UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik

Entwicklung eines DL-Learner Plugins für Protégé

Bachelorarbeit

Leipzig, 13. August 2008
Professor: Prof. Dr. Ing. habil. Klaus-Peter Fähnrich
Betreuer: Dipl. Inf. Jens Lehmann

vorgelegt von

Christian Kötteritzsch
geb. am: 19.03.1986

Studiengang Informatik

Inhaltsverzeichnis

1 Einleitung

Durch den ständigen Wachstum des Internets bietet es heute eine für den Menschen schon unüberschaubare Menge an Informationen. Dabei leisten Suchmaschinen wie Google[1] oder Yahoo[2] hier schon gute Arbeit, können aber keine semantischen Zusammenhänge erkennen, da diese auf menschliche Darstellung ausgelegt und somit nicht oder nur schwierig für Maschinen zu interpretieren sind. Die Idee des Semantic Web ist eine Erweiterung der bestehenden Daten um maschinenlesbare Informationen, um eine automatische Interpretation und Weiterverarbeitung der Informationen zu gewährleisten. Zu diesem Zweck braucht man Wissensbasen, die Ontologien. Dabei sind diese hierarchisch angeordnet und enthalten logische Beziehungen der einzelnen Objekte. Damit soll es möglich werden, dass Maschinen diese Informationen interpretieren können, um dem Benutzer bessere Ergebnisse bei der Suche von Daten zu ermöglichen[HKRS08].

Für die Modellierung von Ontologien gibt es verschiedene Werkzeuge, wie z.B. SWOOP, Protégé oder Ontoverse. Ontoverse ist ein webbasierter Ontologie-Editor, welcher das interdisziplinäre Erstellen von Ontologien ermöglicht, aber keine Möglichkeit zur unterstützenden Ontologie-Erstellung bietet. SWOOP und Protégé sind beide in Java geschriebene Editoren, in die ein Reasoner integriert wurde, um die Konsistenz der erstellten Klassen zu überprüfen. Dadurch ist es möglich Fehler bei der Erstellung zu finden, doch eine Hilfe bei der Erstellung von komplexen Klassenstrukturen bieten diese Editoren nicht. Ein weiteres Problem stellt die heute oft angewandte getrennte Entwicklung von Schemata und Instanzdaten dar, denn dies kann schnell zu Inkonsistenzen in den Daten führen. Dabei gibt es einerseits viele Ontologien, die keine Instanzdaten besitzen, wie dies auf der Seite SchemaWeb[3], z.B. an der Ontologie *moviedatabase* dargestellt wird (Abbildung 1). Andererseits gibt es große Wissensbasen mit viele Instanzen, wie z.B. DBpedia[4] oder andere Wissensbasen aus LOD-cloud[5], die kein ausdrucksstarkes Schema besitzen.

Da die Modellierung komplexer Klassenstrukturen neben der eigentlichen Modellierung des Schemas eine der schwierigsten Aufgaben bei der Ontologie-Erstellung ist, wird in dieser Arbeit ein Plugin für Protégé beschrieben, mit dem es möglich ist ein Basisschema mit Klassenhierarchie und Eigenschaften zu entwickeln, um anschließend komplexe Strukturen zu ergänzen, damit das Reasoning ermöglicht oder verbessert werden kann. Dabei ist die Wahl auf diesen Ontologie-Editor gefallen, da er einer der am häufigsten verwendeten Editoren ist und dadurch einer großen Anzahl an Nutzern zur Verfügung steht.

[1] http://www.google.de
[2] http://www.yahoo.de
[3] http://www.schemaweb.info
[4] http://dbpedia.org/
[5] http://linkeddata.org/

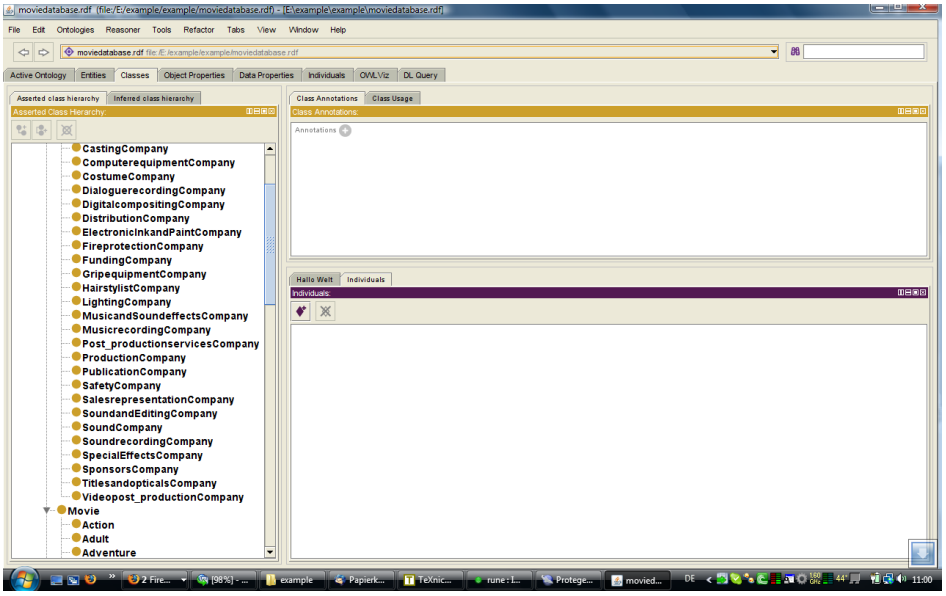

Abbildung 1: Diese Abbildung zeigt eine Ontologie in Protégé , die links das Schema anzeigt und rechts unten in violett die Instanzdaten. Hier ist zu sehen, dass keine Instanzen vorhanden sind.

Im 2. Kapitel dieser Arbeit wird die Grundidee des Semantic Web erklärt und auf die Ontologiesprachen RDF und OWL eingegangen. Im 3. Kapitel gibt es einen Überblick über Protégé und wie man dafür Plugins entwickelt. Das 4. Kapitel widmet sich dem DL-Learner, welcher die Grundlage dieses Plugins bildet. Anschließend wird in Kapitel 5 das Plugin selbst vorgestellt und auf Besonderheiten bei der Implementierung eingegangen. Im 6. Kapitel findet eine Evaluation der getätigten Arbeit statt. Danach wird in Kapitel 7 auf verwandte Arbeiten im Bereich der Ontologie-Erstellung eingegangen. Im letzten Kapitel wird die Arbeit noch einmal zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

2 Die Idee des Semantic Web

Das Internet besteht aus einer für den Menschen unüberschaubaren Menge an Informationen, die in verschiedenen Formaten gespeichert sind. Um zu einem bestimmten Thema alle Daten herrauszufiltern, sind heutige Suchmaschinen meist nicht in der Lage, da die Repräsentation für den Menschen als Nutzer ausgelegt ist und es für Maschinen somit nicht möglich ist diese zu interpretieren und weiterzuverarbeiten. Dieses Problem der maschinellen Verarbeitung versucht das Semantic Web mit Hilfe von einheitlichen Standards zu lösen. Dabei werden die Daten direkt in maschinenlesbarer Form dargestellt.

In Abbildung 2[6] wird der Layer Cake des Semantic Web dargestellt, der folgende Standards verwendet. Als Grundlage für die Eindeutigkeit werden so genannte Uniform Resource Identifiers (URI's) verwendet, um Objekte darzustellen. Diese werden durch Angabe einer Adresse, z.B. *http://purl.org/dc/elements/1.1/title* dargestellt. Dabei ist es nicht wichtig, ob diese URI's auf eine Internetseite verweisen oder nicht, es wird aber als gute Praxis angesehen, wenn diese tatsächlich Informationen über die entsprechende Ressource enthalten. Weiterhin verwendet das Semantic Web XML als Basis, da es eine standardisierte Meta-Sprache ist, die bereits Maschinenlesbarkeit bereitstellt. Auf diesen Grundlagen bauen die Ontologiesprachen RDF und OWL auf, die im nächsten Abschnitt genauer vorgestellt werden. Darauf wird eine vereinigende Logik mit Hilfe von Beweisen und Vertrauen definiert. Als letztes definiert Semantic Web noch Benutzerschnittstellen für Anwendungen.

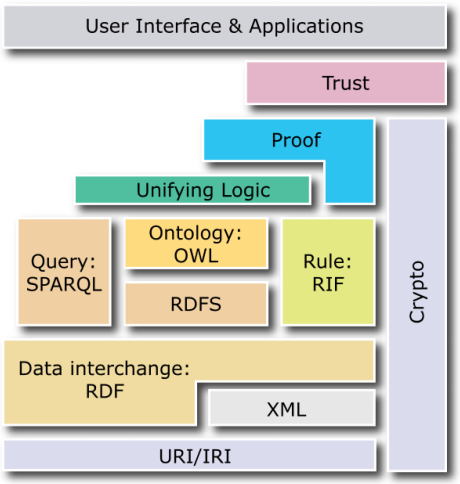

Abbildung 2: Darstellung des Layer Cakes des Semantic Web, das in Schichten aufgebaut ist und einheitliche Standards wie XML verwendet.

[6]Quelle:http://www.w3c.it/talks/2008/wsb08/images/layerCake.png

2.1 RDF

Resource Description Framework[7] (RDF) ist eine formale Sprache zur Beschreibung strukturierter Informationen, durch die der Anwender in der Lage ist, Daten auszutauschen ohne dabei deren Bedeutung zu verlieren. Die Darstellung wird meist durch einen gerichteten Graph realisiert, kann aber auch mit Hilfe von XML, durch Tripel dargestellt werden. Von einer hierarchischen Anordnung wird meist abgesehen, da diese nicht immer sinnvoll ist, z.B. wenn sich keiner der dargestellten Begriffe unterordnen lässt.[HKRS08] RDF Tripel bestehen aus drei Bestandteilen, wie in Abbildung 3 zu sehen ist.

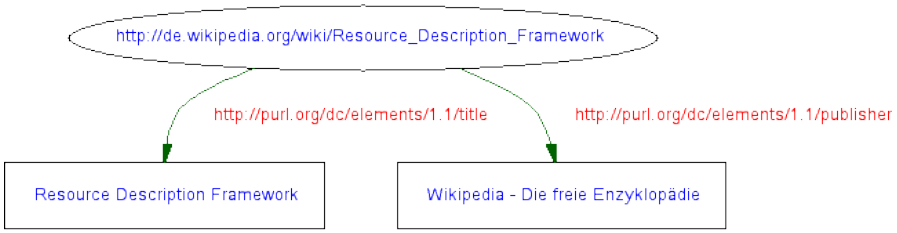

Abbildung 3: Einfacher RDF Graph. Dabei besteht dieser aus zwei Tripeln.

1. **Subjekt(Ressource):**

   Ressourcen sind alle Dinge, die durch RDF Ausdrücke beschrieben werden können. Sie werden durch Ellipsen in der grafischen Darstellung modelliert.

2. **Prädikat(Eigenschaftselement):**

   Das Eigenschaftselement gibt Auskunft über die ihm zugeordnete Ressource und stellt den Bezug zum Objekt her, verbindet also die Ressource mit dem Objekt. In der grafischen Modellierung wird das Eigenschaftselement durch eine benannte Kante dargestellt.

