

Bachelorarbeit

-

Umsetzung einer hierarchisch-ikonografischen Suche auf numismatischen Daten mithilfe eines Apache-Jena Reasoners

Eingereicht an der
Goethe-Universität Frankfurt am Main
Institut für Informatik

von

Steven Jerome Nowak und Nico Lambert

Unter der Leitung von:
Dr. Karsten Tolle

17.03.2025

Inhaltsverzeichnis

1	Einleitung	4
1.1	Einführung	4
1.2	Thema	5
1.3	Begriffserklärungen	7
1.4	Relevanz	12
2	Hierarchisch-ikonografische Suche	13
2.1	Allgemein	13
2.2	Ausgangssituation beim Projekt Corpus Nummorum	14
2.3	Verwendung eines Reasoners	15
3	Reasoner Auswahl für eine hierarchisch-ikonografische Suche	18
3.1	Welche Reasoner werden in Betracht gezogen?	18
3.2	Charakterisierung der zur Auswahl stehenden Reasoner	19
3.2.1	Transitive Reasoner	19
3.2.2	Generic Rule Reasoner	20
3.2.3	RDFS Rule Reasoner	26
3.2.4	OWL Reasoner	29
3.2.4.1	Full OWL Reasoner	30
3.2.4.2	Mini OWL Reasoner	30
3.2.4.3	Micro OWL Reasoner	31
3.2.5	Reasoner Übersicht	31
3.3	Welcher Reasoner bietet sich am besten für die Umsetzung einer solchen Suche an?	31
3.3.1	Welche Reasoner kommen nicht in Frage / wieso nicht?	32

3.3.2	Reasoner Wahl für eine hierarchische Suche	35
4	Visualisierung einer hierarchischen Suche	39
4.1	Ausgangssituation und notwendige Änderungen	40
4.2	Möglichkeiten und MockUps	42
4.3	Wahl der Visualisierung	45
4.3.1	Gespräch mit einer Corpus Nummorum Mitarbeiterin	45
4.3.2	Schwierigkeiten und Lösungsansätze bei der Umsetzung der Vorstellungen von Frau Dr. Peter	48
4.3.3	Endgültige Umsetzung der Visualisierung	51
5	Fazit	55
5.1	Beantwortung der Forschungsfrage	55
5.2	Ausblick / Zukünftige Arbeit	56
6	Quellen	57
7	Anhang	60
7.1	RDF Präfixe	60

1 Einleitung

Im ersten Kapitel unserer Abschlussarbeit stellen wir zuerst grundlegende Hintergrundinformationen zu unserem Thema dar. Im Folgenden gehen wir genauer auf die Thematik ein und formulieren unsere Forschungsfrage. Abschließend erklären wir essenzielle Begrifflichkeiten und erläutern die Relevanz unserer Arbeit.

1.1 Einführung

Am Corpus Nummorum¹ (CN) beschäftigt man sich mit der Aufarbeitung und Darstellung antiker Münzen. Hierbei werden die Informationen zu den von Archäologen gefundenen Münzen dargestellt. Insgesamt umfasst der Datenbestand mehrere zehntausend Münzen die „von der archaischen Zeit bis hin zu den unter römischer Herrschaft emittierten Prägungen von Moesia inferior, Thrakien, Mysien und der Troas“ [[Corpus Nummorum](https://www.corpus-nummorum.eu/) / 1].



original - License: [Public Domain Mark 1.0](#) - Photographer: Reinhard Saczewski

[Enlarged](#) [Download](#)

Link to Source:
<https://ikmk.smb.museum/object?id=18203033>



cn coin 59413

Connected to Type	CN Type 21711
Mint	Pantikapaion
Date	c. 345-340 BC Classical Period
Obverse	<p>Design</p> <p>Bearded head of a satyr, left, with long hair and blunt nose. go to the NLP result of this description</p>
Reverse	<p>Legend</p> <p>ΠΑΝ</p> <p>Design</p> <p>Griffin standing left, right forepaw raised, holding spear in mouth; at its feet, ear of corn. go to the NLP result of this description</p>
Metrology	
Denomination	stater
Material	gold
Diameter	21 mm
Weight	9.10 g
Axis	11

Abbildung 1: Darstellung einer Münze am CN

¹<https://www.corpus-nummorum.eu/>

Wie in der obigen Abbildung dargestellt, erstreckt sich die Repräsentation einer Münze vom Typ über die Münzstätte und die Datierung bis hin zur Beschreibung der Münzseiten sowie der Metrologie. Das Abbild einer Münze stellt dabei verschiedene, für das Zeitalter relevante, Lebewesen oder Objekte dar, welche miteinander interagieren.

Die Speicherung der numismatischen Daten erfolgt mit Hilfe einer Resource Description Framework (RDF) Datenbank. Hierbei werden Münzdaten und ihre direkten Beziehungen zueinander in Tripelform hinterlegt. Des Weiteren existiert für die Datenspeicherung eine My Structured Query Language (MySQL) Datenbank. Das Grundgerüst der hierarchischen Struktur der RDF-Datenbank bilden hierbei die „Subclass of“ Beziehungen zwischen verschiedenen Klassen, sowie die „Type of“ Beziehungen zwischen Klassen und ihren Elementen. Die Wurzeln des „Hierarchiebaums“ bilden hierbei die Kategorien: „Person“, „Animal“, „Plant“ und „Object“. Jede Klasse, wie beispielsweise „Deities“, „Greek“ oder „Weapons“, sowie deren Elemente, wie „Athena“, „Ares“ oder „Bow“, sind genau einer dieser vier Generalisierungen untergeordnet.

Um die Münzen ikonografisch suchen zu können, haben Danilo Pantic und Mohammed Sayed Mahmod bereits eine auf dem CN RDF Dump basierende Tripel Suche entwickelt². Durch die Eingabe von Subjekt, Prädikat und Objekt für die Vorderbeziehungsweise Rückseite der Münze werden genau die numismatischen Objekte ausgegeben, deren Darstellung der Eingabe des Nutzers entspricht. Lautet die Eingabe für die Vorderseite beispielsweise „Artemis holding Bow“, so werden nur die Münzen angezeigt, auf deren Vorderseite abgebildet ist, dass Artemis einen Bogen hält. Aktuell bietet der Corpus Nummorum jedoch keine Möglichkeit, den numismatischen Datenbestand hierarchisch zu durchsuchen. Dies stellt eine erhebliche Einschränkung beim Suchen von Münzen dar, denn Eingaben, wie „Female holding Shield“, „Male wearing Cuirass“ oder „Person holding Object“ liefern keine Ergebnisse. Die Identifikation von Klassenobjekten, sowie das schnelle Navigieren innerhalb der Hierarchie, nach oben und nach unten, sind jedoch wesentliche Elemente einer effizienten Suche.

1.2 Thema

Im Zuge unserer Bachelorarbeit beschäftigen wir uns mit der Umsetzung einer hierarchisch-ikonografischen Suche auf numismatischen RDF-Daten am Corpus

²<https://github.com/Danilopa/Bachelorthesis>

Nummorum. Aufgrund dessen betrachten wir nur die vorhandene RDF-Datenbank, nicht aber die ebenfalls vorhandene MySQL-Datenbank.

Die Umsetzung setzt sich aus zwei Themenbereichen zusammen:

Zum einen befassen wir uns mit der technischen Umsetzung einer hierarchischen Suche, welche am Backend erfolgt. Da die von uns zu implementierende hierarchische Suche auf ikonografischen Objekten basiert, handelt es sich in unserem Fall um eine hierarchisch-ikonografische Suchfunktion. Der zentrale Bestandteil ist hierbei die Nutzung eines Reasoners. Ein Reasoner ermöglicht es, komplexe hierarchische Beziehungen und semantische Strukturen im Datenbestand zu interpretieren und für uns aus menschlicher Sicht, logisch, konsistente Schlussfolgerungen zu ziehen. Wir benötigen einen Reasoner, um die in der RDF-Datenbank noch nicht direkt hinterlegten Informationen zu Klassenbeziehungen durch logische Schlussfolgerungen dynamisch herzuleiten. Im Rahmen dieser Arbeit stellen wir dar, wieso ein Reasoner hierbei eine wichtige Rolle spielt, welche Reasoner zur Auswahl stehen und welcher davon am besten für eine hierarchische Suche geeignet ist. Dies werden wir anhand des Fallbeispiels Corpus Nummorum und auch im Allgemeinen untersuchen. Dazu stellen wir verschiedene Standard-Reasoner von Apache-Jena gegenüber und gehen auf ihre Funktionsweise, Vor- / Nachteile, sowie Methoden und Ergebnisse bei verschiedenen Testläufen ein. Die Untersuchungen basieren auf den Ergebnissen der Apache-Jena-Fuseki Version 5.0.0. Die Suchfunktion soll hierbei ermöglichen, nach verschieden stark generalisierten oder schwach spezialisierten, Gruppen ikonografischer Motive suchen zu können. Subjekte und Objekte können somit nicht mehr ausschließlich spezifische Entitäten, wie „Hades“ repräsentieren, sondern auch Gruppen, wie „Astrological“, „Egyptian“, „Leafes“ etc..

Des Weiteren werden wir verschiedene Möglichkeiten der Visualisierung der hierarchischen Suche erläutern und präsentieren. Hauptaugenmerk ist hierbei die Benutzerfreundlichkeit, insbesondere für neue Nutzer. Im Fokus liegt dabei die Möglichkeit des schnellen hierarchischen Anpassens der Elemente, welche in der Tripel-Suche verwendet werden. Hierfür bauen wir zum einen auf der Arbeit³ von Danilo Pantic und Mohammed Sayed Mahmod auf. Zum anderen entwickeln wir die Benutzeroberfläche unter Berücksichtigung der Wünsche einer Corpus Nummorum Mitarbeiterin weiter.

³<https://github.com/Danilopa/Bachelorthesis>

In unserer Bachelorarbeit setzen wir uns konkret mit folgenden Fragestellungen auseinander:

Welcher Apache-Jena Reasoner bietet sich an, um eine hierarchisch-ikonografische Suche auf RDF-Daten zu implementieren? Wie kann eine solche Suchfunktionalität am besten dargestellt werden?

1.3 Begriffserklärungen

Ontologie:

Eine Ontologie ist eine Darstellung einer Menge von Begriffen und den Beziehungen zwischen Ihnen. Sie liefert Maschinen Hintergrundwissen, sodass diese zu den gleichen Schlussfolgerungen wie ein Mensch kommen können. Gängige Beispiele für Ontologien sind hierbei vor allem RDF und OWL. Die Bausteine einer Ontologie bilden dabei die folgenden Begriffe [[Wikipedia](#) / 2]:

- Klasse: Sie stellt eine Gruppierung von Daten mit ähnlichen Attributen dar (Beispiel: Ein Schwert und ein Speer sind beides Elemente der Klasse Waffen.).
- Instanz: Jede Instanz repräsentiert eine Entität einer Klasse (Beispiel: Ein Auto ist eine Instanz der Klasse Fahrzeuge.).
- Typ: Ein Typ bildet eine Gruppierung innerhalb einer Klasse ab (Beispiel: Max Muster ist eine Instanz vom Typ männlich in der Klasse Menschen.).
- Relation: Eine Relation spiegelt das Verhältnis zwischen verschiedenen Instanzen wieder (Beispiel: Die Erde liegt innerhalb der Milchstraße.).
- Vererbung: Durch eine Vererbung wird dargestellt, dass Eigenschaften und Relationen auch für untergeordnete Klassen und Instanzen gelten (Beispiel: Die Klasse motorisierte Fahrzeuge vererbt an die Klasse Autos, dass jede Instanz der Klasse einen Motor besitzt.).
- Axiom: Jedes Axiom beinhaltet absolute Informationen, welche immer gültig sind (Beispiel: Der 31. Dezember ist der letzte Tag des Jahres.).

Reasoner:

Ein Reasoner beziehungsweise eine Reasoning Engine, zu Deutsch auch als Regel- oder Inferenzmaschine bezeichnet, ist eine Softwarekomponente [[Dipl.-Ing. \(FH\)](#)]

[Stefan Luber / 3](#)]. Die Fähigkeiten eines Reasoners liegen zum einen darin, „die Konsistenz von Ontologien“ zu überprüfen. Des Weiteren kann ein Reasoner, durch die Auswertung von Ableitungsregeln, aus „logischen Axiomen und Aussagen“ neue Schlussfolgerungen ziehen [[Anne Augustin, Marco Kranz, Ralph Schäfermeier / 4](#)]. Wenn zum Beispiel die beiden Aussagen „Wenn er aufgeräumt hat, ist das Zimmer ordentlich.“ und „Das Zimmer ist nicht ordentlich.“ vorhanden sind, dann schlussfolgert der Reasoner „Er hat nicht aufgeräumt.“ Somit ist ein Reasoner in der Lage, auf Basis bereits bestehender Informationen, neue Erkenntnisse herzuleiten. Diese Funktion kann beispielsweise durch Vorwärts- aber auch Rückwärtsverkettung der Informationen ermöglicht werden. Die Herleitung neuer Informationen ist allerdings ebenso auf Basis einer zugrunde liegenden Wahrscheinlichkeitsberechnung erreichbar. Die Fähigkeiten eines Reasoners werden unter anderem in der Robotik, im Feld der Künstlichen Intelligenz und für die semantische Suche im Web eingesetzt [[Dipl.-Ing. \(FH\) Stefan Luber / 3](#)].

URI:

Uniform Resource Identifier (URI) dienen dazu, alle möglichen Elemente als eine Ressource zu definieren. Dabei beginnt jede URI mit einem Muster, wie „http“ oder „mailto“. Diese Schemata haben die Aufgabe, die Syntax der URI zu definieren. Das folgt daraus, dass URIs keine festen Regeln besitzen, sondern jedes Muster seine komplett eigenen Syntax verwendet [[Elke von Lienen / 5](#)].

RDF:

Das Resource Description Framework (RDF) ist ein von dem World Wide Web Consortium (W3C) entwickeltes System. Dieses basiert auf einem Graphenmodell, welches Informationen in Form von Tripeln darstellt. Ein Tripel besteht hierbei aus einem Subjekt, einem Prädikat und einem Objekt. Subjekt und Objekt eines Tripels werden im Graphen jeweils als separate Knoten repräsentiert, wobei das Prädikat die Beziehung gerichtet von Subjekt zu Objekt beschreibt. Dabei werden in RDF häufig URIs verwendet, um Ressourcen eindeutig zu identifizieren [[W3C / 6](#)]. Die Zielsetzung liegt hierbei in der Darstellung von Metadaten im Web [[Elke von Lienen / 5](#)]. Zu diesem Zweck bietet das Framework mehrere Funktionen, darunter eine Umgebung zur Codierung, zum Austausch und zur Weiterverarbeitung der Metadaten. Darüber hinaus definiert das Framework ein Modell, welches die Organisation von Daten als

gerichteten Graph beschreibt [W3C, Eckhardt Schön / 6, 7].

RDFS:

Das Resource Description Framework Schema (RDFS) wurde vom World Wide Web Consortium (W3C) entwickelt, mit dem Ziel, das RDF-System semantisch zu erweitern [Elke von Lienen / 5]. Zu diesem Zweck stellt es ein Vokabular bereit, welches die Zusammenhänge der Knoten im Graphen erklärt [W3C / 6]. Dieses beinhaltet wichtige Komponenten, wie „rdfs:Class“ oder „rdfs:subClassOf“. Hierbei wird „rdfs:Class“ verwendet, um Klassen zu definieren. „rdfs:subClassOf“ wird eingesetzt, um eine Hierarchie zwischen mehreren Klassen herzustellen. Um die Konsistenz dieser Zusammenhänge zu gewährleisten, werden für die Metadaten weitere Metadaten erstellt, welche die Daten als Individuen identifizieren und sie unterschiedlichen Klassen zuordnen. Dabei kann eine Klasse wiederum eine Instanz einer anderen Klasse darstellen. Dies erzeugt eine hierarchische Struktur. Somit kann zum Beispiel die Klasse Autos eine Instanz der Klasse Fahrzeuge sein. Die rein semantische Form von RDFS bedingt, dass es keine neuen Datenformate definiert, sondern lediglich die URIs festlegt. Jene können folglich beispielsweise von Reasonern genutzt werden, um Schlussfolgerungen über die Beziehung der Daten zu treffen [Elke von Lienen / 5].

OWL:

Die Web Ontology Language (OWL) wurde als Erweiterung von RDF vom World Wide Web Consortium (W3C) entwickelt. Ihr Einsatzgebiet ist die Verarbeitung von Daten, die nicht nur visualisiert, sondern auch verarbeitet werden sollen [W3C / 8]. Die Semantik der Sprache baut hierbei auf der des Resource Description Framework auf. Zudem ist zu beachten, dass zwischen zwei Generationen der Ontologie unterschieden wird: OWL 1 und OWL 2. Hierbei stellt OWL 2 eine funktionale Weiterentwicklung von OWL 1 dar [W3C / 9]. Im Rahmen unserer Abschlussarbeit ist ausschließlich OWL 1 von Relevanz.

OWL 1 basiert auf Spezifikationen, welche im Jahr 2004 definiert wurden. Dabei wird die Sprache in drei verschiedene Profile eingeteilt. Diese sind OWL Lite, OWL DL und OWL Full. OWL Lite ist hierbei formal die einfachste Sprache und hat den kleinsten Anwendungsbereich. Sie ist vor allem dann empfehlenswert, wenn eine hierarchische Struktur im Mittelpunkt der Daten steht. OWL Full ist das Gegenstück

zu OWL Lite. Die Full Variante von OWL stellt die komplexeste der drei Varianten dar. Sie unterstützt unter anderem die Syntax von RDFS. Im Vergleich zu den beiden anderen Sprachen, geht sie mit dem Nachteil einher, dass nicht garantiert werden kann, dass alle auf der Sprache basierenden Schlussfolgerungen auch terminieren. Dem entgegen stellt OWL DL einen Kompromiss aus OWL Lite und OWL Full dar. So besitzt OWL DL zwar ein größeres Vokabular als OWL Lite, stellt aber im Gegensatz zu OWL Full sicher, dass jede Schlussfolgerung der Sprache terminiert. Ein weiterer wichtiger Aspekt ist, dass die komplexeren OWL Varianten Erweiterungen der einfacheren Formen darstellen. Somit ist jede für OWL DL gültige Ontologie auch für OWL Full und jede OWL Lite Ontologie sowohl für OWL DL als auch für OWL Full gültig [W3C / 8].

OWL 2 beruht auf Spezifikationen, welche im Jahr 2009 entwickelt wurden. Dabei wird die Sprache genau wie OWL 1 in verschiedene Profile unterteilt. Die drei wesentlichen Profile sind OWL EL, OWL QL und OWL RL. Im Vergleich zu OWL 1 sind die drei Untersprachen von OWL 2 keine Erweiterungen zueinander. Auf der Webseite von W3C⁴ werden diese genauer beschrieben.

