Darstellung mittels yEd.
[3] Eigene Darstellung mittels yEd.

2.3.4 Logisches Wissen

Bei logischem Wissen geht es im Grunde genommen um eine logische Implikation. Aus A folgt B bzw. $A \rightarrow B$, was so viel heisst wie "Wenn A gilt, kann geschlossen werden, dass auch B gilt.".

Möchte man darstellen, dass ein Ausflug teambildend ist, müssen folgende Voraussetzungen erfüllt sein: Für eine gewisse Anzahl Teilnehmer geeignet und gute Zusammenarbeit fördernd.

Formal ausgedrückt: $A \wedge B \rightarrow C$. Hierbei beudetet A: geeignet für eine gewisse Anzahl Teilnehmer und B: Förderung der Zusammenarbeit.

2.4 Wissensrepräsentationsformalismen

Formalisiert man die unter 2.3 genannten Arten des Wissens, so gelangt man zu den folgenden Wissenrepräsentationsformalismen:

- Logik
  - Aussagenlogik
  - Prädkatenlogik erster Stufe
- Semantische Netze und Frames
- Regelbasierte Sprachen

Wie unter 2.1 beschrieben, bildet die Basis eines Expertensystemes unter anderem eine Wissensdatenbank. Zur Erleichterung des Aufbaus einer Wissendatenbank und zur grafischen Darstellung, eignen sich Graphen sehr gut. Diese werden im nachfolgenden Kapitel beschreiben.

3 Graphrepräsentationen

Sofern im Text nicht anders vermerkt, basiert das nachfolgende Kapitel auf [4].

Graphen bilden eine der Grundlagen für semantische Daten, besonders für semantische Netze. In diesem Zusammenhang spricht man auch von Graphdatenbanken.

Der Unterschied gegenüber den gängigen Datenbanken, wie zum Beispiel den relationalen oder hierarchischen Datenbanken, liegt vorallem darin, dass Objekte beliebig verknüpft werden können.

Eine Graphdatenbank ist analog einem Graphen aufgebaut. Sie nutzt also dessen Struktur, bestehend aus Knoten, Kanten und Eigenschaften, um Daten bzw. Wissen darzustellen und abzulegen. Eine Graphdatenbank ist also ein Graph mit zyklenfreier Nachbarschaft. Dies bedeudet, dass jedes Element einen direkten Verweis auf seine benachbarten Elemente enthält und somit keine Abfragen auf dessen Relationen notwendig sind.

Die Knoten einer Graphdatenbank repräsentieren Entitäten. Eigenschaften sind auf Knoten bezogene Informationen. Kanten verbinden Knoten mit Knoten oder Knoten mit Eigenschaften und stellen die Beziehung zwischen diesen dar.

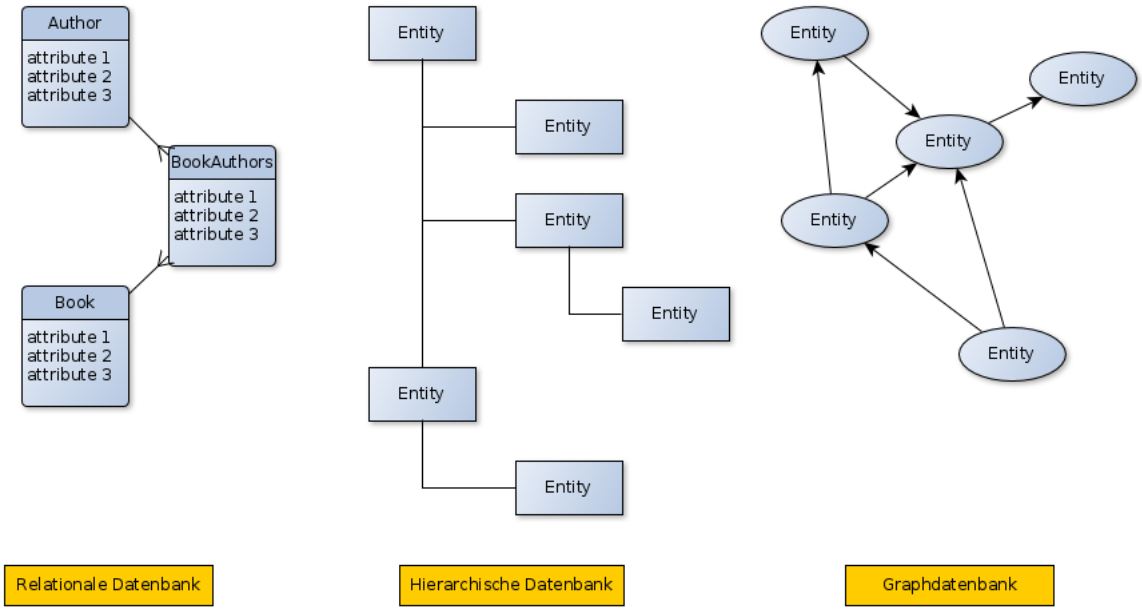


Abbildung 3.1: Darstellung der verschiedenen Datenbanktypen.[1]

[1] Eigene Darstellung mittels yEd

Gegeben seien die folgenden Aussagen über Hotels:

```
Ein Wellnesshotel ist ein Hotel.
Ein Familienhotel ist ein Hotel.
Wellnesshotels sind mit Familienhotels verwandt.
```

Listing 3.1: Aussagen über Hotels[2]

Verwendet man diese Hotelangaben um daraus eine Graphdatenbank zu erstellen, so ergibt sich folgende Graphdatenbank:

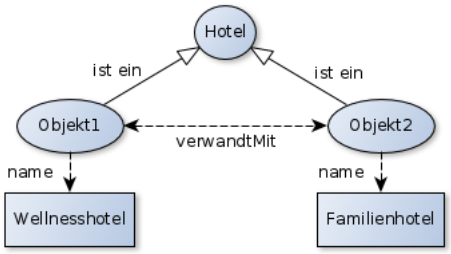


Abbildung 3.2: Aussagen über Hotels als Graphdatenbank.[3]

In der Graphdatenbank existieren also zwei Objekte, *"Objekt1"* und *"Objekt2"*, mit den Eigenschaften *"ist ein"*, *"verwandtMit"* und *"name"*.

[3]Eigene Darstellung mittels yEd

Möchte man eine Graphdatenbank als Grundlage für eine semantische Datenbank aufbauen, so empfiehlt es sich, zuerst die wichtigsten Klassen und Individuen zu definieren. Es ist lohnen schon zu Beginn des Aufbaus schrittweise vorzugehen.

Nimmt man das Beispiel des Reispeplaners, ist es sinnvoll nicht von Anfang an eine komplette Reise abzubilden, sondern zuerst mit der Abbildung lokaler Ausflüge zu beginnen. Später können diese mittels Logik zu einer Reise kombiniert werden.

Nach Überlegung ergeben sich die Klassen *Ausflug*, *Land*, *Region* und *Ort*. Doch wie gelangt man zu diesen Klassen? Nach unserer Erfahrung lohnt es sich, konkrete Fragen zu stellen, welche die semantische Datenbank schliesslich beantworten können sollte. Es ist zudem hilfreich, sich praktische Beispiele von der Anwendung solch einer Datenbank zu überlegen.

So könnte eine konkrete Anwendung eine Familie sein, welche einen eintägigen Ausflug planen möchte und deren Kinder bereits in einem Alter sind, in dem sie Beschäftigung benötigen. Dies führt schliesslich zu den Kriterien *familienfreundlich*, *regional* und *actionreich*. Dies sind ausschliesslich Überlegungen für die spätere Entwicklung des Modelles. Eine Graphdatenbank kann nicht ohne Weiteres komplexe Anfragen im Sinne der genannten Kriterien oder Inhalte beantworten. Die Zusammenhänge könnte man durch Zuweisung von Kriterien an die Objekte statisch modellieren. Dies würde jedoch den Vorteil einer semantischen Datebank zu Nichte machen, nämlich die Gewinnung von Schlüssen aus komplexen Zusammenhängen.

Hat man Klassen und Kriterien definiert, fällt auf, dass damit noch keine konkreten Anfragen beantwortet werden können. Es fehlen die Individuen und die Relationen.
Daher wird für einen Ausflug das "Individuum *Seilpark Balmberg*" erstellt. Der Seilpark befindet sich in der Ortschaft *Balmberg*, welche in der Region *Solothurn* und damit in der *Schweiz* liegt.
Damit erscheint es logisch, wenn ein Land in Regionen unterteilt ist, diese wiederum Orte beinhalten. Daher werden für Länder, Regionen und Orte ebenso Individuen erstellt.

Dies kann über die Relationen *hatRegion* und *hatOrt* abgebildet werden.

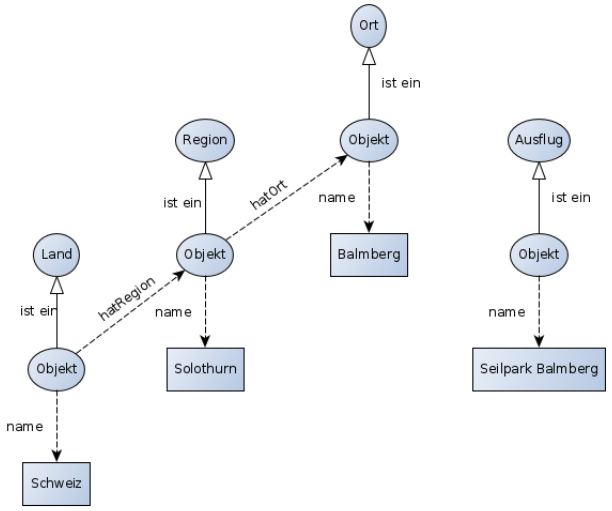


Abbildung 3.3: Beispiel einer Graphdatenbank basierend auf dem Beispiel eines Reiseplaners[4]

Wie aus der Grafik ersichtlich, hat das Individuum *Seilpark Balmberg* noch keine Relationen zu anderen Individuen. Daher ist zum jetzigen Zeitpunkt die Verknüpfung für Mehrwert bei Abfragen unklar.

Da eine Graphdatenbank eine hohe Abstraktionsebene darstellt, werden Wissensrepräsentationsformen sowie ihre Anwendung im folgenden Kapitel genauer erläutert.

[4]Eigene Darstellung mittels yEd[5]

4 Wissensrepräsentationsformen

Nach Darstellung der Grundlage von Graphen und Graphdatenbanken im letzten Kapitel, wollen wir jetzt Repräsentationsformen von Wissen analysieren.
In der Wissensmodellierung (knowledge engineering) gibt es verschiedene Formen der Wissensrepräsentation um Wissen in wissensbasierten Systemen formal abzubilden. Auf diese Art abgelegte Informationen werden als Wissensdatenbank bzw. Wissensbasis bezeichnet. [5]

Das folgende Kapitel basiert auf [3] und beschreibt einige klassische Formen der Wissensrepräsentation, nämlich semantische Netze, Wissensnetze und Frames.

Im Gegensatz zu Regeln stehen hier die Objekte und nicht Zusammenhänge und logische Abhängigkeiten im Vordergrund.

Semantische Netze und Frames versuchen das menschliche Gedächtnis abzubilden. Sie wurden hauptsächlich zur Analyse von Wörtern und Sätzen verwendet. Ein weiterer Aspekt ist die verständliche Darstellung von Klassen und ihren Beziehungen. Die Konzepte der semantischen Netze und Frames haben die Entwicklung der objektorientierten Programmierung beeinflusst.

4.1 Semantische Netze

Eine zusammengehörige Gruppe von Objekten wird als Klasse bezeichnet. Ein einzelnes Objekt heisst Individuum. Es gibt Beziehungen zwischen Objekten, zwischen Objekten und Klassen und zwischen Klassen.

Folgende Beziehungen werden unterschieden:

- "ist ein" Relation

  Es handelt sich um Ober- und Unterklassen.

  Beispiel: *Ein Baum ist eine Pflanze*.

- "Instanz von" Relation

  Die Relation sagt aus von welchem Typ ein Individum ist.

  Beispiel: *Birke ist eine Instanz der Klasse Baum*.

- Eigenschaft

  Klassen und Objekte haben Eigenschaften.

  Beispiel: *Pflanzen erzeugen Sauerstoff*.

Eigenschaften sind transitiv: Ist also ein Hund ein Tier und ein Tier ein Lebewesen, ist auch ein Hund ein Lebewesen. Weiter unterliegen Eigenschaften de, Gesetz der Vererbung. Ein Tier benötigt Sauerstoff, damit benötigt auch ein Hund Sauerstoff.

In semantischen Netzen werden Objekte und Klassen als Knoten abgebildet. Beziehungen und Eigenschaften werden als Kanten dargestellt. Aussagen wie Existenzaussagen und Oder-Aussagen können mit semantischen Netzen nicht abgebildet werden. Komplexe Abbildungen, wie zum Beispiel das Modellieren einer Aktion, sind trotz reinen zweistelligen Beziehungen möglich.

4.2 Frames

In Frames werden die wesentlichen Charakteristika eines Objektes als Eigenschaften abgebildet. Dabei unterstützen Frames die Konzepte der Hierarchie und der Vererbung. Frames können auch generische Informationen wie Standartwerte (Defaults) und Wertebeschränkungen (Listen) enthalten.

```
frame Abenteuerreise is a Reise :
    default hatReiseleiter is true
instance 'Dschungeltrip' is a kind of Abenteuerreise :
    anbieter is dernettereiseanbieter
    and typ is abenteuer
    and kosten is 1000.
constraint preise
    when the typ of an Abenteuerreise changes to X
    then check that X is {dschungel or abenteuer or nevernkitzel or natur}
    otherwise write( 'Der Typ der Reise wurde nicht geändert' )
    and nl.
```

Listing 4.1: Beispiel eines Frames anhand einer Reise.

4.3 Wissensnetze

Bei Wissensnetzen handelt es sich um eine bestimmte Art der Wissensrepräsentation. Dabei werden die Konzepte der semantischen Netze verwendet.

Wissen wird in Wissensnetzen objektorientiert oder mittels Frames abgebildet. Eine grafische Darstellung erfolgt zusätzlich mittels Topic Maps, auch Wissenslandkarte genannt.

"In einer Wissenslandkarte repräsentiert die Entfernung zweier Begriffe oder Wissensinhalte deren inhaltliche Nähe zueinander. Ein Wissensnetz ist somit ein Informationssystem, in dem zusätzlich die semantischen Beziehungen der Begriffe untereinander verwaltet werden. Diese Meta-Ebene ermöglicht eine **semantische Suche** – eine Suche, die über das Auffinden reiner Zeichenketten weit hinausgeht." [3, S. 89].

Die Knoten erhalten die Bedeutung von Instanzen (Individuen) oder Klassen. Das Problem der Mehrfachvererbung durch Einführung des Rollenkonzeptes umgangen. Durch Erweiterung der Klassen kann eine Instanz dann eine bestimmte Rolle einnehmen.

Wissensnetze haben alle Voraussetzungen, um effektives Wissensmanagement zu bieten. Dafür muss das Wissen aber immer aktuell und umfassend sein.

Ein sehr wichtiger und zeitintensiver Teil des Knowledge Engineerings ist die tatsächliche Modellierung, das heisst die Überlegung, ob eine abzubildende Information ein Objekt, eine Instanz oder eine Eigenschaft ist. Eventuell kann sie auch als Regel abgebildet werden kann. Regeln werden im Kapitel 9 Regeln — SWRL genauer erläutert.

An dieser Stelle scheint es uns wichtig zu erwähnen, dass das semantische Netz ein sehr gutes Hilfsmittel ist, um einen Überblick über die Informationen und ihre Anwendbarkeit zu erhalten. Würde man das Wissen direkt in die semantische Datenbank übertragen, wäre diese nicht viel mächtiger als eine traditionelle Wissensspeicherung. Zu einem späteren Punkt erklären wir dies genauer.

Möchte man ein semantisches Netz als Hilfsmittel zum Aufbau der Ontologie eines Reiseplaners nützen, ist das Vorgehen demjenigen der Graphdatenbank sehr ähnlich.

Für den Aufbau wichtig sind weiterhin die zuvor gemachten Überlegungen, das heisst ~~auch~~ die Klassen, Individuen und deren Relationen. Aktuell sind dies die *Klassen* Ausflug, Land, Region und Ort, die *Individuen* Schweiz, Solothurn, Bern und Seilpark Balmberg sowie die *Relationen* hatRegion und hatOrt.

Wie ist das weitere Vorgehen? Was unterscheidet ein semantisches Netz von einer Graphdatenbank?
Im Kapitel Graphdatenbanken wurde bereits eine wichtige Entität vorweggenommen, die den Hauptunterschied zwischen einem semantischen Netz und einer Graphdatenbank darstellt: Individuen. Diese lassen sich in Graphdatenbanken zwar abbilden, aber weniger intuitiv und aufwändiger.

