Universität Frankfurt am Main
Fachbereich Biologie und Informatik (15)
Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme

Diplomarbeit

vorgelegt von: Fabian Wleklinski
E-Mail: fabian@wleklinski.de

Betreuer: Herr Karsten Tolle

Bearbeitungszeitraum: 2. Mai bis 3. November 2003

Erstprüfer: Herr Prof. Dott.-Ing. R. Zicari

Suche im Semantic Web

Erweiterung des VRP um eine intuitive und RQL-basierte Anfrageschnittstelle

Kurzfassung / Abstract

Datenflut im World Wide Web – ein Problem jedes Internetbenutzers. Klassische Internetsuchmaschinen sind überfordert und liefern immer seltener brauchbare Resultate. Das Semantic Web verspricht Hoffnung – maßgeblich basierend auf RDF. Das Licht der Öffentlichkeit erblickt das Semantic Web vermutlich zunächst in spezialisierten Informationsportalen, so genannten Infomediaries. Besucher von Informationsportalen benötigen eine Abfragesprache, welche ebenso einfach wie eine gewöhnliche Internetsuchmaschine anzuwenden ist. Eine derartige Abfragesprache existiert für RDF zur Zeit nicht. Diese Arbeit stellt eine neuartige Abfragesprache vor, welche dieser Anforderung genügt: eRQL. Bestandteil dieser Arbeit ist der mittels Java implementierte eRQL-Prozessor eRqlEngine, welcher unter http://www.wleklinski.de/rdf/ und unter http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/ bezogen werden kann.

Schlagwörter: Semantisches Web, RDF, RQL, eRQL, Informationsportal

Chaos inside the World Wide Web – a problem of each internet user. Classical internet search engines cannot handle the flood of web pages anymore, and often deliver poor results. The Semantic Web raises hope – significantly based on RDF. The Semantic Web will probably gain popularity inside specialized information portals at first, so called infomediaries. Visitors of information portals need a query language, which can be used as easily as a common internet search-engine. But no such query language does exist for RDF. This thesis presents a novel query language which satisfies this requirement: eRQL. Part of this thesis is eRqlEngine – an eRQL processor for Java which can be obtained at http://www.wleklinski.de/rdf/ and http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/.

Keywords: Semantic Web, RDF, RQL, eRQL, Infomediary

Ehrenwörtliche Erklärung zur Diplomarbeit

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Frankfurt am Main, 3. November 2003

Fabian Wleklinski

Inhaltsübersicht

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
POI	Abkürzung für Point Of Interest, eine Wortschöpfung im Rahmen dieser Arbeit. Bezeichnet die lokale Umgebung einer RDF-Aussage, in welche all jene Aussagen einbezogen werden, die mit dieser Aussage eine Ressource oder ein Literal im weitesten Sinne gemeinsam haben. Siehe 6.1.2 Umgebung und Abfragemodus.
RDBMS	Abkürzung für Relationales Datenbankmanagementsystem. Bezeichnet alle gewöhnlichen, relationalen Datenbanksysteme wie z. B. Oracle, MySQL, PostgreSQL, IBM DB/2, MS SQL Server, und viele mehr.
RDF	Abkürzung für Resource Description Format [RDF-HOME]. Eine Empfehlung des W3C zur einheitlichen und plattformunabhängigen Kodierung von Informationen, welche dem Zweck der Beschreibung von Ressourcen dienen (z. B. der Beschreibung von Webseiten, Personen, Büchern, Filmen, …). Siehe auch 2.2 Resource Description Framework (RDF).
RDF/S	Kurzschreibweise für RDF und RDFS.
RDFS	Abkürzung für RDF Schema [RDFS]. Eine Empfehlung des W3C zur Definition so genannter RDF Vokabularien, worunter die Definition gültiger RDF-Konstrukte innerhalb einer Domäne verstanden wird (z. B. innerhalb des Produktkataloges eines Versandhauses, innerhalb eines Literaturverzeichnisses, …). Siehe auch 2.3 RDF Schema (RDFS).
RQL	Abkürzung für RDF Query Language. Bezeichnet eine SQL-ähnliche Abfragesprache für RDF, welche Bestandteil der ICS-FORTH RDFSuite [ICS-FORTH] ist. Siehe auch 5 RQL – eine RDF Abfragesprache.
SQL	Abkürzung für Structured Query Language. Weit verbreitete Abfragesprache für relationale Datenbankmanagementsysteme (RDBMS). SQL als Quasi-Standard für Abfragesprachen ermöglicht eine weitgehende Unabhängigkeit von Applikationen zu einem konkreten Datenbankserver. SQL hat durch seine Syntax den Begriff der SELECT-FROM-WHERE-Abfragesprache geprägt.
W3C	Konsortium aus derzeit knapp 400 Firmen, welches verschiedenste Empfehlungen (oftmals fälschlicherweise auch Standards genannt) in Zusammenhang mit dem Internet erarbeitet und öffentlich publiziert [W3C]. Dank seiner illustren Mitgliederliste [W3C-MEMB] genießen Empfehlungen des W3C eine große, kommerzielle Akzeptanz und dadurch eine relativ große Aufmerksamkeit in der Entwicklergemeinde.
WWW	Abkürzung für World Wide Web, umgangssprachlich oft als *Web* bezeichnet.
XML	Abkürzung für Extensible Markup Language. Bezeichnet eine Empfehlung des W3C zur einheitlichen und plattformunabhängigen Kodierung beliebiger strukturierter wie semistrukturierter Informationen.

Vorwort

Die vorliegende Arbeit ist an der Professur *Datenbanken und Informationssysteme* des Fachbereiches *Biologie und Informatik*[1] der *Johann Wolfgang Goethe-Universität Frankfurt am Main*[2] als Diplomarbeit von Fabian Wleklinski entstanden.

Ein Ansporn zur Wahl dieses Themas war die Allgegenwärtigkeit der im weitesten Sinne bearbeiteten Problematik: der Überflutung von Datennetzen – insbesondere des World Wide Web – durch maschinenunlesbare Informationen.

Diese Arbeit sollte ohne Kenntnisse des Semantic Web gelesen und verstanden werden können. Spezielle Begrifflichkeiten des Semantic Web werden in 2 Grundlagen vorgestellt und erläutert. Es wird jedoch vorausgesetzt, dass der Leser über Grundkenntnisse in Zusammenhang mit dem World Wide Web verfügt.

Über Verbesserungsvorschläge, konstruktive Kritik oder andersartige Rückmeldungen per E-Mail an fabian@wleklinski.de freue ich mich sehr. Diese Arbeit sowie alle zugehörigen Quellcodes, Kompilate und weitere Dateien sind in verschiedenen Dateiformaten erhältlich, und können unter http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/ sowie unter http://www.wleklinski.de/rdf/ bezogen werden.

Mein Dank gebührt meinem Betreuer Karsten Tolle[3], sowie den Korrekturlesern dieser Arbeit Benjamin Ellermann[4], Martin Klossek[5] und Martin Meedt[6]. Zahlreiche Anregungen bezüglich Struktur und Gliederung dieser Arbeit habe ich der an der Fachhochschule Stuttgart entstandenen *Dokumentvorlage für Diplomarbeiten und andere wissenschaftliche Arbeiten*[7] entnommen, deren Autor Prof. Dr. Wolf-Fritz Riekert gleichfalls mein Dank gilt.

Diese Arbeit wurde mittels *MS Word XP*, *MathType 5.0* und *Adobe Acrobat 6.0*[8] nach den Regeln der neuen Rechtschreibung erstellt.

[1] www.dbis.informatik.uni-frankfurt.de, www.informatik.uni-frankfurt.de
[2] www.uni-frankfurt.de
[3] www.dbis.informatik.uni-frankfurt.de/~tolle/index_e.html
[4] www.landenhausen.de
[5] www.klossek3000.de
[6] martin@meedt.de
[7] v.hdm-stuttgart.de/~riekert/theses/, v.hdm-stuttgart.de/~riekert/
[8] www.microsoft.com/office/, www.mathtype.com, www.adobe.de/products/acrobatstd/

1 Einleitung

1.1 Aufgabenstellung

Das Semantic Web ist in aller Munde, hat gegenwärtig jedoch noch nicht das Licht des World Wide Web erblickt. Nichtsdestotrotz existieren Teilbereiche, in welchen das Semantic Web bereits Anwendung findet. Zu den frühen Anwendern des Semantic Web werden vermutlich fachspezifische Informationsportale[9] zählen, so genannte *Infomediaries*[10]. In diesen Informationsportalen können Benutzer wesentlich präziser und umfangreicher nach Informationen suchen, als dies herkömmlicher Internetsuchmaschinen ermöglichen.

Aufgabenstellung ist es, eine Möglichkeit zur Interaktion zwischen dem Informationsportal und seinem Benutzer zu schaffen. Zu diesem Zweck ist eine Abfragesprache zu konzipieren, welche auch für technisch unversierte Benutzer ähnlich intuitiv einsetzbar ist, wie z. B. die Abfragesprachen der Internetsuchmaschine Google. Die zu konzipierende Abfragesprache soll auf der bereits vorhandenen Abfragesprache RQL [RQL-OVW] aufbauen, die Implementierung eines Prozessors zu erleichtern. Für die Durchführung der Abfragen gegen das zugrunde liegende RDF-Modell soll nach Möglichkeit der RDF-Parser VRP [VRP-HOME] eingesetzt werden.

Weiterhin Bestandteil der Aufgabenstellung ist die Implementierung eines Abfrageprozessors für die konzipierte Sprache mittels Java [JAVA-HOME].

1.2 Information Overkill – die Herausforderung

Die viel beschworene Informationsgesellschaft ist da: Globale Vernetzung für Jedermann, Internet-Zugänge für wenige Cents, 3,3 Milliarden sichtbare Webseiten[11], geschätzt mehrere hundert Milliarden unsichtbare Webseiten (Stand 2000), 17 neue Webseiten pro Sekunde [CARLSON].

Gemeinsam mit der Informationsgesellschaft kam die Informationsüberflutung – auch *data smog* genannt.

---

[9] Gemäß [TDUDEN]: `Portal [lat.-mlat.]: [„Vorhalle“] s; -s, -e: [prunkvolles] Tor, Pforte, großer Eingang`

[10] *Infomediaries* sind domänenspezifische Portalseiten mit einem oftmals redaktionell aufbereitetem Angebot fachlicher Informationen.

[11] Öffentliche und nicht-dynamische Webseiten laut *Google*, Stand 30.10.2003.

Internetsuchmaschinen – unzureichende Werkzeuge

Klassische Internetsuchmaschinen wie Google[12], AltaVista[13], AlltheWeb[14], … sind überlastet, und werden der Flut an Daten nicht mehr Herr. Dass sie das nicht mehr werden, liegt weniger im Web selber, als in der Funktionsweise der Internetsuchmaschinen an sich begründet:

Internetsuchmaschinen ermöglichen eine Volltextsuche nach Stichwörtern – nicht mehr, und nicht weniger. Vorstellbar als eine Art gigantisches Stichwortverzeichnis für das World Wide Web. Damit unterliegen sie allen Problemen, denen Stichwortverzeichnisse unterliegen – als da wären:

- **Einbeziehung von Synonymen**

  Gemäß [DUDEN]: `Synonym s; -s, -e: bedeutungsähnliches, -gleiches Wort, z. B. schauen statt sehen, Metzger statt Fleischer.`

  Diese Bedeutungsähnlichkeit ist für alle gebräuchlichen Internetsuchmaschinen problematisch: Die Suche nach „Teein“ beispielsweise bringt mit Google „lediglich“ 1.760 Fundstellen (Stand: 30. Oktober 2003). Die Suche nach dem Synonym „Koffein“ jedoch resultiert in 35.800 Fundstellen – zwanzig mal mehr.

  Ein Spezialfall dieser Problematik ist die Kulturabhängigkeit – nahezu sämtliche Begriffe, die als Suchbegriff in Frage kommen, sind sprach- und damit kulturabhängig. Die Suche nach „Desoxyribonukleinsäure“ beispielsweise bringt mit Google magere 4.170 Fundstellen (Stand: 30. Oktober 2003), die Suche nach „deoxyribonucleic acid“ – der englischsprachigen Schreibweise desselben Begriffes – hingegen 75.000 Fundstellen; das entspricht der siebzehnfachen Anzahl (noch nicht mit eingerechnet sind hier anderssprachige Varianten dieses Begriffes.)

- **Ignoranz von Homonymen**

  Gemäß [DUDEN]: `Homonym s; -s, -e: Word, das mit einem anderen gleich lautet, aber in der Bedeutung [u. Herkunft] verschieden ist, z. B. Lerche-Lärche; im weitesten Sinne auch: = Homogramm; vgl. aber: Homöonym.`

Eben jene gleich lautenden Wörter können Internetsuchmaschinen nicht unterscheiden. So findet die Suche nach „Java“ mit gebräuchlichen Suchmaschine Millionen von Fundstellen, z. B. mit Google 32.300.000 Fundstellen am 30. Oktober 2003. Davon beziehen sich die wenigsten Treffer auf die indonesische Insel Java[15], sondern vielmehr auf das Homonym der Programmiersprache Java[16].

[12] http://www.google.com
[13] http://www.altavista.com
[14] http://www.alltheweb.com
[15] http://de.wikipedia.org/wiki/Java_(Indonesien)
[16] http://java.sun.com

- **Rechtschreibfehler**

  Fehler in der Rechtschreibung wirken sich bei gebräuchlichen Internetsuchmaschinen verheerend aus: Die Suche nach „teakwondo“ beispielsweise ergibt mit Google lediglich 7.410 Fundstellen (Stand: 30. Oktober 2003), die Suche nach „taekwondo“ hingegen 267.000 – die 36-fache Anzahl.

  Einige Suchmaschinen wie z. B. Google versuchen mittlerweile, Schreibfehlern des Benutzers auf die Schliche zu kommen – sie weisen darauf hin, dass eine Abfrage mit anderen, aber ähnlichen Suchbegriffen zu einer größeren Anzahl an Fundstellen führt:

Abbildung 1 – Klassische Internetsuchmaschinen – Rechtschreibkorrektur

- **Wortformen**

  Variationen in der Wortform können gebräuchliche Internetsuchmaschinen nicht erkennen: Die Suche mit Google nach „finanziell“ beispielsweise erbringt zwar 270.000 Fundstellen (Stand: 30. Oktober 2003), die Suche nach „Finanzen“ jedoch 1.120.000 Fundstellen – vier mal mehr.

- **Sinnzusammenhang**

  Sinnverwandte Begriffe können Internetsuchmaschinen nicht erkennen: Die Suche mit Google nach „Steuergerechtigkeit“ beispielsweise erbringt lediglich 9.560 Fundstellen (Stand: 30. Oktober 2003), die Suche nach „Steuerrecht“ hingegen 137.000 Fundstellen.

Als wären diese Nachteile gewöhnlicher Suchmaschinen nicht genug, haben Suchmaschinen zusätzlich auch Probleme mit nahezu allen strukturierten Informationen, welche nicht HTML-kodiert sind: diese Informationen können von vielen Suchmaschinen gar nicht erst erfasst werden. Dazu zählen einerseits alle herstellerspezifischen Dateiformate wie z. B. PDF-, PPT-, DOC-, XLS-Dateien und zahlreiche mehr, allerdings auch nicht-proprietäre Dateiformate wie Postscript, MPEG, JPEG und zahlreiche weitere.

Mit wenigen Ausnahmen können Internetsuchmaschinen generell keine binären Dateien indizieren, Ausnahmen stellen hier wenige Suchmaschinen wie z. B. Google mit der Indizierung einiger Binärformate wie MS Word, PDF, Postscript, … dar:

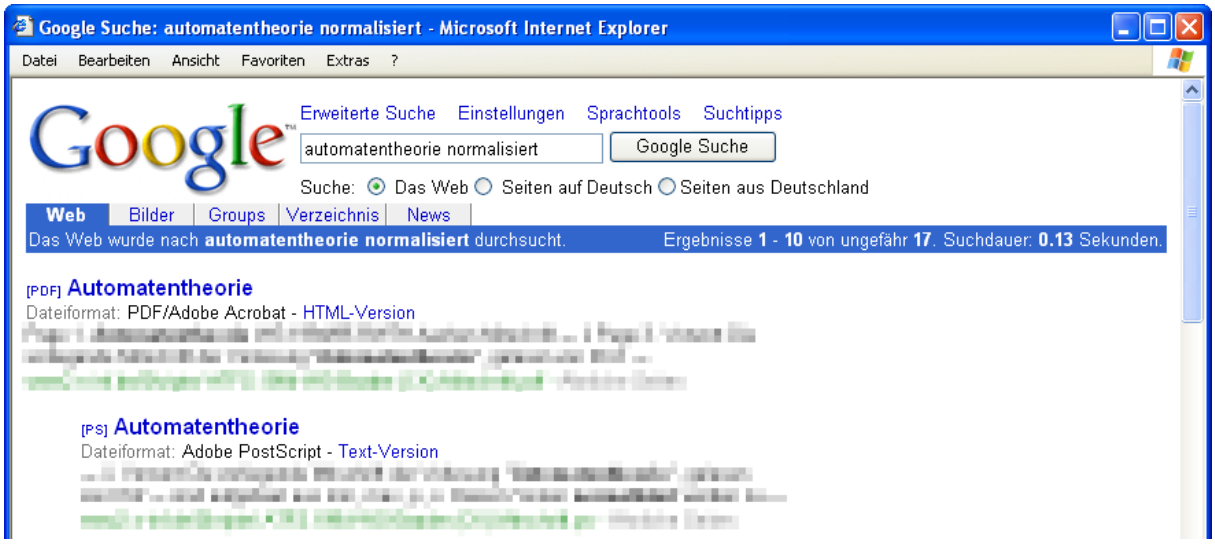

Abbildung 2 – Klassische Internetsuchmaschinen – Binärdateien

Aber selbst für diese Suchmaschinen stellen sämtliche nicht-textuellen Medien wie Grafikdateien, Videodateien, Klangdateien, … eine unüberwindbare Hürde dar.

Sollte eine Suchmaschine die gewünschten Informationen finden, so gibt sie in jedem Fall das vollständige Dokument zurück, bzw. einen Verweis darauf. Eine weitere Filterung der Rückgabe ist weder vorgesehen noch mit dem Prinzip der Suchmaschine vereinbar. Eine Abfrage wie „Geburtsdatum von Goethe“ führt zwar tatsächlich zu einer Seite, auf welcher das Geburtsdatum von Johann Wolfgang von Goethe erwähnt ist, jedoch liefert sie in der Tat das gesamte Dokument zurück – und nicht etwa nur die gewünschte Information: 28. August 1749.

1.3 Das Semantic Web – der Hoffnungsträger

Das Semantic Web, in der deutschsprachigen Literatur oft auch *Semantisches Web* bezeichnet, verspricht eine Lösung all der zuvor genannten Probleme:

Durch die Hinterlegung zusätzlicher maschinenlesbarer Sinnesinformationen zu den eigentlichen Dokumenten werden Maschinen wie z. B. Suchmaschinen und/oder Informationssysteme in die Lage versetzt, textuelle wie nicht-textuelle (d. h. binäre) Inhalte zu „verstehen“. Selbstverständlich können Maschinen diese Inhalte nicht im eigentlichen Sinne „verstehen“, jedoch können sie diese Inhalte exakter erfassen, als sie dies mit gewöhnlichen natürlichsprachlichen oder gar binären Inhalten könnten. Sie können dies, weil diese speziellen Sinnesinformationen besonders exakt sind: mittels einer formalsprachlichen Notation, welche oftmals RDF ist (siehe 2.2 Resource Description Framework (RDF).

Für die Formulierung von maschinenlesbaren Sinnesinformationen kann RDF bedenkenlos eingesetzt werden. Es existieren zahlreiche Parser, Autoren- und Validierungswerkzeuge für diesen Zweck. Für die Abfrage von RDF-Daten, zum Beispiel innerhalb von Informationsportalen, ist jedoch vor allem eine geeignete Abfragesprache erforderlich, die aber in einer intuitiven und von breiten Anwendermassen benutzbarer Form derzeit noch nicht existiert.

1.4 Inhalte und Struktur dieser Arbeit

Die Informationsüberflutung ist das Problem, und das Semantic Web auf Basis von RDF ein viel versprechender Lösungsansatz. In 2 Grundlagen wird zunächst ein Einblick in die Technologien des Semantic Web vermittelt – namentlich *RDF* und *RDF Schema.*

Für die Nutzung des Semantic Web in Informationsportalen bedarf es einer Ad Hoc-Abfragesprache, welche von den Besuchern der Portale unmittelbar genutzt werden kann – ohne technisches Fachwissen, ohne langwierige Einarbeitung und ohne Kenntnis der genauen Datenstrukturen. Diese Ansprüche an eine Ad Hoc-Abfragesprache werden in 3 Ziele erarbeitet.

Es existieren derzeit bereits zahlreiche Abfragesprachen für RDF, von denen eine Auswahl in 4 Stand der Technik vorgestellt wird. Jede dieser Abfragesprachen wird hinsichtlich der Ansprüche einer Ad Hoc-Abfragesprache untersucht. Es zeigt sich jedoch, dass keine der untersuchten Sprachen für diesen Zweck geeignet ist, denn jede untersuchte Sprache erinnert eher an SQL denn an Google: es besteht Bedarf für eine neuartige Ad Hoc-Abfragesprache.

Die neuartige Ad Hoc-Abfragesprache wird auf Basis der bereits bestehenden Abfragesprache *RQL* entwickelt, welche in 5 RQL – eine RDF Abfragesprache ausführlich vorgestellt wird. Entwickelt und definiert wird die Ad Hoc-Abfragesprache mit dem Namen *eRQL* in 6 eRQL – Ad Hoc-Abfragen für Informationsportale. Der Zusammenhang von *eRQL* und *RQL* ist die Art und Weise der Auswertung: eRQL-Abfragen werden in RQL-Abfragen umgewandelt, diese dann ausgewertet. Diese Umwandlung ist beschrieben in 7 Umwandlung von eRQL- in RQL-Abfragen.

Im Rahmen dieser Arbeit ist die Implementierung eines eRQL-Prozessors mit dem Namen *eRqlEngine* entstanden, welcher in 8 eRqlEngine – ein eRQL-Prozessor vorgestellt wird. Dieser Prozessor wandelt eRQL-Abfragen in eine oder mehrere RQL-Abfragen um, welche er an einen RQL-Prozessor weiterreicht. Ebenfalls im Rahmen dieser Arbeit ist auf Basis des RDF-Parsers *VRP* die Implementierung eines rudimentären RQL-Prozessors namens *RqlEngine* entstanden, welcher in 9 RqlEngine – ein RQL-Prozessor erläutert wird.

Während der Erarbeitung oder Entwicklung entstandene offene Fragen sowie Kritik und weitere Entwicklungsmöglichkeiten rund um eRQL, eRqlEngine und RqlEngine sind in 10 Ausblick zusammengetragen.