SPARQL:

Die SPARQL Protocol and RDF Query Language (SPARQL) wurde vom World Wide Web Consortium (W3C) als definierte Sprache konzipiert, um die Bearbeitung und Abfrage von Daten in RDF-Graphen zu ermöglichen [W3C / 10].

Eine Abfrage mit SPARQL beruht „auf dem Abgleich von Graphmustern, das heißt dem Abgleich von Sätzen aus Dreiergruppenmustern, die konjunktive (UND) oder disjunktive (ODER) Bedingungen bilden“. Ein sogenanntes Dreiergruppenmuster besteht hierbei, ebenso wie ein RDF-Datum, aus einem Subjekt, einem Prädikat und einem Objekt. Im Vergleich zu RDF unterscheidet sich dieses Tripel jedoch dadurch, dass weder Subjekt noch Prädikat oder Objekt einen festen Wert besitzen müssen. Alle drei Elemente können jeweils durch eine Variable ersetzt werden, wenn der Wert des Tripel-Elements nicht eindeutig festgelegt ist. Eine Variable wird folgendermaßen definiert „?variablenname“ Für jede SPARQL-Abfrage wird geprüft, ob das gefragte Tripel mit Teilen der vorhandenen RDF-Daten übereinstimmt. Dies ist für ein RDF-Datum der Fall, wenn das Tripel der SPARQL-Abfrage nach Ersetzen der Variablen durch die entsprechenden Tripel-Elemente aus dem RDF-Datum dem

⁴https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/#Profile_Specification

RDF-Datum entspricht [CORDIS / 11]. Folgendermaßen sieht die Verwendung in der Praxis aus: Um das Abgabedatum der Bachelorarbeit mit dem Titel „Umsetzung einer hierarchisch-ikonografischen Suche auf numismatischen Daten mithilfe eines Apache-Jena Reasoners“ zu erhalten, kann man an den Abfrageendpoint des entsprechenden Datengraphen folgende Abfrage stellen:

Abbildung 2: SPARQL-Abfrage um das Abgabedatum zu erfragen (Beispiel angelegt an [CORDIS / 11])

Wie auch im obigen Beispiel werden Präfixe in SPARQL-Abfragen verwendet, um die Abfrage übersichtlicher zu gestalten. Ein Präfix wird wie folgt definiert „PREFIX prefixName: <uri>.“

Prolog und Datalog:

Bei Prolog handelt es sich um eine logikbasierte Programmiersprache. Diese eignet sich vor allem zum lösen „nicht-numerischer Probleme, die eine explizite Repräsentation der verfügbaren Informationen nahelegen oder fordern“ [Sven Naumann / 12]. Syntaktisch sowie semantisch ist Prolog mit Datalog sehr vergleichbar. Datalog stellt hierbei eine Programmiersprache für deduktive Datenbanken dar. Die beiden Programmiersprachen unterscheiden sich hierbei unter anderem in den folgenden Punkten [Wikipedia / 13]:

- Beim Prolog dürfen im Gegensatz zum Datalog zusammengesetzte Terme als Argumente von Prädikaten genutzt werden.
- Datalog-Programme müssen bezüglich Negation und Rekursion in Schichten organisiert werden, um sicher gehen zu können, dass die Regelauswertung eindeutig ist.
- Bei Programmen, die in Datalog verfasst werden, ist es egal, in welcher Reihenfolge die Regeln definiert werden.

1.4 Relevanz

Die Bedeutung dieser Arbeit wird deutlich, wenn man das Projekt Corpus Nummorum⁵ betrachtet. Hier werden die Daten von zehntausenden Münzen dokumentiert. Um mit einer solchen Datenmengen in angemessener Zeit arbeiten zu können, ist eine Software erforderlich, die es dem Benutzer ermöglicht, schnell und einfach in den Daten recherchieren zu können.

Den Grundstein hierfür haben Danilo Pantic und Mohammed Sayed Mahmod mit ihrer Arbeit gelegt. Jedoch gibt es noch einige Einschränkungen. Eine dieser Limitationen ist, dass der Suchraum nicht ausgeweitet werden kann. Zwar ist die Suche nach Münzen mit Subjekt, Prädikat und Objekt Tripel für beispielsweise spezifische Personen, wie Athena, möglich, jedoch kann man nicht nach übergeordneten Klassen von Entitäten in einer Abfrage suchen. Zum Beispiel ist es nicht ausführbar, als Subjekt „Deities“, „Greek“ oder ähnliches einzutragen. Die hierarchisch-ikonografische Suche hingegen ermöglicht dies. Sie bietet außerdem die Möglichkeit, das gesuchte Subjekt oder Objekt schnell und einfach spezifischer oder unspezifischer zu definieren. Somit kann man in der Hierarchie schnell nach oben und nach unten navigieren. Ist die Eingabe zum Beispiel „Deities wearing Crown“ so kann „Deities“ schnell spezialisiert werden (zum Beispiel zu „Apollo“) oder generalisiert werden (zum Beispiel zu „Person“). Des Weiteren ermöglicht die hierarchisch-ikonografische Suche die Lokalisierung von Subjekten / Objekten, welche auf der selben Hierarchieebene stehen. Um all diese, durch die hierarchisch-ikonografische Suche hervorgerufenen, Neuerungen für den Nutzer möglichst bedienungsfreundlich zu machen, ist die Wahl der Visualisierung von entscheidender Bedeutung. Deshalb beschäftigen wir uns intensiv mit verschiedenen Darstellungsmöglichkeiten.

⁵<https://www.corpus-nummorum.eu/>

2 Hierarchisch-ikonografische Suche

Im folgenden Kapitel setzen wir uns mit der hierarchisch-ikonografischen Suche auseinander. Zu Beginn erklären wir, die Bedeutung von „hierarchisch-ikonografisch“ im Kontext dieser Arbeit. Darauffolgend beschreiben wir die technische Ausgangssituation für die Implementierung einer solchen Suche am Corpus Nummorum. Hierbei stellen wir Ziel und Zweck unserer Umsetzung dar. Zum Ende hin erläutern wir, wieso und inwiefern für die Durchführung ein Reasoner benötigt wird.

2.1 Allgemein

Bevor wir uns mit dem Einsatz einer hierarchisch-ikonografischen Suche am Projekt Corpus Nummorum auseinander setzen, sollte zuvor definiert werden, was überhaupt eine solche Suchfunktionalität mit sich bringt.

Im Kontext dieser Arbeit wird eine hierarchische Suche definiert, als eine Suche, deren Funktionen über das Finden eines spezifischen Zieles hinausgehen. Sie soll unter anderem die Möglichkeit bieten, den Rahmen der Suche auszuweiten und zu verkleinern. Dabei soll es dem Nutzer möglich sein, den Suchraum unter Berücksichtigung der vorhandenen Eingabemöglichkeiten zu generalisieren, aber auch zu spezifizieren. Damit dies möglich ist, müssen die Suchobjekte in einer Beziehung zueinander stehen. Ebendies bedeutet, dass jedes Suchziel in eine von zwei Rollen kategorisiert wird. In der ersten Variante beschreibt das Suchziel ein spezifisches Objekt. Hierfür gibt es keine Spezialisierungsmöglichkeit. Ein Beispiel hierfür ist die Person Julius Cäsar. Andernfalls ist das gesuchte Objekt eine Generalisierung von mehreren einzelnen Zielen. Im Fall der zweiten Kategorie spiegelt die Suche nach dem Suchobjekt somit keine direkte Suche nach einem spezifischen Objekt wieder. Es handelt sich stattdessen um eine Suche nach allen Entitäten, welche spezifische Elemente der Kategorie sind. Dem Nutzer soll es möglich sein, schnell zwischen den in Verbindung stehenden Varianten wechseln zu können.

Des Weiteren definieren wir im Rahmen dieser Arbeit eine ikonografische Suche, als eine auf Daten von Bildern basierende Suchfunktion. Hierbei wurden die Bilddaten hauptsächlich durch den Einsatz von Natural Language Processing (NLP) extrahiert. Mögliche Suchobjekte fallen hierbei in eine von zwei Kategorien. Zum

einen kann das Ziel der Suche ein spezifisch abgebildetes Objekt sein. Beispiele hierfür wären unter anderem die mythologischen Figuren Odysseus und Zeus. Zum anderen kann das Ziel der Suche aber auch eine auf einer Münzseite dargestellte Interaktion umfassen. Ein Beispiel hierfür ist: „Odysseus bekämpft Polyphem“.

Führt man nun die Definitionen der hierarchischen und der ikonografischen Suche zusammen, so ergibt sich für die hierarchisch-ikonografische Suche, dass sie im Wesentlichen die Suche nach Interaktionen von beliebigen Elementen einer verallgemeinerten Gruppe oder spezifischen Entitäten in Bildern ermöglichen soll. Um dies technisch möglich zu machen benötigen wir, wie eingangs bereits angedeutet, einen Reasoner. Eine genauere Erläuterung dazu finden sie unter Abschnitt 2.3 und 3.

Zusätzlich muss die hierarchisch-ikonografische Suchfunktion geplant werden. Es ist wichtig, die angeforderten Möglichkeiten der Suchfunktionalität den Nutzern möglichst benutzerfreundlich anzubieten. Über das genaue Vorgehen hierzu können Sie unter Abschnitt 4.1 beziehungsweise 4.2 mehr erfahren.

2.2 Ausgangssituation beim Projekt Corpus Nummorum

Da geplant ist, die Resultate dieser Arbeit, aufbauend auf der „Iconographic Search“⁶, zukünftig am Corpus Nummorum⁷ zu implementieren, ist der aktuelle Stand des Projekts von maßgeblicher Bedeutung. Die Form der Datenbank bildet hierbei das Grundgerüst der Suchfunktionalität. Das Erste, was es hierbei zu beachten gilt, ist, dass die Informationen des Projekts neben einer für uns irrelevanten MySQL-Datenbank in einer RDF-Datenbank gespeichert werden. Somit liegen die Daten in Form von Tripeln vor. Hieraus folgt, dass eine für die Tripelstruktur ausgelegte Abfragesprache wie SPARQL verwendet werden muss, um die gewünschten Daten aus der Datenbank zu ziehen. Wirft man nun einen genaueren Blick auf die Datenbank, welche am SPARQL Endpoint von Corpus Nummorum vorliegt, dann wird klar, dass diese basierend auf der Syntax von RDFS aufgebaut wurde. Dies bedeutet, dass zum Beispiel die hierarchische Verbindung zwischen der URI von „Person“ und der URI von „Deities“ durch „<http://www.w3.org/2000/01/rdf-schema#isSubClass>“ definiert wird. Dies ist besonders von Bedeutung, im Hinblick auf die zukünftige Verwendung eines Reasoners. Das liegt daran, dass Reasoner aus Effizienzgründen nicht alle möglichen Befehle

⁶<https://github.com/Danilopa/Bachelorthesis>

⁷<https://www.corpus-nummorum.eu/>

abdecken können. Selbst wenn ein Reasoner die nötige Funktionalität besitzt, um einen Befehl auszuführen, bedeutet dies nicht, dass er auch effizient darin ist.

Schaut man sich nun an, wie die ikonografischen Erscheinungsbilder der Münzen innerhalb der Datenbank hinterlegt sind, stößt man darauf, dass die Daten auf zwei unterschiedlichen Art und Weisen vorliegen. Einerseits werden die auf den Münzbildern zusehenden Entitäten in einer Subjekt-Prädikat-Objekt-Beziehung gespeichert, welche auch bereits in der von Danilo Pantic und Mohammed Sayed Mahmod entwickelten Suche verwendet wird. Hierbei ist es bislang nur möglich, nach spezifischen Subjekten / Objekten, wie „Cattle“, zu suchen, aber nicht nach Gruppen, wie „Fish“ oder „Fruit“. Die große Schwachstelle der bisherigen, nicht hierarchischen, Umsetzung ist somit, dass es nur bedingt möglich ist, die Münzsuche zu generalisieren. Was ist damit gemeint? Die einzige Variante, um bis dato nach allen Münzen zu suchen, auf denen beispielsweise eine beliebige Person ein Messer hält, ist die Eingabe einer individuellen Münzbeschreibungen für jede einzelne Person. Dieses Vorgehen ist allerdings aus zwei Gründen sehr problematisch. Zum einen kann es sehr zeitaufwendig sein, eine Vielzahl von Eingaben für eine einzelne Münzsuche zu tätigen. Zum zweiten limitiert es die Präzision mit der gesucht werden kann, da alle Elemente einer Gruppe dem Nutzer bekannt sein müssen, bevor er die nötigen Münzanfragen definieren kann. Hier setzt die Idee einer hierarchischen Suche an. Ziel und Zweck unserer Umsetzung am CN ist somit, eine Möglichkeit zu bieten, mit genau einer Abfrage hierarchische Gruppierungen ikonografischer Entitäten suchen zu können. Andererseits werden die auf den Münzbildern abgebildeten Entitäten zusätzlich separat gespeichert. Mit der Implementierung einer hierarchisch-ikonografischen Suche ergibt sich die Möglichkeit umzusetzen, dass nicht nur nach Subjekten gesucht werden kann, welche in einer Subjekt-Prädikat-Objekt-Beziehung vorkommen, sondern auch nach Entitäten, welche nicht in einer solchen Beziehung gespeichert sind. Somit ist es besonders wichtig, auf die richtigen Datenpunkte zuzugreifen.

2.3 Verwendung eines Reasoners

Wie bereits erwähnt, ist es unser Ziel, die bereits bestehende Suchfunktion von Danilo Pantic und Mohammed Sayed Mahmod zu erweitern. Um unser Vorhaben umzusetzen, so dass man nicht mehr nur nach Münzbeschreibungen der Form „spezifisches Subjekt - Prädikat - spezifisches Objekt“, sondern auch nach solchen mit RDF-Tripeln der Form „spezifisches Subjekt / allgemeine Subjektgruppe - Prädikat - spezifisches Objekt

/ allgemeine Objektgruppe“ suchen kann, benötigen wir einen Reasoner.

Was spricht für den Einsatz eines Reasoners? In einer großen Datenmenge mit mehreren tausenden Datensätzen ist es sehr kostspielig, alle Verbindungen und Beziehungen zwischen den einzelnen Objekten, in einer Datenbank, abzubilden. Dies hat mehrere Gründe. Zum einen, verliert man bei einer solchen Größenordnung schnell den Überblick. Das ist auch dann der Fall, wenn die grundlegenden Verknüpfungen, der in unserem Falle vorliegenden RDF-Tripel, bereits vorhanden sind. Deshalb ist es schier unmöglich die Datenbank manuell zu vervollständigen. Davon abgesehen würde dies aufgrund der größeren Datenmenge einen enormen zeitlichen Mehraufwand, unter anderem beim Exportieren des Datensatzes, mit sich bringen. Des Weiteren würde das zusätzliche Ablegen aller Beziehungen, zwischen einzelnen Datensätzen, in der Datenbank viel Speicherplatz in Anspruch nehmen. Unter anderem deshalb werden im Falle des Projekts Corpus Nummorum nur die direkten Beziehungen einzelner Objekte gespeichert. Demzufolge ist in der Datenbank hinterlegt, dass „Artemis“, „Athena“ oder auch „Aphrodite“ zu den Gruppen „Woman“, „Greek“ und „Deity“ gehören. Im Gegensatz dazu, ist es für einen Computer ohne weitere Unterstützung unbekannt, dass sie auch jeweils zur Gruppe „Person“ gehören. Obwohl die Verbindung existent ist, dass alle Entitäten der Klasse „Greek“ auch zur Klasse „Person“ gehören sollen, ist es dem Computer nicht möglich den Rückschluss zwischen „Artemis“, „Athena“ und allen anderen Personen zur Kategorie „Person“ herzustellen. Dies hat zur Folge, dass zur Zeit bei der Tripel Eingabe „Person holding Bow“ keine Münzen gefunden werden können, da ein Computer ohne weitere Hilfe nicht eigenständig logisch-konsistente Schlüsse zieht. Folglich sind die Suchergebnisse unvollständig, da das System nicht weiß, dass Münzen mit Abbildungen, wie „Hera holding Bow“, „Demeter holding Bow“ usw. gesucht werden. Wie bereits in Abschnitt 1.3 erwähnt, übernimmt ein Reasoner genau diese Aufgabe. Durch logische Schlussfolgerungen leitet er alle Beziehungen und Verbindungen des zugrundeliegenden Datensatz auf Basis der bereits existierenden Informationen ab.

Der prinzipielle Ansatz, den eine Reasoning Engine bei der Entwicklung einer Schlussfolgerung verfolgt, besteht aus vier Schritten [Dipl.-Ing. (FH) Stefan Luber / 3]:

1. Der Reasoner erstellt eine Zusammenfassung auf der Grundlage der ihm zur Verfügung stehenden Daten. Dieses Ergebnis bildet das Grundgerüst für alle Schlussfolgerungen der Reasoning Engine .

2. Der Reasoner wendet die in der Zusammenfassung implementierten Regeln auf die Daten an.
3. Die Ergebnisse werden der Informationsbasis hinzugefügt.
4. Schritt zwei und drei werden wiederholt, bis keine neuen Schlussfolgerungen mehr gefunden werden können.

Wie wir selbst beim Testen mit der Apache-Jena-Fuseki Version 5.0.0 erfahren haben, erfolgt dieses Vorgehen temporär. Somit wird kein zusätzlicher Speicherplatz benötigt.

Die Verwendung einer Reasoning Engine sorgt dafür, dass menschliche Fehler beim Hinzufügen aller Objektbeziehungen reduziert / vermieden werden. Zusätzlich kann man durch Hilfe von Reasoning laut Prof Dr. Hellberg „Zusammenhänge in Daten erkennen, die für Menschen möglicherweise nicht offensichtlich sind“, weshalb es auch in der KI eingesetzt wird [[Prof. Dr. Hellberg / 14](#)].

3 Reasoner Auswahl für eine hierarchisch-ikonografische Suche

Zu Beginn dieses Kapitels erläutern wir unsere Auswahl an Reasonern, die für die Umsetzung einer hierarchischen Suche auf numismatischen Daten zur Diskussion stehen. Darauf aufbauend charakterisieren wir die einzelnen Reasoner. Dabei gehen wir auf ihre Vor- und Nachteile, sowie die Erkenntnisse unserer Reasoner-Tests, ein. Schließlich vergleichen wir die zuvor charakterisierten Reasoner miteinander. Zum Abschluss dieses Kapitels zeigen wir auf, welche Reasoner ungeeignet beziehungsweise geeignet für eine hierarchische Suche sind.