Bei Verwendung eines semantischen Netzes besteht jedoch die Gefahr, eine ähnliche Modellierung wie bei der Graphdatenbank zu erhalten. Durch Verwendung des Editors Protégé der Universität Stanford und durch OWL 2 als Ontologiesprache konnten wir für die Modellierung der Ontologie einen guten Rahmen finden.

Ziel ist, die bereits formulierten Kriterien *familienfreundlich*, *regional* und *actionreich* so abzubilden, dass entsprechende Abfragen gestellt werden können.
Unter den bisherigen Entitäten lässt sich in unserem Beispiel nur das Kriterium *regional* abbilden.

Wenn wir davon ausgehen, dass der Standort bzw. die Region der Familie bekannt sind — diese seien der Einfachheit halber Solothurn —, sollte die Folgerung möglich sein, dass die Region des Seilparks Balmberg dieselbe wie diejenige der Familie ist.

Man kann mittels der Relation *hatOrt* definieren, dass die *Region Solothurn* den *Ort Balmberg* hat. Dies lässt den Schluss nicht zu, dass sich der Seilpark Balmberg im selben *Ort* oder in der selben *Region* befindet. Dies kann durch die Einführung der neuen Relation *hatStandort* beseitig werden. Das heisst, das Individium *Seilpark Balmberg* muss mit dem Ort *Balmberg* verbunden werden: $hatStandort(SeilparkBalmberg, Balmberg)$.

Das oben Genannte kann abgebildet werden wie folgt:

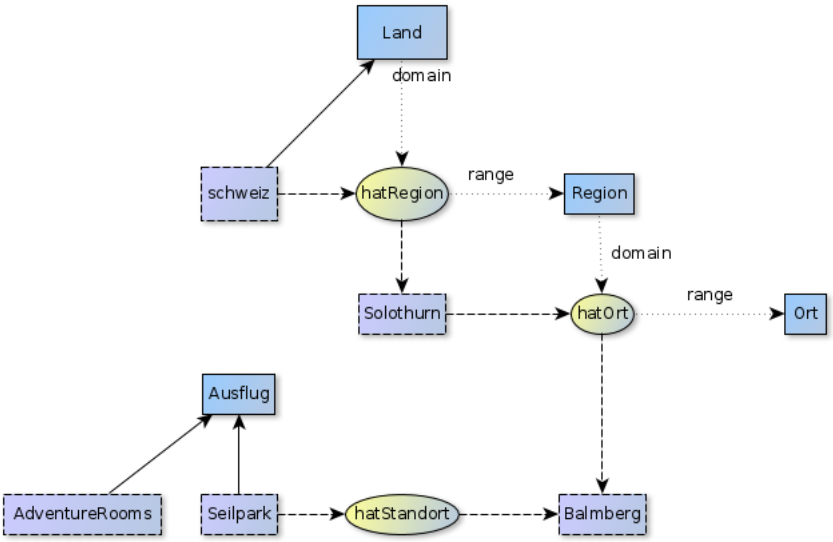


Abbildung 4.1: Abbildung von Wissen mittels eines semantischen Netzes.[1]

Bei dieser Abbildung handelt es sich um eine Übersicht der *Informationen*.

Wie kann ich aber schliessen, dass das Individuum *Seilpark Balmberg* die *Region Solothurn* hat? Aus dem vorhandenen *semantischen* Netz lässt sich dies nicht folgern. Somit fehlt die Möglichkeit einer Folgerung.

---

[1] Eigene Darstellung mittels yEd

5 Wissen abbilden mittels Ontologien

Die bisherigen Kapitel bieten einen groben Überblick über Objekte und deren Anwendung. Um bei der Wissensmodellierung von Nutzen zu sein, müssen sie aber in eine geeignete Form gebracht werden.
Wissen wird im knowledge engineering meist in Form von Ontologien abgelegt.

Der Begriff Ontologie wird in verschiedenen wissenschaftlichen Bereichen verwendet, so zum Beispiel in der Philosophie, der Psychologie und der Informatik. In diesem Dokument soll nur der Aspekt der Informatik erklärt werden.

Das folgende Kapitel basiert auf dem Artikel "Was bedeutet eigentlich Ontologie?" [6] sowie dem Artikel "Eine kurze Geschichte der Ontologie" [1].

In der Informatik steht Ontologie für "... eine formale Beschreibung des Wissens in einer Domäne in der Form von Konzepten der Domäne, deren Beziehung untereinander und der Eigenschaft dieser Konzepte und Beziehungen, sowie der in der Domäne gültigen Axiome und Prinzipien." [1, S.310].

Zum besseren Verständnis wird diese Definition im folgenden Abschnitt kurz untersucht:

- *Domäne*

  Unter einer Domäne versteht man einen Ausschnitt der Welt, dessen Grenzen klar definiert sind. Die Domäne und ihre Grenzen werden durch den Anwendungsfall festgelegt.

- *Konzepte*

  Konzepte werden in anderen Teilen dieses Dokumentes als Objekte oder Klassen bezeichnet. Dabei kann es sich um materielle Konzepte, wie zum Beispiel Teile eines Wagens oder um immaterielle Konzepte, wie zum Beispiel die Lösungssuche, handeln.

- *Beziehungen*

  Objekte stehen in Beziehungen zueinander. Es werden die Beziehungen "ist ein" und "Instanz von" unterschieden. In unserem Beispiel sind eine Abenteuerreise eine Reise und der Seilpark Balmberg eine Instanz eines Ausfluges ($Abenteuerreise\ istEin\ Ausflug$ und $SeilparkBalmberg\ instanzVon\ Ausflug$).
  Die Eigenschaften eines Objektes werden auch als Beziehungen dargestellt (eine Reise hat einen Reiseführer).

- *Axiome und Prinzipien*

  Damit werden die in der Domäne vorhandenen Regeln bezeichnet.

Mit diesen Elementen sind alle wichtigen Punkte einer Ontologie abgedeckt. Dabei ist es, wie in der Definition festgelegt, wichtig, dass die gesamte Abbildung in einer formalen Sprache beschrieben wird.

5.1 Anwendung von Ontologien

Ontologien werden für wissensbasierte Anwendungen verwendet. Konkret werden sie hauptsächlich dort verwendet, wo Semantik zur Formulierung von Informationen genutzt wird. Eines der bekanntesten Beispiele ist das semantische Web. Im Gegensatz zum syntaktischen Web versucht das semantische Web Zeichen nicht nur abzugleichen, sondern auch ein Verständnis und Schlussfolgerungen einzubauen.

Die Aufgabe von Ontologien ist die Ermöglichung und Verbesserung von Kommunikation zwischen Computeranwendungen untereinander und zwischen Computeranwendungen und Mensch. Wichtig: Es darf sich immer nur um eine bestimmte Wissensdomäne (Domain Ontology) handeln.
Einige Berufsgruppen haben bereits begonnen, ihr Wissen mit Hilfe von standardisierten Sprachen abzubilden.

Diese Sprachen, wie die in Kapitel 8 vorgestellte Sprache OWL, wurden vom W3C-Konsortium (World Wide Web Consortium) standardisiert.

5.1.1 Anwendung des semantischen Webs

Eine Anwendung für das semantische Web besteht aus der Gewinnung von zusätzlichem Wissen, was anhand einer Wissensbasis in Form einer Wissensdatenbank und einer Inferenzmaschine geschieht. Ein Ontologieschema legt in der Wissensbasis fest, welche Arten von Aussagen möglich sind. Die Aussagen enthalten das konkrete Wissen. Sowohl das Schema, wie auch die Wissensabbildung werden in einer formalen Sprache, wie z.B. OWL oder RDFS, abgebildet.

```
Eine Abenteuerreise ist eine Reise.
```

Listing 5.1: Beispiel einer Aussage in einer Wissensbasis.

In der einfachsten Anwendung kann das abgelegte Wissen direkt abgefragt werden. Bei komplexeren Anfragen sind die Funktionalitäten einer *Inferenzmaschine* notwendig. Mittels Inferenzregeln ist diese fähig Schlussfolgerungen zu ziehen und dadurch neue, komplexe Aussagen zu treffen.

Die Transitivitätsregel ist eine einfache, von der Inferenzmaschine verwendete Regel: "Wird eine Relation p als transitiv deklariert und es gilt xpy und ypz, dann kann xpz gefolgert werden." [6, Seite 289] Dem Entwickler ist es möglich, neben den vorhandenen Regeln, wissensdomänenspezifische Regeln zu spezifizieren. Die Inferenzmaschine berücksichtigt diese bei ihrer Auswertung.

```
Handelt es sich bei einem Objekt um eine Reise, benötigt das Objekt mindestens
    vier Teilnehmer/innen und ist das Objekt teambildend, so handelt es sich
    bei dem Objekt auch um einen Teamevent.
```

Listing 5.2: Beispiel einer Regel in einer Wissensbasis.

Damit eine Inferenzmaschine Schlüsse ziehen kann, muss die Wissensbasis in einer formalen Sprache vorliegen. Sie darf keine syntaktischen Fehlern enthalten. Das Schema und die Aussagen werden von der Maschine geladen und intern als Graph gespeichert (siehe Kapitel 4).
Durch einen Algorithmus zum Abgleich von Graphen werden Abfragen implementiert. Durch wiederholtes Anwenden von Regeln können *Schlussfolgerungen* gezogen werden.
Die Anwendung dieser Regeln erfolgt wiederum mit Hilfe von Algorithmen zum Abgleich von Graphen. Die dadurch entstandenen Aussagen werden der Wissensbasis hinzugefügt und stehen für Abfragen zur Verfügung.

Wir brauchen Ontologien. Dabei spielt die Wahl der Domäne eine grosse Rolle. Bei unserer Wissensmodellierung haben wir gelernt, das sich nicht jeder Themenbereich als Domäne eignet. Bei Umsetzung der ursprünglichen Problemdomäne, nämlich dem Erlernen des Programmierens anhand der Programmiersprache Prolog, mussten wir feststellen, dass diese nicht als Domäne geeignet ist. Zum Beispiel lässt sich das Konzept der Rekursion in einer Ontologie zwar lexikal abbilden, der Mehrwert (einer Ontolgie gegenüber der herkömmlichen Wissensabbildung) in Form von Inferenz, ist jedoch nicht gegeben. Die Möglichkeiten einer Wissensmodellierung anhand einer Ontologie sind bei Themen, die Inferenz und damit Schlussfolgerungen zulassen, viel grösser.

Im nachfolgenden Kapitel wird eine Erklärung gegeben, wie Inferenz und Resolution zum Ziehen von Schlüssen verwendet werden.

6 Inferenz und Resolution

6.1 Inferenz

Grundsätzlich geht es bei Inferenz um den Prozess von Schlussfolgerungen mit Hilfe von Resolution (siehe 6.2). Die logische Inferenz ist ein Prozess der Inferenz und der Resolution, welcher die Folgebeziehungen zwischen Sätzen zum Ausdruck bringt. [2, S. 163].

6.1.1 Inferenz in Computern

Das grundsätzliche Problem bei Computern im Bezug auf Inferenz ist, dass ein Computer keine Interpretation vornehmen kann und nichts über die (Um-) Welt weiss, bzw. nur, was in seiner Wissensdatenbank gespeichert ist (vgl. S. J. Russell [2, S. 164]).

Angenommen, man möchte einen Computer fragen:

```
''Ist eine Abenteuerreise eine Reise?''
```

Listing 6.1: Beispielanfrage an eine Wissensdatenbank eines Computers

so weiss der Computer weder was ein Abenteuerreise ist, noch kennt er das Konzept des Reisens an sich. Das Einzige, was er tun kann, ist in der Wissensdatenbank zu suchen nach:

```
''Eine Abenteuerreise ist eine Reise.''
```

Listing 6.2: Aussage in einer Wissensdatenbank eines Computers

Findet der Computer diese Aussage in der Wissensdatenbank, so spielt es keine Rolle, dass er das Konzept der Abenteuerreise oder des Reisens nicht kennt. Die Schlussfolgerung, dass eine Abenteuerreise eine Reise ist, trifft unter allen Gegebenheiten und Interpretationen zu, welche für die Wissensdatenbank zutreffen (vgl. [2, S.164]).

Zusammenfassend ist die formale Inferenz in der Lage, gültige Schlussfolgerungen zu ziehen, auch wenn der Computer die Interpretationen des Anwenders nicht kennt. Der Computer zieht immer logisch gültige Schlüsse, unabhängig von der (menschlichen) Interpretation. Da der Mensch in der Regel die Interpretation kennt, erscheinen die Schlüsse dem Menschen logisch (vgl. [2, S. 165]).

6.2 Resolution

Resolution, aus dem Lateinischen "resolutio", zu Deutsch "Auflösung", ist eine Verallgemeinerung des Modus Ponens [2, S. 279]. Der Modus ponens "...stellt eine universelle Schlussregel dar, die unabhängig vom jeweiligen Problem angewandt werden kann." [3, Seite 41] Beispiel: *Regen = nasse Strasse. Regen gegeben, also muss die Strasse nass sein.*

Die Methode der Resolution wurde 1965 von J. A. Robinson entwickelt. Dabei handelt es sich um einen vollständigen Algorithmus der Theorembeweisung für Prädikatenlogik erster Stufe. [2, S. 18] In der einfachsten Form der Resolution handelt es sich um eine Inferenz-Regel der Aussagenlogik. [2, S. 277]

Eine Verallgemeinerung der einfachen Form der Inferenz-Regel zur Resolution kann als Regel zur kompletten Inferenz der Prädikatenlogik erster Stufe genutzt werden. [2, S. 278]

6.3 Inferenz und Resolution zur Ziehung von Schlüssen

Inferenz in der Semantik kann grundsätzlich als das Entdecken von neuen Beziehungen zwischen Entitäten beschrieben werden. Das beduetet, automatische Prozeduren, in Form von so genannten *Reasonern*, leiten neue Beziehungen ab. Wie die neuen Beziehungen generiert werden, ist eine Frage der Implementation, z.B. durch Hinzufügen zu den vorhandenen Daten oder durch einfache Rückgabe derselben (vgl. [7, Abschnitt 1]).

Reasoner sind Komponenten, welche eine Folgerung von implizitem Wissen zulassen bzw. bieten. Es handelt sich um eine Art "Verstehen" durch Maschinen. Ziel ist es, aus explizitem Wissen in Form einer Ontologie implizites Wissen zu gewinnen.

Eine detaillierte Beschreibung, wie man einen Reasoner in der Praxis umsetzt, findet sich in Form des *Pellet*-Reasoners der Firma Clark & Parsia im Abschnitt 6.4. Der *Pellet*-Reasoner basiert auf Beschreibungslogik. Eine Einführung der Grundlagen der Beschreibungslogik folgt im nächsten Abschnitt.

6.3.1 Beschreibungslogik

Beschreibungslogiken sind Formalismen um Wissen darzustellen. Sie sind dabei eine Teilmenge der Prädikatenlogik und stellen den Kern von Wissensrepräsentationssystemen dar. Sie sind eine Struktur für eine Wissensbasis und den damit verbundenen Methoden zur Folgerung (vgl. [8]).

Die Struktur, welche Beschreibungslogiken als Wissensbasis bereitstellt, besteht aus einem Schema (Tbox, Regeln) und aus Daten (Abox, Fakten).

Details zu Beschreibungslogiken finden sich unter [8].

Interpretation

"Man bezeichnet die Zuordnung von Ereignissen aus einer realen Welt zu aussagenlogischen Variablen als Interpretation." [3, S. 36]

> Sei M die Menge aller aussagenlogischen Formeln. Eine Funktion
> $I : M \to \{W, F\}$
> heisst Interpretation.

Listing 6.3: Definition einer Interpretation [1]

Modell

> Sei I eine Interpretation und X eine aussagenlogische Formel.
> Ist X unter I wahr, so bezeichnet man I als Modell von X.

Listing 6.4: Definition Modell [2]

Semantische Folgerung

"Der Zusammenhang von Formeln kann durch den Begriff der semantischen Folgerung dargestellt werden." [3, S. 39]

> Sei X eine Menge von aussagenlogischen Formeln, Y eine aussagenlogische Formel.
> Y ist eine semantische Folgerung von X
> falls jedes Modell von X auch Modell von Y ist.
> Man schreibt dafür $X \models Y$ und sagt auch Y folgt aus X.