3. **Objekt(Ressource oder Literal):**

   Das Objekt beschreibt den Wert eines Prädikates. Dabei kann das Objekt auch ein *Blank Node*, also ein leerer Knoten sein. Objekte werden als Rechtecke in der grafischen Modellierung dargestellt.

Eine weitere Sprache zur Wissensrepräsentation ist OWL, welches im nächsten Abschnitt genauer erklärt wird.

[7] http://www.w3.org/RDF/

2.2 OWL

Web Ontology Language[8] (OWL) ist eine Wissensrepräsentationssprache, welche auf RDF aufbaut und komplexes Wissen effizient darstellen kann. Es basiert auf Beschreibungslogik und ermöglicht somit ein Gleichgewicht zwischen Ausdrucksstärke und effizientem Schlussfolgern. Da Ontologiesprachen, die Sprachkonstrukte mit hoher Komplexität zur Verfügung stellen, oft unentscheidbar sind, gibt es in OWL 3 verschiedene Versionen, wie OWL Lite, OWL DL und OWL Full, die den Anwender eine Wahl zwischen verschiedenen Ausdrucksstärken lasst.[HKRS08]

1. **OWL Lite**

   OWL Lite ist eine Teilsprache von OWL DL und OWL Full, mit dem Ziel eine einfach zu implementierende Teilsprache bereitzustellen, die die wichtigsten Sprachelemente enthält. Diese Teilsprache heißt *SHIF*. Dadurch ist OWL Lite entscheidbar, aber weniger Ausdrucksstark.

2. **OWL DL**

   OWL DL(Description Logic) enthält alle Sprachkonstrukte von OWL Lite, ist aber nur eine Teilsprache von OWL Full. Es ist äquivalent zu einer entscheidbaren Untermenge der Prädikatenlogik erster Stufe und heißt *SHOIN(D)*.

3. **OWL Full**

   OWL Full besteht aus den selben Sprachkonstrukten wie OWL DL, verzichtet aber auf die dort eingeführten Einschränkungen. Dadurch werden prädikatenlogische Ausdrücke eines höheren Grades erreicht, aber die Ontologien werden damit unentscheidbar.

Weiterhin werden Klassen, Eigenschaften und Individuen, die die Instanzen von Klassen darstellen als Grundbausteine von OWL Ontologien deklariert. Dabei gibt es in OWL zwei spezielle Klassen, die jede Ontologie enthält, nämlich *owl:thing* und *owl:nothing*. Die Klasse *owl.thing* ist dabei die allgemeinste Klasse, die alles enthält. Wohingegen *owl:nothing* im Allgemeinen leer ist. Im Folgenden werden nun ein paar Sprachkonstrukte vorgestellt, die Beziehungen von Klassen und Instanzen modellieren.

2.2.1 Sprachkonstrukte in OWL

In OWL gibt es verschiedene Sprachkonstrukte, mit deren Hilfe man die Abhängigkeiten zwischen Klassen und Individuen darstellen kann. Die Folgenden sind die Wichtigsten, die vor allem für das Verständnis des Plugins notwendig sind.

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

1. Axiome

   - **rdfs:subClassOf**

     Diese Relation drückt aus, dass eine Klasse eine Unterklasse von einer anderen ist, also ein spezielleres Konzept besitzt. Des Weiteren gilt, dass jede Klasse eine Unterklasse von *owl:thing* ist und *owl:nothing* Unterklasse von allen anderen Klassen. Ein Beispiel wäre hier $Student \subseteq Person$, denn ein Student ist eine Person, besitzt aber bestimmte Eigenschaften, die nicht jede Person hat, ist also ein spezielleres Konzept als Person.

   - **owl:equvalentClass**

     Durch diese Relation werden zwei Klassen als äquivalent deklariert, d.h. sie haben die gleichen Instanzen, was das Beispiel $Mann \equiv nichtFrau$ verdeutlicht. Dabei wird verdeutlicht, dass ein Mann äquivalent zu keiner Frau ist.

   - **owl:disjointWith**

     Mit diesem Konstrukt können zwei Klassen als disjunkt erklärt werden, d.h. dass kein Individuum der einen Klasse, in der anderen vorkommen darf. $Mann \neq Frau$ zeigt, dass kein Individuum, welches als Mann deklariert wurde, auch eine Frau sein kann.

2. Klassenbeschreibungen

   - **owl:intersectionOf**

     IntersectionOf wird als logisches „und“ aufgefasst und umfasst die Individuen, die gleichzeitig zu allen Klassen gehören, die an der Intersection beteiligt sind. Das Beispiel $Mediziner \equiv Mann \cap Arzt$ zeigt, dass ein Mediziner männlich und Arzt sein muss, also alle Eigenschaften erfüllen muss.

   - **owl:unionOf**

     Mit *unionOf* wird das logische „oder“ ausgedrückt und umfasst die Individuen zweier Klassen, die entweder zu einer der beiden oder zu beiden Klassen gehören. Hier wäre z.B $Person \equiv Mann \cup Frau$ ein Beispiel, da eine Person männlich oder weiblich sein kann.

   - **owl:complementOf**

     Durch *complementOf* drückt man das Komplement einer Klasse aus, also genau die Individuen, die nicht zu der Klasse selbst gehören. $\neg Mann$ wäre das Komplement von Mann und meint genau die Individuen, die nicht männlich sind.

   - **owl:allValuesFrom**

     Mit dem Konstrukt *allValuesFrom* kann eine Aussage über die Individuen der Klasse getroffen werden. Dabei müssen hier alle Individuen eine bestimmt Eigenschaft erfüllen. Mit dem Beispiel $Kind \equiv Person \cap \forall Alter \leq 14$ wird ausgedrückt, dass ein Kind eine Person sein muss, bei der alle Instanzen ein Alter kleiner oder gleich 14 haben müssen.

- **owl:someValuesFrom**

  Im Gegensatz zu *owl:allValuesFrom* wird hiermit ausgedrückt, dass mindestens ein Individuum eine bestimmte Eigenschaft erfüllen muss um Instanz der Klasse zu sein. Das Beispiel $Vater \equiv Mann \cap \exists Kind$ drückt aus, dass ein Vater männlich sein muss und mindestens ein Kind besitzen muss.

- **owl:hasValue**

  Mit dem Konstrukt *owl:hasValue* drückt man aus, dass eine Klasse nur Instanzen von einem bestimmten Typ haben darf. Mit dem Beispiel $Fleischgericht \equiv Mahlzeit and has Value Fleisch$ kann man angeben, dass alle Fleischgerichte Mahlzeiten vom Typ Fleisch sein müssen.

- **owl:minCardinality**

  Hiermit wird angegeben, wie viele Individuen mit bestimmten Eigenschaften eine Klasse mindestens haben kann. Das Beispiel aller Väter mit mindestens drei Kindern kann durch $VaterMitWenigerAls3Kindern \equiv Mann \cap AnzahlKinder \leq 3$ angegeben werden.

- **owl:maxCardinality**.

  Durch *owl:maxCardinality* wird die maximale Anzahl von Individuen mit bestimmten Eigenschaften angegeben, die eine Klasse besitzen darf. Hier kann als Beispiel die Klasse mit Vätern, die mehr als drei Kinder haben, angegeben werden $VaterMitMehrAls3Kindern \equiv Mann \cap AnzahlKinder \geq 3$.

2.2.2 Manchester OWL Syntax

Mit der Manchester OWL Syntax kann man komplexe Klassenbeschreibungen aufschreiben, die leichter zu lesen und zu schreiben sind. Es wurde sowohl von der abstrakten OWL Syntax, als auch der Syntax der Beschreibungslogik beeinflusst. In der folgenden Tabelle 1[9]. werden einige Konstrukte in Manchester Syntax dargestellt.

[9] Quelle:http://www.co-ode.org/resources/reference/manchester_syntax

Tabelle 1: Manchester OWL Syntax Beispiele mit Erklärung

Manchester OWL Syntax	Erläuterung
hasChild some Man	Beschreibung eines Mannes, der Kinder hat, die männlich sind.
hasSibling only Woman	Darstellung einer Person, die nur weibliche Zwillinge hat.
hasCountryOfOrigin value England	Hier wird etwas beschrieben, was als Ursprungsland England hat.
hasChild min 3	Es wird eine Person, die mindestens drei Kinder hat, beschrieben.
hasChild exactly 3	Beschreibung einer Person mit genau drei Kindern
hasChild max 3	Hierbei handelt es sich um eine Klasse, die höchstens drei Kinder hat.
Doctor and Female	Beschreibung einer Klasse der weibliche Doktoren
Man or Woman	Darstellung einer Klasse, die Männer oder Frauen als Instanzen enthält.
not Child	Hier wird die Negation von Kind dargestellt

3 Protégé

Protégé ist einer der am häufigsten genutzten OWL-Ontologie Editoren und wurde ursprünglich dazu verwendet, Wissensdatenbanken oder Ontologien zu medizinischen Forschungszwecken zu erstellen. Heute ist es eine Open Source Anwendung, die unter der Mozilla Public License frei zur Verfügung steht. Es bietet eine Vielzahl an Werkzeugen zur Erstellung, Darstellung und Verarbeitung von Ontologien.

3.1 Funktionsweise und Überblick

Protégé bietet einen OWL Editor zur Modellierung und Modifizierung von RDF/OWL Ontologien an, welcher es ermöglicht, auf verteilte Ontologien, die in einem Dokument zusammengefasst wurden, zuzugreifen. Dafür bietet es einen extra Tab für die Erstellung der Klassen, der Eigenschaften und der Individuen, um die einzelnen Eigenschaften einzusehen und zu modifizieren. Die Abbildung 4 zeigt die Ansicht für die Klassen mit ihren Individuen.

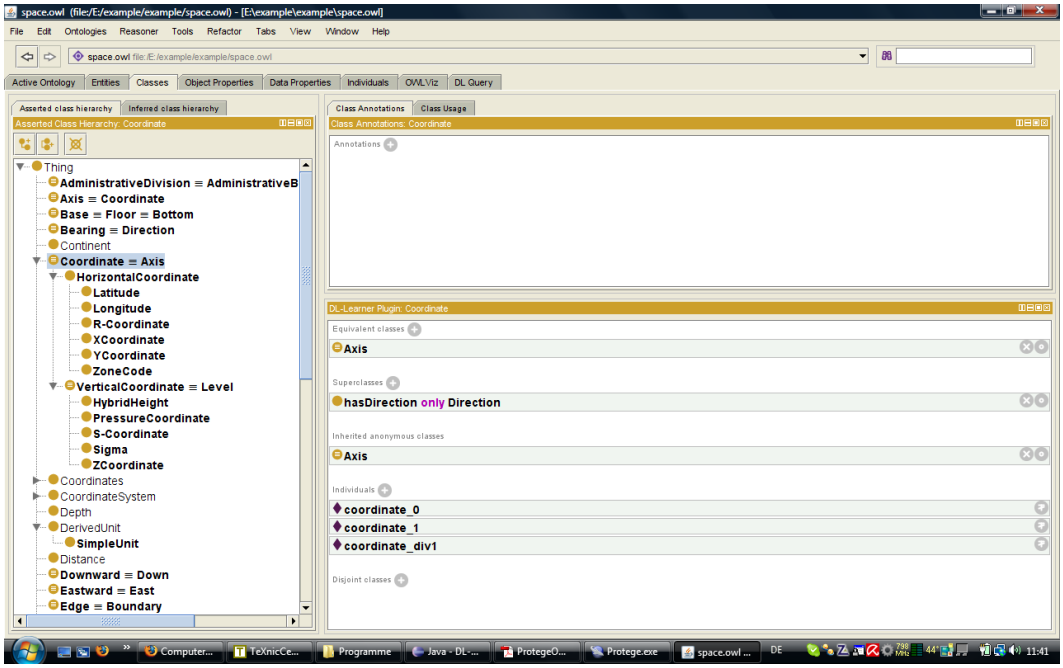

Abbildung 4: Klassenansicht in Protégé . Dabei wird links die Hierarchie der Klassen dargestellt und rechts unten werden die Superklassen, die äquivalenten Klassen und die Individuen, die zu einer Klasse gehören angezeigt.