3.1 Welche Reasoner werden in Betracht gezogen?

Im Kontext unserer Abschlussarbeit untersuchen wir lediglich die Reasoner von Apache-Jena. Diese werden in der Dokumentation⁸ für die Verwendung von Fuseki mit aufgeführt:

- Transitive Reasoner
- Generic Rule Reasoner
- RDFS Rule Reasoner
- Full OWL Reasoner
- Mini OWL Reasoner
- Micro OWL Reasoner

Im Zuge unserer Testphase mit der Apache-Jena-Fuseki Version 5.0.0 werden wir den Einsatz der verschiedenen Apache-Jena Reasoner testen. Unsere Auswahl und Testumgebung begründen sich vor allem damit, dass der SPARQL Endpunkt beim Projekt Corpus Nummorum auf Apache-Jena-Fuseki basiert. Daraus folgend wäre es weniger sinnvoll, sich mit anderen, als den von Apache-Jena selbst entwickelten, Reasonern zu beschäftigen. Daher wäre auch die Betrachtung der weit verbreiteten Reasoner Pellet⁹ und HermiT¹⁰ nicht zweckhaft, da diese auf OWL 2 Syntax basieren, während

⁸<https://jena.apache.org/documentation/fuseki2/fuseki-configuration.html>

⁹<https://github.com/stardog-union/pellet>

¹⁰<http://www.hermit-reasoner.com/>

Apache-Jena standardmäßig OWL 1 Syntax unterstützt. Außerdem würde es den zeitlichen Rahmen unserer Arbeit überschreiten, wenn die Anzahl der zu untersuchenden Reasoner zu groß wird. Schlussendlich wäre es somit ebenso wenig zielführend, wenn wir uns mit der Entwicklung eines eigenen, an die von Apache-Jena angelehnten, Reasoners befassen. Zum einen ist das nicht das Ziel unserer Arbeit und zum anderen bringt es ein enormes Fehlerpotential sowie einen großen Zeitaufwand mit sich. Dies ist aus unserer Sicht im zeitlichen Rahmen der Bachelorarbeit nicht realistisch umsetzbar und wurde von unserem Professor nicht gefordert.

3.2 Charakterisierung der zur Auswahl stehenden Reasoner

Im vorhergehenden Abschnitt wurde erläutert, welche Reasoner für die Implementierung am Corpus Nummorum zur Auswahl stehen. Im Folgenden werden wir nun diese sechs Reasoner, speziell im Fokus auf ihre Eigenschaften, näher betrachten. Im Anschluss werden die zuvor charakterisierten Reasoner tabellarisch verglichen.

3.2.1 Transitive Reasoner

Der Transitive Reasoner hat im Vergleich zu den anderen Reasonern einen sehr stark eingeschränkten Funktionsbereich. Dies geht damit einher, dass er nur zwei Arten von Ableitungsregeln durchführen kann. Jene sind „`rdfs:subPropertyOf`“ und „`rdfs:subClassOf`“ [[Apache Software Foundation / 15](#)].

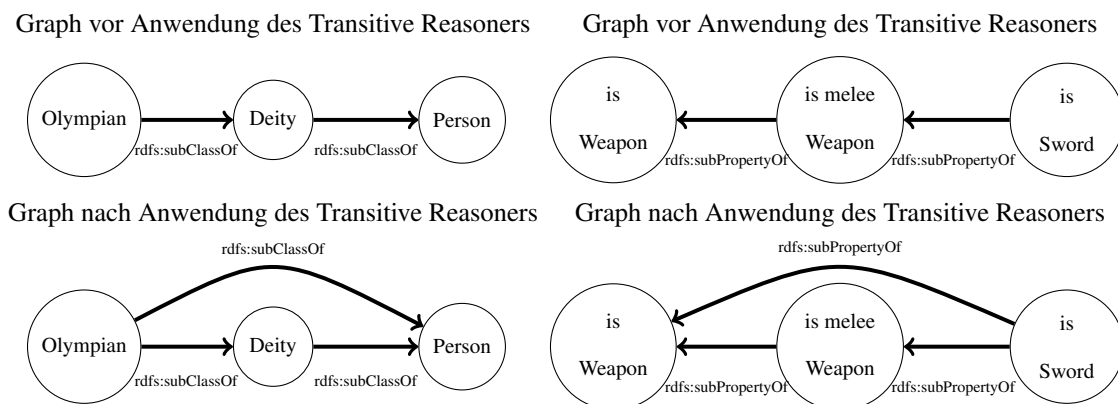


Abbildung 3: Visualisierung der transitiven Beziehungen

Wie in den obigen Graphen veranschaulicht wird, werden anhand der beiden Eigenschaften logische Schlussfolgerungen wie folgt erlangt: Wenn gegeben ist, dass ein Objekt in Beziehung zu zwei anderen Objekten steht, dann folgert der Transitive Reasoner, dass auch die beiden anderen Objekte in einer Beziehung zueinander stehen müssen. Zum einen bedeutet dies also, dass wenn eine „`rdfs:subClassOf`“ Beziehung von „Olympian“ zu „Deity“ und von „Deity“ zu „Person“ vorliegt, dass dann auch eine solche Beziehung von „Olympian“ zu „Person“ existieren muss. Des Weiteren zeigt die obige Abbildung, dass wenn eine „`rdfs:subPropertyOf`“ von „`is sword`“ zu „`is melee weapon`“ und gleichzeitig von „`is melee weapon`“ zu „`is weapon`“ existiert, dann liegt nach Anwendung des Reasoners eine solche Beziehung ebenso zwischen „`is sword`“ und „`is weapon`“ vor. Der Reasoner ist beim Schlussfolgern auf seinen beiden Eigenschaften in Bezug auf Laufzeit und Speicherverbrauch effizienter als eine rein regelbasierte Implementierung. Deshalb wird er standardmäßig vom RDFS Rule Reasoner und Micro OWL Reasoner für diese Herleitungen verwendet. Auch der Generic Rule Reasoner bietet die optionale Möglichkeit den Transitive Reasoner einzusetzen [[Apache Software Foundation / 15](#)].

3.2.2 Generic Rule Reasoner

Ein weiterer von Apache-Jena entwickelter Reasoner ist der Generic Rule Reasoner. Er ist eine Spezialisierung des FB Rule Reasoners [[GRG / 16](#)]. Es ist ein regelbasierter und auf RDF-Eingabedaten gestützter Reasoner. Er ermöglicht es dem Benutzer, eigens formulierte Regeln anzuwenden [[Apache Software Foundation / 15](#)]. Diese Regeln stehen in einer separaten Regel Datei, einer „`.ttl`“ Datei. Diese bezeichnen wir im Folgenden als „`rules.ttl`“. Hierbei werden zu Beginn nötige Abkürzungen definiert, um die Regeln übersichtlicher und kompakter formulieren zu können. Eine Abkürzung wird als Präfix bezeichnet und in einer Regeldatei wie folgt definiert: „`@prefix prefixname: <uri>.`“

Jede Regel besteht aus einer / mehreren Bedingungen und einer Schlussfolgerung. Beim Testen mit der Apache-Jena-Fuseki Version 5.0.0 ist uns aufgefallen, dass es besonders wichtig ist, die Regeln nach folgender Syntax zu definieren: „`(Bedingung1) (Bedingung2) (..) .. → (Schlussfolgerung).`“. Hierbei gilt es vor allem zu beachten, einzelne Bedingungen zu umklammern und zwischen den Klammern das Leerzeichen nicht zu vergessen. Des Weiteren ist es sehr wichtig, einen Punkt am Ende jeder Regel zu setzen. Das Missachten des zuvor beschriebenen hat bei uns einige

Fehler erzeugt. Ein einfaches Beispiel für eine vollständige „rules.ttl“ wäre somit:

```
@PREFIX owl: <http://www.w3.org/2002/07/owl#>
(?x owl:sameAs ?y) → (?y owl:sameAs ?x).
```

Abbildung 4: Beispiel einer „rules.ttl“

Um die Funktionsweise und die Ergebnisse des Generic Rule Reasoners zu überprüfen, haben wir beim Testen mit der Apache-Jena-Fuseki Version 5.0.0 eine solche „rules.ttl“ in die Config-Datei eingebunden. Mit der Config-Datei haben wir Apache-Jena-Fuseki schließlich gestartet und unter „http://localhost:3030/“ aufgerufen. Gemäß [Gen / 17] soll es möglich sein, die Regeln auch erst nach der Erstellung der Reasoner-Instanz festzulegen. Das ist uns beim Testen jedoch nicht gelungen. Wir haben die für den Reasoner bestimmten Regeln in jedem Testfall vor dem Aufrufen des localhost definiert.

Aus der offiziellen Dokumentation von Apache-Jena geht zudem hervor, dass der hier beschriebene Generic Rule Reasoner ein Interface bzw. eine Schnittstelle darstellt. Diese sei [Gen / 17] in der Lage, jede „rule engine combinations“ aufzurufen und zu nutzen. Jegliche „rule engine combinations“ ist eine Einstellung, beziehungsweise ein Modus, in dem der Reasoner eingestellt werden kann. In Entsprechung zur Quelle [Apache Software Foundation / 15] gibt es die folgenden drei gängigen Modi: Zum einen existiert der Vorwärtsmodus durch die „Forward chaining engine“. Des Weiteren kann der Reasoner auch im Rückwärtsmodus per „Backward chaining engine“ eingestellt werden. Möchte man nun die Vorteile beider Modi kombinieren, so ist auch die Einstellung per „Hybrid rule engine“ möglich [Apache Software Foundation / 15].

Standardmäßig arbeitet der Generic Rule Reasoner unseres Kenntnisstandes nach, durch Vorwärtsverkettung, also mit der „Forward chaining engine“. Um den Modus in der Config-Datei zu ändern, kann man ihn mit Hilfe „ja:mode“ „MODUSNAME“ anpassen. Hierbei ist „MODUSNAME“ durch „FORWARD“, „BACKWARD“ oder „HYBRID“ zu ersetzen. Aus [Gen / 17] schließen wir, dass das Einstellen des gewünschten Modus in einer Programmiersprache beispielsweise durch die Methode „setMode“ erreichbar ist.

Die in der „Forward chaining engine“ eingesetzte Vorwärtsverkettung, ist laut [Prof. Dr. Richard Lackes, Dr. Markus Siepermann / 18] eine „Vorgehensweise, bei der

man von einer Anfangssituation auf die Endsituation schließt“. Entsprechend der offiziellen Wikipedia Definition des Wortes „Vorwärtsverkettung“, ist die Vorgehensweise transitiv. Durch eine Bedingung wird eine Erkenntnis gefolgert, welche wieder als Bedingung verwendet werden kann / wird. Dieses Prinzip wird angewendet, bis keine neuen Erkenntnisse mehr abgeleitet werden können [Wikipedia / 19]. [Apache Software Foundation / 15] legt dar, dass bei der Konfiguration per „Forward chaining engine“, welche auf dem RETE-Algorithmus basiert, eben nur jenes Vorwärtsverkettungsprinzip verwendet wird. Alle ausgeführten Regeln werden als Vorwärtsregeln behandelt. Dies ist auch dann der Fall, wenn sie in der Rückwartssyntax (\leftarrow) geschrieben sind. In der sogenannten Vorbereitungsphase werden die benötigten Modelldaten an die Regelengine übermittelt. Dies geschieht [Apache Software Foundation / 15] zufolge nicht nur bei der ersten Abfrage des Inferenzmodells, sondern bei etwaigen „prepare“()-Aufrufen. Währenddessen sorgen ausgelöste Regeln dafür, dass neue Tripel generiert werden, bis keine weiteren Regel mehr ausgelöst werden können. Hierbei verhält sich das Inferenzdiagramm so, als wenn die neu abgeleiteten Tripel schon zuvor zum Datenbestand gehört hätten. Gemäß [Apache Software Foundation / 15] ist es auch möglich, bestehende / generierte Tripel zu entfernen. Allerdings sollte man beachten, dass ein Entfernen von Tripeln dafür sorgen kann, dass weitere Regeln ausgelöst werden, welche unter Umständen nur durch das Entfernen von Tripeln aktiviert werden. Beim Testen mit der Apache-Jena-Fuseki Version 5.0.0 haben wir dies nicht ausprobiert, da das Entfernen von Tripeln für unsere Arbeit nicht relevant ist. Wir möchten schließlich nur, noch nicht in der Datenbank vorhandene, Beziehungen zwischen Objekten erzeugen. Weder ist es für unserer Vorhaben nötig die in der Datenbank existierenden, noch die neu generierten, Beziehungen wieder zu entfernen. Desto mehr Beziehungen existieren, desto einfacher ist es, eine hierarchische Suche zu implementieren. Dies hat den einfachen Grund, dass dann mehr Informationen vorliegen. Der offiziellen Apache-Jena-Dokumentation [Apache Software Foundation / 15] zufolge ist es auch möglich Regeln zu formulieren, welche unendlich lange dafür sorgen können, dass neue Tripel erzeugt und weitere Regeln abgeleitet werden können. Somit sei also das Erzeugen einer unendlich Schleife, durch ungünstig formulierte Regeln, möglich. Hierzu sei gesagt, dass weder beim Testen mit einer kleiner Datenbasis, noch beim Testen mit dem Datenbestand vom Projekt Corpus Nummorum eine von uns definierte Regel zu einer unendlichen Generierung von Tripeln geführt hat. Das lässt sich deshalb sicher sagen, da sonst unser Programm durch das Aktivieren des Reasoners abgestürzt wäre. Zwar ist unser Programm in einzelnen Fällen abgestürzt, jenes war jedoch anderen Ursachen bedingt. Diese waren beispielsweise, das falsche Einbinden

der Config-Datei oder Fehler in der Syntax der „rules.ttl“. Apache-Jena gibt in ihrer offiziellen Dokumentation [[Apache Software Foundation / 15](#)] zudem an, dass ein separater Zugriff auf die bereits vorhanden gewesenen Tripel und die neu abgeleiteten Aussagen im Vorwärtsmodus möglich sei. Dazu können wir sagen, dass uns beim Testen mit der Apache-Jena-Fuseki Version 5.0.0 zwei Aspekte aufgefallen sind: Es ist möglich, unabhängig vom eingestellten Modus, bereits zuvor existierende Tripel, als auch neu abgeleitete Tripel zu sehen und zu bearbeiten. Hierfür navigiert man unter „http://localhost:3030/“ zu „datasets“ dann zu „edit“ und dann zu „list current graphs“. Hier kann man auf alle Tripel zu greifen. Allerdings ist es nicht ersichtlich, welche Informationen bereits zuvor existiert haben und welche erst durch den Einsatz des Reasoners erzeugt wurden. Dementsprechend gehen wir davon aus, dass sich diese Aussage auf die Verwendung des Reasoners in einer Programmiersprache bezieht [[Apache Software Foundation / 15](#)].

Beim Verwenden des Generic Rule Reasoners im Rückwärtsmodus, durch die „Backward chaining“, werden nur die Regeln angewendet, die das gesuchte Ziel in ihrer Schlussfolgerung beinhalten. Zu Beginn wird die Abfrage in ein Ziel übersetzt und anschließend per Abgleich aller gespeicherten Tripel und der Auflösung anhand der Rückwärtsverkettung-Regeln versucht zu erfüllen [[Apache Software Foundation / 15](#)]. Anders formuliert, ist der Rückwärtsmodus der Definition von [[Prof. Dr. Richard Lackes, Dr. Markus Siepermann / 20](#)] zufolge eine „Vorgehensweise bei der man mit dem Endziel beginnt (d.h. mit dem Sachverhalt, den man aufgrund der Problemstellung erreichen möchte); dieses Ziel wird in Unterziele aufgeteilt, diese werden ebenfalls wieder aufgeteilt etc., bis die Ziele elementare Fakten sind, von denen man weiß, ob sie zutreffen oder nicht“. Ausgehend vom Ziel werden in dieser Konfigurationseinstellung des Reasoners die definierten Regeln, wie auch bei der Vorwärtsverkettung, transitiv miteinander verknüpft und schließlich von oben nach unten, von links nach rechts mit Rückverfolgung ausgeführt [[Apache Software Foundation, Wikipedia / 15, 21](#)]. Wenn der Wert eines Tripels in der Bedingung einer zu überprüfenden Regel noch nicht bekannt ist, so wird versucht, ihn mit Hilfe anderer Regeln abzuleiten. [[Wikipedia / 21](#)]. Die Regeln sind im Wesentlichen im Datalog formuliert. Da es sich bei der Regelsprache also nicht um einen vollständigen Prolog handelt, ist es nicht möglich, komplexe Regeln zu formulieren, welche in der Lage sind, stark verschachtelte oder rekursive Datenstrukturen zu erzeugen. [[Apache Software Foundation / 15](#)]. Demzufolge müssen die Regeln einfach formuliert werden.

Möchte man nun die Vorteile beider Ansätze kombinieren, so werden sowohl die „Forward chaining engine“ als auch die „Backward chaining engine“ in der sogenannten „Hybrid rule engine“ eingesetzt. Hierbei wird zuerst die „Forward chaining engine“ eingesetzt um neue Tripel, beziehungsweise Schlussfolgerungen, zu erzeugen, welche im Inferenzmodell zwischengespeichert werden. Gehen wir nun davon aus, es wird eine Anfrage gestellt, die ein Tripel fordert, welches so noch nicht im Inferenzmodell existiert, was passiert dann? Für solche Fälle wird die „Backward chaining engine“ eingesetzt. Durch den Einsatz der „Backward chaining engine“ wird, wie zuvor beschrieben, überprüft, ob durch Anwenden der zur Verfügung stehenden Regeln das gesuchte Tripel abgeleitet werden kann. Ist dies der Fall, so wird durch Einsatz der „Backward chaining engine“ dieses Tripel nachträglich erzeugt.

Durch dieses Vorgehen ist es möglich eine höhere Leistung / Effizienz zu erzielen, da nur die Rückwärtsregeln eingesetzt werden die für die vorliegende Problematik von Relevanz sind. Dieser Ansatz ist deshalb so effizient, da allgemeinere Schlussfolgerungen im Voraus berechnet werden. Hingegen werden für spezifischere Abfragen nur relevante (rückwärts) Regeln eingesetzt [[Apache Software Foundation / 15](#)]. Beim Testen ist uns aufgefallen, dass unabhängig vom eingestellten Modus, die Regeln in einer zufälligen Reihenfolge ausgeführt werden.

Neben dem Bereitstellen verschiedener Modi, ermöglicht der Generic Rule Reasoner gemäß [[Apache Software Foundation / 15](#)] auch die Einbindung OWL spezifischer „Regelerweiterungen“. Beim Testen des Reasoner haben wir durch eigene Regeln unter anderem „owl:sameAs“ und „owl:equivalentClass“ nach definiert.