Listing 6.5: Definition semantische Folgerung[3]

[1] [3, S. 36]
[2] [3, S. 37]
[3] [3, S. 39]

Ableitbarkeit

Es ist erforderlich, "...dass das Folgern von Formeln auf der Ebene der Gültigkeit von der Berechnung von Formeln auf der Inferenzebene unterschieden wird. Dies wird mit dem Begriff des Ableitens getan." [3, S. 42]

```
Y ist aus X ableitbar,
    X ⊢ Y
wenn eine endliche Folge von Inferenzschritten existiert,
so dass man von X zu Y gelangt.
```

Listing 6.6: Definition Ableitbarkeit[4]

Korrektheit und Vollständigkeit

"Der Bezug zwischen beiden Begriffen (semantische Folgerung und Ableitbarkeit, Anm. der Autoren) wird durch die Begriffe der Korrektheit und der Vollständigkeit hergestellt." [3, S. 43]

```
Ein Beweis-Verfahren heisst korrekt, wenn für beliebige Formeln X,Y gilt:
    Falls X ⊢ Y gilt, dann gilt auch X ⊨ Y.
Ein Beweis-Verfahren heißt vollständig, wenn für beliebige Formeln X,Y
    gilt:
    Falls X ⊨ Y gilt, dann gilt auch X ⊢ Y.
```

Listing 6.7: Definition Korrektheit und Vollständigkeit[5]

6.4 Pellet

Sofern im Text nicht anders vermerkt, basiert das nachfolgende Kapitel auf [9].

Bei Pellet handelt es sich um einen Reasoner auf Basis von Beschreibungslogik. OWL ist eine syntaktische Variante der Beschreibungslogik. Bei der Entwickelung von Pellet wurde von Anfang an nur die OWL-DL [6]-Sprache berücksichtigt. Diese ist wiederum eine Teilsprache von OWL-full, siehe Kapitel 8.

Eine komplette Unterstützung des OWL-full Profils ist generell nicht möglich, da dieses nicht entscheidbar ist. Daher beschränkt sich Pellet auf die Verwendung von OWL-DL. [9, Seite 13]

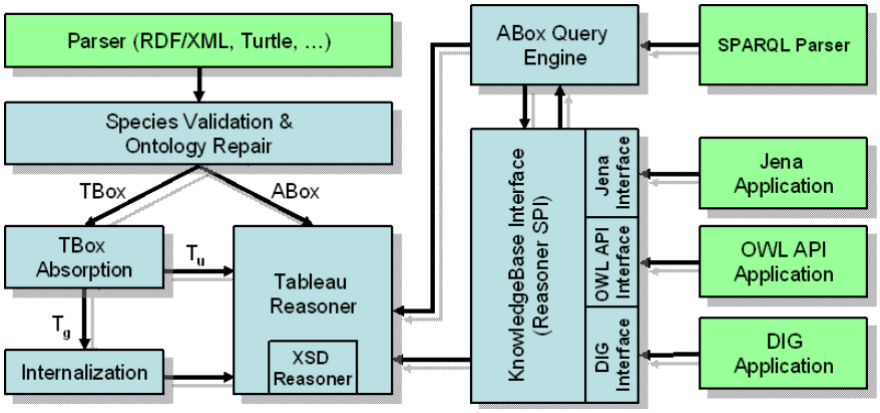


Abbildung 6.1: Hauptkomponenten des Pellet-Reasoners.[7]

Die obenstehende Abbildung zeigt die Hauptkomponenten von Pellet. Dabei ist der Tableaux-Reasoner die Kernkomponente, welche die Wissensbasis auf deren Konsistenz prüft. Der Entscheid bei der Entwicklung von Pellet, ausschliesslich OWL-DL zu verwenden, führte zu einer modularen Struktur. Module ihrerseits sind z.B. ein

[4] [3, S. 42]
[5] [3, S. 43]
[6] http://www.w3.org/TR/owl-ref/#OWLDL
[7] [9, S. 6]

Reasoner zur Datentypenprüfung eines XML-Schemas oder eine “Query-Engine”. Genaueres dazu wird in den nachfolgenden Abschnitten beschrieben.

Hier eine Übersicht der wichtigsten Komponenten von Pellet:

—	Tableaux-Reasoner	Komponente, zum Ziehen von Schlüssen durch Konsistenzprüfung einer Ontolgie.
Abox	Assertional Box	Komponente, welche Aussagen zu Individuen enthält, d.h. OWL-Fakten wie Typen, Eigenschaftswerte und logische Äquivalenz.
Tbox	Terminological Box	Komponente, welche Klassenaxiome enthält, d.h. OWL-Axiome wie z.B. Unterklassen, Gleichheit von Klassen und Klasseneinschränkungen.
KB	Knowledge Base	Eine Kombination einer Abox und Tbox, damit eine komplette OWL-Ontologie.

Tabelle 6.1: Beschreibung der wichtigsten Komponenten von Pellet[8]

6.4.1 Vorgehen des Pellet-Reasoners

Wird eine Ontologie mittels Parser geladen, wird diese auf deren Gültigkeit bezüglich OWL-DL geprüft. Während des Ladens werden Klassenaxiome in der Tbox, Aussagen über Individuen in der Abox abgelegt. Die Axiome der Tbox werden durch das Standardverfahren von OWL-DL-Reasonern vorverarbeitet. Sie werden durch verschiedene Optimierungsverfahren vereinfacht (siehe 6.4.3) und anschliessend in den Tableaux-Reasoner eingespiesen.

Laden und Parsen der Daten

Pellet bietet diverse Schnittstellen um Ontologien zu laden. Pellet selbst implementiert keinen RDF/OWL-Parser. Es ist aber in verschiedenen RDF/OWL-Werkzeugen, welche solch einen Parser anbieten, integriert. Dabei unterstützt Pellet die unterschiedlichen Datenstrukturen der Werkzeuge. Weiter stellt der Reasoner Schnittstellen zur Beantwortung von Anfragen der Werkzeuge zur Verfügung.

Tableaux-Reasoner

Der Tableaux-Reasoner hat nur eine Funktion: Eine Ontologie auf ihre Konsistenz zu prüfen. Eine Ontologie ist dann konsistent, wenn nur eine Interpretation der Ontologie existiert und sie alle Fakten und Axiome dieser erfüllt. [10] Solch eine Interpretation wird als Modell der Ontologie bezeichnet.

Der Tableaux-Reasoner sucht nach solch einem Modell durch Vervollständigung. Dieses wird inkrementell als eine Art Tafel (eben, Tableau) aufgebaut.Dieses Vorgehen wird Tableaxu-Algorithmus genannt. Die Vervollständigung beginnt mit einem initialen Graphen der Abox. Dabei repräsentieren die Knoten Individuen und Literale (z.B. Zeichenketten, Datumswerte oder auch Nummern). Jedem Knoten wird der entsprechende (Daten-) Typ zugewiesen. Die gerichteten Kanten zwischen den Knoten stellen die Eigenschaften dar.

Durch wiederholtes Anwenden der Regeln zur Erweiterung des Graphen, versucht der Reasoner einen widerspruchsfreien Graphen zu bilden. Dies tut er solange bis entweder ein Widerspruch (Kontradiktion) auftritt oder keine Regeln mehr anwendbar sind.

Der Tableaux-Algorithmus basiert auf dem Tableau-Kalkül. Dabei wird mittels Widerspruchsbeweis aufgezeigt, dass eine Interpretation existiert, in der sowohl sämtliche Anforderungen (Prämisse) als auch die Negation der Folgerung (Konklusion) wahr sind. Kann ein solches Modell gefunden werden, ist damit bewiesen, dass die ursprüngliche “Formel” falsch ist. Die in der Abox definierten Daten sind dann konsistent, wenn beim Aufbau des Baumes inklusive der Negation der Folgerung ein Widerspruch auftritt. [11]

Für das Tableau-Kalkül werden Transformationsregeln benötigt. Mit diesen Regeln wird ein Argument unter einer Interpretation wahr gemacht. Es wird also ein Modell erzeugt.[11]

Nachfolgend sind die Transformationsregeln zum Erzeugen von Modellen aufgezeigt.

Argument	**Modell**
P	I (P) = W
¬ Q	I (Q) = F
P ∧ Q	I (P) = I (Q) = W
P ∨ Q	I (Q) = W oder I (P) = W oder beides
P → Q	I (P) = F oder I (Q) = W oder beides

Tabelle 6.2: Transformationsregeln des Talbeau-Kalküls[9]

Mittels nachfolgendem Beispiel soll die Anwendung des Tableau-Kalküles verdeutlicht werden.
Gegeben sind folgende Fakten:

- $S : hatStandort(SeilparkBalmberg, Balmberg)$
- $O : hatOrt(Solothurn, Balmberg)$

Die genannten Fakten bilden die *Prämisse*. Durch Anwendung des Kalküls soll der Schluss (*Konklusion*) erreicht werden, dass sich der Seilpark in der Region Solothurn befindet:

- $R : hatRegion(SeilparkBalmberg, Solothurn)$

Es soll ~~also~~ $\Gamma \models \varphi$ gelten, wobei Γ die Menge der Prämissen und φ die Konklusion ist:

- $\Gamma : \{S \land O \rightarrow R, S, O\}$
- $\varphi : \{R\}$

Weil das Tableau-Kalkül darauf basiert, einen Widerspruch zu erreichen, wird die Negation der Konklusion ($\neg\varphi$) mit der Menge der Prämissen (Γ) vereinigt:

- $\Gamma \cup \{\neg\varphi\}$

  bzw.

- $\{S \land O \rightarrow R, S, O\} \cup \{\neg R\}$

  folgt: $\{S \land O \rightarrow R, S, O, \neg R\}$

Zur Anwendung des Kalküls werden alle Sätze der gesamten Menge untereinander aufgeschrieben:

(a) $S \land O \rightarrow R$

(b) S

(c) O

(d) $\neg R$

(a), (b), (c) und (d) bilden zusammen den *Wurzelknoten* des Graphen.

Durch Anwendung der Transformationsregeln wird der Knoten (a) zu zwei Unterkonten bzw. zu zwei Zweigen expandiert:

(e) $\neg(S \land O)$

(f) R

Nach Bildung des Wurzelknotens wird der Baum anhand der Tiefensuche-Traversierung expandiert. Zuerst wird der Wurzelknoten {(a), (b), (c), (d)}, danach der linke (e) und schliesslich der rechte Teilbaum (f) durchlaufen.

Die Transformationsregeln werden auf den Wurzelknoten angewendet. Dabei wird der linke Teilbaum (e) durch einen neu gebildeten Teilbaum (bestehend aus (g) und (h)) ersetzt, der rechte Teilbaum (f) bleibt bestehen:

(g) $\neg S$

(h) $\neg O$

Der linke Teilbaum ist somit vollständig expandiert und wird ausgewertet. Dabei werden die Knoten (b), (c), (d), (g) und (h) (logisch) vereint. Dabei ergibt sich, dass die Knoten (g) und (h) einen Widerspruch mit den Knoten (b) und (c) bilden:

- $(b) \cup (g)$ also $S \cup \neg S = \{\}$

  und

- $(c) \cup (h)$ also $O \cup \neg O = \{\}$

An dieser Stelle wird die Tiefensuche im linken Unterknoten abgebrochen, dafür in den rechten Unterknoten verlagert. Dieser ist bereits expandiert und wird ausgewertet. Dazu muss Knoten (f) mit den Knoten (b), (c) und (d) (logisch) vereinbar sein. Auch dort herrscht ~~somit~~ ein Widerspruch:

- $(d) \cup (f)$ also $\neg R \cup R = \{\}$

Obwohl alle Knoten expandiert und ausgewertet wurden, konnte kein Modell für $\{S \wedge O \rightarrow R, S, O, \neg R\}$ gefunden werden. Dies wiederum bedeutet, dass die Negation $\Gamma \models \neg\varphi$ nicht zutrifft. Somit ist $\Gamma \models \varphi$ bewiesen.

Datentypenprüfung

In OWL werden Datentypen in einem XML-Schema beschrieben. Dadurch sind viele einfache Datentypen wie numerische Datentypen (Ganz- und Fliesskommazahlen) und Zeichenketten gegeben. Zudem kann OWL eigene Datentypen definieren.

Das Modul zur Datentypenprüfung zeigt auf ob die Schnittmenge von Datentypen konsistent ist. Eine Schnittmenge von Datentypen ist dann inkonsistent, wenn diese keine Elemente gemeinsam haben.

Der Tableaux-Reasoner nutzt dieses Modul um zu prüfen, ob die Schnittmenge aller Datentypen für jeden Literal-Knoten des Graphens erfüllbar ist.

Schnittstelle zur Wissensdatenbank (KB)

Die Schnittstelle zur Wissensdatenbank entscheidet, wann die Konsistenz der Abox geprüft werden muss, wann alle Konzepte neu klassifizert werden müssen und wann alle Individuen umgesetzt werden. Alle Aufgaben zum Schlussfolgern können unter geeigneter Umwandlung auf eine Prüfung der Konsistenz der Wissensdatenbank (KB) reduziert werden.

Die Schnittstelle bietet die Möglichkeit beliebige atomare Anfragen zu beantworten. Diese können Klassen, Eigenschaften oder Individuen betreffen. Wahrheitsabfragen werden in Erfüllbarkeitsprobleme umgewandelt.

Für alle Anfragen, welche mehrere Ergebnisse liefern, sind theoretisch mehrere Konsistenzprüfungen notwendig. Da dies aber sehr aufwändig ist, werden zur Optimierung im Unterabschnitt 6.4.3 beschriebene Verfahren eingesetzt.

Die Schnittstelle zur Wissensdatenbank, wie auch der Rest der Komponenten von Pellet, bauen auf der ATerm-Bibliothek [10] auf. ATerm (Annotated Term) ist ein abstratker Datentyp, welcher für den Austausch von baumartigen Datenstrukturen zwischen verteilten Applikationen entwickelt wurde.

Abox Query-Engine

Die Schnittstelle zur Wissensdatenbank wird mit der Abox-Query-Engine verbunden. Diese beantwortet konjunktive (verknüpfte) Anfragen. Das Modul unterstützt in SPARQL oder RQDL [11] geschriebene Anfragen. Dabei bestehen jedoch gewisse Einschränkungen, Details siehe [9, Seiten 10 und 11].

Die Abox-Query-Engine besteht im Grunde aus mehreren Query-Engines, welche Anfragen beantworten. Dabei gibt es eine zentrale Query-Engine, welche Anfragen vorverabeitet und für die Beantwortung die entsprechende Query-Engine auswählt. Details zum Ablauf, siehe [9, Seite 11].

[10] https://strategoxt.org/Tools/ATermLibrary
[11] http://www.w3.org/Submission/RDQL/
[12] [9, S. 11]

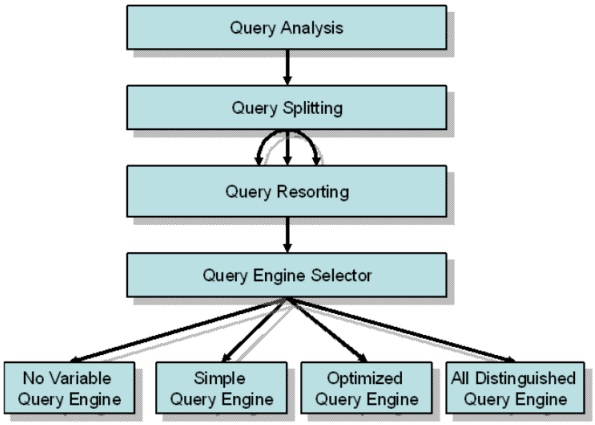


Abbildung 6.2: Ablauf der Beantwortung einer Anfrage in Abox-Query-Engine.[12]

Gültigkeitsprüfung bezüglich OWL-DL

Werkzeuge zur Modellierung und zum Export von Ontologien im OWL-Format bieten häufig Charakteristika von OWL-full zur Modellierung an. Für die Gültigkeitsprüfung einer Ontologie wird diese analysiert und gegebenenfalles mittels Heuristiken von OWL-full in OWL-DL umgewandelt.

Diese Umwandlung ist jedoch nur in gewissen Fällen möglich, so z.B. bei gleicher Bezeichnung von Klassen, Eigenschaften und Individuen, andernfalls werden die OWL-full Charakteristika ignoriert oder der Prozess wird abgebrochen.

6.4.2 Behandlung von Regeln in Pellet

Pellet ermöglicht die Verwendung von Regeln zur Schlussfolgerung. Dies unter Verwendung der SWRL-Regelsprache.