Des Weiteren bietet es dem Nutzer folgende Fähigkeiten:

- Laden und speichern von RDF und OWL Ontologien
- Bearbeiten und Visualisieren von Klassen, Properties und SWRL[10] Regeln
- definieren von logischen Klassenmerkmalen als OWL Ausdrücke
- Ausführung von Reasonern
- Editieren von OWL Individuen

3.2 Die Pluginarchitektur von Protégé

Protégé besitzt eine ausgereifte Pluginarchitektur, deshalb ist es sehr einfach neue Plugins zu integrieren. Hierfür müssen lediglich drei Dinge beachtet werden.

1. Eine abstrakte Plugin Klasse implementieren, wie z.B. AbstractOWLClassViewComponent. Dafür müssen die folgende Methoden implementiert werden.
   - initialiseClassView() Diese Methode initialisiert den View des erstellten Plugins.
   - updateView() wird aufgerufen, sobald das Plugin Änderungen im View erzeugt.
   - disposeView() Diese Methode wird verwendet, wenn das Plugin beendet wird, um den View des Plugins zu zerstören.
2. Eine XML Datei die das Plugin beschreibt, sodass Protégé die auszuführenden Dateien findet. Es beschreibt das Plugin durch eine Extension ID, ein Label und einen Pointer, der auf die Klasse zeigt, welche die abstrakte Pluginklasse implementiert.
3. Eine Manifest Datei, welche die Abhängigkeiten des Plugins zu anderen Bibliotheken beschreibt. Die Manifestdatei muss sich in dem Ordner META-INF befinden, damit Protégé es erkennen kann.[11]

[10] Semantic Web Rule Language
[11] Quelle:http://www.co-ode.org/downloads/protege-x/plugin-code-example.php

4 DL-Learner

DL-Learner[12] steht für Beschreibungslogik Lerner (Description Logic Learner), dessen Ziel es ist, ein DL/OWL basiertes Machine Learning Tool zur Verfügung zu stellen, das überwachte Lernaufgaben (engl. supervised learning) lösen soll und Wissensingenieuren bei der Erstellung von Wissensbasen hilft.

Als Eingabe erhält der DL-Learner ein Lernproblem, positive und negative Beispiele und einen Lernalgorithmus. Als Wissensbasis nutzt er eine Ontologie und einen Reasoner zur Klassifizierung dieser. Er überprüft die Wissensbasis und gibt mit Hilfe der positiven und negativen Beispiele neue Konzepte an, die er für geeignet hält. Die Generierung der Konzepte wird mit Hilfe eines Refinement Operators oder der genetischen Programmierung realisiert. Im Anschluss wird geprüft, ob das vorgeschlagene Konzept alle angegebenen Beispiele abdeckt. Anhand dieser Abdeckung wird das Qualitätsmaß des Konzeptes bestimmt und geprüft ob es hinreichend geeignet ist. Dieser Prozess wird so lange wiederholt, bis ein Konzept gefunden wurde, welches die Anforderungen an das Qualitätsmaß erfüllt. Die Abbildung 5 stellt dies noch einmal grafisch dar.

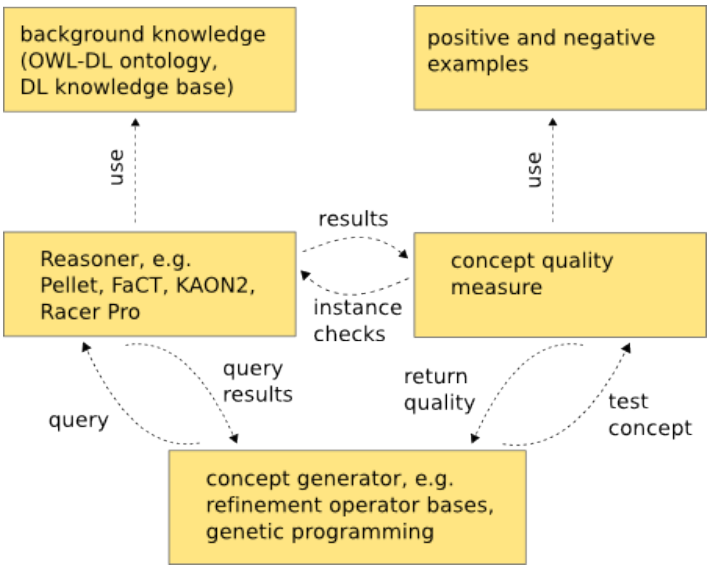

Abbildung 5: „Generate and Test“ Ansatz, der beim DL-Learner zum lernen von Konzepten verwendet wird.

[12] http://dl-learner.org

4.1 Die Architektur des DL-Learners

Der DL-Learner nutzt ein auf Komponenten basierendes Modell und besteht aus vier verschiedenen Komponententypen: der Wissensbasis, dem Lernproblem, dem Lernalgorithmus und dem Reasoning Service. Für jeden Komponententyp gibt es mehrere Komponenten, z.B. vom Typ Lernalgorithms, einem auf genetischer Programmierung basierenden Ansatz und einem auf dem sogenannten Refinement-Operator basierenden Ansatz. Dadurch ist es sehr einfach eine neue Komponente zu den bestehenden hinzuzufügen. Es muss einfach die jeweilige Klasse der Komponente erweitert werden und in die *components.ini* eingetragen werden. Trotz der leichten Integration neuer Komponenten in den DL-Learner, besteht eine gewisse Abhängigkeit bei der Initialisierung, sodass man bei der Integration des DL-Learners, in andere Tools, auf bestimmte Aspekte achten sollte. So ist es z.B. nicht möglich einen Reasoner zu initialisieren, wenn noch keine Wissensbasis angegeben wurde. Bevor man ein Lernproblem auswählen kann, muss der Reasoner angegeben werden und bevor der Lernalgorithmus angegeben werden kann, muss das Lernproblem festgelegt werden.

4.2 Lernprobleme

Das Lernproblem gibt an, ob es sich um ein ein-,zwei- oder dreiwertiges Lernproblem handelt, d.h. ob nur positive Beispiele oder ob positive und negative Beispiele angegeben werden müssen. Bei der Angabe von positiven und negativen Beispiele wird noch unterschieden, ob die Beispiele, die nicht positiv oder negativ sind, mit betrachtet werden oder nicht. Wenn diese mit betrachtet werden, handelt es sich um ein dreiwertiges Lernproblem. Im Folgenden werden die Lernprobleme kurz beschrieben, die in dem Plugin Verwendung finden.

PosNegDefinitonLP

Das PosNegDefinitionLP ist ein zweiwertiges Lernproblem, dass das Ziel hat, ein Konzept zu finden, bei dem die positiven Beispiele aus der Definition folgen und die Negativen nicht. Zum Beispiel kann eine Klassenbeschreibung $father \equiv Male \cap \exists hasChild.TOP$ gelernt werden, indem Männer mit Kindern als positive Beispiele und Männer ohne Kinder und Frauen als negative Beispiele angegeben werden.

PosNegInclusionLP

Das PosNeginclusionLP hat das Ziel ein passendes Inklusionsaxiom zu finden, das durch positive und negative Beispiele gegeben ist. Es ist ähnlich wie bei dem PosNegDefinitionLP, mit der Ausnahme, dass die positiven und negativen Beispiele nicht aus dem Axiom folgen, wenn sie zur Wissensbasis

hinzugefügt wurden. Das Problem bei diesem Lernproblemes ist die Messung der Qualität der gefundenen Konzepte. Die Lösung ist, dass man sich die Negation des gefundenen Konzeptes betrachtet, d.h. für gute Lösungen dürfen keine positiven Beispiele aus der Negation des Konzeptes folgen. Als Beispiel kann man die Klassenbeschreibung *essbare Dinge* als Superklasse von *Mahlzeit* lernen. Hier wird nun die Negation, *not essbare Dinge*, der Beschreibung genommen und überprüft, ob aus dieser positive Beispiele folgen.

4.3 Lernalgorithmus

Es gibt zwei verschiedene Arten von Lernalgorithmen, zum Einen ein auf dem Refinement-Operator basierter Ansatz[LH08b] und zum Anderen ein Ansatz, der mit Hilfe von genetischer Programmierung[LH08a] realisiert wird. Im Folgenden wird der verwendete *ExampleBasedROLComponent* Algorithmus beschrieben.

ExampleBasedROLComponent

Dieser Algorithmus ist ein, auf einem downward Refinement-Operator basierender Lernalgorithmus, der eine Klasse auf eine Menge von spezielleren Klassen abbildet. Dabei geht der Algorithmus wie folgt vor. Er baut einen Suchbaum auf und fängt bei der obersten Klasse owl:thing an. Anschließend werden die Klassen, die in der Ontologie vorhanden sind, mit Hilfe der positiven und negativen Beispiele überprüft. Eine Heuristikfunktion wählt den vielversprechendsten Knoten im aktuellen Suchbaum aus. Auf diesem wird der Refinement-Operator angewendet. Dieser Vorgang wird so lange wiederholt bis eine hinreichend gute Klasse gefunden wurde. In der Abbildung 6 wird dieser Baum anhand eines Beispieles dargestellt.