Des Weiteren lässt sich über den bereits beschriebenen Reasoner sagen, dass er eine Vielzahl an Methoden besitzt. Darunter befinden sich sowohl für den Reasoner spezifische Methoden als auch Methoden, die durch die Vererbung des FB Rule Reasoner nutzbar sind. Für die Implementierung einer hierarchischen Suche sind vor allem folgende Methoden interessant: Durch die Methode „setOWLTransition“ ist es möglich, die Einbindung beziehungsweise Übersetzung von OWL-Schemata in Regeln zu aktivieren / deaktivieren. Dies ist vor allem dann von Vorteil / notwendig, wenn Teile der Datenbasis auf OWL basieren, beziehungsweise zukünftig OWL-Schemata in die Datengrundlage integriert werden soll. Mit Hilfe der Methode „setRules“, kann man die für den Reasoner zu beachtenden Regeln festlegen. Durch Nutzen der Methode „setFunctorFiltering“ kann man angeben, ob bestimmte Operatoren aus den RDF-Tripeln „in Literalen von Regeln im Ausgabemodell entfernt werden sollen“. All-

gemein ist die Methode zwar sehr sinnvoll, für unsere Arbeit am Corpus Nummorum ist sie allerdings, wie bereits zuvor beschrieben, nicht zielführend. Des Weiteren kann die Methode „setMode“ sehr nützlich sein. Hiermit kann man einen der zuvor beschriebenen Inferenzmodi einstellen. Auch das setzen von Konfigurationsparametern durch Einsetzen der Methode „setParameter“ kann von Vorteil sein. Zusätzlich kann es nützlich sein, unabhängig vom Verwendungszweck des Reasoners, mit setTraceOn den Tracing Modus, eine Art Debug-Modus zu aktivieren / deaktivieren [[Apache Software Foundation / 15](#)].

Abschließend lässt sich bezüglich des Generic Rule Reasoners folgendes sagen: Der Reasoner kommt vor allem mit dem Vorteil einher, dass benutzerdefinierte Regeln erstellt werden können, nach welchen der Generic Rule Reasoner logische Schlussfolgerungen zieht. Diese besitzen zumeist eine einfache und verständliche Syntax. Durch die Möglichkeit der Integration des Transitive Reasoners können, wie bereits erwähnt, auf einfache Art und Weise transitive Eigenschaften festgelegt werden. Für die Umsetzung einer hierarchischen Suche ist dies besonders von Vorteil. Zwar unterstützen die Regeln keine Selbstreferenzen und „verschachtelte“ Regeln, im Hinblick auf unsere Nutzung spielt dies aber keine große Rolle. Ein weiterer Vorteil des Reasoners ist, dass OWL-Syntax eingebunden werden kann und Eigenschaften, wie „owl:sameAs“ und „owl:equivalentClass“, definiert werden können. Des Weiteren lässt sich positiv erwähnen, dass verschieden Modi eingestellt werden können, nach welchen der Reasoner, auf unterschiedliche Art und Weise, logische Schlussfolgerungen ziehen kann [[Apache Software Foundation / 15](#)]. Für die Umsetzung einer hierarchischen Suche haben wir mit der Apache-Jena-Fuseki Version 5.0.0 maßgeblich den Vorwärtsmodus getestet. Dieser empfiehlt sich hierbei, denn es wurde schnell sowohl „Artemis ist eine Person“ hergeleitet, als auch „Zu der Gruppe von Personen gehört Artemis“. Im Hinblick auf den Einsatz für die Implementierung einer hierarchischen Suche gilt jedoch zu beachten, dass die Art der Datenverarbeitung bei einer großen Datenmenge nicht besonders effizient sein soll [[Apache Software Foundation / 15](#)]. Beim Testen mit der Apache-Jena-Fuseki Version 5.0.0 haben wir mit dem aktuellen Dump vom Corpus Nummorum dahingehend zwar keine Probleme gehabt, bei weiteren deutlichen Vergrößerungen des Datenbestandes könnte dies jedoch zu Laufzeit-Problemen führen.

3.2.3 RDFS Rule Reasoner

Eine Unterklasse des zuvor beschriebenen Reasoners ist der RDFS Rule Reasoner. Dieser Reasoner gehört ebenfalls wie der Generic Rule Reasoner zur Klasse der FB Rule Reasoner [[RDF / 22](#)].

Die Eingabedaten des Reasoners entsprechen dem RDF-Format. Die Regeln, nach welchen der Reasoner logische Schlussfolgerungen zieht, kommen aus dem RDF-Schema. Der Reasoner ist effizient darin, Beziehungen von Klassen, sowie ihre Eigenschaften, zu erkennen. Hieraus lassen sich einfach neue Erkenntnisse ableiten. Der von Jena entwickelte Reasoner beinhaltet weitgehend alle der RDF-Core-Arbeitsgruppe beschriebenen RDFS-Konsequenzen. Hierbei wurden seitens Jena bewusst nicht alle Konsequenzen beachtet, da nicht alle als nötig empfunden wurden [[Apache Software Foundation / 15](#)].

Für den RDFS Rule Reasoner gibt es, genau wie für den Generic Rule Reasoner, drei unterschiedliche Modi, in denen der Reasoner konfiguriert werden kann. Diese werden als „Compliance levels“ bezeichnet und heißen „Full“, „Default“ und „Simple“.

Die Konfiguration im „Full compliance level“ sorgt dafür, alle RDFS-Axiome und Regeln, ausgenommen der „bNode Closure“ Regeln und Datentypen (rdfD 1), implementiert werden. Die „bNode Closure“ Regeln beschreiben hierbei ausschließlich die Handhabung mit Leerzeichen. Im Rahmen von RDF-Graphen stellen Leerzeichen anonyme Knoten dar. Anonyme Knoten sind Knoten, die weder eine URI noch einen gleichbleibenden Wert besitzen. Solche Knoten können deshalb nicht eindeutig referenziert werden. Angesichts der Tatsache, dass das Fehlen einer eindeutigen URI beziehungsweise eines konstanten Wertes das Ziehen logischer Schlussfolgerungen im Zusammenhang mit anonymen Knoten ohnehin unterbindet, ist eine Implementierung dieser Regeln obsolet. Stattdessen kann ein Graph, der Leerzeichen beinhaltet übersetzt werden. Die Übersetzung sorgt dafür, dass die Leerzeichen durch Variablen ersetzt werden. Andernfalls treten Probleme in den Abfragen auf. Zum „Full compliance level“ muss des Weiteren gesagt werden, dass hierbei ein großer Ressourcen Aufwand entsteht. Die Ursache hierfür liegt darin, dass jedes im Graphen vorhandene Tripel-Element dahingehend überprüft werden muss, ob mindestens eine sogenannte „rdfs:ContainerMembershipProperty“ verwendet wird [[Apache Software Foundation /](#)

15]. Ein Container stellt hierbei eine Sammlung von Ressourcen dar. Hierbei werden „rdf:Container“ in drei Kategorien aufgeteilt. Die Kategorien sind „rdf:bag“, „rdf:seq“ und „rdf:alt“. „rdf:bag“ stellt hierbei eine einfache ungeordnete Menge von Ressourcen dar. Um hingegen, basierend auf den „rdfs:ContainerMembershipProperty“, eine sortierte Kollektion zu bilden, wird „rdf:seq“ verwendet. Die letzte der drei Kategorien, „rdf:alt“, hat den Zweck, alternative Optionen anzubieten. Für beispielsweise einen Namen könnte der englische Name die Standard Variante sein. Die Übersetzung in andere Sprachen, wie Deutsch, Griechisch oder Japanisch könnten hierbei die Alternativen repräsentieren [W3C, Pierre-Antoine / 23, 24].

Im „Default compliance level“ wird auf die Prüfung auf „rdfs:ContainerMembershipProperty“ verzichtet. Die Konfiguration in diesem Modus sorgt dafür, dass nicht, wie im „Full compliance level“, alles als eine Ressource gesehen wird. Dafür umfasst dieser Modus alle axiomatischen Regeln. Als Beispiel wird hierfür von der offiziellen Apache-Jena Dokumentation [Apache Software Foundation / 15] die Abfrage eines „leeren“ RDFS InfModel angegeben. Selbst in einem solchen Fall werden demnach Tripel wie „[rdf:type rdfs:range rdfs:Class]“ zurückgegeben.

Die einfachste und überschaubarste Konfiguration des RDFS Rule Reasoners erfolgt durch das „Simple compliance level“. Hierbei werden, von denen für eine hierarchische Suche nützliche Eigenschaften nur „rdf:subPropertyOf“- und „rdf:subClassOf“-Beziehungen implementiert. Im Gegensatz zum „Default compliance level“ werden in diesem Modi alle Axiome ignoriert. Apache-Jena bezeichnet diesen Modus trotzdem als den „wahrscheinlich nützlichsten“ der drei zur Verfügung stehenden Modi. Als Begründung, wieso dieser Modus nicht der Standard Konfigurationsmodus ist, wird hierbei lediglich die unvollständige Implementierung genannt [Apache Software Foundation / 15]. Hierzu sei gesagt, dass wir beim Testen des RDFS Rule Reasoner mit der Apache-Jena-Fuseki Version 5.0.0 den Reasoner ausschließlich im „Default“ Modus getestet haben. Das hängt damit zusammen, dass wir die anderen Modi des Reasoners nicht in unserer Config Datei einstellen konnten. Wir haben zwar versucht den „Simple“ Modus zu aktivieren, für „rdfs:subClassOf“ und „rdfs:subPropertyOf“ Beziehungen haben wir allerdings nur Fehler erhalten. Beim Verwenden des „Default“ Modus haben wir keine Schwierigkeiten gehabt.

Über den RDFS Rule Reasoner lässt sich zudem sagen, dass er eine Vielzahl an Methoden besitzt. Zum einen umfassen diese für den Reasoner spezifische Methoden

als auch zum anderen die vom FB Rule Reasoner geerbten Methoden [[Apache Software Foundation / 15](#)]. Beim Testen des Reasoners haben wir keine seiner Methoden explizit getestet. Allerdings haben wir beim Testen des Reasoners versucht, OWL Syntax zu aktivieren / zu nutzen und eigene Regeln zu nutzen. Dies hat jedoch nicht funktioniert, obwohl der RDFS Rule Reasoner gemäß [[RDF / 22](#)] den Generic Rule Reasoner erweitert beziehungsweise eine Unterklasse von jenem ist. Unserem Verständnis nach müsste es somit möglich sein, wir haben es aber nicht umgesetzt bekommen.

In unseren Untersuchungen haben wir die Vollständigkeit des „rdfs:Class“ Konstruktes beim Anwenden des Reasoners mit den Ergebnissen des Generic Rule Reasoners verglichen. Dabei sind wir zu der Erkenntnis gekommen, dass der Datensatz unter Anwendung des RDFS Rule Reasoner genau 94 „rdfs:Class“ Beziehungen für „https://www.wikidata.org/wiki/“ URI's besitzt. Dies entspricht genau der Anzahl an Beziehungen, die vorliegen, wenn der Generic Rule Reasoner mit unseren selbst erstellten Regeln verwendet wird. Beim Testen des RDFS Rule Reasoners haben wir außerdem die Erkenntnis gewonnen, dass er bei einer größer werdenden Datenmenge an Leistung verliert. Das stimmt mit den Ergebnissen eines von Apache-Jena veröffentlichten Leistungsvergleiches überein:

Set	#concepts	total instances	#instances of concept	JenaRDFS
1	155	1550	310	0.07
2	780	7800	1560	0.25
3	3905	39050	7810	1.16

Abbildung 5: Leistungsinformationen des RDFS Rule Reasoners bei unterschiedlichen Datenmengen

Abschließend lässt sich zum RDFS Rule Reasoner sagen, dass er potenziell für die Implementierung einer hierarchischen Suche auf RDF-Daten geeignet sein kann. Dies liegt vor allem daran, dass er auf RDF-Daten optimiert ist und schnell Beziehungen von Klassen und deren Eigenschaften erkennen, sowie neue Vererbungsbeziehungen ableiten, kann. Die für eine hierarchische Suche benötigte Eigenschaft „rdfs:subClassOf“ unterstützt der Reasoner. Ein weiterer positiver Aspekt des Reasoners ist es, dass er in drei unterschiedlichen Modi konfiguriert werden kann. Seitens Apache ist hierbei vor allem der „Simple“ Modus zu empfehlen, gemäß unseren Erfahrungen präferieren wir den „Default“ Modus. Der Reasoner kommt allerdings mit dem Nachteil einher, dass

ausschließlich RDFS-Regeln unterstützt werden. Zudem werden Apache-Jena zufolge „zukünftige Arbeiten zur Anpassung der Regel-Engines zur Nutzung der Fähigkeiten der anspruchsvolleren Datenbank-Backend“ in Betracht gezogen [[Apache Software Foundation / 15](#)]. Bei Verwendung dieses Reasoners könnte es somit zukünftig passieren, dass man Anpassungen vornehmen muss.

3.2.4 OWL Reasoner

Apache-Jena hat zusätzlich zu den bereits beschriebenen Reasonern einen Reasoner basierend auf der Web Ontology Language (OWL) von W3C entwickelt. Hierbei ist zu beachten, dass, wie in Abschnitt 1.3 bereits erwähnt, OWL zwei verschiedene Definitionen, OWL 1 und OWL 2, besitzt [[W3C / 8](#)]. Dies ist von Bedeutung, da der OWL Reasoner von Apache-Jena auf der älteren OWL 1 Definition basiert. Dabei ist der Reasoner eine Implementation der OWL Lite Sprache [[Apache Software Foundation / 15](#)]. Dies hat zur Folge, dass der Reasoner nicht garantieren kann, in seinen Schlussfolgerungen zu terminieren. Des Weiteren unterstützt der Reasoner nicht alle OWL-Konstrukte [[W3C / 8](#)]. Schaut man sich an, welche Funktionen der Reasoner nicht unterstützt, so fällt auf, dass „owl:complementOf“ und „owl:oneOf“ überhaupt nicht unterstützt werden. Des Weiteren funktioniert „owl:unionOf“ nur in einer limitierter Form. Das hängt damit zusammen, dass zwar von den Teilmengen auf die Vereinigte Menge geschlussfolgert werden kann, jedoch nicht von der vereinten Menge auf die Teilmengen. Das Ziel, welches Apache-Jena mit diesem Reasoner erreichen wollte, war es, einen Reasoner zu entwickeln, welcher alle Funktionalitäten ihres RDFS Reasoners abdeckt und zusätzlich Teile der OWL Syntax unterstützt. Im Sinne dessen, geht der Reasoner über den Rahmen der OWL Lite Sprache hinaus. Das liegt daran, dass nur die komplexeste der OWL 1 Sprachen, OWL Full, Support für die Syntax von RDFS bietet. Der OWL Reasoner von Apache-Jena existiert in drei Versionen: Full, Mini und Micro. Hierbei ist zu beachten, dass der OWL Reasoner von Apache-Jena, besonders in Bezug auf die komplexeren Versionen, im Bereich der Speichernutzung gegenüber anderen OWL Reasonern, wie zum Beispiel Pellet, nicht gleichwertig ist. Dies hat sich auch in unseren Tests widerspiegelt, denn auf der vollständigen Datenbank des Projekts Corpus Nummorum haben die Full und Mini Version des OWL Reasoner einen „OutOfMemory Error“ erzeugt. Jedoch sind bereits zukünftige Verbesserungen diesbezüglich seitens Apache-Jena geplant [[Apache Software Foundation / 15](#)].

3.2.4.1 Full OWL Reasoner

Der Full OWL Reasoner bietet von allen drei Versionen des Reasoners die größte Funktionalität. Im Rahmen dessen unterstützt der Reasoner alle Konstrukte der anderen beiden Versionen. Hierbei wird in einigen Fällen, wie zum Beispiel der Nutzung von „owl:someValuesFrom“, zusätzlich garantiert, dass die Schlussfolgerung terminiert. Diese erweiterte Implementierung kommt jedoch auch mit ihren Nachteilen einher. Ein komplexerer Reasoner braucht entsprechend mehr Zeit für seine Schlussfolgerungen. Dadurch ist er im Punkt Effizienz schlechter als die beiden anderen Versionen [[Apache Software Foundation / 15](#)]. Je nach Situation kann es sogar der Fall sein, dass die Full Version des Reasoners um mehr als ein 50-faches langsamer arbeitet, als die Micro Version [[José R. Hilera, Luis Fernández-Sanz and Adela Díez / 25](#)]. Auch im Vergleich mit dem RDFS Rule Reasoners, welchen der OWL Reasoner erweitert, fällt die schlechte Effizienz auf. Der Grund dafür ist, dass beim Full OWL Reasoner einige Optimierungen des RDFS Rule Reasoners nicht implementiert wurden. Dies hat zur Folge, dass der RDFS Rule Reasoner auf RDFS Konstrukten um ein Vielfaches schneller arbeitet, als der Full OWL Reasoner [[Apache Software Foundation / 15](#)].

3.2.4.2 Mini OWL Reasoner

Der Mini OWL Reasoner ist eine effizientere Version des Full OWL Reasoners. Dies bedeutet, dass der Reasoner genau die gleichen Konstrukte unterstützt, wie der Full OWL Reasoner. Hierbei ist er allerdings schneller in seinen Schlussfolgerungen [[Apache Software Foundation / 15](#)]. Genauer gesagt wurde bewiesen, dass die Mini Version des OWL Reasoners um ein Vierfaches schneller in der Entwicklung von Schlussfolgerungen sein kann [[José R. Hilera, Luis Fernández-Sanz and Adela Díez / 25](#)]. Diese höhere Effizienz wird durch eine niedrigere Komplexität erreicht. Das Reduzieren der Komplexität kommt jedoch nicht ohne Kosten einher. Dies ist der Tatsache geschuldet, dass der Mini OWL Reasoner bei manchen Konstrukten, wie zum Beispiel „owl:someValuesFrom“, sogenannte „bNode introduction“ nicht implementiert. Dies kann in vereinzelt Fällen dazu führen, dass der Mini Reasoner in seinen Schlussfolgerungen nicht terminiert. Außerdem lässt der Mini OWL Reasoner, genauso wie der Full OWL Reasoner, einige Optimierungen des RDFS Rule Reasoners aus. Deswegen folgt auch in diesem Fall, dass im Falle von Schlussfolgerungen auf RDFS Konstrukten der RDFS Rule Reasoner um ein Vielfaches schneller ist [[Apache Software Foundation / 15](#)].

3.2.4.3 Micro OWL Reasoner

Der Micro OWL Reasoner ist die am meisten auf Performance ausgelegte Version der OWL Reasoner. Anders als bei den Full und Mini OWL Reasonern, implementiert der Reasoner die selben Optimierungen wie der RDFS Rule Reasoner. In Kombination mit der geringsten Komplexität führt dies dazu, dass der Micro OWL Reasoner als einzige der drei Optionen Schlussfolgerungen in einem ähnlichen Zeitraum wie der RDFS Rule Reasoner durchführen kann. Dies hat zur Folge, dass der Micro OWL Reasoner die meisten Verluste im Bereich der Funktionalität machen muss. Hierdurch unterstützt der Micro OWL Reasoner einige Konstrukte, wie zum Beispiel „owl:cardinality“, „owl:allValuesFrom“ und „owl:someValuesFrom“, überhaupt nicht, anstatt, wie der Mini OWL Reasoner, nur auf die Funktionalität des garantierten Terminierens mancher OWL-Konstrukte zu verzichten [[Apache Software Foundation / 15](#)].