Damit in der Wissensdatenbank gespeicherte Konzepte in Regeln verfügbar sind, wird das $\mathcal{AL}$-Log-Framework eingesetzt. Dieses verbindet Beschreibungslogik mit Regeln. [12, Seiten 4 und 5] $\mathcal{AL}$-Log ist ein integriertes System zur Repräsentation von Wissen, welches auf der Beschreibungslogik $\mathcal{AL}$ und der deduktiven Datenbanksprache Datalog basiert. [13]

Bei $\mathcal{AL}$ handelt es sich um eine minimale Sprache der Beschreibungslogik, Details siehe [8, Seite 51]. Als Implementation des $\mathcal{AL}$-Log-Frameworks nutzt Pellet einen Datalog-Reasoner. [12, Seiten 4 und 5]

Bei Datalog handelt es sich um eine deklarative Logiksprache, in welcher jede Formel eine funktionsfreie Hornklausel ist. Im Unterschied zu Prolog muss jede Variable, die im Kopf einer Klausel vorkommt, auch im Körper der Klausel vorkommen. Die Reihenfolge der Klauseln spielt keine Rolle. Alle Anfragen terminieren und jede mögliche Antwort wird ausgegeben. [14]

In der Pellet-Implementation wird das $\mathcal{AL}$-Log-Framework dahingehend erweitert, dass es die $\mathcal{SHOIQ}(\mathcal{D})$ Variante der $\mathcal{AL}$-Beschreibungslogik nutzen kann. Zusätzlich erlaubt die Erweiterung die Verwendung von OWL-Datentypen und SWRL-Funktionen im Körper von Datalog-Regeln. [12, Seite 5]

Bei $\mathcal{SHOIQ}(\mathcal{D})$ handelt es sich um eine Erweiterung der Beschreibungslogik $\mathcal{AL}$. $\mathcal{SHOIQ}(\mathcal{D})$ unterstützt unter anderem zusätzliche Konzepte wie Rollenhierarchien, inverse Eigenschaften und qualifizierte Kardinalitätseinschränkungen. Diese Erweiterung erlaubt zusätzlich die Verwendung von Datentypen, Datenwerten und Datentypeneigenschaften. [15]

6.4.3 Optimierungen

Dieser Abschnitt basiert, sofern nicht anders im Text vermerkt, auf [9, Seiten 16 bis 19]

Beschreibungslogiken, wie $\mathcal{SHOIQ}(\mathcal{D})$ haben im ungünstigsten Fall eine sehr hohe Komplexität. Daher besteht ein grosser Unterschied zwischen Design und praktischer Umsetzung einer Entscheidungsprozedur.

Um dennoch akzeptable Leistungen bei Schlussfolgerungen mittels Beschreibungslogik zu erhalten, nutzen moderne Reasoner verschiedene Arten von Optimierungen, so z.B.:

- **Normalisierung und Vereinfachung**
  Alle Konzepte der Wissensdatenbank werden in eine Form gebracht, die hilft, dass Widersprüche bei der Tableaux-Erweiterung möglichst früh erkannt werden. Die Konzepte werden in die Normalform gebracht. Diese Vereinfachung erkennt offensichtliche Fehler während der Normalisierung. Mehrfaches Vorkommen eines gleichen Konzeptes wird eliminiert.
- **Tbox-Absorption**
  Bei dem Prozess der Tbox-Absorption wird ebenfalls versucht, gleichartige Konzepte (GCI — General Concept Inclusion) zu eliminieren, dies mittels Ersatz durch atomare Konzepte.
- **Dependency-directed Backjumping**
  Der Prozess des dependency-directed Backjumpings eliminiert unproduktive Backtracking-Suchen, indem er Verzweigungspunkte identifiziert welche Konflikte verursachen. Er springt zurück und überspringt dabei die Konfliktpunkte ohne nach Alternativen zu suchen.

Weitere Verfahren zur Optimierung sowie mehr Details finden sich unter [9, S. 17 bis 19].

6.4.4 Unterschiede zwischen Pellet und Prolog

Wie unter 6.4.2 erwähnt, nutzt Pellet einen Datalog-Reasoner als Implementation des $\mathcal{AL}$-Log-Frameworks. Der wohl wichtigste Unterschied zu Prolog ist daher, dass in Datalog alle Anfragen terminieren und jede mögliche Antwort ausgegeben wird. Eine Reihenfolge der Klauseln spielt daher keine Rolle. Im Gegensatz dazu ist eine Termination von Anfragen in Prolog nicht immer gegeben. So sind Endlosschleifen je nach Reihenfolge der Regeln durchaus möglich. [3, Seite 175]

Ein weiterer Unterschied ist die Art, wie Schlüsse gezogen werden. Prolog basiert auf dem SLD-Resolutionsverfahren [Details siehe 3, Seite 68]. Im Gegensatz dazu kommt bei Pellet der unter 6.4.1 erklärte Tableau-Algorithmus zum Einsatz.

Durch den Aufbau von Prolog ist es möglich so genannte Constraint-Satisfaction-Probleme (Bedingungserfüllungsprobleme) zu lösen [Details siehe 3, Seite 148]. Mit Pellet bzw. Ontologien und Regeln ist dies nicht der Fall. Versuche dazu finden sich den Publikationen von Xiong and Jiang [16], Sleeman and Chalmers [17] und Croitoru and Compatangelo [18].

Das unter 6.4.1 genannte Beispiel beantwortet exakt die unter 4.3 gestellte Frage: Wie gelangt man zum Schluss, dass das Individuum *Seilpark Balmberg* in der *Region Solothurn* liegt.

Wie gelangt man effektiv zu dieser Information? Kann diese durch reine Folgerung erreicht werden?

Die Antwort hierzu lautet ja und nein. Modelliert man die Situation beispielsweise in Protégé und unter Benützung des Pellet-Reasoners für Schlussfolgerungen, so nimmt man an, dass er dies finden kann. Aufgrund seiner Möglichkeiten sollte der Reasoner dies beantworten können. Die nachfolgende Grafik zeigt jedoch, dass dies nicht immer der Fall ist.

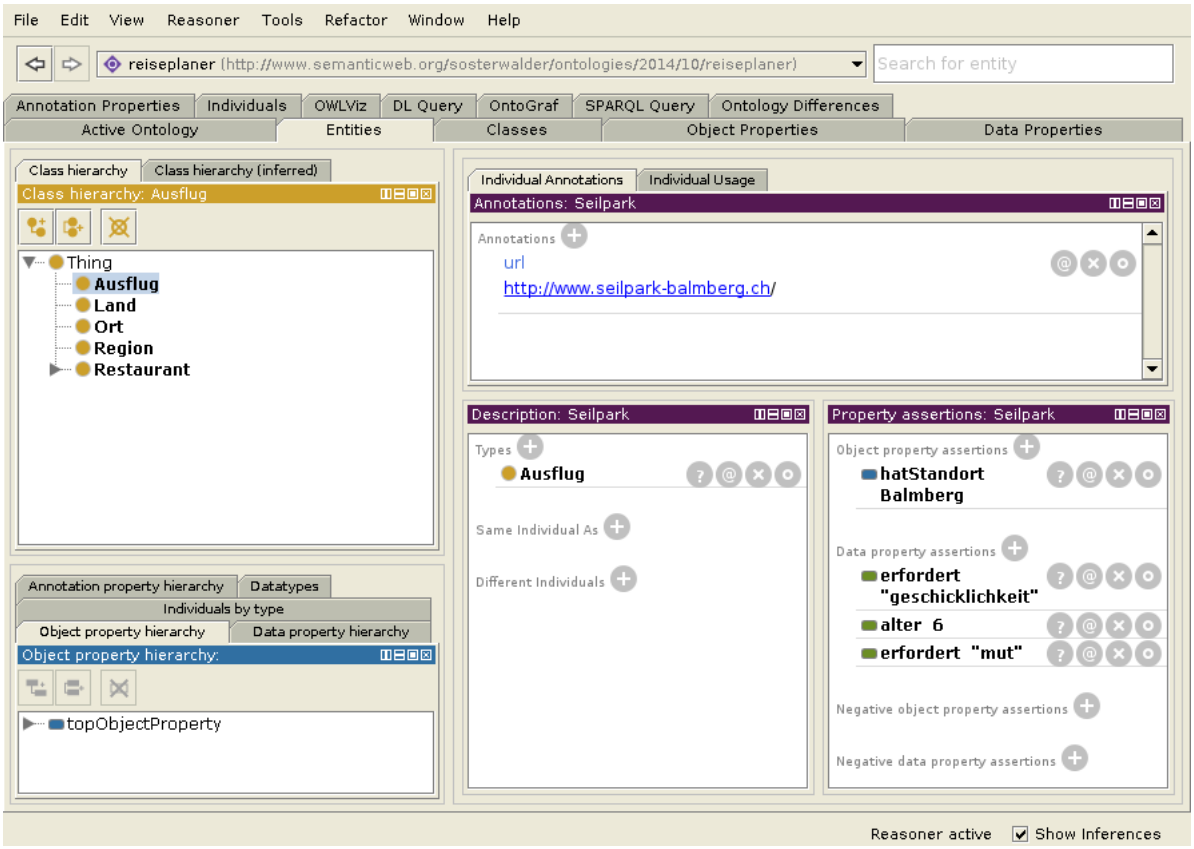


Abbildung 6.3: Darstellung des Individuums *Seilpark Balmberg* in Protégé.[13]

Wenn man den Relationen die entsprechenden Eigenschaften wie Symmetrie oder Transitivität gibt, ist diese Folgerung durchaus möglich. Bei unserem Beispiel haben wir die Eigenschaften bewusst weggelassen, da andernfalls irreführende Folgerungen auftreten. So wäre z.B. ein Ort auch ein Land und umgekehrt.

Die eigentliche Folgerung haben wir mittels einer Regel vorgenommen. Zu einem späteren Zeitpunkt erklären wir genauer, siehe Kapitel 9.

[13] Eigene Darstellung mittels Stanford Protégé Version 5.0.0 beta 15

7 RDF

Das folgende Kapitel basiert auf der Spezifikation des W3C [19].

Die bisherigen Kapitel bieten einen Überblick über die verschiedenen Methoden und Angehensweisen des knowledge engineerings. Die wichtigsten theoretischen Grundlagen wurden in den vorigen Kapiteln erklärt. Im Kapitel 5, " Wissen abbilden mittels Ontologien", wurde gezeigt, dass für die Darstellung von Ontologien verschiedene Sprachen entwickelt wurden.

Wir haben uns für OWL (in der Version 2) entschieden, da es sich um die meistbenutzte Ontologiesprache handelt. OWL wird in der RDF-Syntax verfasst wird, wollen wir letztere erklären.

Das "Resource Description Framework" (RDF) ist ein Framework um Informationen aus Ressourcen zu formulieren. Als Ressourcen können Dokumente, Leute, Objekte aber auch abstrakte Inhalte dienen. Im Web können Informationen mit RDF ~~auch~~ verarbeitet werden, anstatt diese nur anzuzeigen. RDF bietet ein gemeinsames Framework um die Informationen zwischen Anwendungen auszutauschen ohne dabei die Bedeutung der Informationen zu verändern.

RDF ist die Grundlage des sematic Web, welches die Flexibilität von RDF vollumfänglich ausnutzt. Alle Daten im semantic Web werden in RDF abgebildet. RDF ermöglicht die Verknüpfung von Daten. Dadurch werden für eine Ressource mehr Informationen zusammen gefügt.[20]

7.1 RDF Data Model

In RDF werden Informationen als Aussagen abgebildet. Der Aufbau dieser ist immer gleich und weist die folgende Struktur eines Tripels auf:

```
<Subjekt> <Prädikat> <Objekt>
```

Listing 7.1: Tripel-Struktur einer RDF-Aussage

Eine RDF Aussage bildet eine Beziehung zwischen zwei Ressourcen (Entitäten) nämlich Subjekt und Objekt ab. Das Prädikat repräsentiert die Beziehung zwischen den zwei Ressourcen. Die Beziehung wird in RDF als Eigenschaft (Property) abgebildet.

```
<Seilpark> <hatStandort> <Balmberg>
```

Listing 7.2: Beispiel einer RDF-Aussage

Eine Entität kann in mehreren Tripeln referenziert werden. Zudem ist es möglich die gleiche Ressource in einer Aussage als Objekt, in einer anderen Aussage als Subjekt zu verwenden. Damit werden Verbindungen zwischen mehreren Tripeln möglich. Dies ist ein wichtiger Aspekt von RDF.

Tripel werden in sogenannten RDF-Graphen abgebildet. Diese bestehen aus Knoten und Pfeilen. Subjekte und Objekte werden als Knoten, Prädikate als Pfeile dargestellt. Genaueres dazu findet sich im Kapitel 3, "Graphrepräsentationen".

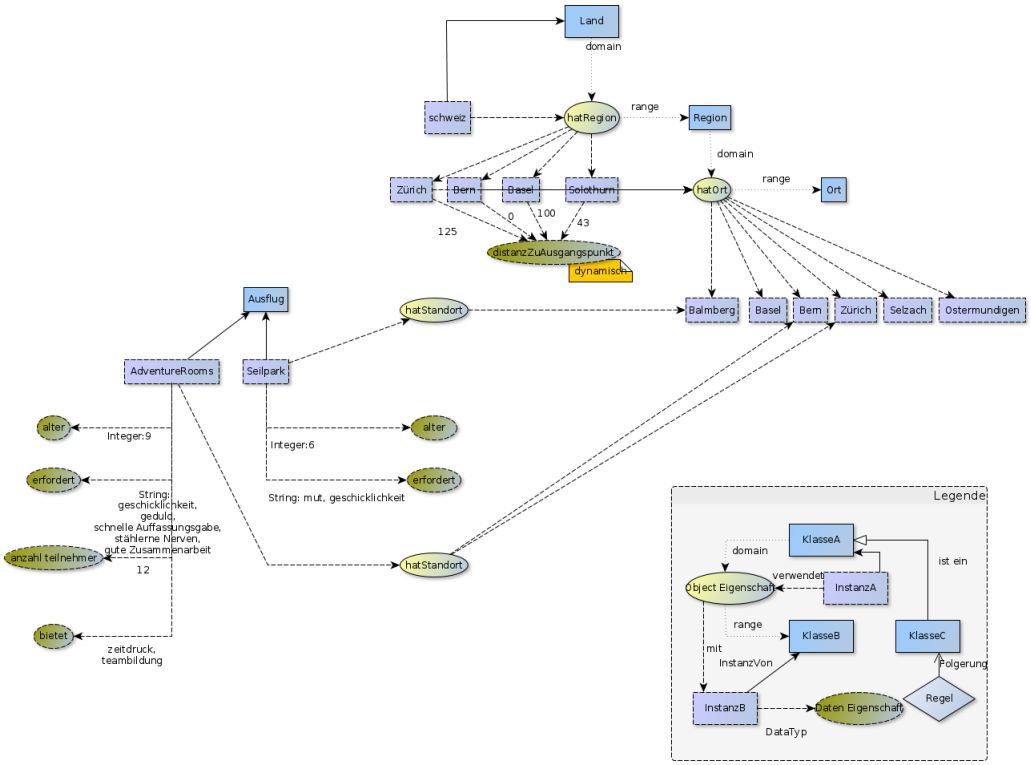


Abbildung 7.1: Ausschnitt der Ontologie des Reisplaners[1]

Man unterscheidet drei Typen von RDF-Daten, welche in Tripeln auftreten: Ressource-Knoten (IRIs), leere Knoten (Blank) und Literale.

7.1.1 IRIs (International Resource Identifier)

Wie der Name besagt, stellt ein IRI eine Ressource dar. Dabei handelt es sich um einen globalen Identifier, IRIs können also von verschiedenen Nutzern wiederverwendet werden. Es gibt verschieden Formen von IRIs. Zum Beispiel werden URLs als Web-Adresse verwendet werden. Eine andere Form der IRIs bietet eine Kennung einer Ressource ohne deren Standort oder Zugriff preiszugeben. IRIs können in allen drei Positionen eines Tripels auftreten.

Die genaue Spezifikation von IRIs findet sich unter RFC 3987.

7.1.2 Literale Knoten

Beim Begriff Literal handelt es sich um ein Synonym für Werte. Literale sind Basiswerte, die nicht IRIs entsprechen. Für die richtige Werteinterpretation wird den Literalen ein Datentyp zugeordnet. Dies können Strings, Datumswerte oder auch Nummern sein. Einem String kann zusätzlich eine Sprache zugewiesen werden.

Literale können in einem Tripel ~~können~~ nur als Objekt verwendet werden.

7.1.3 Leere Knoten (blank nodes)

Ein leerer Knoten stellt eine Ressource ohne URI dar. Vorteil: Diese Knoten benötigen keinen globalen Identifier. Leere Knoten können mit einer einfachen Variablen in der Algebra verglichen werden. Sie bilden ein Objekt ab, wobei der Wert irrelevant ist.

