UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik

Reparatur und Erweiterung von OWL Ontologien

Masterarbeit

Leipzig, 05.01.2010

Betreuer:
Dipl. Inf. Jens Lehmann

vorgelegt von

Lorenz Bühmann
geb. am: 15.11.1981

Studiengang Informatik

Zusammenfassung

Ontologien sind in der Informatik bereits seit vielen Jahren ein Werkzeug zur Modellierung und Strukturierung von Wissen in unterschiedlichen Domänen. Das steigende Interesse am Semantic Web in den letzten Jahren resultiert in einer immer größer werdenden Anzahl an Ontologien. Der Erstellungs- und Wartungsprozess bei Ontologien kann vor allem in ausdrucksmächtigen Beschreibungssprachen wie OWL (Web Ontology Language) selbst für Experten in der modellierten Domäne schwierig sein. Das kann mitunter zu logischen Fehlern bei der Modellierung führen, oder aber es wird aus Angst oder Unkenntnis auf die ausdrucksstärkeren Konstrukte in OWL verzichtet. Weil aber gerade ausdrucksstarke Ontologien viele Vorteile wie komplexe Anfragemöglichkeiten oder Schließen von impliziten Wissen mit sich bringen, ist hier ein Bedarf ein Werkzeugen vorhanden, um den Ersteller bzw. Nutzer von Ontologien dabei zu unterstützen. Ziel dieser Arbeit ist die Entwicklung eines Tools, welches Fehler unterschiedlicher Art erkennt und geeignete Reparaturvorschläge machen kann.

Inhaltsverzeichnis

1 Einleitung

Ontologien sind in der Informatik bereits seit vielen Jahren ein Werkzeug zur Modellierung und Strukturierung von Wissen in unterschiedlichen Domänen. War der Einsatz von Ontologien in den Anfängen vor allem in den Bereichen der Lebenswissenschaften, wie z.B. Medizin oder Biologie zu finden, so werden sie seit der Entstehung des Semantic Web auch in anderen Domänen stärker verwendet. So werden Ontologien heutzutage u.a. auch in Bereichen wie Software-Engineering, Wikis oder einer Vielzahl von Bereichen in Industrie und Wirtschaft eingesetzt. Das steigende Interesse am Semantic Web in den letzten Jahren resultiert in einer immer größer werdenden Anzahl an Ontologien. Der Erstellungs- und Wartungsprozess bei Ontologien kann vor allem in ausdrucksmächtigen Beschreibungssprachen wie OWL (Web Ontology Language) selbst für Experten in der modellierten Domäne schwierig sein. Das kann mitunter zu logischen Fehlern bei der Modellierung führen, oder aber es wird aus Angst oder Unkenntnis auf die ausdrucksstärkeren Konstrukte in OWL verzichtet. Weil aber gerade ausdrucksstarke Ontologien viele Vorteile wie komplexe Anfragen oder Schließen von impliziten Wissen mit sich bringen, ist hier ein Bedarf ein Werkzeugen vorhanden, um den Ersteller bzw. Anwender von Ontologien dabei zu unterstützen.

Ziel dieser Arbeit ist die Entwicklung eines Tools, welches Fehler unterschiedlicher Art erkennt und geeignete Reparaturvorschläge machen kann. Dabei werden wir die vorhandenen Methoden zur Evaluation von Ontologien untersuchen, und darauf aufbauend die (optimierten) Techniken zusammen mit neuen Techniken aus dem Bereich des Ontology Learning in eine möglichst benutzerfreundlichen Anwendung integrieren. Für den Bereich Ontology Learning werden wir dabei auf das DL-Learner Framework aufsetzen und die Algorithmen zum Lernen von Klassenbeschreibungen zur Verbesserung/Erstellung des Ontologie-Schemas verwenden.

Ausblick

Unsere Arbeit ist folgendermaßen aufgebaut: In Kapitel 2 werden wir einen Überblick über die Grundlagen dieser Arbeit geben, und dabei sowohl auf technologische und theoretische Grundlagen des Semantic Web, als auch auf den DL-Lerner als Basis unseres Tools eingehen. Kapitel 3 behandelt dann eine Beschreibung von möglichen Fehlern und Problemen in Ontologien. Dabei werden wir neben den Ursachen für Fehler auch verschiedene Einteilungen nach Fehlerarten erläutern. In Kapitel 4 werden wir uns mit einem der beiden großen Teile unseres entwickelten Tools, der Reparatur logischer Fehler auseinandersetzen. Dabei geht es in diesem Abschnitt vor allem um das Finden, Erklären und die Reparaturunterstützung. Im Kapitel 5 beschreiben wir den zweiten Hauptteil unseres Tools, der Erweiterung von Ontologien. Wir werden dort zunächst einige Grundlagen zum Lernen von Klassenbeschreibungen in OWL, sowie eine Kurzfassung

des verwendeten Lernalgorithmus darstellen. Außerdem werden wir dort eine Menge von Regeln vorstellen, die im Anschluß an eine Erweiterung manchmal sinnvoll sein können. Im Kapitel 6 beschreiben wir die Implementierungsdetails unseres Tools, dabei insbesondere den Aufbau, Ablauf und das User Interface. Für die Demonstration unseres Tools, werden wir in Kapitel 7 die Fähigkeiten an Beispielen vorführen. Abschließend werden wir in Kapitel 8 die wesentlichen Punkte unserer Arbeit zusammenfassen und einen Ausblick auf zukünftige Weiterentwicklungen geben.

2 Grundlagen

In diesem Kapitel beschäftigen wir uns mit den wesentlichen Grundlagen für unsere Arbeit. Wir werden dafür zunächst die Idee des Semantic Web erklären, und danach auf den Begriff, die Arten und Verwendung von Ontologien eingehen. Im Anschluß daran behandeln wir die formalen Grundlagen zur Repräsentation von Ontologien, und gehen dabei im Speziellen auf Beschreibungslogiken und die darauf basierende Sprache OWL ein. Abschließend werden wir den DL-Learner als Basis unseres entwickelten Tools beschreiben, dabei vor allem Aufbau und den Lernprozess näher eingehen.

2.1 Semantic Web

Das Semantic Web ist eine immer stärker verbreitete Erweiterung des World Wide Web (WWW), in der den Informationen des WWW eine genaue, maschinenverständliche Bedeutung (Semantik) gegeben wird. Dies ermöglicht neben einer automatischen Verarbeitung solcher Daten durch Maschinen wie Computer, auch einen leichteren Austausch, sowie bessere Integrationsmöglichkeiten. Durch den Einsatz von standardisierten Sprachen und Technologien, welche vom World Wide Web Consortium (W3C) als Recommendations veröffentlicht und verwaltet werden, ist es möglich die Daten explizit in einheitlichen Formaten zu beschreiben. Diese (teilweise) standardisierten Sprachen und Technologien, welche die Basis für das Semantic Web bilden, werden gemeinsam in einer Schichtenarchitektur (Abb. 1) organisiert. Wir werden einige der Schichten hier kurz beschreiben, verweisen jedoch für ausführliche Informationen auf die jeweiligen W3C Recommendations.

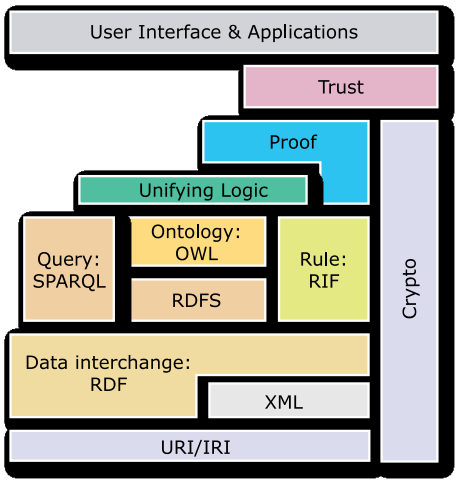


Abb. 1: Semantic Web Layer Cake (2009)[1]. Er stellt die einzelnen Ebenen des Semantic Web dar.

URI/IRI URIs[2] und IRIs dienen zur (weltweit) eindeutigen Bezeichnung von Ressourcen. Eine Ressource kann dabei jedes Objekt sein, was im Kontext der jeweiligen Anwendung ein klare Identität besitzt (z.B. Bücher, Orte, Menschen, Beziehungen zwischen diesen Dingen, abstrakte Konzepte usw.).

XML Die XML Technologiefamilie bildet das syntaktische Grundgerüst des Semantic Web und besteht u.a. XML[3], XML-Schema[4], Namespaces[5] zur Datenspeicherung, sowie zum Beispiel XPath[6] für Anfragen darauf. Es handelt sich dabei vereinfacht gesagt um einen Standard, mit dem es möglich ist semi-strukturierte Daten zu erstellen und auszutauschen. Allerdings fehlt XML ein formale Semantik, so dass diese Daten i.A. nicht von Maschinen „verstanden“ werden können.

RDF/RDFS RDF (Resource Description Framework) ist seit 2004 W3C Standard und definiert ein Datenmodell zur Beschreibung maschinenverarbeitbarer Semantik von Daten. Mit RDF können Informationen über einzelne Ressourcen formal beschrieben werden. Zur Beschreibung der Informationen werden dabei Aussagen in Form von Tripeln gemacht, wobei ein Tripel aus Subjekt, Prädikat und Objekt besteht:

Subjekt Subjekte sind die Ressourcen, über die Aussagen getroffen werden.

Prädikat Prädikate können als Attribute bzw. Eigenschaften von Subjekten angesehen werden.

Objekt Objekte sind die Wertzuweisungen der jeweiligen Eigenschaften. Das können zum einen andere Ressourcen sein, oder aber es handelt sich um Literale. Literale beschreiben Datenwerte, die im Gegensatz zu den Ressourcen keine separate Existenz besitzen.

Das RDF-Datenmodell basiert auf gerichteten Graphen, so dass Tripel bzw. Mengen von Tripeln als Graphen darstellbar sind.

Beispiel 1 (RDF als Graph)**.** Wenn man RDF als Graphen darstellt, werden Ressourcen üblicherweise durch Ellipsen und Literale durch Rechtecke visualisiert. In Abb. 2 werden zwei Aussagen über die Ressource tom gemacht. Die erste Aussage ist dass tom einen Bruder hat, und zwar tim. Tim ist dabei eine weitere Ressource. Die zweite Aussage wird über den Familiennamen von tom gemacht. So hat tom in diesem Beispiel den Familiennamen $Schulz$, welcher hier in Form eines Literals zugewiesen wurde.

[1]Quelle: `http://www.w3.org/2007/03/layerCake.png`
[2]Informationen über URI Standard: `http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/`
[3]XML: `http://www.w3.org/TR/2006/REC-xml11-20060816`
[4]XML-Schema: `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`
[5]Namespace: `http://www.w3.org/TR/2009/REC-xml-names-20091208/`
[6]XPath: `http://www.w3.org/TR/2007/REC-xpath20-20070123/`
[6]RDF: `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`

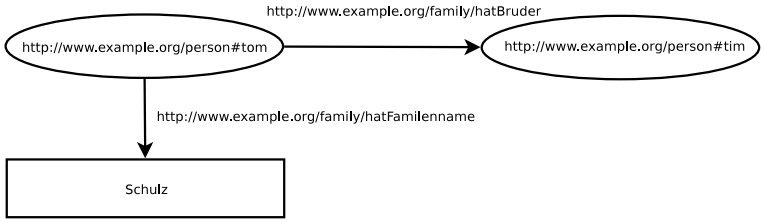


Abb. 2: Beispiel für graphische Darstellung von RDF Tripeln.

Die Darstellung als Graph dient eher zur Veranschaulichung, und hat in der Praxis kaum Relevanz. Die am weitesten verbreitete Form zur Darstellung und Serialisierung ist das Format RDF/XML. Die Grundlage bildet dabei die Syntax von XML, was auch dafür sorgt dass dieses Format die beste Tool-Unterstützung hat.

Beispiel 2 (RDF im RDF/XML Format)**.** Das Beispiel 1 sieht in RDF/XML folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:fam="http://example.org/family/">
  <rdf:Description rdf:about="http://example.org/person#tom">
    <fam:hatBruder rdf:resource="http://example.org/person#tim"</>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/person#tom">
    <fam:hatFamilienname>Schulz</fam:hatFamilienname>
  </rdf:Description>
</rdf:RDF>
```

Neben dieser Notation gibt es noch weitere wie N3 oder Turtle, welche die Lesbarkeit vereinfachen.

Das RDF-Daten-Modell bietet keine Möglichkeit Eigenschaften zu beschreiben und Beziehungen zwischen Eigenschaften und Ressourcen zu definieren. Um diese Beziehungen zu deklarieren wird RDF-Schema (RDFS), eine vom W3C spezifizierte Erweiterung von RDF, benutzt. Mit RDFS ist es möglich Aussagen über generische Mengen von Individuen (Klassen), z.B. $Autoren, Personen, Tiere,$ zu machen. Außerdem kann man Aussagen über logische Zusammenhänge zwischen Individuen, Klassen und Relationen machen. So kann man zum Beispiel ausdrücken „Autoren sind Personen“ oder „nur Personen schreiben Bücher“. Mit RDFS ist es also möglich schematisches Wissen zu spezifizieren, so dass RDFS auch als leichtgewichtige Ontologiesprache bezeichnet wird.

Allerdings ist die Ausdrucksmächtigkeit von RDF/RDFS beschränkt, denn man kann u.a. keine komplexen Klassenbeschreibungen (z.B. eine Klasse, die Vereinigung von zwei anderen ist)

definieren. Eine Sprache die dazu in der Lage ist, ist OWL. Sie wird von uns in Abschnitt 2.5 ausführlicher behandelt.

SPARQL Um auf den (großen) Mengen von RDF Tripeln (meist in sogenannten Triple-Stores verwaltet) arbeiten zu können, gibt es die Anfragesprache SPARQL, welche seit 2008 W3C Standard ist. Es handelt sich dabei um eine graph-basierte Anfragesprache, welche eine ähnliche Syntax wie SQL aus dem Bereich der relationalen Datenbanken besitzt.

Beispiel 3 (SPARQL Query)**.** Eine SPARQL Anfrage für die Namen aller Hauptstädte von Europa ist zum Beispiel:

```
PREFIX ex: <http://example.org/ontology#>
SELECT ?hauptstadt
WHERE {
  ?x ex:Name ?hauptstadt.
  ?x ex:istHautpstadtVon ?y.
  ?y ex:liegtAufKontinent ex:Europa.
}
```

Variablen sind dabei durch ein vorangestelltes ? gekennzeichnet. Hier werden also alle Variablenbelegungen von `?hauptstadt` zurückgegeben, die auf das Muster dieser 3 Tripel passen

Anfragen in SPARQL an Triple-Stores können zum Beispiel über sogenannte SPARQL-Endpoints[7] erfolgen. Bei SPARQL-Endpoints handelt es sich um RDF-Datenquellen die über SPARQL angesprochen werden.

2.2 Ontologien

Der Begriff Ontologie stammt ursprünglich aus der Philosophie, und bezeichnet dort eine eigene Disziplin, die sich mit der „Lehre des Seins“ befasst. In der Informatik hat der Begriff eine andere Bedeutung. Hier wird eine Ontologie als ein formales Wissensmodell angesehen, das für die Bereitstellung von Wissensstrukturen, zum Wissensaustausch und als Basis der automatischen Wissensverarbeitung verwendet wird. Verwendung finden Ontologien vor allem in den Bereichen Informationsintegration, Wissensmanagement, Expertensysteme und zunehmend im Semantic Web. Es gibt viele Definitionen für den Begriff Ontologie, eine sehr verbreitete von T.Gruber lautet:

„An ontology is an explicit specification of a conceptualization.“[6]

[7] Eine Liste aktuell erreichbarer SPARQL-Endpoints: `http://esw.w3.org/topic/SparqlEndpoints`

Konzeptualisierung steht hier für die Erstellung eines Modells einer Domäne, und explizit bedeutet, dass die dabei erstellten Konzepte und deren Eigenschaften eindeutig definiert sind. Eine spätere und darauf aufbauende Definition lautet:

> „An ontology is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon. Explicit means that the types of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group.“ [37]

In dieser Definition wird zusätzlich zu der vorherigen verlangt, dass eine formale Definition verwendet wird, damit die Ontologie maschinen-lesbar ist. Außerdem wird hier von einem gemeinsamen Wissen ausgegangen, was bedeutet dass dieses Wissen nicht nur auf eine einzelne Person ausgerichtet ist, sondern von mehreren Menschen akzeptiert wird.

Ontologien können je nach Formalisierungsgrad unterschiedlich repräsentiert werden:

informal natürlichsprachliche Beschreibung

semi-informal Beschreibung in eingeschränkter und strukturierten Form einer natürlichen Sprache

semi-formal Beschreibung in einer künstlichen und formal definierten Sprache

formal Beschreibung mit äußerst genau definierten Begriffen mit formaler Semantik in einer vollständigen und korrekten Sprache

Im Allgemeinen bestehen Ontologien aus Konzepten, Relationen, Objekten und Axiomen:

Konzepte Konzepte stehen für eine Menge von Objekten, die gleiche Eigenschaften besitzen.

Relationen Relationen repräsentieren Beziehungen zwischen Konzepten einer Domäne.

Objekte Objekte werden verwendet, um Individuen einer Domäne zu repräsentieren.

formale Axiome Sie dienen zur Modellierung von Sätzen, die immer wahr sind.

Im Normalfall werden Ontologien in Form von speziellen Sprachen repräsentiert. In den Anfängen geschah dies in Form Semantischen Netzen oder Frames, später dann mit Hilfe von Beschreibungslogiken, deren Semantik mit Hilfe der mathematischen Logik eindeutig und vollständig spezifiziert ist. Heutzutage sind es in der Regel weitgehend maschinenverarbeitbare Sprachen, die

durch das W3C standardisiert wurden. Dazu gehören vor allem das Resource Description Framework (RDF) und RDF-Schema für eher einfache Ontologien, sowie die Web Ontology Language (OWL) für komplexere Ontologien. Darüber hinaus gibt es noch Regelbasierte Sprachen, wie die Semantic Web Rule Language (SWRL), die Teile von OWL mit Regeln kombiniert oder auch F-Logic, deren Semantik auf der Semantik der Logikprogrammierung basiert.

2.3 Beschreibungslogiken

Beschreibungslogiken (engl. Description Logics (DL)) sind eine Familie von Sprachen zur Wissensrepräsentation. Die meisten Beschreibungslogiken sind eine Teilmenge der Prädikatenlogik erster Stufe (engl. First Order Predicate Logic (FOL)), im Gegensatz zu dieser aber entscheidbar. Dies ermöglicht über eine Beschreibungslogik zu schließen, d.h. aus vorhandenem Wissen neues Wissen zu gewinnen. DLs unterscheiden sich üblicherweise hinsichtlich ihrer Verwendung von Konstruktoren zur Beschreibung von komplexeren Konzepten und Rollen aus atomaren, sowie den verfügbaren Axiomen um Fakten über Konzepte, Rollen und Individuen auszudrücken.

Die Syntax einer Beschreibungslogik ist gegeben durch ein Vokabular und eine Menge von Konstruktoren. Ein Vokabular (oder Signatur) ist dabei die (disjunkte) Vereinigung von Mengen von

- atomaren Konzepten (N_C) : sie entsprechen 1-stelligen Prädikaten in der FOL und definieren eine Menge bzw. Klasse von Objekten, z.B. $Vogel(x)$, $Frau(x)$
- atomaren (abstrakten) Rollen (N_R): sie entsprechen 2-stelligen Prädikaten in der FOL und definieren Relationen zwischen Objekten, z.B. $hatKind(x, y)$, $kennt(x, y)$
- Individuen (N_I): sie entsprechen Konstanten in der FOL, z.B. $Tweety$, $Tina$

Bei manchen DLs gibt es zusätzlich noch konkrete Rollen. Dabei handelt es sich um Rollen, mit denen man Individuen mit konkreten Werten aus unterschiedlichen Datentypen (u.a. Integer, String) in Relation setzen kann. So könnte man damit z.B. das Alter einer Person auf einen festen Wert setzen.

Jede Beschreibungslogik bietet eine Menge von Konstruktoren an (Tab. 1), mit denen es möglich ist komplexe Konzepte oder Rollen aus atomaren induktiv zu definieren (beschreiben). Um die Zum Beispiel kann man ausgehend von den atomaren Konzepten $Person$ und $weiblich$ unter Verwendung des $\sqcap$-Konstruktors das komplexe Konzept „alle Personen die weiblich sind“ durch $Person \sqcap weiblich$ beschreiben. Mit der zusätzlichen Nutzung der Existenzrestriktion ($\exists r.C$) könnte man hingegen z.B. durch $Person \sqcap \exists hatKind.\top$ die Menge „aller Personen, die ein Kind haben“, beschreiben.

Um den Audrücken von Beschreibungslogiken auch eine Bedeutung zu geben, wird wie in der Modelltheoretischen Semantik nach Tarski üblich eine Interpretation verwendet. Eine Interpretation

Name	**Syntax**	**Semantik**
atomares Konzept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
atomare Rolle	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Individuum	a	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
allgemeinstes Konzept (top)	$\top$	$\Delta^{\mathcal{I}}$
speziellstes Konzept (bottom)	$\bot$	$\emptyset$
Nominal	$\{a\}$	$\{a^I\}$
Schnitt	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Vereinigung	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
self Konzept	$\exists r.\texttt{Self}$	$\{a \in \Delta^{\mathcal{I}} \mid (a,a) \in r^{\mathcal{I}}\}$
existienzielle Restriktion	$\exists r.C$	$\{a \mid \exists b.(a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
universelle Restriktion	$\forall r.C$	$\{a \mid \forall b.(a,b) \in r^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
max. Kardinalitätsrestriktion	$\leq n\ r.C$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \leq n\}$
min. Kardinalitätsrestriktion	$\geq n\ r.C$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \geq n\}$
Inverse Rolle	r^-	$\{(a,b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (b,a) \in r^{\mathcal{I}}\}$

Tabelle 1: $\mathcal{SROIQ}$ Syntax und Semantik von Konstruktoren für Rollen und Konzepte.

$\mathcal{I}$ ist ein Paar mit $\mathcal{I} = \left(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\right)$, wobei $\Delta^{\mathcal{I}}$ eine nicht-leere Menge ist, genannt die Domäne der Interpretation, und $\cdot^{\mathcal{I}}$ eine Interpretationsfunktion ist, die jedem atomaren Konzept A eine Teilmenge $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, jeder atomaren Rolle R eine binäre Relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, und jedem Individum a ein Element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ zuordnet. Erweitert auf komplexe Konzeptbeschreibungen wird die Interpretationsfunktion induktiv wie in Tab. 1 abgebildet.

Wissensbasen basierend auf Beschreibungslogiken werden üblicherweise in eine TBox und ABox unterteilt. Die TBox enthält dabei das sogenannte terminologische Wissen, d.h. in ihr enthalten sind Axiome, die die Struktur der zu modellierenden Domäne (häufig auch als konzeptionelles Schema bezeichnet) beschreiben. Die Axiome einer TBox haben üblicherweise die Form $C \sqsubseteq D$ (Konzeptinklusion) oder $C \equiv D$ (Konzeptäquivalenz), wobei C, D (möglichweise komplexe) Konzepte sind. Damit lassen sich zum Beispiel Aussagen wie $Student \sqsubseteq Mensch$ machen, was in diesem Fall ausdrücken soll, dass jeder $Student$ ein $Mensch$ ist. Im Gegensatz dazu stehen in der ABox ausschließlich Axiome die konkrete Situationen (Daten) der Welt beschreiben, d.h. Aussagen über die Individuen und ihre Beziehungen untereinander machen. Für gewöhnlich handelt es sich dabei zum einen um Konzeptzuweisungen $C\,(a)$, die ausdrücken sollen, dass ein Indivduum a zu einem Konzept C (der Interpretation von C) gehört, und zum anderen um Rollenzuweisungen $R\,(a,b)$, die zusichern, dass ein Individuum a zu b in einer Relation R steht. Sind zum Beispiel $tweety, anne$ und tom Individuen, dann bedeutet $Vogel\,(tweety)$, dass $tweety$ ein $Vogel$ ist, und $hatKind\,(anne, tom)$, dass tom ein Kind von $anne$ ist. Manchmal wird zusätzlich zur TBox und

ABox noch eine RBox eingeführt. In ihr stehen Axiome die sich auf Rollen beziehen, wie z.B. Transitivität ($Trans(r)$) oder Rolleninklusion ($R \sqsubseteq S$) von (komplexen) Rollen R und S. Wir verzichten hier allerdings auf eine RBox, so dass diese Axiome mit in der TBox stehen.

Um eine Unterscheidung nach den vorhandenen Konstruktoren zu erleichtern, wird in der DL-Community ein Namensschema aus Abkürzungen verwendet. So ist zum Beispiel $\mathcal{ALC}$ die kleinste, aussagenlogisch abgeschlossene Beschreibungslogik und $\mathcal{S}$ steht für $\mathcal{ALC}$ mit transitiven Rollen. Weitere Abkürzungen stehen in [2].

Die Erfüllbarkeitsrelation $\mathcal{I} \models \alpha$ zwischen einer Interpretation $\mathcal{I}$ und einem DL Axiom α (gelesen als „$\mathcal{I}$ erfüllt α“, oder „$\mathcal{I}$ ist ein Modell von α“) wird erwartungsgemäß wie folgt definiert:

$$\begin{aligned}
\mathcal{I} &\models C \sqsubseteq D \text{ gdw. } C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \\
\mathcal{I} &\models C \equiv D \text{ gdw. } C^{\mathcal{I}} = D^{\mathcal{I}} \\
\mathcal{I} &\models R \sqsubseteq S \text{ gdw. } R^{\mathcal{I}} \subseteq S^{\mathcal{I}} \\
\mathcal{I} &\models R \equiv S \text{ gdw. } R^{\mathcal{I}} = S^{\mathcal{I}} \\
\mathcal{I} &\models C(a) \quad \text{gdw. } a^{\mathcal{I}} \in C^{\mathcal{I}} \\
\mathcal{I} &\models R(a,b) \text{ gdw. } \left(a^{\mathcal{I}}, b^{\mathcal{I}}\right) \in R^{\mathcal{I}}
\end{aligned}$$

Eine Interpretation $\mathcal{I}$ ist ein Modell einer Ontologie $\mathcal{O}$, wenn $\mathcal{I}$ alle Axiome in $\mathcal{O}$ erfüllt. Eine Ontologie $\mathcal{O}$ impliziert ein Axiom α (geschrieben $\mathcal{O} \models \alpha$), wenn $\mathcal{I} \models \alpha$ für jedes Modell $\mathcal{I}$ von $\mathcal{O}$. Ein Axiom α ist eine Tautologie wenn jede beliebige Interpretation $\mathcal{I}$ ein Modell von α ist (oder anders formuliert α von der leeren Ontologie impliziert wird). Mit der Signatur einer Ontologie $\mathcal{O}$ (von einem Axiom α) bezeichnen wir wie in [5] die Menge $Sig(\mathcal{O})(Sig(\alpha))$ von atomaren Konzepten, atomaren Rollen und Individuuen die in $\mathcal{O}$ (respektive α) vorkommen. Seien $\mathbf{S}$ und $\mathbf{S}_1$ zwei Signaturen. Dann ist nach [5] die Einschränkung einer $\mathbf{S}$-Interpretation $\mathcal{I} = \left(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\right)$ auf $\mathbf{S}_1$ eine Interpretation $\mathcal{I} \mid_{\mathbf{S}_1} = \left(\Delta^{\mathcal{I}_1}, \cdot^{\mathcal{I}_1}\right)$ über $\mathbf{S}_1$, so dass $\Delta^{\mathcal{I}_1} = \Delta^{\mathcal{I}}$ und $X^{\mathcal{I}_1} = X^{\mathcal{I}}$ für alle $X \in \mathbf{S}_1$. Analog ist die Expansion einer $\mathbf{S}_1$-Interpretation zu $\mathbf{S}$ eine $\mathbf{S}$-Interpretation $\mathcal{I}$, so dass $\mathcal{I} \mid_{\mathbf{S}_1} = \mathcal{I}_1$. Eine triviale Expansion einer $\mathbf{S}_1$-Interpretation $\mathcal{I}_1$ zu $\mathbf{S}$ ist eine Expansion von $\mathcal{I}_1$ zu $\mathbf{S}$, so dass $X^{\mathcal{I}} = \emptyset$ für alle atomaren Konzepte und atomaren Rollen $X \in \mathbf{S} \setminus \mathbf{S}_1$.

Die deduktive Hülle $\mathcal{O}^\star = \{\alpha \mid \mathcal{O} \models \alpha\}$ einer Ontologie $\mathcal{O}$ enthält alle Axiome α, die aus $\mathcal{O}$ folgen. Ein Axiom α' ist schwächer als ein Axiom α, wenn gilt $\alpha \models \alpha'$ und $\alpha' \not\models \alpha$.

Eine wichtige Eigenschaft von DLs ist es, dass man über das vorhandene Wissen schließen kann. Das bedeutet man kann implizites Wissen aus dem explizit in der Wissensbasis vorhandenen Wissen ableiten. Typische Inferenzprobleme sind nach [2] neben

Konsistenz der Wissensbasis Eine Ontologie ist konsistent wenn es eine Interpretation $\mathcal{I}$ gibt, die sowohl Modell von ABox als auch TBox ist.

für die TBox:

Erfüllbarkeit Kann ein Konzept C Individuen enthalten oder muss es immer leer sein? Ein Konzept C ist erfüllbar in einer Ontologie $\mathcal{O}$, wenn ein Modell $\mathcal{I}$ von $\mathcal{O}$ existiert, so dass $C^{\mathcal{I}}$ nicht leer ist.

Subsumption Ein Konzept C wird durch ein Konzept D in einer Ontologie $\mathcal{O}$ subsumiert, wenn $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ für jedes Modell $\mathcal{I}$ von $\mathcal{O}$.

Äquivalenz Sind zwei Konzepte semantisch gleich? Zwei Konzepte C und D sind äquivalent in einer Ontologie $\mathcal{O}$, wenn $C^{\mathcal{I}} = D^{\mathcal{I}}$ für jedes Modell $\mathcal{I}$ von $\mathcal{O}$.

Disjunktheit Sind zwei Klassen disjunkt? Zwei Konzepte C und D sind disjunkt in einer Ontologie $\mathcal{O}$, wenn $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ für jedes Modell $\mathcal{I}$ von $\mathcal{O}$.

und für eine ABox:

Konzeptzugehörigkeit Ist Individuum a Element von Konzept C? Ein Individuum a gehört zu einem Konzept C in einer Ontologie $\mathcal{O}$, wenn $a \in C^{\mathcal{I}}$ für jedes Modell $\mathcal{I}$ von $\mathcal{O}$.

Retrieval Finden aller Individuuen die zum Konzept C gehören. Ein Retrieval $R_{\mathcal{O}}(C)$ von einem Konzept C bezüglich einer Ontologie $\mathcal{O}$ ist die Menge von allen Instanzen von C: $R_{\mathcal{O}}(C) = \{a \mid a \in N_I \wedge \mathcal{O} \models C(a)\}$.

Mit den Subsumptionstests kann man die Konzepte einer Terminologie in einer Hierarchie, die Subsumptions-Hierarchie genannt, organisieren. Dieser Prozess wird häufig als Klassifikation (engl. classification) bezeichnet. Zudem gibt es oft den Begriff der Realisierung (engl. realization), bei dem zu jedem Individuum das speziellste Konzept berechnet, zu dem es gehört.

Alle Inferenzen können wie in [2] beschrieben auf die Unerfüllbarkeit der Wissensbasis zurückgeführt werden, so dass sich für Konzepte C, D in einer Ontologie $\mathcal{O}$ folgende Äquivalenzen ergeben:

Erfüllbarkeit $C \equiv \bot \Leftrightarrow \mathcal{O} \cup \{C(a)\}$ unerfüllbar, für ein neues Individuum a

Subsumption $C \sqsubseteq D \Leftrightarrow \mathcal{O} \cup \{(C \sqcap \neg D)(a)\}$ unerfüllbar, für ein neues Individuum a

Äquivalenz $C \equiv D \Leftrightarrow \mathcal{O} \cup \{((C \sqcap \neg D) \sqcup (\neg C \sqcap D))(a)\}$ unerfüllbar, für ein neues Individuum a

Disjunktheit $C \sqcap D \equiv \bot \Leftrightarrow \mathcal{O} \cup \{(C \sqcap D)(a)\}$ unerfüllbar, für ein neues Individuum a

Konzeptzugehörigkeit $C(a) \Leftrightarrow \mathcal{O} \cup \{\neg C(a)\}$ unerfüllbar

Retrieval Prüfen der Konzeptzugehörigkeit für alle Individuen in $\mathcal{O}$.

Zum Lösen dieser Inferenzprobleme existieren vollständige und korrekte Entscheidungsverfahren, meist basierend auf dem Tableaukalkül[1] oder Resolutionskalkül. Werkzeuge die diese Funktionen implementieren werden üblicherweise als Reasoner bezeichnet. Es existieren eine Vielzahl von Reasonern[8], die sich in der Regel in der Effizienz für bestimmte Problemarten unterscheiden. Dazu zählen unter anderem Pellet[9][36],Fact++[10][39] oder RacerPro[11][8] als Beispiele für das Tableauverfahren, aber auch andere wie KAON2[12][31] der z.B. das Resolutionsverfahren anwendet. Da die meisten Reasoner (noch) Tableaualgorithmen anwenden, werden wir das im folgenden Abschnitt 2.4 kurz beschreiben und in unserer Arbeit darauf beschränken.

2.4 Tableau Algorithmus

Tableau-Algorithmen versuchen die Erfüllbarkeit eines Konzepts C zu entscheiden, indem sie versuchen ein Modell bzw. eine Interpretation $\mathcal{I}$ zu konstruieren, in der $C^{\mathcal{I}}$ nicht leer ist. Ein Tableau ist ein Graph (sog. Vervollständigungsbaum (Completion Graph)), der ein solches Modell repräsentiert. Knoten im Tableau entsprechen Individuen (Elemente von $\Delta^{\mathcal{I}}$), Kanten sind Relationsbeziehungen zwischen Individuen (Elemente von $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$). Ein typischer Algorithmus beginnt mit einem einzelnen prototypischen Individuum, das das Konzept C als Markierung besitzt. Er versucht ein Tableau zu konstruieren, indem die Existenz weiterer Individuen oder zusätzlicher Restriktionen bzgl. Individuen abgeleitet wird. Der Inferenzmechanismus besteht aus der Anwendung von konsistenzerhaltenden Tableauexpansionsregeln, die den logischen Konstruktoren der verwendeten Sprache entsprechen [2]. So gehört zum Beispiel ein Indivduum, das mit einem Konzept $C \sqcap D$ markiert ist, sowohl zu C als auch zu D. Wenn ein Individuum x mit $\exists r.C$ markiert ist, dann gibt es ein weiteres Individuum y das zum Konzept C gehört, mit dem x durch eine Relation r in Beziehung steht. Regel heißt anwendbar, wenn durch ihre Anwendung das Tableau echt verändert wird. Ein Baum ist vollständig, wenn keine weitere Expansionsregel mehr anwendbar ist. Der Algorithmus terminiert, wenn entweder ein vollständiges Tableau vorliegt oder ein offensichtlicher Widerspruch (sog. Clash) gefunden wurde. Bei Nicht-determinismus ist die Suche möglicher Erweiterungen notwendig, d.h. ein Konzept ist unerfüllbar, wenn jede Erweiterung als

[8] Übersicht von aktuell verfügbaren Reasonern: `http://www.cs.manchester.ac.uk/~sattler/reasoners.html`

[9] Pellet Reasoner: `http://clarkparsia.com/pellet/`

[10] Fact++ Reasoner: `http://owl.man.ac.uk/factplusplus/`

[11] RacesPro Reasoner: `http://www.racer-systems.com/products/racerpro/index.phtml`

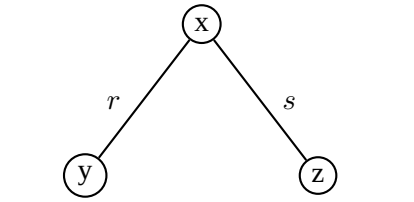
[12] KAON2 Reasoner: `http://kaon2.semanticweb.org/`

widersprüchlich abgeleitet werden kann oder erfüllbar, wenn eine mögliche Erweiterung zu einer vollständigen und nicht widersprüchlichen Struktur führt.

Als Notation eines Tableau wird eine Graphnotation verwendet. Die Erstellung eines Tableau erfolgt durch die Konstruktion eines gerichteten Graphen (ein Baum), in dem jeder Knoten x mit einer Menge $\mathcal{L}(x)$ von Konzepten markiert ist. Jeder Kante $\langle x, y\rangle$ wird der Name einer Rolle $\mathcal{L}(\langle x, y\rangle)$ zugeordnet. Befindet sich ein Konzept C in der Markierung eines Knotens ($C \in \mathcal{L}(x)$), so repräsentiert dies ein Modell, in dem das Individuum x Teil der Interpretation von C ist. Ist eine Kante $\langle x, y\rangle$ mit einer Rolle r markiert ($\mathcal{L}(\langle x, y\rangle) = r$), so entspricht dies einem Modell, in dem das mit $\langle x, y\rangle$ korrespondierende Tupel in der Interpretation von r liegt.

Beispiel 4 (Tableau Graph)**.** Abbildung stellt ein Tableau für das Konzept $A \sqcap \exists r.C \sqcap \exists s.D \sqcap \forall s.E$ dar.

$$\mathcal{L}(x) = \{A \sqcap \exists r.C \sqcap \exists s.D \sqcap \forall s.E, A, \exists r.C, \exists s.D, \forall s.E\}$$



$$\mathcal{L}(y) = \{C\} \qquad \mathcal{L}(z) = \{D, E\}$$

Die Unerfüllbarkeit eines Knotens ergibt sich, wenn ein Widerspruch, ein sog. Clash vorliegt. Ein Knoten enthält einen Clash, falls ein Konzept und dessen Negation in der Knotenmarkierung enthalten sind. Weiter liegt ein Clash vor, wenn mehr als m und weniger als n Füller für eine Relation gefordert sind: $(\geq ms) \sqcap (\leq nr)$ mit $s \sqsubseteq r, m > n$ und $m, n \in \mathbb{N}$. Ein Clash ergibt sich auch, wenn das speziellste Konzept $\bot$ in der Markierung eines Knotens enthalten ist.

Ein Tableau heißt widerspruchsfrei, wenn keine Knotenmarkierung einen Clash enthält. Ein Pfad in einem Tableau heißt abgeschlossen, wenn entlang des Pfades ein Widerspruch (Clash) gefunden wurde. Ein Tableau heißt abgeschlossen, wenn alle Pfade dessen abgeschlossen sind. Eine Anfrage ist unerfüllbar, wenn das Tableau abgeschlossen ist, sonst erfüllbar.

Beim Einsatz von Beschreibungslogiken in praktischen Anwendungsfeldern besteht der grundsätzliche Konflikt zwischen der Ausdrucksmächtigkeit der Sprachmittel und der praktischen Handhabbarkeit (tractability) der Inferenzverfahren. Nicht-deterministische Tableau-Algorithmen können eine sehr hohe Komplexität besitzen, insbesondere komplexe Konstrukte wie transitive oder inverse Rollen erhöhen diese und können sehr lange Laufzeiten nach sich ziehen. Um

die Handhabbarkeit bei ausdrucksstarken Beschreibungslogiken zu gewährleisten, empfiehlt es sich Optimierungsstrategien einzusetzen. Moderne Reasoner implementieren eine Reihe von Optimierungsverfahren [2], um akzeptable Performanz unter realen Bedingungen sicherzustellen. Dazu gehören u.a. die Normalisierung und Vereinfachung von Konzeptbechreibungen. Normalisierung bezeichnet die syntaktische Transformation von Konzeptbeschreibungen in eine einheitliche Normalform. Zum Beispiel kann man jede Konzeptbeschreibung in eine Negations-Normalform (Negation steht nur vor atomaren Konzepten) umformen, u.a. durch Anwendung der Morgan'schen Gesetze. Durch die Normalisierung ist man in der Lage mögliche Widersprüche schon frühzeitig zu erkennen, z.B. wird $C \sqcap \neg (C \sqcup D)$ in den Ausdruck $(C \sqcap \neg C) \sqcap neg$ transformiert, welcher offensichtlich einen Widerspruch enthält. Desweiteren können Ausdrücke auch vereinfacht werden, so wird z.B. $C \sqcap \bot$ zu $\bot$ vereinfacht.

$(A(a), \neg A(a))$

2.5 Web Ontology Language (OWL)