Das folgende Schaubild zeigt den Zusammenhang der Inhalte dieser Arbeit:

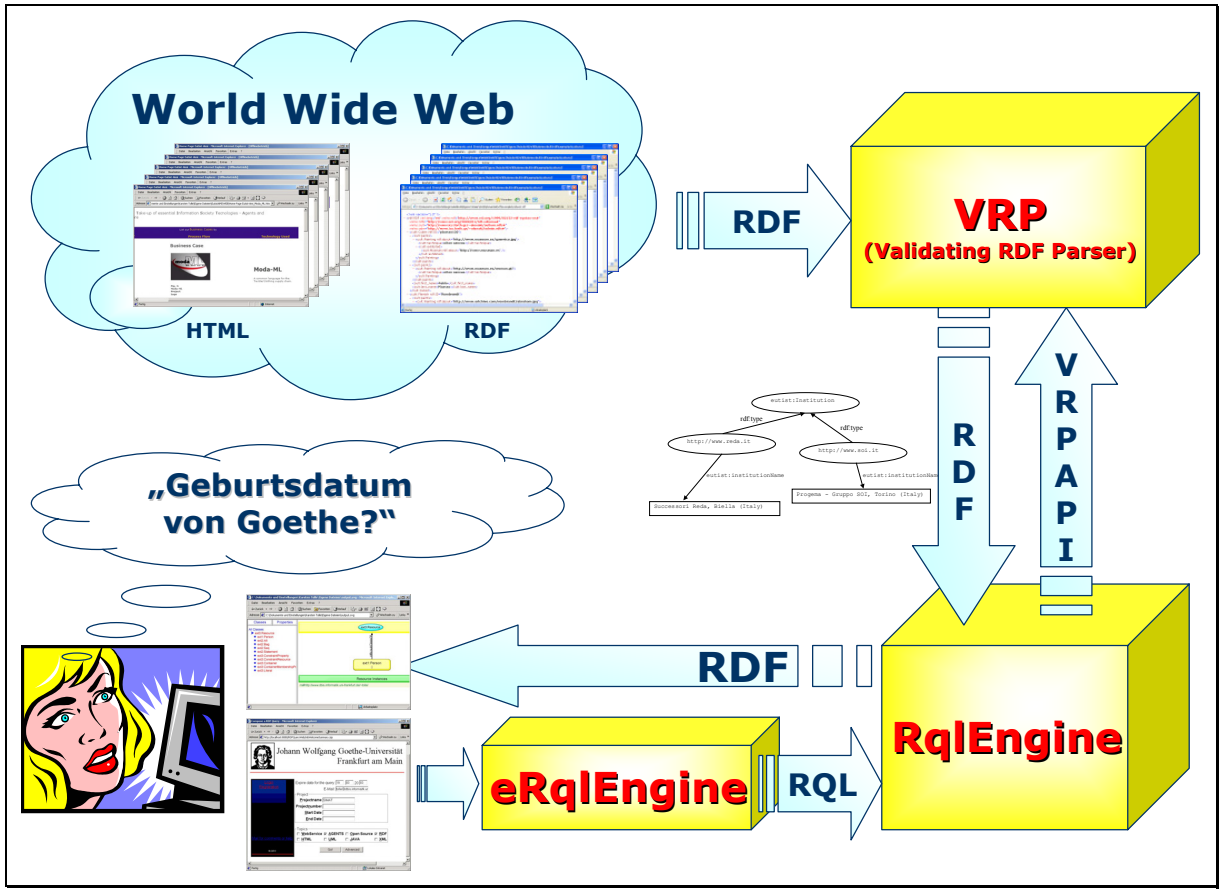

Abbildung 3 – Systemüberblick

2 Grundlagen

Das Semantic Web ist der Oberbegriff für zahlreiche Standards, Techniken und Ideen zur Abfrage, Übertragung, Speicherung und Verarbeitung von Daten, welche der maschinenlesbaren Hinterlegung von Sinnesinformationen dienen.

Unter *Sinnesinformationen* werden im Gegensatz zu einem Dokument an sich, welches im Allgemeinen auf einen menschlichen Leser zugeschnitten ist, zusätzliche Informationen verstanden, welche ausschließlich der Repräsentation des Sinngehaltes dienen. (Wohingegen ein gewöhnliches Dokument neben dem eigentlichen Sinngehalt auch Layout- und Strukturinformationen enthält.)

In diesem Kapitel werden spezielle Begrifflichkeiten des Semantic Web erläutert, so weit sie von dieser Arbeit betroffen sind. Insbesondere wird ein Einblick in RDF und RDF Schema vermittelt. Darüber hinausgehende Informationen können der bzw. den jeweiligen Spezifikation(en) entnommen werden [RDF-HOME].

2.1 Terminologie

Die im Folgenden aufgeführten Begriffe entstammen einer offiziellen Terminologie, und sind mit der Angabe des englischsprachigen Originalbegriffes und der URL der jeweiligen Spezifikation gekennzeichnet.

Wenn im Folgenden von Prädikaten wie `rdf:type`, `rdfs:subClassOf`, etc. gesprochen wird, so handelt es sich dabei um eine abgekürzte Schreibweise, welche zur Dokumentation verwendet wird: `rdf` bezieht sich in diesem Fall auf den Namensraume von RDF[17], `rdfs` auf den von RDF Schema[18]. Die ausgeschriebenen Namen dieser Prädikate lauten also `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` für `rdf:type`, bzw. `http://www.w3.org/2000/01/rdf-schema#subClassOf` für `rdfs:subClassOf`.

Begriffe der offiziellen Terminologie:

- **Alternative (englisch: alternative)**

  Eine *Alternative* repräsentiert eine nicht geordnete Menge beliebiger RDF-Ressourcen, welche gleichwertig in dem Sinne sind, dass sie alternativ verwendet werden können.

  Spezifikation: `http://www.w3.org/TR/REC-rdf-syntax/#Alternative`.

[17] http://www.w3.org/1999/02/22-rdf-syntax-ns
[18] http://www.w3.org/2000/01/rdf-schema#

- **Anonyme Ressource (englisch: anonymous resource)**

  Eine *Anonyme Ressource* ist eine spezielle *Ressource*, welche nicht über eine URI ansprechbar ist. Ein häufiges Einsatzgebiet für anonyme Ressourcen sind beispielsweise die *vergegenständlichten Aussagen*.

- **Aussage (englisch: statement)**

  Eine *Aussage* entspricht der Kombination von *Subjekt*, *Prädikat* und *Objekt*, und stellt einen Sachverhalt dar.

  Spezifikation: `http://www.w3.org/TR/REC-rdf-syntax/#Statement`.

- **Container (englisch: container)**

  Ein *Container* stellt gemäß der RDF-Terminologie eine spezielle *Ressource* dar, welche beliebig viele andere Ressourcen beinhaltet. Derzeit sieht die RDF-Spezifikation drei Arten von Containern vor: *Alternativen*, *Folgen* und *Sammlungen*.

  Spezifikation: `http://www.w3.org/TR/REC-rdf-syntax/#containers`.

- **Definitionsbereich (englisch: domain)**

  Der *Definitionsbereich* eines Prädikates ist die Menge der RDF-Ressourcen, welche Subjekt für dieses Prädikat sein dürfen, d. h. von welchen dieses Prädikat „ausgehen" darf. Der *Definitionsbereich* wird durch das Prädikat `rdfs:domain` spezifiziert.

  Spezifikation: `http://www.w3.org/TR/rdf-schema/#ch_domain`.

- **Folge (englisch: sequence)**

  Eine *Folge* repräsentiert eine geordnete Menge beliebiger RDF-Ressourcen.

  Spezifikation: `http://www.w3.org/TR/REC-rdf-syntax/#Sequence`.

- **Kante (englisch: edge)**

  Eine Kante repräsentiert ein *Prädikat* in der Graphendarstellung eines RDF-Modells, und verbindet eine *Ressource* mit einem Literal, oder mit einer weiteren *Ressource*. Kanten sind in der Graphendarstellung gerichtet, d. h. verfügen über einen expliziten Start- und Endknoten, welche im Allgemeinen nicht vertauscht werden können, ohne das Modell zu modifizieren (dies entspräche einer Vertauschung von *Subjekt* und *Objekt* gegenüber dem *Prädikat*).

- **Klasse (englisch: class)**

  Eine *Klasse* repräsentiert eine *class* im Sinne von RDF Schema, d. h. eine Menge gleichartiger RDF-Ressourcen. Für eine ausführliche Beschreibung siehe 5.3.1 Daten-, Schema- und Metaschemaebene.

Im Folgenden wird von *Klassen, Schemaklassen* und *Metaschemaklassen* die Rede sein. Klassen sind dabei der Oberbegriff für die Schemaklassen und Metaschemaklassen, oder allgemein: für alle Arten von Klassen.

Spezifikation: http://www.w3.org/TR/rdf-schema/#ch_classes.

- **Knoten (englisch: node)**

  Ein Knoten repräsentiert eine *Ressource* oder ein Literal in der Graphendarstellung eines RDF-Modells. Zwei Knoten können durch eine *Kante* (d. h. ein *Prädikat*) miteinander verbunden werden.

- **Literal (englisch: literal)**

  Ein *Literal* ist neben einer Ressource die zweite Möglichkeit, einer Aussage ein Objekt zuzuweisen. Während es sich bei Ressourcen aber um Dinge handelt, welche durch weitere Aussagen näher beschrieben werden können, sind Literale einfache Texte, Ziffern oder Datumsangaben.

  Spezifikation: http://www.w3.org/TR/REC-rdf-syntax/#literal.

- **Objekt (englisch: object)**

  Das *Objekt* ist diejenige Komponente einer RDF-Aussage, welche Endpunkt des Prädikates ist. In der Graphendarstellung eines RDF-Modells entspricht dies dem Endknoten einer jeden Kante, in der *Tripeldarstellung* der dritten Komponente eines jeden Tripels.

  Spezifikation: http://www.w3.org/TR/REC-rdf-syntax/#object.

- **Prädikat (englisch: property)**

  Das *Prädikat* ist diejenige Komponente einer (RDF-) Aussage, welche Subjekt und Objekt miteinander verbindet. In der Graphendarstellung eines RDF-Modells entspricht dies der Kante zwischen je zwei Knoten, in der *Tripeldarstellung* der zweiten Komponente eines jeden Tripels.

  Spezifikation: http://www.w3.org/TR/REC-rdf-syntax/#predicate.

- **Ressource (englisch: resource)**

  Eine *Ressource* ist der Oberbegriff für alles, was sich mittels RDF beschreiben lässt, und zur Beschreibung verwendet werden kann. Dabei kann es sich um beliebige Sachen aus der realen Welt handeln (Bücher, Firmen, Produkte, …), Sachen aus der virtuellen Welt (Websites, E-Mail-Adressen, …), Personen, oder beliebige andere „Dinge“.

  Spezifikation: http://www.w3.org/TR/REC-rdf-syntax/#resource.

- **Sammlung (englisch: bag)**

  Eine *Sammlung* repräsentiert eine nicht geordnete Menge beliebiger RDF-Ressourcen.

  Spezifikation: http://www.w3.org/TR/REC-rdf-syntax/#Bag.

- **Subjekt (englisch: subject)**

  Das *Subjekt* ist diejenige Komponente einer RDF-Aussage, welche Ausgangspunkt des Prädikates ist. In der Graphendarstellung eines RDF-Modells entspricht dies dem Startknoten einer jeden Kante, in der *Tripeldarstellung* der ersten Komponente eines jeden Tripels.

  Spezifikation: http://www.w3.org/TR/REC-rdf-syntax/#Subject.

- **Tripeldarstellung (englisch: N-Triples)**

  Die *Tripeldarstellung* ist ein zeilenweises und textbasiertes Speicherformat für RDF-Modelle. Jede Zeile entspricht dabei einer einzelnen Aussage, ein RDF-Modell mit *n* Aussagen wird also zu einer Textdatei mit *n* Zeilen. Jede Zeile enthält dabei die drei Komponenten *Subjekt*, *Prädikat* und *Objekt* in dieser Reihenfolge, separiert durch Leerraum (wie Leerzeichen, Tabulatoren, etc.).

  Diese Tripeldarstellung ist die einzige in Zusammenhang mit diesem Dokument verwendete. Allgemein existieren jedoch noch weitere Tripeldarstellungen, insbesondere existiert z. B. eine Tripeldarstellung mit der Reihenfolge Prädikat, Subjekt, Objekt.

  Spezifikation: http://www.w3.org/TR/rdf-testcases/#ntriples.

- **Unterklasse (englisch: subclass)**

  Eine *Unterklasse* repräsentiert eine so genannte *subclass* im Sinne von RDF Schema [RDFS], d. h. eine Spezialisierung einer anderen *Klasse*.

  Spezifikation: http://www.w3.org/TR/rdf-schema/#ch_subclassof.

- **Unterprädikat (englisch: subproperty)**

  Ein *Unterprädikat* repräsentiert ein so genanntes *subproperty* im Sinne von RDF Schema [RDFS], d. h. eine Spezialisierung eines anderen *Prädikates*.

  Spezifikation: http://www.w3.org/TR/rdf-schema/#ch_subpropertyof.

- **Vergegenständlichte Aussage (englisch: reified statement)**

  Unter einer *vergegenständlichten Aussage* versteht die RDF-Terminologie die Betrachtung einer Aussage (welche aus Subjekt, Prädikat und Objekt besteht) als eine neue Ressource. Diese Betrachtungsweise ermöglicht es, Aussagen über Aussagen zu formulieren.

In dem folgenden Beispiel aus [RDF] wird eine Aussage gefällt über die Aussage „"Ora Lassila" ist S:Creator von http://www.w3.org/Home/Lassila“ :

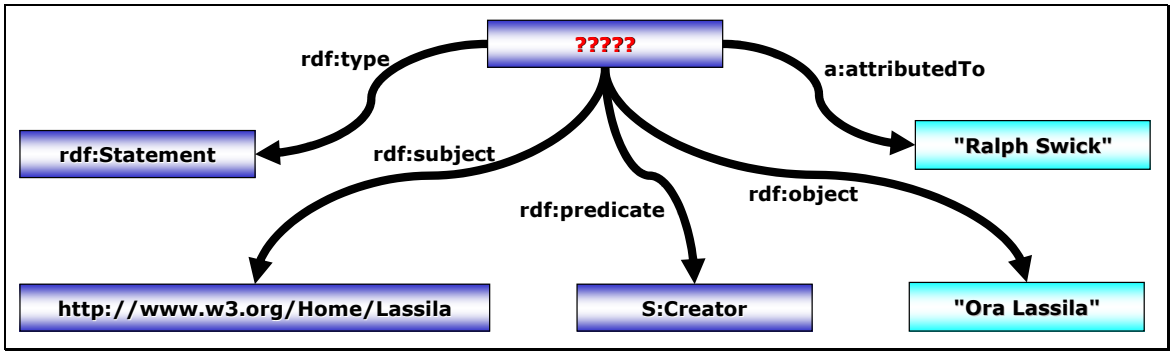

Abbildung 4 – Vergegenständlichte Aussagen

Die Ressource, welche die beschriebene Aussage darstellt, ist mit Fragezeichen betitelt – es handelt sich um eine Ressource ohne Namen, eine so genannte *anonyme Ressource*.

Spezifikation: `http://www.w3.org/TR/REC-rdf-syntax/#higherorder`.

- **Wertebereich (englisch: range)**

  Der *Wertebereich* eines Prädikates entspricht der Menge der RDF-Ressourcen, welche dieses Prädikat als Objekt verwenden darf, d. h. zu welchen dieses Prädikat „hinführen“ darf. Der Wertebereich wird durch das Prädikat `rdfs:range` spezifiziert.

  Spezifikation: `http://www.w3.org/TR/rdf-schema/#ch_range`.

Begriffe im Rahmen dieser Arbeit:

Während die obig aufgezählten Begriffe einer offiziellen Terminologie entstammen, sind die folgenden Begriffe eigene Wortschöpfungen, die ausschließlich im Rahmen dieser Arbeit Einsatz finden:

- **Berühren**

  Zwei Aussagen A_1 und A_2 *berühren* sich im Rahmen dieser Arbeit, wenn mindestens eine der Ressourcen bzw. Literale, die in A_1 enthalten sind, auch in A_2 enthalten ist.

$$A_1 \text{ berührt } A_2 \quad \Leftrightarrow \quad A_i = \{s_i, p_i, o_i\} \quad \wedge \quad A_1 \cap A_2 \neq \varnothing \tag{2.1}$$

  Anmerkung: Zwei Literale gelten als identisch, wenn ihre textuelle Repräsentation übereinstimmt (sie gelten auch dann als identisch, wenn Sie sich nur hinsichtlich der Groß- und Kleinschreibung unterschieden). Dieser Begriff ist nicht Bestandteil der offiziellen RDF-Terminologie.

- **Dokument**

  Ein *Dokument* repräsentiert im Rahmen dieser Arbeit einen logischen Verbund von Aussagen, welche bezüglich ihrer Lokalität zusammengehören. Dabei kann es sich z. B. um eine RDF-

Datei handeln, um eine spezielle RDF-Datenbank, aber auch um eine Internet-Domain. Dieser Begriff ist nicht Bestandteil der offiziellen RDF-Terminologie.

- **Point Of Interest**

  Ein Point Of Interest (POI) ist eine spezielle Art von Umgebung um eine (oder mehrere) Ressourcen (oder Literale). Die POI-Umgebung beinhaltet dabei all jene Ressourcen und Literale, welche in der Graphenrepräsentation des RDF-Modells zu den vorgegebenen Ressourcen bzw. Literalen benachbart sind.

  Dieser Begriff ist nicht Bestandteil der offiziellen RDF-Terminologie. Weitere Informationen siehe 6.1 Eigenschaften von eRQL.

- **Überlappen**

  Zwei Mengen von Aussagen M_1 und M_2 *überlappen* sich im Rahmen dieser Arbeit, wenn es aus beiden Mengen jeweils mindestens eine Aussage gibt, die sich berühren:

  $$M_1 \text{ überlappt } M_2 \quad \Leftrightarrow \quad \exists A_1 \in M_1, A_2 \in M_2 : A_1 \text{ berührt } A_2 \tag{2.2}$$

  Dieser Begriff ist nicht Bestandteil der offiziellen RDF-Terminologie.

- **Umgebung**

  Einzelne Aussagen oder gar einzelne Ressourcen haben im Sinne einer Suche innerhalb eines Informationsportals eine zu hohe Granularität. eRQL erweitert daher in verschiedenen Situationen eine einzelne Ressource (bzw. ein einzelnes Literal, bzw. eine Aussage) um dessen Umgebung. Diese beinhaltet neben der entsprechenden Aussage selbst alle weiteren Aussagen des RDF-Modells, welche bezüglich der Graphenrepräsentation zu dieser Aussage benachbart sind.

  Dieser Begriff ist nicht Bestandteil der offiziellen RDF-Terminologie. Weitere Informationen siehe 6.1.2 Umgebung.

- **verbunden**

  Zwei RDF-Ressourcen R_1 und R_2 werden im Rahmen dieser Arbeit als *mittels rdf:type verbunden* bezeichnet, wenn eine Aussage existiert, welche `rdf:type`, R_1 und R_2 als Prädikat, Subjekt und Objekt beinhaltet. Auf entsprechende Weise kann auch eine Ressource R mittels eines Prädikates mit einem Literal L verbunden sein.

2.2 Resource Description Framework (RDF)

RDF ist eine Empfehlung des *World Wide Web Consortium* [W3C] zum Austausch und zur Speicherung strukturierter wie semi-strukturierter Daten im World Wide Web (WWW) [RDF-FAQ]. Die RDF-Homepage des W3C ist unter [RDF-HOME] zu finden.

Anders als bei XML ermöglicht RDF die Vermischung von Daten verschiedener und voneinander unabhängiger Anwendungen.

Durch RDF können beliebige (Sinn-) Zusammenhänge zwischen ebenso beliebigen Objekten beschrieben werden; der Produktkatalog eines Online-Versandhauses genauso wie das Literaturverzeichnis einer wissenschaftlichen Veröffentlichung.

Um diesem überaus generellen Anspruch gerecht zu werden, beschränkt sich RDF im Wesentlichen auf zwei sehr allgemein gehaltene Vorschriften:

1. Ein RDF-Modell ist nichts anderes ist als eine Menge von beliebig vielen Aussagen, die in einer speziellen, maschinenlesbaren Notation formuliert sind.
2. Jede Aussage eines RDF-Modells besteht aus je genau einem Subjekt, einem Prädikat und einem Objekt.

In der englischsprachigen Terminologie wird von *statements* gesprochen, die sich aus je einem *subject*, *property* und *object* zusammensetzen.

2.2.1 Alles ist eine Ressource

Analog zu Javas „Alles ist ein Objekt" lautet das Motto von RDF: „Alles ist eine Ressource". Insbesondere ist auch jedes Prädikat lediglich eine spezielle Ressource. Auch jede Aussage ist, als Ganzes betrachtet, wiederum eine Ressource. Diese Betrachtungsweise ist nützlich, um Aussagen über Aussagen formulieren zu können.

Keine Regel ohne Ausnahme, und diese stellen die Literale dar. Genau genommen gilt: „Alles ist eine Ressource oder ein Literal". Eine ausführliche Diskussion dieser Dualität findet sich in [IOCTL].

2.2.2 Speicherung und Austausch von RDF

RDF und RDF Schema sind kein Dateiformat, und auch nicht als ein solches definiert. Bei der Definition von RDF und RDF Schema wird stattdessen ein mathematisches Modell verwendet, um die Semantik der zulässigen Operationen darauf mittels Formeln und logischen Operatoren definieren zu können.

Die Empfehlung des W3C sieht jedoch zwei Dateiformate vor, welche zur Serialisierung von RDF verwendet werden können, um eine Speicherung in Dateiform zu ermöglichen: ein erstes Dateiformat auf Basis von XML, und ein zweites auf Basis einer einfachen Textdatei. Mit der „Graphendarstellung" hat das W3C noch eine dritte Form der Darstellung von RDF definiert, welche jedoch in erster Linie dem Menschen dient, und weniger maschinenlesbar ist. Ein Beispiel für die Serialisierung eines RDF-Modells nach XML findet sich in 5.6 Rückgabe.

Das W3C empfiehlt allerdings keine relationale RDF-Repräsentation, welches für die Speicherung durch ein relationales Datenbankmanagementsystem (*RDBMS*) nötig ist. Hierbei geht derzeit jeder Hersteller einer RDF-Datenbank auf Basis eines RDBMS seinen eigenen Weg.

2.3 RDF Schema (RDFS)

RDF Schema (*RDFS*) ist eine Empfehlung des *World Wide Web Consortium* [W3C] zur Beschreibung von RDF-Vokabularen. Die RDFS-Homepage des W3C ist unter [RDFS] zu finden.

RDF Schema erlaubt es, RDF-Ressourcen zu klassifizieren. RDF an sich – ohne RDF Schema – unterscheidet lediglich zwischen allem, was ein Prädikat ist, und allem, was kein Prädikat ist. Mittels RDF Schema ist eine wesentliche exaktere Klassifizierung möglich.

In einem konkreten Einsatzgebiet für RDF ist es im Allgemeinen nicht erwünscht, Informationen über beliebige Objekte speichern und verarbeiten zu können. So erwartet eine B2B-Anwendung des oben genannten Produktkataloges, in diesem ausschließlich Produktinformationen vorzufinden, und sicherlich keine Literaturverweise – auch wenn es sich in beiden Fällen um RDF handelt. An dieser Stelle kommt RDF Schema als Typsystem ins Spiel:

Mittels RDF Schema lassen sich die Ressourcen zu so genannten „Klassen" gruppieren. Dazu wird entweder eine bestehende Klasse verwendet, oder zunächst eine neue Klasse definiert. Anschließend werden alle Ressourcen, welche dieser Klasse zugehörig sein sollen, dieser Klasse zugewiesen. RDF Schema stellt die Mittel zur Verfügung, um neue Klassen zu definieren, und Ressourcen zu klassifizieren.

2.3.1 RDF ist nicht RDF Schema

RDF und RDF Schema sind unabhängig voneinander. Insbesondere ist RDF nicht auf RDF Schema angewiesen, sondern kann ebenso mit anderen Schemasprachen kombiniert werden, sofern diese geeignet sind. Insbesondere kann RDF auch völlig ohne Schemasprache eingesetzt werden.