3.2.5 Reasoner Übersicht

Eigenschaften	Transitiv-Reasoner	GenericRule Reasoner	RDFS Reasoner	OWLMicro Reasoner	OWLMini Reasoner	OWLFull Reasoner
spezifische RDFS Syntax						
rdfs:subClassOf	✓	✓ *	✓	✓	✓	✓
rdf:typeOf	✗	✓ *	✓	✓	✓	✓
weitere RDFS Konstrukte**	✗	✓ *	✓	✓	✓	✓
spezifische OWL Syntax						
owl:equivalentClass	✗	✓ *	✗	✓	✓	✓
owl:sameAs	✗	✓ *	✗	✗	✓	✓
weitere OWL-Konstrukte**	✗	✓ *	✗	✓	✓	✓
Benutzerdefinierte Regeln	✗	✓	✗	✗	✗	✗

* Wird nur über benutzerdefinierte Regeln und Einstellungen ermöglicht

** Gibt an, dass weitere, aber nicht zwingend alle Konstrukte unterstützt werden.

3.3 Welcher Reasoner bietet sich am besten für die Umsetzung einer solchen Suche an?

Zuvor haben wir uns einen Überblick über die Eigenschaften der potenziellen Reasoner verschafft. Nun stellt sich die Frage, welcher Reasoner im Hinblick auf die Implementierung einer hierarchischen Suche, insbesondere in Bezug auf das Projekt Corpus Nummorum, am besten geeignet ist. In diesem Abschnitt gehen wir zunächst darauf ein, welche Reasoner hierfür nicht geeignet sind. Anschließend zeigen wir auf, welche Reasoner, aus welchen Gründen, im Allgemeinen für die Umsetzung einer hierarchischen

Suche geeignet sind. Abschließend erörtern wir, welcher Reasoner sich im speziellen für das Projekt Corpus Nummorum am besten eignet.

3.3.1 Welche Reasoner kommen nicht in Frage / wieso nicht?

Aufgrund unserer Rechercheergebnisse und unserer Testresultate mit der Apache-Jena-Fuseki Version 5.0.0 ist es aus unserer Sicht eindeutig, dass vor allem der Transitive Reasoner für eine hierarchische Suche ungeeignet ist. Allerdings muss man beachten, dass der Reasoner die für eine hierarchische Suche auf RDF-Daten zwingend benötigte Eigenschaft „`rdfs:subClassOf`“ unterstützt und diese auch effizient ausführt. Weiterhin wird jedoch zusätzlich nur „`rdfs:subPropertyOf`“ ermöglicht. Dies sind die beiden einzigen Eigenschaften, auf die der Reasoner spezialisiert ist und welche er ausführen kann. Dadurch ist der Transitive Reasoner in seinen Fähigkeiten sehr limitiert. Dies spiegelt sich speziell in der fehlenden Unterstützung von „`rdf:type`“ wieder. Hierdurch verursacht der Reasoner deutlich aufwendigere SPARQL Abfragen. Das basiert auf der Tatsache, dass es nicht möglich ist, durch die fehlende Unterstützung von „`rdf:type`“, mit dessen Hilfe abzufragen, ob ein Element zu einer nicht direkt verwandten Klasse gehört [[Apache Software Foundation / 15](#)]. Es benötigt zuerst die Abfrage, ob das Element zu einer der Unterklassen der untersuchten Klasse gehört. Diese Problematik zeigt sich wie folgt: Bei einer Kategorie wie „Countries“ werden nicht direkt alle Länder gefunden. Stattdessen müssen zunächst alle Spezialisierungen von „Countries“ wie „European countries“, „Asian countries“, „African countries“ und so weiter lokalisiert werden. Dieser Vorgang muss solange für jede neue Kategorien wiederholt werden, bis keine Klassen mehr gefunden werden können. An diesem Punkt ist es schließlich möglich, alle Länder zu extrahieren.

Obwohl die unterstützten Eigenschaften des Reasoners für den Moment unseren Anforderungen für eine hierarchische Suche entsprechen, spricht gegen den Einsatz des Reasoners am Corpus Nummorum, dass er zukünftig eventuell schnell ausgetauscht werden müsste. Das hängt damit zusammen, dass sich in Gesprächen mit unserem Professor Herr Dr. Tolle herausgestellt hat, dass zukünftig Eigenschaften wie „`skos:exactMatch`“, „`owl:sameAs`“ und „`owl:equivalentClass`“ in die Datenbasis integriert werden könnten. Diese für zukünftige Arbeiten potenziell wünschenswerten, sowie auch etwaige andere neue, Eigenschaften werden bekanntlich weder unterstützt, noch sind sie mit dem Transitive Reasoner umsetzbar. Insofern müsste der Reasoner schnell ersetzt werden, wenn die Datenbasis um zusätzliche Eigenschaften erweitert

wird. Dies hätte einen zeitlichen Aufwand, durch unter anderem Anpassungen an den SPARQL Abfragen, zur Folge. Des Weiteren wird der direkte Einsatz des Transitive Reasoners, sowohl am Corpus Nummorum als auch allgemein, für eine derartige Suche durch andere Reasoner weitgehend obsolet. Das ist unter anderem der Tatsache geschuldet, dass es möglich ist, den Reasoner im Generic Rule Reasoner optional zu aktivieren. Außerdem ist der Transitive Reasoner standardmäßig sowohl im RDFS Rule Reasoner, als auch in der Micro Variante von OWL integriert [[Apache Software Foundation / 15](#)]. Folglich wäre es nicht sinnvoll, den Reasoner eigenständig zu nutzen, anstatt ihn in Kombination mit einem anderen Reasoner zu verwenden.

Zum Transitive Reasoner lässt sich abschließend sagen, dass der Reasoner grundlegend die nötigen Voraussetzungen für den Einsatz beim Implementieren einer hierarchischen Suche mitbringt. Seine limitierten Fähigkeiten, vor allem im Hinblick auf zukünftige Erweiterungen, sowie die Möglichkeit ihn anderweitig nutzen zu können, sorgen allerdings dafür, dass eine eigenständige Verwendung in diesem Fall nicht zu empfehlen ist.

Unseres Erachtens nach bieten sich neben dem Transitive Reasoner auch die beiden OWL Varianten Full und Mini nicht für die Implementierung einer hierarchischen Suche an. Positiv auffallend ist, dass der Full OWL Reasoner im Vergleich zu allen anderen betrachteten Reasonern die komplexeste Inferenzlogik bietet [[Apache Software Foundation / 15](#)]. Hierbei deckt er, wie auch der Mini OWL Reasoner, das für die hierarchische Suche essenzielle Konstrukt „`rdfs:subClassOf`“ ab. Zusätzlich wird „`rdf:type`“ unterstützt. Dies kann vor allem für das Finden von speziellen Elementen sehr hilfreich sein. Des Weiteren bilden die beiden Reasoner auch wesentliche Konstrukte der OWL Syntax ab. Diese Komplexität kommt jedoch mit einigen Nachteilen einher. Die Möglichkeit, RDFS-Konstrukte für Schlussfolgerungen zu verstehen und anzuwenden erweist sich nur dann als sinnvoll, wenn im vorliegenden Datensatz auch OWL-Konstrukte vorliegen. Vergleicht man die Ergebnisse der beiden Reasoner auf reinen RDF-Daten mit dem RDFS Rule Reasoner so fällt auf, dass der RDFS Rule Reasoner in deutlich kürzerer Zeit zum gleichen Ergebnis gelangt. Folglich ist der RDFS Rule Reasoner auf reinen RDF-Daten um ein Vielfaches effizienter [[Apache Software Foundation, José R. Hilera, Luis Fernández-Sanz and Adela Díez / 15, 25](#)]. Daraus schließt sich, dass der Einsatz des Full und auch des Mini OWL Reasoners auf reinen RDF-Daten im Vergleich zum RDFS Rule Reasoner keinen Vorteil mit sich

bringt. Somit ist bis dato auch am Corpus Nummorum, aufgrund des ausschließlich auf RDF basierenden Datenbestandes, der RDFS Rule Reasoner den beiden OWL Varianten vorzuziehen.

Auch im Falle dessen, dass OWL-Daten vorliegen, ist nicht automatisch der Mini oder der Full OWL Reasoner die beste Wahl. Benötigt man spezifische OWL-Konstrukte, empfehlen wir, dass man sich zunächst mit dem Micro OWL Reasoner auseinandersetzt. Sollte man nicht in Erwägung ziehen, spezifische OWL-Konstrukte zu verwenden, die nur vom Full und Mini OWL Reasoner unterstützt werden, wie zum Beispiel „owl:sameAs“ (siehe Abschnitt 3.2.4), dann ist der Micro OWL Reasoner die beste Wahl. Dies folgt aus den teils fehlenden Optimierungen der anderen beiden Varianten. Hierdurch kann in Kombination mit der geringeren Komplexität die Micro Version auf RDF- und OWL-Daten deutlich schneller schlussfolgern. Sollten allerdings Eigenschaften, wie beispielsweise „owl:sameAs“ oder „owl:equivalentClass“, benötigt werden, so hat der Micro OWL Reasoner im direkten Vergleich einen Nachteil. Betrachtet man Full und Mini OWL Reasoner untereinander, so fällt auf, dass der Mini OWL Reasoner die exakt selben OWL-Konstrukte wie der Full OWL Reasoner abbildet. Hierbei ist der Mini OWL Reasoner, wie bereits in Abschnitt 3.2.4.2 erwähnt, effizienter in seinen Schlussfolgerungen [Apache Software Foundation, José R. Hilera, Luis Fernández-Sanz and Adela Díez / 15, 25]. Der Full OWL Reasoner ist gegenüber dem Mini OWL Reasoner nur dann zu priorisieren, wenn dieser aufgrund fehlender „bNode introduction“ vereinzelt nicht terminiert [Apache Software Foundation / 15].

Ein weiterer Nachteil der OWL Reasoner besonders in Bezug auf die komplexeren Varianten ist, dass sie sehr ineffizient in der Speicherverwendung sind [Apache Software Foundation / 15]. Dies haben wir auch bei unseren eigenen Tests festgestellt (siehe Abschnitt 3.2.4). Sowohl der Mini OWL Reasoner als auch der Full OWL Reasoner haben jeweils, für den vollen Datensatz des Projekts Corpus Nummorum, einen „OutOfMemory Error“ bei unserer Hardware erzeugt. Folglich bieten sich die beiden Varianten weder an, wenn sehr große Datenmengen vorliegen, wie am Corpus Nummorum oder die Hardware im Bereich des Arbeitsspeichers ein limitierender Faktor ist. Dies kann zum Beispiel bei eigenen, privaten, Projekten der Fall sein. Auch die Syntax, auf welcher die Reasoner von OWL aufbauen, ist nicht von Vorteil. Wie bereits in Abschnitt 3.2.4 erwähnt basieren die OWL Reasoner von Apache-Jena auf der veralteten OWL 1 Syntax. Diese wurde bereits vor über 15 Jahren von OWL 2 abgelöst. Sollte also in Zukunft geplant sein, bereits vorhandenen Daten mit einer auf

OWL 2 Syntax basierenden Datenmenge zu vereinen, dann wird der Reasoner nicht in der Lage sein, auf Grundlage der OWL 2 Syntax Schlussfolgerungen zu ziehen [W3C / 9].

Zu den beiden OWL Reasonern lässt sich zusammenfassend sagen, dass der Full OWL Reasoner und der Mini OWL Reasoner für unser Ziel, eine hierarchische Suche zu implementieren, nicht sonderlich geeignet sind. Hierbei spielt es keine Rolle, ob OWL-Konstrukte im Datenbestand vorhanden sind oder nicht. Denn obwohl die beiden Reasoner ein breites Spektrum an Eigenschaften unterstützen, sind sie aufgrund ineffizienterer Arbeit für ein solches Projekt, besonders bei großen Datenmengen, ungeeignet. Explizit in Bezug auf unsere Arbeit am Corpus Nummorum lässt sich sagen, dass die Vorteile der Reasoner erst dann zum Tragen kommen würden, wenn OWL am CN implementiert wird. Allerdings sind selbst in diesem Fall ihre spezifischen OWL-Konstrukte nicht zwingend erforderlich.

3.3.2 Reasoner Wahl für eine hierarchische Suche

Unserer Auffassung nach ist es unabhängig von einer eindeutigen Ausgangs- und Datenlage nicht explizit festlegbar, welcher Reasoner von Apache-Jena am besten für eine hierarchische Suche auf numismatischen Daten geeignet ist. Wir vertreten die Ansicht, dass sowohl der Generic Rule Reasoner, sowie der RDFS Rule Reasoner als auch der Micro OWL Reasoner geeignete Reasoner für eine solche Suche sein können.

Der RDFS Rule Reasoner kommt vor allem mit seinem Vorteil, dass er auf der RDFS Syntax basiert. Somit arbeitet er besonders effizient auf reinen RDF-Daten, wodurch Merkmale und Beziehungen von Klassen schnell erkannt werden. Dem Reasoner ist es dadurch möglich, neue Erkenntnisse leicht abzuleiten. Dies macht ihn besonders empfehlenswert für eine hierarchische Suche auf reinen RDF-Daten. Durch Anwendung der RDFS-Regeln können Instanzen automatisch den richtigen Klassen zugeordnet werden. Zudem unterstützt der Reasoner die benötigte Eigenschaft „`rdfs:subClassOf`“, durch die standardmäßige Integration des Transitive Reasoners [Apache Software Foundation / 15].

Wie bereits in Abschnitt 3.2.3 erwähnt, verfügt der Reasoner über verschiedene Modi, was bei unterschiedlichen Anforderungsprofilen an eine hierarchische Suche von Vorteil sein kann. Unserer Meinung nach würde bis jetzt der „Simple“ Modus

ausreichen, da dort nur „`rdfs:subClassOf`“ und „`rdfs:subPropertyOf`“ implementiert werden. Dies reicht für eine standardmäßige hierarchische Suche völlig aus. Jedoch muss gesagt sein, dass durch unterstützte Eigenschaften anderer Modi wie „`rdf:type`“ die Implementierung der Suche deutlich vereinfacht werden würde.

Der Reasoner geht auch mit dem Nachteil einher, dass er ausschließlich RDF / RDFS-Regeln versteht und somit keine OWL-Konstrukte unterstützt. Sollte der Datenbestand OWL-Konstrukte beinhalten oder zukünftig OWL-Eigenschaften benötigt und integriert werden, so ist dieser Reasoner nicht zu empfehlen.

Der Micro OWL Reasoner von Apache-Jena bietet sich wiederum an, wenn man daran interessiert ist, spezifische OWL-Konstrukte wie „`owl:equivalentClass`“ zu verwenden. Diese stehen hierbei zusätzlich zu dem für die hierarchische Suche benötigten Konstrukt „`rdfs:subClassOf`“ zur Verfügung. Das ist der Tatsache geschuldet, dass der Micro OWL Reasoner eine Erweiterung des RDFS Rule Reasoners darstellt. Das bedeutet, dass er neben spezifischen OWL Eigenschaften alle Konstrukte des RDFS Rule Reasoners unterstützt [[Apache Software Foundation / 15](#)]. Hierbei ist jedoch zu beachten, dass der Micro OWL Reasoner nur dann sinnvoll erscheint, wenn die Verwendung von OWL-Konstrukten im Datensatz geplant ist. Der Grund dafür ist, dass der RDFS Rule Reasoner auf reinen RDF-Daten effizienter die gleichen Ergebnisse ermittelt [[José R. Hilera, Luis Fernández-Sanz and Adela Díez / 25](#)].

Wie bereits in Abschnitt [3.2.4.1](#) und [3.2.4.2](#) beschrieben bieten der Full OWL Reasoner und der Mini OWL Reasoner mehr OWL spezifische Eigenschaften [[Apache Software Foundation / 15](#)]. Dennoch bevorzugen wir den Micro OWL Reasoner aus einem einfachen Grund: Er kann um ein Vielfaches schneller schlussfolgern. Dies ist bedingt durch fehlende Optimierungen der anderen beiden Varianten sowie der geringeren Komplexität des Micro OWL Reasoners [[José R. Hilera, Luis Fernández-Sanz and Adela Díez / 25](#)].

Der Generic Rule Reasoner kommt vor allem mit dem Vorteil einher, dass für unterschiedlichste Zwecke individuelle Regeln definiert werden können. Auch er basiert auf Eingabedaten im RDF-Format und unterstützt potenziell die Eigenschaften „`rdfs:subClassOf`“ und „`rdfs:subPropertyOf`“, als „von Natur aus“ transitiv, dadurch

dass der Transitive Reasoner integriert werden kann. Im Ausführen dieser Eigenschaften ist der Generic Rule Reasoner dank der Hilfe des darauf spezialisierten Transitive Reasoners sehr effizient. Im Vergleich zum RDFS Rule Reasoner ist die Skalierung nahezu identisch [Apache Software Foundation / 15]. Auch der Generic Rule Reasoner unterstützt standardmäßig keine OWL-Syntax. Im Gegensatz zu anderen Reasonern, ist es hier allerdings möglich, durch eigens formulierte Regeln Eigenschaften, wie „rdf:type“ oder OWL-Konstrukte, wie „owl:sameAs“ und „owl:equivalentClass“, nachzudefinieren. Hierbei gilt jedoch zu beachten, dass die Regeln, wie bereits in Abschnitt 3.2.2 erwähnt, keine Rekursion und keine Selbstreferenzen beinhalten können. Dies macht das Ganze unter Umständen komplizierter, aber nicht unmöglich. Zusätzlich können Regeln endlos Schleifen verursachen, vor allem, wenn man sie ungünstig formuliert und sie sich immer gegenseitig bedingen [Apache Software Foundation / 15]. Das ist uns, wie bereits in Abschnitt 3.2.2 erwähnt, beim Testen mit den numerischen Corpus Nummorum Daten allerdings nicht passiert.

Zusammenfassend können wir sagen, dass die drei beschriebenen Reasoner alle samt ihre Vor-, aber auch ihre Nachteile im Hinblick auf die Nutzung bei der Implementierung einer hierarchischen Suche besitzen. Im Allgemeinen sollte man bei der Wahl des Reasoners die aktuelle, aber auch zukünftige Situation des Datenbestandes, sowie das Anforderungsprofil an die Navigation der hierarchischen Suche, berücksichtigen. Im Falle dessen, dass es sich sicher um eine rein auf RDF-Daten basierende Suche handeln wird ist, vor allem aufgrund seiner Effizienz auf solchen Daten, der RDFS Rule Reasoner zu empfehlen. Sollten allerdings für die Umsetzung der hierarchischen Suche spezifische Regeln benötigt werden, so hat der Reasoner im Vergleich zum Generic Rule Reasoner das Nachsehen. Auch im Falle dessen, dass OWL-Konstrukte benötigt werden, ist dieser Reasoner keine sinnvolle Wahl. In diesem Falle gilt es zwischen dem Generic Rule Reasoner und dem Micro OWL Reasoner abzuwägen. In den Überlegungen sollte vor allem eine Rolle spielen, ob die benötigte OWL Logik mit Hilfe von eigenständigen Regeln durch den Generic Rule Reasoner definiert und implementiert werden kann. Grundlegend ist es für die Implementierung einer hierarchischen Suche von Vorteil, eine Regel für „rdf:type“ definieren zu können. Diese kann wie folgt aussehen:

$$(?subClass \text{ rdfs:subClassOf } ?Class), (?Class \text{ rdfs:subClassOf } ?superClass) \rightarrow (?subClass \text{ rdfs:subClassOf } ?superClass)$$

Im Falle dessen, dass die nötigen Regeln abgebildet werden können, sollte die Wahl

auf den Generic Rule Reasoner fallen. Das hängt vor allem mit der Effizienz, sowie der durch die eigens definierbaren Regeln gegebene Flexibilität, des Reasoners zusammen. Demzufolge bietet sich der Micro OWL Reasoner unsere Meinung nach an, wenn der Datenbestand OWL spezifische Eigenschaften enthält, welche nicht mit Hilfe des Generic Rule Reasoners abgebildet werden können.