Die Web Ontology Language (kurz OWL)[13] ist eine weit verbreitete Sprache des Semantic Web, die dazu verwendet werden kann Ontologien zu beschreiben. Sie ist vor kurzem in der Version 2 als W3C Recommendation verabschiedet worden, und basiert formal auf der Beschreibungslogik $\mathcal{SROIQ}$ ([13]). Die enge Verwandschaft zu den Beschreibungslogiken spiegelt sich auch in ihren Konstrukten wieder (s. Tab. 2), wobei in OWL statt den Begriffen atomares Konzept, Rolle und Instanz, die Begriffe von Klasse (`owl:Class`), Property (`owl:ObjectProperty`, `owl:DataProperty`) und Individuum (`owl:Individual`) verwendet werden. Außerdem werden komplexe Klassen in OWL als Class Expressions bezeichnet. Die Inferenzprobleme in OWL sind analog zu denen in Beschreibungslogiken, so dass wir hier auf Abschnitt 2.3 verweisen.

Mit der Einführung von OWL 2 wurde auch eine Menge von Sprachprofilen[14] definiert, die die Arten der verwendbaren Konstrukte je nach Profil einschränken. Die dadurch eingeschränkte Ausdrucksmächtigkeit dient dem Ziel, die Effizienz im Reasoning zu erhöhen. Dabei erreicht jedes der Profile die Effizienz auf einem anderen Weg und ist jeweils sinnvoll in unterschiedlichen Anwendungsbereichen. Die Wahl des Profils ist dabei abhängig von der Struktur der Ontologie und den bevorzugten Inferenz-Aufgaben.

OWL verfügt über einige Eigenschaften, aus deren Unwissenheit oftmals Fehler hervorgehen können:

- In OWL existiert keine Unique Name Assumption (UNA). Zu zwei gegebenen Individuen a, b wird oftmals in DLs angenommen, dass diese unterschiedliche Objekte bezeichnen, d.h.

[13] W3C Seite zu OWL 2: `http://www.w3.org/2007/OWL/wiki/OWL_Working_Group`
[14] OWL 2 Profile: `http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/`

OWL Konstrukt	DL Repräsentation
`owl:Thing`	$\top$
`owl:Nothing`	$\bot$
`owl:equivalentClass`	$C \equiv D$
`rdfs:subClassOf`	$C \sqsubseteq D$
`owl:complementOf`	$C \sqsubseteq \neg D$
`owl:intersectionOf`	$C \sqcap D$
`owl:unionOf`	$C \sqcup D$
`owl:oneOf`	$\{a, b\}$
`owl:complementOf`	$\neg C$
`owl:allValuesFrom`	$\forall r.C$
`owl:someValuesFrom`	$\exists r.C$

Tabelle 2: Auszug aus dem Zusammenhang zwischen OWL - hier in RDF/XML Syntax - und Beschreibungslogik. Für vollständige und umfangreiche Informationen über Syntax und Semantik von OWL verweisen wir auf die W3C Recommendations[13]

$a^{\mathcal{I}} \neq b^{\mathcal{I}}$ für jede Interpretation $\mathcal{I}$. In OWL gilt diese Annahme jedoch nicht, das bedeutet zwei unterschiedliche Namen können sich auf dasselbe Objekt beziehen, was dazu führt, dass in OWL Verschiedenheit explizit ausgedrück werden muss. Ein Übersehen dieser Notwendigkeit kann mitunter zu ungewollten Inferenzen führen. Wenn wir zum Beispiel einmal annhemen dass eine OWL Ontologie die Aussagen $hatMutter\,(tom, anne)$ und $hatMutter\,(tom, tina)$ enthält, und zudem $hatMutter$ als funktional definiert wurde, dann folgt aus dieser Ontologie, dass $anne$ und $tina$ dasselbe Objekt bezeichnen. Um mit dem Fehlen der UNA umgehen zu können, bietet OWL die Konstrukte $owl : sameAs$ und $owl : differentFrom$ an, welche ausdrücken das zwei Objekte gleich bzw. verschieden sind

- Weil OWL auf der Semantik der Beschreibungslogik $\mathcal{SROIQ}$ basiert, gilt auch hier die in Beschreibungslogiken vorherrschende Open World Assumption (OWA). Unter der OWA versteht man die Eigenschaft, dass das Fehlen von Informationen nicht als negative Information gewertet wird. Wenn zum Beispiel explizit nur das Axiom $hoert\,(tom, mozart)$ in der OWL Ontologie vorhanden ist, bedeutet das nicht dass tom nur $mozart$ hört, sondern nur dass weitere Informationen zur Zeit unbekannt sind. Das hat für das Reasoning weitreichende Konsequenzen, zum Beispiel würde ein Reasoner auf die Anfrage ob tom zum Konzept $\forall mag.Klassik$ gehört mit „nein“ antworten, denn auch wenn $mozart$ zum Konzept $Klassik$ gehört, kann tom noch andere Interpreten hören, darüber ist nur nichts bekannt. Die OWA kann vor allem bei Nutzern, die bisher nur in Bereichen mit der Closed World Semantik zu tun hatten (wie zum Beispiel Datenbanken, Logikprogrammierung) zu Unverständnis bzw. Verwirrung führen.

2.6 DL-Learner

Der DL-Learner[15][25] ist ein in Java implementiertes Open-Source Framework zum Lernen in Wissensbasen basierend auf Beschreibungslogiken. Er bietet unter der Verwendung von Algorithmen des Maschinellen Lernens das Lösen von Lernproblemen, ähnlich denen in der Induktiven Logik Programmierung (ILP), an. So ist es zum Beispiel damit möglich Klassenbeschreibungen in OWL zu lernen.

Das Framework besitzt einen komponentenbasierten Aufbau (Abb. 3), der es ermöglicht einzelne Komponenten einfach auszutauschen, oder aber auch eigene neue zu integrieren. Es besteht aus folgenden vier Komponententypen:

Reasoning Für die zum Lernen benötigten Reasoner gibt es unterschiedliche Möglichkeiten. Zum einen kann man entweder über die Schnittstellen DIG 1.1[16] oder OWL API[17] jeden Standard OWL Reasoner ansprechen, oder einen internen approximativen Reasoner verwenden.

Wissenbasis Es werden verschiedene Formate als Eingabe von Wissensbasen unterstützt, so gibt es neben vielen OWL Formaten wie z.B. RDF/XML und Turtle auch ein DL-Learner internes Format, sowie eine Möglichkeit Fragmente von Wissen von einem SPARQL-Endpunkt zu extrahieren.

Lernproblem Der DL-Learner bietet zur Zeit drei unterschiedliche Arten von Lernproblemen an. Neben dem Lernen von Axiomen, dazu gehören Klassendefinitionen und Superklassen Axiome, kann man noch auf Basis von (nur) positiven bzw. positiven und negativen Beispielen lernen lassen.

Lernalgorithmus Es stehen unterschiedliche Lernalgorithmen zum Lösen der Lernprobleme zur Auswahl. Neben eher einfachen Bruteforce- oder Random-Algorithmen gehören dazu u.a. auch Algorithmen basierend auf Refinementoperatoren [24; 27; 28; 29] oder Genetischen Algorithmen [23].

Der DL-Learner bietet zahlreiche Schnittstellen zum Einsatz an, so gibt es neben einem CLI und einer GUI auch eine WSDL Schnittstelle, um den DL-Learner als Webservice nutzen zu können. Desweiteren existiert ein Plugin für Protégé[19][33] und es wird an einem Plugin für das OntoWiki[20] gearbeitet.

[15] DL-Learner Projekt: `http://dl-learner.org/Projects/DLLearner`
[16] DIG 1.1: `http://dig.sourceforge.net/`
[17] OWL API: `http://owlapi.sourceforge.net/`
[18] Quelle: `http://dl-learner.org/wiki/Architecture`
[19] Protégé: `http://protege.stanford.edu/`
[20] OntoWiki: `http://ontowiki.net/Projects/OntoWiki`

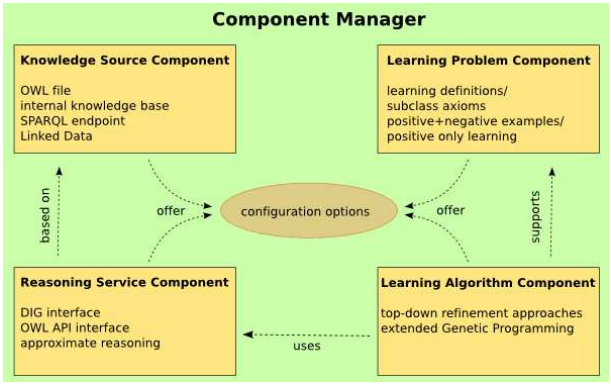


Abb. 3: Architektur des DL-Learner, in dem jede der vier Komponententypen einzeln konfiguriert und mit Hilfe eines Komponentenmanagers verwaltet werden kann.[18]

3 Fehler in Ontologien

Fehler in OWL-Ontologien können aus den unterschiedlichsten Gründen und in einer Vielzahl an Arten auftreten. Wir werden in diesem Kapitel zunächst einige Gründe für die Entstehung von Fehlern nennen, und im Anschluss daran drei verschiedene Möglichkeiten beschreiben, eine Unterteilung der Fehler in unterschiedliche Kategorien vorzunehmen. Wir werden dabei sowohl die jeweiligen Kategorien erläutern, als auch erwähnen ob und wenn ja wie die automatische Behebung solcher Fehler umsetzbar ist.

Durch die zunehmende Wichtigkeit von Ontologien, vor allem auch durch den Einsatz im Semantic Web entstehen auch immer mehr Fehler bei der Verwendung. Einige mögliche Gründe dafür sind:

- Schwierigkeiten im Verständnis bei der Modellierung.
- Zusammenführen von mehreren Ontologien zu einer.
- Migration zu OWL aus anderen Formaten wie z.B. XML oder Text.
- Verteiltes Erstellen von Ontologien.

Fehler in OWL Ontologien können verschiedene Ursachen haben, wobei eine erste grobe Einteilung zu folgenden drei Kategorien führt: syntaktische, semantische und Modellierungsfehler.

syntaktische Fehler Syntaktische Fehler in OWL Ontologien treten häufig vor allem dann auf, wenn man versucht diese manuell zu erstellen und dabei auf Ontologie-Erstellungswerkzeuge verzichtet. Dies liegt zum einen an der sicherlich gewöhnungsbedürftigen OWL/XML Syntax, als auch an der einheitlichen Verwendung von URIs. Für diese Form von Fehlern gibt es aber mittlerweile geeignete Werkzeuge wie XML-Parser und OWL Validatoren[21] zum einfachen Finden und Korrigieren dieser.

semantische Fehler Für OWL Ontologien, die keine Konflikte auf der syntaktischen Ebene enthalten, sind semantische Fehler genau solche, die durch OWL Reasoner gefunden werden. Im Allgemeinen handelt es sich dabei um inkonsistente Ontologien und um unerfüllbare Klassen.

Modellierungsfehler Als Modellierungsfehler bezeichnen wir Fehler, die nicht notwendigerweise syntaktische oder semantische Fehler sind, aber trotzdem zu Unstimmigkeiten oder unerwarteten Resultaten bei der Modellierung in der Wissensbasis führen. Dazu können unter anderem zum Beispiel ungewollte Inferenzen, Redundanz oder nicht verwendete atomare

[21] OWL Syntax Validator: `http://owl.cs.manchester.ac.uk/validator/`

Klassen und Properties gehören. Da Modellierungsfehler stark von den Absichten des Modellierers abhängen, können Werkzeuge wie Reasoner können meist nur eine Unterstützung beim Finden solcher Fehler leisten, nicht aber zum Beispiel feststellen, welche Inferenzen gewollt sind, und welche nicht.

Eine andere Einteilung von Fehlern in Ontologien wird in [3] gegeben. Der Fokus liegt dort auf der Evaluation von Taxonomien. Dabei werden die Taxonomie-Fehler in die drei Haupttypen Inkonsistenz, Unvollständigkeit und Redundanz unterteilt. Diese werden wie in Abb.4 veranschaulicht in weitere Klassen unterteilt.

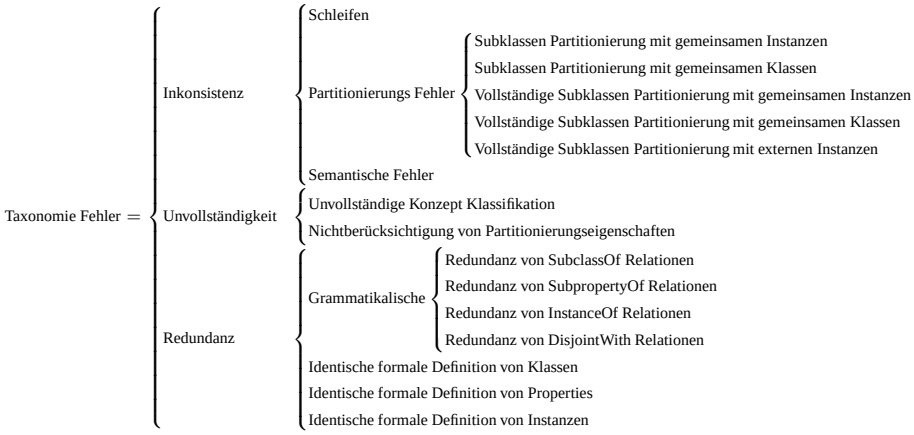


Abb. 4: Einteilung der Taxonomie Fehler

Inkonsistenz

Schleifen

Schleifen treten dann auf, wenn eine Klasse als Spezialisierung oder Generalisierung von sich selbst auf einem beliebigen Level der Hierarchie definiert wird.

Partitionierungsfehler

Es gibt verschiedene Vorgehensweisen eine Klasse in Subklassen zu zerlegen. Zwei Prinzipen dabei sind zum einen eine vollständige Zerlegung und zum anderen eine Zerlegung in disjunkte Subklassen. Vollständigkeit bedeutet, dass keine Instanzen der Klasse existieren, die nicht zu einer der Subklassen gehören, oder anders ausgedrückt besteht die Menge der Instanzen der zerlegten Klasse aus der Vereinigung der Mengen der Instanzen der Subklassen. Eine disjunkte Zerlegung bedeutet, dass

keine Instanz einer Subklasse zu einer anderen gehört, also die Mengen der Instanzen der Subklassen paarweise disjunkt sind. Partitionierungfehler entstehen, wenn eine Instanz mehr als einer disjunkten Subklasse zugeordnet wird, oder eine Klasse erstellt wird, die Subklasse von mehreren disjunkten Klassen ist. In einer vollständigen Zerlegung treten Fehler auf, wenn Instanzen existieren, die keiner der Subklassen angehören, sondern nur der zerlegten Klasse.

Semantische Fehler

Semantische Fehler treten auf, wenn der Entwickler einer Ontologie eine inkorrekte semantische Klassifizierung macht, d.h. eine Klasse wird als Subklasse einer anderen Klasse zugeordnet, zu der keine semantische Beziehung existiert. Zum Beispiel ist $Auto$ als Subklasse von $Tier$ eine in unserem Sinne falsche Klassifizierung. Hierbei ist erkennbar, dass es schwer ist solche Fehler automatisch zu finden, weil es ein subjektives Verständnis vorraussetzt.

Unvollständigkeit

Unvollständige Konzept Klassifizierung

Diese Fehler entstehen, wenn möglicherweise wichtige Konzepte einer Domäne bei der Klassifizierung übersehen werden. Wenn zum Beispiel das Konzept $Verkehrsmittel$ klassifiziert werden soll, so ist eine Unterteilung in PKW und Bus sicherlich unvollständig, da es noch weitere gibt wie zum Beispiel das $Motorrad$ oder das $Flugzeug$.

Nichtberücksichtigung von Wissen über Disjunktheit

Diese Fehler treten auf, wenn der Entwickler eine Menge von Subklassen zu einer gegebenen Klasse definiert, aber vergisst diese als disjunkt zu deklarieren.

Nichtberücksichtigung von Wissen über Vollständigkeit

Fehler dieser Art entstehen, wenn der Entwickler eine Partitionierung einer Klasse definiert, und dabei vergisst diese Zerlegung als vollständig zu definieren.

Nichtberücksichtigung der (inversen) Funktionalitätseigenschaft einer Property

Wenn es zu einer Property pro Subjekt nur einen Wert gibt, so gilt diese als funktional. Diese Eigenschaft kann sowohl für DatatypeProperties, als auch für ObjectProperties gelten. So hat

zum Beispiel eine Person über 18 Jahren in Deutschland nur eine Personalausweisnummer, oder ein Mensch nur ein Geburtsdatum. Vergisst oder ignoriert man eine Property als funktional auszuzeichnen, so kann das zu Inkonsistenzen in den Daten führen. analog gilt das für inverse Properties.

Redundanz

Redundanz von SubclassOf, SubpropertyOf und InstanceOf Relationen

Redundante SubclassOf Relationen zwischen verschiedenen Klassen treten auf, wenn mehr als eine direkte oder indirekte SubclassOf Relation existiert. Direkt heißt dass Subklasse und Superklasse gleich sind, und indirekt bedeutet dass eine SubclassOf Relation zwischen einer Klasse und ihrer Superklasse auf einem beliebigen Level in der Hierarchie existierst. Gleiches gilt auch für die SubpropertyOf Relationen im Bezug auf die Property Hierarchie. Im Fall von InstanceOf Relationen entsteht Redundanz wenn man eine Instanz mehrfach derselben Klasse, oder einer Klasse und einer ihrer Superklassen zuordnet.

Redundanz von DisjointWith Relationen

Redundante DisjointWith Relationen entstehen, wenn ein Konzept mehr als einmal explizit mit einem anderen als disjunkt definiert wird. Auch hier gibt es wieder die Unterscheidung zwischen direkt und indirekt, analog zu den SubclassOf, SubpropertyOf und InstanceOf Relationen.

Identische formale Definition von Instanzen, Klassen und Properties

Diese Fehler treten auf, wenn der Entwickler verschiedene (oder gleiche) Namen für zwei oder mehr Instanzen, Klassen und Properties definiert, aber jeweils die gleiche formale Definition verwendet.

Haben die ersten beiden Ansätze eine Unterscheidung nach möglichen Fehlerarten vorgenommen, so gibt es auch noch einen etwas anderen Ansatz, der Unterscheidung nach der Konsistenz einer Ontologie. Nach [9] werden drei Formen von Konsistenz in einer Ontologie unterschieden: Strukturelle Konsistenz, Logische Konsistenz und Nutzer-definierte Konsistenz. Diese Formen unterscheiden sich wie folgt:

Strukturelle Konsistenz Eine Ontologie gilt als strukturell konsistent, wenn die Struktur der Ontologie den Sprachkonstrukten entspricht, die durch die zugrundeliegende Ontologiesprache vorgegeben sind. Strukturelle Konsistenz kann erzwungen werden, in dem man eine Menge von strukturellen Bedingungen der jeweiligen Sprache testet. Zum Beispiel könne das Bedingungen sein, wie „das Komplement einer Klasse ist eine Klasse“, „eine Property kann nur Subproperty von einer Property sein“ usw. .

Logische Konsistenz Eine Ontologie wird als logisch konsistent angesehen, wenn die Ontologie mit der zugrundeliegenden logischen Theorie der verwendeten Ontologiesprache übereinstimmt.

Nutzer-definierte Konsistenz Das bedeutet, ein Nutzer kann zusätzliche eigene Bedingungen definieren, damit eine Ontologie als konsistent angesehen wird. Zum Beispiel könnte der Nutzer verlangen, dass eine Klasse nur eine Superklasse besitzt damit die Ontologie konsistent ist.

Diese drei Ansätze haben gezeigt dass eine Vielzahl von möglichen Fehlern in Ontologien existieren, und es viele Möglichkeiten gibt diese zu unterscheiden. Es wird auch deutlich, dass nicht alle diese Fehler automatisch durch Werkzeuge erkannt werden können, da oft weder das subjektive Wissen vorhanden ist, noch erkennbar ist, was bei der Modellierung einer Ontologie beabsichtigt ist, und was nicht.

4 Debugging

Mit dem Beginn des Semantic Web kommen immer mehr OWL Ontologien als Wissensbasis zum Einsatz, und eine immer größer werdende Community kommt mit OWL und somit indirekt mit den logischen Gegebenheiten ausdrucksstarker Beschreibungslogiken wie $\mathcal{SROIQ}$ in Kontakt. Da die Komplexität bei der Modellierung auch zur Entstehung von logischen Fehlern führen kann, ist es wichtig Nutzer und Entwickler von Ontologien beim Finden, Erklären und Beheben dieser zu unterstützen. Wir werden in diesem Kapitel zunächst kurz auf die Schwierigkeit beim manuellen Finden solcher Fehler eingehen. Danach werden wir einige terminologische Grundlagen, u.a. den Begriff Erklärung für die Fehler, formal definieren. Daran anschließend werden wir die unterschiedlichen Arten (Black-Box, Glass-Box) für die Berechnung von einer oder mehreren Erklärungen betrachten, und dabei die Unterschiede, sowie mögliche Vor- und Nachteile herausstellen. Durch die Abhängigkeit dieser durchaus effizienten Algorithmen von OWL Reasonern, stellen wir danach ein Optimierungsmöglichkeit vor, bei denen der eigentliche Suchraum für den Reasoner durch die Extraktion von Modulen verkleinert wird, was zu einem erheblichen Geschwindigkeitsvorteil führen kann. Weil das eigentliche Auflösen logischer Konflikte vor allem mit vielen gegebenen Erklärungen eine komplexe Aufgabe sein kann, werden wir zusätzlich auf Unterstützungsmöglichkeiten eingehen, bevor wir abschließend noch auf einige Probleme bei inkonsistenten Ontologien hinweisen werden.

Alle heutigen Reasoner sind in der Lage logische Fehler zu erkennen, und den Nutzer somit darauf hinzuweisen. Es werden dabei sowohl inkonsistente Ontologien, als auch unerfüllbare Konzepte erkannt. Zu wissen welches Konzept zum Beispiel keine Instanzen haben kann ist sicherlich eine große Unterstützung bei der Entwicklung oder Nutzung von Ontologien. Allerdings kann es vor allem bei größeren Ontologien mit vielen logischen Axiomen selbst für Experten in der Domäne und trotz der Verwendung von Werkzeugen wie z.B. Protégé schwierig und sehr aufwändig sein, den Grund für diese Fehler (das heißt die Axiome dafür) zu finden, und dann eine geeignete Reparatur vorzunehmen. Hier ist es hilfreich, die Nutzer dabei zu unterstützen was vor allem durch das Finden bzw. Berechnen von Erklärungen (die relevanten Axiome) für diese Fehler getan werden kann[35].

4.1 Grundlagen

Definition 1 (Inkonsistente Ontologie)**.** *Eine Ontologie $\mathcal{O}$ ist inkonsistent, wenn kein Model für sie existiert.*

Definition 2 (Unerfüllbares Konzept)**.** *Eine Konzept C ist unerfüllbar in einer Ontologie $\mathcal{O}$, wenn für jede Interpretation I von $\mathcal{O}$ gilt $C^I = \emptyset$.*

Definition 3 (Inkohärente Ontologie)**.** *Eine Ontologie $\mathcal{O}$ ist inkohärent, wenn ein unerfüllbares Konzept in $\mathcal{O}$ existiert.*

Definition 4 (Erklärung)**.** *Sei $\mathcal{O}$ eine Ontologie und η eine Folgebeziehung (engl. entailment) mit $\mathcal{O} \models \eta$. Eine Menge von Axiomen $\mathcal{J} \subseteq \mathcal{O}$ ist eine Erklärung für η in $\mathcal{O}$, wenn $\mathcal{J} \models \eta$, und $\mathcal{J}' \not\models \eta$ für alle $\mathcal{J}' \subset \mathcal{J}$*

(Anm.: in der Literatur werden neben dem Begriff Erklärung (engl. explanation, justification) manchmal auch die Begriffe MUPS (Minimal Unsatisfiability Preserving Sub-Tbox) oder MinAs (Minimal Axiom-set) verwendet.)

Intutitiv ist eine Erklärung die kleinste Teilmenge von Axiomen in der Ontologie, so dass die Folgebeziehung gilt. Dabei ist zu beachten dass es nicht nur genau eine Erklärung für Folgebeziehungenen gibt, sondern es mehrere Erklärungen pro Folgebeziehung geben kann. Im schlechtesten Fall sind es exponentiell viele bezogen auf die Anzahl der Axiome in der Ontologie, d.h. $2^{|\mathcal{O}|}$ für eine Ontologie $\mathcal{O}$. Zusammenfassend gelten folgende Eigenschaften für Erklärungen:

- Eine Folgebeziehung kann mehrere Erklärungen besitzen.
- Falls eine Folgebeziehung mehrere Erklärungen besitzt, so können sich diese überschneiden.
- Das Entfernen eines Axioms von der Erklärung führt dazu, dass die Folgebeziehung in den verbleibenden Axiomen nicht länger gilt.
- Für eine Reparatur muss mindestens ein Axiom von jeder Erklärung entfernt werden.
  Das bedeutet insbesondere, dass man für einen Reparaturplan alle Erklärungen benötigt.

Im weiteren Verlauf dieses Kapitels behandeln wir Erklärungen weitestgehend in Bezug auf die Unerfüllbarkeit von Konzepten. Wir möchten aber darauf hinweisen, dass Erklärungen auch für andere Problemstellungen, wie zum Beispiel bei der Frage nach dem Grund warum ein Konzept C von einem Konzept D subsumiert ($C \sqsubseteq D$) wird, geeignet sind. Weil diese jedoch auf die Unerfüllbarkeit von Konzepten zurückführbar sind (vgl. Abschnitt 2.3), beschränken wir uns hier darauf.

Für das Berechnen von einer bzw. mehreren Erklärungen gibt es unterschiedliche Ansätze (s. Abb. 5). Wir werden zunächst die Ansätze für eine einzelne Erklärung beschreiben, und anschließend zeigen wie man alle (oder meherer) Erklärungen finden kann.

4.2 Berechnen einer einzelnen Erklärung

Im Allgemeinen werden zwei unterschiedliche Techniken für das Berechnen von Erklärungen unterschieden: Black-Box und Glass-Box:

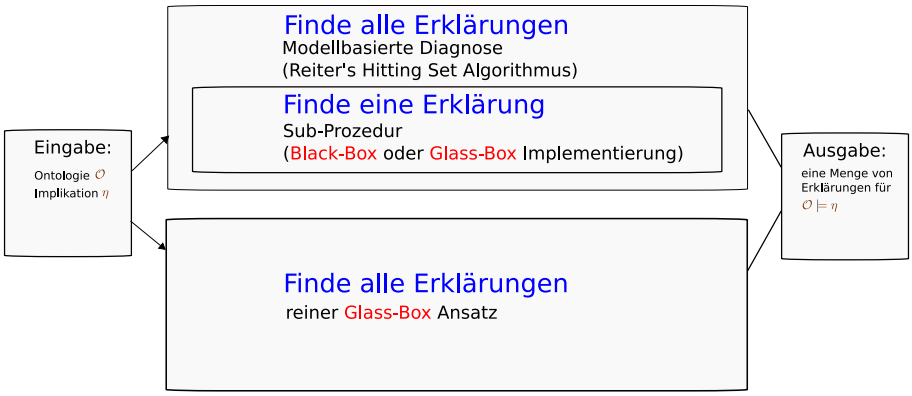


Abb. 5: Übersicht über die unterschiedlichen Ansätze zur Berechnung von einer bzw. mehrerer Erklärungen.

- Black-Box Algorithmen nutzen den Reasoner als Subroutine. Sie werden auch als Reasoner-unabhängig bezeichnet. Die einzige Vorraussetzung ist ein korrekt und vollständig arbeitender Reasoner
- Glass-Box Algorithmen sind in den Reasoner selbst integriert und erfordern somit eine i.A. nicht triviale Modifikation der Interna des Reasoners. Glass-Box Algorithmen sind damit als Reasoner-abhängig anzusehen.

Wir werden im folgenden auf beide Techniken genauer eingehen, und versuchen anhand von Algorithmen und Beispielen wesentliche Unterschiede herauszustellen.

4.2.1 Black-Box

Black-Box Algorithmen haben überlichweise folgenden Ablauf: ausgehend von einem unerfüllbaren Konzept C in einer Ontologie $\mathcal{O}$ fügen wir zunächst solange Axiome aus $\mathcal{O}$ zu einer neuen Ontologie $\mathcal{O}'$ hinzu, bis C in $\mathcal{O}'$ unerfüllbar ist (Expansions-Phase). Wir erhalten damit eine Obermenge der Erklärung. In der zweiten Phase entfernen wir dann solange überflüssige Axiome aus $\mathcal{O}'$ bis wir eine minimale Erklärung für die Unerfüllbarkeit von C erhalten (Minimierungs-Phase). Ein Algorithmus dafür wurde in [18] vorgeschlagen und wird von uns nachfolgend genauer beschrieben.

Als Eingabe dient eine Ontologie $\mathcal{O}$ und ein unerfüllbares Konzept C in dieser Ontologie, für das eine Erklärung gefunden werden soll. In Zeile 1 wird zunächst eine leere Ontologie $\mathcal{O}'$ erzeugt. Die Zeilen 2 - 4 beschreiben die bereits erwähnte Expansions-Phase. Hier wird solange eine Menge von Axiomen s aus $\mathcal{O}$ zu $\mathcal{O}'$ hinzugefügt, bis C unerfüllbar in $\mathcal{O}'$ ist. Die anschließende Minimierungs-Phase findet dann in der zweiten Schleife (Zeile 6 - 9) statt. Dort wird für jedes Axiom α in der vorher

Algorithmus :EINZELNE_ERKLÄRUNG$_{Black-Box}$

Eingabe : Ontologie $\mathcal{O}$, Unerfüllbares Konzept C

Ausgabe : Ontologie $\mathcal{O}'$