Ebenso wenig wie jede XML-Anwendung DTD oder gar XML Schema unterstützt, muss auch nicht jede RDF-Anwendung RDF Schema unterstützen. Oftmals treten RDF und RDF Schema aber im Doppelpack auf, so dass man sie als Komplementärtechnologien betrachten darf: wo RDF ist, ist auch RDF Schema zumeist nicht weit. Die Kombination von RDF und RDF Schema wird im Folgenden unter der Bezeichnung *RDF/S* zusammengefasst.

2.3.2 Vordefinierte RDF Schemaklassen

RDF Schema erlaubt die Definition von RDF Schemaklassen und -Prädikaten, und bedient sich dafür nichts Weiterem als der bereits bekannten Mittel: RDF-Aussagen. Zu diesem Zweck stellt RDF Schema von Haus aus bereits vordefinierte RDF Schemaklassen und -Prädikate zur Verfügung, welche verwendet werden müssen, um eigene RDF Schemaklassen zu definieren [RDFS].

2.3.3 Vergleich zu XML Schema

Anders als im Fall von *XML Schema* [XMLSCHEMA], wo ein Schema eine bestimmte Art von Dokument vollständig und abschließend beschreibt (auch *Grammatik* genannt), werden mit

RDF Schema so genannte *Vokabulare* beschrieben. Der entscheidende Vorteil der Vokabulare besteht darin, dass sie miteinander kombiniert werden können – auch innerhalb ein und desselben Dokumentes. Mit XML Schema ist dies nicht möglich. Dadurch unterstützt RDF Schema ausdrücklich die Vermischung von Daten verschiedener und voneinander unabhängiger Anwendungen.

2.3.4 Vergleich zu Typsystemen der OOP

Typsysteme wie RDF Schema gibt es überall dort, wo große Mengen an Daten vorhanden sind. Bekannte Typsysteme finden sich zum Beispiel in Zusammenhang mit relationalen Datenbanksystemen und *SQL* (*DDL* bzw. *data definition language*), aber vor allem auch bei Programmiersprachen.

Die Verwendung der Begriffe *class* und *property* in der englischsprachigen RDF-Terminologie legt nahe, einen Vergleich mit Typsystemen der objektorientierten Programmierung (OOP) anzustellen, wie sie aus Sprachen wie JAVA, Delphi, C++, C#, … bekannt sind. Und in der Tat handelt es sich dabei um eine ähnliche Technik.

Vergleicht man RDF Schema aber mit Typsystemen, wie sie aus der objektorientierten Programmierung bekannt sind, so fallen die vier folgenden, gravierenden Unterschiede ins Auge:

1. Mehrfachklassifikation
2. Erweiterbares Metaschema
3. Autarke Prädikate
4. Mehrfache Prädikate

Weiterhin existieren noch einige weniger schwerwiegendere Unterschiede, wie z. B.:

- Funktionen à la **`topclass`** und **`leafclass`** stehen in der objektorientierten Programmierung zumeist nicht zur Verfügung
- In der objektorientierten Programmierung können Klassen oft durch so genannte *Modifier* beeinflusst werden, und z. B. zu einer abstrakten, statischen oder privaten (…) Klassen werden – diese Möglichkeiten sieht RDF Schema derzeit nicht vor

Mehrfachklassifikation

In der Welt der objektorientierten Programmierung führt die Mehrfachklassifikation eher ein exotisches Dasein, und wird unter den verbreiteten Sprachen lediglich von C++ unterstützt – alle anderen Sprachen verzichten darauf, und erlauben lediglich die Zuordnung zu einer einzigen Klasse.[19] Eine Gemeinsamkeit ist es jedoch, dass keine Ressource völlig „klassenlos“ sein kann – selbst eine nicht explizit klassifizierte Ressource gehört implizit stets der Klasse `rdfs:Ressource` an.

[19] Auf das Konzept der Schnittstellen aus z. B. Java, Delphi und .NET soll an dieser Stelle nicht eingegangen werden, da es keine „vollständige“ Vererbung einer Klasse, sondern nur die Vererbung der Schnittstelle einer Klasse realisiert.

Erweiterbares Metaschema

Unter einem Metaschema versteht man all jene Mittel, die zur Verfügung stehen, um ein Schema zu definieren.

In der Welt der objektorientierten Programmierung besteht das Metaschema einer Programmiersprache zumeist aus speziellen Schlüsselwörtern (z. B. class, extends, inherits, final, public) und speziellen syntaktischen Konstrukten (z. B. Klammerung durch { und }). Insbesondere sind die Metaschemata zumeist ein fester Bestandteil der jeweiligen Sprache, und können weder modifiziert noch erweitert werden.

Als Beispiel sei der folgende JAVA-Quellcode gegeben, das Prinzip lässt sich aber auf andere objektorientierte Sprachen übertragen:

```
public class Kunde extends Person { … }
```

Obige JAVA-Anweisung deklariert eine Klasse „Kunde“, und definiert sie als Erweiterung der bestehenden Klasse „Person“. Dazu werden die Metaschema-Mittel public, class und extends eingesetzt, sowie je eine öffnende bzw. schließende geschweifte Klammer zur Kennzeichnung des Klassenbeginns bzw. des Klassenendes.

Angenommen, der Autor der Klasse möchte diese Klasse „persistierbar“ gestalten (d. h. sie mit Funktionen zum Speichern und Einlesen versehen), ohne diese Funktionen direkt in dieser Klasse zu implementieren, wird er diese Funktionalität vermutlich in einer neuen Klasse implementieren, und die bestehende Klasse von dieser neuen Klasse ableiten:

```
public class PersistierbareKlasse { … }
public class Kunde extends PersistierbareKlasse { … }
```

Offenbar ist die Klasse „Kunde“ nun nicht mehr von „Person“ abgeleitet, obwohl der Sinnzusammenhang „Jeder Kunde ist eine spezielle Art von Person“ unverändert wahr ist. Das fixe Metaschema von JAVA und anderen Programmiersprachen „zwingt“ Entwickler gewissermaßen dazu, Metaschema-Eigenschaften mittels Schema-Eigenschaften nachzubauen. Würde auch JAVA über ein erweiterbares Metaschema wie RDF Schema verfügen, wäre möglicherweise die folgende Konstruktion möglich:

```
public metaclass PersistierbareKlasse { … }
public PersistierbareKlasse Kunde extends Person { … }
```

Diese Möglichkeit bietet leider weder JAVA, noch eine andere der genannten OO-Sprachen – jedoch RDF Schema bietet sie. (Sprachen mit Mehrfachvererbung wie z. B. C++ bieten diese Möglichkeit zwar, jedoch besitzt auch C ein unveränderliches Metaschema.)

Autarke Prädikate

Hinter diesem etwas merkwürdig klingenden Titel verbirgt sich die Idee, Prädikate – d. h. Eigenschaften von Klassen – unabhängig von diesen Klassen selber zu definieren. Gängige Praxis in

der objektorientierten Programmierung ist es, bei der Definition einer Klasse zugleich die Namen und Datentypen der verfügbaren Eigenschaften dieser Klasse festzulegen:

```
public class Kunde extends Person {
  public int KundenNummer;
}
```

RDF Schema beschreitet einen anderen Weg, indem Klassen und Prädikate zunächst unabhängig voneinander definiert werden. Bei der Definition eines Prädikates kann – optional – bestimmt werden, auf welche Klassen dieses Prädikat angewendet werden kann (Definitionsbereich), und welche Klassen als Wert verwendet werden dürfen (Wertebereich):

```
<rdfs:Class rdf:ID="Kunde">
  <rdfs:subClassOf rdf:resource="#Person"/>
</rdfs:Class>
<rdf:Property rdf:ID="KundenNummer">
  <rdfs:domain rdf:resource="#Kunde"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</rdf:Property>
```

Mehrfache Prädikate

Sofern ein Prädikat auf eine RDFS-Klasse angewendet werden darf, kann es beliebig oft auf diese Klasse angewendet werden – oder überhaupt nicht.

Anders als in der objektorientierten Programmierung, wo ein Prädikat maximal einen Wert beinhalten kann, und in vielen Fällen (z. B. bei Grunddatentypen wie integer, long, etc.) auch nicht „leer“ sein kann, können RDF-Prädikate – sofern sie mit der jeweiligen Klasse und dem jeweiligen Wert verwendet werden dürfen – keinmal, einmal oder beliebig oft eingesetzt werden:

```
<rdfs:Class rdf:ID="Person"/>
<rdf:Property rdf:ID="TelefonNummer">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
```

Dies würde auf JAVA bezogen einen Quelltext wie den folgenden zulassen:

```
public class Person {
  public string TelefonNummer = "069 12345678";
  public string TelefonNummer = "0170 1234567";
  public string TelefonNummer = "0172 1234567";
}
```

3 Ziele

In 1 Einleitung wurden das Problem des Information Overkill, sowie die Bedeutung des Semantic Web als ein viel versprechender Lösungsansatz erläutert. Anschließend wurden in 2 Grundlagen detailliert die Werkzeuge vorgestellt, welche in Zusammenhang mit dem Semantic Web bereits heute zur Verfügung stehen.

Zu den frühen Anwendern des Semantic Web werden vermutlich fachspezifische Informationsportale zählen, so genannte *Infomediaries*. Damit Besucher der Informationsportale die gewünschten Informationen schnell und präzise finden können, ist eine entsprechende Benutzungsschnittstelle notwendig – beispielsweise auf einer Abfragesprache basierend. Diese Abfragesprache muss auch für technisch unversierte Benutzer ähnlich intuitiv einsetzbar sein, wie z. B. die Abfragesprache der Internetsuchmaschine Google. Weiterhin muss es mittels dieser Abfragesprache möglich sein, ohne Kenntnisse der Datenstruktur zu operieren – weshalb sie im Folgenden als *Ad Hoc-Abfragesprache* bezeichnet wird.

Den folgenden Ansprüchen muss eine Ad Hoc-Abfragesprache für Informationsportale gerecht werden:

1. **Einfachheit**

   Einfache Abfragen sollten ohne spezielle Vorkenntnisse möglich sein. Eine Gewöhnung an die Abfragesprache sollte eher in Sekunden denn in Minuten oder gar Stunden möglich sein.

2. **Schemaunabhängigkeit**

   Abfragen müssen ohne Kenntnisse über die Struktur des RDF-Modells möglich sein. Der Abfragesteller muss nicht über die Kenntnis verfügen, ob z. B. das Prädikat `author` dem Namensraum A oder B entstammt, ob es ein textuelles Literal oder eine Ressource als Objekt benötigt, und in welchem Kontext es überhaupt angewendet werden darf.

3. **Mächtigkeit**

   Mächtige Abfragen sollten durch bottom-up-Konstruktion aus einfachen Abfragen zusammensetzbar sein.

4. **Domänenunabhängigkeit**

   Die Abfragesprache darf nicht auf eine spezielle Domäne zugeschnitten, sondern muss in jeder Domäne gleichermaßen einsetzbar sein.

5. **Erweiterbarkeit**

   Die Syntax der Abfragesprache soll ausreichend flexibel sein, um mit steigenden Ansprüchen unter Wahrung der Rückwärtskompatibilität mitwachsen zu können.

6. Schemaunterstützung

Zusätzlich zu der Möglichkeit der Suche sollte die Abfragesprache eine Schema-Unterstützung mitbringen, naheliegenderweise eine Unterstützung für *RDF Schema*. Erst durch diese Unterstützung kann der Abfragesteller durch den Datenbestand eines Informationsportals navigieren (browsen), ohne gezielt suchen zu müssen.

4 Stand der Technik

4.1 Eignung existierender Abfragesprachen für Ad Hoc-Abfragen

Zur Abfrage von RDF-Daten existieren bereits verschiedene Abfragesprachen und Abfrageschnittstellen, von denen eine Auswahl im Folgenden untersucht wird:

- *RQL*
- *rdfDB*
- *RDQL*
- *SquishQL*
- *SeRQL*
- *METALOG*
- *RDF API Draft* (API)

Weiterhin wird die Verwendung von XML Abfragesprachen in Betracht gezogen. Worin liegt nun der Unterschied zwischen einer Abfragesprache und einer Abfrageschnittstelle? Eine Abfrageschnittstelle, genannt *API* (*Application Programming Interface*), bietet zumeist elementare Methoden für den Zugriff auf eine Sammlung von Daten. Der Entwickler einer Software kann diese Methoden für seine eigenen Zwecke benutzen, indem er sie von seiner Software aufrufen lässt, und die Rückgabewerte verarbeitet. Eine Abfragesprache ist zumeist auf höherem Abstraktionsniveau angesiedelt, und eher deklarativ. Entsprechende Beispiele aus der XML-Welt sind DOM oder SAX (APIs) auf der einen Seite, und XPath, XQuery oder XSLT (Abfragesprachen) auf der anderen Seite.

Im Folgenden werden die gebräuchlichsten Abfragetechniken für RDF vorgestellt und hinsichtlich ihrer Eignung als Ad Hoc-Abfragesprache für Informationsportale bewertet. Diese Auflistung erhebt keinen Anspruch auf Vollständigkeit, da zwischenzeitlich eine große Zahl an RDF-Abfragesprachen und Sprachvarianten entstanden ist, welche jedoch in den wesentlichen Eigenschaften mit den genannten Sprachen übereinstimmen. An diese Betrachtungen schließt sich eine Zusammenfassung der Ergebnisse an, siehe 4.1.9 Resüme.

4.1.1 XML Abfragesprachen

Eine der interessantesten Eigenschaften von RDF ist die Möglichkeit zur XML-Serialisierung: ein RDF-Modell kann in einem exakt spezifizierten Format als XML-Dokument dargestellt werden.

Für XML-Dokumente stehen wiederum zahlreiche Abfragesprachen zur Verfügung, die erprobt und verbreitet sind. Dazu zählen z. B. XPath [XPATH], XSLT [XSLT] und XQuery [XQUERY].

Es stellt sich unweigerlich die Frage, ob nicht bestehende XML-Abfragemöglichkeiten wie XQuery, XPath und XSLT auch für die Abfrage von RDF-Daten herangezogen werden können.

Die folgenden Aspekte sprechen für bzw. wider den Einsatz von XML Abfragesprachen:

+ Für die Verwendung einer dieser XML Abfragesprachen spricht deren Mächtigkeit, Verbreitung und Kostengünstigkeit. Dies trifft sowohl auf XPath, XSLT als auch XQuery zu. Für alle Sprachen existieren leistungsstarke und erprobte Implementierungen auf verschiedenen Plattformen.

– Gegen die Verwendung sprechen die grundsätzlich unterschiedlichen Datenmodelle, welche XML einerseits und RDF andererseits zugrunde liegen: im Falle von XML kommt ein hierarchisches (d. h. baumförmiges) Datenmodell zum Einsatz, im Falle von RDF ein graphenförmiges:

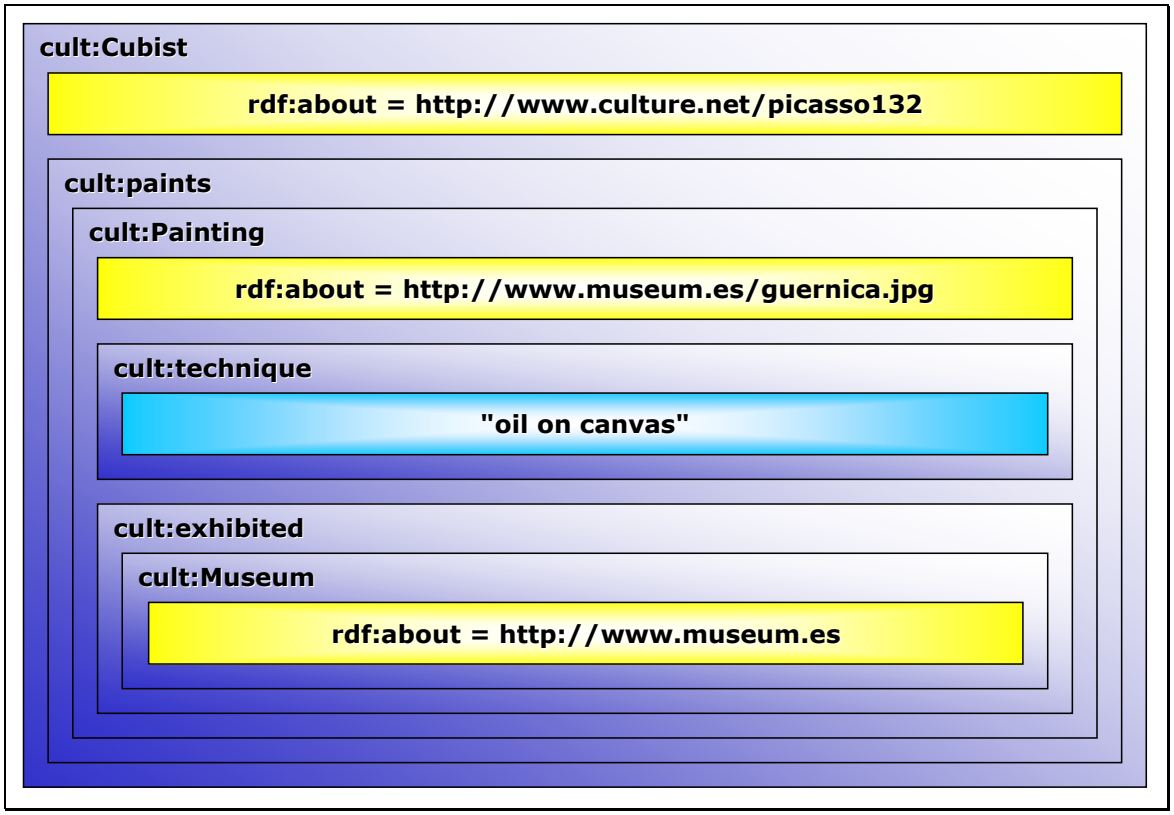

Abbildung 5 – Datenmodell von XML

Im Falle von RDF kommt jedoch ein graphenförmiges Datenmodell zum Einsatz:

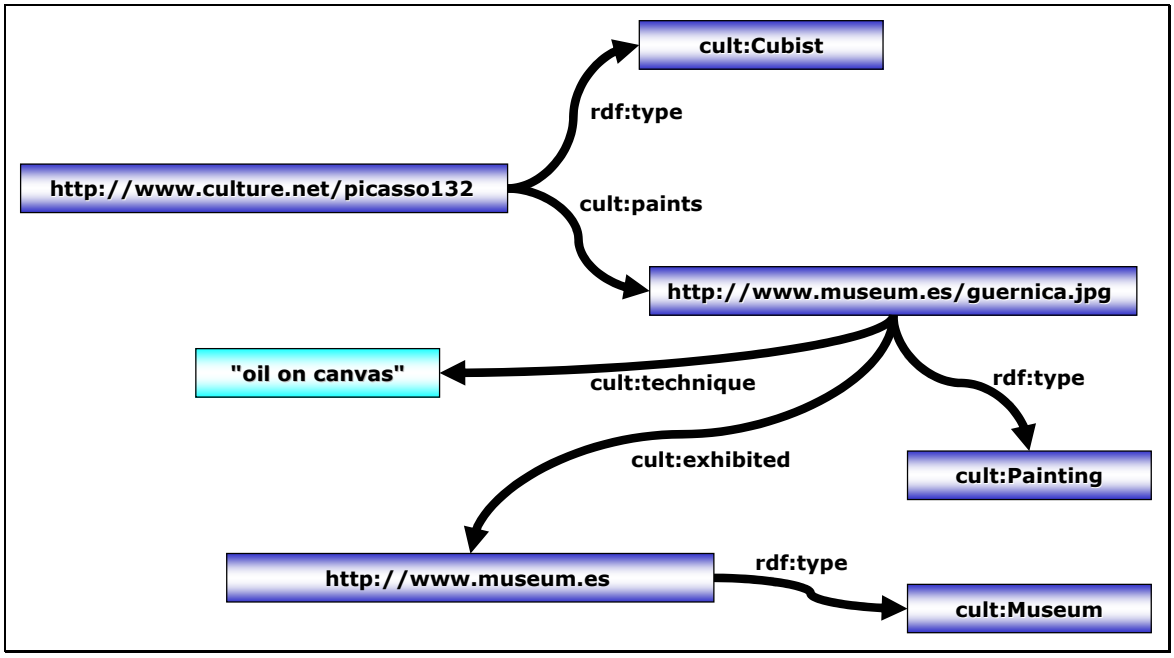

Abbildung 6 – Datenmodell von RDF

- Gegen die Verwendung spricht weiterhin die Tatsache, dass die Abbildung von RDF nach XML nicht eindeutig ist: Es gibt mehrere Möglichkeiten, ein RDF-Modell als XML-Dokument darzustellen. Daraus resultiert, dass entweder von einer normalisierten Variante der Serialisierung ausgegangen werden muss, oder eine entsprechend universelle XPath-, XSLT- bzw. XQuery-Abfrage verwendet werden muss.
- Letztlich spricht gegen alle genannten XML Abfragesprachen, dass sie nicht ausreichend intuitiv sind, um auch von einem technischen Laien verwendet werden zu können. Die Abfrage von RDF-Daten durch XPath sieht beispielsweise wie folgt aus[IWI-IUK]:

```
//rdf:Description[dc:*]/@about
```

Resümee

Gleich mehrere XML Abfragesprachen sind mächtig, erprobt, verbreitet und kostengünstig. Die völlig verschiedenen Datenmodelle, auf denen XML und RDF basieren, stehen einer Anwendung als Ad Hoc-Abfragesprache für RDF-Daten jedoch im Wege.

Für den Sonderfall, dass auf die Erstellung von RDF-Dokumenten Einfluss genommen werden kann, bzw. die XML-Struktur von RDF-Dokumenten bekannt ist, sind XML Abfragesprachen ein probates Mittel für bestimmte RDF-Abfragen. Für allgemeine RDF-Abfragen, insbesondere gegen beliebige RDF-Dokumente, sind XML Abfragesprachen dagegen nicht geeignet.

Weitere Überlegungen zum Einsatz von XML Abfragesprachen zur RDF-Abfrage sind unter [IWI-IUK] verfügbar.

4.1.2 RQL

RQL ist eine funktionale und typsichere Abfragesprache für RDF-Daten, und wurde von Greg Karvounarakis[20] am *ICS-FORTH* [ICS-FORTH] in Griechenland entwickelt. RQL ist Bestandteil der *RDFSuite* [RDFSUITE], und steht unter der GPL-kompatiblen *RDFSuite license* [RQL-OVW]. Eine ausführliche Beschreibung von RQL findet sich in 5 RQL – eine RDF Abfragesprache.

RQL operiert auf einem Graphenmodell (nicht etwa auf den einzelnen RDF-Tripeln) und bietet so genannte *Pfadausdrücke*, welche die Navigation innerhalb des Graphens erlauben, und somit Selektion und Projektion auf dem Modell unterstützen [RQL-MAN].

Charakteristisch für RQL ist dessen weit reichende und tief integrierte Unterstützung für RDFS (RDF Schema). Durch Metaschema-Abfragen ist das Browsen von RDF-Modellen möglich.

RQL bietet mächtige Funktionen wie Gruppierungsfunktionen, Arithmetische Funktionen, Aggregierungsfunktionen, Unterstützung für XML Schema, Unterstützung für XML-Namensräume, Quantoren und Vieles mehr [RQL-OVW].

Beispiele zu RQL-Abfragen

```
SELECT
     X, Y
FROM
     {X}last_modified{Y}
WHERE
     Y = max(SELECT Z FROM {W}last_modified{Z})

SELECT
     $X, $Y
FROM
     {$X}creates{$Y}

SELECT
     X, Y
FROM
     subclassof(domain(creates)){X}, subclassof(range(creates)){Y}
```

Derzeit existiert eine vollständige Implementierung eines RQL-Prozessors, die Referenzimplementierung des Autors. Sie kann unter [RQL-Demo] online getestet werden. Weiterhin unterstützt das *sesame*-Framework (siehe 4.1.6 SeRQL) eine Teilmenge von RQL.