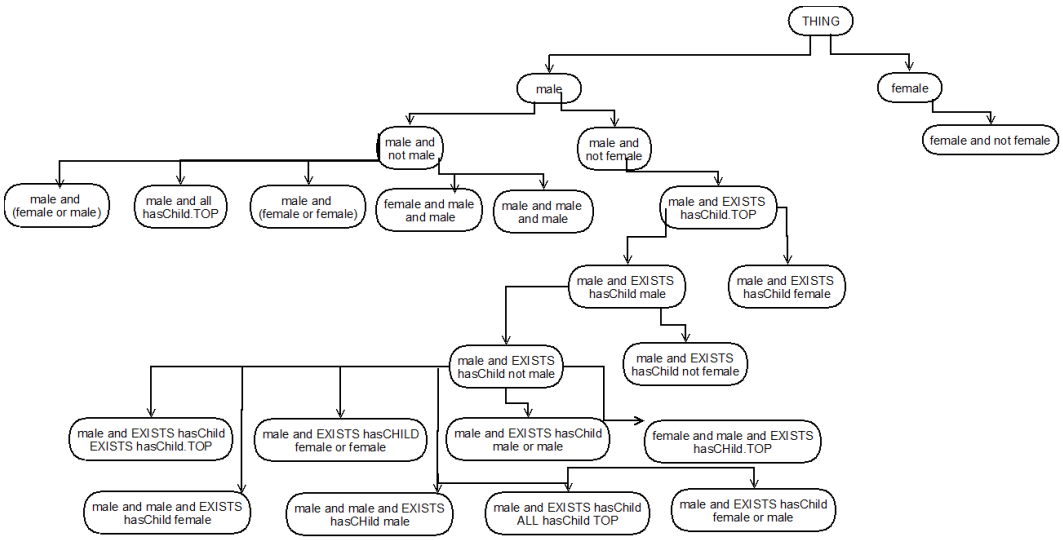

Abbildung 6: Suchbaum des Lernalgorithmus. Dabei werden die vorgeschlagenen Klassenbeschreibungen zunehmend komplexer.

5 Das Protégé Plugin

5.1 Einleitung

Das Protégé Plugin soll dem Nutzer helfen komplexe Klassen zu erstellen, wodurch es möglich ist ausdrucksstarke Ontologien zu erstellen. Dabei sollen Klassen vorgeschlagen werden können, die die bereits bestehende Ontologie bereichert und zur Verbesserung der Ausdrucksstärke beiträgt. Hierzu werden die Individuen, die zu der ausgewählten Klasse gehören als positive Beispiele und alle Anderen, die nicht zu dieser Klasse gehören als negative Beispiele automatisch selektiert. Der Nutzer kann diese Auswahl verändern um das Lernen von neuen Klassenbeschreibungen anzupassen und zu verbessern.

5.2 Anforderungen

Da es sich bei diesem Plugin um ein Plugin zur Unterstützung von Wissensigeneuren handelt, sollte die Benutzeroberfläche sehr einfach und leicht verständlich sein, um dem Nutzer einen schnellen Einstieg zu gewährleisten

Funktionale Anforderungen

Die grafische Benutzeroberfläche ist in drei Bereiche unterteilt. Im ersten Bereich soll der Lernalgorithmus über einen Button gestartet und abgebrochen werden können. Weiterhin soll in diesem Bereich über eine Liste, die vom DL-Learner vorgeschlagenen Klassenbeschreibungen ausgewählt werden können und zur Ontologie hinzugefügt werden. In dem zweiten Bereich, den der Nutzer nur über einen Button erreicht, sollen über Checkboxen die positiven und negativen Beispiele ausgewählt werden können. Ebenfalls kann hier die Auswahl auch rückgängig gemacht werden. Der dritte Bereich ist ein extra Fenster, welches durch Doppelklick auf ein vorgeschlagenes Konzept angezeigt wird. In diesem werden weitere Details zu dem vorgeschlagenen Konzept angezeigt.

5.3 Spezifikation

Aufbauend auf diesen Anforderungen ergeben sich folgende Produktfunktionen.

1. **Produktfunktion:** Anzeigen des Panels zur Auswahl der positiven und negativen Beispiele

   Akteur: Benutzer

   Beschreibung: Der Benutzer soll durch Klick auf einen Button ein Panel zur Auswahl der positiven und negativen Beispiele angezeigt bekommen.

2. **Produktfunktion:** Auswahl der Beispiele die positiv oder negativ sind

   Akteur: Benutzer

   Beschreibung: Der Benutzer soll durch ankreuzen der Individuen auswählen, ob diese zu den positiven oder negativen Beispielen gehören.

3. **Produktfunktion:** Starten des Lernalgorithmus

   Akteur: Benutzer

   Beschreibung: Der Benutzer soll durch Klick auf den Lernbutton den Lernalgorithmus starten können.

4. **Produktfunktion:** Abbruch des Lernalgorithmus

   Akteur: Benutzer

   Beschreibung: Der Benutzer soll die Möglichkeit haben den Lernalgorithmus abzubrechen.

5. **Produktfunktion:** Auswahl einer Klassenbeschreibung und hinzufügen zu der Ontologie

   Akteur: Benutzer

   Beschreibung: Der Benutzer soll aus einer Liste eine vorgeschlagene Klassenbeschreibung auswählen und diese per Knopfdruck zur Ontologie hinzufügen können.

6. **Produktfunktion:** Anzeige der Details zu den vorgeschlagenen Klassenbeschreibungen.

   Akteur: Benutzer

   Beschreibung: Der Benutzer soll nach Doppelklick eines Konzeptes aus der Liste ein Fenster mit Details, wie Genauigkeit und abgedeckte positive und negative Beispiele, angezeigt bekommen.

Die Abbildung 7 zeigt das Klassendiagramm des Plugins. Dabei wurde ein MVC-Architektur-Prinzip realisiert, d.h. es gibt Komponenten die für die Anzeige der grafischen Oberfläche verantwortlich sind, Komponenten die für die Behandlung, der vom Nutzer getätigten Eingaben verantwortlich sind und Komponenten, die für die Manipulation der Daten verantwortlich sind. Eine Beschreibung der einzelnen Klassen folgt nun.

- **ProtegePlugin**

  Die Klasse *ProtegePlugin* implementiert die abstrakte Pluginklasse, in diesem Fall *AbstractOWLClassViewComponent*, was darauf hinweist, dass es sich um eine Ansichtskomponete für Klassen handelt.

- **ButtonList**

  In der *ButtonList* werden die Klassen *OWLEquivalentClassAxiomFrameSection* und *OWLSubClassAxiomFrameSection* gespeichert.

- **OWLEquivalentClassAxiomFrameSection**

  Hiermit wird die Ausgabe der äquivalenten Klassen einer Ontologie dargestellt und der Klassenbeschreibungseditor aufgerufen.

- **OWLSubClassAxiomFrameSection**

  Übernimmt die selbe Aufgabe wie *OWLEquivalentClassAxiomFrameSection*, jedoch für Subklassen.

- **OWLClassDescriptionEditorWithDLLearnerTab**

  Diese Klasse übernimmt die Anzeige des Klassenbeschreibungeditor, welcher um eine inneren Klasse *DLLearnerView* erweitert wurde, die die Anzeige des DL-Learner Plugins übernimmt.

- **SuggestClassPanel**

  Diese Komponente übernimmt die Anzeige der vom DL-Learner vorgeschlagenen Klassenbeschreibungen.

- **PosAndNegSelectPanel**

  Um komplexe Klassenbeschreibungen lernen zu können, müssen positive und negative Beispiele ausgewählt werden. Dafür ist diese Klasse zuständig.

- **MoreDetailForSuggestedConceptsPanel**

  Hiermit kann man sich Details, wie Genauigkeit und abgedeckte positive und negative Beispiele, zu dem ausgewählten Konzept anzeigen lassen.

- **ActionHandler**

  Das ist der Controller des Plugins, der die Verarbeitung der Ereignisse durchführt.

- **DLLearnerModel**

  Im *DLLearnerModel* findet der eigentlich Lernvorgang statt. Hier werden unter anderem auch Lernproblem, Reasoner und der Lernalgorithmus festgelegt. Ebenfalls werden hier die vom DL-Learner vorgeschlagenen Klassenbeschreibungen gespeichert und in Manchester OWL Syntax umgewandelt, bevor sie in die Ontologie hinzugefügt werden.

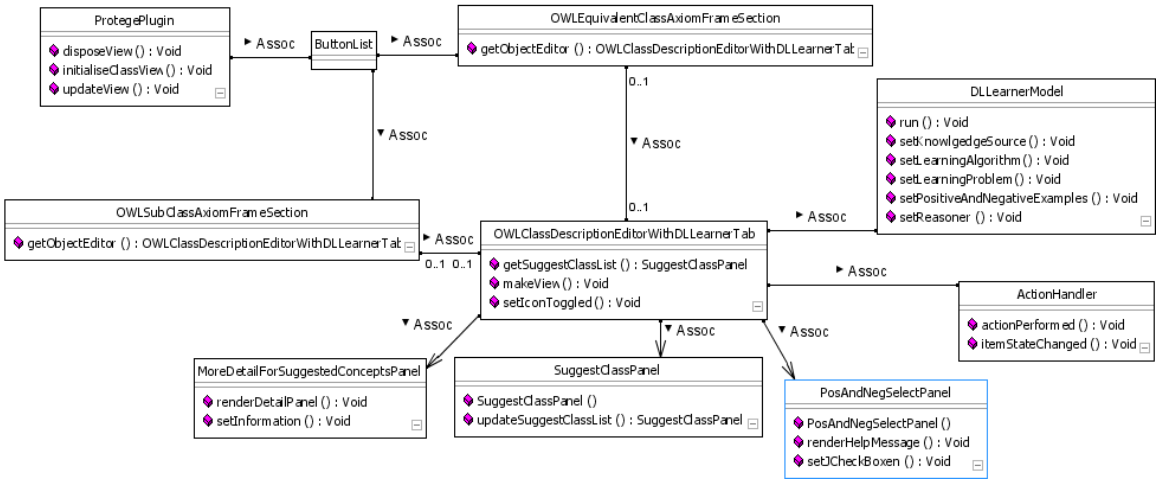

Abbildung 7: Klassendiagramm des Plugins. Der Übersichtlichkeit halber wurden nur die wichtigsten Methoden dargestellt.

5.4 Implementierung und Funktionsweise

Bei der Implementierung wurde stets darauf geachtet, dass das MVC Prinzip realisiert wurde. Deshalb ist ein separater Controller für die Behandlung der Nutzereingaben entstanden. Ebenfalls wurde ein Model für die Daten, die für den Austausch zwischen Protégé und dem DL-Learner notwendig sind, erzeugt und auch die grafische Oberfläche wurde in mehrere Teile unterteilt, um eine bessere Implementierung zu gewährleisten.

Dabei mussten weitere Dateien implementiert werden, die für die Erkennung des Plugins in Protégé notwendig waren. Die in Listing 1 dargestellte XML Datei ist für die Identifikation des Plugins notwendig. Dabei muss der Pfad zu der Klasse angegeben werden, welche die abstrakte Pluginklasse implementiert. Ebenso muss der Name des Plugins unter *label* eingetragen werden.

Ausserdem muss eine Manifestdatei erstellt werden, welche die Abhängigkeiten des Plugins zu anderen Bibliotheken darstellt. Diese Datei wird in Listing 2 dargestellt. Hier wird unter anderem der ClassPath zu den benötigten Bibliotheken gesetzt, damit Protégé diese finden kann. Im nächsten Abschnitt wird auf Schwierigkeiten bei der Entwicklung des Plugins eingegangen.