```
1  O' ← ∅
2  solange C ist erfüllbar in O' tue
3      wähle eine Menge von Axiomen s ⊆ O \ O'
4      O' ← O' ∪ s
5  benutze Schnelle Reduktion von O'
6  für jedes Axiom α ∈ O' tue
7      O' ← O' − {α}
8      wenn C ist erfüllbar in O' dann
9          O' ← O' ∪ {α}
10 zurück O'
```

erzeugten Ontologie $\mathcal{O}'$ überprüft, ob ein Entfernen des Axioms zur Erfüllbarkeit von C führt. Ist dies der Fall, so wird das Axiom α wieder zur Ontologie $\mathcal{O}'$ hinzugefügt.

Dieser Algorithmus enthält in der dargestellten Form zwei Möglickeiten zur Optimierung. Die eine Möglichkeit befindet sich in Zeile 3. Hier ist es sinnvoll die Axiome nach einem bestimmten Kriterium auszuwählen, und nicht einfach zufällige Mengen zu wählen. Eine Möglichkeit hier ist es, Axiome nach der syntaktischen Relevanz auszuwählen. Das bedeutet, es werden die Axiome bevorzugt, die in ihrer Signatur das Konzept C bzw. in weiteren Schleifen-Durchläufen bezüglich der hinzugefügten Axiome gemeinsame Entitäten enthalten.

Die zweite Optimierungsmöglichkeit liegt zwischen den beiden eigentlichen Phasen (Zeile 5). In [18] wird vorgeschlage vor der eigentlichen Minimierungs-Phase eine Phase der schnellen Reduktion (engl. fast pruning) zu verwenden. Die Idee dahinter ist folgende: die eigentliche Minimierungs-Phase überprüft immer nur ein einzelnes Axiom. Weil in dieser Phase somit genau $|\mathcal{O}'|$ mal die Schleife durchlaufen wird, und jeweils eine Erfüllbarkeitsanfrage an den Reasoner gestellt wird, kann das mit unter sehr lange dauern. Um das zu begrenzen, kann man eine Sliding-Window Technik verwenden. Das bedeutet dass nicht nur für ein einzelnes Axiom überprüft wird ob C nach Entfernen erfüllbar in $\mathcal{O}'$ ist, sondern für eine Menge von Axiomen in der Anzahl der Fenstergröße n. Eine Alternative dazu ist z.B. auch ein Vorgehen nach dem Divide and Conquer Ansatz.

4.2.2 Glass-Box

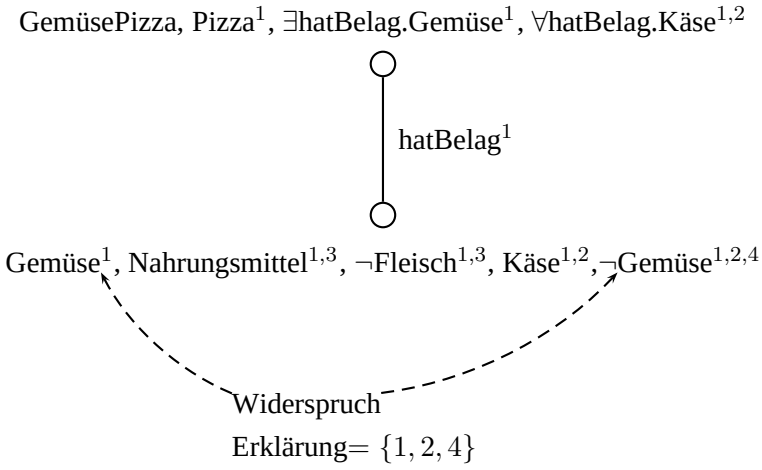
Eine alternative Variante zum Black-Box Ansatz ist die schon erwähnte Glass-Box Methode. Hierbei werden, im Gegensatz zu der Black-Box Methode, Interna der Reasoner modifiziert um die minimale Menge an Axiomen zu identifizieren, die für eine Folgebeziehung relevant sind. Das bedeutet insbesondere, dass die jeweils verwendeten Glass-Box Techniken speziell auf einen bestimmten Reasoner angepasst sind, und eine Adaption auf andere Reasoner nicht ohne weiteres möglich ist. Aktuelle Beispiele für die Umsetzung der Glass-Box Variante sind Pellet, CEL und DION. Wir bezeichen den Algorithmus zunküftig mit *EINZELNE_ERKLÄRUNG*$_{Glass-Box}$.

Wie wir bereits in Abschnitt 2.3 erwähnt haben, nutzen eine Mehrzahl der bekannten Reasoner das Tableauverfahren. Die Idee der Glass-Box Methode ist es, das schon verhandene Tableau-Verfahren so zu erweitern, dass die Axiome, die für jeweilige Modifikationen im Tableau verantwortlich sind, durch das Tableau mitgeführt werden. Diese Technik wird häufig als Tableau Tracing (kurz: Tracing) bezeichnet.

Beispiel 5. Betrachten wir zur Veranschaulichung einmal folgendes Beispiel:

$$\begin{aligned}
\text{GemüsePizza} &\sqsubseteq \text{Pizza} \sqcap \exists \text{hatBelag.Gemüse} && (1)\\
\text{Pizza} &\sqsubseteq \forall \text{hatBelag.Käse} && (2)\\
\text{Gemüse} &\sqsubseteq \text{Nahrungsmittel} \sqcap \neg \text{Fleisch} && (3)\\
\text{Käse} &\sqsubseteq \neg \text{Gemüse} && (4)
\end{aligned}$$

Man sieht leicht, dass das Konzept $GemüsePizza$ wegen den Axiomen 1,2 und 4 unerfüllbar ist. Stellt man jetzt an einem um das Tableau Tracing erweiterten Reasoner eine Anfrage nach der Erfüllbarkeit von $GemüsePizza$, so ergibt sich folgender Graph:



Man sieht dabei in den jeweiligen Termen der Knoten die für die Herkunft verantwortlichen Axiome als Superskript vermerkt, wobei wir hier aber anmerken müssen, dass dies nur eine vereinfachte Darstellung ist, und die technischen Umsetzungen sicherlich etwas komplexer aussehen können.

Dieses Verfahren scheint auf den ersten Blick einfach umsetzbar zu sein, allerdings gibt es einige Hindernisse, die beim Verwenden zu beachten sind:

1. Die Tableau-Algorithmen nutzen wie schon in 2.4 erwähnt eine Vielzahl von Optimierungen und damit verbundene Umformungen der tatsächlich in der Ontologie vorhandenen Axiome.
   Lösung: Tracing schon beim Vorverarbeiten anwenden.

2. Es kann mehr als ein Axiom geben, das für einen Tableauzustand verantwortlich ist. Betrachten wir zum Beispiel folgende Wissensbasis:

$$C \sqsubseteq \exists R.\,(D \sqcap E) \tag{1}$$
$$C \sqsubseteq \forall R.E \tag{2}$$
$$D \sqsubseteq \neg E \tag{3}$$

$$C \xrightarrow{R^1} D^1, E^1, E^{1,2}, (\neg E)^{1,3}$$

   Lösung: Tracing so anpassen dass Mengen von Axiommengen verwendet werden

$$C \xrightarrow{R^1} D^{\{\{1\}\}}, E^{\{\{1\},\{1,2\}\}}, (\neg E)^{\{\{1\}\}}$$

In [16] wurde gezeigt, dass im Reasoner integriertes Tracing für die Beschreibunglogik $\mathcal{SHIF}\,(D)$ einen hinsichtlich der Berechnungszeit nur geringfügigen Overhead verursacht, so dass eine eindeutige Empfehlung gegeben wird, diese Funktion zu aktivieren wenn sie verfügbar ist. Allerdings wird auch darauf hingewiesen, dass das dort beschriebene Tracing die ausdrucksstärkere Logik $\mathcal{SHOIN}\,(D)$ nur annähernd abdeckt. Das bedeutet, dass die Ausgabe des Tracing Algorithmus nicht unbedingt eine Erklärung sein muss, sondern auch eine Obermenge einer Erklärung sein kann. Ursache dafür sind speziell die nichtdeterministischen Verschmelzungoperationen, verursacht durch Maximum-Kardinalitätsrestriktionen. In einem solchen Fall kann jedoch wieder eine Minimierungsphase wie Schritt 2 im Black-Box Ansatzes verwendet werden, um die überflüssigen Axiome zu entfernen.

4.3 Berechnen von allen Erklärungen

Bisher haben wir nur Verfahren beschrieben, die eine einzelne Erklärung finden. Wie wir aber in 4.1 bereits erwähnt haben, kann eine Folgebeziehung mehr als eine Erklärung haben, und sie gilt erst dann nicht mehr, wenn aus jeder Erklärung mindestens ein Axiom entfernt wurde.

Weil eines der Hauptziele in unserem Programm die Reparatur ist, so reicht es hier nicht aus nur eine einzelne Erklärung zu finden. Ein einfacher Lösungsweg wäre es, dass wir zunächst eine einzelne Erklärung berechnen und dort mindestens ein Axiom entfernen. Falls das Konzept danach immer noch unerfüllbar ist, so könnten wir eine weitere Erklärung finden und wieder mindestens ein Axiom entfernen. Das könnten wir jetzt in einem iterativen Prozess solange wiederholen, bis das Konzept erfüllbar ist. Was bei dieser Vorgehensweise sofort auffällt ist, dass bei vielen Erklärungen mit jeweils vielen Axiomen, ein solcher Prozess selbst für Experten in dieser Domäne schwierig wird. Da die jeweils folgende Erklärung noch nicht bekannt ist, wissen wir nicht welche Axiome eventuell dort auch vorkommen, und somit möglichweise eine Kernursache sind. Ein wesentlich effizienteres Vorgehen wäre alle Erklärungen zu finden, und somit einen Reparaturplan zu ermöglichen.

Wie schon beim Berechnen einer einzelnen Erklärung gibt es auch hier wieder die Unterscheidung zwischen Black-Box und Glass-Box Ansätzen. Wir werden im Folgenden nur einen Black-Box Ansatz ausführlich beschreiben, weil der Glass-Box Ansatze viele Probleme mit sich bringt[18]. Eine Möglichkeit alle Erklärungen ausgehend von einer initialen Erklärung zu finden wurde in [18] vorgeschlagen, wir werden sie hier genauer vorstellen. Sie basiert auf einer Abwandlung des in [34] vorgestellten Hitting Set Tree (HST) Algorithmus. Dieser Algorithmus stammt aus dem Bereich der Modellbasierten Diagnose, einem übergreifenden Begriff für den Prozess der Berechnung von Diagnosen für fehlerhafte Systeme. Eine Diagnose ist vereinfacht gesagt eine minimale Menge von Komponenten in einem fehlerhaften System, deren Ersetzung das System reparieren würde, so dass es (wieder) so funktioniert wie es soll. Modellbasiert deshalb, weil mit Hilfe von Modellen das beabsichtige Verhalten des Systems abstrahiert wird, um es dann mit dem beobachteten Verhalten vergleichen zu können. Anwendung finden Modellbasierte Diagnosetechniken u.a. bei dem Entwurf und Testen von elektronischen Bauteilen.So kann zum Beispiel die Erstellung von komplexen Schaltkreisen aus vielen einzelnen Komponenten damit modelliert und getestet werden.

4.3.1 HittingSet Algorithmus

Gegeben sei eine universelle Menge U, und eine Menge $S = \{s_1, \cdots, s_n\}$ von Teilmengen von U, die Konfliktmengen, d.h. die für den Fehler verantwortlichen Systemkomponenten, sind. Ein Hitting Set T für S ist eine Teilmenge von U, so dass $s_i \cap T = \emptyset$ für alle $1 \leq i \leq n$ gilt. T ist ein minimaler Hitting Set für S, wenn T ein Hitting Set für S ist, und es kein $T' \subset T$ gibt, das ein

Hitting Set für S ist. Reiter's Algorithmus wird benutzt, um die minimalen Hitting Sets für eine Menge $S = \{s_1, \cdots, s_n\}$ von Mengen zu berechnen, in dem ein markierter Baum konstruiert wird, der sogenannte Hitting Set Tree (HST). In einem HST ist jeder Knoten mit einer Menge $s_i \in S$ und jede Kante mit einem Element $\sigma \in \bigcup_{s_i \in S}$markiert. Für jeden Knoten v in einem HST sei $H(v)$ die Menge von Kantenmarkierungen von der Wurzel bis zum Knoten v. Dann ist die Markierung für v eine beliebige Menge $s \in S$ so dass $s \cup H(v) = \emptyset$, falls so eine Menge existiert. Angenommen s ist die Markierung von einem Knoten v, dann existiert für jedes Element $\sigma \in s$ ein Nachfolgeknoten v_σ für v, der mit v durch eine mit σ markierte Kante verbunden ist. Im Hinblick auf die Berechnung aller Erklärungen ist die universelle Menge U die Menge aller Axiom in der Ontologie, und eine Konfliktmenge s ist eine einzelne Erklärung.

Algorithmus 1 beschreibt das Verfahren beim Berechnen aller Erklärungen. Der Algorithmus erhält als Eingabe eine Ontologie $\mathcal{O}$ und ein unerfüllbares Konzept C. Zunächst berechnen wir in Zeile 2 eine initiale einzelne Erklärung und fügen sie zur Menge der Erklärungen $\mathcal{J}$ hinzu (Zeile 3). Dabei ist es egal ob wir einen Glass-Box oder Black-Box Algorithmus verwenden. Diese Erklärung entspricht jetzt der Wurzel in unserem HST. Danach wählen wir ein beliebiges Axiom α aus der Erklärung (Zeile 4) und entfernen dieses aus der Ontologie $\mathcal{O}$. Mit der neuen Ontologie $\mathcal{O} \setminus \alpha$ rufen wir jetzt die rekursive Prozedur *EXPANDIERE_HST* auf (Zeile 6). Dort berechnen wir nochmals eine einzelne Erklärung für die Unerfüllbarkeit von C, aber diesmal in unserer neuen Ontologie (Zeile 6). Wenn es eine neue Erklärung gibt, dann rufen wir für jedes Axiom β in dieser neuen Erklärung erneut die Prozedur *EXPANDIERE_HST* auf, wobei wir diesmal aus der Ontologie zusätzlich noch das Axiom β entfernt haben. Das ganze läuft solange rekursiv ab bis es in dem Pfad des HST keine neuen Erklärungen mehr gibt.

Algorithmus :ALLE_ERKLÄRUNGEN

Eingabe : Ontologie $\mathcal{O}$, Unerfüllbares Konzept C

Ausgabe : Menge $\mathcal{J}$ von Erklärungen

1 **Global:** $\mathcal{J} \leftarrow HS \leftarrow \emptyset$
2 $erklärung \leftarrow$ EINZELNE_ERKLÄRUNG$(C, \mathcal{O})$
3 $\mathcal{J} \leftarrow \mathcal{J} \cup erklärung$
4 **für jedes** $\alpha \in erklärung$ **tue**
5 $\quad pfad \leftarrow \emptyset$
6 $\quad$ SUCHE_HST$(C, \mathcal{O} \setminus \{\alpha\}, \alpha, pfad)$
7 **zurück** $\mathcal{J}$

Algorithmus 1 : Berechnen von allen Erklärungen als Black-Box Ansatz

Prozedur :EXPANDIERE_HST

Eingabe : Ontologie $\mathcal{O}$, Unerfüllbares Konzept C

Ausgabe : keine Ausgabe - modifiziert globale Variablen $\mathcal{J}, HS$

wenn $pfad \cup \{\alpha\} \subseteq h$ in $h \in HS$ **oder** es existiert ein Präfix-Pfad p für ein $h \in HS$, so dass $p = h$ **dann**
  zurück (**Optimierung 1: vorzeitiges Pfadverlassen)
wenn es existiert eine $erklärung \in \mathcal{J}$, so dass $pfad \cap erklärung = \emptyset$ **dann**
  $erklärung\_neu \leftarrow erklärung$ (**Optimierung 2: wiederverwenden von Erklärungen)
sonst
  $erklärung\_neu \leftarrow$ EINZELNE_ERKLÄRUNG($C, \mathcal{O}$)
wenn $erklärung\_neu \neq \emptyset$ **dann**
  $\mathcal{J} \leftarrow \mathcal{J} \cup erklärung\_neu$
  $pfad\_neu \leftarrow pfad \cup \{\alpha\}$
  für jedes $\beta \in erklärung\_neu$ **tue**
    EXPANDIERE_HST($C, \mathcal{O} \setminus \{\beta\}, \beta, pfad\_neu$)
sonst
  $HS \leftarrow HS \cup \{\alpha\}$

Prozedur `Expandieren eines Pfades im Hitting Set Tree`

Beispiel 6 (Hitting Set Tree)**.** Für die Ontologie

$$\begin{aligned}\mathcal{O} = \{&\alpha_1 = A \sqsubseteq B \sqcap C\\ &\alpha_2 = B \sqsubseteq \neg C \sqcap E\\ &\alpha_3 = C \sqsubseteq \forall r.\neg D \sqcap E \sqcap F\\ &\alpha_4 = E \sqsubseteq \exists r.D\\ &\alpha_5 = F \sqsubseteq \neg B\}\end{aligned}$$

sieht ein möglicher HST dann folgendermaßen aus:

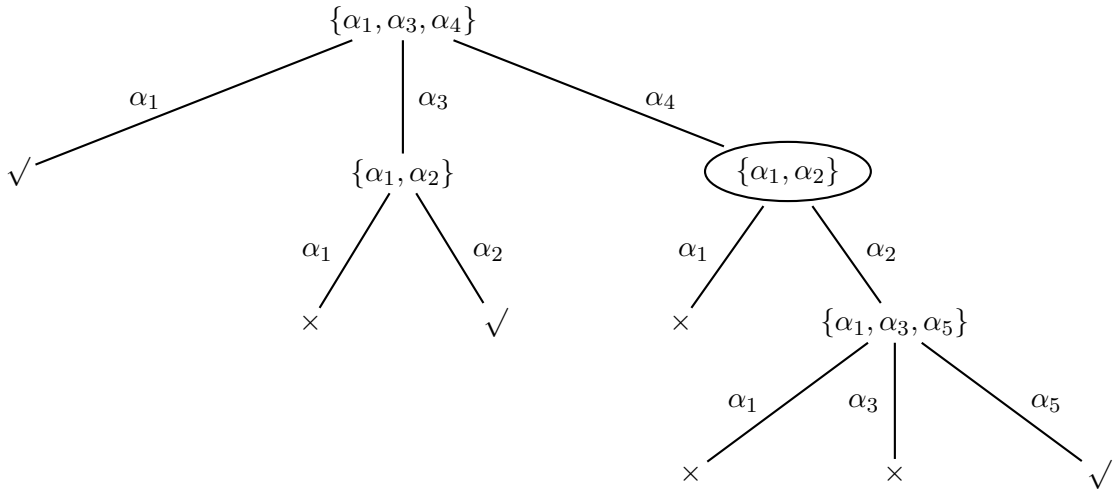


Abb. 6: Beispiel eines Hitting Set Tree

Die wesentliche Komplexität in diesem Algorithmus liegt in der Anzahl der Aufrufe von *EINZELNE_ERKLÄRUNG*. Deshalb wurden in [18] Optimierungen vorgeschlagen, um die Anzahl der Aufrufe zu reduzieren:

Vorzeitiges Beenden eines Pfades Sobald ein HS-Pfad gefunden wurde, ist auch jede Obermenge davon ein HS und weitere Aufrufe von *EINZELNE_ERKLÄRUNG* sind überflüssig. Außerdem kann der aktuelle Pfad vorzeitig verlassen werden, wenn bereits ein Pfad betrachtet wurde, der mit dem aktuellen Pfad begann und somit dort bereits alle möglichen Zweige betrachtet wurden. In Abb. 6 sind diese Optimierungen mit $\times$ markiert.

Wiederverwendung von Erklärungen Weil der aktuelle Pfad eine Representation der von der Ontologie entfernten Axiome ist, kann als neuer Knoten in dem Pfad eine schon gefundene

Erklärung verwendet werden, falls die Menge der Axiome in der Erklärung disjunkt mit der Menge der Axiome im aktuellen Pfad ist. In Abb. 6 sind solche Knoten von einer Ellipse umrandet.

4.4 Berechnen von fein-granularen Erklärungen

Die bisher berechneten Erklärungen bestehen aus Axiomen, die so auch in der Ontologie vorkommen. Das bedeutet die Axiome in den Erklärungen entsprechen Axiomen die in der Ontologie stehen. Dadurch kann es in einer Ontologie mit langen Axiomen dazu kommen, dass in Erklärungen Teile von Axiomen irrelevant für die Folgebeziehung sind. Zum Beispiel ist $\mathcal{J} = \{C \sqsubseteq A \sqcap B, C \sqsubseteq \neg D, A \sqsubseteq D\}$ eine Erklärung für die Unerfüllbarkeit von C, aber der zweite Teil der Konjunktion in $C \sqsubseteq A \sqcap B$ ist dafür eigentlich unbedeutend. Neben der Unterstützung beim Verstehen durch die Fokussierung auf relevante Teile, ist auch der Aspekt der Reparatur zu betrachten. Wenn wir z.B. davon ausgehen dass zur Reparatur das erste Axiom entfernt wird, dann würden mehr Folgebeziehungen verloren gehen, als notwendig. Wir würden damit einen gewissen Grad an „Über-Reparatur“ erreichen.

Es ist also durchaus sinnvoll, einen fein-granularen Ansatz zu verwenden. In der Vergangenheit gab es dafür einige Vorschläge [17][20][21], aber eine genaue Definition für fein-granulare Erklärungen, wurde erst in [11] gegeben, woebi wir diesen ansatz auch in unserem Tool integriert haben.

Gründe und Motivation für fein-granulare Erklärungen sind nach [11] folgende:

1. Ein Axiom in einer Erklärung kann überflüssige Teile enthalten.
2. Eine Erklärung kann relevante Informationen verbergen. So gibt es zum Beispiel für die Unerfüllbarkeit von C in $\mathcal{O} = \{C \sqsubseteq \neg A \sqcap B, C \sqsubseteq A \sqcap \neg B\}$ 2 unterschiedliche Gründe, zum einen $C \sqsubseteq A \sqcap \neg A$ und zum anderen $C \sqsubseteq B \sqcap \neg B$, aber nur eine einzelne Erklärung, die beide Axiome enthält. Dieses Problem wird auch als *interne Maskierung* bezeichnet.
3. Eine Erklärung kann relevante Axiome verdecken. Wenn wir $\mathcal{O} = \{\alpha_1 : C \sqsubseteq A \sqcap \neg A \sqcap B, \alpha_2 : C \sqsubseteq \neg B\}$ betrachten, so gibt es dort für die Unerfüllbarkeit von C nur eine einzige Erklärung, welche aus α_1 besteht, obwohl offensichtlich auch α_2 eine Rolle dabei spielt. Dieses Verhalten wird als *externe Maskierung* bezeichnet.
4. Mehrere Erklärungen können einen fein-granularen Kern verbergen. Sei zum Beispiel $\mathcal{O} = \{\alpha_1 : C \sqsubseteq A \sqcap D, \alpha_2 : C \sqsubseteq A \sqcap E, \alpha_3 : A \sqsubseteq B \sqcap \neg B\}$, dann gibt es für die Unerfüllbarkeit von C genau 2 Erklärungen $\{\alpha_1, \alpha_3\}$ und $\{\alpha_2, \alpha_3\}$, aber offensichtlich nur eine fein-granulare Erklärung, $\{C \sqsubseteq A, A \sqsubseteq \neg B \sqcap B\}$. Neben leichterem Verständis können fein-granulare Erklärungen somit auch zum Finden von Modellierungsfehlern, in unserem Fall Redundanz, beitragen.

In [11] werden 2 Arten von fein-granularen Erklärungen definiert, lakonische Erklärungen und präzise Erklärungen. Lakonische Erklärungen sind intuitiv Erklärungen, die keine überflüssigen Teile enthalten. Präzise Erklärungen werden abgeleitet von lakonischen Erklärungen, und sind Erklärungen in denen die Axiome so flach, klein und semantisch minimal wie möglich sind.

Definition 5 (Axiom Länge)**.** *Seien X und Y Paare von Konzepten oder Rollen, C und D Konzepte, A ein atomares Konzept und R eine Rolle. Die Länge eines Axioms wird wie folgt definiert:*

$$\begin{aligned} |X \sqsubseteq Y| &:= |X| + |Y| \\ |X \equiv Y| &:= 2\left(|X| + |Y|\right) \\ |Sym\left(R\right)| = |Trans\left(R\right)| &:= 1 \end{aligned}$$

mit

$$\begin{aligned} |\top| = |\bot| &:= 0 \\ |A| = |\{i\}| = |R| &:= 1 \\ |\neg C| &:= |C| \\ |C \sqcap D| = |C \sqcup D| &:= |C| + |D| \\ |\exists R.C| = |\forall R.C| = |\geq nR.C| = |\leq nR.C| &:= 1 + |C| \end{aligned}$$

Für die Definition von lakonischen Erklärungen wird zusätzlich eine erfüllbarkeitserhaltende Strukturtransformation $\delta\left(\mathcal{J}\right)$ definiert:

$$\begin{aligned} \delta\left(\mathcal{O}\right) &:= \bigcup\nolimits_{\alpha \in R \cup A} \delta\left(\alpha\right) \cup \bigcup\nolimits_{C_1 \sqsubseteq C_2 \in T} \delta\left(\top \sqsubseteq \text{nnf}\left(\neg C_1 \sqcup C_2\right)\right) \\ \delta\left(D\left(a\right)\right) &:= \delta\left(\top \sqsubseteq \neg\{a\} \sqcup \text{nnf}\left(D\right)\right) \\ \delta\left(\top \sqsubseteq C \sqcup D\right) &:= \delta\left(\top \sqsubseteq A'_D \sqcup C\right) \cup \bigcup\nolimits_{i=1}^{n} \delta\left(A'_D \sqsubseteq D_i\right) \text{ für } D = \prod\nolimits_{i=1}^{n} D_i \\ \delta\left(\top \sqsubseteq C \sqcup \exists R.D\right) &:= \delta\left(\top \sqsubseteq A_D \sqcup C\right) \cup \{A_D \sqsubseteq \exists R.A'_D\} \cup \delta\left(A'_D \sqsubseteq D\right) \\ \delta\left(\top \sqsubseteq C \sqcup \forall R.D\right) &:= \delta\left(\top \sqsubseteq A_D \sqcup C\right) \cup \{A_D \sqsubseteq \forall R.A'_D\} \cup \delta\left(A'_D \sqsubseteq D\right) \\ \delta\left(\top \sqsubseteq C \sqcup \geq nR.D\right) &:= \delta\left(\top \sqsubseteq A_D \sqcup C\right) \cup \{A_D \sqsubseteq \geq nR.A'_D\} \cup \delta\left(A'_D \sqsubseteq D\right) \\ \delta\left(\top \sqsubseteq C \sqcup \leq nR.D\right) &:= \delta\left(\top \sqsubseteq A_D \sqcup C\right) \cup \{A_D \sqsubseteq \leq nR.A'_D\} \cup \delta\left(A'_D \sqsubseteq D\right) \\ \delta\left(A'_D \sqsubseteq D\right) &:= \delta\left(A'_D \sqsubseteq D\right) \text{(wenn } D \text{ die Form } A \text{ oder } \neg A \text{ hat)} \\ \delta\left(A'_D \sqsubseteq D\right) &:= \delta\left(\top \sqsubseteq \neg A'_D \sqcup D\right) \text{(wenn } D \text{ nicht die Form } A \text{ oder } \neg A \text{ hat)} \\ \delta\left(\beta\right) &:= \beta \text{ für jedes andere Axiom} \end{aligned}$$

Diese Transformation δ entfernt alle verschachtelten Beschreibungen, und führt somit zu Axiomen, die so klein und flach wie möglich sind.

Zusammen mit der deduktiven Hülle $\mathcal{O}^\star$ einer Ontologie $\mathcal{O}$ ergibt sich jetzt folgende formale Definition für lakonische und präzise Erklärungen:

Definition 6 (Lakonische Erklärung)**.** *Sei $\mathcal{O}$ eine Ontologie mit $\mathcal{O} \models \eta$. Dann ist $\mathcal{J}$ eine lakonische Erklärung für η in $\mathcal{O}$ wenn gilt:*

1. *$\mathcal{J}$ is eine Erklärung für η in $\mathcal{O}^\star$*
2. *$\delta(\mathcal{J})$ ist eine Erklärung für η in $(\delta(\mathcal{O}))^\star$*
3. *Für jedes $\alpha \in \delta(\mathcal{J})$ existiert kein α' so dass*
   - *(a) $\alpha \models \alpha'$ und $\alpha' \not\models \alpha$*
   - *(b) $|\alpha'| \leq |\alpha|$*
   - *(c) $\delta(\mathcal{J}) \setminus \{\alpha\} \cup \delta(\{\alpha'\})$ ist eine Erklärung für η in $(\delta(\mathcal{O}))^\star$*

Intuitiv ist eine lakonische Erklärung eine Erklärung, die nur Teile von Konzepten enthält und diese Teile so schwach wie möglich sind.

Definition 7 (Präzise Erklärung)**.** *Sei $\mathcal{O}$ eine Ontologie und η eine Folgebeziehung mit $\mathcal{O} \models \eta$. Sei $\mathcal{J}$ eine Erklärung für $\mathcal{O} \models \eta$ und $\mathcal{J}' = \delta(\mathcal{J})$. Dann ist $\mathcal{J}'$ präzise in Bezug auf $\mathcal{J}$, wenn $\mathcal{J}$ eine lakonische Erklärung für $\mathcal{O} \models \eta$ ist.*

Intuitiv ist eine präzise Erklärung eine Ableitung einer lakonischen Erklärung, in der die Axiome so flach, klein und schwach wie möglich sind. Sie ist damit für eine Reparatur geeigneter. Eine präzise Erklärung ist dabei immer als präzise in Bezug auf eine lakonische Erklärung anzusehen - eine Erklärung selbst kann nicht präzise sein.

Das Problem mit der bisherigen Definition einer lakonischen Erklärung ist die Verwendung der deduktiven Hülle $O^\star$. So ist zum Beispiel für die Unerfüllbarkeit von C in $\mathcal{O} = \{C \sqsubseteq D \sqcap \neg C \sqcap E, A \sqsubseteq B\}$ neben $\{C \sqsubseteq D \sqcap \neg D\}$ nach Definition 6 auch $\{B \sqcap \neg B\}$ eine lakonische Erklärung. Zwar ist das nach 6 offensichtlich korrekt, sorgt aber beim Verstehen einer Folgebeziehung für unnötige Verwirrung. Es sollten deshalb nur syntaktisch relevante lakonische Erklärungen berechnet werden, so dass in [11] ein Filter $\mathcal{O}^+$ für die deduktive Hülle eingeführt wurde.

Definition 8 ($\mathcal{O}^+$ Filter)**.** *Sei $\mathcal{O}^+_{\mathcal{SHOIQ}} = \{\alpha' \mid \alpha' \in \sigma(\alpha)$ mit $\alpha \in \mathcal{O}\}$.*

$$
\begin{aligned}
\sigma\left(C_1 \sqcup \cdots \sqcup C_n \sqsubseteq D_1 \sqcap \cdots \sqcap D_m\right) &:= \left\{C_i' \sqsubseteq D_j' \mid C_i' \in \beta\left(C_i\right), D_i' \in \tau\left(D_j\right)\right\} \\
\sigma\left(C \equiv D\right) &:= \sigma\left(C \sqsubseteq D\right) \cup \left(D \sqsubseteq C\right) \cup \{C \equiv D\} \\
\sigma\left(R \sqsubseteq S\right) &:= \{R \sqsubseteq S\} \\
\sigma\left(R \equiv S\right) &:= \{R \sqsubseteq S\} \cup \{S \sqsubseteq R\} \\
\sigma\left(Trans\left(R\right)\right) &:= \{Trans\left(R\right)\} \\
X\left(A\right) &:= \{max_X, A\} \text{ für einen Konzeptnamen } A \text{ oder } \{i\} \\
X\left(C_1 \sqcup \cdots \sqcup C_n\right) &:= \left\{C_1' \sqcup \cdots \sqcup C_n' \mid C_i' \in X\left(C_i\right)\right\} \\
X\left(C_1 \sqcap \cdots \sqcap C_n\right) &:= \left\{C_1' \sqcap \cdots \sqcap C_n' \mid C_i' \in X\left(C_i\right)\right\} \\
X\left(\neg C\right) &:= \left\{\neg C' \mid C' \in \overline{X}\left(C\right)\right\} \\
X\left(\exists R.C\right) &:= \left\{\exists R.C' \mid C' \in X\left(C\right)\right\} \cup \{\top\} \\
X\left(\forall R.C\right) &:= \left\{\forall R.C' \mid C' \in X\left(C\right)\right\} \cup \{\top\} \\
X\left(\geq nR.C\right) &:= \left\{\geq mR.C' \mid C' \in X\left(C\right), n \leq m \leq n'\right\} \cup \{max_X\} \\
\tau\left(\leq nR.C\right) &:= \left\{\leq mR.C' \mid C' \in \beta\left(C\right), 0 \leq m \leq n\right\} \cup \{\top\} \\
\beta\left(\leq nR.C\right) &:= \left\{\leq mR.C' \mid C' \in \tau\left(C\right), n \leq m \leq n'\right\} \cup \{\bot\} \\
X\left(\{j_1 \ldots j_n\}\right) &:= X\left(\{j_1\} \sqcup \cdots \sqcup \{j_n\}\right)
\end{aligned}
$$

Eine wichtige Eigenschaft von $\mathcal{O}^+$ ist die Vollständigkeit, die garantiert dass lakonische Erklärungen für eine einfache Reparatur verwendet werden können, denn eine Abschächung einer Ontologie $\mathcal{O}$ mit $\mathcal{O} \models \eta$ so dass keine der lakonischen Erklärungen für $\mathcal{O} \models \eta$ folgen, bewirkt dass auch η nicht länger aus $\mathcal{O}$ folgt.

Definition 9 ($\mathcal{O}^+$ Vollständigkeit)**.** *Sei $\mathcal{O}$ eine Ontologie, $\mathcal{O}^+$ eine Menge von Axiomen so das $\mathcal{O} \subseteq \mathcal{O}^+ \subseteq \mathcal{O}^\star$, und η eine Folgebeziehung, so dass $\mathcal{O} \models \eta$. $\mathcal{O}^+$ ist vollständig für η und $\mathcal{O}$, wenn für die Menge von allen lakonischen Erklärungen $\mathfrak{J}$ für η in Bezug auf $\mathcal{O}^+$, für jedes $\mathcal{O}'$ mit $\mathcal{O}' \subseteq \mathcal{O}^+$ gilt: wenn $\mathcal{O}' \not\models \mathcal{J}$ für alle $\mathcal{J} \in \mathfrak{J}$, dann $\mathcal{O}' \not\models \eta$.*

Um lakonische Erklärungen berechnen zu können, wurde in [11] Algorithmus 3 vorgeschlagen.

Algorithmus :LAKONISCHE_ERKLÄRUNGEN

Eingabe : Ontologie $\mathcal{O}$, Unerfüllbares Konzept C

Ausgabe : Menge $\mathcal{J}$ von präzisen Erklärungen

$\mathcal{J} \leftarrow$ OPLUS_ERKLÄRUNGEN
für $J \in \mathcal{J}$ **tue**
  wenn *IST_LAKONISCH* $(J, C) = falsch$ **dann**
    $\mathcal{J} = \mathcal{J} \setminus J$
zurück $\mathcal{J}$

Algorithmus 3 : Berechnen von lakonischen Erklärungen

Algorithmus :OPLUS_ERKLÄRUNGEN

Eingabe : Ontologie $\mathcal{O}$, Unerfüllbares Konzept C

Ausgabe : Menge $\mathcal{J}$ von Erklärungen in $\mathcal{O}^+$

$\mathcal{O}' \leftarrow \mathcal{O}$
$\mathcal{J} \leftarrow \emptyset$
$\mathcal{J}' \leftarrow$ ALLE_ERKLÄRUNGEN $(\mathcal{O}', C)$
wiederhole
  $\mathcal{J} \leftarrow \mathcal{J}'$
  für $J \in \mathcal{J}$ **tue**
    $\mathcal{O}' \leftarrow (\mathcal{O}' \setminus J) \cup$ BERECHNE_OPLUS (J)
  $\mathcal{J}' \leftarrow$ ALLE_ERKLÄRUNGEN$(\mathcal{O}', C)$
bis $\mathcal{J} = \mathcal{J}'$
zurück $\mathcal{J}$

Algorithmus 4 : Berechnen von Erklärungen in $\mathcal{O}^+$

Algorithmus :IST_LAKONISCH

Eingabe : Erklärung J, Unerfüllbares Konzept C

Ausgabe : $wahr$ wenn J lakonisch ist, sonst $falsch$

$\mathcal{J} \leftarrow$ ALLE_ERKLÄRUNGEN (BERECHNE_OPLUS $(\delta(J)), C)$
zurück $\mathcal{J} = \{\delta(J)\}$

Algorithmus 5 : Testen ob eine Erklärung lakonisch ist

4.5 Optimierungen

Weil wir uns in unserer Arbeit auf Reasoner mit dem Tableauverfahren beschränken, und diese für ausdrucksstarke Beschreibungslogiken eine exponentielle Komplexität bezüglich der Anzahl der Axiome besitzen, stellt eine Reduzierung dieser eine gute Optimierungsmöglichkeit dar. Ein Ansatz dafür ist das Extrahieren eines Moduls aus einer Ontologie [5].

4.5.1 Modularisierung

Definition 10 (Modul)**.** *Sei L eine Beschreibungslogik, $\mathcal{O}_1 \subseteq \mathcal{O}$ zwei Ontologien ausgedrückt in L und sei **S** eine Signatur. Dann ist $\mathcal{O}_1$ ein **S**-Modul in $\mathcal{O}$ bezüglich L, wenn für jede Ontologie P und jedes Axiom α ausgedrückt in L mit $Sig\,(P \cup \{\alpha\}) \cap Sig\,(\mathcal{O}) \subseteq$ **S** gilt: $P \cup \mathcal{O} \models \alpha$ $gdw.$ $P \cup \mathcal{O}_1 \models \alpha$.*

Vereinfacht gesagt ist ein Modul ein Teil einer Ontologie, das bezüglich einer gegebenen Signatur alle relevanten Informationen enthält.

Es gibt verschiedene Methoden ein solches Modul aus einer Ontologie zu extrahieren. Eine Möglichkeit dafür ist die Verwendung der Eigenschaft der Lokalität.

Definition 11 (Lokalität)**.** *Sei **S** eine Signatur. Ein Axiom α ist lokal bezüglich **S**, wenn jede triviale Expansion von jeder beliebigen **S**-Interpretation zu $\mathbf{S} \cup Sig\,(\alpha)$ ein Modell von α ist. Die Menge aller Axiome, die lokal bezüglich **S** sind, wird mit $lokal\,(\mathbf{S})$ bezeichnet. Eine Ontologie $\mathcal{O}$ ist lokal bezüglich **S**, wenn $\mathcal{O} \subseteq lokal\,(\mathbf{S})$ gilt.*

Intuitiv ist eine Ontologie $\mathcal{O}$ lokal bezüglich einer Signatur **S**, wenn wir jede beliebige Interpretation für die Symbole in **S** nehmen und zu einem Modell von $\mathcal{O}$ erweitern können, so dass die übrigen Symbole als leere Menge interpretiert werden.

Ein Modul basierend auf der Lokalitätseigenschaft wird dann folgendermaßen definiert:

Definition 12 (Module basierend auf Lokalität)**.** *Sei $\mathcal{O}$ eine Ontologie und **S** eine Signatur. Dann ist $\mathcal{O}_1 \subseteq \mathcal{O}$ ein lokalitäts-basiertes **S**-Modul in $\mathcal{O}$, wenn $\mathcal{O} \setminus \mathcal{O}_1$ lokal bezüglich $\mathbf{S} \cup Sig\,(\mathcal{O}_1)$ ist.*

Zum Testen ob ein Axiom α lokal bezüglich einer Signatur **S** ist, genügt es alle atomaren Konzepte und Rollen, die nicht in S vorkommen mit der leeren Menge zu interpretieren, und dann zu testen ob α in allen Interpretationen der verbleibenden Symbole erfüllt ist, d.h. ob α eine Tautologie ist. Ein Vorschlag für eine Transformationsfunktion aus [4] für die Beschreibungslogik $\mathcal{SHOIQ}$ lautet:

$$\tau(\mathcal{O}, \mathbf{S}) ::= \bigcup_{\alpha \in \mathcal{O}} \tau(\alpha, \mathbf{S}) \quad \text{(a)}$$

$$\begin{aligned}
\tau(\alpha, \mathbf{S}) ::= & \tau(C_1 \sqsubseteq C_2, \mathbf{S}) && = (\tau(C_1, \mathbf{S}) \sqsubseteq \tau(C_2, \mathbf{S})) && \text{(b)}\\
& \tau(R_1 \sqsubseteq R_2, \mathbf{S}) && = (\bot \sqsubseteq \bot) \text{ wenn } Sig(R_1) \not\subseteq \mathbf{S}, \text{ sonst} && \\
& && = \exists R_1.\top \sqsubseteq \bot \text{ wenn } Sig(R_2) \not\subseteq \mathbf{S}, \text{ sonst } (R_1 \sqsubseteq R_2) && \text{(c)}\\
& \tau(a : C, \mathbf{S}) && = a : \tau(C, \mathbf{S}) && \text{(d)}\\
& \tau(r(a, b), \mathbf{S}) && = \top \sqsubseteq \bot \text{ wenn } r \notin \mathbf{S}, \text{ sonst } = r(a, b) && \text{(e)}\\
& \tau(Trans(r), \mathbf{S}) && = \bot \sqsubseteq \bot \text{ wenn } r \notin \mathbf{S}, \text{ sonst } = Trans(r) && \text{(f)}\\
& \tau(Funct(R), \mathbf{S}) && = \bot \sqsubseteq \bot \text{ wenn } Sig(R) \not\subseteq \mathbf{S}, \text{ sonst } = Funct(R) && \text{(g)}\\
\tau(C, \mathbf{S}) ::= & \tau(\top, \mathbf{S}) && = \top && \text{(h)}\\
& \tau(A, \mathbf{S}) && = \bot \text{ wenn } A \notin \mathbf{S}, \text{ sonst } = A && \text{(i)}\\
& \tau(\{\alpha\}, \mathbf{S}) && = \{\alpha\} && \text{(j)}\\
& \tau(C_1 \sqcap C_2, \mathbf{S}) && = \tau(C_1, \mathbf{S}) \sqcap \tau(C_2, \mathbf{S}) && \text{(k)}\\
& \tau(\neg C_1, \mathbf{S}) && = \neg\tau(C_1, \mathbf{S}) && \text{(l)}\\
& \tau(\exists R.C_1, \mathbf{S}) && = \bot \text{ wenn } Sig(R) \not\subseteq \mathbf{S}, \text{ sonst } \exists R.\tau(C_1, \mathbf{S}) && \text{(m)}\\
& \tau(\geq n\, R.C_1, \mathbf{S}) && = \bot \text{ wenn } Sig(R) \not\subseteq \mathbf{S}, \text{ sonst } \geq n\, R.\tau(C_1, \mathbf{S}) && \text{(n)}\\
& && && \text{(o)}
\end{aligned}$$

wobei $\mathbf{S}$ eine $\mathcal{SHOIQ}$ Signatur, C ein Konzept, α ein Axiom und $\mathcal{O}$ eine Ontologie bezeichet. Vereinfacht ausgedrückt ist also folgendes zu tun: für alle atomaren Konzepte A und alle atomaren Rollen r mit $A, r \notin \mathbf{S}$, sowie alle R die eine Rolle r oder r^- mit $r \notin \mathbf{S}$ sind: (1) ersetze alle Konzepte der Form $A, \exists R.C$ oder $\geq n\, R.C$ mit $\bot$; (2) entferne alle Transitivitätsaxiome $Trans(r)$; (3) ersetzte alle Zuweisungen $a : A$ und $r(a, b)$ mit dem Kontradiktionsaxiom $\top \subseteq \bot$.

Beispiel 7. Folgendes Axiom α ist zum Beispiel lokal bezüglich der Signatur $\mathbf{S}_1 = \{Pizza, Salami\}$, denn die Transformation $\tau(\alpha, \mathbf{S}_1)$ führt zu folgenden Ersetzungen

$$\alpha : \underbrace{SalamiPizza}_{\bot\text{[nach (i)]}} \equiv Pizza \sqcap \underbrace{\exists hatBelag.Salami}_{\bot\text{[nach (m)]}}$$

was $\bot \equiv Pizza \sqcap \bot$ ergibt, und offensichtlich eine Tautologie ist. In Bezug auf die Signatur $\mathbf{S}_2 = \{SalamiPizza, hatBelag\}$ ist α jedoch nicht lokal, denn $\tau(\alpha, \mathbf{S}_2)$ ergibt

$$\alpha : SalamiPizza \equiv \underbrace{Pizza}_{\bot\text{[nach (i)]}} \sqcap \exists hatBelag.\underbrace{Salami}_{\bot\text{[nach (i)]}}$$

und $SalamiPizza \equiv \bot \sqcap \exists hatBelag.\bot$ ist keine Tautologie.

Obwohl die aktuellen Reasoner für Aufgaben wie das Testen auf Tautologien auch für ausdrucksstarke Beschreibungslogiken wie $\mathcal{SHOIQ}$ optimiert sind, kann es durchaus Fälle geben, in denen solche Operationen zu aufwändig sind. Deshalb existiert eine weitere Möglichkeit Axiome auf Lokalität zu überprüfen, das Testen auf *syntaktische* Lokalität. Dabei wird im Gegensatz zur vorhergehenden $semantischen$ Vorgehensweise, ein approximativer Test der Lokalitätsbedingungen angewendet.

Definition 13 (Syntaktische Lokalität für $\mathcal{SHOIQ}$)**.** *Sei* **S** *eine Signatur. Die folgende Grammatik definiert rekursiv zwei Mengen von Konzepten* $\text{Con}^{\bot}$**(S)** *und* $\text{Con}^{\top}$**(S)** *für eine Signatur* **S***:*

$$\begin{aligned} \text{Con}^{\bot}(\boldsymbol{S}) :: &= A^{\bot} \mid (\neg C^{\top}) \mid (C \sqcap C^{\bot}) \mid (\exists r^{\bot}.C) \mid (\exists r.C^{\bot}) \mid (\geq n\, r^{\bot}.C) \mid (\geq n\, r.C^{\bot}) \\ \text{Con}^{\top}(\boldsymbol{S}) :: &= (\neg C^{\bot}) \mid (C_1^{\top} \sqcap C_2^{\top}) \end{aligned}$$

mit $A^{\bot} \notin$ ***S*** *ist ein atomares Konzept,* C *ist ein Konzept,* $C^{\bot} \in \text{Con}^{\bot}(\boldsymbol{S})$*,* $C_i^{\top} \in \text{Con}^{\top}(\boldsymbol{S})$ (*für* $i = 1, 2$)*, und* $Sig\left(r^{\bot}\right) \not\subseteq$ ***S****.*
Ein Axiom α *ist syntaktisch lokal bezüglich* ***S****, wenn es eine der folgenden Formen besitzt:* $(1)\, r^{\bot} \sqsubseteq r, (2)\, Trans\left(r^{\bot}\right), (3)\, C^{\bot} \sqsubseteq C\, oder\ (4)\, C \sqsubseteq C^{\top}$*. Die Menge aller Axiome, die lokal bezüglich* ***S*** *sind, wird mit* $s\_lokal\,(\boldsymbol{S})$ *bezeichnet. Eine Ontologie* $\mathcal{O}$ *ist lokal bezüglich* ***S****, wenn* $\mathcal{O} \subseteq s\_lokal\,(\boldsymbol{S})$ *gilt.*

Intuitiv werden alle Symbole $A^{\bot}$ und $r^{\bot}$, die sich nicht in **S** befinden, mit dem leeren Konzept bzw. der leeren Rolle $\bot$ ersetzt, so dass sich in $Con^{\bot}(\mathbf{S})$ $\big($resp. $Con^{\top}(\mathbf{S})\big)$ alle Konzepte, die äquivalent zu $\bot$ $(\top)$ sind, befinden.

Beispiel 8. Folgendes Axiom ist nach Def. 13 (3) zum Beispiel syntaktisch lokal bezüglich der Signatur $\mathbf{S} = \{Pizza, hatBelag\}$, denn nach dem Ersetzen der Symbole die nicht in S enthalten sind durch $\bot$, erhalten wir eine Tautologie $\bot \equiv \bot$:

$$\underbrace{SalamiPizza}_{\bot} \equiv Pizza \sqcap \exists\, hatBelag.\underbrace{\underbrace{Salami}_{\bot}}_{\bot}$$

Dass die Extraktion von Modulen als Optimierung für das Berechnen von Erklärungen anwendbar ist, und lokalitäts-basierte Module auch alle Axiome aus den Erklärungen enthalten, wurde formal bereits in [38] bewiesen. Es wird dort gezeigt, dass ein lokalitäts-basiertes Modul ein starkes Subsumptions-Modul ist.

Definition 14 (Starkes Subsumptions-Modul)**.** *Seien* $S \subseteq \mathcal{O}$ $\mathcal{SHOIQ}$ *Ontologien, und* A *ein Konzeptname. Dann ist* S *ein Subsumptions-Modul für* A*, wenn für alle* $B \in$ CN *gilt:* $A \sqsubseteq_{\mathcal{O}} B \iff A \sqsubseteq_S B$*.*

Ein Subsumptions-Modul S für A in $\mathcal{O}$ wird stark genannt, wenn für alle $B \in$ CN gilt: $A \sqsubseteq_{\mathcal{O}} B$ impliziert dass $\mathcal{J} \subseteq S$, für alle Erklärungen $\mathcal{J}$ für $A \sqsubseteq B$ in $\mathcal{O}$.

Außerdem wird dort auch die Effizienz dieses Ansatzes gezeigt, in dem bei einigen empirischen Messungen eine Verbesserung bei der Berechnung von Erklärungen von mehreren Größenordnungen festgestellt wurde.

4.6 Unterstützung bei der Reparatur

Wir haben bisher gezeigt, wie man eine oder mehrere Erklärungen berechnen kann, sowie einen feingranularen Ansatz dargestellt, der sowohl bei dem Verständnis als auch bei der Reparatur hilfreich sein kann. Weil die Konzepte in Ontologien sinnvollerweise eng miteinander in Beziehung stehen, kann es durchaus sein, dass mehr als nur ein Konzept unerfüllbar ist, und die Unerfüllbarkeit eines Konzeptes evtl. von der Unerfüllbarkeit anderer induziert wird. Bei einer Reparatur wird der Nutzer also auf folgende mögliche Probleme treffen, die unterschiedlich schwer zu lösen scheinen:

1. Eine kleine Anzahl von kleinen Erklärungen.
2. Eine kleine Anzahl von großen Erklärungen.
3. Eine große Anzahl von Erklärungen.

Ist der erste Fall meist noch ohne Probleme lösbar, so bereiten vor große und viele Erklärungen mitunter Probleme. Es scheint also hilfreich den Nutzer hierbei besonders zu unterstützen, und den Fokus des Nutzers zum einen auf die möglicherweise wichtigeren Axiome innerhalb einer Erklärung zu lenken, als auch auf die

4.6.1 Unerfüllbare Wurzelkonzepte und abgeleitete unerfüllbare Konzepte