Gegen die Verwendung von RQL als Ad Hoc-Abfragesprache spricht dessen hohe Komplexität und die dadurch bedingte mangelnde Intuitivität. Für die Verwendung von RQL spricht dessen sehr gute Unterstützung von RDF Schema, sowie die extrem einfache Notation der Kurzschreibweise.

[20] mgregkar@ics.forth.gr

4.1.3 rdfDB

rdfDB Query Language [RDFDB-QL] ist die SQL-ähnliche Abfragesprache der Open-Source RDF-Datenbank *rdfDB* [RDFDB] von R. V. Guha[21]. Diese Abfragesprache operiert auf einem Graphenmodell und bietet neben den Möglichkeiten zur Datenabfrage auch Möglichkeiten zur Datenmanipulation.

Beispiele zu rdfDB Query Language-Abfragen

```
SELECT
     ?x
FROM
     test1
WHERE
     (worksFor ?x W3C) (name ?x ?y)

INSERT INTO
     test1 (worksFor DanB W3C) (worksFor DanC W3C)
```

Aufgrund seiner Anlehnung an SQL ist die rdfDB Query Language sicherlich eine gute Wahl für Anwender, die bereits mit SQL, und den dahinter stehenden Konzepten wie Selektion und Projektion vertraut sind. Unbedarfte Anwender oder gar technische Laien dürften einige Zeit benötigen, sich diese Konzepte anzueignen und die Sprache einsetzen zu können.

4.1.4 RDQL

RDQL ist eine SQL ähnliche, deklarative RDF-Abfragesprache [RDQL], basierend auf *SquishQL*, siehe 4.1.5 SquishQL. RDQL ist Bestandteil des Frameworks *Jena*[22].

Beispiele für RDQL-Abfragen

```
SELECT
     ?x,?y
FROM
     <http://example.com/sample.rdf>
WHERE
     (?x,<dc:name>,?y)
USING
     dc FOR <http://www.dc.com#>

SELECT
     ?x
WHERE
     (<http://somewhere/res1>, <http://somewhere/pred1>, ?x)
```

[21] rdfdb-feedback@guha.com

[22] http://www.hpl.hp.com/semweb/jena2.htm

```
SELECT
     ?x, ?y
WHERE
     (<http://never/bag>, ?x, ?y)
AND
     ! ( ?x eq <rsyn:type> && ?y eq <rsyn:Bag>)
USING
     rsyn FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Auch für *RDQL* gilt, dass es aufgrund seiner SQL-Anlehnung eher für technisch versierte denn unversierte Anwender geeignet ist.

4.1.5 SquishQL

SquishQL ist die Sprache des RDF-Abfrageprozessors *Inkling* [SquishQL] und basiert auf *rdfDB* (siehe 4.1.3 rdfDB). Dieser Prozessor steht unter der GPL[23] und wurde (unter anderem) von Libby Miller[24], Dan Brickley[25] und Leigh Dodds[26] in JAVA implementiert. *Inkling* kann als Datenbestand sowohl Postgres-Datenbanken, als auch In-Memory-Datenbanken verwenden.

Beispiele für SquishQL-Abfragen

```
SELECT
     ?x, ?l, ?c
FROM
     file:rdf/Job.rdf
WHERE
     (web::type ?x rdfs::Class)(rdfs::label ?x ?l)(rdfs::description ?x ?c)
USING
     web FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
     rdfs FOR http://www.w3.org/2000/01/rdf-schema#
```

```
SELECT
     ?feedUrl, ?title
WHERE
     (dc::subject ?feedUrl ?subject) (rss::title ?feedUrl ?title)
AND
     ?subject ~ 'economics'
USING
     rss for http://purl.org/rss/1.0/
     dc for http://purl.org/dc/elements/1.1
```

Auch für *SquishQL* gilt, dass es aufgrund seiner SQL-Anlehnung eher für technisch versierte denn unversierte Anwender geeignet ist.

[23] http://www.gnu.org/copyleft/
[24] libby.miller@bristol.ac.uk
[25] daniel.brickley@bristol.ac.uk
[26] leigh@xmlhack.com

4.1.6 SeRQL

SeRQL (gesprochen: „circle“) ist die RDF-Abfragesprache von *sesame*[27], einem Open-Source-Framework zur Ablage und Abfrage von RDF-Daten. *Sesame* wurde von der niederländischen Firma *aidministrator*[28] in JAVA entwickelt, und steht unter der LGPL[29]. Neben *SeRQL* unterstützt *Sesame* (teilweise) die Abfragesprachen *RQL* (siehe 4.1.2 RQL) und *RDQL* (siehe 4.1.4 RDQL). *SeRQL* bietet Unterstützung für XML Schema, Pfadausdrücke, und vieles mehr [SERQL].

Beispiel für SeRQL-Abfragen

```
SELECT
     Author, Paper
FROM
     {Paper} <rdf:type> {<foo:Paper>};
     <foo:keyword> {"RDF", "Querying"};
     <dc:author> {Author}
USING NAMESPACE
     dc = <!http://purl.org/dc/elements/1.0/>,
     foo = <!http://www.foo.org/bar#>
```

Auch für *SeRQL* gilt, dass es aufgrund seiner SQL-Anlehnung eher für technisch versierte denn unversierte Anwender geeignet ist.

4.1.7 METALOG

METALOG ist eine Abfragesprache für RDF, die gänzlich anderer Natur als die bisher vorgestellten Abfragesprachen. *METALOG* wurde 1998 vorgestellt [METALOG-98] und ist die erste logische Abfrageschicht für RDF. *METALOG* verwendet zu diesem Zweck logische Formeln, insbesondere Implikationen.

Beispiel für eine METALOG-Abfrage

```
NAMESPACE URI
     "http://purl.org/schemas/DublinCore/RDF"
ALIAS
     uri1
IF
     { uri1:Creator(Doc,Person) AND uri1:Language(Doc,Language) }
THEN
     { Speaks (Person, Language) }
```

METALOG ist stärker an die natürliche Sprache angelehnt als andere Abfragesprachen, und versucht insbesondere auch dem Kriterium „Einfache Abfrageerstellung“ gerecht zu werden [METALOG-98, Kapitel „METALOG“), weshalb es für den Einsatz als Ad Hoc-Abfragesprache zunächst in die nähere Betrachtung kommt.

[27] http://sesame.aidministrator.nl
[28] http://www.aidministrator.nl
[29] http://www.gnu.org/licenses/lgpl.html

Letztlich erinnert die Konstruktion mit IF-THEN-Konstrukten aber eher an eine Programmiersprache oder an eine Inferenzmaschine (zur Ableitung von Wissen aus vorhandenem Wissen), denn an eine natürliche Sprache, so dass auch METALOG für den unbedarften Benutzer eher gewöhnungsbedürftig ist.

4.1.8 RDF API Draft

Der *RDF API Draft* [RDFAPI] ist der Entwurf einer Programmierschnittstelle (*API*) zur Einbindung von RDF-Parsern in RDF-Applikationen.

4.1.9 Resümee: existierende Abfragesprachen

Die Analyse der existierenden Mittel zur Abfrage von RDF in Bezug auf die Eignung als Ad Hoc-Abfragesprache für Informationsportale lässt sich wie folgt zusammenfassen:

+ Zahlreiche Abfragesprachen verschiedenster Art stehen für die Abfrage von RDF-Daten zur Verfügung. Dazu zählen an SQL angelehnte Sprachen wie *RQL* oder *SquishQL*, logische Sprachen wie *METALOG*, und auch XML Abfragesprachen wie *XQuery* oder *XPath*. Diese Abfragesprachen sind kostenlos und erprobt, relativ performant, weit verbreitet und teilweise sogar standardisiert.

– Sämtliche untersuchten Abfragesprachen verwenden eine formale Notation vergleichbar mit SQL oder einer Programmiersprache. Ein technisch unbedarfter Anwender kann sich eine solche Sprache – wenn überhaupt – nur mit großem Lernaufwand aneignen.

– Die untersuchten Abfragesprachen bieten umfangreiche Möglichkeiten zur exakten Selektion und Projektion[30], und fordern dem Abfragesteller auch entsprechend präzise Angaben dazu ab. Diese Exaktheit ist für den Einsatz als Ad Hoc-Abfragesprache in einem Informationsportal unerwünscht, erhöht jedoch die Komplexität der Sprache unnötig.

4.2 Bedarf für eine neuartige RDF-Abfragesprache

Aufgrund der im vorherigen Abschnitt (4.1.9 Resüme) aufgeführten Nachteile der existierenden RDF-Abfragesprachen für den Einsatz in Informationsportalen, insbesondere der mangelnden Intuitivität, wird im Rahmen dieser Arbeit eine neue und neuartige RDF-Abfragesprache vorgestellt.

Obwohl bereits mehrere Abfragesprachen für RDF existieren, genügt keine von ihnen dem Anspruch, einfach und unkompliziert für Ad Hoc-Abfragen eingesetzt werden zu können, insbesondere nicht ohne besondere Vorkenntnisse seitens des Anwenders.

Die aufgeführten Vorteile, vor allem die relativ hohe Performance und Verfügbarkeit, sprechen jedoch dafür, bei der Entwicklung auf eine der bereits vorhandenen Abfragesprachen zurückzu-

[30] im Sinne der relationalen Datenbanklehre: Selektion des Datenbestandes auf die interessierenden Datensätze, und Projektion dieser Datensätze auf die interessierenden Felder.

greifen. Dazu werden die entsprechenden Abfragen zunächst in eine existierende Abfragesprache (die *Zwischensprache*) übersetzt, um anschließend von einem Prozessor für diese Abfragesprache verarbeitet zu werden.

4.2.1 Verwendung von RQL als Zwischensprache

Von den betrachteten Abfragesprachen bringt RQL die besten Voraussetzungen für die Grundlage einer Ad Hoc-Abfragesprache mit. Entscheidende Argumente für RQL sind:

+ RQL bietet eine leistungsfähige Kurzschreibweise (ohne SELECT-FROM-WHERE), welche der Idee einer Ad Hoc-Abfragesprache sehr nahe kommt.
+ RQL ist eine sehr mächtige Sprache. Somit ist die Wahrscheinlichkeit groß, dass sich auch bei Weiterentwicklung der Ad Hoc-Abfragesprache alle zukünftig möglichen Abfragen auf RQL abbilden lassen.
+ RQL ist zudem sehr gut für Schemaabfragen geeignet, siehe dazu auch 4.2.2 Verwendung von RQL für Schemaabfragen.

  RQL unterstützt eine vereinfachte Schreibweise für Reguläre Ausdrücke, welche an das Verhalten des Kommandozeileninterpreters unter MS-DOS erinnert. So sind z. B. die Symbole „*“ und „?“ umdefiniert zu der Bedeutung von „.*“ und „.“, wie man sie von PERL[31], Grep[32], etc. kennt.

Allerdings bringt die Verwendung von RQL auch Probleme mit sich:

– RQL ist eine sehr junge Sprache, die zudem nicht standardisiert ist. Vermutlich wird die Spezifikation von RQL noch Änderungen unterworfen sein, welche wiederum Änderungen an einer Implementierung für die zu entwickelnde Abfragesprache nach sich ziehen können.
– RQL ist eine sehr mächtige und komplexe Sprache. Die Implementierung eines RQL-Prozessors ist daher alles andere als trivial. Für die Implementierung der neuen Abfragesprache bedeutet das, eine tendenziell eher enge Auswahl an RQL-Prozessoren für die eigentliche Verarbeitung der Abfragen zur Verfügung zu haben.

4.2.2 Verwendung von RQL für Schemaabfragen

Die Kurzschreibweisen-Notation von RQL ist für das Navigieren durch RDF-Modelle ausreichend intuitiv, auch für unbedarfte Benutzer. Schemaabfragen sind durch RQL mit minimalen Kenntnissen über das Schema möglich, und für die Realisierung von Informationsportalen besonders nützlich.

[31] http://www.perl.com
[32] http://www.gnu.org/software/grep/grep.html

eRQL wird daher keine explizite Unterstützung für Schemaabfragen beinhalten, sondern diesen Aspekt der Suche vollständig RQL überlassen. Das bedeutet:

- Sucht der Benutzer eines Informationsportals nach gezielten Begriffen, wird er eine *eRQL*-Abfrage formulieren, welche zunächst in RQL transformiert wird, um dann von einem RQL-Prozessor ausgeführt zu werden.
- Möchte der Benutzer eines Informationsportals durch die Daten navigieren, kann er eine RQL-Abfrage (in Kurzschreibweisen-Notation) absetzen, welche dann direkt von dem darunter liegenden RQL-Prozessor verarbeitet wird.

Durch diese Tandem-Lösung ist gewährleistet, dass ein Benutzer des Informationsportals mittels RQL auch komplexe Abfragen formulieren und so die Beschränkungen von *eRQL* umgehen kann.

In einem Informationsportal könnten verschiedene Themen oder Teilbereiche durch verschiedene RDFS-Klassen modelliert, verfeinerte/generalisierte Themen durch verfeinerte/generalisierte Klassen abgebildet, und so eine browserartige Navigation von Themen zu Unterthemen realisiert werden.

In einem E-Commerce-Portal könnten Produktbereiche und Kategorien durch RDFS-Klassen modelliert werden, einzelne Produkte durch entsprechend klassifizierte RDF-Ressourcen.

5 RQL – eine RDF Abfragesprache

RQL ist eine funktionale und typsichere Abfragesprache für RDF-Daten, und wurde von Greg Karvounarakis[33] am *ICS-FORTH* [ICS-FORTH] in Griechenland entwickelt. RQL ist Bestandteil der *RDFSuite* [RDFSUITE], und steht unter der GPL-kompatiblen *RDFSuite license* [RQL-OVW]. Ein kurzer Überblick über RQL findet sich in 4.1.2 RQL.

RQL ist zu komplex, als dass es vom technisch unversierten Besucher eines Informationsportals unmittelbar verwendet werden könnte. Im Rahmen dieser Arbeit wird RQL jedoch verwendet, um eRQL-Abfragen zunächst in RQL-Abfragen zu überführen, bevor diese dann von einem RQL-Prozessor ausgewertet werden (siehe 4.2.1 Verwendung von RQL als Zwischensprache, 7 Umwandlung von eRQL- in RQL-Abfragen). Weiterhin wird RQL für Schemaabfragen direkt eingesetzt, siehe 4.2.2 Verwendung von RQL für Schemaabfragen.

In diesem Kapitel wird RQL vorgestellt und dessen Funktionsweise beschrieben. Für die Spezifikation von RQL sei auf [RQL-FUNC] verwiesen.

[33] mgregkar@ics.forth.gr

5.1 Szenario: Ein Kultur-Informationsportal

Für die Beispiele dieses Kapitels wird das folgende Szenario aus [RQL-FUNC] herangezogen:

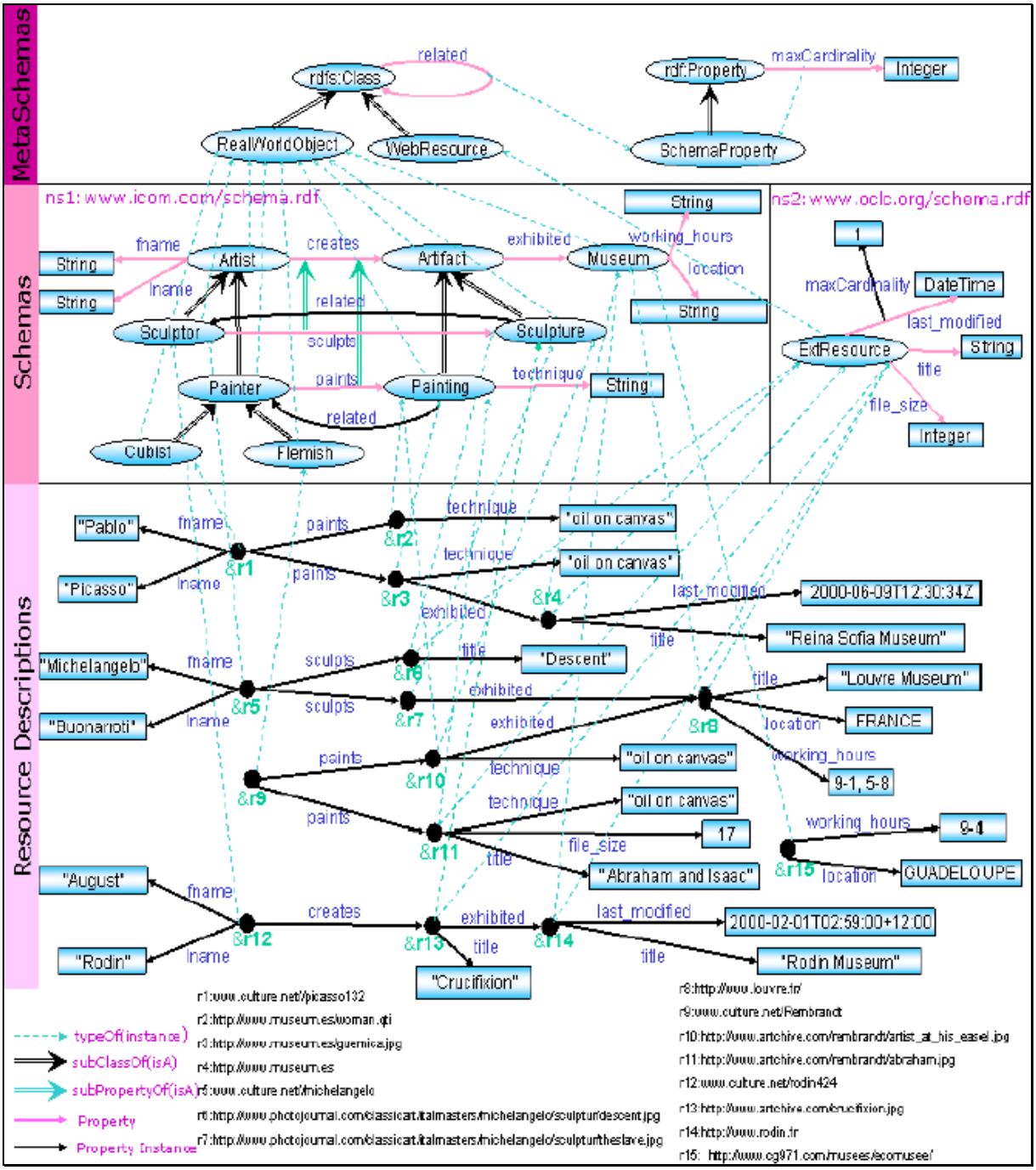

Abbildung 7 – Szenario: Ein Kultur-Informationsportal

Interessant ist zunächst der untere Bereich des Graphen. Hier sind Informationen über verschiedene Künstler wie Picasso oder Michelangelo zu erkennen sowie über die jeweils erzeugten Kunstwerke und die ausstellenden Museen.

Im mittleren Bereich des Graphen ist ein Klassifizierungsschema zu erkennen, welches die Ressourcen des unteren Bereiches in Klassen wie *Künstler* (`Artist`), *Kunstwerk* (`Artifact`) und *Museum* (`Museum`) kategorisiert. Weiterhin sind hier Beziehungen zu erkennen, welche die Spezia-

lisierungen von Klassen widerspiegeln, wie z. B. *Maler* (`Painter`) und *Bildhauer* (`Sculptor`) als Spezialisierung von *Künstler*, sowie *Kubist* (`Cubist`) und *flämisch*[34] (`Flemish`) wiederum als Spezialisierung von *Maler*.

Die drei vertikal angeordnete Schichten *Resource Descriptions*, *Schemas* und *MetaSchemas* werden in 5.3.1 Daten-, Schema- und Metaschemaebene erklärt.

Die Bezeichnungen *r1*, *r2*, *r3*, … dienen als symbolische Namen der verwendeten URIs. Sie werden im Rahmen dieser Arbeit verwendet, um die Schreibweise der URIs zu verkürzen. RQL unterstützt allerdings derzeit keine „abgekürzten" URIs in diesem Sinne.

5.2 Einführung in RQL

In der folgenden Auflistung werden die charakteristischen Eigenschaften von RQL überblicksartig beschrieben, um dem Leser einen grundlegenden Eindruck von der Mächtigkeit und Gestalt dieser Sprache zu vermitteln:

Allgemein

- Konzipiert für RDF, d. h. Unterstützung des RDF-Graphenmodells „von Hause aus"
- Syntax ähnelt der von SQL (`SELECT … FROM … WHERE`)
- Kurzschreibweise ohne SQL-Syntax möglich, z. B. `Artist` zum Ermitteln aller RDF-Ressourcen, welche der RDFS-Klasse `Artist` (oder einer ihrer Unterklassen wie `Painter` oder `Sculptor`) angehören
- Pfadausdrücke wie z. B. `{X}dc:subject{Z}.rdf:value{W}` ermöglichen die Navigation im RDF-Graphen
- Namensraum-Definitionen ermöglichen die Kurzformulierung von URIs (`dc:title` statt `http://www...#title`)

RDF Schema (RDFS)

- Die RDFS-Klassenhierarchie wird automatisch berücksichtigt. Sofern nicht anders angegeben, schließt eine Klasse von Ressourcen oder Prädikaten stets auch die Unterklassen oder Unterprädikate ein
- RDF-Daten und die zugehörigen RDFS-Schemainformationen können durch Abfrage-Konstrukte wie `{x;C}` einfach miteinander kombiniert werden

Operatoren

- Vergleichsoperatoren: =, <, >, …
- Vergleich mit einem Regulären Ausdruck: `LIKE`, z. B. `…WHERE Vorname LIKE "F*"`

[34] Es ist mir nicht gelungen, ein entsprechendes, deutschsprachiges Substantiv zu *flemish* zu finden.

- ^-Operator für die Deaktivierung der Klassen-Transitivität, z. B. `^Artist` zum Ermitteln aller RDF-Ressourcen, welche der RDFS-Klasse `Artist` angehören (und nicht einer ihrer Unterklassen wie `Painter` oder `Sculptor`)

Syntax

- URIs müssen durch den Präfix `&` gekennzeichnet werden
- Variablen für RDFS-Klassen müssen mit dem Präfix `$` gekennzeichnet sein, Variablen für Prädikate mit dem Präfix `@`, alle anderen Variablen dürfen kein Präfix besitzen

Funktionen

- Typfunktionen: **`superClassOf`**, **`superClassOf^`**, **`subClassOf`**, **`subClassOf^`**, **`subPropertyOf`**, **`subPropertyOf^`**, **`domain`**, **`range`**, **`typeof`**
- Mengenfunktionen: **`union`**, **`intersect`**, **`minus`**
- Konstruktoren: **`bag`**, **`seq`**
- Aggregierende Funktionen: **`min`**, **`max`**, **`avg`**, **`sum`**, **`count`**

Rückgabe

- Eine RQL-Abfrage gibt generell ein neues RDF-Modell zurück
- Das von einer RDF-Abfrage zurückgegebene Modell ist eine Sammlung von Folgen

RQL-Beispiele

- `dc:title` (kurz für: `SELECT X,Y FROM {X}dc:title{Y}`)
- `SELECT {Y} FROM {X}dc:title{Y},{X}dc:subject{Z}.rdf:value{W} WHERE W="Mathematik"`
- `Painter < Artist` (Entspricht `TRUE`, Identisch mit: `Painter IN subclassof(Artist)`)

5.2.1 Selektion und Projektion

RQL folgt dem von SQL bekannten Konzept der Selektion und Projektion, und verwendet dazu auch selbige Syntax:

```
SELECT projektion FROM … WHERE selektion
```

Dabei wird in den Anweisungen, welche `FROM` folgen, die Datenquelle festgelegt, und die entsprechenden Variablen gebunden. In der Projektion wird aus der Menge der Variablen, die gebunden worden sind, diejenige Teilmenge aufgezählt, welche für jede Fundstelle zurückgegeben werden soll. In der (optionalen) Selektion werden mittels der gebundenen Variablen und Vergleichsoperatoren ggf. eine oder mehrere Bedingungen definiert, welcher eine Fundstelle entsprechen muss, um in die Rückgabe übernommen zu werden.

Beispiel – Titel aller Kunstwerke im Rodin Museum

Abfrage:
```
SELECT Y FROM {X}title{Y},{X}exhibited{Z}.title{W} WHERE
W="Rodin Museum"
```

Y
Crucifixion

Erläuterung

Hierbei wird als erste Datenquelle die Verwendung des Prädikates `title` definiert und als zweite Datenquelle die aufeinander folgende Verwendung der Prädikate `exhibited` und `title` (in dieser Reihenfolge). Die Variablen `W`, `X`, `Y` und `Z` werden an die verschiedenen Ressourcen gebunden, die an den jeweiligen Fundstellen als Subjekt bzw. Objekt der Prädikate dienen. Die Bindung der Variablen `Z` ist dabei optional, da sie innerhalb dieser Abfrage nicht weiter benötigt wird.

Durch die "doppelte Bindung" von `X` (sowohl an das Subjekt einer Verwendung von `title`, als auch an das das Subjekt einer Verwendung von `exhibited`) wird erzwungen, dass `X` an all jene Ressourcen gebunden wird, welche sowohl das Prädikat `title`, als auch das Prädikat `exhibited` besitzen (was nur für Kunstwerke zutrifft).

Die Selektion bestimmt in diesem Fall, dass die Variable `W` (welche an das Objekt der Verwendung von `title`, d. h. den Titel des ausstellenden Museums gebunden ist) den Wert „Rodin Museum" annehmen muss, damit die Fundstelle in die Rückgabe übernommen wird. Die Projektion bestimmt, dass für jede Fundstelle lediglich der Wert der Variablen `Y` zurückgegeben wird, welche an das Subjekt der Verwendung von `title`, d. h. den Titel des Kunstwerkes gebunden ist.

5.2.2 RQL Vergleichsoperatoren

RQL stellt all jene Vergleichsoperatoren zur Verfügung, welche von anderen Abfragesprachen bereits bekannt sind. Im Einzelnen sind dies =, `!=`, <, >, <=, >=, `LIKE`.

Einige dieser Operatoren besitzen eine Doppelbedeutung. Dazu zählen beispielsweise die Operatoren <, >, <= und >=, welche je nach Kontext entweder die bekannte, arithmetische Bedeutung tragen, oder aber eine Aussage über die Klassenrelation treffen (RQL definiert, dass Klasse K_1 < Klasse K_2 genau dann wenn K_1 von K_2 abgeleitet ist). Für die exakte Semantik dieser Operatoren siehe [RQL-DOC].

Beim Vergleichen von Texten mittels „=" oder „LIKE" unterscheidet RQL zwischen Groß- und Kleinschreibung. Textuelle Vergleiche, die bezüglich der Groß-/ Kleinschreibung tolerant sind, sieht RQL derzeit nicht vor[35].

[35] Abgesehen von der Möglichkeit, die entsprechende Toleranz durch einen Regulären Ausdruck mit entsprechenden Zeichenklassen zu erzwingen.

5.3 RQL Datenmodell und Schemaoperationen

Schemaabfragen sind durch RQL mit minimalen Kenntnissen über das Schema möglich, und für die Realisierung von Informationsportalen besonders nützlich, siehe auch 4.2.2 Verwendung von RQL für Schemaabfragen. In diesem Abschnitt werden daher das von RQL verwendete Datenmodell sowie die durch RQL definierten Operationen darauf vorgestellt.

5.3.1 Daten-, Schema- und Metaschemaebene

Aus Sicht von RDF ist alles eine Ressource[36] und davon wiederum manches ein Prädikat. Bringt man jedoch RQL [RQL-FUNC] ins Spiel, werden aus den Ressourcen und Prädikaten plötzlich *Schemaklassen*, *Metaschemaklassen*, *Schemaprädikate* und vieles mehr.

Hintergrund ist, dass RQL jedes RDF-Modell logisch in drei Schichten aufteilt, und jede RDF-Ressource genau einer dieser drei Schichten zuweist:

1. **Metaschemaebene**

   Der *Metaschemaebene* sind alle Klassen und Prädikate zugeordnet, welche RDF Schema von Hause aus mitbringt: `rdfs:Class`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, etc.

   Die Klassen und Prädikate der Metaschemaebene werden *Metaschemaklassen* und *Metaschemaprädikate* genannt.

   Weiterhin sind der Metaschemaebene alle Ressourcen zugeordnet, welche mittels `rdfs:subClassOf` bzw. `rdfs:subPropertyOf` von einer dieser Klassen bzw. Prädikate abgeleitet sind. Dadurch wächst die Metaschemaebene um benutzerdefinierte Metaschemaklassen und Metaschemaprädikate.

2. **Schemaebene**

   Der *Schemaebene* sind alle Klassen und Prädikate zugeordnet, welche mittels `rdf:type` mit einer Ressource der Metaschemaebene verbunden sind, d. h. den Typ einer Metaschemaklasse oder eines Metaschemaprädikates besitzen.

   Die Klassen und Prädikate der Schemaebene werden *Schemaklassen* und *Schemaprädikate* genannt. In der Schemaebene sind beispielsweise alle benutzerdefinierten Schemaklassen angesiedelt, da diese mittels `rdf:type` mit der Metaschemaklasse `rdfs:Class` verbunden sind.

3. **Datenebene**

   Der *Datenebene* werden alle RDF-Ressourcen zugewiesen, welche nicht der Schemaebene oder der Metaschemaebene zugewiesen werden. Somit enthält die Datenebene diejenigen RDF-Ressourcen, welche die eigentlichen „Nutzdaten“ repräsentieren – frei von Schema- oder Metaschemadaten.

[36] Oder ein Literal. Literale können allerdings nicht als Prädikate verwendet werden.

Damit ist jede Ressource eindeutig einer der drei Ebenen zugeordnet.

5.3.2 RQL-Datentypen

Um eine Abfragesprache wie RQL zu definieren, welche nicht nur RDF, sondern auch RDF Schema berücksichtigt, ist es nötig, ein Datenmodell einzuführen, welches sowohl RDF als auch RDF Schema formal beschreibt. Hierbei handelt es sich um ein rein mathematisches Modell, welches der Definition von RQL dient.

Die folgende Auflistung beschreibt die RQL-Datentypen und stellt jeweils ein Symbol dafür vor:

Typen der Metaschemaebene

τ_{M_C}	Metaschemaklasse
τ_{M_P}	Metaschemaprädikat

Typen der Schemaebene

τ_C	Schemaklasse
$\tau_P[\tau,\tau]$	Schemaprädikat

Typen der Datenebene

τ_U	URI (Ressource der Datenebene)
τ_L	Literal

Container-Typen

$\{\tau\}$	Eine *Sammlung* mit beliebig vielen Ressourcen des Typ τ
$[1{:}\tau_1, 2{:}\tau_2, \ldots, n{:}\tau_n]$	Eine *Folge* von *n* Ressourcen, wobei die i-te Ressource vom Typ τ_i ist
$(1{:}\tau_1 + 2{:}\tau_2 + \ldots + n{:}\tau_n)$	Eine *Alternative* zwischen *n* Ressourcen, wobei die i-te Ressource vom Typ τ_i ist

Für jeden Container-Typ gilt, dass seine Instanzen beliebig viele Ressourcen beinhalten können. Ein Container muss jedoch keine Ressourcen beinhalten, sondern kann auch leer sein (d. h. *n=0*). Jede Komponente einer Aussage (Subjekt, Prädikat oder Objekt) besitzt aus der Sicht von RQL also exakt einen der genannten Datentypen. Die Menge aller Datentypen ist:

$$\begin{aligned}\tau := \quad & \tau_{M_C} \cup \tau_{M_P} \cup \tau_C \cup \tau_P[\tau,\tau] \cup \tau_U \cup \tau_L \cup \{\tau\} \cup \\ & [1{:}\tau_1, 2{:}\tau_2, \ldots, n{:}\tau_n] \cup (1{:}\tau_1 + 2{:}\tau_2 + \ldots + n{:}\tau_n)\end{aligned} \tag{5.1}$$

5.3.3 Schema-Operationen

In diesem Abschnitt werden die Operatoren erklärt, welche RQL für die Abfrage von Schemainformationen zur Verfügung stellt. Diese Operatoren sind umso wichtiger, als dass der Besucher

eines Informationsportals diese Operatoren verwenden muss, um durch die (Schema-) Daten zu navigieren („browsen“).

Die im Folgenden verwendete Notation $\frac{A}{B}$ ist zu lesen als „Wenn A eintritt bzw. gültig ist, dann tritt auch B ein bzw. ist auch B gültig“. Die Notation $e : \tau_C$ ist zu lesen als „e von der Art τ_C“.

Ist im Folgenden von einer „Menge von Klassen“, „Menge von Metaschemaklassen“ oder einer anderen Menge die Rede, so ist damit auch immer der Fall der leeren Menge beinhaltet, also z. B. „gar keine Klasse“ bzw. „gar keine Metaschemaklasse“.

Die Verwendung des ^-Operators hinter einem Funktionsnamen, z. B. **`subClassOf^`** anstelle von **`subClassOf`**, verändert zwar den Umfang der Rückgabe (d. h. die Anzahl der zurückgegebenen Ressourcen), nicht aber den Typ der Rückgabe. Daher wird im Folgenden auf den ^-Operator nicht gesondert eingegangen. Die Informationen dieses Abschnittes wurden [RQL-FUNC] entnommen.

5.3.3.1 subClassOf() und superClassOf()

Die Anwendung von **`subClassOf`** auf eine Schemaklasse gibt eine Menge von Schemaklassen zurück (5.2). Wird **`subClassOf`** hingegen auf eine Metaschemaklasse angewendet, wird entsprechend eine Menge von Metaschemaklassen zurückgegeben (5.3).

Als Gegenstück zu **`subClassOf`** gibt auch **`superClassOf`** bei Anwendung auf eine Schemaklasse eine Menge von Schemaklassen (5.6), und bei Anwendung auf eine Metaschemaklasse eine Menge von Metaschemaklassen (5.7) zurück.

Optional können beide Funktionen mit einem zweiten, numerischen Parameter aufgerufen werden, welcher die Tiefe einschränkt, bis zu der nach Klassen gesucht wird. Dieser Parameter beeinflusst lediglich die Anzahl, nicht aber den Typ der zurückgegebenen Ressourcen, siehe (5.4), (5.5), (5.8) und (5.9).

Beispiele

$\texttt{subClassOf}\,(\text{Artist}:\tau_C) = \{\text{Sculptor}:\tau_C, \text{Painter}:\tau_C, \text{Cubist}:\tau_C, \text{Flemish}:\tau_C\}$

$\texttt{subClassOf\^{}}\,(\text{Artist}:\tau_C) = \texttt{subClassOf}\,(\text{Artist}:\tau_C, 1) = \{\text{Sculptor}:\tau_C, \text{Painter}:\tau_C\}$

$\texttt{subClassOf}\,(\text{rdfs:Class}:\tau_M) = \{\text{RealWorldObject}:\tau_M, \text{WebResource}:\tau_M\}$

$\texttt{superClassOf}\,(\text{Artist}:\tau_C) = \{\text{Ressource}:\tau_C\}$

$\texttt{superClassOf}\,(\text{Ressource}:\tau_C) = \{\ \}$

$\texttt{superClassOf}\,(\text{rdfs:Class}:\tau_M) = \{\ \}$

Regeln

$$\frac{e : \tau_C}{\texttt{subClassOf}\,(e) : \{\tau_C\}} \tag{5.2}$$

$$\frac{e : \tau_M}{\texttt{subClassOf}(e) : \{\tau_M\}} \tag{5.3}$$

$$\frac{e : \tau_C,\ n : \texttt{integer}}{\texttt{subClassOf}(e, n) : \{\tau_C\}} \tag{5.4}$$

$$\frac{e : \tau_M,\ n : \texttt{integer}}{\texttt{subClassOf}(e, n) : \{\tau_M\}} \tag{5.5}$$

$$\frac{e : \tau_C}{\texttt{superClassOf}(e) : \{\tau_C\}} \tag{5.6}$$

$$\frac{e : \tau_M}{\texttt{superClassOf}(e) : \{\tau_M\}} \tag{5.7}$$

$$\frac{e : \tau_C,\ n : \texttt{integer}}{\texttt{superClassOf}(e, n) : \{\tau_C\}} \tag{5.8}$$

$$\frac{e : \tau_M,\ n : \texttt{integer}}{\texttt{superClassOf}(e, n) : \{\tau_M\}} \tag{5.9}$$

5.3.3.2 subPropertyOf() und superPropertyOf()

Während **`subClassOf`** und **`superClassOf`** für die Anwendung auf Klassen hilfreich sind, werden **`subPropertyOf`** und **`superPropertyOf`** auf Prädikate angewendet, und geben entsprechend eine Menge von Prädikaten zurück, siehe (5.10) und (5.12).

Auch **`subPropertyOf`** und **`superPropertyOf`** können optional mit einem zweiten, numerischen Parameter aufgerufen werden, welcher die Tiefe einschränkt, bis zu der nach Prädikaten gesucht wird. Auch dieser Parameter beeinflusst lediglich den Umfang, aber nicht den Typ der zurückgegebenen Ressourcen, siehe (5.11) und (5.13).

Beispiele

$$\texttt{subPropertyOf}(\text{creates}:\tau_P\ [\text{Artist, Artifact}]) =$$
$$\{\text{sculpts}:\tau_P\ [\text{Sculptor, Sculpture}],\ \text{paints}:\tau_P\ [\text{Painter, Painting}]\}$$

$$\texttt{superPropertyOf}(\text{creates}:\tau_P\ [\text{Artist, Artifact}]) = \{\ \}$$

Regeln

$$\frac{e : \tau_P\ [\tau_1, \tau_2]}{\texttt{subPropertyOf}(e) : \tau_P\ [\tau_1, \tau_2]} \tag{5.10}$$

$$\frac{e : \tau_P\ [\tau_1, \tau_2],\ n : \texttt{integer}}{\texttt{subPropertyOf}(e, n) : \tau_P\ [\tau_1, \tau_2]} \tag{5.11}$$

$$\frac{e \texttt{ : } \tau_P \left[\tau_1, \tau_2\right]}{\texttt{superPropertyOf}\left(e\right) \texttt{ : } \tau_P \left[\tau_1, \tau_2\right]} \tag{5.12}$$

$$\frac{e \texttt{ : } \tau_P \left[\tau_1, \tau_2\right], n \texttt{ : integer}}{\texttt{superPropertyOf}\left(e, n\right) \texttt{ : } \tau_P \left[\tau_1, \tau_2\right]} \tag{5.13}$$

5.3.3.3 typeof()

Die Funktion **typeof** ermittelt zu einer gegebenen URI, Klasse oder einem Prädikat den zugehörigen Typ. Bei dem Typ handelt es sich um eine Schemaklasse, wenn **typeof** auf eine Ressource aus der Datenebene angewendet wird, und um eine Metaschemaklasse, wenn **typeof** auf eine Ressource aus der Schema- oder Metaschemaebene angewendet wird.

Regel

$$\frac{e{:}\tau, \left(\tau{=}\tau_U \mid \tau{=}\tau_C \mid \tau{=}\tau_P\left[\tau_1, \tau_2\right]\right)}{\text{typeof}\left(e\right){:}\left(\left\{\tau_C \mid \tau_M\right\}\right)} \tag{5.14}$$

5.3.3.4 domain()

domain ermittelt für ein bestimmtes Prädikat die Menge der Klassen, auf welche dieses Prädikat angewendet werden kann. Genau genommen wird die Menge der Klassen ermittelt, deren Instanzen als Subjekt für das Prädikat dienen dürfen.

Regel

$$\frac{e{:}\tau_P\left[\tau_1, \tau_2\right], \left(\tau_1{=}\tau_C \mid \tau_1{=}\tau_M\right)}{\text{domain}\left(e\right){:}\tau_1} \tag{5.15}$$

5.3.3.5 range()

range als Gegenstück zu **domain** ermittelt für ein bestimmtes Prädikat die Menge der Klassen, deren Instanzen als Objekt für das Prädikat dienen dürfen.

Da die Objekte von RDF-Aussagen im Gegensatz zu den Subjekten nicht zwingend Ressourcen sein müssen, sondern auch Literale sein können, gibt die Funktion **range** im Gegensatz zu **domain** unter Umständen auch eine literale Klasse zurück, siehe (5.16).

Regel

$$\frac{e{:}\tau_P\left[\tau_1, \tau_2\right], \left(\tau_2{=}\tau_C \mid \tau_2{=}\tau_M \mid \tau_2{=}\tau_L\right)}{\text{range}\left(e\right){:}\tau_2} \tag{5.16}$$

5.3.3.6 namespace()

`namespace` ermittelt für eine bestimmte Ressource die URI des Namensraumes, in welcher diese Ressource definiert ist.

Beispiel

$$\text{namespace}(\text{Artist}:\tau_C) = \texttt{http://www.icom.com/schema.rdf}$$

Regeln

$$\frac{e:\tau, \tau \in \{\tau_C, \tau_M, \tau_P[\tau_1, \tau_2], \tau_L\}}{\text{namespace}(e):\{\tau_U\}} \tag{5.17}$$

5.3.3.7 topclass und leafclass

`topclass` ermittelt alle Schemaklassen, welche nicht von einer anderen Schemaklasse abgeleitet sind. **`topclass`** (und **`leafclass`**) liefern keine Klassen des Metaschemas zurück.

In einer gewöhnlichen Vererbungshierarchie, wie man sie z. B. aus Programmiersprachen (C++, JAVA, Delphi, …) kennt, muss jede Klasse von einer anderen Klasse abgeleitet sein, was dazu führt, dass stets eine einzige, bekannte Wurzelklasse des Ableitungsbaumes existiert. RDF Schema verlangt nicht, dass jede Klasse von einer anderen Klasse abgeleitet ist, ergo existiert auch keine Wurzelklasse des Ableitungsbaumes. Statt dessen existieren „mehrere Wurzelklassen“, nämlich all jene Klassen, welche nicht von einer anderen Klasse abgeleitet sind. Diese Klassen lassen sich durch **`topclass`** ermitteln. Im Beispiel entspricht **`topclass`** den Klassen `ExtResource`, `Artist`, `Artifact` und `Museum` (nicht etwa `Resource`, da diese Klasse nicht dem Schema sondern dem Metaschema entstammt).

Das Gegenstück zu **`topclass`** trägt den Namen **`leafclass`** und entspricht allen Schemaklassen, von welchen keine weiteren Schemaklassen abgeleitet sind. Im Beispiel entspricht **`leafclass`** den Klassen `ExtResource`, `Sculptor`, `Sculpture`, `Painting`, `Cubist`, `Flemish` und `Museum`. Anmerkung: **`topclass`** und **`leafclass`** werden – anders als andere Funktionen – ohne öffnende und schließende Klammer verwendet.

Beispiel – oberste Klassen finden

Abfrage: `topclass`

topclass
ExtResource
Artist
Artifact
Museum

Beispiel – unterste Klassen finden

Abfrage: `leafclass`

leafclass
ExtResource
Sculptor
Sculpture
Painting
Cubist
Flemish
Museum

Regeln

$$\frac{e = \text{topclass}}{e : \{\tau_C\}} \tag{5.18}$$

$$\frac{e = \text{leafclass}}{e : \{\tau_C\}} \tag{5.19}$$

5.3.3.8 topproperty und leafproperty

`topproperty` ermittelt alle Prädikate, welche nicht von anderen Prädikaten abgeleitet sind. Das Gegenstück zu **`topproperty`** trägt den Namen **`leafproperty`** und entspricht allen Prädikaten, von welchen keine weiteren Prädikate abgeleitet sind.

Anmerkung: **`topproperty`** und **`leafproperty`** werden – anders als andere Funktionen von RQL – ohne öffnende und schließende Klammer verwendet.

Beispiel – oberste Prädikate finden

Abfrage: `topproperty`

topproperty
maxCardinality
related
last_modified
title
file_size
last_name
location
exhibited
first_name
technique
working_hours
creates

Beispiel – unterste Prädikate finden

Abfrage: `leafproperty`

leafproperty
maxCardinality
related
last_modified
title
file_size
last_name
location
exhibited
first_name
technique
working_hours
sculpts
paints

Anmerkungen

Im Szenario existieren nur zwei Prädikate, welche von einem anderen Prädikat abgeleitet sind: `sculpts` und `paints` – abgeleitet von `creates`. Daher unterscheiden sich die Rückgaben von **`topproperty`** und **`leafproperty`** lediglich in diesen drei Prädikaten.

Regel

$$\frac{e \in \{\text{topproperty}, \text{leafproperty}\}}{e : \{\tau_P[\tau_1, \tau_2]\}} \tag{5.20}$$

5.4 RQL Pfadausdrücke

So genannte *Pfadausdrücke* bilden den Kern von RQL Abfragen. Sie beschreiben eine Beziehung zwischen Ressourcen, Prädikaten und Schemainformationen, und nehmen eine Bindung dieser Komponenten an Variablen vor. Diese Variablen wiederum können in der Selektion und Projektion von RQL Abfragen verwendet werden.

Im Folgenden werden alle Pfadausdrücke aufgeführt und erläutert, welche RQL zur Verfügung stellt, gruppiert nach Verwendungszwecken. Die entsprechende Spezifikation findet sich in [RQL-FUNC], dort allerdings gruppiert nach der Ebene des RQL Datenmodells auf welches sich die einzelnen Pfadausdrücke beziehen.

5.4.1 Instanzen einer Klasse

Alle Instanzen einer Klasse lassen sich durch `SELECT X FROM KLASSE{X}` finden. Dabei bezeichnet `KLASSE` den Namen der Schemaklasse oder Metaschemaklasse, deren Instanzen gesucht sind, und `X` die Variable, an welche die gefundenen Instanzen gebunden werden. Zu beachten ist, dass nicht nur Instanzen der Klasse `KLASSE`, sondern auch Instanzen von Klassen, wel-

che von KLASSE abgeleitet sind, gefunden werden. Diese Abfrage kann verkürzt als KLASSE formuliert werden.

Ein RDF-Modell wird reduziert auf alle Ressourcen, die mittels rdf:type mit der Ressource KLASSE verbunden sind, oder mit einer von KLASSE via rdfs:subClassOf abgeleiteten Ressource. Voraussetzung ist, dass es sich bei KLASSE um eine Klasse handelt.

Ausschließen abgeleiteter Klassen

Die Miteinbeziehung abgeleiteter Klassen lässt sich durch den ^-Operator unterbinden:

```
SELECT X FROM ^KLASSE{X}
```

bzw. ^KLASSE in Kurzschreibweise.

Zusätzliche Klasseninformationen zurückgeben

Falls zusätzlich die zu den Instanzen gehörenden Klassen benötigt werden, lässt sich der Befehl um eine Klassenvariable erweitern, welche jeweils an die Klasse der entsprechenden Instanz gebunden wird:

```
SELECT X,$Y FROM KLASSE{X;$Y}
```

Nach wie vor kann auch der ^-Operator angewendet werden.

Instanzen aller Klassen zurückgeben

Falls der Name der Klasse nicht bekannt ist, oder aus einem anderen Grund Instanzen aller gefundenen Klassen zurückgegeben werden sollen, kann auf die Angabe des Klassennamens verzichtet werden, und statt dessen eine Klassenvariable verwendet werden:

```
SELECT X,$Y FROM {X}$Y
```

(Anmerkung: Zu dem selben Ergebnis führt auch `SELECT X,$Y FROM $Y{X}`).

Da eine RDF-Ressource beliebig oft klassifiziert sein kann, finden sich Ressourcen durchaus mehrmals in der Rückgabe – je einmal pro Klasse. Es ist zu beachten, dass Metaschemaklassen nicht zurückgegeben werden.

Beispiel 1 – Finden der URIs aller Künstler

Abfrage: `SELECT X FROM Artist{X}`

oder: `SELECT X FROM Artist{X;$Y}`

oder: `Artist`

X	
&r5	(Michelangelo)
&r12	(Rodin)
&r1	(Picasso)
&r9	(Rembrandt)

Beispiel 2 – Finden der URIs und Klassen aller Künstler

Abfrage: `SELECT X,$Y FROM Artist{X;$Y}`

X	$Y
&r5 (Michelangelo)	Sculptor
&r12 (Rodin)	Sculptor
&r1 (Picasso)	Cubist
&r9 (Rembrandt)	Flemish

Beispiel 3 – Finden aller Schemaklassen

Abfrage: `SELECT X FROM Class{X}`

oder: `SELECT $X FROM $X`

oder: `Class`

X
Resource
ExtResource
Artist
Sculptor
Artifact
Sculpture
Painting
Museum
Painter
Cubist
Flemish

Beispiel 4 – Finden aller Arten von Prädikaten

Abfrage: `SELECT X FROM Property{X}`

oder: `SELECT @X FROM @X`

oder: `Property`

X
maxCardinality
related
last_modified
title
file_size
last_name
creates
paints
exhibited
sculpts
first_name
technique
working_hours
location

Beispiel 5 – Finden aller Klassifizierungen von www.museum.es

Abfrage: `SELECT X,$KLASSE FROM {X}$KLASSE WHERE X=&http://www.museum.es`

X	$KLASSE
&r4 (http://www.museum.es)	ExtResource
-"-	Museum

5.4.2 Ableitungen einer bestimmten Klasse

Um alle abgeleiteten Klassen der Schemaklasse oder Metaschemaklasse KLASSE zu erhalten, genügt ein `SELECT $X FROM KLASSE{$X}`. Dabei bezeichnet `KLASSE` den Namen der gewünschten Klasse, und `$X` eine Variable.

Ein RDF-Modell wird reduziert auf alle Ressourcen, die direkt oder indirekt mittels `rdf:subClassOf` mit der Ressource `KLASSE` verbunden sind, *und* zugleich mittels `rdf:type` mit `rdfs:Class` oder einer davon abgeleiteten Klasse verbunden sind. Voraussetzung ist, dass es sich bei `KLASSE` um eine Klasse handelt.

Beispiel 1 – Finden aller Arten von Kunstwerken

Abfrage: `SELECT $KUNSTWERKARTEN FROM Artifact{$KUNSTWERKARTEN}`

$KUNSTWERKARTEN
Artifact
Sculpture
Painting

5.4.3 Verwendungen eines bestimmten Prädikates

Der folgende Datenpfad ermöglicht es, alle Verwendungen eines bestimmten Prädikates zu finden, dessen Name bekannt ist:

```
SELECT S,O FROM {S}PROPERTY{O}
```

Für jede Fundstelle des Prädikates `PROPERTY` (oder eines davon abgeleiteten Prädikates) werden das jeweils zugehörige Subjekt und Objekt an die Variablen `S` und `O` gebunden. Die obige Abfrage kann verkürzt als `PROPERTY` formuliert werden.

Ausschließen abgeleiteter Prädikate

Die Miteinbeziehung abgeleiteter Prädikate lässt sich durch den ^-Operator unterbinden:

```
SELECT S,O FROM {S}^PROPERTY{O}
```

bzw. `^PROPERTY` in Kurzschreibweise.

Zusätzliche Klasseninformationen zurückgeben

Möchte man zusätzlich zu den gefundenen Subjekten und Objekten auch die jeweiligen Klassen des Subjektes und Objektes erfahren, lässt sich die Abfrage durch zwei zusätzliche Variablen `$SK` und `$OK` für die Subjektklasse und die Objektklasse erweitern zu:

```
SELECT S,$SK,O,$OK FROM {S;$SK}PROPERTY{O;$OK}
```

Nach wie vor kann auch der ^-Operator angewendet werden.

Verzicht auf nicht benötigte Variablen

Falls nur die Subjekte oder nur die Objekte der Prädikat- Fundstelle von belang sind, kann auf die entsprechende(n) Variable(n) in der Abfrage verzichtet werden:

```
SELECT S FROM {S}PROPERTY
SELECT O,$OK FROM PROPERTY{O;$OK}
```

Falls nur für die Subjekte (oder nur für die Objekte) die zusätzlichen Klasseninformationen benötigt werden, und für die Objekte (bzw. für die Subjekte) die Klasseninformationen nicht benötigt werden, so kann auf die entsprechende Klassenvariable verzichtet werden:

```
SELECT S,O,$OK FROM {S}PROPERTY{O;$OK}
SELECT S,$SK,O FROM {S;$SK}PROPERTY{O}
```

Entsprechendes gilt, falls für die gefundenen Subjekte (oder für die gefundenen Objekte) ausschließlich die Klasseninformationen von Interesse sind:

```
SELECT $SK,O,$OK FROM {$SK}PROPERTY{O;$OK}
SELECT S,$SK,$OK FROM {S;$SK}PROPERTY{$OK}
```

Alle obigen Varianten können kombiniert werden:

```
SELECT $SK,O FROM {$SK}PROPERTY{O }
```

Beispiel 1 – Finden aller „Herstellungsvorgänge“

Abfrage: `SELECT KUENSTLER,KUNSTWERK FROM {KUENSTLER}creates{KUNSTWERK}`

oder: `creates`

KUENSTLER	KUNSTWERK
&r12 (Rodin)	&r13 (Crucifixion)
&r1 (Picasso)	&r2 (Woman)
-"-	&r3 (Guernica)
&r9 (Rembrandt)	&r10 (Portrait of the Artist at His Easel)
-"-	&r11 (Abraham and Isaac)
&r5 (Michelangelo)	&r6 (Descent)
-"-	&r7 (The Slave)

Beispiel 2 – Finden aller „reinen Herstellungsvorgänge“

Abfrage: `SELECT KUENSTLER,KUNSTWERK FROM {KUENSTLER}^creates{KUNSTWERK}`

oder: `^creates`

KUENSTLER	KUNSTWERK
&r12 (Rodin)	&r13 (Crucifixion)

Beispiel 3 – Finden aller „Malvorgänge“

Abfrage: `SELECT X,Y FROM {X}paints{Y}`

oder: `paints`

X	Y
&r1 (Picasso)	&r2 (Woman)
-"-	&r3 (Guernica)
&r9 (Rembrandt)	&r10 (Portrait of the Artist at His Easel)
-"-	&r11 (Abraham and Isaac)

Beispiel 4 – Finden aller „Malvorgänge“ mit jeweiligen Klassen

Abfrage: `SELECT MALER,GEMAELDE FROM {MALER;$K1}paints{GEMAELDE;$K2}`

oder: `paints`

MALER	$K1	GEMAELDE	$K2
&r1 (Picasso)	Cubist	&r2 (Woman)	Painting
-"-	-"-	&r3 (Guernica)	-"-
&r9 (Rembrandt)	Flemish	&r10 (Artist at His Easel)	-"-
-"-	-"-	&r11 (Abraham and Isaac)	-"-

5.4.4 Definitions- und Wertebereich eines Prädikates

Definitions- und Wertebereich des Prädikates `PROPERTY` liefert ein `SELECT $X,$Y FROM {$X}PROPERTY{$Y}`.

Dabei steht `PROPERTY` für den Namen des entsprechenden Prädikates, und stehen `$X` und `$Y` für zwei Variablen, welche mit allen Kombinationen von Klassen gefüllt werden, deren Instanzen mittels des Prädikates `Property` verbunden werden dürfen.

Beispiel – „Malende“ Klassen, und „gemalt werdende“ Klassen

Abfrage: `SELECT $X,$Y FROM {$X}paints{$Y}`

$X	$Y
Painter	Painting
Cubist	-"-
Flemish	-"-

5.4.5 Vollständige Aussagen

Die bislang vorgestellten Datenpfade ermitteln Informationen über ein RDF-Modell, indem von einem bestimmtem Prädikat oder einer bestimmten Klasse ausgehend nach benachbarten Informationen gefragt wird (d. h. nach jeweiligen Instanzen, abgeleiteten Klassen, etc.).

Mit dem folgenden Datenpfad lassen sich Aussagen eines RDF-Modells „vollständig“ erfragen, d. h. als Tripel mit Subjekt, Prädikat und Objekt zurückgeben:

```
SELECT S,@P,O FROM {S}@P{O}
```

Bei `S`, `@P` und `O` handelt es sich um drei Variablen, welche für jedes gefundene Tripel das Subjekt, das Prädikat und das Objekt aufnehmen.

Anwendung auf der Schemaebene

Obige Konstruktion gibt Aussagen auf der Datenebene eines RDF-Modells zurück. Falls Aussagen auf der Schemaebene gefragt sind, müssen die Variablen `S` und `O` zu diesem Zweck durch Klassenvariablen ersetzt werden:

```
SELECT $SK,@P,$OK FROM {$SK}@P{$OK}
```

Anwendung auf der Daten- und Schemaebene

Die kombinierte Anwendung auf Daten- und Schemaebene erlaubt es, alle Aussagen auf der Datenebene zu finden, und um ihre jeweiligen Klassen anzureichern:

```
SELECT $S,$SK,@P,$O,$OK FROM {S;$SK}@P{O;$OK}
```

Hierbei ist zu beachten, dass literale Subjekte und Objekte nicht mit einer Schemaklasse verbunden sind, und daher dazu führen, dass die entsprechende Verwendung dieses Prädikates nicht zurückgegeben wird (siehe folgendes Beispiel).

Beispiel 1 – Alle Aussagen, die von „Pablo Picasso“ ausgehen

Abfrage:
```
SELECT S,@P,O FROM {S}@P{O} WHERE
S=&http://www.culture.net/picasso132
```

S	@P	O
http://www.culture.net/picasso132	last_name	Picasso
-"-	paints	&r2
-"-	-"-	&r3
-"-	first_name	Pablo

Beispiel 2 – Alle Aussagen, die von „Pablo Picasso“ ausgehen – samt Klassen

Abfrage:
```
SELECT S,$SK,@P,O,$OK FROM {S;$SK}@P{O;$OK} WHERE
S=&http://www.culture.net/picasso132
```

S	$SK	@P	O	$OK
http://www.culture.net/picasso132	Cubist	paints	&r2	Painting
"	-"-	-"-	&r3	-"-

Beispiel 3 – Alle die „Erzeugung“ betreffenden Aussagen

Abfrage:
```
SELECT X,@P,Y FROM {X}@P{Y} WHERE @P=creates
```

X	@P	Y
&r12 (Rodin)	creates	&r13 (Crufixion)

Beispiel 4 – Alle zulässigen Kombinationen von „Erzeuger“ und „Erzeugtem“

Abfrage: `SELECT $X,@P,$Y FROM {$X}@P{$Y} WHERE @P=creates`

$X	@P	$Y
Artist	creates	Artifact
-"-	-"-	Sculpture
-"-	-"-	Painting
Sculptor	-"-	Artifact
-"-	-"-	Sculpture
-"-	-"-	Painting
Painter	-"-	Artifact
-"-	-"-	Sculpture
-"-	-"-	Painting
Cubist	-"-	Artifact
-"-	-"-	Sculpture
-"-	-"-	Painting
Flemish	-"-	Artifact
-"-	-"-	Sculpture
-"-	-"-	Painting

5.4.6 Schemaklassen und abgeleitete Schemaklassen

Alle Schemaklassen sowie die jeweiligen abgeleiteten Schemaklassen lassen sich durch `SELECT $X,$Y FROM $X{$Y}` aufspüren. Es ist zu beachten, dass Metaschemaklassen nicht zurückgegeben werden. Bei `$X` und `$Y` handelt es sich um zwei Variablen, welche jeweils so an zwei gefundene Schemaklassen gebunden sind, dass die Klasse `$Y` eine Ableitung der Klasse `$X` ist.

Beispiel – Alle spezialisierten Künstler finden

Abfrage:
```
SELECT $KLASSE,$ABLEITUNG FROM $KLASSE{$ABLEITUNG} WHERE
$KLASSE=Artist
```

$KLASSE	$ABLEITUNG
Artist	Artist
-"-	Sculptor
-"-	Painter
-"-	Cubist
-"-	Flemish

5.4.7 Zusammengesetzte Pfadausdrücke

Alle bislang vorgestellten Pfadausdrücke haben gemeinsam, dass sie sich jeweils auf genau eine Aussage beziehen. Ist im Beispiel die Menge aller Kunstwerke gesucht, welche im Louvre ausgestellt sind, so lässt sich das mit den bisher vorgestellten Mitteln erreichen:

Beispiel 1 – „Alle Kunstwerke im Louvre“ (URI bekannt)

Abfrage: `SELECT X FROM {X}exhibited{Y} WHERE Y=&http://www.louvre.fr/`

X
http://www.photojournal.com/classicart/italmasters/

x
michelangelo/sculptur/theslave.jpg
http://www.artchive.com/rembrandt/artist_at_his_easel.jpg

Ist hingegen die exakte URI der Ressource „Louvre“ nicht bekannt, und soll stattdessen nach dem Titel gesucht werden, so geht dies mit den bisher vorgestellten Pfadausdrücken nicht. Statt dessen müssen zwei Pfadausdrücke durch einen Punkt („.“) miteinander wie folgt verbunden werden:

Beispiel 2 – „Alle Kunstwerke im Louvre“ (URI nicht bekannt)

Abfrage:
```
SELECT X FROM {X}exhibited{Y}.title{Z} WHERE Z LIKE
"*Louvre*"
```

x
http://www.photojournal.com/classicart/italmasters/ michelangelo/sculptur/theslave.jpg
http://www.artchive.com/rembrandt/artist_at_his_easel.jpg

Durch zusammengesetzte Pfadausdrücke lassen sich Ketten von Aussagen beschreiben. Bemerkenswert ist, dass diese Aussagenketten stets nur vollständig gefunden werden. Bei obigem Beispiel führt dieses Verhalten dazu, dass Museen, welche keinen Titel besitzen, überhaupt nicht erst gefunden werden.

5.5 Kurz und bündig – die RQL Kurzschreibweise

Schemaabfragen sind durch RQL mit minimalen Kenntnissen über das Schema möglich, und für den Einsatz in Informationsportalen besonders nützlich, siehe 4.2.2 Verwendung von RQL für Schemaabfragen.

Die RQL Kurzschreibweise kommt dem Einsatz als Ad Hoc-Abfragesprache entgegen, wie die folgenden Beispiele zeigen. Eine detaillierte Beschreibung der abgekürzten Schreibweise kann [RQL-FUNC, 1.3.1 Basic Querying Functionality] entnommen werden, zwei der Einsatzzwecke – Instanzen einer Klasse finden, sowie Verwendungen eines Prädikates finden – werden im Folgenden beschrieben.

5.5.1 Instanzen einer Klasse finden

Um zu einer gegebenen Klasse alle Instanzen zu ermitteln, genügt es, den Namen der jeweiligen Klasse anzugeben, z. B. `Painter`. Bei einer solchen Abfrage bezieht RQL automatisch die Vererbungshierarchie der Klassen ein und liefert die Instanzen der abgeleiteten Klassen ebenfalls zurück. Falls dieses Verhalten unerwünscht ist, kann der ^-Operator eingesetzt werden. Siehe dazu auch 5.4.1 Instanzen einer Klasse.

Beispiel 1 – Finden der URIs aller „Maler“

Abfrage: `Painter`

oder: `SELECT X FROM Painter{X}`

X
&r1 (Picasso)
&r9 (Rembrandt)

Beispiel 2 – Finden der URIs aller Maler – ohne Spezialisierungen

Abfrage: `^Painter`

oder: `SELECT X FROM ^Painter{X}`

Ergebnis

Leer, da keine direkten Instanzen der Klasse `Painter` existieren, sondern lediglich Instanzen der davon abgeleiteten Klassen `Cubist` und `Flemish`..

5.5.2 Verwendungen eines Prädikates finden

Um zu einem gegebenen Prädikat alle Verwendungen zu ermitteln, genügt es, den Namen des jeweiligen Prädikates anzugeben, z. B. `creates`. Bei einer solchen Abfrage bezieht RQL automatisch die Vererbungshierarchie der Klassen ein und liefert die Verwendungen der abgeleiteten Prädikate ebenfalls zurück. Falls dieses Verhalten unerwünscht ist, kann der ^-Operator eingesetzt werden. Siehe dazu auch 5.4.3 Verwendungen eines bestimmten Prädikates.

Beispiel 1 – Finden aller „Herstellungsvorgänge“

Abfrage: `creates`

oder: `SELECT KUENSTLER,KUNSTWERK FROM {KUENSTLER}creates{KUNSTWERK}`

KUENSTLER	KUNSTWERK
&r12 (Rodin)	&r13 (Crucifixion)
&r1 (Picasso)	&r2 (Woman)
-"-	&r3 (Guernica)
&r9 (Rembrandt)	&r10 (Portrait of the Artist at His Easel)
-"-	&r11 (Abraham and Isaac)
&r5 (Michelangelo)	&r6 (Descent)
-"-	&r7 (The Slave)

Beispiel 2 – Finden aller Herstellungsvorgänge – ohne Spezialisierungen

Abfrage: `^creates`

oder:
```
SELECT KUENSTLER,KUNSTWERK FROM
{KUENSTLER}^creates{KUNSTWERK}
```

KUENSTLER	KUNSTWERK
&r12 (Rodin)	&r13 (Crucifixion)

5.6 Rückgabe

RQL liefert stets ein RDF-Modell zurück, insofern ist RQL also eine Abbildung von RDF-Modellen nach RDF-Modellen.

Sofern die Selektion einer RQL-Abfrage mehr als eine Fundstelle selektiert, ist die Rückgabe von RQL stets eine *Sammlung* von Ressourcen. Jede einzelne dieser Ressourcen ist wiederum eine *Folge* von Ressourcen, sofern die Projektion der Abfrage mehr als eine Komponente enthält (Sammlungen oder Folgen, die nur eine oder gar keine Ressource beinhalten, werden nicht zurückgegeben – in diesem Fall wird entweder die leere Menge, oder die Ressource an sich zurückgegeben).

Die Implementierung *RqlEngine* (siehe 9 RqlEngine – ein RQL-Prozessor) weicht von diesem Vorgehen ab, und liefert das Resultat in Form einer speziellen Java-Datenstruktur zurück.

Beispiel

Abfrage: `^sculpts`

Ergebnis