[1] Eigene Darstellung mittels yEd.

Leere Knoten können in einem Tripel Subjekt oder Objekt darstellen.

7.2 Multiple Graphs

Eine der neuesten Erweiterung von RDF sind multipe Graphen. Diese wurden eingeführt, um Teilmengen einer Tripelsammlung zu definieren. Ursprünglich stammt dieser Mechanismus von der Abfragesprache SPARQL (siehe 10 SPARQL). Man unterscheidet zwischen benannten (named) und unbenannten (unnamed) Graphen. Bei unbenannten Graphen enthalten die Tripel jeweils die gesamte URI.

```
<http://example.org/bob> <is published by> <http://example.org>.
```

Listing 7.3: Beispiel eines unbenannten (unnamed) Graphen

Bei benannten Graphen wird ein Identifikator (identifier) hinzugefügt, auf welchen referenziert wird.

```
Identifier:
    http://example.org/bob
Graph:
    <Bob> <is a> <person>.
    <Bob> <is a friend of> <Alice>.
```

Listing 7.4: Beispiel eines benannten (named) Graphen

Multipe Graphen eines RDF-Dokumentes stellen eine Datenmenge dar. Sie bestehen standartmässig aus einem unbenannten (unnamed) Graphen und mehreren benannten (named) Graphen. Dabei ist der unbenannte Graph der sog. Basisgraph (default).

7.3 RDF Vokabular

RDF wird typischerweise in Kombination mit einem Vokabular oder anderen Konventionen verwendet, welche semantische Informationen über verwendete Ressourcen bieten.

Um Vokabulare zu definieren, bietet RDF die RDF-Schema-Sprache. Erst diese ermöglicht es semantische Eigenschaften von RDF-Daten zu definieren. Damit kann beispielsweise festgelegt werden, welche Ressourcen an welcher Position eines Tripels verwendet werden sollen.

RDF verwendet den Klassen-Bezeichner (class) um Kategorien zu definieren, welche die Klassifizierung von Ressourcen erlauben. Die Beziehung zwischen einer Instanz und ihrer Klasse wird durch den Typen-Bezeichern (type) ausgedrückt.

Mit der RDF-Schema-Sprache können Hierarchien im Bereich der Klassen und Sub-Klassen sowie der Eigenschaften und Untereigenschaften gebildet werden. Auf Subjekten bzw. Objekten können Typeneinschränkungen mittels dem Domänen- (domain) bzw. dem Bereichs-Bezeichner (range) vorgenommen werden. [2]

7.4 RDF Formen

RDF kann in verschiedenen Formen niedergeschrieben werden. Sie alle führen zu den exakt selben Tripeln, sind also logisch äquivalent. Die Einsatzgebiete sind jedoch unterschiedlich. Formen von RDF sind: Turtle-Schreibweise (N-Triples, Turtle, TriG und N-Quads), JSON-LD, RDFa und RDF/XML.

Da wir letztere Schreibweise in unserem Beispiel verwendet haben, werden wir sie genauer vorgestellen.

[2] http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/#section-rdfa [19]

7.4.1 RDF/XML

Bei RDF/XML handelt es sich um eine Schreibweise von RDF, welche die XML-Syntax verwendet. Bei der Entwicklung von RDF in den 1990er Jahren, war RDF/XML die einzig existierende Schreibweise.

In RDF/XML werden Tripel in dem XML-Element, *rdf:RDF*, spezifizert. Das XML-Element *rdf:description* definiert eine Menge von Tripeln, welche als Subjekt den IRI ihres *about*-Attributes haben.

Ein Description-Element kann Unterelemente beinhalten. Der Name des Unterelementes ist ein IRI, welcher in der RDF-Eigenschaft *rdf:type* abgebildet ist. Dabei repräsentiert jedes Unterelement ein Tripel.

Handelt es sich bei dem Objekt eines Tripels auch um einen IRI, hat das Unterelement keinen Inhalt und der Objekt-IRI-Knoten wird durch das *rdf:resource*-Attribut beschrieben.

Ist das Objekt eines Tripels ein Literal, wird der Inhalt der RDF-Eigenschaft zum Literalwert.
Der Datentyp der RDF-Eigenschaft wird durch als Attribut angegeben. Bei Fehlen von Datentyp und Sprache wird angenommen, dass der Literal vom Datentyp “String” ist.

```
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
    <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>


<rdf:RDF xmlns="http://www.semanticweb.org/mira/ontologies/2014/9/FamilyOnto#"
     xml:base="http://www.semanticweb.org/mira/ontologies/2014/9/FamilyOnto"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <owl:Ontology rdf:about="http://www.semanticweb.org/mira/ontologies
        /2014/9/FamilyOnto"/>
    <rdf:Description>
        <rdf:type rdf:resource="&swrl;IndividualPropertyAtom"/>
        <swrl:propertyPredicate rdf:resource="http://www.semanticweb.org/mira/
            ontologies/2014/9/FamilyOnto#isAncestor"/>
        <swrl:argument1 rdf:resource="urn:swrl#a"/>
        <swrl:argument2 rdf:resource="urn:swrl#x"/>
    </rdf:Description>
</rdf:RD>
```

Listing 7.5: Beispiel RDF Elemente[3]

[3] Eigenes Beispiel

8 OWL

Im letzten Kapitel wurden Voraussetzungen geschaffen um die eigentliche Ontologiesprache, OWL zu analysieren. In diesem Kapitel, welches auf der W3C-Spezifikation [21] basiert, wird OWL näher betrachtet.

Bei OWL (Web Ontology Language) handelt es sich um eine Ontologiesprache für das semantische Web. Mit dieser Sprache können Ontologien beschrieben werden. [22] Wie RDF wurde auch OWL nicht nur für die Anzeige sondern auch für die Verarbeitung von Informationen entwickelt.

Durch zusätzliches Vokabular, wie Beziehungen zwischen Klassen, erweiterter formaler Semantik hat der Benutzer mehr Möglichkeiten als bei RDF/XML oder auch XML.

8.1 OWL Syntax

Für OWL stehen verschiedene Schreibweisen zur Verfügung, welche für verschiedene Anwendungen definiert wurden:

- *RDF/XML Syntax für OWL*

  Entspricht der RDF/XML Syntax mit einer spezifischen Übersetzung für OWL-Konstrukte. Es ist die einzige Syntax, welche von allen OWL2-Werkzeugen unterstützt wird (OWL2 ist die letzte Version von OWL).

- *OWL/XML Syntax*

  XML Syntax für OWL.

- *Functional-Style Syntax*

  Sie dient der Vereinfachung der ursprünglichen Spezifikation. Sie unterstützt Grundlagen zur Implementation von OWL2 durch APIs und Reasoners.

- *Turtle Syntax*

  Bei der Turtle Syntax handelt es sich um eine textuelle Syntax, welche die Beschreibung eines RDF-Graphen in kompakter und natürlicher Form erlaubt.

- *Manchester Syntax*

  Vereinfacht das Lesen und Verstehen für Leser ohne Kenntnisse der Prädikatenlogik.

Anmerkung: Es existieren Werkzeuge, welche eine Übersetzung zwischen den verschiedenen Schreibweisen zulassen.

Bei OWL handelt es sich nicht um eine Schemasprache. Im Gegensatz zu XML beschreibt OWL nicht, wie ein Dokument aufgebaut sein muss. So kann das Vorhandensein eines bestimmten Elementes nicht vorgeschrieben werden.

```
<#green-goblin>
    rel:enemyOf <#spiderman> ;
    a foaf:Person ;    # in the context of the Marvel universe
    foaf:name "Green Goblin".
```

Listing 8.1: Beispiel der Turtle Syntax[1]

[1] Beispiel von `http://www.w3.org/TR/turtle`

8.2 Wissen modellieren

Bei OWL handelt ist eine wissensbasierte Repräsentationssprache. Sie widerspiegelt das Wissen einer spezifischen Domäne. Dabei wird versucht, eine Domäne mittels OWL so abzubilden, dass Teile dem menschlichen Wissen entsprechen. Eine Möglichkeit alle Aspekte des menschlichen Wissens abzubilden besteht aber nicht. Trotzdem kann OWL als potente Modellierungssprache bezeichnet werden. Das Ergebnis einer solchen Modellierung ist eine Ontologie, wie bereits ausgeführt.

Für die Wissensabbildung über OWL werden grundlegende Konzepte wie Axiome, Entitäten und Ausdrücke (Expressions) benützt.

- *Axiom*

  Grundlegende Aussage.

  Grundaussagen oder Basispropositionen wie "Abenteuerreisen sind Reisen" werden in OWL Axiome genannt. Es handelt sich um "Teile des Wissens", welche je nach Sachlage wahr oder falsch sein können. Dies ist ein essentieller Unterschied zu Entitäten und Ausdrücken.

- *Entitäten*

  Elemente, welche konkrete Objekte aus der realen Welt abbilden.

  Grundbestandteile einer Aussage werden Entitäten genannt, zum Beispiel Objekte wie "Abenteuerreise" und "Reise" oder Beziehungen wie "sind".
  Konkrete Objekte werden dabei als Individuen (Individuals), Kategorien (für Individuen) als Klassen und Beziehungen (zwischen Individuen) als Eigenschaften (Properties) bezeichnet.
  Beziehungen werden unterteilt in Relationen zwischen zwei Individuen (ObjectProperties), spezifischen Datenwerten von Objekten (DataProperties) und zusätzliche Informationen zu einem Objekt (AnnotationProperties).

- *Expression (Ausdruck)*

  Ausdrücke sind Kombinationen aus Entitiäten. Sie ermöglichen den Aufbau von komplexen Beschreibungen.

  Entitäten, wie z.B. Klassen, können mit Hilfe eines Konstruktors kombiniert werden. Beispiel: "Abenteuerreise" und "Reise". Diese Kombination wird als ClassExpression abgebildet und ist eine neue Entität.

  Ausdrücke (Ausdruckssprache) sind für Klassen sehr vielfältig, hingegen nicht für Eigenschaften.

```
EquivalentClasses(
    :Restaurant
    :Gaststätte
)
```

Listing 8.2: Beispiel eines Ausdrucks: Zwei Klassen bedeuten das Gleiche

8.3 Die wichtigsten Elemente von OWL

8.3.1 Klassen, Subklassen und Individuen

Klassen werden dazu verwendet, um Individuen mit Gemeinsamkeiten zu gruppieren. Klassen stellen daher eine Menge von Individuen dar. Für die Abbildung der menschlichen Denkweise, werden in der Modellierung Klassen verwendet, beispielsweise Reisen oder Abenteuerreisen.

Beispiel: *Abenteuerreise* ist (nach unserer Definition) eine Klasse.

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#Ausflug -->
<owl:Class rdf:about="&reiseplaner;Ausflug"/>
```

Listing 8.3: Beispiel einer Klasse: Ausflüge

Subklassen Im obigen Abschnitt wurden die Klassen Reise sowie Abenteuerreise eingeführt. Der menschliche Leser weiss intuitiv, dass eine Abenteuerreise auch eine Reiseform ist.

In OWL stehen diese beiden Klassen jedoch nicht in Beziehung. Es handelt sich um zwei unterschiedliche Klassen mit unterschiedlichen Bezeichnungen. Um die Schlussfolgerung, eine Abenteuerreise ist auch eine Reise, zu erreichen, muss dies definiert werden. In OWL geschieht dies mit dem Subklassen-Axiom *subClassOf*.

Beispiel: *Landgasthaus* ist (nach unserer Definition) Subklasse von *Restaurant*.

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#Landgasthaus -->
<owl:Class rdf:about="&reiseplaner;Landgasthaus">
    <rdfs:subClassOf rdf:resource="&reiseplaner;Restaurant"/>
</owl:Class>
```

Listing 8.4: Beispiel einer Subklasse

Nicht nur zur Darstellung von Abhängigkeiten werden Subklassen verwendet, sondern auch zur Modellierung von Klassenhierarchien. Mittels Subklassen werden allgemeine Beziehungen der Klassen abgebildet, so zum Beispiel die Relation "ein Landgasthaus ist ein Restaurant."

Bei **Individuen** handelt es sich um *Instanzen von Klassen*. Ein Individuum ist Mitglied der Menge einer Klasse und bildet daher deren Konzept ab. Individuuen können gleichzeitig Mitglied von mehreren Klassen sein.

Beispiel: *Seilpark Balmberg* ist (nach unserer Definition) ein Individuum der Klasse *Ausflug*.

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#Seilpark -->
<owl:NamedIndividual rdf:about="&reiseplaner;Seilpark">
    <rdf:type rdf:resource="&reiseplaner;Ausflug"/>
    <reiseplaner:alter rdf:datatype="&xsd;integer">6</reiseplaner:alter>
    <reiseplaner:erfordert>mut</reiseplaner:erfordert>
    <reiseplaner:erfordert>geschicklichkeit</reiseplaner:erfordert>
    <reiseplaner:url>http://www.seilpark-balmberg.ch/</reiseplaner:url>
    <reiseplaner:hatStandort rdf:resource="&reiseplaner;Balmberg"/>
</owl:NamedIndividual>
```

Listing 8.5: Beispiel eines Individuums

Mittels der *owl:sameAs-Relation* kann ausgedrückt werden, dass es sich bei zwei oder mehreren Individuen um die gleichen Individuen handelt.

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#Seilpark -->
<owl:NamedIndividual rdf:about="&reiseplaner;Seilpark">
    <rdf:type rdf:resource="&reiseplaner;Ausflug"/>
    ...
    <owl:sameAs rdf:resource="&reiseplaner;Seilpark_Balmberg"/>
</owl:NamedIndividual>
```

Listing 8.6: Beispiel einer Gleichstellung von Individuen: Seilpark und Seilpark-Balmberg sind das gleiche Individuum

Durch die *owl:AllDifferent-Relation* kann das Gegenteil ausgesagt werden: Zwei oder mehrere Individuen sind nicht die gleichen Individuen.

```
<rdf:Description>
    <rdf:type rdf:resource="&owl;AllDifferent"/>
    <owl:distinctMembers rdf:parseType="Collection">
        <rdf:Description rdf:about="&reiseplaner;Seilpark"/>
        <rdf:Description rdf:about="&reiseplaner;Seilpark_Pilatus"/>
    </owl:distinctMembers>
</rdf:Description>
```

Listing 8.7: Beispiel einer Differenzierung von Individuen: Seilpark ist nicht das gleiche Individuum wie Seilpark-Pilatus

Eigenschaften

Objekt-Eigenschaften (ObjectProperties)

Objekt-Eigenschaften beschreibt die Beziehung zwischen zwei Individuen.

```
<owl:ObjectProperty rdf:about="&reiseplaner;hatOrt">
    <rdfs:range rdf:resource="&reiseplaner;Ort"/>
    <rdfs:domain rdf:resource="&reiseplaner;Region"/>
</owl:ObjectProperty>
...

<owl:Thing rdf:about="&reiseplaner;Bern">
    <rdf:type rdf:resource="&owl;NamedIndividual"/>
    <reiseplaner:distanzZuAusgangpunkt rdf:datatype="&xsd;integer">0</
        reiseplaner:distanzZuAusgangpunkt>
    <reiseplaner:hatOrt rdf:resource="&reiseplaner;Bern"/>
    <reiseplaner:hatOrt rdf:resource="&reiseplaner;Ersigen"/>
</owl:Thing>
```

Listing 8.8: Beispiel einer Objekteigenschaft *hatOrt* und deren Anwendung

Durch Auswahl von Klassen kann eingeschränkt werden, auf welche Individuen eine Objekt-Eigenschaft angwendet werden darf. Dies gilt sowohl für die Quelle einer Beziehung (Domäne, Domains) als auch für das Ziel einer Beziehung (Wertebereich, Ranges).