Für das Projekt Corpus Nummorum eignet sich unserer Meinung nach am ehesten die Einsetzung des Generic Rule Reasoners. Dies geht zum einen daraus hervor, dass zukünftig eventuell Elemente der OWL Syntax in den numismatischen Datenbestand integriert werden. Einerseits sind dies größtenteils Elemente, wie „owl:sameAs“, welche durch individuell formulierte Regeln definiert werden können. Zum anderen ist der Datenbestand aktuell dahingehend unvollständig, dass teilweise der Name (Label) für bestimmte Klassen fehlt. Hierfür benötigt es individuelle Regeln, um diese nachträglich dynamisch generieren zu können. Dies ist nur mit Hilfe des Generic Rule Reasoners möglich.

4 Visualisierung einer hierarchischen Suche

Wir haben bereits diskutiert, welche Reasoner verwendet werden können, um eine hierarchische Suche am Backend zu ermöglichen. In diesem Kapitel beschäftigen wir uns mit der Visualisierung einer solchen Suchfunktionalität. Hierfür stellen wir zuerst die Benutzeroberfläche dar, auf welcher wir aufbauen. Im Folgenden präsentieren wir die Änderungen, welche wir vor der Umsetzung der Visualisierung einer solchen Suche an der uns zur Verfügung gestellten Benutzeroberfläche vorgenommen haben und erklären, wieso diese aus unserer Sicht nötig waren. Anschließend beschreiben wir zunächst unsere Ideen, wie die hierarchische Navigation in die bestehende Suche integriert werden kann. Im Vorfeld haben wir uns, in Absprache mit unserem Professor Herrn Dr. Karsten Tolle, dafür entschieden, dass es am sinnvollsten ist, wenn nicht wir alleine zwischen den verschiedenen Darstellungsmöglichkeiten eine endgültige Entscheidung treffen, sondern jemand, der mit unserem Ergebnis zukünftig täglich arbeitet. Somit stellen wir im Anschluss die Ergebnisse des Gespräches mit einer Repräsentantin des Corpus Nummorum dar. Zum Abschluss des Kapitels beschreiben wir potenzielle Probleme der Implementierungswünsche der CN-Mitarbeiterin und gehen auf unsere Lösungsansätze ein. Außerdem besprechen wir, wie die Vorstellungen der CN-Mitarbeiterin samt Schwierigkeiten letztendlich zur endgültigen Form der Visualisierung geführt haben.

4.1 Ausgangssituation und notwendige Änderungen

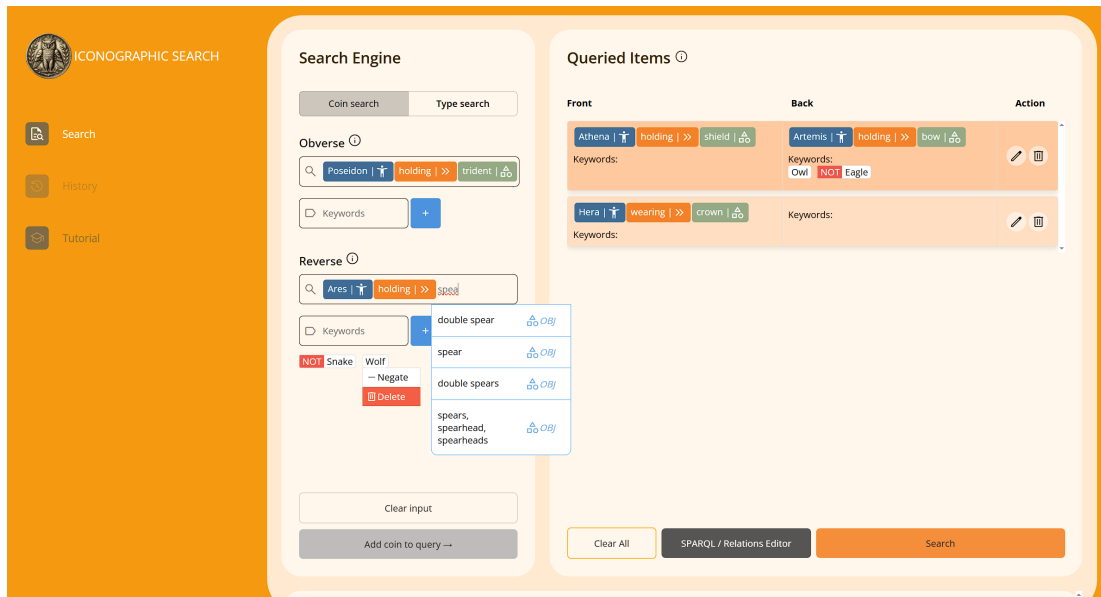


Abbildung 6: Uns zur Verfügung gestellte Benutzeroberfläche

Die obige Abbildung stellt die von Danilo Pantic und Mohammed Sayed Mahmod entwickelte Benutzeroberfläche¹¹ für die Münzsuche dar. Hierbei fällt vor allem auf, dass es pro Münzseite nur ein Eingabefeld für das RDF-Tripel gibt. Die Eingabe für Subjekt, Prädikat und Objekt erfolgt somit in einem gemeinsamen Feld. Dies ist im Hinblick auf die Visualisierung einer hierarchischen Suche ungeeignet. Aufgrund von Platzmangel erscheint es uns schwierig, ausgehend vom ausgewählten Subjekt oder Objekt die Empfehlungen für die nächste Generalisierung- / Spezialisierung-Stufe übersichtlich darzustellen. Des Weiteren haben wir es beim Testen nicht geschafft, immer, das gewünschte „Tag-Element“, also die gewollte ausgewählte Empfehlung, aus dem Eingabefeld aufseiten des Programmcodes anzusprechen. Dies wird benötigt, um im weiteren Verlauf unserer Arbeit, die korrekten Generalisierung- beziehungsweise Spezialisierung-Vorschläge anzuzeigen. Aufgrund des gemeinsamen Eingabefeldes ist es außerdem nicht möglich, lediglich nach einem Prädikat, nur nach einem Objekt oder nur nach der Kombination Prädikat + Objekt zu suchen. Stattdessen ist es ausschließlich möglich nach den Kombinationen Subjekt, Subjekt + Prädikat und Subjekt + Prädikat + Objekt, zu suchen. Als Folge der zuvor genannten Aspekte, haben wir uns dazu entschieden, die Benutzeroberfläche zu überarbeiten, bevor wir uns an die Umsetzung einer solchen

¹¹<https://github.com/Danilopa/Bachelorthesis>

Suche begeben. Unser Fokus lag dabei auf der Aufteilung der Eingabefelder für die Tripel-Elemente:

The screenshot displays the 'ICONOGRAPHIC SEARCH' web application. On the left is a vertical orange sidebar with a logo and three menu items: 'Search', 'History', and 'Tutorial'. The main content area is divided into two panels. The left panel, titled 'Search Engine', contains two sections: 'Obverse' and 'Reverse'. Each section has three input fields for 'Subject', 'Predicate', and 'Object', followed by a 'Keywords' section with a text area and a '+ -' toggle. At the bottom of the 'Search Engine' panel are 'Clear input' and 'Add coin to query ->' buttons. The right panel, titled 'Queried Items', has a table with three columns: 'Front', 'Back', and 'Action'. At the bottom of this panel are four buttons: 'Clear All', 'SPARQL / Relations Editor', 'Type search', and 'Coin search'.

Abbildung 7: Anpassungen an der Benutzeroberfläche

Wie in der obigen Abbildung zu sehen ist, besitzen Subjekt, Prädikat sowie Objekt, sowohl für die Vorderseite als auch für die Rückseite der Münze, nun ein eigenes Eingabefeld. Dabei ist zu beachten, dass wir die bisherige Suchfunktionalität über das Eingabefeld beibehalten. Aus der Aufteilung auf mehrere Eingabefelder hat sich in zweierlei Hinsicht ein Platzproblem ergeben: Zum einen haben wir in der „Search Engine“ keinen geeigneten Platz für die Auswahl zwischen „Coin Search“ und „Type Search“ gefunden. Deshalb haben wir uns dazu entschieden, das „Search“ Feld in „Coin Search“ und „Type Search“ aufzuteilen. Dadurch konnten wir etwas Platz gewinnen. Für den Benutzer ändert sich in der Anwendung nichts, da zuvor nur seine letzte Wahl zwischen „Coin Search“ und „Type Search“ für die Münzsuche relevant war. Des Weiteren ist uns aufgefallen, dass die Eingabe vieler Keywords problematisch ist, da sich dadurch der weitere Inhalt der „Search Engine“ nach unten verschieben kann. Das hat dazu geführt, dass Teile der Webseite nicht mehr nutzbar waren, da sie von anderen Elementen überdeckt wurden oder den Rahmen des Bildschirms verlassen haben. Infolgedessen haben wir für die Auflistung der Keywords ein extra Feld unter die entsprechenden Eingabefelder hinzugefügt. In diesem haben wir das Scrolling aktiviert. Nur in diesem Bereich werden nun die hinzugefügten Keywords angezeigt. Diese Änderung hat wiederum zu einem neuen Problem geführt: Die bisherige Darstellung, dass durch das Bewegen des Mauszeigers über ein Keyword, dieses negiert oder gelöscht werden kann, wurde in

dem jetzt begrenzten Keyword-Bereich zu unübersichtlich. Aus diesem Grund haben wir uns entschieden, das Hinzufügen von Keywords auf zwei Buttons aufzuteilen. Beim Hinzufügen eines Keywords kann dieses nun durch Anklicken des „+ Buttons“ normal oder durch Klicken des „- Button“ negiert hinzugefügt werden. Zusätzlich haben wir uns dafür entschieden, für die Prädikate einen Button hinzuzufügen, durch welchen der Nutzer alle zur Auswahl stehenden Prädikate angezeigt bekommt. Das ist aus unserer Sicht eine sinnvolle neue Funktion, da es derzeit nur wenige Prädikate gibt. Hierdurch kann der Benutzer leicht feststellen, welche Prädikate für die Münzsuche existieren.

4.2 Möglichkeiten und MockUps

Wie bereits in Abschnitt 1.4 erwähnt, hat das Design der Benutzeroberfläche einen großen Einfluss darauf, wie effizient die Funktionen der hierarchisch-ikonografischen Suche genutzt werden können. Im Zuge dessen haben wir mehrere verschiedene Ansätze für die visuelle Darstellung entwickelt. Diese kommen alle mit ihren eigenen Vor- und Nachteilen. Für die Empfehlungen der Tripel-Suche haben wir drei Darstellungsmöglichkeiten:

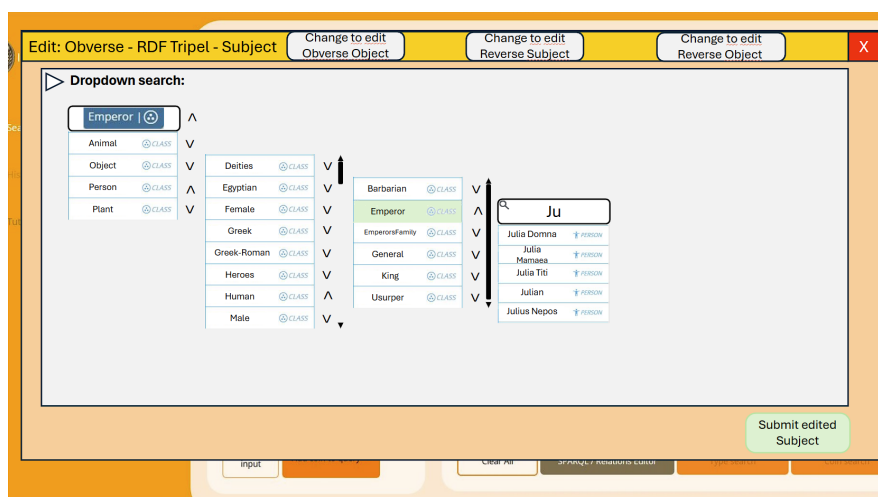


Abbildung 8: Forrest search

Wie in der obigen Abbildung zu sehen ist, stellt die erste Darstellungsmöglichkeit ein separates Fenster da, welches durch einen Button neben dem entsprechenden Subjekt oder Objekt Feld aufgerufen werden kann. In diesem wird die Auswahl / Bearbeitung des jeweiligen Subjekts / Objekts über eine „Forrest search“ ermöglicht. Hierbei hat der Nutzer zu Beginn die absoluten Oberklassen der Hierarchie („Animal“,

„Object“, „Person“ und „Plant“) zur Auswahl. Von hier aus kann sich der Nutzer durch die Ebenen der Hierarchie navigieren, indem er per Knopfdruck die direkten Spezialisierung der Ausgewählten Kategorie angezeigt bekommt. Um dabei dem Nutzer die Möglichkeit zu geben, in einer großen Auswahl an spezifischen Elementen das gewünschte Subjekt oder Objekt zu finden, wird auf der untersten Ebene die Auswahl basierend auf einer Texteingabe gefiltert. Dies führt dazu, dass nur die Spezialisierungen angezeigt werden, deren Namen mit der Texteingabe des Nutzers beginnen. Beispielsweise werden in Abbildung 8 nur diejenigen Personen der Klasse „Emperor“ angezeigt, deren Namen mit „Ju“ beginnen. Zum Start der Bearbeitung eines spezifischen Tripel-Elements einer spezifischen Münzbeschreibung befindet sich die „Forrest search“ in dem selben ausgeklappten Zustand, in dem sie zuvor verlassen wurde. Zudem ist es per Button möglich die Auswahl / Bearbeitung eines anderen Subjekts / Objekts der aktuellen Münzbeschreibung aufzurufen.

Abbildung 9: Rank search

Die zweite Umsetzungsmöglichkeit stellt, wie in Abbildung 9 zu sehen, zwei Buttons dar. Diese ermöglichen dem User die direkten Generalisierungen oder direkten Spezialisierungen des aktuell ausgewählten Subjekts / Objekts anzuzeigen und dieses durch einen der Vorschläge zu ersetzen. Ein Beispiel hierfür wäre das beim Generalisieren von „Artemis“ die Klassen „Female“, „Greek“, „Olympic“ und „Woman“ empfohlen werden, da diese die hierarchischen Eltern von „Artemis“ sind. Im Gegensatz dazu,

werden bei der Spezialisierung von „Armour“ die spezifischen Entitäten „Aegis“, „Cuirass“, „Military attire“ und „Scale cuirass“ angezeigt.

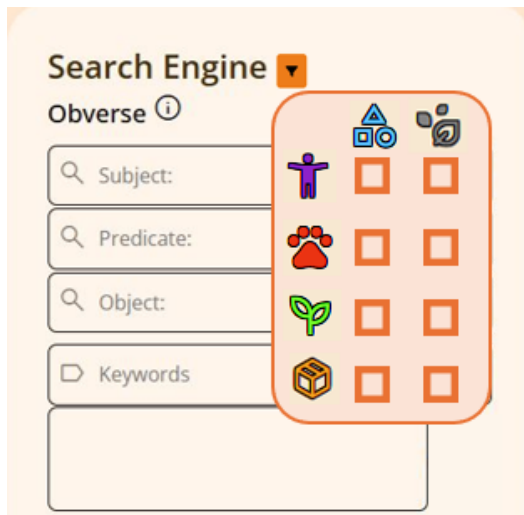


Abbildung 10: Filter search VarA

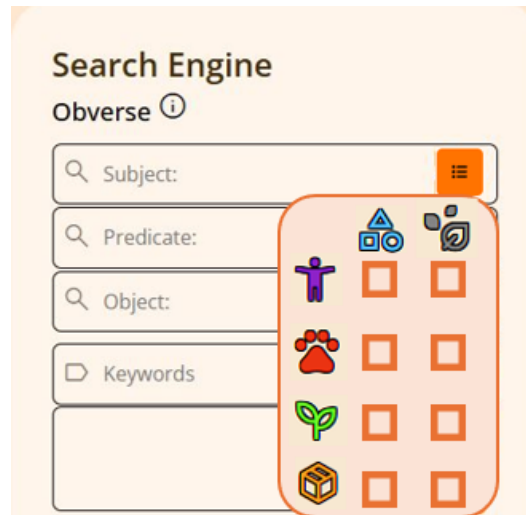


Abbildung 11: Filter search VarB

Abbildung 10 und 11 repräsentieren unsere letzte Darstellungsmöglichkeit. Es handelt sich um ein Feld mit Suchfiltern. Über dieses kann bestimmt werden, von welchen absoluten Generalisierungen, also „Person“, „Animal“, „Plant“ oder „Object“, die Spezialisierungen empfohlen werden sollen. Zusätzlich kann man entscheiden, ob die Empfehlungen Gruppen, also Kategorien, wie beispielsweise „Greek“ oder spezifische Elemente, wie zum Beispiel „Artemis“, enthalten sollen. Für die Implementierung des Suchfilters gibt es zwei verschiedene Optionen:

Option 1 (siehe Abbildung 10): Ein globaler Suchfilter, welcher auf alle Sucheingaben zutrifft.

Option 2 (siehe Abbildung 11): Ein separater Suchfilter für jedes Eingabefeld, welcher nur die Eingaben in diesem Feld filtert.