Viele Erklärungen treten oft immer dann auf, wenn es viele unerfüllbare Konzepte gibt. Dabei kann es sein, dass die Unerfüllbarkeit des einen Konzeptes durch die Unerfüllbarkeit anderer Konzepte verursacht wird. Es bietet sich also an [16], zwischen unerfüllbaren Wurzelkonzepten und abgeleiteten unerfüllbaren Konzepten zu unterscheiden, und den Fokus bei einer Reparatur zuerst auf die Wurzelkonzepte zu legen, da durch deren Reparatur auch davon abgeleitete unerfüllbare Konzepte repariert werden können. Intuitiv sind abgeleitete unerfüllbare Konzepte, die dessen Unerfüllbarkeit von anderen Konzepten abhängt und unerfüllbare Wurzelkonzepte alle anderen. Etwas genauer bietet sich eine Definiton mit Hilfe von Erklärungen an:

- Ein Konzept ist ein abgeleitetes unerfüllbares Konzept, wenn es eine Erklärung besitzt, die echte Obermenge einer Erklärung für andere unerfüllbare Konzepte ist.
- Alle anderen unerfüllbaren Konzepte sind unerfüllbare Wurzelkonzepte.

Das Aufteilen von Wurzelkonzepten und abgeleiteten Konzepten ist nicht ohne weiteres möglich, denn man kann nicht einfach einen Reasoner verwenden um zu ermitteln, welche der unerfüllbaren Konzepte Subkonzepte von anderen sind. Ursache dafür ist die Equivalenz jedes unerfüllbaren Konzeptes zum Bottom-Konzept, wodurch alle unerfüllbaren Konzepte untereinander equivalent sind. Es gibt jedoch zwei unterschiedliche Vorgehensweisen, um eine solche Aufteilung vorzunehmen:

1. Berechnen aller Erklärungen für jedes unerfüllbare Konzept, und danach wie in Definition zwischen abgeleiteten und Wurzelkonzepten unterscheiden.
2. Eine Kombination aus einer strukturelle Analyse, d.h. Abhängigkeiten anhand der Axiome erkennen, und einer Heuristik, die Hauptfehlerursachen wie Negation transformiert.

Weil die erste Methode vor allem bei vielen unerfüllbaren Konzepten und vielen Erklärungen schnell ineffizient wird, beschränken wir uns hier auf Ansatz 2, den wir auch in unserem Tool verwenden. Dieser besteht nach [16] aus zwei Teilen:

Finden von Abhängigkeiten - Strukturelle Analyse
Bei der strukturellen Analyse wird versucht, mit Hilfe der vorhandenen Axiome in der Ontologie, Abhängigkeiten zwischen unerfüllbaren Konzepten zu finden. Dafür kann man folgende Regeln anwenden nach denen A ein abgeleitetes unerfüllbares Konzept ist wenn:

$A \sqsubseteq B \in \mathcal{O},$ und Konzept B ist unerfüllbar
$A \sqsubseteq C_1 \sqcap C_2 \sqcap \cdots \sqcap C_n \in \mathcal{O},$ und ein beliebiges Konzept $C_i\,(1 \leq i \leq n)$ ist unerfüllbar
$A \sqsubseteq C_1 \sqcup C_2 \sqcup \cdots \sqcup C_n \in \mathcal{O},$ und alle Konzepte $C_i\,(1 \leq i \leq n)$ sind unerfüllbar
$A \sqsubseteq \exists R.B \in \mathcal{O},$ und B ist unerfüllbar
$A \sqsubseteq \forall R.B, A \sqsubseteq \geq nR\,(\text{oder } A \sqsubseteq \exists R) \in \mathcal{O},$ und B ist unerfüllbar
$A \sqsubseteq \geq nR\,(\text{oder } A \sqsubseteq \exists R)\,, domain\,(R) = B \in \mathcal{O},$ und B ist unerfüllbar
$A \sqsubseteq \geq nR\,(\text{oder } A \sqsubseteq \exists R)\,, range\left(R^{-}\right) = B \in \mathcal{O},$ und B ist unerfüllbar

Anzumerken ist, dass dieser Ansatz nicht alle Abhängigkeiten findet (Transitivität oder Nominale werden z.B. nicht beachtet), aber dafür die gefundenen Abhängigkeiten korrekt sind. Außerdem

werden hierbei keine inferrierten Abhängigkeiten sichtbar, denn zum Beispiel für $A \equiv\geq 1R$ und $B \equiv\geq 2R$ kann ein Reasoner im Normalfall $B \sqsubseteq A$ folgern. Da aber das schon erwähnte Problem mit unerfüllbaren Konzepten (jedes unerfüllbare Konzept wird von allem subsumiert) gilt, ist dafür ein alternatives Verfahren zu verwenden.

Finden von inferrierten Abhängigkeiten
Hierbei wird das Problem mit den unerfüllbaren Konzepten umgangen, indem eine Subsumtions-erhaltende Transformation der gegebenen TBox vorgenommen wird. Dabei wird versucht alle Inkonsistenzen in der TBox zu beseitigen, und trotzdem die Subsumtionshierarchie soweit wie möglich zu erhalten. Die Transformation der TBox besteht dabei lediglich aus zwei voneinander unabhängigen Transformationsregeln, welche zwei der gängisten Ursachen für die Unerfüllbarkeit von Konzepten entfernen:

1. Ersetze alle Vorkommen der Form $\neg C$ in den Axiomen durch ein neues atomares Konzept C'.
2. Ersetzte alle Vorkommen der Form $\leq nr$ in den Axiomen der TBox durch ein neues atomares Konzept R'.

Um einen möglichst vollständigen Erhalt der Konzepthierarchie zu gewährleisten, wird jede Ersetzung in einem Cache gespeichert, und nach jeder Ersetzung mit Hilfe eines Reasoners die Subsumptionen zwischen erfüllbaren Konzepten in der originalen Wissensbasis getestet, und wenn vorhanden zugehörige Beziehungen in der transformierten Wissensbasis hinzugefügt. Dieser Ablauf sichert einen soweit wie möglichen Erhalt der Hierarchie in der transformierten Wissensbasis. Nach der Transformation kann dann mit einem Reasoner auf Abhängigkeiten zwischen unerfüllbaren Konzepten getestet werden, die nicht durch die strukturelle Analyse gefunden wurden.

4.6.2 Metriken für Axiome

Wenn man eine oder mehrere Erklärungen gegeben hat, stellt sich für das Ziel einer Reparatur die Frage, welche der in den Erklärungen enthaltenen Axiome zum Entfernen oder Ändern ausgewählt werden sollen. Deshalb bieten sich hier Metriken an, mit deren Hilfe man eine Rangfolge der Axiome vornehmen kann. Für ein solches Ranking kann man sehr unterschiedliche Maße verwenden, wobei wir in unserer Arbeit nur einige nach [19] verwendet haben, und deshalb im Folgenden kurz darauf eingehen werden:

Frequenz als ein Maß für die Häufigkeit des Vorkommens eines Axioms in unterschiedlichen Erklärungen. Wenn ein Axiom in n Erklärungen für möglicherweise jeweils unterschiedliche

unerfüllbare Konzepte vorkommt, kann ein Entfernen dieses Axioms bis zu n Konzepte reparieren, d.h. erfüllbar machen. Hier gilt im Allgemeinen, dass Axiome mit höherer Frequenz eher zu entfernen sind.

Syntaktische Relevanz für die Ontologie, bezogen auf die Verwendung der Entitäten in der Signatur des Axioms in anderen Axiomen der Ontologie. Die Idee ist dabei, dass eine häufige Verwendung der Entitäten des Axioms in anderen Axiomen darauf schließen lässt, dass diese Entitäten eine möglicherweise zentrale Rolle innerhalb der Ontologie spielen. Ein Entfernen oder Ändern derartiger sich auf die Entitäten beziehenden Axiome könnte daher unbeabsichtigt sein. Als Richtlinie gilt hier, dass Axiome mit geringer syntaktischer Relevanz bevorzugt zu entfernen sind.

Semantische Relevanz für die Ontologie, bezogen auf die Auswirkungen wenn ein Axiom entfernt oder geändert wurde. Dazu gehören sowohl verloren gegangene als auch neu gewonnene logische Folgebeziehungen. Je niedriger für ein Axiom der Wert der semantischen Relevanz ist, um so eher sollte man es entfernen.

Neben den einzelnen Metriken, kann man zusätzlich noch ein aggregiertes, nach einzelnen Kriterien gewichtetes Maß aus diesen angeben. Trotz allem muss klar sein, dass diese Metriken nur als Hilfsmittel dienen, und ein Handeln nach den Kriterien keinesfalls als feste Regel anzusehen ist.

4.7 Schwierigkeiten in inkonsistenten Ontologien

Inkonsistenz in Ontologien kann aus unterschiedlichen Gründen entstehen, wie z.B. durch:

1. Inkonsistenz durch die Zuordnung von Individuen. Wenn ein Individuum a z.B. zu einer Max-Kardinalitätsrestriktion $C \equiv\leq nR$ gehört ($C(a)$), aber zu mehr als n unterschiedlichen Individuen in der dieser Relation R steht, oder wenn a z.B. zwei disjunkten Klassen zugeordnet ist, dann ist diese Ontologie inkonsistent.

2. Inkonsistenz durch die Zuordnung von Individuen zu unerfüllbaren Konzepten oder Rollen. Wenn ein Individuum a z.B. zu einem unerfüllbaren Konzept C gehört oder in einer unerfüllbare Rolle R in Relation $R(a, b)$ steht, ist diese Ontologie inkonsistent.

3. Neben den ABox Axiomen, kann es auch durch Nominale in der TBox zu Inkonsistenzen kommen, wenn z.B. zwei Nominale ein gemeinsames Individuum enthalten, die Nominale selbst aber als disjunkt definiert sind.

Während die bisher vorgestelleten Verfahren und Optimierungsmöglichkeiten sehr gut für unerfüllbare Konzepte funktionieren, so ist es bei inkonsistenten Ontologie oftmals schwieriger und es treten einige einige Probleme auf: Probleme:

- Nach [12] ist die Anzahl der Erklärungen für inkonsistente Ontologien oft deutlich höher als für die Unerfüllbarkeit von Konzepten. Weil der Algorithmus für das Finden von allen Erklärungen im Worst Case exponentielle Komplexität hinsichtlich der Anzahl der Erklärungen besitzt, kann die Zeit für eine solche Berechnung im Vergleich zur der Erklärung von der Unerfüllbarkeit von Konzepten deutlich höher sein. Allerdings kann es durchaus ausreichen einige wenige Erklärungen zu berechnen, und somit einen Einblick in eventuelle Kernprobleme zu erhalten. erkennen)
- Bei der Black-Box Methode4.2.1 stellt sich die Frage mit welchen Axiomen man bei der Expansionsphase beginnen soll. Die meisten Reasoner geben keine Auskunft warum die Ontologie inkonsistent ist (die Ausnahme ist hier Pellet). Während man bei unerfüllbaren Konzepten einfach mit Axiomen, in deren Signatur das Konzept enthalten ist, beginnt, scheint es eine einfache, aber nicht unbedingt effiziente Möglichkeit in der Expansionsphase alle Axiome zu übernehmen.
- Bei inkonsistenten Ontologien kann man keine Unterscheidung zwischen Wurzel- und abgeleiteten Konzepten, wie in 4.6.1 beschrieben, machen. Allerdings könnte man, falls eine Inkonsistenz nach Fall 2 vorliegt, zunächst alle ABox-Axiome entfernen, und danach die Algorithmen aus 4.6.1 anwenden. So hätte man einen Einblick in mögliche Kernursachen für die Inkonsistenz.
- Man kann keine lokalitäts-basierten Module extrahieren. Für das Axiom $\top \sqsubseteq \bot$ würde immer die ganze Ontologie extrahiert werden.

4.8 Reparatur

Nach dem Finden von Fehlern durch Reasoner und dem Berechnen von Erklärungen durch die bereits vorgestellten Verfahren (beides kann weitgehend automatisch getan werden), bleibt es in der Regel im letzten Schritt dem Nutzer überlassen, welche Axiome in den Erklärungen möglicherweise inkorrekt sind. Intuitiv gibt es zwei Möglichkeiten, um diese Fehler zu beheben:

1. Man kann das ausgewählte Axiom entfernen. Das hat den Vorteil, dass wegen der Monotonie von Beschreibungslogiken und OWL keine neuen Folgebeziehungen, und somit auch keine neuen Inkonsistenzen entstehen können. Nachteil bei diesem Schritt kann es allerdings sein dass mögliche relevante Informationen verloren gehen, oder nur Teile vom Axiom falsch sind.

2. Man kann das ausgewählte Axiom so umschreiben, dass der Konflikt aufgelöst wird. Der Nachteil bei dieser Methode ist allerdings, dass neue Folgebeziehungen und damit möglicherweise auch Inkonsistenzen entstehen können.

Um den Nutzer bei der Ausführung zu unterstützen, kann man hier zum einen durch fein-granulare Erklärungen (4.4) mit minimalen Axiomen so wenig Änderungen wie möglich gewährleisten, und zum anderen durch das Berechnen und Anzeigen von möglichen verlorenen und neuen Informationen (4.6.2) eine Reaktion darauf gestatten. newnnn

5 Erweiterung

In diesem Kapitel geht es um die Erweiterung von Ontologien, d.h. das Anreichern des Schemas um neue Axiome. Wir werden dabei zunächst darauf eingehen warum und in welchen Fällen das sinnvoll ist. Eine Möglichkeit dies (semi)-automatisch zu tun, ist es mit Hilfe von maschinellen Lernalgorithmen Vorschläge für Klassenbeschreibungen zu generieren, und dem Nutzer den (oder die) für ihn relevanten davon auswählen zu lassen. Darauf werden wir hier etwas genauer eingehen, in dem wir speziell das Lernproblem für Klassen beschreiben, dessen Ziel es ist, Klassenbeschreibungen zu finden, die genau die Individuen enthalten, die in der zu beschreibenden Klasse enthalten sind. Zur Lösung dieses Problem widmen wir uns danach dem einen dafür im DL-Learner 2.6 integrierten und unserem Tool verwendeten Lernalgorithmus. Weil es oft keine Lösungen für das Lernproblem gibt (das bedeutet es werden nicht genau die Individuen von der zu lernenden Klasse abgedeckt), besteht die Möglichkeit dafür Hilfestellungen zu geben, um Schema und Instanzdaten zueinander konsistent (wir meinen hiermit nicht die logische Konsistenz) zu halten. Mit diesem Problem und dafür einfachen Lösungen beschäftigen wir uns abschließend in diesem Kapitel.

Durch das immer stärker aufkommende Semantic Web gibt es auch immer mehr Wissensbasen und Anwender. Durch immer komplexere Ontologien und immer größere Mengen an Instanzdaten ist es oft schwierig beides in Einklang zu halten. Es gibt prinzipiell zwei grundlegende Ansätze zum Erstellen von Ontologien. Zum einen gibt es den Top-Down-Ansatz, bei dem zunächst das Schema erstellt wird, und danach die Instanzen passend integriert werden können. Die andere Variante ist der sogenannte Bottom-Up-Ansatz. Hierbei wird ausgehend von den Instanzen versucht ein valides Schema zu erstellen. Dieses Vorgehen wird häufig bei der Extraktion aus Datenbanken oder Texten angewendet.

Um die Erstellung (hier vor allem beim Bottom-Up-Ansatz) und Wartung von Ontologien zu unterstützen, und die Qualität des Schemas zu erhöhen, wird in [26] eine semi-automatische Methode zum Lernen von Klassenbeschreibungen vorgeschlagen. Dabei wird ein maschineller Lernalgorithmus verwendet, der dem Nutzer Vorschläge für Klassenbeschreibungen macht. Der Nutzer kann dann entscheiden welcher der Vorschläge für ihn relevant ist, und diese Information zur Ontologie hinzufügen. Das Verwenden von maschinellem Lernen statt der manuellen Erstellung des Schemas hat zwei grundlegende Vorteile[26]:

- Die vorgeschlagenen Klassenbeschreibungen stimmen mit den Instanzdaten überein.
- Das Auswählen von Vorschlägen, statt der manuellen Analyse der Struktur ist wesentlich einfacher.

Beispiel 9. Stellen wir uns einmal eine Wissensbasis mit der Domäne Familie vor, in dem es u.a. Klassen wie $Vater$, $Mutter$, $Frau$, $Mann$ und Rollen wie $hatKind$ gibt. Außerdem sind

eine Menge von Fakten über Individuen vorhanden, so könnte z.B. das Individum tom zu den Klassen $Vater$ und $Mann$, und $anna$ zur Klasse $Frau$ gehören. Der Lernalgorithmus könnte dann beispielsweise folgende Beschreibungen für die Klassen $Vater$ vorschlagen:

$$Mann \sqcap \exists hatKind.\top \tag{1}$$

$$Mann \sqcap \exists hatKind.Frau \tag{2}$$

$$Mann \sqcap \exists hatKind.Mann \tag{3}$$

Der Nutzer könnte sich jetzt für einen der drei Vorschläge entscheiden, und für (1) z.B. das Axiom $Vater \sqsubseteq Mann \sqcap \exists hatKind.\top$ zur Wissensbasis hinzufügen.

5.1 Lernproblem für Klassen

Das Lernen von Klassenbeschreibungen ist im Prinzip nichts anderes als das Lösen eines Lernproblems. Ziel ist es dabei eine Klassenbeschreibung zu finden, in der genau die Individuen enthalten sind, die Instanzen zu beschreibenden Klasse sind. In [26] wird das Lernproblem für Klassen wie folgt definiert:

Definition 15 (Lernproblem für Klassen)**.** *Sei A eine Klasse und $\mathcal{O}$ eine Ontologie. Das Lernproblem für Klassen ist das Finden einer Klassenbeschreibung C so dass $R_{\mathcal{O}}(C) = R_{\mathcal{O}}(A)$.*

Oftmals existieren keine Lösungen des Lernproblems, aber annähernd perfekte Lösungen. So könnten die gelernten Klassenbeschreibungen entweder nicht alle Individuen abdecken, die in der zu beschreibende Klasse enthalten sind, oder sie decken noch zusätzliche Individuen ab. Möglich sind selbstverständlich auch Fälle in denen beides zutrifft. Diese Vorschläge können aus der Sicht des Anwenders trotzdem korrekt bzw. sinnvoll sein. So ist z.B. vorstellbar, dass der Nutzer sich für den ersten Vorschlag aus Beispiel 9 entscheidet, obwohl es ein Individuum in der Klasse $Vater$ gibt, welches (als Subjekt) mit keinem anderen Individuum in der Relation $hatKind$ steht. Hier kann es sein, dass dieses Individuum fälschlicherweise der Klasse $Vater$ zugeordnet wurde, oder dass die Information über mögliche $hatKind$-Beziehungen fehlt. Sie könnten z.B. vergessen worden sein, oder sie sind zur Zeit unbekannt.

5.2 Lernalgorithmus CELOE

Das Lernen von Klassenbeschreibungen ist vergleichbar mit dem Suchen (Generieren) von Klassenbeschreibungen, die dabei auf ihre Genauigkeit bei der Abdeckung der Individuen von der zu beschreibenden Klasse getestet werden. Der im DL-Learner integrierte Lernalgorithmus CELOE (Class Expression Learning for Ontology Engineering) ist ein Algorithmus, der unter Verwendung

von einem Refinement-Operator[27; 28; 29] versucht eine Lösung für das Lernproblem (für Klassen) zu finden.

Definition 16 (Refinement-Operator)**.** *Eine Quasiordnung ist eine reflexive und transitive Relation. In einem quasi-geordneten Raum* $(S, \preceq)$ *ist ein abwärts (aufwärts) Refinement-Operator* ρ *eine Abbildung von* S *auf* 2^S*, so dass für alle* $C \in S$ *gilt:* $C' \in \rho(C)$ *impliziert* $C' \preceq C\,(C \preceq C')$*.* C' *wird dann als Spezialisierung (Generalisierung) von* C *bezeichnet.*

Unter Verwendung von einem sogenannten „Generate and Test"-Ansatz beim Lernen von Klassen, ergibt sich im DL-Learner folgender Ablauf (dargestellt in Abb. 7):
Es werden ausgehend von der zu beschreibenden Klasse eine Vielzahl von Klassenbeschreibungen generiert, wobei dafür das gegebene Hintergrundwissen verwendet wird. Diese werden jeweils hinsichtlich ihrer Qualität durch Heuristiken bewertet, dass bedeutet es wird überprüft wie gut sie als Lösung für das Lernproblem sind. Für die Qualitätsmessungen wird ein Reasoner verwendet, um zu testen wieviele Individuen der zu beschreibenden Klasse abdeckt werden, und wieviele zusätzlich. Dieses Verfahren wird solange wiederholt, bis hinreichend gute Lösungen gefunden wurden oder eine gegebene Zeit überschritten wurde. Bei CELOE wird ein Top-Down Ansatz verwendet, das bedeutet

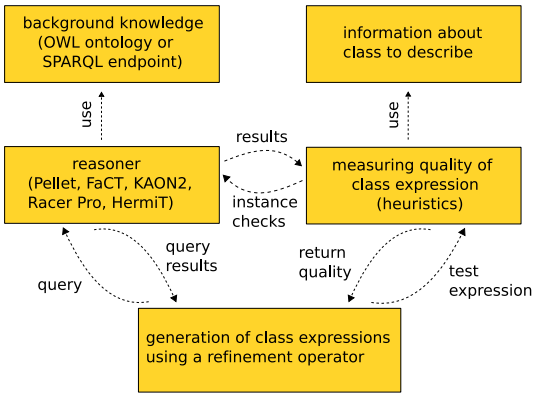


Abb. 7: Überblick über den Ablauf und Zusammenhänge einzelner Komponenten beim Lernen mit CELOE.[22]

beim Lernen von Klassenbeschreibungen werden ausgehend von der allgemeinsten Klasse $\top$ durch Anwendung des Refinement-Operators (abwärts) speziellere (komplexe) Klassen generiert, die dann selbst wieder spezialisiert werden können. Durch wiederholtes Anwenden entsteht somit ein Baum. Die Knoten in dem Baum werden auf Grundlage von Heuristiken[26] bewertet, so dass Knoten, die bestimmte Kriterien nicht erfüllen, nicht weiter expandiert werden müssen. Die Bewertung der