Listing 1: XML Datei des Protégé Plugins

```
<?xml version="1.0" ?>
  <plugin>
    <extension id="org.dllearner.tools.protege.ProtegePlugin"
      point="org.protege.editor.core.application.ViewComponent">
      <label value="DL-Learner Plugin"/>
      <class value="org.dllearner.tools.protege.ProtegePlugin"/>
      <headerColor value="@org.protege.classcolor"/>
      <category value="@org.protege.classcategory"/>
    </extension>
  </plugin>
```

Listing 2: Manifestdatei des Protégé Plugins

```
Manifest-Version: 1.0
Bundle-Name: DL-Learner Plugin
Bundle-SymbolicName: org.dllearner.tools.protege;singleton:=true
Bundle-Category: protege
Bundle-ClassPath: .,lib/junit-4.4.jar,lib/jamon-2.7.jar
Import-Package: org.osgi.framework,org.apache.log4j
Export-Package: lib
Bundle-Activator: org.protege.editor.core.plugin.
   DefaultPluginActivator
Require-Bundle: org.eclipse.equinox.registry,org.eclipse.equinox.
   common
```

Probleme bei der Implementierung

Da Protégé in dieser Version noch keine Möglichkeit des Testens, direkt in Eclipse anbietet, wurde ein Ant Build Script geschrieben, welches die benötigten Klassen und Bibliotheken zusammen mit den oben erwähnten Dateien in ein jar Archiv packt und direkt in das Pluginverzeichnis von Protégé kopiert.

Ein weiteres Problem ist bei der Implementierung des neuen Tabs aufgetreten. Da es keinen Erweiterungspunkt für die Entwicklung eines neuen Tabs für den Klassenbeschreibungseditor gab, wurde der bestehende Klassenbeschreibungseditor genommen und um eine weitere innere Klasse ergänzt, die die Funktionalität des DL-Learners bereit stellt. Dadurch mussten auch die Klassen *OWLEquivalentClassAxiomFrameSection* und *OWLSubClassAxiomFrameSection* angepasst werden, da diese nun nicht mehr auf den standard Klassenbeschreibungseditor zugreifen durften. Durch diese Anpassung des Klassenbeschreibungseditor kann es zu Problemen bei neueren Versionen von Protégé kommen, da diese eventuell eine Änderung erfahren haben, die diese Inkompatibel mit der aktuell benutzten Version macht.

Beschreibung der Klassen und deren Funktionen

An dieser Stelle folgt nun eine Beschreibung der wichtigsten Klassen und deren Methoden.

Die wichtigste Klasse für den Aufbau der grafischen Oberfläche des Plugins ist die Klasse *DLLearnerView*, die eine innere Klasse von *OWLClassDescriptionEditorWithDLLearnerTab* ist. Die folgenden Methoden sind dabei wichtig.

- **DLLearnerView()**

  Das ist der Konstruktor der Klasse *DLLearnerView*. Hier werden alle Panels und Buttons der grafischen Oberfläche Initialisiert und die Komponenten an die Listener angemeldet. Abgesehen davon, wird auch noch der *ActionHandler* und das *DLLearnerModel* instantiiert.

- **makeView()**

  Diese Methode setzt die einzelnen Komponenten an die richtige Position im DL-Learner Tab und startet den *SimpleSuggestionLearningAlgorthm*, um die ersten Vorschläge anzuzeigen.

- **getSolutions()**

  Nach dem Schließen des Fensters gibt diese Methode alle hinzugefügten Klassenbeschreibungen an die grafische Oberfläche von Protégé weiter, die der Nutzer hinzugefügt hat.

- **getSolution()**

  Hier wird nur eine Klassenbeschreibung an die grafische Oberfläche zurückgegeben, und zwar die letzte, die der Nutzer zur Ontologie hinzugefügt hat.

Der *ActionHandler* implementiert die Funktionalität der Listener, die sich um die Nutzereingabe kümmern und die jeweils passenden Aktionen ausführen.

- **actionPerformed()**

  Bei dieser Methode wird zwischen den einzelnen Aktionen unterschieden, die beim Drücken der einzelnen Buttons geschehen müssen. Anschließend wird die passende Aktion ausgeführt.

- **itemStateChanged()**

  Diese Methode verwaltet das Hinzufügen und Löschen von positiven und negativen Beispielen beim Klicken auf die Checkboxen der einzelnen Beispiele.

- **mouseClicked()**

  Hier wird, bei jedem klick auf ein Konzept in der Liste, das *evaluatedDescription* Objekt gesucht, dass nötig ist für die Anzeige der weiteren Informationen und zum Hinzufügen des Konzeptes zur Ontologie.

- **mousePressed()**

  Mit dieser Methode wird der Button zum Hinzufügen der Konzepte aktiviert. Dies geschieht nachdem etwas ausgewählt wurde.

Das *DLLearnerModel* implementiert alle Funktionen die zur Kommunikation mit dem DL-Learner notwendig sind und konvertiert die vom DL-Learner vorgeschlagenen Klassenbeschreibungen in OWLAPI Klassenbeschreibungen.

- **initReasoner()**

  Mit dieser Methode wird der einfache Lernalgorithmus *SimpleSuggestionLearningAlgorithm* gestartet und die ersten neu gelernten Klassenbeschreibungen zurückgegeben.

- **addToListModel()**

  Diese Methode fügt die gelernten Konzepte in das Model für die Ausgabeliste der grafischen Oberfläche ein. Dabei werden die Neusten immer an die erste Stelle geschrieben.

- **setPositiveAndNegativeExamples()**

  Mit Hilfe dieser Funktion werden die Beispiele die als positiv oder negativ deklariert wurden, ausgewählt und für den Lernalgorithmus zur Verfügung gestellt.

- **setKnowledgeSource()**

  Bei *setKnowledgeSource()* wird die Wissensbasis für den Lernalgorithmus bereitgestellt. Hierbei wurde eine neue Klasse *OWLAPIOntologie* erstellt, die eine ganze Ontologie übergibt, sodass diese beim Lernen nicht erneut geladen werden muss.

- **setReasoner()**

  Der Reasoner wird mit Hilfe dieser Methode festgelegt.

- **setLearningProblem()**

  Es gibt zwei Lernprobleme, das *PosNegDefinitionLP*, welches genutzt wird um äquivalente Klassen vorzuschlagen und das *PosNegInclusionLP*, welches genutzt wird um Subklassen vorzuschlagen. Dabei wird die Wahl, welches Lernproblem verwendet werden muss, über die ID geregelt, die beim starten des Plugins übergeben wird.

- **setLearningAlgorithm()**

  Hier wird der Lernalgorithmus festgelegt, mit dem neue Klassenbeschreibungen gelernt werden können.

- **run()**

  Das ist die Implementierung des Threads des Lernvorgangs. Dabei wird der Lernalgorithmus gestartet und nach der Ausführung werden die Ergebnisse an das Listenpanel, zur Darstellung der vorgeschlagenen Konzepte weitergegeben.

- **setPositiveConcept()**

  Mit Hilfe dieser Methode wird zu den ausgewählten positiven Beispiele das Konzept aus der Ontologie gesucht. Dies wird später beim Hinzufügen der Klassenbeschreibung benötigt.

- **setNewClassOWLAPI**

  Durch diese Funktion wird das vom DL-Learner vorgeschlagene Konzept in OWLAPI Syntax umgewandelt, damit es beim Hinzufügen in die Ontologie von Protégé erkannt wird.

- **ChangeDLLearnerDescriptionsToOWLAPIDescriptions**

  Diese Methode fügt die durch den Nutzer ausgewählten Klassenbeschreibungen in die Ontologie ein.

Aufrufbeziehungen innerhalb der Klassen

Der nun folgende Abschnitt beschreibt die Aufrufbeziehungen der Klassen während eines Lernvorganges. In Abbildung 8 ist dies grafisch dargestellt. Die Klasse *ProtegePlugin* ist die abstrakte Plugin Klasse, die jedes Plugin implementieren muss. Diese benutzt die beiden Klassen *OWLEquivalentClassAxiomFrameSection* und *OWLSubClassAxiomFrameSection*, um die Listen für äquivalente - und Subklassen darzustellen. In ihnen befindet sich die Methode *getObjectEditor()*, welche die Klasse *OWLClassDescriptionEditorWithDLLearnerTab* aufruft. Diese Klasse enthält eine innere Klasse *DLLearnerView*, die über die Methode *makeView()* die grafische Oberfläche aufbaut. Bevor die Anzeige aufgebaut wird, ruft sie im *DLLearnerModel* die Funktion *initReasoner()* auf, die einen Lernalgorithmus, den *SimpleSuggestionLearningAlgorithm*, startet, der über die Methode *getSimpleSuggestions()* die ersten Vorschläge liefert. Nachdem die Oberfläche aufgebaut ist, hat der Nutzer die Wahl, ob er einen der vorgeschlagenen Konzepte annimmt oder ob er sich noch mehr Konzepte vorschlagen lässt.

Wenn der Nutzer weitere Vorschläge haben will, muss er sich über den Button Advanced, das Auswahlfenster der positiven und negativen Beispiele anzeigen lassen. Dabei wird im *ActionHandler* die Funktion *actionPerformed()* aufgerufen. Diese ruft im *DLLearnerView*, die Methode *setIconToggled()*, die den Pfeil auf dem Advanced Button verändert, und die Funktion *setExamplePanelVisible()*, die das Panel für die positiven und negativen Beispiele sichtbar macht, auf. In diesem Panel kann der Nutzer die positiven und negativen Beispiele auswählen, wobei bei jeder Selektion/Deselektion im *ActionHandler* die Funktion *itemStateChanged()* aufgerufen wird, die im *DLLearnerModel*, die Funktion *getPositiveVector()* aufruft, um das jeweilige Beispiel zu selektieren oder zu deselektieren. Nachdem der Nutzer die Beispiele ausgewählt hat, klickt er auf Suggest Class. Diese ruft im *ActionHandler* die Funktion *actionPerformed* auf. Hier wird zunächst einmal im *DLLearnerModel* mit den Funktionen *setKnowlgedgeSource()*, *setReasoner()*, *setPositiveAndNegativeExamples()*, *setLearningProblem()* und *setLearningAlgorithm()*, die Wissensbasis, der Reasoner, die Beispiele, das Lernproblem und der Lernalgorithmus, festgelegt. Anschließend wird in einem neuen Thread die Methode *run()* im *DLLearnerModel* gestartet, die den Lernalorithmus startet. Nachdem der Lernalgorithmus beendet wurde, wird im *DLLearnerView* die Methode *getSuggestClassList()* aufgerufen, welche die Klasse *SuggestClassPanel* zurückgibt, auf dem die Funktion *setSuggestList()* aufgerufen wird, die die neuen Vorschläge anzeigt. Nun wird die Auswahl der positiven und negativen Beispiele wieder zurückgesetzt, damit ein neuer Lernprozess, vom Nutzer, gestartet werden kann. Dabei wird im *DLLearnerView* die Methode *getPosAndNegSelectPanel()* aufgerufen. Diese gibt die Klasse *PosAndNegSelectPanel* zurück und auf ihr wird dann die Funktion *unsetCheckBoxes()* ausgeführt, die alle Check Boxen zurücksetzt.