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#hatRegion -->
<owl:ObjectProperty rdf:about="&reiseplaner;hatRegion">
    <rdfs:domain rdf:resource="&reiseplaner;Land"/>
    <rdfs:range rdf:resource="&reiseplaner;Region"/>
</owl:ObjectProperty>
```

Listing 8.9: Beispiel von Einschränkungen der Objekteigenschaft *hatRegion*

Subeigenschaften

Analog zu Subklassen lassen sich Untereigenschaften (Subproperties) definieren.

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#hatGemeinde -->
<owl:ObjectProperty rdf:about="&reiseplaner;hatGemeinde">
    <rdfs:subPropertyOf rdf:resource="&reiseplaner;hatRegion"/>
</owl:ObjectProperty>
```

Listing 8.10: Beispiel der Objekteigenschaft *hatGemeinde* als Subeigenschaft von *hatRegion*

Datentypen-Eigenschaften (DataProperties)

Datentypen-Eigenschaften definieren und beschreiben einen spezifischen Datenwert eines Individuums. Dies können beispielsweise das Alter oder die Grösse einer Person sein. Datentypen-Eigenschaft können frei definiert werden und, bei Bedarf, an vordefinierte Wertetypen gebunden werden. Eine Auflistung vordefinierter Wertetypen findet sich unter [2].

```
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#anzahlTeilnehmer -->
<owl:DatatypeProperty rdf:about="&reiseplaner;anzahlTeilnehmer">
    <rdfs:range rdf:resource="&xsd;integer"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topDataProperty"/>
</owl:DatatypeProperty>
...
<!-- http://www.semanticweb.org/sosterwalder/ontologies/2014/10/
    reiseplaner#AdventureRooms -->
<owl:NamedIndividual rdf:about="&reiseplaner;AdventureRooms">
    ...
    <reiseplaner:anzahlTeilnehmer rdf:datatype="&xsd;integer">12</
        reiseplaner:anzahlTeilnehmer>
    ...
</owl:NamedIndividual>
```

Listing 8.11: Beispiel der Datentypen-Eigenschaft *anzahlTeilnehmer* und deren Anwendung bei einem Individuum

[2] http://www.w3.org/TR/owl2-syntax/#Datatype_Maps

8.4 Ontologien

Wissensdomänen zu einem Thema werden in Ontologien abgebildet. Diese können für verschiedenen Anwendungen genutzt werden.

```
<owl:Ontology rdf:about="http://www.semanticweb.org/sosterwalder/
    ontologies/2014/10/reiseplaner"/>
```

Listing 8.12: Beispiel einer Definition einer Ontologie

Ontologien werden als OWL-Dokumente abgespeichert. Dabei steht es dem Autor frei, sie im Internet zur Verfügung zu stellen. Namespaces können zusätzlich beim Schreiben einer Ontologie verwendet werden. Sie dienen der Unterscheidbarkeit von Ontologien. Beispiel: "*meineOntologie#*". Um eine Onotlogie anhand deren Namespace benutzten zu können, wird ein Präfix definiert und dieser einer Ontologie zugewiesen. Beispiel: "*PREFIX hallo: <meineOntologie#>*". Somit können die Elemente der Ontologie mit dem Präfix angesprochen werden. Beispiel: "*hallo:objekt1DerOntologie*".

Informationen aus einer beliebigen Ontologie können in einer anderen verwendet werden. Dafür ist ein Import der Ontologie nötig.

```
<owl:Ontology rdf:about="http://example.com/owl/families\">
  <owl:imports rdf:resource="http://example.org/otherOntologies/families.
      owl" />
</owl:Ontology>
```

Listing 8.13: Beispiel eines Importes einer Ontologie

Um bei der Verwendung von Objekten aus anderen Ontologien eine Umbenennung zu umgehen, kann direkt auf das entsprechende Objekt referenziert werden.

```
<owl:DatatypeProperty rdf:about="hasAge">
  <owl:equivalentProperty rdf:resource="\&otherOnt;age"/>
</owl:DatatypeProperty>
```

Listing 8.14: Beispiel einer Referenzierung auf ein Objekt einer externen Ontologie

8.5 OWL Untersprachen

OWL ist in drei Untersprachen aufgeteilt: OWL Lite, OWL DL und OWL full. Jede dieser Sprachen wurde für verschiedene Anwendergruppen entwickelt.

Die Version 2 von OWL unterscheidet zusätzlich zwischen OWL2 QL, OWL2 EL und OWL2 RL. Dies sind weitere Unterklassen von OWL2 DL, welche ihrerseits wiederum eine Untersprache von OWL2 full ist.

Genauere Informationen über die einzelnen Untersprachen finden sich unter [3], über OWL2-Profile unter [4].

Die drei wichtigsten Unterklassen werden nachfolgend kurz vorgestellt.

- *OWL Lite*

  "Bietet primär eine Hierarchie zur Klassifikation sowie einfache Bedingungen, so wird zwar z.B. Kardinalität geboten, jedoch nur mit den Werten 0 und 1." [23, Seite 12]

- *OWL DL*

  "Bietet das Maximum an Ausdrucksmöglichkeiten unter Einhaltung der Vollständigkeit (alle Folgebeziehungen werden berücksichtigt und miteinbezogen) und der Entscheidbarkeit (alle Berechnungen enden nach endlicher Zeit)." [23, Seite 12]

[3] http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.3
[4] http://www.w3.org/TR/owl2-profiles/

- *OWL full*

  "Bietet das Maximum an Ausdrucksmöglichkeiten und die syntaktische Freiheit von RDF aber ohne Garantien betreffend Vollständigkeit und Entscheidbarkeit." [23, Seite 12]

8.6 OWL-Werkzeuge

Typischerweise unterscheidet man zwischen zwei Arten von Ontologiewerkzeugen, einem Editor und einem Reasoner. Der Editor ermöglicht das Erstellen und Ändern von Ontologien. Der Reasoner leitet logische Folgerungen aus dem bestehenden Wissen ab.

Die Ontologiesprachen erlauben Ontologien zu modellieren. Für die Formulierung und das Lesen aber sind sie umständlich. Mit Hilfe von Ontologiewerkzeugen, zum Beispiel einem Editor, lassen sich Ontologien intuitiv und übersichtlich modellieren.

In unserem Beispiel haben wir den in der Fachwelt häufig verwendeten Editor Protégé der Universität Stanford benützt.

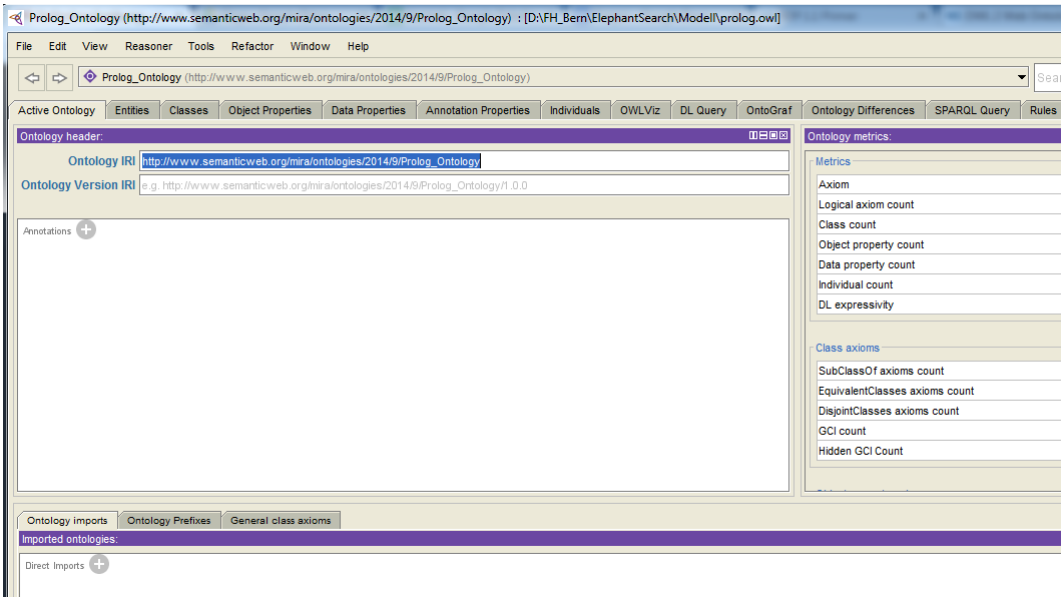


Abbildung 8.1: Übersichtsfenster des Protégé-Editors[5]

[5] Eigens erstellter Screenshot von Protégé

9 Regeln — SWRL

Das vorangehende Kapitel 8, "OWL" hat gezeigt, wie mittels der Ontologiesprache OWL in der Version 2 eine Ontologie erstellt werden kann. Weiter zeigte das Kapitel wie innerhalb einer Ontologie Klassen, Individuen und Beziehungen aufgebaut werden können. Beim Erstellen solch einer Ontologie, fällt auf, dass der Nutzen der Inferenz mit den beschriebenen Entitäten nicht voll ausgeschöpft werden kann. Damit ein Reasoner die Inferenz vollumfänglich nutzen kann, mussten wir Regeln in Form der Regelsprache *SWRL* einführen.

Das folgende Kapitel basiert, sofern nicht anders vermerkt, auf [24].

Bei SWRL handelt es sich um eine auf OWL und RuleML [1] basierende Regelsprache. Sie erlaubt es Regeln in Form von OWL-Konzepten auszudrücken und bietet dadurch vielfältige Möglichkeiten der Inferenz.

9.1 Aufbau

Eine SWRL-Regel besteht aus einem Kopf und einem Körper. Hierbei folgt der Kopf immer auf den Körper. Beide bestehen aus positiven Konjunktionen von Atomen:

$$atom \wedge atom \cdots \rightarrow atom \wedge atom$$

Listing 9.1: Beispiel von positiven Konjunktionen von Atomen

Informal ausgedrückt heisst das: "Wenn alle Teile des Körpers wahr sind, dann ist auch der Kopf wahr". SWRL unterstützt keine negierten Atome oder Disjunktionen.

Ein Atom ist Ausdruck der Form:

$$p(\text{arg } 1, \text{arg } 2, \cdots argn)$$

Listing 9.2: Beispiel eines Atoms

p ist hierbei ein Prädikat, arg 1, arg 2 und $argn$ sind die Argumente bzw. Terme des Ausdrucks.

Prädikate können OWL-Klassen, -Eigenschaften oder -Datentypen sein. Argumente können OWL-Individuen, -Werte oder darauf verweisende Variablen sein.

In SRWL können Atome ausschliesslich mit UND-verknüpft werden. Eine ODER-Verknüpfung gibt es nicht. Die UND-Verknüpfung wird mittels eines Kommas , vorgenommen. Eine ODER-Verknüpfung wird duch mehrere Regeln mit unterschiedlichen Regelkörpern, aber gleichen Regelköpfen (Schlussfolgerungen) abgebildet.

9.1.1 Atomare Typen

SWRL unterscheidet zwischen den folgenden atomaren Typen:

- Klassen
- Eigenschaften von Individuen
- Wertespezifischen Eigenschaften
- Unterscheidung von Individuen

[1] www.ruleml.org

- Gleichsetzungen von Individuen
- Wertebereichen
- Vordefinierten Atomen

Klassen

Ein Klassenatom besteht aus einem Prädikat, sowie nur einem Argument. Das Prädikat ist eine OWL-Klasse, das Argument ein OWL-Individuum oder eine Variable.

```
Abenteuerreise(?x)
Ausflug(Seilpark_Balmberg)
```

Listing 9.3: Beispiele von Klassen-Atomen

Möchte man nun mittels SWRL z.B. aussagen, dass alle Abenteuerreisen auch Reisen sind, so geschieht dies mittels:

```
Abenteuerreise(?x) → Reise(?x)
```

Listing 9.4: Beispiel der SWRL-Regel, dass alle Abenteuerreisen auch Reisen sind

Eigenschaften von Individuen

Ein Atom, welches Eigenschaften von Individuen darstellt, besteht aus einem Prädikat und zwei Argumenten. Das Prädikat ist eine OWL-Eigenschaft bzw. eine Relation zwischen zwei Individuen. Die Argumente sind somit ein OWL-Individuum oder eine Variable.

```
anzahlTeilnehmer(?a, ?anzahl)
erfordert(?a, "mut")
```

Listing 9.5: Beispiele von Atomen zur Darstellung von Eigenschaften von Individuen

Wir definieren als praktisches Beispiel ein Individuum als *Teamevent*, wenn es mindestens 4 maximal aber 20 Teilnehmer zulässt. Weiter muss es über die Eigenschaft der Teambildung verfügen. Drückt man dies per SWRL-Regel aus, ergibt sich folgender Ausdruck:

```
anzahlTeilnehmer(?a, ?anzahl), greaterThanOrEqual(?anzahl, 4), lessThanOrEqual(?anzahl
    , 20), bietet(?a, "teambildung") → teamevent(?a, true)
```

Listing 9.6: Beispiel der SWRL-Regel um ein Objekt ?*a* als Teamevent zu definieren

Wertespezifische Eigenschaften

Ein Atom, welches wertespezifische Eigenschaften darstellt, besteht aus einem Prädikat sowie zwei Argumenten. Das Prädikat ist eine wertespezifische Eigenschaft. Das erste Argument ein OWL-Individuum, das zweite Argument ist ein OWL-Datenwert.

```
erfordert(?a, "mut")
durchschnittspreis(?r, ?preis)
```

Listing 9.7: Beispiele von Atomen zur Darstellung von wertespezifischen Eigenschaften von Individuen

Dass ein Objekt als aufregend (actionreich) gilt, wenn es Mut erfordert, kann mit folgender Regel beschrieben werden:

```
erfordert(?a, "mut") → action(?a, true)
```

Listing 9.8: Beispiel einer Regel, welche ausdrückt, dass ein Objekt actionreich ist, wenn es Mut erfordert

Unterscheidung von Individuen

Ein Atom, welches die Unterscheidung von Individuen bezeichnet, besteht aus einem Prädikat sowie zwei Argumenten. Das Prädikat ist das *differentFrom*-Prädikat, die Argumente sind OWL-Individuen.

```
differentFrom(?x,?y)
differentFrom(Seilpark_Balmberg,Seilpark_Pilatus)
```

Listing 9.9: Beispiele von Atomen zur Unterscheidung von zwei Individuen

Eine genauere Beschreibung der Verwendung bzw. des Sinnes dieses Atoms findet sich unter 9.2.

Gleichsetzung von Individuen

Das Atom, welches die Gleichheit von zwei Individuen kennzeichnet, besteht aus einem Prädikat sowie zwei Argumenten. Das Prädikat ist das *sameAs*-Prädikat, die Argumente sind OWL-Individuen.

```
sameAs(?x,?y)
sameAs(Seilpark, Seilpark_Balmberg)
```

Listing 9.10: Beispiele von Atomen zur Gleichsetzung von zwei Individuen

Wertebereiche

Das Atom, welches einen Wertebereich kennzeichnet, besteht aus einem Datentyp oder einer Menge von Literalen sowie einem Argument.

```
xsd:int(?x)
[3, 4, 5](?x)
```

Listing 9.11: Beispiele von Atomen zur Kennzeichnung eines Wertes

Im ersten Beispiel ist das Objekt, gekennzeichnet durch die Variable ?*x*, eine ganzzahlige Zahl.
Im zweiten Beispiel hat das Objekt einen der Werte 3, 4 oder 5.

Vordefinierte Atome

Bei den vordefinierten Atomen handelt es sich um Prädikate, welche ein oder mehrere Argumente entgegennehmen. Sofern alle Argumente dem Prädikat genügen, geben sie den Wahrheitswert "richtig" zurück. Andernfalls "falsch".
Die vordefinierten Atome haben das Präfix *swrldb*. Eine detaillierte Beschreibung aller vordefinierter Atome findet sich unter [2].

Möchte man ausdrücken, dass zum Beispiel Objekte, welche einen Durchschnittspreis zwischen 20 und 30 haben, *preiswert* (wertespezifische Eigenschaft des Attributes *preissegment*) sind, so geschieht dies mittels der SWRL-Regel:

```
durchschnittspreis(?r, ?preis), lessThanOrEqual(?preis, 30), greaterThan(?preis, 20) →
    preissegment(?r, "preiswert")
```

Listing 9.12: Beispiel einer SWRL-Regel, welche besagt, dass ein Objekt die wertespezifische Eingenschaft *preiswert* hat

[2] http://www.daml.org/2004/04/swrl/builtins

9.2 Open world assumption

Wie auch OWL geht SWRL von der so genannten *"open world assumption"* aus. Diese besagt, der Wahrheitswert einer Aussage kann unabhängig des effektiven Wahrheitwertes wahr sein.

Möchte man zum Beispiel ausdrücken, zwei OWL-Individuen sind Kollaborateure (sie arbeiten zusammen an einer Publikation) gilt folgende Regel:

$$Publication(?p), hasAuthor(?p, ?y), hasAuthor(?p, ?z) \rightarrow collaboratesWith(?y, ?z)$$

Listing 9.13: Beispiel einer SWRL-Regel zum Ausdrücken von Kollaboration

Aufgrund der "open world assumption" von OWL ist es nicht möglich auszusagen, dass zwei OWL-Individuen automatisch unterschiedlich sind, wenn sie verschiedene Namen haben. Mittels den *sameAs*- und *differentFrom*-Relationen bietet SWRL die Lösung des Problemes:

$$Publication(?p), hasAuthor(?p, ?y), hasAuthor(?p, ?z), differentFrom(?y, ?z) \rightarrow collaboratesWith(?y, ?z)$$

Listing 9.14: Beispiel einer SWRL-Regel zum Ausdrücken von Kollaboration mit expliziter Unterscheidung zwischen den Kollaborateuren

Analog gilt dies für die Gleichsetzung von zwei Individuen. Diese müssen explizit mit der *sameAs*-Relation gleichgesetzt werden.