Für das Bearbeiten der Keywords haben wir ebenfalls Darstellungsmöglichkeiten entworfen, da die Benutzeroberfläche von Danilo Pantic und Mohammed Sayed Mahmod diesen Aspekt nicht eingeschlossen hat:

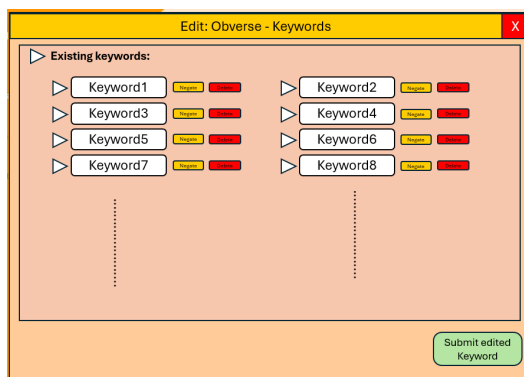


Abbildung 12: Keyword Menu

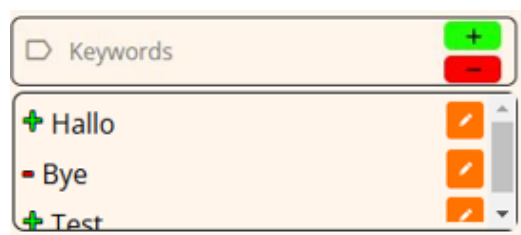


Abbildung 13: Keyword Button

Zum einen könnte man, wie in Abbildung 12 zu sehen ist, ein separates Fenster aufrufen, in welchem alle Keywords aufgelistet werden. Von dort aus können sie per Button negiert oder gelöscht werden. Diese Möglichkeit dient vor allem der Einheitlichkeit, wenn für die Tripel-Suche die „Forrest search“ (siehe Abbildung 8) implementiert wird. Abbildung 13 zeigt die zweite Möglichkeit. Jedes Keyword im Anzeigefeld erhält einen eigenen Button. Dieser transferiert das spezifische Keyword in das Eingabefeld, wodurch es dann editiert werden kann.

4.3 Wahl der Visualisierung

In diesem Abschnitt stellen wir den Entscheidungsprozess und die endgültige Umsetzung der Visualisierung für die hierarchisch-ikonografische Suche am Corpus Nummorum dar. Zunächst gehen wir auf die Ergebnisse aus dem Gespräch mit Frau Dr. Ulrike Peter vom CN-Projekt ein. Anschließend skizzieren wir die Herausforderungen bei der Umsetzung Ihrer Vorstellung und gehen auf mögliche Lösungsansätze ein. Abschließend legen wir fundiert die endgültige Darstellung der hierarchischen Suche dar.

4.3.1 Gespräch mit einer Corpus Nummorum Mitarbeiterin

In einem virtuellen Treffen haben wir Corpus Nummorum Mitarbeiterin Frau Dr. Ulrike Peter unsere, in Abschnitt 4.2 beschriebenen, Umsetzungsmöglichkeiten für eine hierarchisch-ikonografische Suche vorgestellt. Wichtig war uns vor allem: Welche Visualisierungstechnik gefällt Frau Dr. Peter am besten? Wieso? Was möchte Sie hierbei gegebenenfalls zusätzlich umgesetzt haben und warum? Zusätzlich ist es auch interessant für uns zu verstehen, warum andere Vorschläge von Ihr eventuell nicht

dieselbe Zustimmung erhalten.

Aus dem Gespräch haben wir folgende Rückmeldung von ihr erhalten:

An der Umsetzungsidee „Forrest search“ (siehe Abbildung 8) hat Ihr besonders gut gefallen, dass ein sehr großer Teil der Suche visuell dargestellt wird. Ihrer Ansicht nach erhält vor allem der unerfahrene / neue Nutzer einen schnellen und guten Überblick über die zur Auswahl stehenden Entitäten. Außerdem ist es bei Ihr sehr positiv angekommen, dass schnell zwischen einzelnen Entitäten gewechselt werden kann, welche Teil derselben Kategorie sind. Allerdings hat Sie angemerkt, dass es sehr zeitaufwendig für den Nutzer ist, immer erst ein separates Fenster öffnen zu müssen, um von der ausgewählten Entität in der Hierarchie nach oben / unten navigieren zu können. Deshalb hat Sie den Vorschlag gebracht, dass man das separate Fenster auch einfach auf einen anderen Bildschirm verschieben könnte und zur Änderung der Eingabe dann die Tripel-Elemente in der Schnellsuche anpassen kann. Somit könnte die „Forrest search“ auch als Gedankenstütze für die hierarchische Ordnung gelten, wenn man nicht jedes Mal ein separates Fenster öffnen will. Neben den bisherigen Schwierigkeiten an dieser Darstellungsmöglichkeit hat Frau Dr. Peter angemerkt, dass es unter Umständen schwierig ist, alle ähnlichen Entitäten in dieser Darstellung zu finden, da diese über mehrere Klassen verteilt sein können. Dies sei vor allem dann der Fall, wenn eine flache Hierarchie existiert, wodurch eine spezifische Person wie Telemachos eine direkte Spezialisierung von vielen Kategorien ist und somit unter all diesen nach ähnlichen Personen gesucht werden muss.

Zum Visualisierungsvorschlag „Rank search“ (siehe Abbildung 9) haben wir weitgehend positive Rückmeldung von Frau Dr. Peter erhalten. An dieser Idee hat ihr sehr gut gefallen, dass man für eine ausgewählte Entität durch nur einen Klick die nächsten Generalisierung- / Spezialisierung-Stufe vorgeschlagen bekommt. Allerdings hat Sie dahingehend Kritik geäußert, dass alle Vorschläge auf einmal zu sehen sind und es keine Möglichkeit gibt diese zu begrenzen. Ihr Wunsch ist es, dass wir im Falle einer Umsetzung ein zusätzliches Suchfeld einzubauen. Die Vorschläge sollen mit der mit der Texteingabe beginnen. Zusätzlich hat sich Frau Dr. Peter gewünscht, dass wir auch noch Buttons einbauen um auf die oberste / unterste Ebene in der Hierarchie, ausgehend von der aktuell ausgewählten Entität, navigieren zu können. Negativ aufgefallen ist Ihr bei dieser Idee, dass wenn beispielsweise eine Entität durch den Spezialisierung-Button ausgewählt wurde, der Nutzer erst per Generalisierung-Button die alte Entität auswählen muss, um dann im erneuten Spezialisierungsvorgang alle

hierarchisch nächst niedrigeren Entitäten der originalen Entität sehen zu können. Dies ist der Fall, wenn man sich zum Beispiel verklickt hat / umentscheidet etc.. Ein Beispiel hierfür ist: Als Subjekt ist „Male“ ausgewählt, dieses möchte man spezialisieren. Nach Anklicken des entsprechenden Buttons und Eingabe von „Ap“ wählt man nun versehentlich „Apollo“ aus. Um die anderen Spezialisierung-Vorschläge erneut sehen zu können, muss man den direkten Generalisierung-Button drücken „Male“ auswählen und von hier an wieder von vorne beginnen. Dem entgegen war Sie mit unserem Lösungsvorschlag für dieses Problem sehr zufrieden: Im Falle der Umsetzung bauen wir einen weiteren Button ein, welcher die Funktion hat, ausgehend von der aktuellen Entität hierarchisch gleichgestellte Entitäten anzuzeigen. Hierbei bestimmen wir als Grundlage von unserer ausgewählten Entität die direkten Überklassen und geben deren direkte Spezialisierungen wieder. Ein Beispiel hierfür wäre, dass Kategorien, wie „Roman Deities“, „Egyptian Deities“ und „Hindu Deities“, empfohlen werden, wenn der Knopf auf die Eingabe „Greek Deities“ angewendet wird.

Unsere dritte Umsetzungsidee, die „Filter search“ (siehe Abbildung 10 und 11) war in den Augen von Frau Dr. Peter die ungeeignetste der Vorschläge. Ihr hat besonders missfallen, dass bei dieser Idee keine Navigation durch die Hierarchie möglich ist. Des Weiteren war Sie der Meinung, dass die Symbole der Filter nicht sonderlich verständlich seien. Alles in allem konnte Sie dieser Idee nichts Positives abgewinnen.

Neben den Umsetzungsmöglichkeiten der hierarchisch-ikonografischen Suche haben wir Frau Dr. Peter unsere Vorschläge für eine Anpassung der Keywords unterbreitet: Ihrer Ansicht nach ist das „Keyword Menu“ (siehe Abbildung 12) zu viel für die benötigte Funktionalität. Im Gegensatz dazu, gefiel Frau Dr. Peter der Vorschlag bezüglich „Keyword buttons“ (siehe Abbildung 13). In Ihren Augen ermöglicht diese eine einfache und schnelle Erstellung und Bearbeitung der Keywords.

Frau Dr. Peter hat uns nach Sichtung aller Ideen mitgeteilt, dass sie sich für die Umsetzung der Suchfunktionalität wünscht, dass von den genannten Vorschlägen „Forrest search“ und „Rank search“ implementiert werden. Da dies zeitlich im Rahmen der neun Wochen Bearbeitungszeit der Bachelorarbeit aber wahrscheinlich schwierig umsetzbar ist, priorisiert Sie die Umsetzung der „Rank search“ (siehe Abbildung 9). Für eine Anpassung der Keywords kommt für Sie nur der Vorschlag „Keyword buttons“ (siehe Abbildung 13) infrage. Darauf aufbauend möchte Sie auf

jeden Fall, dass die Empfehlungen sich anpassen. Damit meint Sie, dass nur die Elemente empfohlen werden, welche im Kontext Sinn ergeben. Das heißt, Empfehlungen zu der Eingabe berücksichtigen auch zuvor getätigte Eingaben für andere Tripel-Elemente. Zudem hat Sie den Wunsch geäußert, dass man für Subjekt und Objekt auch Kombination von Klassen wie zum Beispiel „weiblich-griechische Götter“ verwenden kann.

4.3.2 Schwierigkeiten und Lösungsansätze bei der Umsetzung der Vorstellungen von Frau Dr. Peter

Aus dem Gespräch mit Frau Dr. Ulrike Peter sind einige Aspekte deutlich geworden, welche Sie an unseren Visualisierungsvorschlägen, im Falle einer Implementierung, angepasst haben möchte. Zusätzlich hat sie vereinzelt zusätzliche Anpassungen geäußert, die es im besten Fall zu implementieren gilt. Hierbei treten jedoch (potenziell) folgende Schwierigkeiten auf:

Neben der Bearbeitung des Tripel-Elements per „Forrest search“ (siehe Abbildung 8), wünscht sich Frau Dr. Peter, dass diese zusätzlich auch nur als eine Art Übersicht in einem separaten Browserfenster genutzt werden kann (siehe Abschnitt 4.3.1). Die Problematik hierbei liegt zum einem darin, dass wir bei unserer Planung die „Forrest search“ nur als ein statisches Div und nicht als eine neue Seite geplant hatten die in einem extra Browserfenster aufgerufen wird. Dahingehend hatten wir auch den zeitlichen Aufwand kalkuliert. Problematisch ist vor allem, dass man ein Div zwar verschiebbar machen kann, allerdings ist es basierend auf unseren Tests nicht einfach möglich, das Div auf einem anderen Bildschirm sichtbar zu machen. Es wäre zwar möglich, dass extra Fenster als eine extra Seite zu entwickeln, allerdings ist dies deutlich zeitaufwendiger als die Implementierung eines zunächst versteckten Divs. Wir müssten die benötigten Informationen, Resultate der SPARQL Abfragen und die Eingabe des Users zwischen den URLs hin und her schicken. Die Schwierigkeit liegt in unserer Augen darin, zusätzlich dafür Sorge zu tragen, dass die korrekten Eingaben beim erneuten Öffnen wieder erscheinen. Außerdem hätten wir das große Problem, dass das Browserfenster während der gesamten Bearbeitungszeit geöffnet bleiben könnte. Hierbei müsste sich der Inhalt des Fensters dynamisch, ohne es zu schließen, aktualisieren: Angenommen der Benutzer gibt zunächst per Schnellsuche das Tripel „Artemis holding Bow“ für die Vorderseite ein und öffnet dann für Artemis die „Forrest search“. Danach tätigt er die Eingabe „Emperors wearing Crown“ für die Vorderseite

einer neuen Münzbeschreibung mithilfe der Schnellsuche. Nun will der Nutzer den Rahmen der Suche von „Emperors wearing Crown“ zu einer Spezialisierung wie „Julius Cäsar wearing Crown“ eingrenzen. Die nicht dynamische Seite zeigt immer noch „Artemis“ als Subjekt, obwohl der Benutzer eigentlich „Emperors“ anpassen will. Dadurch muss der Nutzer im Baum erst einmal nach „Emperors“ suchen, um von dort alle direkten Spezialisierungen sehen zu können. Es gilt zu beachten, dass eine Aktualisierung des Browserfensters durch code-seitiges schließen und öffnen das Problem theoretisch lösen würde. Jedoch hätten wir dann das Problem, dass das Fenster immer wieder geöffnet wird. Das mehrmalige Öffnen der „Forrest search“ störte Frau Dr. Peter allerdings schon an anderer Stelle (siehe Abschnitt 4.3.1). All dies erscheint uns problematisch, da die Umsetzung aufgrund mangelnder Erfahrung unsererseits wahrscheinlich den zeitlichen Rahmen dieser Arbeit sprengen würde.

Als Kompromisslösung für die Umsetzung der „Forrest search“ könnten wir uns vorstellen, diese in Kombination mit einer der anderen beiden Ideen als eine rein statische Gedankenstütze zu implementieren. Somit würde die „Forrest search“ nur noch als eine Visuelle Darstellung der RDF-Datenbank dienen. Diese kann dauerhaft in einem separaten Browserfenster geöffnet bleiben. Dadurch, dass die Möglichkeit wegfällt, hierüber Tripel-Elemente zu bearbeiten, ersparen wir uns die Arbeit, das Browserfenster dynamisch zu aktualisieren.

Eine Umsetzung der ursprünglichen „Filter search“ (siehe Abbildung 10 und 11) ist im Zusammenhang mit den Kritikpunkten von Frau Dr. Peter nicht möglich. Zwar ist es möglich, zwischen Klassen und Elemente zu unterscheiden, aber das Problem ist, dass es aus unserer Sicht keine Möglichkeit gibt, durch reine Anpassung der Filter, einen Wechsel innerhalb Klassenebene zu ermöglichen. Einzig und alleine die unpassende Symbolik der Filter könnte man durch kurze Texte ersetzen und somit dieses Problem lösen. Um ein hierarchisches Wechseln auf Klassenebene in Kombination von Filtern zu ermöglichen, muss diese Idee mit einer anderen kombiniert werden.

Wie bereits in Abschnitt 4.3.1 erwähnt soll bei der Umsetzung der „Rank search“ ein Button implementiert werden, welcher Entitäten auf der gleichen hierarchischen Ebene lokalisiert. Dabei folgt jedoch aus der Methode zur Bestimmung dieser ähnlichen Entitäten, dass nicht immer alle gültigen Subjekte / Objekte gefunden werden, welche auf der gleichen Hierarchiestufe existieren. Ein Beispiel hierfür sind die spezifischen Personen „Artemis“ und „Julius Caesar“. Diese stellen als direkte Spezialisierungen

der Kategorien „Greek“ und „Roman“ gleich starke Spezialisierungen der Obersten Kategorie „Person“ dar, besitzen aber keine direkten gemeinsamen Generalisierungen. Deswegen werden diese nicht als ähnlich eingestuft. Als Kompromiss würden wir somit nur hierarchische Geschwister des „Tag-Elements“ anzeigen.

Des Weiteren hat es Frau Dr. Ulrike Peter gestört, dass keine Schnellsuche nach kombinierten Kategorien, wie „weiblich-griechische Götter“, ermöglicht wird. Stattdessen muss der Nutzer derzeit mehrere Suchanfragen kombinieren, um das gewünschte Ergebnis zu erzielen. Um dies mit einer einzigen Suchanfrage zu ermöglichen, ohne kombinierte Klassen in die Datenbank aufzunehmen, müsste sowohl die gesamte Suchfunktion als auch die Eingabe überarbeitet werden. Das geht damit einher, dass bis zu diesem Zeitpunkt die gesamte Suche darauf ausgelegt wurde, pro Anfrage für jede Seite der Münze genau ein Subjekt, ein Prädikat und ein Objekt zu erhalten. Im Rahmen unserer Arbeit ist das aus unserer Sicht kaum umzusetzen.

Abschließend ist zu erwähnen, dass es problematisch werden könnte, wie von Frau Dr. Peter gewünscht, die Empfehlungen dahingehend anzupassen, dass sie im Kontext vollständig Sinn ergeben. Zwar scheint es uns möglich, die Empfehlungen für ein Tripel-Element einer Münzseite, aufbauend auf den bereits bestehenden Eingaben der jeweiligen Münzseite, anzupassen, über beide Münzseiten hinweg scheint es uns jedoch komplizierter. Im Falle dessen, dass man eine Eingabe für das Objekt der Vorderseite tätigt, müsste man beim seitenabhängigen Anpassen nur überprüfen, ob Münzen existieren, auf deren Vorderseite mit der Eingabe beginnende Objekte in Kombination mit den bereits vorhandenen Eingaben für Subjekt und Prädikat bestehen. Hierbei ist allerdings zu beachten, dass man somit durch reine Nutzung der Tripel-Eingabe einzelne Münzbeschreibungen erstellen kann, die keine Ergebnisse erzeugen. Das liegt daran, dass beispielsweise für Eingaben der Rückseite nicht überprüft wird, ob es zur Eingabe entsprechende Entitäten gibt, die zum einen in einem Tripel mit den anderen Elementen der Rückseite auf einer Münze existieren und zum anderen, dass auf derselben Münze das Münzbild der Vorderseite den entsprechenden Eingaben entspricht. Um dies zu ermöglichen, müsste man allerdings die ohnehin groß werdenden SPARQL Anfragen durch das Hinzufügen zusätzlicher Variablen weiter vergrößern. Dies steigert das Fehlerpotenzial und ist mit weiterem zeitlichem Aufwand verbunden.

4.3.3 Endgültige Umsetzung der Visualisierung

Unter Berücksichtigung des Gespräches mit Frau Dr. Peter und den Schwierigkeiten die wir bei der Implementierung ihrer Vorstellungen gesehen haben, haben wir die Benutzeroberfläche wie folgt umgesetzt:

The image shows a web application interface for searching numismatic data. It is divided into two main sections: 'Search Engine' and 'Queried Items'.

Search Engine:

- Obverse:** Contains four search fields: 'Subject', 'Predicate', 'Object', and 'Keywords'. Each field has a magnifying glass icon and a set of three orange squares with icons (a list, a grid, and a single item). The 'Keywords' field also has a trash icon.
- Reverse:** Contains the same four search fields as the Obverse section.
- Buttons:** At the bottom of the Search Engine section are two buttons: 'Clear input' and 'Add coin to query --'.

Queried Items:

- At the top, it says 'Queried Items' with a help icon.
- Below this, there are three columns: 'Front', 'Back', and 'Action'.
- At the bottom of this section are four buttons: 'Clear All', 'SPARQL / Relations Editor', 'Type search', and 'Coin search'.