[22]Quelle: `http://dl-learner.svn.sourceforge.net/viewvc/dl-learner/trunk/resources/architecture_celoe.eps?revision=1855`

Knoten (bzw. Klassenbeschreibungen) hat neben für den Algorithmus wichtigen Kriterium welcher Knoten (als nächstes) expandiert werden soll, noch einen weiteren Nutzen. Die Bewertung der Klassenbeschreibungen kann den Nutzer bei der Auswahl helfen, weil er somit sehen kann, wie gut die Klassenbeschreibung mit den Instanzdaten korrespondiert. In CELOE fließen in die Bewertung im wesentlichen drei Kriterien ein (sei A die zu beschreibende Klasse, C die gelernte Beschreibung):

- Der Anteil der Individuen von A, die von C abgedeckt werden (recall).
- Der Anteil der Individuen von C, die zu A gehören (precision).
- Die Länge der Beschreibung (Länge hier als die Summe der Anzahl der Konzepte, Rollen, Quantoren und Verknüpfungssymbole). Weil kurze Ausdrücke i.A. einfacher zu lesen sind, werden lange Ausdrücke bestraft.

Für die genauen Details, wie aus diesen dann der Wert für die Klassenbeschreibung gebildet wird, verweisen wir auf [26].

Weil das Berechnen der Genauigkeit vor allem durch die Verwendung von Retrieval-Anfragen sehr teuer für große Wissensbasen sein kann, und der Lernalgorithmus im Normalfall mehrere tausend Beschreibungen testet, wurden in [26] Optimierungen vorgestellt:

- Weil bekannt ist, dass jede Klassenbeschreibung, die für eine Klasse A getestet wird, auf jeden Fall eine Subklasse der Superklassen von A ist, kann statt vom $\top$-Konzept ausgehend von den Superklassen von A begonnen werden.
- Bei CELOE wird ein im DL-Learner integrierter approximativer Reasoner verwendet. Dieser verwendet eine eigene Prozedur für Instanz-Tests. Dafür wird zunächst die Wissensbasis dematerialisiert, d.h. mit Hilfe eines Standard OWL Reasoners werden für jede atomare Klasse die Instanzen und für jede Property die in Relation stehenden Individuen berechnet. Danach können Instanz-Tests auf diesem Wissen vorgenommen werden. Das ist möglich weil beim Lernen davon ausgegangen werden kann, dass sich die Wissenbasis in dieser Zeit nicht verändert. Ein zusätzlicher Vorteil von dieser Optimierung ist der Aufbau einer Closed-Word-Semantik. Wir hatten bereits in 2.5 erwähnt, dass in OWL die Open World Asssumption (OWA) gilt. Dadurch wäre der Lernalgorithmus aber nicht in der Lage, Beschreibungen zu generieren und gut zu bewerten, die z.B. universelle Restriktionen enthalten ($\forall r.C$), weil ein Instanz-Test dafür mit einem Standard OWL Reasoner in der Regel negativ ausfallen würde. Mit dem Reasoner im DL-Learner sind solche Beschreibungen als Ergebnis jedoch möglich.
- Um die Zahl der Instanz-Tests weiter zu reduzieren, wird die Berechnung der Genauigkeit approximiert. (s. [26])

Beispiel 10 (Generierung von Konzepten durch Refinement-Operator)**.** Für das Konzept $Vater$ aus Beispiel 9 könnte zum Beispiel folgender Pfad im Suchbaum durch wiederholtes Anwenden des abwärts-Refinement-Operators entstehen:

$$\top \longrightarrow Person \longrightarrow Mann \longrightarrow Mann \sqcap \exists hatKind.\top \longrightarrow Mann \sqcap \exists hatKind.Person$$

Ein Hinzufügen von Axiomen aus den gelernten Klassenbeschreibungen kann unter Umständen auch zu einer inkonsistenten Ontologie führen. Das muss allerdings nicht unbedingt negativ sein, denn falls das Axiom korrekt ist, so war die Inkonsistenz schon vorher vorhanden, nur nicht erkennbar („hidden inconsistencies“[26]).

5.3 Reparatur

Wie bereits erwähnt sind die (ausgewählten) Vorschläge beim Lernen nicht immer Lösungen, die zu 100% korrekt sind. Hat sich der Nutzer für eine in seinen Augen sinnvolle Definition für eine zu beschreibende Klasse entschieden, so gibt es im Fall einer nicht 100%ig korrekten Definition mögliche positive und negative Beispiele, die für das Lernproblem fehlerhaft sind. Als positive Beispiele bezeichnen wir hier Individuen der zu beschreibenden Klasse, die nicht abgedeckt werden und als negative Beispiel Individuen, die zusätzlich zu den Instanzen der zu beschreibenden Klasse abgedeckt werden. Bei positiven Beispielen ist das dann immer der Fall, wenn sie nicht alle Eigenschaften der gelernten Klassenbeschreibung besitzen. Ursache dafür ist entweder eine falsche Klassenzuordnung oder das Fehlen von Informationen um diese Klassenbeschreibung zu erfüllen. Negative Beispiele kann man immer genau dann als fehlerhaft ansehen, wenn sie zwar zu der gelernten Klassenbeschreibung gehören, aber der eigentlichen zu beschreibenden Klasse nicht zugeordnet sind. Ursache kann hierbei eine fehlende Klassenzuordnung sein, oder aber es existieren möglicherweise falsche Informationen über das jeweilige Beispiel.

Weil diese Fehler möglichweise ungewollt sind, gibt es hierfür einfache, intuitive Lösungsvorschläge: Für positive Beispiele gibt es zum einen die Möglichkeit, dass sie entweder fehlerhaft der Klasse zugeordnet sind, so dass ein Entfernen oder eine Zuordnung zu einer anderen Klasse sinnvoll ist, oder aber dass mögliche Informationen über dieses Individuum fehlen, und diese vervollständigt werden können/sollten. Bei negativen Beispielen hat man die Möglichkeit, sie zusätzlich noch der zu beschreibenden Klasse zuzuordnen, oder mögliche Informationen zu entfernen, so dass sie nicht mehr die gelernte Klassenbeschreibung erfüllen. Außerdem kann man bei beiden Fällen noch das jeweilige Individuum vollständig aus der Ontologie entfernen, wobei das jedoch nicht trivial ist. Eine simple aber sehr restriktive Möglichkeit hierfür ist es alle Axiome zu entfernen, in denen das Indivuum vorkommt.

Sei im Folgenden $\mathcal{O}$ eine Ontologie, und A die Klasse, zu der eine Klassenbeschreibung C mit $A \equiv C$ gelernt wurde. Sei außerdem p ein fehlerhaftes positives Beispiel, d.h. $\mathcal{O} \models A(p) \wedge \mathcal{O} \not\models C(p)$ und n ein fehlerhaftes negatives Beispiel, also $\mathcal{O} \not\models A(n) \wedge \mathcal{O} \models C(n)$. Wir bezeichnen zudem mit $\mathcal{J}_\eta$ jeweils die Menge der Erklärungen für $\mathcal{O} \models \eta$.

Fehlerhaftes positives Beispiel p

1. Entfernen der Zuordnung von p zu A, d.h.
   entfernen mindestens eines Axioms aus jeder Erklärung $J \in \mathcal{J}_{A(p)}$
   $\mathcal{O}' = \mathcal{O} \setminus S, \forall J \in \mathcal{J}_{A(p)} : S \cap J \neq \emptyset$

2. Vollständiges Entfernen von p aus $\mathcal{O}$, d.h.
   alle Axiome α in denen p vorkommt werden aus $\mathcal{O}$ entfernt.
   $\mathcal{O}' = \mathcal{O} \setminus \{\alpha \mid \alpha \in \mathcal{O} \wedge p \in \mathbf{S}(\alpha)\}$

3. Ergänzen von fehlenden Informationen oder bearbeiten von fehlerhaften Informationen von p damit $\mathcal{O} \models C(p)$
   Weil in der Regel komplexe Klassenbeschreibungen gelernt werden, muss man zunächst erst einmal bestimmen, welche Teile davon p nicht erfüllt. Dies kann man mit Hilfe eines Reasoners machen, wobei generell gilt, dass für eine Konjunktion immer alle Teile erfüllt sein müssen, und für eine Disjunktion immer mindestens eines. Wir verwenden hierbei wieder den im DL-Learner integrierten Reasoner mit Closed-World-Semantik (s. 5.2) um den Problemen mit der OWA aus dem Weg zu gehen. Ausgehend von den ermittelten fehlerhaften Teilen ergeben sich dann folgende Fälle, wobei C_i einen solchen Teil repräsentieren soll:

   - $C_i \equiv B$
     (a) p der Klasse B zuordnen
     $\mathcal{O}' = \mathcal{O} \cup \{B(p)\}$
     (b)
   - $C_i \equiv \forall R.D$
     (a) alle Axiome der Form $R(p, a)$ entfernen, in denen a nicht zur Klasse D gehört
     $\mathcal{O}' = \mathcal{O} \setminus \{R(p, a) \in \mathcal{O} \mid \mathcal{O} \not\models D(a)\}$
     (b) alle Axiome mit der Property R und dem Subjekt p entfernen
     $\mathcal{O}' = \mathcal{O} \setminus \{R(p, a) \in \mathcal{O}\}$
   - $C_i \equiv \exists R.D$
     (a) mindestens ein Axiom der Form $R(p, a)$ hinzufügen, wobei a zur Klasse D gehört
     $\mathcal{O}' = \mathcal{O} \cup \{R(p, a) \mid \mathcal{O} \models D(a)\}$

- $C_i \equiv \leq nR.D$
  (a) mindestens so viele Axiome der Form $R(p,a)$, mit a gehört zur Klasse D, entfernen bis die Anzahl kleiner gleich der maximalen Anzahl n ist
  $\mathcal{O}' = \mathcal{O} \setminus S$,
  $S \subseteq \{R(p,a) \in \mathcal{O} \mid \mathcal{O} \models D(a)\} \wedge |\{R(p,a) \in \mathcal{O} \mid \mathcal{O} \models D(a)\}| - |S| \leq n$
- $C_i \equiv \geq nR.D$
  (a) mindestens so viele Axiome der Form $R(p,a)$, mit a gehört zur Klasse D, hinzufügen bis die Anzahl größer gleich der maximalen Anzahl n ist
  $\mathcal{O}' = \mathcal{O} \cup S$,
  $S = \{R(p,a) \mid \mathcal{O} \models D(a)\} \wedge |\{R(p,a) \in \mathcal{O} \mid \mathcal{O} \models D(a)\}| + |S| \geq n$

Fehlerhaftes negatives Beispiel n

1. n der Klasse A zuordnen
   $\mathcal{O}' = \mathcal{O} \cup \{A(n)\}$

2. Vollständiges Entfernen von n aus $\mathcal{O}$, d.h.
   alle Axiome α in denen n vorkommt werden aus $\mathcal{O}$ entfernt.
   $\mathcal{O}' = \mathcal{O} \setminus \{\alpha \mid \alpha \in \mathcal{O} \wedge n \in \mathbf{S}(\alpha)\}$

3. Bearbeiten von fehlerhaften Informationen von n damit $\mathcal{O} \not\models C(p)$
   Weil in der Regel komplexe Klassenbeschreibungen gelernt werden, muss man zunächst erst einmal bestimmen, welche Teile davon n erfüllt. Dies kann man mit Hilfe eines Reasoners machen, wobei generell gilt, dass für eine Konjunktion immer ein Teil nicht erfüllt sein darf, und für eine Disjunktion immer alle Teile nicht erfüllt sein dürfen. Wir verwenden hierbei wieder den im DL-Learner integrierten Reasoner mit Closed-World-Semantik (s. 5.2) um den Problemen mit der OWA aus dem Weg zu gehen.Ausgehend von den ermittelten fehlerhaften Teilen ergeben sich dann folgende Fälle, wobei C_i einen solchen Teil repräsentieren soll:

   - $C_i \equiv B$
     (a) Entfernen der Zuordnung von n zu B, d.h.
     entfernen mindestens eines Axioms aus jeder Erklärung $J \in \mathcal{J}_{B(n)}$
     $\mathcal{O}' = \mathcal{O} \setminus S, \forall J \in \mathcal{J}_{B(n)} : S \cap J \neq \emptyset$
   - $C_i \equiv \forall R.D$
     (a) mindestens ein Axiom der Form $R(n,a)$ hinzufügen, in dem a nicht zur Klasse D gehört
     $\mathcal{O}' = \mathcal{O} \cup \{R(n,a) \mid R(n,a) \notin \mathcal{O} \wedge \mathcal{O} \not\models D(a)\}$

- $C_i \equiv \exists R.D$
  (a) alle Axiome der Form $R(n,a)$ entfernen, in denen a zur Klasse D gehört
  $\mathcal{O}' = \mathcal{O} \cup \{R(n,a) \mid R(n,a) \in \mathcal{O} \wedge \mathcal{O} \models D(a)\}$
  (b) alle Axiome der Form $R(n,a)$ entfernen
  $\mathcal{O}' = \mathcal{O} \setminus \{R(n,a) \mid R(n,a) \in \mathcal{O}\}$
- $C_i \equiv\leq mR.D$
  (a) mindestens so viele Axiome der Form $R(n,a)$, in denen a zur Klasse D gehört, hinzufügen bis die Anzahl echt größer als die maximale Anzahl m ist
  $\mathcal{O}' = \mathcal{O} \cup S$,
  $S = \{R(n,a) \mid R(n,a) \notin \mathcal{O} \wedge \mathcal{O} \models D(a)\}$ und
  $|\{R(n,a) \mid R(n,a) \in \mathcal{O} \wedge \mathcal{O} \models D(a)\}| + |S| > m$
- $C_i \equiv\geq mR.D$
  (a) mindestens so viele Axiome der Form $R(n,a)$, in denen a gehört zur Klasse D gehört, entfernen bis die Anzahl echt kleiner als die minmale Anzahl m ist
  $\mathcal{O}' = \mathcal{O} \setminus S$,
  $S \subseteq \{R(n,a) \mid R(n,a) \in \mathcal{O} \wedge \mathcal{O} \models D(a)\}$ und
  $|\{R(n,a) \mid R(n,a) \in \mathcal{O} \wedge \mathcal{O} \models D(a)\}| - |S| < m$

Bei allen diesen Möglichkeiten (genauer gesagt, nur bei denen wo etwas hinzugefügt wird) ist zu beachten , dass sie möglicherweise zu einer inkonsistenten Ontologie führen können. Hier sollte man also entweder nur solche Vorschläge anbieten, die die Konsistenz erhalten, oder aber falls die Anwendung einer dieser Vorschläge eigentlich korrekt ist, daran anschließend einen Konfliktauflösungsschritt mit den Methoden aus Kapitel 4 starten.

6 Implementierung

Wir haben die Techniken und Algortihmen aus den vorherigen Kapiteln 4 und 5 im Rahmen unserer Arbeit in einem Tool implementiert und integriert. Dieses Tool, genannt ORE[23] (Ontology Repair and Enrichment), wollen wir hier genauer beschreiben, und dabei vor allem auf Aufbau, Ablauf sowie User Interface eingehen.

ORE ist ein Tool, welches wir aufbauend auf dem DL-Learner Framework als Aufgabe dieser Arbeit entwickelt haben. Es ist damit möglich die Quailität von OWL Ontologien zu verbessern, in dem wir die Techniken aus Kapitel 4 und 5 verwenden. Zur Zeit kann man mit ORE logische Fehler beheben, sowie die Qualität des Schemas durch die Möglichkeiten des Maschinellen Lerns vom DL-Learner verbessern. Zukünftig soll es damit auch möglich sein, andere Fehler in OWL Ontologien, wie in Kapitel 3 beschrieben, zu finden und zu beheben.

6.1 ORE Aufbau

ORE besteht zur Zeit aus den 4 Kernkomponenten Explanation, Impact, Repair und Learning (s. Abb. 8. Sie werden durch einen zentrale Klasse, den ORE-Manager verwaltet. Zu dem werden die Teilfaufgaben in den einzelnen Komponenten selbst jeweils durch einen Manager gesteuert. Die Aufgaben der einzelnen Teile lauten wie folgt:

Learning Diese Komponente steuert die einzelnen Aufgaben beim Lernen von Klassenbeschreibungen. Dazu gehören u.a. das Initialisieren und Starten des Lernalgorithmus, sowie das anschließende Verwalten erhaltener Resultate.

Explanation Hierin enthalten sind Aufgaben wie das Finden von unerfüllbaren Wurzelkonzepten, das Generieren von Erklärungen (regulär und lakonisch) für die entdeckten logischen Fehler oder auch das Extrahieren von lokalitäts-basierten Modulen, um die Performance zu verbessern.

Impact Hier werden die verschiedenen Metriken für die Axiome in den Erklärungen berechnet. Dazu gehören vor allem auch die Folgerungen, die nach dem Entfernen oder Ändern eines Axioms neu hinzukommen bzw. verloren gehen würden.

Repair Aufgaben der Repairkomponente betreffen u.a. die Erstellung von Vorschlägen für die Nachbearbeitung bei Fällen, wo im Lernprozess eine nicht perfekte Klassenbeschreibung ausgewählt wurde. Desweiteren werden hier Axiome, die während einer Reparatur hinzugefügt oder entfernt wurden verwaltet, so dass eine Undo- und Redo-Funktion zu Verfügung steht.

[23] `http://dl-learner.org/wiki/ORE`

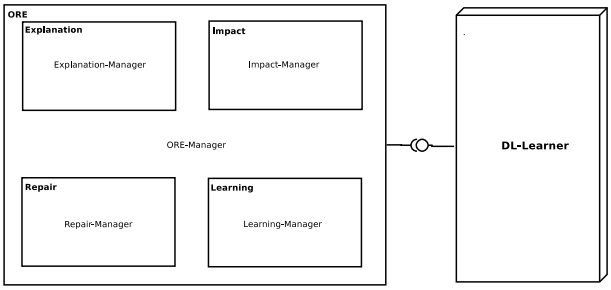


Abb. 8: Aufbau von ORE. Die einzelnen Komponenten werden durch einen zentralen ORE-Manager verwaltet, besitzen zudem jeweils noch einen eigenen Manager für ihre internen Prozesse.

6.2 ORE Ablauf

Der allgemeine Programmablauf sieht wie in Abb. 9 vereinfacht dargestellt folgendermaßen aus: Ausgehend von einer gewählten OWL Ontologie wird zu nächst mit Hilfe des Reasoners (wir verwenden Pellet) getestet ob die Ontologie konsistent ist.schließt Für den Fall, dass sie nicht konsistent ist, schließt sich daran sich ein Debugging-Prozess an. Das gleiche geschieht falls die Ontologie zwar konsistent ist, es aber unerfüllbare Klassen gibt. Trifft beides nicht zu so folgt der Enrichtment-Abschnitt.

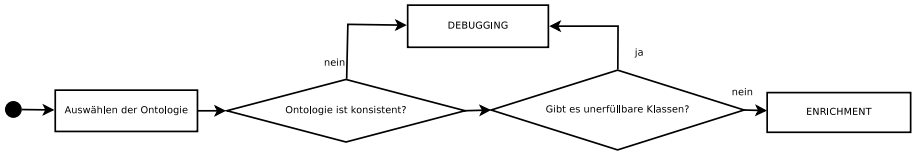


Abb. 9: Der allgemeine Ablauf beim Verwenen von ORE.

Beim Debugging (Abb. 10) werden falls die Ontologie inkonsistent ist, gleich die Erklärungen berechnet, während für unerfüllbare Klassen zunächst nach unerfüllbaren Wurzelklassen gesucht wird, sowie zu der jeweils ausgewählten unerfüllbaren Klasse ein Modul extrahiert, auf welchem dann die Erklärungen berechnet werden können.

In der Enrichment-Phase (Abb. 11 werden zu der jeweils ausgewählten zu beschreibende Klasse zunächst die Beschreibungen mit Hilfe des Lernalgorithmus CELOE vom DL-Learner berechnet. Falls einer der Vorschläge ausgewählt wurde, können dann die fehlerhaften Instanzen angezeigt und repariert werden.

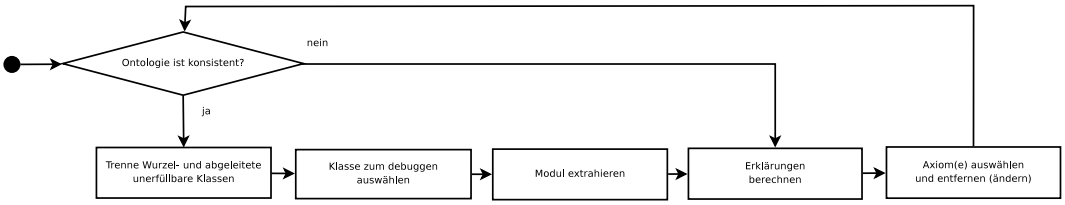


Abb. 10: Der Ablauf beim Debugging von logischen Fehlern mit ORE.

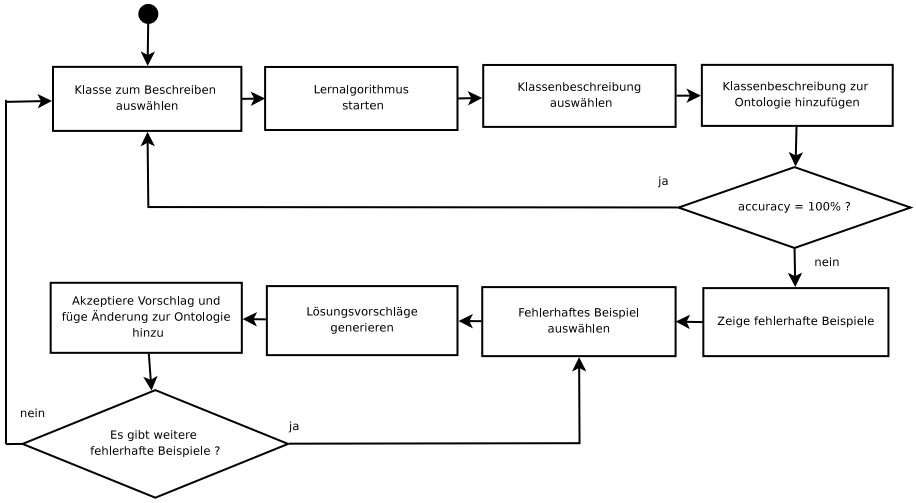


Abb. 11: Der Erweiterungsprozess in ORE unter Verwendung von Lernalgorithmen und anschließende Reparatur fehlerhafter Instanzen.

6.3 ORE UI

Als User Interface haben wir uns in ORE für eine grafische Darstellung in einem Wizard-Konzept entschieden. Das bedeutet man kann schrittweise durch die Dialoge bzw. Panels navigieren, wobei die Abhängigkeiten untereinander so beachtet werden können. Dies ermöglicht es dem Nutzer, mit wenigen Aktionen durch den Prozess der Erweiterung und Reparatur geführt zu werden. Wir beschränken uns hier auf die wesentlichen Schritte, das betrifft insbesondere den Debugging-Prozess und den Enrichment-Prozess, mit anschließender Reparatur fehlerhafter Individuen.

Debugging-Phase

Das Panel beim Debugging ist allgemein in 4 Bereiche unterteilt (Abb.12): Im linken Bereich (1) werden für den Fall das es um das Reparieren von unerfüllbaren Klassen geht, hier die Klassen

ausgelistet und dabei unerfüllbaren Wurzelklassen markiert (Symbol vor den Klassennamen). Der obere rechte Bereich (2) ist für die Abbildung der berechneten Erklärungen verantwortlich. Zu den in Tabellenform dargestellten Erklärungen, befinden sich neben den Axiomen auch eine Reihe von Metriken zu den einzelnen Axiomen, sowie die Auswahl zum Entfernen oder Ändern des Axioms. Die Axiome werden in Manchester Syntax[24] dargestellt, wobei zur Verbesserung der Lesbarkeit die Axiome zusätzlich eingerückt sind, und Schlüsselwörter (z.B. and, or) farbig hervorgehoben werden. Außerdem befinden sich oberhalb noch einige Auswahloptionen, in dem man die maximale Anzahl der darzustellenden Erklärungen, sowie die Art (lakonisch oder regulär) einstellen kann. Im Bereich rechts unten werden (3) die Folgerungen angezeigt, die bei den zum Entfernen ausgewählten Axiomen verloren gehen bzw. neu sind. Es besteht hier die Möglichkeit die verlorengehenden Folgerungen als Axiome zu „retten“,. Im letzten Teil des Debugging-Panels links unten (4) werden die Axiome, die bisher zum Entfernen oder Hinzufügen gewählt wurden, aufgelistet. Neben dem Ausführen des Reparaturplans, gibt es hier auch die Option eines Undo-Schrittes.

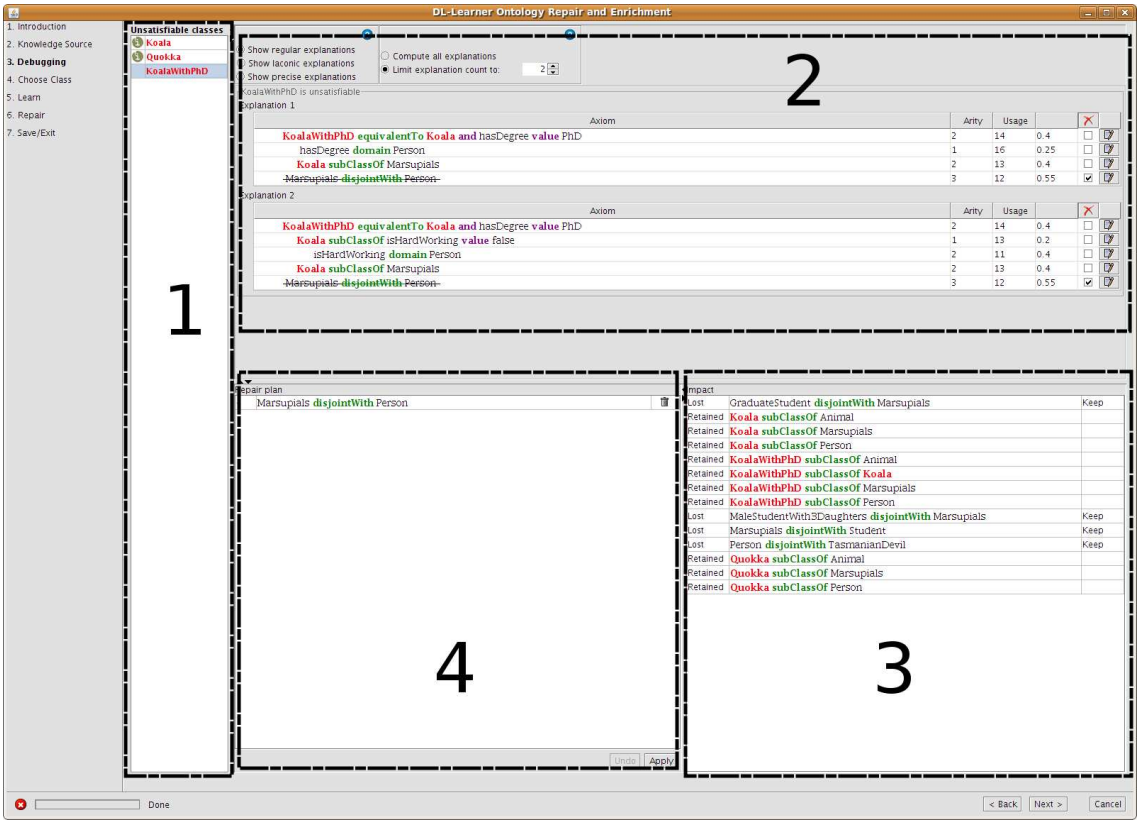


Abb. 12: Panel der Debugging-Phase in ORE.

[24]OWL Manchester Syntax: `http://www.w3.org/2007/OWL/wiki/ManchesterSyntax`

Enrichment-Phase