Aufzählungen können beispielsweise nicht ohne Umstände ausgedrückt werden: Es ist nicht möglich zu sagen, ob eine Publikation nur von einem einzigen Autor stammt. Es kann durchaus sein, dass nur ein Autor der Publikation bekannt ist, diese aber noch über weitere Autoren verfügt.

Dies bezeichnet den Kern der "open world assumption". Der effektive Wahrheitswert ist beispielsweise, die Publikation $?p$ hat mehrere Autoren ($?x$, $?y$ und $?z$). Die Aussage, dass die Publikation $?p$ einen bestimmten Autor $?z$ hat, ist dennoch wahr, auch wenn die übrigen Autoren noch nicht bekannt sind.

In diesem Kapitel wurden *Regeln* eingeführt, die es ermöglichen die im Kapitel 3 (Graphrepräsentationen) gefundenen Kriterien *familienfreundlich*, *regional* und *actionreich* als Schlussfolgerungen zu definieren. Vorgängig müssen diese jedoch als Attribute bzw. DataProperties (sepzifischer Datenwert eines Objektes) definiert werden.

Wir haben die Definition gewählt wie folgt:

Ein Objekt muss über das Attribut *alter* verfügen und der Wert dieses Attributes muss grösser oder gleich der Zahl zwei sein. Ist dies gegeben, erhält ein Objekt automatisch das Attribut *familienfreundlich* mit dem Wahrheitswert (Boolean) *wahr* bzw. *true*.

```
alter(?a, ?alter), swrldb:greaterThanOrEqual(?alter, 2) ⇒ familienfreundlich(?a, true)
```

Listing 9.15: Beispiel der SWRL-Regel für die Eigenschaft *familienfreundlich*

Ein Objekt gilt dann als *regional*, wenn es über einen Standort verfügt und die Region dieses Standortes 50 oder weniger Kilometer vom Ausgangspunkt (welcher dynamisch bestimmt sein kann, also abhängig vom Abfragesteller) entfernt ist.

```
hatStandort(?ausflug, ?ort), hatRegion(?land, ?region), hatOrt(?region, ?ort),
    distanzZuAusgangspunkt(?region, ?distanz), swrldb:lessThanOrEqual(?distanz, 50) ⇒
    regional(?ausflug, true)
```

Listing 9.16: Beispiel der SWRL-Regel für die Eigenschaft *regional*

Eine Regel kann in SWRL keine ODER-Verknüpfung enthalten, wie in Abschnitt 9.1 (Aufbau) beschrieben. Die ODER-Verknüpfungen bildet man ab, durch mehrfache Definition derselben Regel. Der Regelkopf (das heisst die Schlussfolgerung) muss immer gleich sein, die Bedingungen (das heisst die Regelköper) dürfen verschieden sein.

Wir wenden an: Ein Objekt erhält das Attribut *actionreich*, wenn es Mut oder Geschicklichkeit erfordert oder wenn es Nervenkitzel verursacht. Die Eigenschaft *nervenkitzel* ist ihrerseits eine Folgerung aus der Erfordernis von Mut.

Es findet also eine transitive Ableitung statt: $A \sim B \wedge B \sim C \Rightarrow A \sim C$ bzw. $erfordertMut \Rightarrow birgtNervenkitzel \wedge birgtNervenkitzel \Rightarrow istActionreich$ somit gilt $erfordertMut \Rightarrow istActionreich$.

```
erfordert(?a, "mut") ⇒ actionreich(?a, true)
nervenkitzel(?a, true) ⇒ actionreich(?a, true)
erfordert(?a, "geschicklichkeit") ⇒ actionreich(?a, true)
```

Listing 9.17: Beispiel der SWRL-Regel für die Eigenschaft *actionreich*

Es sind jetzt alle Komponenten vorhanden, um Ausflüge abbilden zu können, die den in Kapitel 3 (Graphrepräsentationen) genannten Kriterien entsprechen.

Um nach Ausflügen mit den gewählten Kriterien suchen zu können, benötigt man jedoch eine bestimmte Abfragesprache. Diese wird im kommenden Kapitel aufgezeigt.

---

10 SPARQL

Im letzten Kapitel wurde aufgezeigt, wie Inferenz in einer Ontologie mittels der Regelsprache SWRL vollumfänglich genutzt werden kann.

Wie können Informationen aus der Ontologie und den darauf basierenden Inferenzen abgefragt werden?
Mit SPARQL — einer Abfragesrprache — ist es möglich.

Im folgenden Kapitel, welches, sofern nicht anders vermerkt, auf [25] basiert, stellen wir diese Abfragesprache vor.

Bei SPARQL handelt es sich sich um eine Abfragesprache für RDF. Sie erlaubt es, Abfragen in mehreren Datenquellen vorzunehmen. Dabei werden Anfragen über Graphen vollzogen, auch entlang derer Konjunktionen und Disjunktionen. SPARQL unterstützt weiter Aggregation, Unterabfragen, Negation sowie die Nutzung von Ausdrücken als Werte.

Resultate sind entweder eine Menge von Ergebnissen oder RDF-Graphen.

(vgl. [25, Abstract])

Kennt man die Abfragesprache SQL, so erscheint SPARQL auf den ersten Blick recht ähnlich — der Name lässt dies bereits vermuten. Dies ist nur teilweise der Fall. Klauseln wie *SELECT* und *WHERE* sind ähnlich. In SPARQL gibt es jedoch keine *FROM*-Klausel.
Der Hauptunterschied zwischen SPARQL und SQL ist jedoch die Verwendung von Variablen. Diese werden in SPARQL in der Regel bei jeder Abfrage verwendet. Abfragen ohne Variablen kommen in der Praxis (gemäss unserer Erfahrung) eher selten vor.
SQL enthält in der Regel sehr wenig Variablen.

Eine SPARQL-Abfrage besteht aus folgenden Komponenten:

- Einem oder mehreren Namespaces
- Variablen
- Einem (Teil-) Graphen
- Einer Menge von Gruppierungen und Aggregationen
- Einer Menge von Modifikatoren
- Abfragearten
- Ausdrücken und Wertevergleichen

(vgl. [25, 18.1.10 SPARQL Query])

Für die Praxis besteht eine SPARQL-Abfrage in der Regel aus *Namespaces*, einer *SELECT*-Klausel, einer *WHERE*-Klausel sowie *Modifikatoren*.

10.1 Beispiel einer SPARQL Abfrage

Gegeben sei die folgende Datenbasis:

```
<http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> "SPARQL
    Tutorial".
```

Listing 10.1: Einfache Datenbasis direkt in SPARQL[1]

Die Datenbasis beinhaltet das Objekt *book1* mit dem Attribut *title*, welches den Wert *"SPARQL Tutorial"* enthält.

Möchte ich nun den Titel des Buches abfragen, so kann ich folgende Abfrage verwenden:

```
SELECT
    ?title
WHERE {
    <http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> ?title .
}
```

Listing 10.2: Beispiel einer einfachen SPARQL-Abfrage[2]

Sie ergibt folgendes Resultat:

title
"SPARQL Tutorial"

Tabelle 10.1: Resultat einer einfachen SPARQL-Abfrage[3]

10.2 Namespaces

SPARQL ist eine Abfragesprach für RDF, das seinerseits auf XML basiert. Dadurch sind XML-Namespaces auch in SPARQL verfügbar. Bei Namespaces handelt es sich um Referenzen auf andere XML-basierte Dokumente, welche modular genutzt werden können.
Ein Namespace besteht aus einem Kürzel (welches auch leer sein kann), sowie aus der kompletten Adresse der Ressource (vgl. [26, 2.1 Introduction]).

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" % chktex 1
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
```

Listing 10.3: Beispiel von Namespaces in RDF[4]

10.3 Variablen

Bei Abfragen unterstützt SPARQL die Verwendung von Variablen. Diese können dabei an einer beliebigen Stelle, wie bei (Teil-) Graphen, Gruppierungen, Aggregationen und Modifikatoren eingesetzt werden.
Die Variablen dienen als Platzhalter für abzufragende Objekte, Subjekte und Prädikate. Variablen beginnen entweder mit einem Dollarzeichen *$* oder einem Fragezeichen *?*, welche selbst nicht Teil der Variable sind.

[1] [25, 2.1 Writing a Simple Query]
[2] [25, 2.1 Writing a Simple Query]
[3] [25, 2.1 Writing a Simple Query]
[4] [26, 2.6 Completing the Document: Document Element and XML Declaration]

```
SELECT DISTINCT
    *
WHERE {
  ?object ?predicate ?subject
}
LIMIT
    10
```

Listing 10.4: Beispiel einer SPARQL Abfrage mit den Variablen *?object*, *?predicate* und *?subject*.

Wird obige Anfrage auf einen SPARQL-Endpunkt, wie z.B. *http://dbpedia.org/*, angewendet, so ergibt dies folgendes Resultat:

object	**predicate**	**subject**
http://dbpedia.org/ontology/acceleration	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/torqueOutput	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/birthDate	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/birthYear	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/deathDate	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/deathYear	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/diameter	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/displacement	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/height	... rdf-syntax-ns#type	... owl#FunctionalProperty
http://dbpedia.org/ontology/latestReleaseDate	... rdf-syntax-ns#type	... owl#FunctionalProperty

Tabelle 10.2: Resultat einer Abfrage des DBPedia-Endpunktes limitiert auf 10 Ergebnisse.

10.4 (Teil-) Graphen

SPARQL basiert auf dem Matching eines (Teil-) Graphen in einem gegebenen Graphen. Dabei ist ein (Teil-) Graph eine Menge von Tripeln.

Man unterscheidet zwischen den folgenden Arten von Teilgraphen:

- *Grundlegende Teilgraphen*

  Hierbei muss eine Menge von Tripeln derjenigen im gegebenen Graphen entsprechen

- *Gruppierende Teilgraphen*

  Hierbei müssen *alle* Elemente einer Menge von Tripeln von grundlegenden Teilgraphen wiederum derjenigen im gegebenen Graphen entsprechen

- *Optionale Teilgraphen*

  Zusätzliche Teilgraphen können die Lösungsmenge erweitern

- *Alternative Teilgraphen*

  Versuch, ob zwei oder mehrere mögliche Teilgraphen denjenigen im gegebenen Graphen entsprechen

- *Teilgraphen in namensbasierten Graphen*

  Versuch, ob Teilgraphen denjenigen im gegebenen, namensbasierten Graphen entsprechen

Ein grundlegender Teilgraph ist somit eine Menge von Tripeln, wobei jeder Tripel gemäss RDF-Spezifikation immer aus Subjekt, Prädikat und Objekt bestehen muss. [19, 3.1 Triples]

```
        ?subject ?predicate ?object .
```

Listing 10.5: Beispiel eines grundlegenden Teilgraphen in SPARQL

Ein Teilgraph wird in SPARQL immer zur Gruppierung einer Abfrage verwendet. Dies entspricht in der SQL-Sprache der *WHERE*-Klausel.
Ein Teilgraph wird immer mit dem Punkt-Operator beendet.

Ein gruppierender Teilgraph wird in der SPARQL-Abfrag immer mit geschweiften Klammern *{}* umschlossen. Es ist möglich die Resultate einer Gruppierung zu filtern. Dazu wird *FILTER* als Schlüsselwort verwendet. Dieses bezieht sich immer auf die Gruppierung, in welcher es steht.

```
        {
            ?x foaf:name ?name.
            ?x foaf:mbox ?mbox.
            FILTER regex(?name, "Smith")
        }
```

Listing 10.6: Beispiel eines gruppierenden Teilgraphen mit dem *FILTER*-Schlüsselwort[5]

Im obigem Beispiel werden alle Objekte und die Werte ihrer Attribute (*name* und *mbox*, welche im Namespace *foaf* bezeichnet werden) abgefragt.

Dabei kann eine komplette leere Menge — wenn keine Objekte vorhanden sind — oder aber eine leere Teilmenge — wenn ein Objekt z.B. nicht über die beiden Attribute *name* und *mbox* verfügt — zurückgegeben werden.

Es bleiben nur Objekte zurück, welche über das Attribut *foaf:name* verfügen und dessen Inhalt *"Smith"* ist.

10.5 Gruppierungen und Aggregationen

Aggregationen bezeichnen eine bestimmte Art der Verbindung zwischen Objekten. [27] Aggregationen wenden Ausdrücke bzw. Funktionen auf eine Lösungsmenge an. Dabei enthält eine Lösung normalerweise eine einzelne Gruppe mit allen Lösungen.

Gruppierungen werden mittels *GROUP BY* angegeben.

SPARQL unterstützt aktuell die Aggregationen *COUNT*, *SUM*, *MIN*, *MAX*, *AVG*, *GROUP_CONCAT* und *SAMPLE*.

Die Aggregation wird dann verwendet, wenn ein Resultat über eine Gruppe von Lösungen berechnet werden soll (Beispiel: Maximaler Wert einer bestimmten Variablen).

(vgl. [25, 11 Aggregates])

```
        PREFIX books: <http://books.example/>
        SELECT (
            SUM(?lprice) AS ?totalPrice
        )
        WHERE {
          ?org books:affiliates ?auth .
```

[5] [25, 5.2.2 Scope of Filters]

```
    ?auth books:writesBook ?book .
    ?book books:price ?lprice .
}
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

Listing 10.7: Beispiel einer Abfrage mit Gruppierung und Aggregation[6]

10.6 Modifikatoren

Abfragen mittels SPARQL generieren eine ungeordnete Menge von Lösungen. Jede dieser Lösungen stellt eine Funktion von Variablen zu RDF-Termen dar. So gewonnene Lösungen werden als Sequenz von Lösungen behandelt, zunächst ohne spezifische Ordnung. [25, 15 Solution Sequences and Modifiers]

Um eine Sequenz von Lösungen zu ordnen, werden Modifikatoren verwendet.
Dabei unterscheidet man zwischen folgenden Modifikatoren:

- *Order*

  Sortiert die Lösungen auf- oder absteigend nach einer bestimmten Variablen

- *Projection*

  Erlaubt die Auswahl von spezifischen Variablen mittels der *Select*-Klausel

- *Distinct*

  Stellt sicher, dass die Lösungen in der Lösungsmenge einmalig sind

- *Reduced*

  Verhindert die Elimination von Duplikaten in der Lösungsmenge

- *Offset*

  Definiert, ab welchem Element der Lösungsmenge die gefundenen Lösungen ausgegeben werden

- *Limit*

  Definiert die maximale Anzahl an gefundenen Lösungen, die ausgegeben werden soll

Die *Modifikatoren* werden in der Reihenfolge der obigen Liste angewendet. [25, vgl. 15 Solution Sequences and Modifiers]

10.7 Abfragearten

SPARQL unterscheidet zwischen vier Abfragearten:

- *SELECT*

  Liefert alle Variablen oder eine Teilmenge derselben eines Graphen

- *CONSTRUCT*

  Liefert einen RDF-Graphen durch Substitution der Variablen in einer Menge von gegebenen Tripeln

- *ASK*

  Liefert einen Wahrheitswert (Boolean), welcher angibt, ob die Menge von angefragten Tripeln derjenigen in dem gegebenen Graphen entspricht

[6] [25, 11.1 Aggregate Example]

- *DESCRIBE*

  Liefert einen RDF-Graphen, welcher die gefundenen Ressourcen beschreibt

10.7.1 SELECT

Die *SELECT*-Abfrageart liefert direkt Variablen und deren Belegung, in Form von Projektionen. Dabei entstehen neue Variablen-Belegungen. Die spezifischen, gewünschten Variablen werden in Form einer Liste, durch Leerzeichen getrennt, angegeben.

```
SELECT ?a ?b ?c
WHERE {
    ...
```

Listing 10.8: Beispiel der *SELECT*-Abfrageart in SPARQL

Die *SELECT**-Syntax ist eine Kurzschreibweise und bedeutet die Verwendung bzw. die Auflistung aller in der Abfrage verwendeten Variablen. Davon ausgenommen sind Variablen innerhalb von *FILTER*-Anweisungen sowie Variablen, welche rechts des Minus-Operators stehen. Die Syntax darf nur verwendet werden, wenn die Abfrage keine *GROUP BY*-Aggregation verwendet.

```
SELECT *
WHERE {
 ...
```