An dieser Stelle kann der Nutzer ein vorgeschlagenes Konzept auswählen. Wenn er auf ADD drückt, wird in der Klasse *ActionHandler* die Methode *actionPerformed()* aufgerufen, die überprüft welche

Aktion durchgeführt werden muss. Bei dem Hinzufügen eines neuen Konzeptes ruft der *ActionHandler* im *DLLearnerModel* die Funktion *changeDLLearnerDescriptionToOWLAPIDescription()* auf, die das neue Konzept in die Ontologie einfügt. Zunächst werden die Funktionen *setNewClassOWLAPI()* und *setOldClassOWLAPI()* aufgerufen, die zuerst das vorgeschlagene Konzept in ein OWLAPI Konzept umwandelt. Dies ist Notwendig, damit Protégé das neue Konzept hinzufügen kann. Die zweite Funktion holt sich das alte Konzept, zu dem etwas gelernt werden soll. Nach dem Einfügen wird über den *ActionHandler*, im *DLLearnerView*, die Methode *renderErrorMessage()* aufgerufen, die eine Nachricht anzeigt, ob das Einfügen erfolgreich war. Jetzt können entweder noch mehr Konzept hinzugefügt werden, oder mit Klick auf OK in der Klasse *OWLClassDescriptionEditorWithDLLearnerTab* die Methode *getEditedObjects()* aufgerufen werden, die sich über die Funktion *getSollutions()* in der Klasse *DLLearnerModel*, die vom Nutzer ausgewählten Konzepte holt. Anschließend wird noch in der Klasse *OWLClassDescriptionEditorWithDLLearnerTab* die Funktion *dispose()* aufgerufen, die den View beendet.

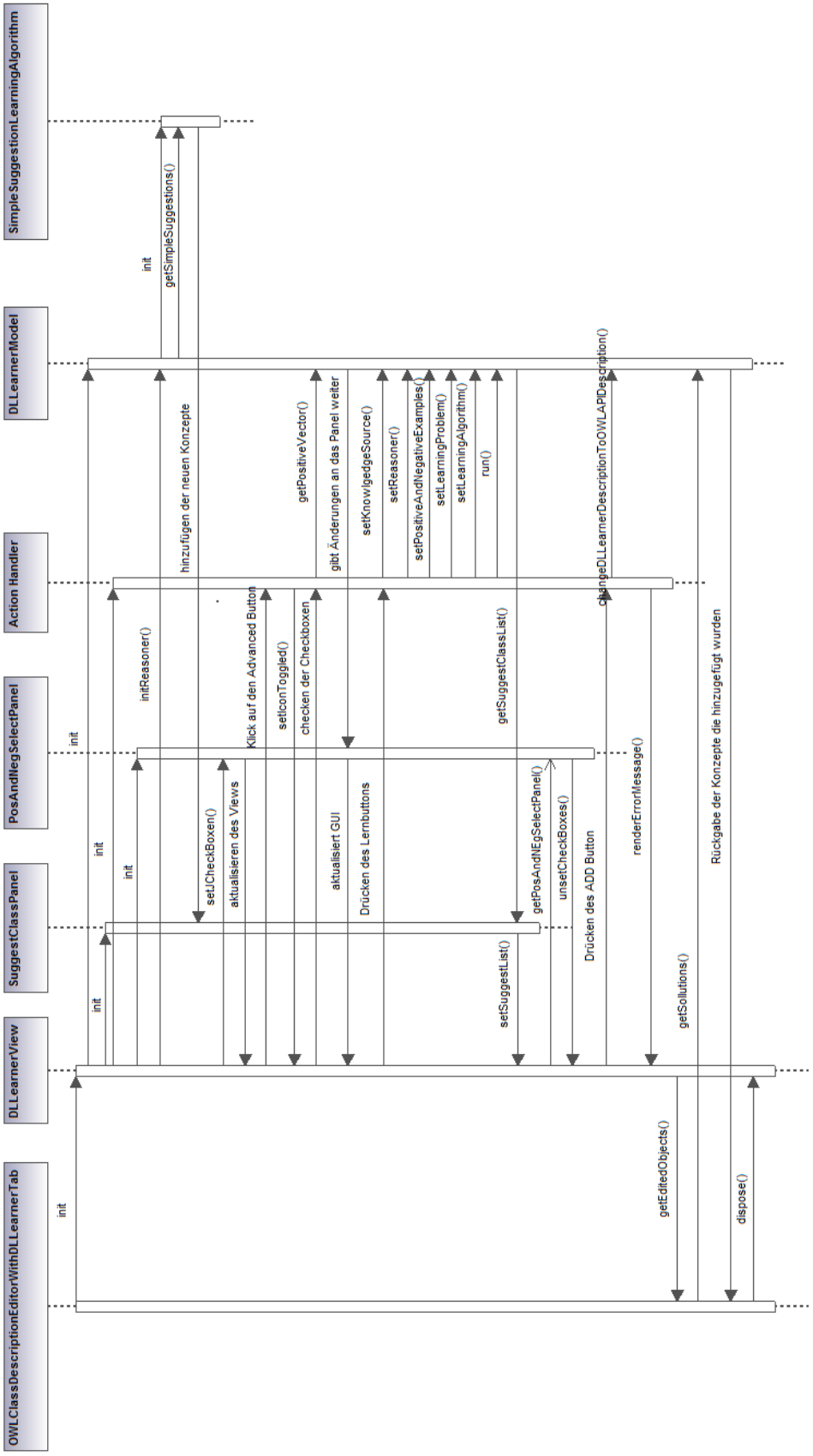

Abbildung 8: Sequenzdiagramm des Plugins

5.5 Die GUI

Die GUI ist sehr Einfach gehalten, wodurch sich der Nutzer auf unkomplizierte Weise neue Klassenkonzepte vorschlagen lassen kann (Abbildung 9). Die grafische Oberfläche ist ein Tab des Editors für Klassenbeschreibungen und lässt über einen Klick auf den Plus Button neben äquivalent - oder subclasses im Pluginview, im Bereich Classes in Protégé erreichen. Nachdem der Tab geöffnet ist sieht der Nutzer einige Vorschläge für neue Klassenkonzepte, die er in die Ontologie übernehmen kann, indem er ein Konzept auswählt und ADD auswählt. Außerdem hat er die Möglichkeit, sich über den Pfeil neben der Aufschrift Advanced, das Panel für die Selektion der positiven und negativen Beispiele anzeigen zu lassen (Abbildung 10). Ebenfalls befinden sich Hilfe Buttons neben der Aufschrift positive und negative Beispiele, bei denen dem Nutzer erklärt wird, wofür sie benötigt werden. Im Panel kann er durch Anklicken ein Beispiel zu den positiven oder negativen hinzufügen. Wenn der Nutzer damit fertig ist, kann er über den Button Suggest Class den Lernalgorithmus starten. Nach Ende des Lernvorgangs werden die neuen Konzepte im oberen Bereich angezeigt und der Nutzer hat jetzt die Wahl, eines dieser Vorschläge in die Ontologie einzufügen. Per Doppelklick erhält der Nutzer weitere Informationen über das vorgeschlagene Konzept, wie Genauigkeit oder abgedeckte positive und negative Beispiele (Abbildung 11). Falls es zu einem Fehler während des Lernvorganges kommen sollte, wird dies in einer Mitteilung unterhalb der Anzeige der neuen Konzepte eingeblendet, sodass der Nutzer sofort mitbekommt, dass der Lernvorgang fehlgeschlagen ist.