Für die Lern-Phase besteht das Panel aus 3 Bereichen (Abb. 13). Der Bereich rechts (1) bietet neben dem Starten und Anhalten des Lernalgorithmus, das Einstellen von einigen Parametern für den Lernalgorithmus, sowie die Auswahl, ob Beschreibungen für Superklassen oder äquivalente Klassen gelernt werden sollen. Im Bereich 2 werden die gelernten Beschreibungen angezeigt. Die Klassenbeschreibungen selbst sind wieder in Manchester Syntax dargestellt. Dazu steht zu jeder Klassenbeschreibung zusätzlich ein Genauigkeitswert, berechnet vom Lernalgorithmus CELOE. Wenn man eine Klassenbeschreibung auswählt, wird unten im Bereich 3 eine abstrakte Darstellung von der Abdeckung der gewählten Klassenbschreibung, bezogen auf die Individuen der zu beschreibenden Klasse, angezeigt.

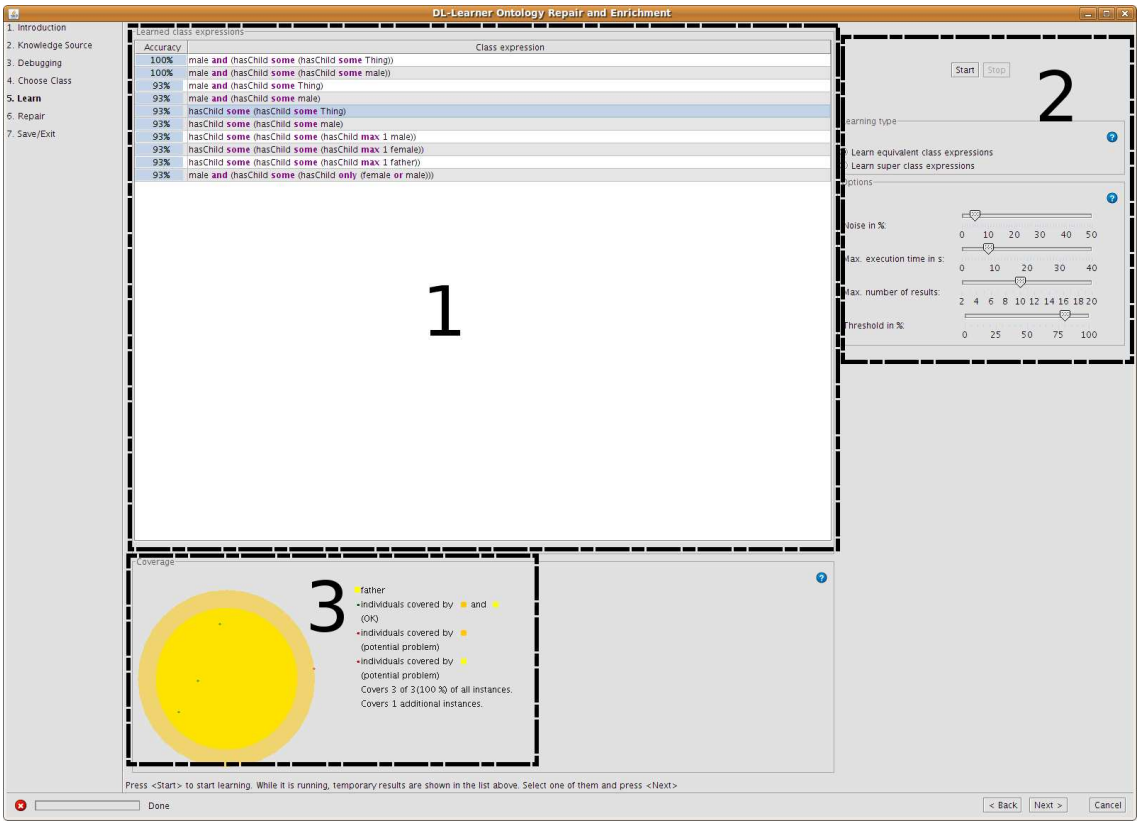


Abb. 13: Panel für die Enrichment-Phase in ORE.

Reparatur eines fehlerhaften Individuums

Für das Reparieren von (möglicherweise) fehlerhaften Individuen erscheint ein Dialog (Abb. 14), der 3 Bereiche enthält. Der obere und wichtigste Bereich (1), stellt die gewählte Klassenbeschreibung dar. Dabei werden die Teile der Beschreibung, die für den Fehler verantwortlich sind, farblich (rot) hervorgehoben. Mit einem Mausklick auf einen solchen Teil öffnet sich ein Popup-Menü, in dem die zur Verfügung stehenden Reparaturvorschläge ausgewählt und ausgeführt werden können. Neben diesem Bereich werden im mittleren Bereich (2) einige informationen über das aktuell zu reparierende Individuum angezeigt. Der untere Bereich dient schließlich der Anzeige von den ausgeführten Reparaturschritten, sowie eine Undo-Funktion dazu.

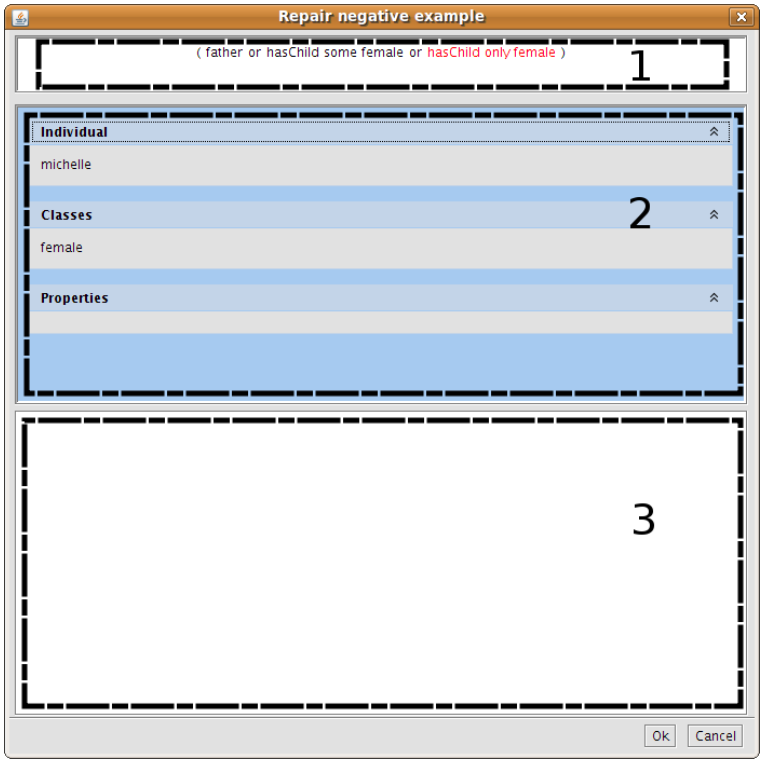


Abb. 14: Panel für Reparatur von Instanzen nach dem Hinzufügen gelernter Klassenbeschreibungen.

7 Fallbeispiel

Wir werden in diesem Kapitel die Fähigkeiten unseres Tools anhand von Beispielen demonstrieren. Wir verwenden dafür zwei verschiedene Ontologien um sowohl den Degugging-Prozess als auch den Erweiterungs-Prozess zu veranschaulichen.

Debugging Anwendung

Um die Debugging-Phase zu zeigen, haben wir eine Ontologie über Toursimus[25] aus dem Protégé-Repository[26] ausgewählt und mit unserem Tool geladen. Nach dem Laden wurde eine unerfüllbare Klasse $Safari$ erkannt (Abb. 15). Ein Auswählen der einzigen unerfüllbaren Klasse $Safari$ führt

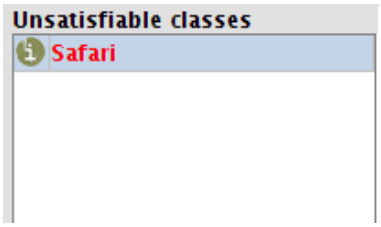


Abb. 15: Liste der erkannten unerfüllbaren Klassen (hier nur $Safari$) in der Tourismus Ontologie. Das Symobol vor der Klasse zeigt an, dass es eine unerfüllbare Wurzelklasse ist (hier klar, weil nur eine Klasse unerfüllbar ist).

dann zu einer gefundenen Erklärung für die Unerfüllbarkeit (Abb. 16). Dabei haben wir uns dafür

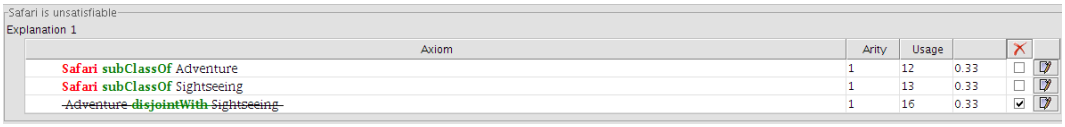


Abb. 16: Die einzige gefundene Erklärung für die Unerfüllbarkeit der Klasse $Safari$.

entschieden, dass das letzte Axiom nicht korrekt ist, und es zum Entfernen markiert. Die durch das Entfernen verlorengehenden Inferenzen werden in Abbildung 17 gezeigt. Wir hätten die Möglichkeit gehabt, diese als Axiome explizit zur Ontologie hinzuzufügen und somit zu erhalten. In dem Beispiel

[25] Tourismus Ontologie: `http://protege.cim3.net/file/pub/ontologies/travel/travel.owl`

[26] Liste von Ontologien: `http://protegewiki.stanford.edu/index.php/Protege_Ontology_Library#OWL_ontologies`

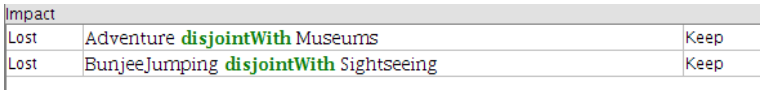


Abb. 17: Die verlorengehenden Inferenzen nach dem das ausgewählte Axiom entfernt werden würde.

haben uns allerdings dagegen entschieden. Deshalb steht im auszuführenden Reparaturplan nur das zum Entfernen ausgewählte Axiom (Abb. 18). Nach dem Ausführen des Reparaturplans ist die Klasse $Safari$ wieder erfüllbar und es sind keine weiteren logischen Inkonsistenzen in der Ontologie vorhanden.

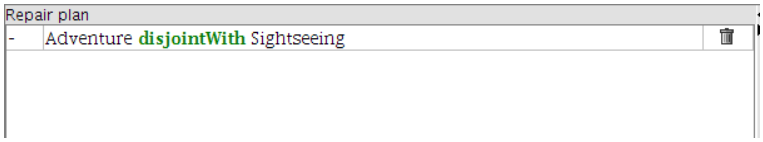


Abb. 18: Der auszuführende Reparaturplan. Nur ein Axiom soll entfernt werden.

Erweiterung Anwendung

Zum zeigen der Erweiterung haben wir eine Ontologie aus dem DL-Learner SVN geladen[27]. Wir habe uns hier entschieden eine Beschreibung für die Klasse $father$ zu lernen. Nach dem Ausführen des Lernalgorithmus haben wir uns aus den berechneten Vorschlägen (Abb. 19) für die Klassenbeschreibung $male\ and\ (hasChild\ some\ Thing)$ entschieden. Diese hat eine Accuracy von 93%, das bedeutet es ist keine perfekte Lösung des Lernproblems. Unser Tool zeigt dann an (Abb. 20), dass es ein Individuum $heinz$ gibt, das nicht zur zu beschreibenden Klasse $father$ gehört, aber zur von uns ausgewählten Klassenbeschreibung. Wir entscheiden uns hier, $heinz$ der Klasse $father$ zuzuordnen. Es sind keine weiteren Individuuen fehlerhaft, womit wir jetzt eine Klassenbeschreibung für die Klasse $father$ in der Ontologie haben, die genau die Individuen von $father$ abdeckt. Wir haben somit das Schema der Ontologie erfolgreich qualitativ erweitert.

Accuracy	Class expression
100%	male **and** (hasChild **some** (hasChild **some** Thing))
100%	male **and** (hasChild **some** (hasChild **some** male))
93%	male **and** (hasChild **some** Thing)
93%	male **and** (hasChild **some** male)
93%	hasChild **some** (hasChild **some** Thing)
93%	hasChild **some** (hasChild **some** male)
93%	hasChild **some** (hasChild **some** (hasChild **max** 1 male))
93%	hasChild **some** (hasChild **some** (hasChild **max** 1 female))
93%	hasChild **some** (hasChild **some** (hasChild **max** 1 father))
93%	male **and** (hasChild **some** (hasChild **only** (female **or** male)))

Abb. 19: Vorschläge des Lernalgorithmus für Klassenbeschreibungen der Klasse $father$

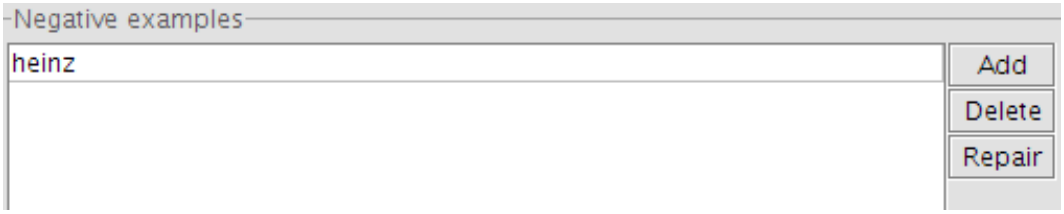


Abb. 20: Das Individuum $heinz$ gehört zur gelernten Klassenbeschreibung, aber nicht zur der zu beschreibenden Klasse $father$. Deshalb ist es ein möglicher Fehler.

[27] Beispiel Ontologie für Erweiterung: `http://dl-learner.svn.sourceforge.net/viewvc/dl-learner/trunk/examples/ore/father3.owl?revision=1250`

8 Verwandte Arbeiten

Das steigende Interesse am Semantic Web in den letzten Jahren hat zu einer stark ansteigenden Zahl bei der Erstellung und Verwendung von Ontologien geführt. Damit nahm auch das Interesse an Techniken zur Evaluation und Qualitätsverbesserung zu. Die verschiedenen Ansätze gehen dabei unterschiedliche Wege. So wurde sich in [32] und [10] zum Beispiel mit Methoden beschäftig, die bei sich ändernden Ontologien entstehende Inkonsistenzen finden und reparieren. In [35] wurde eine Methode, genannt Axiom Pinpointing, vorgestellt, die die für logische Fehler verantwortlichen Axiome findet. Ein nicht auf logische Fehlern ausgerichtetes Verfahren zur Evaluation von Ontologien wurde in [7] dargestellt. Dabei werden durch das Hinzufügen von Eigenschaften (Rigidity, Unity, Identity, Dependency) zu jeder Klasse, problematische Bereiche in der Taxonomie anhand von Regeln identifiziert. Klassen können dann in der Taxonomie rauf- bzw. runtergestellt werden, oder es werden zusätzliche (neue) Klassen eingefügt. Lernen in OWL Ontologien ist u.a. Thema der Dissertation von Jens Lehmann[26]. Dort werden Algorithmen und Heuristiken zum Lernen von Klassenbeschreibungen durch Maschinelles Lernen vorgestellt und evaluiert.

Neben den theoretischen Ansätzen gibt es auch einige Tools, in denen diese praktisch umgesetzt wurden:

Swoop Swoop[28][15] ist ein in Java geschriebener Ontologie Editor, welcher ein Webbrowser ähnliches Design verwendet. Er kann Erklärungen für die Unerfüllbarkeit von Konzepten berechnen und bietet außerdem einen Reparaturmodus, in welchem automatisch Pläne vorgeschlagen werden wobei dafür verschiedene Methoden zur Gewichtung der Axiome verwendet werden. Er kann zwar auch die Erklärungen für inkonsistente Ontologien berechnen, allerdings ist der Reparaturmodus auf die unerfüllbaren Konzepte beschränkt.

RaDON RaDON[29][14] ist ein in Java implementiertes Plugin für das NeOn Toolkit. Es bietet eine Menge von Techniken zum Arbeiten mit inkonsistenten und inkohärenten Ontologien an. Neben dem Berechnen von Erklärungen für Inkohärenz und Inkonsistenz bietet es ähnlich wie Swoop eine manuelle und automatische Reparatur an. Desweiteren ist auch Reasoning mit inkonsistenten Ontologie möglich. Das Tool ist im Gegensatz zu den anderen genannten nicht auf eine einzelne Ontologie beschränkt, sondern kann durch Integration in das NeOn Toolkit auch mit sogenannten Ontologie-Netzwerken umgehen.

Pellint PellInt[30][30] ist ein in Lint geschriebenes Tool für Pellet, welches nach bestimmten Mustern in der Modellierung sucht, die möglicherweise zu Performance Problemen für Reasoner führen

[28] SWOOP: `http://www.mindswap.org/2004/SWOOP/`
[29] RaDON: `http://radon.ontoware.org/demo-codr.htm`
[30] PellInt: `http://pellet.owldl.com/pellint`

können. Diese können dann in bestimmten Fällen repariert werden.

PION und DION PION[31] und DION[32] wurden im SEKT Projekt entwickelt um mit Inkonsistenzen umgehen zu können. PION ist ein inkonsistenz-toleranter Reasoner, der auch in inkonsistenten Ontologien sinnvolle Antworten zurückgeben kann, was für einen normalen Reasoner nicht möglich ist. Er verwendet dafür eine 4-wertige parakonsistente Logik. DION bietet die Möglichkeit zum Berechnen von MUPS und MIPS, darüber hinaus aber keine Möglichkeit zur Reparatur von inkonsistenten oder inkohärenten Ontologien. Zudem ist DION in der Ausdrucksmächtigkeit der Ontologien beschränkt und kann zum Beispiel mit OWL DL Ontologien nicht arbeiten.

Explanation Workbench Die Explanation-Workbench[33] ist ein Plugin für Protégé, in dem für Folgerungen wie z.B. Unerfüllbarkeit von Klassen oder inferrierte Klassensumbsumptionen Erklärungen berechnet werden können. Außerdem bietet es neben den normalen Erklärungen auch die Möglichkeit an fein-granulare Erklärungen zu berechnen, die in [11] als lakonische Erklärungen bezeichnet wurden. Sie beinhalten nur die für die Folgerung relevanten Teile der Axiome, wodurch eine semantisch minimalere Änderungen gestattet wird.

RepairTab Es wurde auf Basis von [22] als für Protégé entwickelt, und soll den Nutzer beim Finden und Beheben von logischen Fehlern in Ontologien unterstützen. Es kann ähnlich wie die Explanation Workbench Erklärungen berechnen, und auch die für die Inkonsistenz relevanten Teile der Axiome anzeigen. Der Unterschied zur Explanation Workbench ist dass dafür ein modifizierter Tableau-Algorithmus genutzt wird. Außerdem werden dem Nutzer die, durch Entfernen oder Modifizieren von Axiomen verlorenen Folgerungen angezeigt, und dafür sinnvolle Vorschläge zum Erhalten dieser gemacht.

Die meisten dieser Programme sind darauf ausgerichtet logische Fehler zu finden und zu beheben, bzw. diese beim Reasoning einfach zu ignorieren (PION). Eine Ausnahme bildet hier Pellint, welches wiederum darauf ausgerichtet mögliche Modellierunsfehler zu finden, die das Reasoning verschlechtern. Unser Tool ORE vereint viele der in den einzelnen Tools vorhandenen Techniken, und bietet zusammen mit den neuen Techniken des DL-Learner ein umfangreiches Werkzeug zur Evaluation und Qualitätsverbesserung von OWL Ontologien an.

[31] PION: `http://wasp.cs.vu.nl/sekt/pion/`
[32] DION: `http://wasp.cs.vu.nl/sekt/dion/`
[33] Explanation Workbench: `http://owl.cs.manchester.ac.uk/explanation/`

9 Zusammenfassung und Ausblick

Wir haben in dieser Arbeit, die sich mit der Reparatur und Erweiterung von Ontologien beschäftigt, eine Vielzahl von Fehlern, die bei der Erstellung bzw. Evolution von Ontologien auftreten können beschrieben. Darauf aufbauend haben wir ein Tool entwickelt, was die Nutzer von Ontologien bei der Reparatur von logischen Fehlern, sowie der Erstellung/Verbesserung des Ontologie-Schemas unterstützen kann. Für die Zukunft ist geplant, weitere Arten von Fehlern durch das Tool abzudecken. Wir denken da insbesondere an Fehler wie Schleifen, Redundanz oder Partitionierungsfehler in den Taxonomien. Hier ist es vorstellbar sich Verfahren aus der Graphentheorie zu Nutze zu machen, um solche Fehler zu erkennen. Um das Tool so einfach wie möglich zu halten, stellen sich hier auch Fragen wie die Darstellung solcher Fehler gegenüber dem Nutzer, bzw. welche Lösungsmöglichkeiten man anbieten könnte.

Literatur

[1] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[2] Franz Baader, editor. *The Description Logic Handbook: Theory, Implementation, and Applications* . Cambridge University Press, 2003.

[3] Asuncion Gómez-Pérez. Evaluation of taxonomic knowledge in ontologies and knowledge bases. In *Proceedings of the 12th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada*, 1999.

[4] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Modular reuse of ontologies: Theory and practice. *J. Artif. Intell. Res. (JAIR)*, 31:273–318, 2008.

[5] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Extracting modules from ontologies: A logic-based approach. In Heiner Stuckenschmidt, Christine Parent, and Stefano Spaccapietra, editors, *Modular Ontologies*, volume 5445 of *Lecture Notes in Computer Science*, pages 159–186. Springer, 2009.

[6] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[7] Nicola Guarino and Christopher A. Welty. An Overview of OntoClean. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 151–172. Springer, Berlin, 2004.

[8] Volker Haarslev and Ralf Moller. Description of the RACER system and its applications. In D. L. McGuinness et al, editor, *Proceedings of the 2001 International Workshop on Description Logics (DL-2001)*. CEUR Workshop Proceedings, 2001.

[9] P. Haase and L. Stojanovic. Consistent evolution of owl ontologies. In *Proceedings of the Second European Semantic Web Conference, Heraklion, Greece*, 2005.

[10] Peter Haase, Frank van Harmelen, Zhisheng Huang, Heiner Stuckenschmidt, and York Sure. A framework for handling inconsistency in changing ontologies. In Yolandal Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC'05)*, volume 3729 of *LNCS*, pages 353–367, Galway, Ireland, November, 6–10 2005. Springer.

[11] Matthew Horridge, Bijan Parsia, and Ulrike Sattler. Laconic and precise justifications in owl. In *7th International Semantic Web Conference (ISWC2008)*, October 2008.

[12] Matthew Horridge, Bijan Parsia, and Ulrike Sattler. Explaining inconsistencies in owl ontologies. In Lluis Godo and Andrea Pugliese, editors, *SUM*, volume 5785 of *Lecture Notes in Computer Science*, pages 124–137. Springer, 2009.

[13] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*. AAAI Press, 2006.

[14] Qiu Ji, Peter Haase, Guilin Qi, Pascal Hitzler, and Steffen Stadtmüller. Radon - repair and diagnosis in ontology networks. In Lora Aroyo, Paolo Traverso, Fabio Ciravegna, Philipp Cimiano, Tom Heath, Eero Hyvönen, Riichiro Mizoguchi, Eyal Oren, Marta Sabou, and Elena Paslaru Bontas Simperl, editors, *ESWC*, volume 5554 of *Lecture Notes in Computer Science*, pages 863–867. Springer, 2009.

[15] A. Kalyanpur, B. Parsia, E. Sirin, B. C. Grau, and J. Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics*, 4(2):144–153, 2006.

[16] A. Kalyanpur, B. Parsia, E. Sirin, and J. Hendler. Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics*, 3(4):268–293, 2005.

[17] Aditya Kalyanpur, Bijan Parsia, and Bernardo Cuenca Grau. Beyond asserted axioms: Fine-grain justifications for owl-dl entailments. In Bijan Parsia, Ulrike Sattler, and David Toman, editors, *Description Logics*, volume 189 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[18] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of owl dl entailments. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon J B Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Guus Schreiber, and Philippe CudrÃ©-Mauroux, editors, *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 267–280, Berlin, Heidelberg, November 2007. Springer Verlag.

[19] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and Bernardo Cuenca Grau. Repairing unsatisfiable concepts in owl ontologies. In *ESWC*, pages 170–184, 2006.

[20] Joey Lam. *Methods for resolving inconsistencies in ontologies*. PhD thesis, University of Aberdeen, 2007.

[21] Joey Sik Chun Lam, Derek H. Sleeman, Jeff Z. Pan, and Wamberto Weber Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. *J. Data Semantics*, 10:62–95, 2008.

[22] Joey Sik Chun Lam, Derek H. Sleeman, Jeff Z. Pan, and Wamberto Weber Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. *J. Data Semantics*, 10:62–95, 2008.

[23] Jens Lehmann. Hybrid learning of ontology classes. In *Proceedings of the 5th International Conference on Machine Learning and Data Mining (MLDM)*, volume 4571 of *Lecture Notes in Computer Science*, page 7883. Springer.

[24] Jens Lehmann. Concept learning in description logics, 2006. Diploma Thesis in Computer Science.

[25] Jens Lehmann. DL-Learner: learning concepts in description logics. *Journal of Machine Learning Research (JMLR)*, 10:2639–2642, 2009.

[26] Jens Lehmann. *Learning OWL Class Expressions*. PhD thesis, Universität Leipzig, 2010. (Laufendes Promotionsverfahren).

[27] Jens Lehmann and Christoph Haase. Ideal downward refinement in the el description logic. Technical report, 2009.

[28] 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.

[29] 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.

[30] Harris Lin and Evren Sirin. Pellint - a performance lint tool for pellet. In Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, *OWLED*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[31] B Motik. Reasoning in description logics using resolution and deductive databases. *PhD theis, University Karlsruhe, Germany*, 2006.

[32] P. Plessers and O. De Troyer. Resolving inconsistencies in evolving ontologies. In *Proc. 3rd European Semantic Web Conference (ESWC 2006)*, pages 200–214. Springer, 2006.

[33] Protege. The protege ontology editor and knowledge acqisition system. http://protege.stanford.edu/, 2000.

[34] R Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[35] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 355–362. Morgan Kaufmann, 2003.

[36] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics*, 5(2):51–53, 2007.

[37] R. Studer, R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data and knowledge engineering*, 25:161–197, 1998.

[38] Boontawee Suntisrivaraporn, Guilin Qi, Qiu Ji, and Peter Haase. A modularization-based approach to finding all justifications for owl dl entailments. pages 1–15. 2008.

[39] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

Ich möchte hiermit allen Menschen danken, die mich bei der Erstellung meiner Arbeit unterstützt haben. Das gilt vor allem für die Mitarbeiter des AKSW aus Raum 5.10 (Sebastian, Michael, Jörg, Claus, Christian). Ein ganz besonderer Dank geht aber an meinen Betreuer Jens Lehmann, der immer geholfen hat, wenn es notwendig war. Danke an alle!

„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“.

Ort Datum Unterschrift