Listing 10.9: Beispiel der *SELECT ** Kurzschreibweise in SPARQL

Durch die *SELECT*-Abfrageart lassen sich neuen Variablen durch Ausdrücke einführen. Diese werden mittels (*SELECT expr AS var*) eingeführt.

```
SELECT ?title (?p * (1 - ?discount) AS ?price)
WHERE {
 ...
```

Listing 10.10: Beispiel eines Ausdrucks der *SELECT*-Abfrageart

10.7.2 CONSTRUCT

Die *CONSTRUCT*-Abfrageart gibt als Antwort einen einzigen, durch eine Vorlage definierten RDF-Graphen zurück: Jede Variable der Abfrage wird durch die passende Lösung aus der Lösungssequenz ersetzt. Die Lösungsmenge wird schliesslich durch Vereinigung der Tripel auf einen einzigen RDF-Graphen reduziert.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT   { <http://example.org/person#Alice> vcard:FN ?name }
```

```
WHERE       { ?x foaf:name ?name }
```

Listing 10.11: Beispiel einer SPARQL-Abfrage unter Nutzung der *CONSTRUCT*-Abfrageart[7]

Die obige Abfrage erzeugt folgende Ausgabe:

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

<http://example.org/person#Alice> vcard:FN "Alice" .
```

Listing 10.12: Beispiel der Ausgabe einer SPARQL-Abfrage unter Nutzung der *CONSTRUCT*-Abfrageart[8]

Entsteht bei diesem Vorgang ein ungültiges Tripel, wird dieses nicht im Ausgabegraphen angezeigt. Enthält die Vorlage eines Graphen Tripel ohne Variablen, erscheinen diese dennoch im Ausgabegraphen.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?x  vcard:N _:v .
            _:v vcard:givenName ?gname .
            _:v vcard:familyName ?fname }
WHERE
 {
    { ?x foaf:firstname ?gname } UNION { ?x foaf:givenname  ?gname } .
    { ?x foaf:surname   ?fname } UNION { ?x foaf:family_name ?fname } .
 }
```

Listing 10.13: Beispiel einer SPARQL Abfrage mit Nutzung der *CONSTRUCT*-Abfrageart unter Verwendung leerer Knoten[9]

Die obige Abfrage erzeugt folgende Ausgabe:

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

_:v1 vcard:N          _:x .
_:x vcard:givenName   "Alice" .
_:x vcard:familyName  "Hacker" .

_:v2 vcard:N          _:z .
_:z vcard:givenName   "Bob" .
_:z vcard:familyName  "Hacker" .
```

Listing 10.14: Beispiel der Ausgabe einer SPARQL Abfrage mit Nutzung der *CONSTRUCT*-Abfrageart unter Verwendung leerer Knoten[10]

[7] [25, 16.2 CONSTRUCT]
[8] [25, 16.2 CONSTRUCT]
[9] [25, 16.2.1 Templates with Blank Nodes]
[10] [25, 16.2.1 Templates with Blank Nodes]

10.7.3 ASK

Die *ASK*-Abfrageart beantwortet in der Ausgabe, ob eine Lösung für die gegebene Abfrage existiert.

Gegeben sei die folgende Datenbasis:

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

Listing 10.15: Einfache Datenbasis direkt in SPARQL[11]

Wird nun folgende Abfrage getätigt:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  {
    ?x foaf:name  "Alice"
}
```

Listing 10.16: Beispiel einer SPARQL-Abfrage mit Nutzung der *ASK*-Abfrageart[12]

So antwort das System: $true$, da die Datenbasis ein Individuum mit dem Attribut $foaf : name$ enthält, dessen Wert *"Alice"* ist.

10.7.4 DESCRIBE

Bei der *DESCRIBE*-Abfrageart antwortet das System mit einem RDF-Graphen, der Informationen in Form von RDF-Daten über Ressourcen enthält. Als Ressourcen kommen entweder *IRI*s oder Variablen in Betracht.

```
DESCRIBE
    ?a ?b ?c
WHERE {
...
```

Listing 10.17: Beispiel der *DESCRIBE*-Abfrageart in SPARQL

Die $DESCRIBE*$-Syntax ist die Kurzschreibweise und bedeutet die Verwendung bzw. die Auflistung aller in der Abfrage verwendeten Variablen.

```
DESCRIBE *
WHERE {
...
```

Listing 10.18: Beispiel der *DESCRIBE ** Kurzschreibweise in SPARQL

[11] [25, 16.3 ASK]
[12] [25, 16.3 ASK]

10.8 Ausdrücke und Wertevergleiche

In SPARQL kann jede Abfrage in SPARQL mit einem Filter versehen werden. Dabei wird versucht ein bestimmtes Muster (durch Beschränkungen) auf Graphen anzuwenden um so die Graphen einzuschränken (filtern).

Jedes RDF-Literal kann durch einen bestimmten Datentyp definiert sein, wie zum Beispiel Wahrheitswerte (Boolean) oder Datum und Uhrzeit (DateTime). Der Filter in SPARQL erlaubt dabei den Wertevergleich auf typisierte RDF-Literale. Die dabei verwendeten Operanden und Datentypen finden sich unter [13] und [14].

Eine SPARQL-Abfrage unter Verwendung eines Filters kann aussehen wie folgt:

```
...
SELECT
    ?book, ?amount
WHERE {
  ?book :isInLibrary :BerneUniversityLibrary
  ?book :hasAmount ?amount
  FILTER (?amount > xsd:integer(10))
}
```

Listing 10.19: Beispiel einer einfachen SPARQL-Abfrage unter Verwendung eines Filters

In diesem Beispiel wird eine Datenbank nach mehrfachen Vorkommen (mindestens 10 Mal) eines Buchexemplares in der Bibliothek der Universität Bern abgefragt. Die Anzahl der Exemplare eines Buches wird durch die Variable ?*amount* zurückgegeben.

Ist man in obigem Beispiel nicht sicher, welchem Datentyp die Variable ?*amount* entspricht, kann man die Funktionalität zur Umwandlung (Casting, Typenumwandlung) von Variablen nützen:

```
...
SELECT
    ?book, ?amount
WHERE {
  ?book :isInLibrary :BerneUniversityLibrary
  ?book :hasAmount ?amount
  FILTER (xsd:integer(?amount) > xsd:integer(10))
}
```

Listing 10.20: Beispiel einer einfachen SPARQL-Abfrage unter Verwendung eines Filters mit Typenumwandlung

Die genaue Auswertung eines Filters lassen sich unter [15], die interne Verwendung und Auswertung von Operatoren unter [16] nachlesen.

10.8.1 Funktionen

SPARQL bietet eine Vielzahl an Operatoren und Funktionen. Diese einzeln aufführen und beschreiben zu wollen, würde den Rahmen dieser Arbeit sprengen. Daher werden nur die Operatoren und Funktionen aufgeführt, welche in der vorliegenden Arbeit hauptsächlich angewendet wurden. Eine genauere Beschreibung findet sich unter [17].

[13] http://www.w3.org/TR/xpath-functions/
[14] http://www.w3.org/TR/sparql11-query/#operandDataTypes
[15] http://www.w3.org/TR/sparql11-query/#evaluation
[16] http://www.w3.org/TR/sparql11-query/#OperatorMapping
[17] http://www.w3.org/TR/sparql11-query/#SparqlOps

BOUND

Mittels *BOUND* lässt sich prüfen, ob ein Objekt eine Wert-Bindung hat. Man kann es als eine Art der Überprüfung auf gewisse Eigenschaften bezeichnen.

Möchte man zum Beispiel wissen, welche Personen über eine Relation zu einem Datum (z.B. ihr Erfassungsdatum) haben, so bestimmt man dies mittels:

```
@prefix foaf:        <http://xmlns.com/foaf/0.1/> .
@prefix dc:          <http://purl.org/dc/elements/1.1/> .
@prefix xsd:          <http://www.w3.org/2001/XMLSchema#> .

_:a   foaf:givenName   "Alice".

_:b   foaf:givenName   "Bob" .
_:b   dc:date          "2005-04-04T04:04:04Z"^^xsd:dateTime .
```

Listing 10.21: Beispiel einer simplen Ontologie zur Nutzung der *BOUND*-Funktion in einer Abfrage[18]

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
SELECT
    ?givenName
WHERE {
    ?x foaf:givenName  ?givenName .
    OPTIONAL { ?x dc:date ?date } .
    FILTER ( bound(?date) )
}
```

Listing 10.22: Beispiel einer SPARQL-Abfrage zur Nutzung der *BOUND*-Funktion[19]

Die obige Abfrage ergibt folgendes Resultat:

givenName
"Bob"

Tabelle 10.3: Resultat einer SPARQL-Abfrage mittels *BOUND*-Filter[20]

Es wird nur die Person *"Bob"* gefunden, da nur bei dieser Person eine Datum-Relation definiert ist.

IF

Die *IF*-Funktion gibt Argumente aufgrund einer Bedingung, welche selbst auch ein Argument ist, zurück. Die Funktion verwendet drei Argumente als Parameter. Die Argumente sind:

- Argument 1: Bedingung
- Argument 2: Rückgabewert wenn die Bedingung (Argument 1) zutrifft
- Argument 3: Rückgabewert wenn die Bedingung (Argument 1) nicht zutrifft

```
IF(?a = 2,?b,?c)
```

Listing 10.23: Beispiel der *IF*-Funktion in SPARQL

[18] [25, 17.4.1.1 BOUND]
[19] [25, 17.4.1.1 BOUND]
[20] [25, 17.4.1.1 BOUND]

In prozeduraler Programmierung ausgedrückt:

```
if (a == 2) {
    return b;
} else {
    return c;
}
```

Listing 10.24: Beispiel der *IF*-Funktion in SPARQL anhand prozeduraler Programmierung ausgedrückt

IN

Die *IN*-Funktion prüft, ob ein Element in einer Liste vorkommt. Das zu prüfende Element wird der *IN*-Funktion vorangestellt, danach folgt die zu prüfende Liste. Kommt das Element in der Liste vor, gibt die Funktion den Wahrheitswert *wahr* bzw. *true* zurück, andernfalls *falsch* bzw. *false*.

```
2 IN (1, 2, 3) % Wahr
2 IN ()        % Falsch
```

Listing 10.25: Beispiel der *IN*-Funktion in SPARQL

In den obigen Beispielen ist der erste Ausdruck wahr, da 2 in der gegebenen Liste von (1, 2, 3) vorkommt. Der zweite Ausdruck ist falsch, da 2 nicht in der leeren Liste () vorkommt.

Analog zur *IN*-Funktion existiert die *NOT IN*-Funktion. Dabei handelt es sich um die Negation der *IN*-Funktion.

Wie beschrieben sind jetzt alle Komponenten vorhanden, um Ausflüge abbilden zu können, die den in Kapitel 3 (Graphrepräsentationen) genannten Kriterien entsprechen.

In diesem Kapitel haben wir die bisher fehlende Abfragesprache abgehandelt.

Um auf das beschriebene Beispiel der Familie, welche einen Ausflug plant, zurückzukommen: Mit der nachfolgenden Abfrage gelingt es, alle Ausflüge mit den Kriterien *familienfreundlich*, *regional* und *actionreich* zu finden.

```
SELECT
    *
WHERE {
    ?object :familiengerecht true;
            :regional true;
            :action true.
}
```

Listing 10.26: Beispiel einer Abfrage um alle familiengerechten, actionreichen und regionalen Ausflüge der Ontologie auszugeben

---

[20] [25, 17.4.1.9 IN]

11 Schlusswort

In der vorliegenden Arbeit haben wir gezeigt, wie ein Knowledge Engineer bei der Modellierung und Formalisierung einer Problemdomäne vorgehen kann. Am Beispiel Reiseplanung haben wir schrittweise ein Expertensystem aufgebaut.

Dieses Expertensystem beinhaltet eine semantische Datenbank, die ihrerseits auf der Basis einer Ontologie beruht.

Wichtige Erkenntniss: Es eignen sich nur Themengebiete, die aufgrund der Ontologie Inferenz und damit Schlussfolgerungen zulassen.

Andernfalls ist der Mehrwert der Ontolgie gegenüber der herkömmlichen Wissensabbildung nicht gegeben.

Während herkömmliche (relationale) Datenbaken nur Beziehungen aufzeigen, liegt der Vorteil der semantischen Datenbank in ihrer Flexibilität und Semantik (Möglichkeit den Relationen eine Bedeutung zu geben).

Nutzt eine Applikation eine semantische Datenbank als Datenmodell, erfordern Anpassungen (Modellierungen) des Datenmodelles keine Programmänderungen — bei geschickter Programmierung.
Modellierungen sind z.B. das Hinzufügen, Bearbeiten oder Löschen von Entitäten (Klassen, Individuen, Relationen oder Eigenschaften).

Im Gegensatz hierzu benötigen Änderungen in relationalen Datenbanken meistens sehr aufwändige Programmänderungen.

Literaturverzeichnis

[1] Informatik Spektrum. Eine kurze geschiche der ontologie, September 2014.

[2] P. Norvig S. J. Russell. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[3] J. Cleve U. Lämmel. *Künstliche Intelligenz*. Carl Hanser Verlag München, 4rd edition, 2012.

[4] Introducing graph data, October 2014. URL `http://www.linkeddatatools.com/introducing-rdf`.

[5] Wikipedia. Wissensrepräsentation, October 2014. URL `http://de.wikipedia.org/wiki/Wissensrepr%C3%A4sentation/`.

[6] Informatik Spektrum. Was bedeutet eigentlich ontologie?, September 2014.

[7] World Wide Web Consortium. Inference, October 2014. URL `http://www.w3.org/standards/semanticweb/inference`.

[8] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0-521-78176-0.

[9] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner, 2005.

[10] Patel-Schneider, Haes, and Horrocks. Owl web ontology language abstract syntax and semantics, w3c recommendation, October 2014. URL `www.w3.org/TR/owl-semantics/`.

[11] F. Bar and B. Foo. Der baum und foo.

[12] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner, 2007.

[13] FrancescoM. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Al-log: Integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998. ISSN 0925-9902. doi: 10.1023/A:1008687430626. URL `http://dx.doi.org/10.1023/A%3A1008687430626`.

[14] John D. Ramsdell. *Datalog User Manual*. MITRE Corporation, http://datalog.sourceforge.net/datalog.html, 2.3 edition, 12 2014.

[15] Wikipedia Foundation. Description logic, December 2014. URL `http://en.wikipedia.org/wiki/Description_logic`.

[16] Haoyi Xiong and Ying Jiang. Constraint satisfaction problem solving based on owl reasoning.

[17] Derek Sleeman and Stuart Chalmers. Assisting domain experts to formulate and solve constraint satisfaction problems. In Steffen Staab and Vojtěch Svátek, editors, *Managing Knowledge in a World of Networks*, volume 4248 of *Lecture Notes in Computer Science*, pages 27–34. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-46363-4. doi: 10.1007/11891451_5. URL `http://dx.doi.org/10.1007/11891451_5`.

[18] Madalina Croitoru and Ernesto Compatangelo. Ontology constraint satisfaction problem using conceptual graphs. In Max Bramer, Frans Coenen, and Andrew Tuson, editors, *Research and Development in Intelligent Systems XXIII*, pages 231–244. Springer London, 2007. ISBN 978-1-84628-662-9. doi: 10.1007/978-1-84628-663-6_17. URL `http://dx.doi.org/10.1007/978-1-84628-663-6_17`.

[19] Raimond Schreiber. Rdf 1.1 primer, October 2014. URL `http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/`.

[20] Gonzales. Cambridge semantics: Rdf 101, October 2014. URL `http://www.cambridgesemantics.com/semantic-university/rdf-101`.

[21] Hitzler, Krötzsch, Parsia, Patel-Schneider, and Rudolph. Owl 2 web ontology language, October 2014. URL www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[22] Tester. Cambridge semantics: Owl 101, October 2014. URL http://www.cambridgesemantics.com/semantic-university/owl-101.

[23] Mira Günzburger Sven Osterwalder. Requirements of elephant search – a semantic search engine for children, 2014.

[24] I. Horrocks, Peter F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml, November 2014. URL http://www.w3.org/Submission/SWRL/.

[25] Seaborne Harris. Sparql 1.1 query language, October 2014. URL http://www.w3.org/TR/sparql11-query/.

[26] Beckett. Rdf/xml syntax specification (revised), October 2014. URL http://www.w3.org/TR/REC-rdf-syntax/.

[27] Wikipedia Foundation. Aggregation, December 2014. URL http://de.wikipedia.org/wiki/Aggregation.

Abbildungsverzeichnis