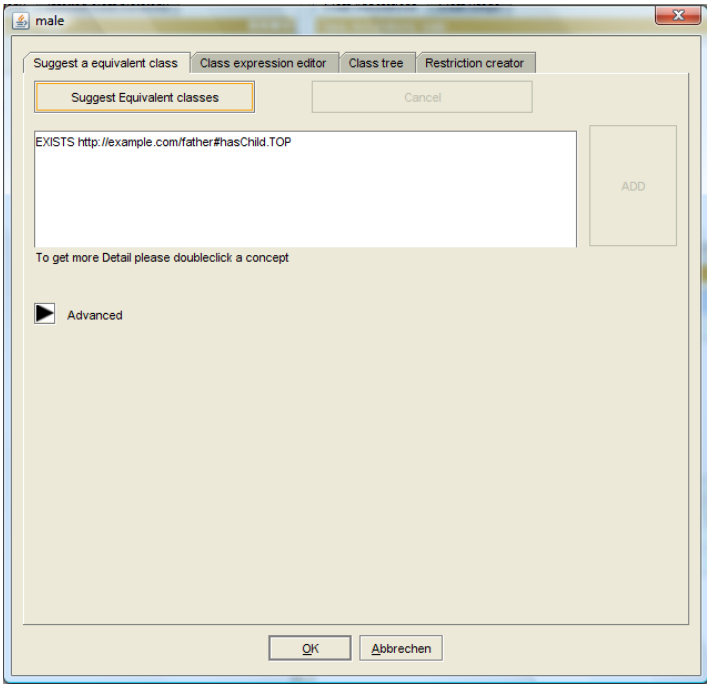

Abbildung 9: Benutzeroberfläche des Plugins ohne angezeigtes Beispielpanel

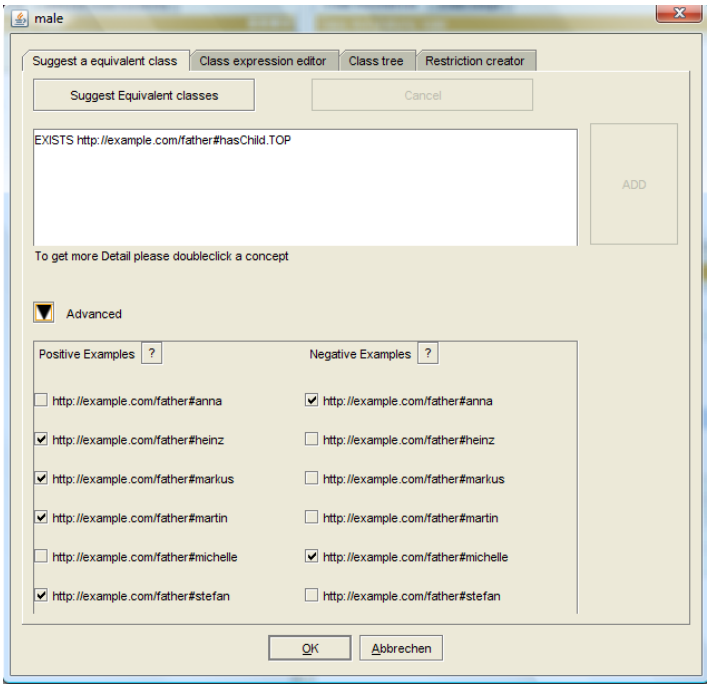

Abbildung 10: Benutzeroberfläche des Plugins mit angezeigtem Beispielpanel

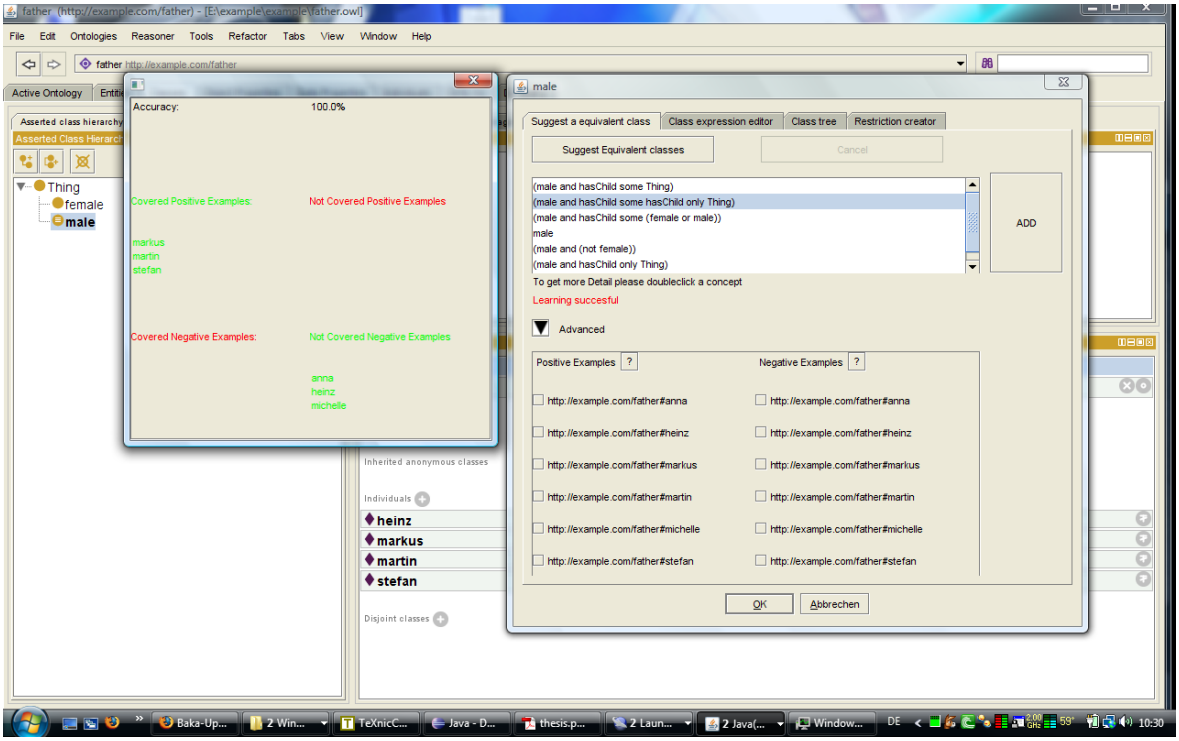

Abbildung 11: Benutzeroberfläche des Plugins mit Anzeige von mehr Details, zu einem vorgeschlagenen Konzept

6 Evaluation

In diesem Kapitel wird eine Evaluation der implementierten Funktionen durchgeführt, um darzustellen, ob diese bereits ausreichende Unterstützung für die Erstellung komplexer Klassenbeschreibungen bieten. Dabei werden mehrere Ontologien getestet, wie z.B. moviedatabase[13], um die Funktionalität zu testen und die Schwächen bei der Implementierung aufzudecken.

In der Ontologie moviedatabase wurden einige Individuen mit bestimmten Eigenschaften zu den Klassen *Movie*, *Director* und *Actor* hinzugefügt. Danach wurde versucht zu der Klasse *Director* eine neue äquivalente Klasse hinzuzufügen. Als positive Beispiele dienten alle Instanzen von *Director*. Dabei sind folgende Resultate entstanden.

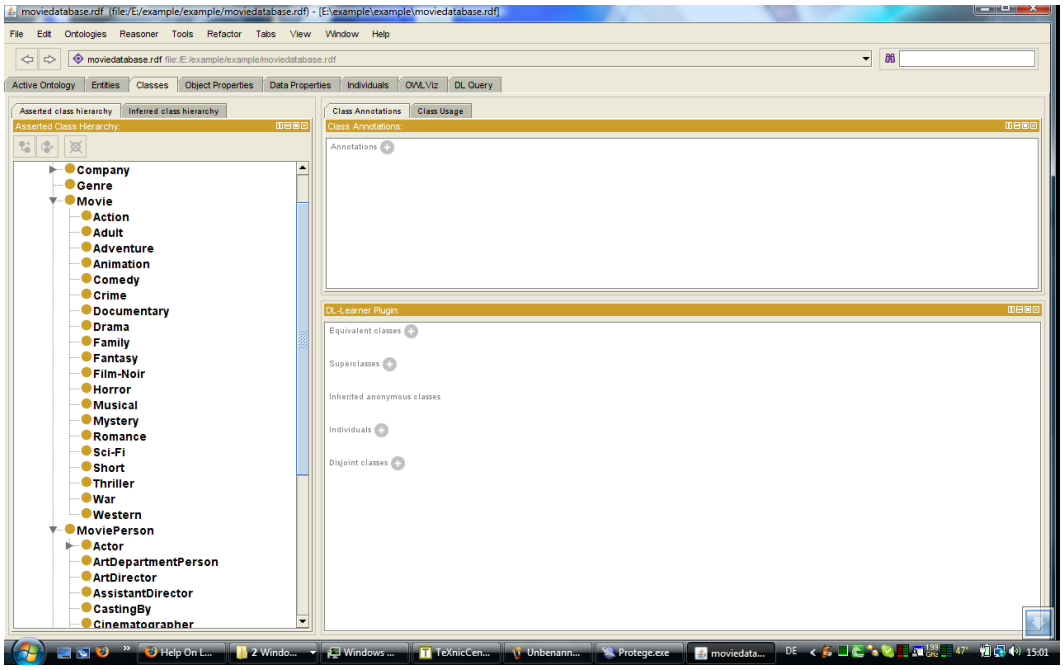

Abbildung 12: Hier wird die Klassenhierarchie der Ontologie moviedatabase dargestellt, welche für die Evaluation genutzt wurde.

- generates some Thing
- has_qualification some Thing
- Thing
- character only Thing

[13] http://139.91.183.30:9090/RDF/VRP/Examples/moviedatabase.rdf

- awards_won only Thing

Alle Ergebnisse dieses Lernvorganges sind zu ungenau. Es wurden nicht einmal die Eigenschaften *has_directionin* oder *produces_movie* bei der Erstellung der Klassenbeschreibungen berücksichtigt.

In einem weiteren Schritt wurde eine neue Klasse *Genre* mit den verschiedenen Genres als Instanzen hinzugefügt. Diese wurde als disjunkt zu *MoviePerson* und *Company* deklariert. Abschließend wurde noch zu jedem Individuum die Eigenschaft *has_movie* hinzugefügt. Es sollte eine neue Superklasse gelernt werden und als positive Beispiele wurden alle Genres ausgewählt. Da nach einer gewissen Zeit keine Klassenbeschreibungen angezeigt wurden, wurde der Lernvorgang abgebrochen. Dabei sind folgende Klassenbeschreibungen vorgeschlagen wurden.

- Thing
- has_directionin only DirectionIn
- has_directionin only Thing
- has_editingin only Thing
- has_genre only Thing
- has_makeupin only Thing

Keine der, trotz der abgebrochenen Lernvorganges, vorgeschlagenen Klassenbeschreibungen trägt zu der Verbesserung der Komplexität der Ontologie bei.

Da weitere Lernvorgänge auch keine besseren Ergebnisse lieferten, wurde der Versuch, neue Klassenbeschreibungen zu lernen, abgebrochen. Abschließend ist zu sagen, dass alle Vorgeschlagenen Klassenbeschreibungen zu ungenau waren und somit nicht zur Erweiterung der Ontologie dienten. Deshalb wurde eine zweite Ontologie betrachtet.

Im folgenden wird diese zweite Ontologie, nämlich food[14], betrachtet und ebenfalls versucht diese um komplexere Klassenbeschreibungen zu erweitern. Dabei wurden neue Individuen hinzugefügt, um bessere Ergebnisse bei dem Lernvorgang zu erhalten. Da in der Ausgangsontologie die Klasse *Fruit* keine Unterklasse von *ConsumableThing* oder *EdibleThing* war, wurde versucht eine neue Klassenbeschreibung zu lernen, die diese als *ConsumableThing* oder *EdibleThing* auszeichnet. Dazu wurden alle Instanzen der Klasse selbst als positive Beispiele deklariert, wobei einige neue Instanzen hinzugefügt wurden. Alle anderen Instanzen dienten als negative Beispiele. Es wurden folgende Klassenbeschreibungen vorgeschlagen.

[14] http://www.w3.org/TR/owl-guide/food.rdf

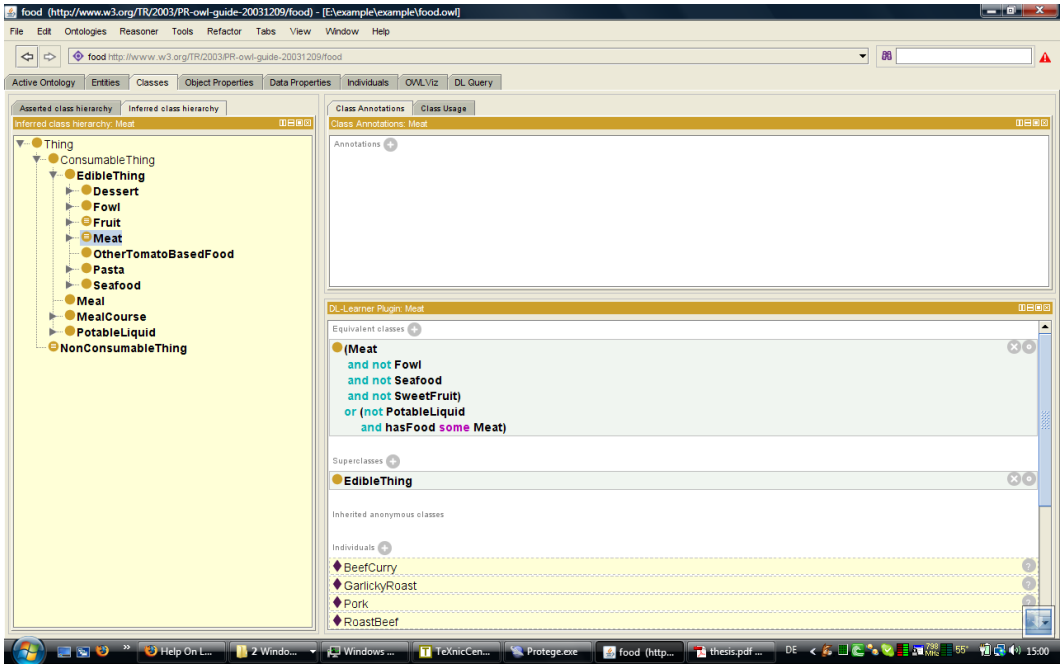

Abbildung 13: Hier wird die Klassenhierarchie der Ontologie food dargestellt, welche für die Evaluation genutzt wurde.