Abbildung 14: Endgültige Visualisierung einer hierarchisch-ikonografischen Suche auf numismatischen Daten

Im Vergleich zu Abbildung 7 haben wir die Webseite wie folgt erweitert:

Zum einen haben wir uns damit beschäftigt die Empfehlungen anzupassen. Dies bedeutet, dass die Empfehlungen nun nicht mehr ausschließlich auf der aktuellen Eingabe des Nutzers basieren, sondern auch die bisherigen Eingaben für die anderen Tripel-Elemente der entsprechenden Seite berücksichtigen. Die empfohlenen Entitäten existieren also in einer Tripel-Beziehung, mit den anderen Eingaben auf der jeweiligen Seite von mindestens einer Münze. Die Berücksichtigung der Eingaben der anderen Münzseite ist uns dabei nicht gelungen, da wir fehlerhafte SPARQL Anfragen erzeugt haben. Die Empfehlungen waren entweder leer oder haben teilweise Entitäten beinhaltet, die in dieser Kombination keine Münzergebnisse zuließen. Den Fehler konnten wir aus Zeitgründen nicht beheben.

Unserer Hauptaugenmerk der Umsetzung lag auf der Implementierung der „Rank search“, da diese von Frau Dr. Peter favorisiert wurde (siehe Abschnitt 4.3.1). Als Ergebnis des Gespräches, beinhaltet diese mehr als die beiden ursprünglich geplanten Funktionen (siehe Abbildung 9):

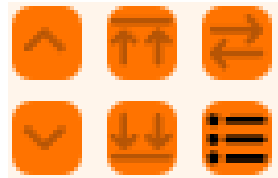


Abbildung 15: Button-Auswahl für die Bearbeitung von Subjekt und Objekt

Wie in der obigen Abbildung zu sehen, haben wir hierfür verschiedene Buttons implementiert, welche im folgenden von links nach rechts, Zeile für Zeile erklärt werden:

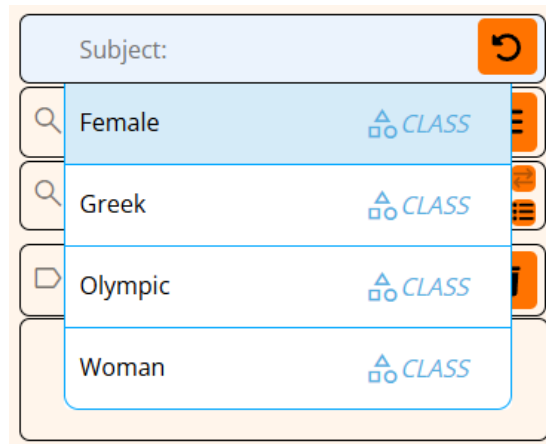
1. Ausgehend von der ausgewählten Entität werden dem Benutzer alle direkt übergeordneten Kategorien empfohlen.
2. Mit Hilfe dieses Buttons bekommt der Benutzer die oberste Klasse empfohlen, zu der die ausgewählte Entität gehört.
3. Durch das Anwenden dieses Buttons bekommt der Nutzer alle hierarchisch-gesehenen Geschwisterentitäten empfohlen. Diese sind jeweils eine Unterklasse / ein Element einer direkten Überklasse der ausgewählten Entität.
4. Zur ausgewählten Entität erhält der Benutzer alle direkt untergeordneten Entitäten. Hierbei kann es sich entweder um Unterklassen oder spezifische Elemente handeln.
5. Im Falle dessen, dass die ausgewählte Entität eine Klasse ist, werden alle Elemente dieser Klasse aufgelistet.
6. Dieser Button listet alle möglichen Subjekte / Objekte auf.

Hierbei gilt es folgende Aspekte zu erwähnen:

1. Sollte es keine Empfehlungen ausgehend von der ausgewählten Entität geben, so sind die entsprechenden Buttons nicht auswählbar.

2. Der sechste Button beinhaltet keine Funktionalität für die hierarchische Suche. Diese Funktionalität haben wir in Folge eines Gespräches mit unserem Professor Herrn Dr. Tolle eingebaut. Er hatte angemerkt, dass es sinnvoll wäre zu sehen, welche Subjekte / Objekte zur Auswahl stehen.

Um die Suche nach der passenden Entität zu erleichtern habe wir für die hierarchischen Empfehlungen ein zusätzliches Suchfeld eingebaut:



The image shows a user interface for a search function. At the top is a light blue input field labeled 'Subject:' with an orange circular arrow icon on the right. Below this is a list of suggestions, each in a white box with a light blue border. The suggestions are 'Female', 'Greek', 'Olympic', and 'Woman'. To the left of each suggestion is a small icon: a magnifying glass for 'Female' and 'Greek', and a folder icon for 'Olympic' and 'Woman'. To the right of each suggestion is a small icon of a triangle with a circle inside, followed by the word 'CLASS'. On the far right of the list, there are three orange buttons with icons: a magnifying glass, a folder, and a trash can.

Abbildung 16: Nach Anklicken eines Buttons für die hierarchische Suche erscheinen die entsprechenden Empfehlungen und ein neues Suchfeld

Hierbei erfolgt der Abgleich wie bei der Schnellsuche von Danilo Pantic und Mohamed Sayed Mahmod. Die Empfehlungen für die Entitäten der hierarchischen Suche müssen mit der eingegebenen Zeichenfolge beginnen. Groß- und Kleinschreibung wird auch hier ignoriert. Sollte der Nutzer keine geeignete Entität finden, kann er mit Hilfe des „Zurück Button“ neben dem Suchfeld, seine ursprünglich ausgewählte Entität behalten.

Zum Bearbeiten der Keywords haben wir uns dafür entschieden einen extra Button einzubauen. Dies beruht, wie bereits erwähnt, auf der Tatsache, dass dies bis jetzt gar nicht möglich war und wir das Bearbeiten der Keywords ermöglichen wollten. Um ein Keyword zu löschen, muss man erst das Keyword bearbeiten, um es aus der Liste zu entfernen und dann kann man das Eingabefeld leeren. Dies ist mithilfe des Buttons mit dem Mülleimer Symbol neben dem Eingabefeld möglich.

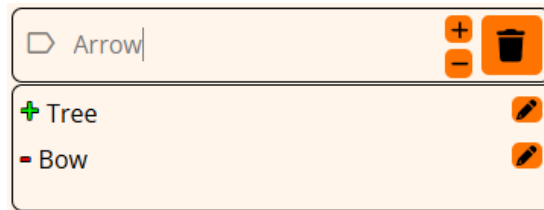


Abbildung 17: Der Button zum Bearbeiten neben dem Keyword sorgt dafür, dass es zurück ins Eingabefeld transformiert wird.

Wie bereits in Abschnitt 4.3.2 erwähnt haben wir in Betracht gezogen die „Forrest search“ in einer statischen Variante zu implementieren. Diese Idee haben wir jedoch in der endgültigen Version nicht umgesetzt. Der Grund dafür liegt darin, dass wir uns zunächst auf die Verbesserungen fokussiert haben die Frau Dr. Peter am wichtigsten waren. Nach der Implementation dieser war nicht mehr genug Zeit übrig, um selbst die reduzierte Form der „Forrest search“ zu implementieren. Dieser Mangel an Zeit ist daraus entsprungen, dass während der Umsetzung einige unerwartete Problem aufgetreten sind. Ein paar Beispiele für diese sind, dass die SPARQL Abfragen durch falsche Anordnung innerhalb der Abfragen um ein Vielfaches länger gedauert haben. Zudem haben wir uns zu sehr damit aufgehalten, die Empfehlungen unter Berücksichtigung der Eingaben beider Münzseiten zu generieren.

Im Gespräch mit Frau Dr. Peter (siehe Abschnitt 4.3.1) wurde, wie bereits erwähnt, deutlich, dass Sie aufgrund der limitierten Funktionalität keine große Begeisterung für das Suchfilterfeld besitzt. Aufgrund dessen haben wir uns dazu entschieden, dies in der endgültigen Version nicht zu implementieren. Stattdessen haben wir es dabei belassen, die bestehende Kategorisierung der Empfehlungen dahingehend zu erweitern, dass Gruppierungen einzelner Entitäten nun als „Class“ gekennzeichnet werden. Des Weiteren haben wir die separate Seite zum Editieren der Keywords aufgrund der Erkenntnisse aus dem Gespräch mit Frau Dr. Peter nicht implementiert. Ein weiterer Punkt, der während des Gesprächs aufkam, aber von uns letztendlich nicht umgesetzt wurde, ist die Suche nach kombinierten Generalisierungen, wie „Female Deities“. Der Grund dafür ist, dass, wie in Abschnitt 4.3.2 bereits erwähnt wurde, der zeitliche Rahmen der Arbeit dies nicht zulässt.

5 Fazit

Im letzten Kapitel unserer Abschlussarbeit befassen wir uns mit der Beantwortung unserer Forschungsfrage. Abschließend geben wir einen Ausblick darauf, wie das Ergebnis unserer Arbeit zukünftig verbessert und erweitert werden kann. Hierbei gehen wir zudem näher darauf ein, welche Aspekte der Implementierung wir im Zuge dieser Arbeit nicht realisieren konnten.

5.1 Beantwortung der Forschungsfrage

Mit dem Abschluss unserer Arbeit können wir sagen, dass sich der Generic Rule Reasoner als eine sehr gute Wahl für die Implementierung einer hierarchisch-ikonografischen Suche auf den numismatischen Daten des Corpus Nummorum erwiesen hat. Hierbei hat uns vor allem geholfen, dass es uns möglich war eigene Regeln zu formulieren, nach welchen der Reasoner logische Schlussfolgerungen zieht. Dies war besonders dafür nötig und hilfreich, um vereinzelt fehlende Informationen von Entitäten, wie das Label / den Namen, nachträglich zu generieren oder in der Datenbank existierende Schleifen, wie etwa eine „`rdfs:subClassOf`“ Beziehung einer Klasse zu sich selbst zu entfernen. Generell ist es durch die Verwendung dieses Reasoners möglich, durch Formulierung einer Regel, kleine Fehler in der Datenbank zu korrigieren. Auch unabhängig vom Corpus Nummorum stellt der Reasoner unserer Meinung nach eine gute Wahl für die Implementierung einer hierarchischen Suche dar. Aufgrund unserer Forschungsergebnisse muss jedoch gesagt sein, dass die perfekte Wahl eines Reasoners für die Umsetzung einer hierarchischen Suche maßgeblich vom Datenbestand und zukünftig geplanten Änderungen abhängt. Im Falle dessen, dass ausschließlich auf RDF-Daten gearbeitet wird, ist der RDFS Rule Reasoner zu empfehlen, denn dieser ist auf das Erkennen und Ableiten von Beziehungen auf RDF-Daten spezialisiert. Sollte der Datenbestand den Umgang mit OWL implizieren, so ist der RDFS Rule Reasoner keine geeignete Option. Neben dem Generic Rule Reasoner, mit jenem man „`owl:sameAs`“ und „`owl:equivalentClass`“ nach implementieren kann bietet sich für diesen Fall der Micro OWL Reasoner an. Mit jenem kann man die wesentlichen Teile der OWL Syntax implementieren. Die Auswahl eines geeigneten Reasoners hängt also maßgeblich mit der individuellen Ausgangslage zusammen.

Zur Visualisierung einer hierarchischen Suche bietet sich unserer Erfahrung nach am

ehesten die „Rank search“ (siehe Abbildung 9) beziehungsweise eine Form dieser an. Dies geht damit einher, dass wir uns viel Zeit dafür genommen haben unterschiedliche Ideen für die Visualisierung (siehe Abschnitt 4.2) zu entwickeln und hiervon die „Rank search“ den größten Zuspruch gefunden hat (siehe Abschnitt 4.3.1). Des Weiteren lässt sich zu dieser Art der Umsetzung sagen, dass sie nicht besonders schwer zu implementieren ist. Hinter jedem Button befindet sich schlussendlich nur eine SPARQL Abfrage, um die passenden Vorschläge für die Entitäten der gewünschten Hierarchieebene zu erhalten. Außerdem gewährt diese Art der Visualisierung einen guten und schnellen Überblick und benötigt wenig Platz.

5.2 Ausblick / Zukünftige Arbeit

Wie bereits mehrmals erwähnt, war für uns der zeitliche Rahmen eine der größten Hürden dieser Arbeit. Deshalb existieren einige Aspekte, welche noch in zukünftigen Arbeiten vertieft werden können. Eine Möglichkeit zum Ausbau der Resultate unserer Arbeit ist es, den Rahmen für zu untersuchende Reasoner auszuweiten. Dies kann zum Beispiel dadurch erfolgen, dass nicht nur ein Blick auf die von Apache-Jena genannten Standard Reasoner geworfen wird, sondern auch andere bekannte Reasoner wie HermiT und Pellet analysiert werden.

Auch für die Visualisierung gibt es einige Punkte, die noch verbessert werden können. Einer dieser Aspekte ist es, wie auch schon im Gespräch mit Frau Dr. Peter erwähnt wurde (siehe Abschnitt 4.3.1), die Suchfunktion und Eingabe insofern anzupassen, dass auch eine Schnellsuche für Kombination von Klassen möglich wird. Eine weitere Funktionalität, welche wir aus zeitlichen Gründen nicht im Rahmen dieser Arbeit umsetzen konnten, ist es dem Nutzer zu ermöglichen, sich einen visuellen Überblick über die Datenbank zu verschaffen. Ein mögliches Beispiel für eine Visualisierung der Datenbank ist unser MockUp „Forrest Search“ (siehe Abbildung 8). Des Weiteren ist es uns, wie in Abschnitt 4.3.3 erwähnt, aus zeitlichen Gründen, nicht gelungen, den Fehler beim Filtern von Empfehlungen zu finden, wenn beide Seiten betrachtet werden. Die Lösung dieses Problems würde eine weitere Verbesserung für zukünftige Entwicklungen darstellen.

6 Quellen

Literatur

- [1] Corpus Nummorum. Corpus Nummorum Online, 2014. URL <https://www.corpus-nummorum.eu/>. URL Date: 03-03-2025.
- [2] Wikipedia. Ontologie (Informatik), 2002. URL [https://de.wikipedia.org/wiki/Ontologie_\(Informatik\)](https://de.wikipedia.org/wiki/Ontologie_(Informatik)). URL Date: 03-03-2025.
- [3] Dipl.-Ing. (FH) Stefan Luber. Was ist eine Reasoning Engine?, 2024. URL <https://www.bigdata-insider.de/was-ist-eine-reasoning-engine-a-6382aa0cdf3e0d3ce1fe978ed4fe8fa2/>. URL Date: 03-03-2025.
- [4] Anne Augustin, Marco Kranz, Ralph Schäfermeier. Semantic Web: Reasoners und Frameworks. URL http://www.ag-nbi.de/lehre/07/S_MWT/Material/05_ReasonersFrameworks.pdf. URL Date: 03-03-2025.
- [5] Elke von Lienen. Entwicklung eines RDF-Web-Services mit Trigger-Funktionalität, 2006. URL <https://www.dbis.informatik.uni-goettingen.de/Teaching/Theses/PDF/Diplom-Lienen-CLZ-032006.pdf>. URL Date: 03-03-2025.
- [6] W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax, 2004. URL <https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-simple-data-model>. URL Date: 03-03-2025.
- [7] Eckhardt Schön. Das Resource Description Framework (RDF) - ein neuer Weg zur Verwaltung von Metadaten im Netz, 1998. URL <https://xn--eckhardt-schn-tmb.de/res/Beruf/rdfeinfuehrung.pdf>. URL Date: 03-03-2025.
- [8] W3C. OWL Web Ontology Language Overview, 2004. URL <https://www.w3.org/TR/2004/REC-owl-features-20040210/>. URL Date: 03-03-2025.
- [9] W3C. OWL 2 Web Ontology Language Document Overview (Second Edition), 2012. URL <https://www.w3.org/TR/owl2-overview/>. URL Date: 03-03-2025.

- [10] W3C. SPARQL 1.1 Overview, 2013. URL <https://www.w3.org/TR/sparql11-overview/#sec-intro>. URL Date: 03-03-2025.
- [11] CORDIS. SPARQL Leitfaden für vernetzte offene CORDIS-Daten. URL <https://cordis.europa.eu/about/sparql/de#:~:text=SPARQL%2DAbfragen%20basieren%20auf%20dem,Objekt%20eine%20Variable%20sein%20kann>. URL Date: 03-03-2025.
- [12] Sven Naumann. PROLOG Eine Einführung, 2007. URL <https://www.uni-trier.de/fileadmin/fb2/LDV/Naumann/prolog.pdf>. URL Date: 08-03-2025.
- [13] Wikipedia. Datalog, 2005. URL <https://de.wikipedia.org/wiki/Datalog>. URL Date: 08-03-2025.
- [14] Prof. Dr. Hellberg. Reasoning, 2025. URL <https://beratung-ki.de/glossar-eintrag/reasoning/>. URL Date: 03-03-2025.
- [15] Apache Software Foundation. Reasoners and rule engines: Jena inference support, 2011. URL <https://jena.apache.org/documentation/inference/>. URL Date: 03-03-2025.
- [16] Class GenericRuleReasoner. URL <https://jena.apache.org/documentation/javadoc/jena/org.apache.jena.core/org/apache/jena/reasoner/rulesys/GenericRuleReasoner.html>. URL Date: 03-03-2025.
- [17] Reasoners and rule engines: Jena inference support, 2011. URL <https://jena.apache.org/documentation/inference/>. URL Date: 03-03-2025.
- [18] Prof. Dr. Richard Lackes, Dr. Markus Siepermann. Vorwärtsverkettung, 2018. URL <https://wirtschaftslexikon.gabler.de/definition/vorwaertsverkettung-50537/version-273756>. URL Date: 03-03-2025.
- [19] Wikipedia. Vorwärtsverkettung, 2004. URL <https://de.wikipedia.org/wiki/Vorw%C3%A4rtsverkettung>. URL Date: 03-03-2025.
- [20] Prof. Dr. Richard Lackes, Dr. Markus Siepermann. Rückwärtsverkettung, 2018. URL <https://wirtschaftslexikon.gabler.de/definition/rueckwaertsverkettung-42933/version-266273>. URL Date: 03-03-2025.

- [21] Wikipedia. Rückwärtsverkettung, 2004. URL <https://de.wikipedia.org/wiki/R%C3%BCckw%C3%A4rtsverkettung>. URL Date: 03-03-2025.
- [22] RDFS1. URL <https://jena.staged.apache.org/documentation/javadoc/jena/org.apache.jena.core/org/apache/jena/reasoner/rulesys/RDFSRuleReasoner.html>. URL Date: 03-03-2025.
- [23] W3C. RDF Schema 1.1, 2014. URL <https://www.w3.org/TR/rdf-schema/>. URL Date: 03-03-2025.
- [24] Pierre-Antoine. rdfs:Container, 2001. URL <https://perso.liris.cnrs.fr/pierre-antoine.champin/2001/rdf-tutorial/node18.html>. URL Date: 03-03-2025.
- [25] José R. Hilera, Luis Fernández-Sanz and Adela Díez. COMPARATION OF OWL ONTOLOGIES REASONERS Testing Cases with Pellet and Jena , 2011. URL <https://www.scitepress.org/Papers/2011/34714/34714.pdf>. URL Date: 03-03-2025.

7 Anhang

7.1 RDF Präfixe

Präfix	URI
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#