- (Fruit)
- (ConsumableThing and Fruit)
- (ConsumableThing and (not NonOysterShellfish))
- (ConsumableThing and (NonConsumableThing or (not NonOysterShellfish)))
- (ConsumableThing and ConsumableThing and (not NonOysterShellfish))
- (ConsumableThing and (not BlandFish))

Alle vorgeschlagenen Klassenbeschreibungen hatten eine Genauigkeit von 14 - 60%. Bei den ersten beiden Vorschlägen wurden nur die Instanzen, die zu der Klasse *Wine* gehörten nicht abgedeckt. Die anderen vier Beschreibungen hatten zu viele negative Beispiele abgedeckt und konnten nicht verwendet werden. Es wurde die Beschreibung *(ConsumableThing and Fruit)* eingefügt.

Nach dem Hinzufügen dieser Klassenbeschreibung wurde die Klasse *Fruit* als Unterklasse zu *ConsumableThing* deklariert. Es wurden nun weitere Instanzen mit der Eigenschaft *madeFromFruit* zu der Klasse *Juice* hinzugefügt. Nun wurde wieder ein Lernvorgang mit allen Instanzen dieser Klas-

se gestartet, um eine neue äquivalente Klasse zu lernen. Es wurden die folgenden Beschreibungen gelernt.

- (ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or Juice))
- (ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or PortableLiquid))
- (ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or (ConsumableThing and Juice)))
- (ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or(ConsumableThing and PortableLiquid)))
- (ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or (Juice and Juice)))
- (ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or (Juice and PortableLiquid)))

Alle 6 Beschreibungen beitzen eine Genauigkeit von 92% und sind somit gut geeignet die Klasse Juice zu erweitern. Es wurden nur einige negative Beispiele von *Fruit* abgedeckt die nicht hätten abgedeckt werden dürfen. Es kam auch hier die Eigenschaft *madeFromFruits* nicht in den Klassenbeschreibungen vor. Die Beschreibung *(ConsumableThing and (not Fowl) and (not Meat) and (not Pasta) and (not Seafood) and (Fruit or Juice))* wurde in die Ontologie eingefügt.

Als letztes wurde die fehlende Ontologie wine[15] hinzu geladen und eine neue Superklasse zu der Klasse *Region* gelernt. Dabei dienten alle Instanzen dieser Klasse als positive Beispiele. Es wurden folgenden Superklassen vorgeschlagen.

- (NonConsumableThing and (not NonConsumableThing))
- (NonConsumableThing and (not Wine))
- (NonConsumableThing and (not Wine))
- (NonConsumableThing)
- (NonConsumableThing and (not BlandFish))

[15] http://www.w3.org/TR/owl-guide/wine.rdf

- (NonConsumableThing and (not BlandFishCourse))

Dabei ist der erste Vorschlag *(NonConsumableThing and (not NonConsumableThing))* nicht sinnvoll, da *Region* nicht zu der Klasse *NonConsumableThing* und gleichzeitig zu dem Komplement dieser Klasse gehören kann. Der zweite Vorschlag eignet sich besser als Superklasse von *Region*, da diese Beschreibung eine geeignete Superklasse für *Region* darstellt. Dieser Vorschlag wurde zu der Ontologie hinzugefügt, obwohl er nur eine Genauigkeit von 64% besaß. Die letzten drei Beschreibungen boten nur eine Genauigkeit von 50% und kamen somit nicht in Frage, als Superklasse ausgewählt zu werden.

Abschließend ist zu sagen, dass bei dieser Ontologie wesentlich bessere Klassenbeschreibungen vorgeschlagen wurden, als bei moviedatabase. Beim Lernen wurden aber trotzdem einige Eigenschaften der positiven Beispiele nicht beachtet.

7 Verwandte Arbeiten

Jetzt wird ein Überblick darüber geben, was es in verwandten Bereichen noch für Forschungsarbeiten gibt. Dabei werden zuerst einige Arbeiten, die sich mit der Erstellung von Ontologien, insbesondere Tools zur Vereinfachung dieser, vorgestellt. Anschließend gibt es einen Überblick über andere Ontologie-Editoren, die solche Features in dieser oder ähnlicher Weise anbieten. Der letzte Teil widmet sich Arbeiten im Machine Learning Bereich, die sich mit dem Lernen von komplexen Klassenbeschreibungen befassen.

Ein Tool zur Visualisierung und Erkennung von Inkonsistenzen von Ontologien wurde in der Arbeit „Ontology-Engineering mit dem MatrixBrowser“[Cao03] vorgestellt. Dabei wurde ein auf F-Logik basierender Inferenzmechanismus implementiert, der implizite Relationen und Inkonsistenzen in Ontologien feststellen kann.

Ein anderes Tool zur Unterstützung bei der Erstellung von Ontologien ist KAON2[16], welches im Institut AIFB[17] der Universität Karlsruhe entwickelt wurde. Dabei handelt es sich um ein Tool für OWL-DL und SWRL[18] Ontologien. Es bietet Methoden zum Reasoning an und kann F-Logik Ontologien bearbeiten.

An dieser Stelle werden ein paar Ontologie-Editoren vorgestellt, bei denen sich eine Implementierung des DL-Learner lohnt oder bei denen es bereits ähnliche Funktionen zur Unterstützung bei der Ontologie-Erstellung gibt. Zunächst wird auf SWOOP eingegangen. Dieser Editor ist, wie Protégé in Java geschrieben und unterstützt bereits das Auffinden und Reparieren von Inkonsistenzen in Ontologien. Deshalb wäre es sinnvoll, ein Plugin zu entwickeln, wie das für Protégé , um eine bessere Unterstützung bei der Erstellung von Ontologien zu ermöglichen.

Ein weiterer Ontologie-Editor der sich zur Umsetzung eines DL-Learner Plugins eignen würde, wäre Ontoverse[19], denn bei diesem handelt es sich um eine Webanwendung, durch die viele Forschergruppen auf gemeinsame Wissensbasen zugreifen und diese bearbeiten können. Man könnte hier den DL-Learner als Werkzeug zur Erstellung komplexer und ausdrucksstärker Ontologien implementieren.
Der letzte Ontologie-Editor der noch vorgestellt wird, ist Chimaera[20][McG00]. Da dieser häufig dazu verwendet wird Fragmente von Wissensbasen zu vereinigen, könnte man hier ebenfalls den DL-Learner integrieren, um bei der Vereinigung, neue eventuell bessere Klassenbeschreibungen vorschlagen zu lassen und die Ausdrucksstärke der vereinigten Ontologien zu erhöhen.

[16] Quelle:http://kaon2.semanticweb.org/
[17] Institut für Angewandte Informatik und Formale Beschreibungsverfahren
[18] Semantic Web Rule Language
[19] Quelle:http://www.ontoverse.org/
[20] Quelle:http://www.ksl.stanford.edu/software/chimaera/

Im letzten Teil werden weitere Ansätze zum Lernen von komplexen Klassenbeschreibungen im Machine Learning Bereich beschrieben. Ein Werkzeug zur Erstellung komplexer Klassenstrukturen im Bereich des Machine Learning ist YINYANG[IP05]. Dabei handelt es sich um ein ebenfalls in Java geschriebenes Tool, welches überwachtes Konzeptlernen durchführt. Er kann neben dem Vorschlagen von komplexen Klassenbeschreibungen auch eine Konsistenzprüfung durchführen, um zu testen, ob alle positiven und keine negativen Beispiele abgedeckt wurden.

Ein anderes Tool ist TextToOnto[MS00], welches Konzepte aus Textbeschreibungen extrahieren kann. Dabei handelt es sich um einen semi-automatischen Vorgang, bei dem erst die Konzepte, dann die Beziehungen zwischen den Konzepten und zum Schluss die Ontologie selbst extrahiert wird. Der Nutzer kann nach jedem Schritt eingreifen und Verbesserungen oder Korrekturen vornehmen.

8 Zusammenfassung und Ausblick

Zusammenfassend ist zu sagen, dass mit diesem Plugin eine erste Hilfe bei der Erstellung von ausdrucksstarken Ontologien entstanden ist. Dabei wurde es in Protégé als neuer Tab des Klassenbeschreibungseditors implementiert und bietet mit der sehr einfach gestalteten grafischen Oberfläche eine bessere Unterstützung bei der Erstellung ausdrucksstarker Ontologien. Es wurde ein einfacher Algorithmus implementiert, der vor dem eigentlichen Lernvorgang ein paar Vorschläge zur Erweiterung der Klassenbeschreibungen gibt. Weiterhin wurde eine Kommunikationsmöglichkeit mit dem DL-Learner implementiert, die es ermöglicht, die in Protégé geladene Ontologie an den DL-Learner zu übergeben, um damit neue Klassenbeschreibungen zu lernen und komplexe Klassen zu erstellen. Dabei greift das Plugin auf OWLAPI zurück, wodurch alle Änderungen sofort übernommen werden, um unmittelbar damit weiter arbeiten zu können.

Der Algorithmus für die Vorschläge zu Klassenbeschreibungen, der vor dem eigentlichen Lernalgorithmus ausgeführt wird, ist dabei noch sehr rudimentär implementiert und sollte deshalb eine Überarbeitung erfahren. Ebenfalls sollte eine Verbesserung der Integration des Plugins in Protégé erfolgen, da zur Zeit die erforderlichen Klassen aus Protégérepository kopiert und angepasst werden mussten. Dadurch kann es zu Problemen mit neuen Versionen kommen, da diese eventuell Änderungen erfahren haben, die mit den alten Versionen nicht mehr kompatibel sind.

Da es sich bei dieser Arbeit um eine erste Version zur unterstützenden Erstellung ausdrucksstarker Ontologien handelt, wäre eine Implementierung weiterer Funktionen sinnvoll. Dabei könnte man eine Erweiterung um das Reparieren von Ontologien, wie es in der Arbeit von Lorenz Bühmann beschrieben ist, realisieren, die es dem Nutzer ermöglicht, Fehler bei der Erstellung der Ontologie zu finden und zu beheben.

9 Literaturverzeichnis

Literatur

[Cao03] Xin Cao. Ontology-engineering mit dem matrixbrowser. Master's thesis, Universität Stuttgart Fakultät Informatik, 2003.

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web Grundlagen*. Springer Verlag, 2008.

[IP05] Luigi Iannone and Ignazio Palmisano. An algorithm based on counterfactuals for concept learning in the semantic web. In Moonis Ali and Floriana Esposito, editors, *IEA/AIE*, volume 3533 of *Lecture Notes in Computer Science*, pages 370–379. Springer, 2005.

[LH08a] Jens Lehmann and Pascal Hitzler. Foundations of refinement operators for description logics. In Hendrick Blockeel, Jude W. Shavlik, and Prasad Tadepalli, editors, *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP)*, volume 4894 of *Lecture Notes in Computer Science*, pages 161–174. Springer, 2008.

[LH08b] Jens Lehmann and Pascal Hitzler. A refinement operator based learning algorithm for the alc description logic. In Hendrick Blockeel, Jude W. Shavlik, and Prasad Tadepalli, editors, *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP)*, volume 4894 of *Lecture Notes in Computer Science*, pages 147–160. Springer, 2008.

[McG00] Deborah L. McGuinness. Conceptual modeling for distributed ontology environments. In *International Conference on Conceptual Structures*, pages 100–112, 2000.

[MS00] A. Maedche and S. Staab. Semi-automatic Engineering of Ontologies from Text. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering*, 2000.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

10 Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, den Christian Kötteritzsch