```
<?xml version="1.0" encoding="UTF-8" ?>
<RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Bag>
    <rdf:li>
      <rdf:Seq>
        <rdf:li rdf:type="resource"
                rdf:resource="http://www.culture.net/rodin424"
        />
        <rdf:li rdf:type="resource"
                rdf:resource="http://www.artchive.com/thinker.jpg"
        />
      </rdf:Seq>
    </rdf:li>
    <rdf:li>
      <rdf:Seq>
        <rdf:li rdf:type="resource"
                rdf:resource="http://www.culture.net/michelangelo"
        />
        <rdf:li rdf:type="resource"
                rdf:resource="http://www.photojournal.com/...descent.jpg"
        />
      </rdf:Seq>
    </rdf:li>
    <rdf:li>
      <rdf:Seq>
        <rdf:li rdf:type="resource"
                rdf:resource="http://www.culture.net/michelangelo"
        />
        <rdf:li rdf:type="resource"
                rdf:resource="http://www.photojournal.com/...theslave.jpg"
        />
      </rdf:Seq>
    </rdf:li>
  </rdf:Bag>
</RDF>
```

6 eRQL – Ad Hoc-Abfragen für Informationsportale

In diesem Kapitel wird eine neue und neuartige Sprache für die Abfrage von RDF-Daten vorgestellt, die den genannten Anforderungen genüge tut (siehe 3 Ziele). In Anlehnung an RQL trägt diese Abfragesprache den Namen:

easy RQL

(abgekürzt: eRQL)

Ausgangspunkt meiner Überlegungen zur Konzeption von *eRQL* ist die Fragestellung gewesen, welche Ansprüche der Besucher eines Informationsportals an dessen Abfragesprache stellen könnte, siehe 3 Ziele. Insbesondere bin ich davon ausgegangen, dass ein Besucher Abfragen im Steno-Stil à la Google tendenziell eher den Vorzug gibt vor Abfragen im Stil von SQL.

Eine Diskussion offener Fragen und Anregungen zu Weiterentwicklungen findet sich in 10 Ausblick.

6.1 Eigenschaften von eRQL

Die folgenden Eigenschaften charakterisieren eRQL in groben Zügen, und vermitteln einen Eindruck dieser Abfragesprache. Einige dieser Aspekte werden im Anschluss an diesen Abschnitt detailliert vorgestellt:

- **Kurz und knapp: Ein-Wort-Abfragen**

  Wie von Google und anderen Internetsuchmaschinen her bekannt, erlaubt eRQL die Suche nach einem bestimmten Begriff, indem dieser als Abfrage verwendet wird: `PICASSO`.

- **Umgebung**

  eRQL erweitert einzelne Ressourcen, Literale oder Aussagen gelegentlich automatisch um benachbarte Aussagen, um dem Abfragesteller zusätzliche Informationen über den Kontext der Fundstellen zurückgeben zu können. Für eine ausführliche Beschreibung dieses Mechanismus' siehe 6.1.2 Umgebung und Abfragemodus.

- **Point Of Interest**

  Ein *Point Of Interest* (*POI*) ist eine spezielle Art von *Umgebung* um eine (oder mehrere) Ressourcen (oder Literale). Die *POI*-Umgebung beinhaltet dabei all jene Ressourcen und Literale, welche in der Graphenrepräsentation des RDF-Modells zu den vorgegebenen Ressourcen bzw. Literalen benachbart sind.

Eine einfache Notation durch den ~-Operator (im Sinne von „unscharf“, „in etwa“) sorgt für leichte Anwendung: `~Picasso`. Für Details siehe 6.1.2 Umgebung und Abfragemodus.

- **Groß-/Kleinschreibung**

  eRQL berücksichtigt generell keine Unterschiede bezüglich der Groß-/Kleinschreibung. Dies betrifft sowohl den Vergleich von Literalen, als auch den Vergleich von Ressourcen/URIs. Siehe dazu auch 6.1.5 Suche nach Texten.

- **Boolesche Verknüpfungen**

  eRQL unterstützt die Operatoren `AND` und `OR`, wie dies von Internetsuchmaschinen, oder auch vom heimischen Betriebssystem bekannt ist: `PABLO OR PICASSO`. Für Details siehe 6.1.3 Boolesche Verknüpfungen.

6.1.1 Kurz und knapp: Ein-Wort-Abfragen

Wie von Google und anderen Internetsuchmaschinen bekannt, erlaubt eRQL die Suche nach einem bestimmten Begriff, indem dieser einfach als Abfrage verwendet wird:

- Eine Abfrage, die nur aus einem Suchbegriff besteht, liefert alle Aussagen zurück, welche diesen Suchbegriff innerhalb ihres Subjektes, Prädikates und/oder Objektes (in beliebiger Groß-/Kleinschreibung) besitzen.
- Zu jeder gefundenen Aussage wird auch deren *Umgebung* zurückgegeben, siehe 6.1.2 Umgebung und Abfragemodus.
- Abfragen, die aus mehreren Suchbegriffen bestehen, werden verarbeitet, als wären die Suchbegriffe durch `AND` miteinander verbunden, siehe 6.1.3 Boolesche Verknüpfungen und Klammerung.

6.1.2 Umgebung und Abfragemodus

Einzelne Aussagen oder gar einzelne Ressourcen haben im Sinne einer Suche innerhalb eines Informationsportals eine zu hohe Granularität. eRQL erweitert daher in verschiedenen Situationen eine einzelne Ressource (bzw. ein einzelnes Literal, bzw. eine Aussage) um dessen *Umgebung*. Diese beinhaltet neben der entsprechenden Aussage selbst alle weiteren Aussagen des RDF-Modells, welche bezüglich der Graphenrepräsentation zu dieser Aussage *benachbart* sind.

Die exakte Definition der *Umgebung* hängt von dem Abfragemodus ab, welcher wiederum durch die Abfrage bestimmt wird:

- Aussagemodus,
- Point Of Interest-Modus oder
- Dokumentmodus.

6.1.2.1 Aussagemodus

Im Aussagemodus, welcher durch den `[…]`-Operator aktiviert wird, beinhaltet die Umgebung einer Aussage lediglich die Aussage selber. Die Umgebung einer Menge von Aussagen beinhaltet exakt diese Aussagenmenge an sich. Die Verwendung des Aussagemodus' liefert verglichen mit dem POI- oder Dokumentmodus die wenigsten Informationen über Fundstellen zurück.

Beispiel

Als Beispiel wird im Folgenden der Use Case aus 6.2.1 Alle Informationen zu „Picasso herangezogen – es gilt, alle Informationen rund um den Begriff „Picasso" zu ermitteln.

Abfrage: `[Picasso]`

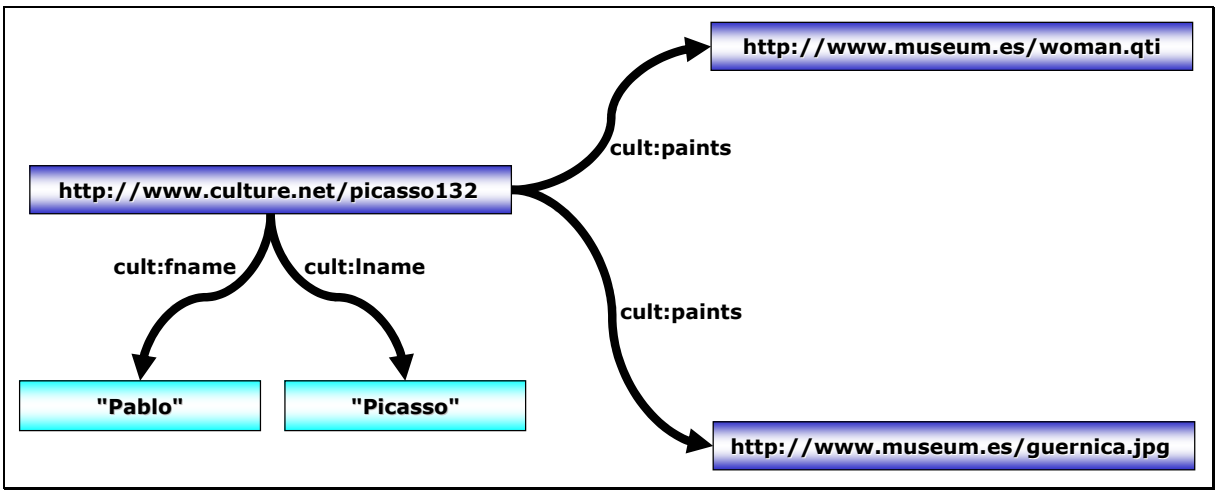

Abbildung 8 – Aussagemodus – Ergebnis der Abfrage: [Picasso]

Es ist zu sehen, dass sämtliche Aussagen des obigen Szenarios (siehe 5.1 Szenario: Ein Kultur-Informationsportal) in dem Ergebnis enthalten sind, welche den Begriff „Picasso" in beliebiger Groß-/Kleinschreibung in ihrem Subjekt, Prädikat und/oder Objekt beinhalten.

6.1.2.2 Point Of Interest-Modus (POI-Modus)

Im *Point Of Interest*-Modus, welcher per Vorgabe aktiv ist, beinhaltet die Umgebung einer Aussage (neben der Aussage an sich) all jene Aussagen, welche diese Aussage *berühren*. Die Verwendung des POI-Modus' liefert mehr Informationen über Fundstellen zurück als der Aussagemodus, jedoch weniger als der Dokumentmodus.

Die POI-Umgebung einer Ressource oder eines Literales besteht aus allen Aussagen, welche diese Ressource bzw. dieses Literal in ihrem Subjekt, Prädikat und/oder Objekt beinhalten, sowie deren Umgebungen. Der Vergleich wird ohne Berücksichtigung der Groß-/Kleinschreibung durchgeführt.

Literale werden automatisch im *Point Of Interest*-Modus ausgewertet, sofern sie nicht direkt in einem <…>- bzw. `[…]`-Operator enthalten sind (dies betrifft textuelle Literale genauso wie URIs). Die Abfragen `Picasso` und `{Picasso}` sind daher äquivalent. Hintergedanke ist, dass der Ab-

fragesteller dadurch weniger eine punktuelle Rückgabe erhält, sondern vielmehr einen Einblick in den gesamten Kontext der Fundstelle bekommt. Dieses Merkmal unterscheidet eRQL fundamental von allen bereits genannten und beschriebenen Abfragesprachen.

Der POI-Modus-Operator `{…}` unterstützt eine alternative Schreibweise, welche die Eingabe erleichtern soll: der ~-Operator. Aber im Gegensatz zu dem `{…}`-Operator bewirkt der ~-Operator in jedem Fall die Ausweitung der Umgebung, auch dann, wenn er unmittelbar auf ein Literal angewendet wird. Die Abfragen `{Picasso}` und `~Picasso` sind daher nicht äquivalent, weil erste eine einfach und zweite eine doppelte POI-Bildung verursacht. Die Abfragen `~{Picasso}`, `~(Picasso)`, `~Picasso` und `{{Picasso}}` hingegen sind äquivalent.

Beispiel – Point Of Interest um Picasso

Abfrage: `Picasso`

oder: `{Picasso}`

oder: `~[Picasso]`

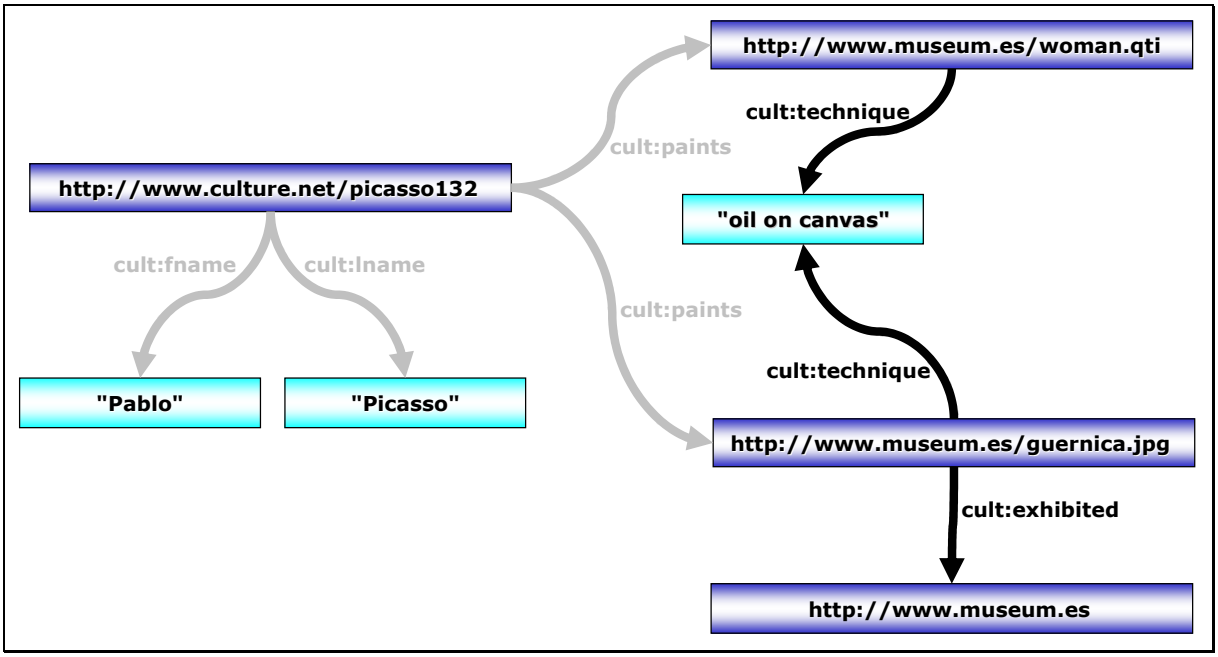

Abbildung 9 – POI-Modus – Ergebnis der Abfrage: Picasso

Sämtliche Aussagen der Abbildung, welche auch im Aussagemodus ermittelt wurden, sind grau dargestellt. Schwarz dargestellt sind hingegen sämtliche Aussagen, welche erst durch die Anwendung des POI-Modus-Operators hinzugekommen sind. Es sind zusätzlich zu allen Aussagen, welche den Begriff „Picasso“ direkt beinhalten, all jene Aussagen hinzugekommen, welche eine oder mehrere der „Picasso-Aussagen“ berühren.

Der POI-Modus-Operator ist der einzige der drei Operatoren, welcher auch bei mehrmaliger Anwendung die Umgebung vergrößert. Dazu wiederum ein Beispiel, welches das obige Beispiel erweitert, in dem es einen doppelten Point Of Interest um das Literal „Picasso“ demonstriert:

Beispiel – Doppelter Point Of Interest um Picasso

Abfrage: `~Picasso`

oder: `{{Picasso}}`

oder: `~~[Picasso]`

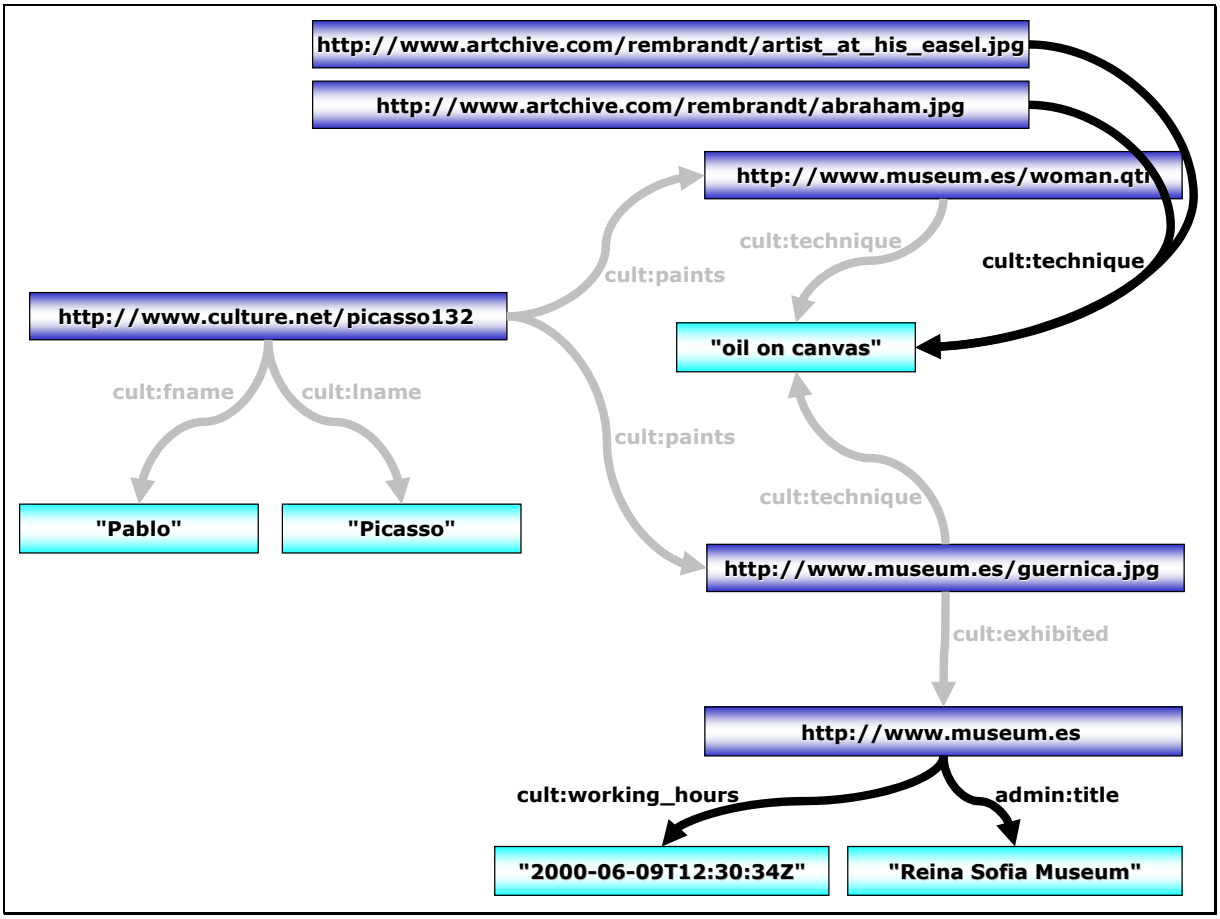

Abbildung 10 – POI-Modus – Ergebnis der Abfrage: ~Picasso

Es ist zu sehen, dass wiederum sämtliche Aussagen ausgegraut sind, welche bereits im vorangegangenen Beispiel (d. h. bei der einfachen POI-Bildung) im Ergebnis enthalten gewesen sind. Neu hinzugekommen sind zwei Aussagen, welche sich auf das „Reina Sofia Museum“ beziehen – einerseits dessen Titel, und andererseits dessen Öffnungszeiten. (Das Museum an sich ist durch die einfache POI-Bildung Teil der Ergebnismenge geworden.)

Neu hinzugekommen sind aber weiterhin zwei Aussagen, die sich auf das Literal „oil on canvas“ beziehen: zwei Rembrandt-Gemälde basieren ebenfalls auf der „oil on canvas“-Technik.

6.1.2.3 Dokumentmodus

Im Dokumentmodus, welcher durch den <...>-Operator aktiviert werden kann, beinhaltet die Umgebung einer Aussage (neben der Aussage an sich) all jene Aussagen, welche demselben Dokument entstammen, wie diese Aussage. Die Verwendung des Dokumentmodus' liefert verglichen mit dem Aussage- oder POI-Modus die meisten Informationen über Fundstellen zurück.

6.1.3 Boolesche Verknüpfungen und Klammerung

Wie von den Internetsuchmaschinen und vielleicht auch vom eigenen Arbeitsplatzrechner bekannt können mehrere Bedingungen durch die Schlüsselwörter `AND` und `OR` miteinander verknüpft werden. Die Verwendung von `AND` ist dabei stets optional, und dient lediglich der besseren Lesbarkeit.

Das Hintereinanderschreiben von Suchbegriffen führt automatisch zu einer `AND`-Verknüpfung aller Suchbegriffe; die Abfrage `PABLO PICASSO` ist also identisch mit `PABLO AND PICASSO`. Wenn dieses Verhalten nicht erwünscht ist, kann mittels doppelter Hochkommata die Betrachtung als ein einziger Suchbegriff erzwungen werden: `"PABLO PICASSO"`.

Eckige, geschweifte und spitze Klammernpaare (d. h. `[…]`, `{…}` und <…>) können verwendet werden, um den Abfragemodus zu ändern, siehe 6.1.2 Umgebung.

Mehrere Bedingungen, die durch `AND` und `OR` miteinander verknüpft sind, können in runde Klammern „(…)“ gestellt werden, um den Operatorvorrang zu ändern: `(PABLO OR PICASSO) AND LOUVRE`. Wird auf den Einsatz von Klammern verzichtet, besitzt `AND` Vorrang vor `OR`. Bedingungen können mittels des `AND`- und `OR`-Operators sowie entsprechender Klammerung beliebig tief geschachtelt werden.

Die Auswertung geschieht (zumindest schematisch) wie folgt:

1. Jede Bedingung wird zunächst als eigenständige Abfrage aufgefasst und ausgewertet. Dadurch wird jede Bedingung zu einer Menge von Aussagen ausgewertet.

   (Streng genommen wird eine Bedingung nicht zu einer Menge von Aussagen ausgewertet, sondern zu einer Menge von „Fundstellen“, wobei jede Fundstelle wiederum einer Menge von Aussagen entspricht. Mit Ausnahme des POI-Modus-Operators besteht aber jede Fundstelle aus exakt einer Aussage.)

2. Die Verknüpfung zweier Bedingungen durch `OR` wird durchgeführt, indem zunächst beide Bedingungen wie beschrieben ausgewertet werden, und die Mengen aller zurückgegebenen Aussagen anschließend vereinigt werden.

3. Die Verknüpfung zweier Bedingungen durch `AND` wird durchgeführt, indem zunächst beide Bedingungen wie beschrieben ausgewertet werden, und anschließend überprüft wird, welche der zurückgegebenen Aussagemengen sich „überlappen“[37]. Falls nicht, ist die Rückgabemenge leer, anderenfalls werden die Rückgabemengen ebenfalls vereinigt.

4. Falls mehr als zwei Suchbegriffe durch `OR` und/oder `AND` verknüpft sind, wird obiges Vorgehen entsprechend angepasst.

[37] D. h. ob es eine Ressource oder ein Literal gibt, welche in beiden Rückgabemengen als Teil einer Aussage enthalten ist, siehe Seite 24.

6.1.4 Erkennung von URIs

RDF unterscheidet zwischen *Ressourcen* und *Literalen*. Ressourcen sind eigenständig definierte „Dinge“, welche über eine eindeutige URI referenziert werden. Literale hingegen sind einfache Texte, XML-Fragmente, Zahlen, Datumswerte und ähnliches.

Die automatische Unterscheidung zwischen URIs und textuellen Literalen ist nicht in jedem Fall mit Sicherheit möglich. Gibt ein Anwender beispielsweise den Text „http://www.louvre.fr“ ein, kann nicht mit Sicherheit entschieden werden, ob der Anwender damit gezielt nach Informationen suchen möchte, welche mit der Ressource „http://www.louvre.fr“ verknüpft sind, oder ob der Anwender einfach nach Texten rund um den Louvre suchen möchte, welche diese URI (zufällig) als Teil des Textes beinhalten.

eRQL wendet ein gänzlich RDF-untypisches Vorgehen an, und behandelt URIs und Literale gleich. Die folgenden Abfragen liefern daher ein identisches Ergebnis:

```
www.Louvre.fr

www.LOUVRE.fr

"www.LOUVRE.fr"
```

Durch die Gleichbehandlung von URIs und textuellen Literalen ist der Besucher des Informationsportals von der Notwendigkeit von Schemakenntnissen befreit, insbesondere davon, vor der Formulierung einer Abfrage wissen zu müssen, ob seine Suchbegriffe in Form von Ressourcen und/oder Literalen vorkommen.

6.1.5 Suche nach Texten

Wahrscheinlicher, als dass der Besucher eines Informationsportals nach URIs suchen möchte, ist der Fall, dass er nach Texten suchen möchte, z. B. nach `PABLO PICASSO`. Dieses Vorgehen kennt und beherrscht er im Allgemeinen bereits von Internetsuchmaschinen.

Der Abfragesteller kann (aber muss nicht) den gesuchten Text in doppelte Anführungszeichen stellen. Was passiert nun, wenn nach einem Text gesucht wird, welcher aus mehr als einem Wort besteht und keine Anführungszeichen verwendet werden?

Eine Abfrage wie z. B. `PABLO PICASSO` ist identisch mit `PABLO AND PICASSO`. Die Rückgabe dieser Abfrage lautet:

Subjekt	Prädikat	Objekt
www.culture.net/picasso132	fname	"Pablo"
-"-	lname	"Picasso"
-"-	paints	www.museum.es/woman.qti
-"-	-"-	www.museum.es/guernica.jpg
www.museum.es/woman.qti	technique	"oil on canvas"
www.museum.es/guernica.jpg	-"-	-"-
-"-	exhibited	http://www.museum.es

Was passiert nun, wenn der Text in Anführungszeichen gestellt wird, also nach `"PABLO PICASSO"` gesucht wird? In diesem Fall wird nach einem Literal gesucht, dessen Wert „Pablo Picasso“ entspricht – solch ein Literal existiert nicht, also wird kein Ergebnis zurückgegeben.

Fazit: Es ist ein Unterschied, ob ein gesuchter Text (der aus mehreren Wörtern besteht) in Anführungszeichen gestellt wird oder nicht. Insbesondere werden mehr Fundstellen zurückgegeben, wenn keine Anführungszeichen verwendet werden.

6.1.6 Operatorvorrang

eRQL unterstützt derzeit weniger als eine Hand voll Operatoren. Diese Operatoren sind im Folgenden aufgelistet, sortiert nach abnehmendem Vorrang:

Operator	Bedeutung
~	Abgekürzte Schreibweise des POI-Modus-Operator, äquivalent mit {…}, siehe 6.1.2 Umgebung und Abfragemodus.
AND	Konjunktion bzw. Schnittmenge mehrerer Teilabfragen, siehe 6.1.3 Boolesche Verknüpfungen und Klammerung.
OR	Disjunktion bzw. Vereinigung mehrerer Teilabfragen, siehe 6.1.3 Boolesche Verknüpfungen und Klammerung.

6.2 Use Cases

In diesem Kapitel werden konkrete Fragestellungen formuliert, welche der Benutzer des Informationsportals aus obigem Szenario (5.1 Szenario: Ein Kultur-Informationsportal) beantwortet wissen haben möchte, und die jeweilige Durchführung mittels eRQL aufgezeigt.

6.2.1 Alle Informationen zu „Picasso“

Abfrage: `Picasso`

Subjekt	Prädikat	Objekt
www.culture.net/picasso132	www.icom.com/schema.rdf#fname	„Pablo“
-"-	www.icom.com/schema.rdf#lname	„Picasso“
-"-	www.icom.com/schema.rdf#paints	www.museum.es/woman.qti
-"-	www.icom.com/schema.rdf#paints	www.museum.es/guernica.jpg

6.2.2 Titel von „http://www.louvre.fr“

Abfrage: `http://www.louvre.fr/ www.icom.com/schema.rdf#title`

Subjekt	Prädikat	Objekt
http://www.louvre.fr/	www.icom.com/schema.rdf#title	„Louvre Museum“

6.2.3 „Ort und Öffnungszeiten des Louvre“

Abfrage: `"Louvre Museum" (location OR working_hours)`

Subjekt	Prädikat	Objekt
http://www.louvre.fr/	www.icom.com/ schema.rdf#location	„FRANCE“
-"-	www.icom.com/ schema.rdf#working_hours	„9-1, 5-8“

6.2.4 Informationen über das „Reina Sofia Museum“

Abfrage: `"Reina Sofia Museum"`

Subjekt	Prädikat	Objekt
http://www.museum.es	last_modified	2000-06-09T12:30:42Z
-"-	title	"Reina Sofia Museum"
http://www.museum.es/guernica.jpg	exhibited	http://www.museum.es

6.2.5 Suche nach „http://www.louvre.fr“

Abfrage: `http://www.louvre.fr`

Subjekt	Prädikat	Objekt
http://www.culture.net/ michelangelo	sculpts	http://www.photojournal.com/ classicart/italmasters/ michelangelo/sculptur/ theslave.jpg
http://www.culture.net/ rembrandt	paints	http://www.archtive.com/ rembrandt/ artist_at_his_easel.jpg
http://www.louvre.fr	title	"Louvre Museum"
http://www.photojournal.com/ classicart/italmasters/ michelangelo/sculptur/ theslave.jpg	exhibited	http://www.louvre.fr
http://www.archtive.com/ rembrandt/ artist_at_his_easel.jpg	-"-	-"-
-"-	technique	"oil on canvas"

6.2.6 „Vorname von Rodin“ finden

Abfrage: `[Rodin] [first_name]`

Subjekt	Prädikat	Objekt
www.culture.net/rodin424	first_name	"August"

6.2.7 „Kunstwerke des Louvre“ ermitteln (nur URIs)

Abfrage: `[Louvre] [exhibited]`

Subjekt	Prädikat	Objekt
http://www.photojournal.com/classicart/itelmasters/michelangelo/sculptur/theslave.jpg	exhibited	http://www.louvre.fr/
http://www.artchive.com/rembrandt/artist_at_his_easel.jpg	-"-	-"-

6.2.8 Kunstwerke des Louvre samt Metainformationen ermitteln

Abfrage: `[Louvre] exhibited`

Subjekt	Prädikat	Objekt
http://www.photojournal.com/classicart/itelmasters/michelangelo/sculptur/theslave.jpg	exhibited	http://www.louvre.fr/
http://www.artchive.com/rembrandt/artist_at_his_easel.jpg	-"-	-"-
-"-	technique	"oil on canvas"
www.culture.net/michelangelo	sculpts	http://www.photojournal.com/classicart/itelmasters/michelangelo/sculptur/theslave.jpg
www.culture.net/Rembrandt	paints	http://www.artchive.com/rembrandt/artist_at_his_easel.jpg
http://www.louvre.fr/	title	"Louvre Museum"

6.3 Formale Semantik

In diesem Abschnitt wird die *formale Semantik* von eRQL definiert. Darunter wird die Beschreibung der Regeln verstanden, nach welchen eine gültige eRQL-Abfrage in ein Abfrageergebnis umgewandelt werden kann. Zur Formulierung wird eine eingewöhnungsbedürftige, formalsprachliche Notation verwendet, welche aber den Vorteil der Exaktheit bietet.

Zu diesem Zwecke werden formale Mittel wie ein statisches Typsystem und Typregeln verwendet, welche nicht Bestandteil der eRQL-Sprache sind. Sie dienen lediglich der Definition und dem Verständnis von eRQL, und können nicht etwa zur Laufzeit als Bestandteil einer eRQL-Abfrage verwendet werden.

Alle aufgeführten Regeln sind *nicht* zwangsläufig die Regeln, nach welchen ein eRQL-Prozessor vorgehen muss. Es handelt sich also nicht um eine Art „Bauanleitung“ für einen eRQL-

Prozessor. Die Regeln beschreiben vielmehr *eine* mögliche Vorgehensweise, zu deren Ergebnis das Ergebnis jedes anderen eRQL-Prozessors äquivalent sein muss.

Dazu wird in Abschnitt 6.3.1 Das Typsystem von eRQL zunächst ein Verständnis dafür vermittelt, welche Datentypen während der Auswertung einer eRQL-Abfrage von Belang sind, um anschließend in Abschnitt 6.3.2 Semantikregeln Regeln für die Reduktion von eRQL-Abfragen auf konkrete Werte zu definieren.

Es wird eine Voraussetzung und eine Folgerung genannt, die eintritt, wenn jene Voraussetzung erfüllt ist. Um die Voraussetzungen und Folgerungen notieren zu können, sind einige spezielle Symbole nötig, welche nur der formalen Definition von eRQL dienen, und nicht etwa Bestandteil der Sprache sind:

Symbol	Bedeutung
⇒	*wird ausgewertet zu*
:	*ist vom Datentyp*
{a1:A,a2:A,…}	Menge bestehend aus a1, a2, … (jeweils vom Typ Aussage)
…∪…	*vereinigt mit*
…∩…	*geschnitten mit*
…∈…	*enthalten in*, z. B. „Ressource … enthalten in Aussage …" oder „Ressource … enthalten in Dokument …" oder „Aussage … enthalten in Dokument …"

6.3.1 Das Typsystem von eRQL

Im weiteren Verlauf dieses Dokumentes werden Symbole wie *m*, *D* und *s* verwendet, um bestimmte Dinge wie ein Datenmodell, den Dokument-Typ oder eine Aussage zu bezeichnen. Um einen Zusammenhang wie „ein Dokument besteht aus Aussagen" formal ausdrücken zu können, sind weiterhin Datentypen wie *Dokument*, *Aussage*, etc. nötig, und entsprechende Symbole dafür. Datentypen werden durch (lateinische) Großbuchstaben referenziert, wie z. B. *D* für *Dokument* und *A* für *Aussage*. Der Ausdruck $d \in D$ beispielsweise bedeutet, dass *d* ein Dokument ist (genau genommen: *d* ist in der Menge aller Dokumente enthalten).

Großbuchstaben (Datentypen) besitzen im Rahmen dieser Arbeit also eine fixe Bedeutung, wohingegen Kleinbuchstaben in jedem Zusammenhang unterschiedlich verwendet werden. Zumeist gilt, dass der Kleinbuchstabe des jeweiligen Datentyps verwendet wird, also z. B. *d* für ein bestimmtes Dokument und *a* für eine bestimmte Aussage.

6.3.1.1 Datenmodell

Der Begriff *Datenmodell* (m) beschreibt die Menge aller Daten, welche einem Informationsportal zur Verfügung stehen, und abgefragt werden können. Im Datenmodell beinhaltet sind beliebig viele *Dokumente* ($d_1,\ldots,d_n$), welche den einzelnen Dateien, RDF-Datenbanken, oder anderen Datenquellen entsprechen:

$$\forall m \in M \quad \Rightarrow \quad m = \{d_1,\ldots,d_n\} \quad n \geq 0 \quad \forall 0 \leq i \leq n : d_i \in D \tag{6.1}$$

M beschreibt die Menge aller Datenmodelle, wobei M ein rein theoretisches Konstrukt ist. M ist dennoch nötig, um mittels der Formulierung m ∈ M formalsprachlich ausdrücken zu können, dass es sich bei m um ein Modell handelt. Im Rahmen dieser Arbeit wird davon ausgegangen, dass dem System ein einziges Datenmodell zur Verfügung steht, in welchem alle referenzierten Dokumente, Aussagen und Ressourcen enthalten sind.

Eine Bedingung wie $\mathrm{dynEnv}(\mathrm{mode}) \in \{\mathrm{document}, \mathrm{poi}\}$ zeigt an, dass der Wert von „mode“ in einer bestimmten Umgebung (die hier „dynEnv“ genannt ist, allerdings spielt der Name hier keine Rolle) den Wert „document“ oder „poi“ annehmen muss.

6.3.1.2 Dokument

Ein *Dokument* (*d*) beinhaltet, genau wie in der realen Welt, beliebig viele Aussagen (*a*), und dient als logisches Strukturelement, d. h. als Zusammenfassung eben jener enthaltenen Aussagen:

$$\forall d \in D \quad \Rightarrow \quad d = \{a_1, \ldots, a_n\} \quad n \geq 0 \quad \forall 0 \leq i \leq n : a_i \in A \tag{6.2}$$

D beschreibt die Menge aller Dokumente. Das *Dokument* besitzt keine Entsprechung in RDF.

6.3.1.3 Aussagengruppen

Eine *Aussagengruppe* ist die Zusammenfassung beliebig vieler Aussagen zu einem logischen Verbund. Einziger Sinn und Zweck der *Aussagengruppe* ist die Realisierung der POIs, siehe 6.1.2 Umgebung und Abfragemodus und 7.5 Point Of Interest-Operator {…}. *G* beschreibt die Menge aller Aussagengruppen.

$$\forall g \in G \quad \Rightarrow \quad g = \{a_1 \in A, \ldots, a_n \in A\} \tag{6.3}$$

6.3.1.4 Aussage

Eine *Aussage* (*a*) entspricht einer RDF-Aussage und besteht jeweils aus einem Subjekt, einem Prädikat und einem Objekt in dieser Reihenfolge. Subjekt und Prädikat sind wiederum jeweils eine *Ressource* (*r*), das Objekt eine *Ressource* (*r*) oder aber ein *Literal* (*l*):

$$\forall a \in A \quad \Rightarrow \quad a = (s, p, o) \quad s, p \in R \quad o \in R \cup L \tag{6.4}$$

A beschreibt die Menge aller Aussagen. Im Gegensatz zur allgemeinen Verwendung des Begriffes *Aussage* im Zusammenhang mit RDF wird in dieser Arbeit davon ausgegangen, dass jeder Aussage die Information über das Dokument hinterlegt ist, welchem sie entstammt (auch dann, wenn diese Aussage mehreren Dokumenten entstammt).

Mittels der Funktion *statements()* kann zu einem gegebenen *Dokument* oder einer gegebenen *Aussagengruppe* die Menge der enthaltenen Aussagen bestimmt werden:

$$\begin{aligned} \text{statements}(d) &:= \{a_1,\ldots,a_n\} \quad \big| d \in D \quad d = \{a_1,\ldots,a_n\} \\ \text{statements}(g) &:= \{a_1,\ldots,a_n\} \quad \big| g \in G \quad g = \{a_1,\ldots,a_n\} \end{aligned} \tag{6.5}$$

6.3.1.5 Ressourcen und Literale

Ressourcen (*r*) und *Literale* (*l*) sind die kleinsten Einheiten der RDF-Welt und repräsentieren jeweils eine bestimmte Sache. Darunter sind Objekte der realen Welt (z. B. Personen, Gegenstände, Tätigkeiten, etc.) genauso zu verstehen wie Objekte der virtuellen Welt (z. B. Websites, Grafikdateien, E-Mail-Adressen, etc.):

$$\forall r \in R \quad \Rightarrow \quad r \text{ ist eine URI} \tag{6.6}$$

$$\forall l \in L \quad \Rightarrow \quad l \text{ ist ein Literal} \tag{6.7}$$

R beschreibt die Menge aller Ressourcen, und *L* die Menge aller Literale. Beide diese Mengen sind disjunkt:

$$R \cap L = \varnothing \tag{6.8}$$

Mittels der Funktionen *values()* und *literals()* können zu einem gegebenen Dokument die Menge der enthaltenen Ressourcen bzw. Literale bestimmt werden, mittels *values()* Ressourcen *und* Literale:

$$\begin{aligned} \text{resources}(a) &:= \{s,p,o\} && \big| a \in A \quad a = (s,p,o) \quad o \in R \\ \text{resources}(a) &:= \{s,p\} && \big| a \in A \quad a = (s,p,o) \quad o \in L \\ \text{resources}(g) &:= \bigcup_i \text{resources}(a_i) && \big| g \in G \quad g = \{a_1,\ldots,a_n\} \\ \text{literals}(a) &:= \{\ \} && \big| a \in A \quad a = (s,p,o) \quad o \in R \\ \text{literals}(a) &:= \{o\} && \big| a \in A \quad a = (s,p,o) \quad o \in L \\ \text{literals}(g) &:= \bigcup_i \text{literals}(a_i) && \big| g \in G \quad g = \{a_1,\ldots,a_n\} \\ \text{values}(a) &:= \{s,p,o\} && \big| a \in A \quad a = (s,p,o) \\ \text{values}(g) &:= \bigcup_i \text{values}(a_i) && \big| g \in G \quad g = \{a_1,\ldots,a_n\} \end{aligned} \tag{6.9}$$

6.3.2 Semantikregeln

Die folgenden Regeln beschreiben die Semantik von eRQL, das heißt die Regeln, nach welchen eRQL-Abfragen zu Abfrageergebnissen ausgewertet werden. Diese Regeln sind jedoch *nicht* zwangsläufig die Regeln, nach welchen ein eRQL-Prozessor vorgehen muss, siehe 6.3 Formale Semantik.

Es ist zu beachten, dass vor der Verarbeitung einer eRQL-Abfrage zunächst eine Vorverarbeitung (auch *preprocessing* genannt) durchgeführt werden muss, siehe 6.4 Vorverarbeitung einer Anfrage. Grund ist, dass eRQL einige Vereinfachungen für den Abfragesteller bietet, zu denen z. B. der ~-Operator zählt, welche zugunsten der Übersichtlichkeit nicht eigenständig in der Semantikdefinition erwähnt sind, sondern bereits in der Vorverarbeitung auf andere Operatoren abgebildet werden.

Die folgenden Regeln verwenden eine formalsprachliche wenn-dann-Notation. Jede einzelne dieser Regeln folgt demselben Muster, wonach die Voraussetzung und ihre Folgerung durch eine horizontale Linie separiert werden:

$$\frac{\text{Voraussetzung}}{\text{Folgerung}}$$

Zu beachten ist, dass solch eine Regel ausschließlich in den Fällen Aussagekraft besitzt, in denen die *Voraussetzung* zutreffend ist. In allen anderen Fällen, in denen die Voraussetzung nicht zutreffend ist, besitzt sie keinerlei Aussagekraft.

6.3.2.1 URIs und Literale

Eine URI oder ein Literal (d. h. ein „Suchbegriff“) *LiteralOrUri* wird zu der Menge aller Aussagen ausgewertet, welche diese URI bzw. dieses Literal enthalten:

$$\frac{\{a_1,\ldots,a_n\}:\{A\}=\{a_i \mid \texttt{LiteralOrUri} \in a_i \in A\}}{\texttt{LiteralOrUri}: L \cup R \Rightarrow \{\{a_1:A\},\ldots,\{a_n:A\}\}:\{G\}} \tag{6.10}$$

6.3.2.2 Boolesche Verknüpfungen und Klammerung

$$\frac{\begin{array}{c}\texttt{Conjunction}_1 \Rightarrow \{g_1,\ldots,g_{j-1}\}:\{G\} \\ \texttt{Conjunction}_2 \Rightarrow \{g_j,\ldots,g_n\}:\{G\}\end{array}}{\texttt{Conjunction}_1 \ \texttt{OR}\ \texttt{Conjunction}_2 \Rightarrow \left\{\bigcup_i \text{statements}(g_i)\right\}:\{G\}} \tag{6.11}$$

$$\frac{\begin{array}{c}\texttt{SubQuery}_1 \Rightarrow \{g_1,\ldots,g_r\}:\{G\} \\ \texttt{SubQuery}_2 \Rightarrow \{h_1,\ldots,h_s\}:\{G\}\end{array}}{\texttt{SubQuery}_1\texttt{AND}\texttt{SubQuery}_2 \Rightarrow \left\{\bigcup_{g_i \cap h_j \neq \emptyset} \text{statements}(g_i) \cup \text{statements}(h_j)\right\}:\{G\}} \tag{6.12}$$

$$\frac{\texttt{Disjunction}_1 \Rightarrow \{g_1,\ldots,g_n\}:\{G\}}{(\texttt{Disjunction}_1) \Rightarrow \{g_1,\ldots,g_n\}:\{G\}} \tag{6.13}$$

6.3.2.3 Umschaltung des Modus' mittels Klammerung

$$\frac{\texttt{Disjunction} \Rightarrow \{g_1,\ldots,g_m\} : \{G\} \qquad \{a_1 : A,\ldots,a_n : A\} = \bigcup_i \text{statements}(g_i)}{\texttt{[Disjunction]} \Rightarrow \{\{a_1 : A\},\ldots,\{a_n : A\}\} : \{G\}} \tag{6.14}$$

$$\frac{\texttt{Disjunction} \Rightarrow \{g_1,\ldots,g_m\} : \{G\} \qquad \{a_1,\ldots,a_n\} = \left\{ a_i \,|\, a_i \in A \quad \exists b \in \bigcup_j \text{statements}(g_j) : b \cap a_i \neq \varnothing \right\}}{\{\texttt{Disjunction}\} \Rightarrow \{a_1,\ldots,a_n\}} \tag{6.15}$$

$$\frac{\texttt{Disjunction} \Rightarrow \{g_1 : G,\ldots,g_n : G\} \qquad \{d_1,\ldots,d_m\} = \left\{ d_i \,|\, d_i \in D \quad \text{values}(d_i) \cap \bigcup_i \text{statements}(g_i) \neq \varnothing \right\}}{\langle \texttt{Disjunction} \rangle \Rightarrow \bigcup_{d_i} \text{statements}(d_i)} \tag{6.16}$$

6.4 Vorverarbeitung einer Anfrage

Oberste Prämisse bei der Konzeption von eRQL war die Einfachheit, um nicht zu sagen: Intuitivität für den Steller der Abfrage. Daher sieht eRQL Vereinfachungen vor, welche nicht in der vorangegangenen Semantikdefinition erwähnt sind, um diese übersichtlich und redundanzarm zu halten.

Im Folgenden sind Umformungsregeln aufgeführt, welche auf jede eRQL-Abfrage angewendet werden müssen, bevor die obige Semantikdefinition verwendet werden kann. Einem eRQL-Prozessor steht es frei, die Auswertung einer eRQL-Abfrage auf eine beliebige andere Art und Weise durchzuführen – das Resultat muss jedoch identisch sein.

Eine konkrete eRQL-Implementierung würde möglicherweise überhaupt keine der folgenden Umformungen vor der Auswertung vornehmen, sondern statt dessen alle Regeln zum Zeitpunkt der Auswertung entsprechend beachten. Zusätzlich würde eine eRQL-Implementierung zusätzliche Optimierungen der Abfrage vornehmen, um Auswertungszeit einzusparen, siehe auch 8.5 Optimierungen.

6.4.1 POI-Modus-Operatoren einfügen

Die Verwendung des POI-Modus-Operators `{…}` um Literale und URIs ist für den Abfragesteller optional. Daher ist um jeden Suchbegriff, d. h. um jedes Wort der Abfrage, welches nicht bereits mit dem Dokument- oder dem Aussagemodus-Operator umklammert ist (also `{…}`, `<…>` bzw. `[…]`), der POI-Modus-Operator zu setzen.

URIs und Literale, welche bereits mit einem POI-Modus-Operator umklammert sind, werden mit einem zweiten POI-Modus-Operator umklammert.

Beispiel:

```
(PICASSO OR [PABLO]) AND {LOUVRE}
⇓
({PICASSO} OR [PABLO]) AND {{LOUVRE}}
```

6.4.2 Tilde-Operatoren ersetzen

Der Tilde-Operator ~ ist eine Vereinfachung für den Abfragesteller und semantisch äquivalent zum POI-Modus-Operator ({…}). Daher müssen seine Vorkommen entsprechend substituiert werden:

Beispiel

```
[guernica] PICASSO ~PABLO ~~LOUVRE
⇓
[guernica] {PICASSO} {{PABLO}} {{{LOUVRE}}}
```

6.4.3 AND-Operatoren einfügen

Die Verwendung des AND-Operators als Konjunktion zwischen Literalen und URIs der Abfrage ist für den Abfragesteller optional. Daher ist zwischen je zwei Wörter der Eingabe, welche weder durch den AND- noch durch den OR-Operator verbunden sind, der AND-Operator einzufügen.

Beispiel

```
(PICASSO PABLO) LOUVRE
⇓
(PICASSO AND PABLO) AND LOUVRE
```

6.5 Zusammenfassung: eRQL

Mit eRQL wurde in den vorangegangenen Abschnitten eine intuitive und dennoch leistungsfähige Abfragesprache für Informationsportale vorgestellt, welche alle Anforderungen erfüllt, die in 3 Ziele genannt worden sind.

So ist eRQL sehr einfach zu verwenden: Suchabfragen bestehen im einfachsten Fall lediglich aus dem gesuchten Begriff – wie es vielen Menschen von Internetsuchmaschinen her bereits geläufig ist. Auch komplizierte Abfragen lassen sich durch einfaches Hintereinanderschreiben der gesuchten Begriffe durchführen.

eRQL erfordert keinerlei Schemakenntnisse: Der Abfragesteller muss – und kann auch gar nicht – spezifizieren, ob der nachzuschlagende Begriff als Subjekt, Prädikat oder Objekt verwendet

werden muss, ob es sich bei ihm um den Teil eines Textes, einer URI, oder eine Zahl handelt, und ob es sich um Groß-, Klein- oder Gemischtschreibung handelt.

eRQL verwendet keine „komplizierte“ Syntax, wie dies bei SQL, RQL, und den meisten anderen Abfragesprachen der Fall ist, insbesondere werden eRQL-Abfragen ohne SELECT-FOR-WHERE konstruiert.

eRQL bietet eine begrenzte Mächtigkeit zugunsten seiner Einfachheit, und insbesondere nur rudimentäre Möglichkeiten zur Selektion und Projektion. Damit einher geht eine geringe Granularität des Datenmodells, welches als kleinste abfragbare Einheit die Aussage verwendet, und nicht etwa die Ressource.

Der Dokumentmodus (siehe 6.1.2 Umgebung und Abfragemodus) sieht zwar das Arbeiten mit RDF-Dokumenten vor, resultiert jedoch schnell in erschlagenden Rückgabemengen, sobald die verwendeten Dokumente an Inhalt zunehmen.

7 Umwandlung von eRQL- in RQL-Abfragen

In diesem Abschnitt wird die Umwandlung von eRQL-Abfragen nach *RQL* beschrieben. Die Abfragesprache eRQL wurde vom Start weg mit Hinsicht darauf konzipiert, eRQL-Abfragen zunächst in RQL-Abfragen umzuwandeln, um diese anschließend von einem RQL-Prozessor auswerten zu lassen.

Für die verschiedenen sprachlichen Konstrukte von eRQL werden im Folgenden Überführungsregeln angegeben. In jenen Fällen, in denen es mehrere Möglichkeiten gibt, eine eRQL-Abfrage nach RQL abzubilden, wurde diejenige gewählt, welche die einfachste Implementierung eines RQL-Prozessors zulässt.

7.1 Groß- und Kleinschreibung

Im Gegensatz zu RQL unterscheidet eRQL *nicht* zwischen der Groß- und Kleinschreibung von textuellen Literalen und auch URIs. Da der RQL-Vergleichsoperator `LIKE` keine Möglichkeit kennt, die Unterscheidung von Groß- und Kleinbuchstaben zu verhindern, wird für die Realisierung von eRQL der folgende Kunstgriff herangezogen:

Jedes Literal, welches Buchstaben enthält, wird in einen Regulären Ausdruck umgewandelt. Dabei wird jeder Buchstabe durch eine Zeichenklasse substituiert, welche aus dem Buchstaben selber besteht, sowie aus der jeweils invertierten Groß-/Klein-Schreibweise dieses Buchstabens.

Beispiel

```
Louvre
⇓
[Ll][Oo][Uu][Vv][Rr][Ee]
```

Anmerkung

Da dieses Vorgehen die extensive Verwendung Regulärer Ausdrücke verlangt, wäre alternativ auch eine Erweiterung von RQL um tolerantere Vergleichsoperatoren à la `ILIKE` (also z. B. …`ILIKE Louvre`…) denkbar. Um nicht mit der RQL-Spezifikation zu brechen verzichtet eRQL auf diese Erweiterung und nimmt den Umweg mittels Regulärer Ausdrücke in Kauf.

7.2 URI bzw. Literal

Jede URI bzw. jedes Literal *X* der Abfrage wird zunächst gemäß 7.1 Groß- und Kleinschreibung in einen Regulären Ausdruck *X'* umgewandelt, und mit einem führenden und schließenden * als Platzhalter für beliebige andere, vorausgehende oder folgende Zeichen. Anschließend wird die folgende RQL-Abfrage erzeugt und durchgeführt:

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}
WHERE
     s LIKE "*X´*" OR @p LIKE "*X´*" OR o LIKE "*X´*"
```

Die Rückgabemenge wird gebildet, indem die Rückgabemenge der obigen RQL-Abfrage übernommen wird.

Algorithmische Beschreibung der Vorgehensweise

1. Erzeuge Regulären Ausdruck aus URI bzw. Literal zur Unterdrückung der Unterscheidung zwischen Groß- und Kleinschreibung
2. Erzeuge RQL-Abfrage (siehe oben) und werte diese aus
3. Gebe Ergebnismenge der RQL-Abfrage zurück

Anmerkungen

Dieses Vorgehen findet alle Aussagen, welche die URI bzw. das Literal *X* in beliebiger Groß-/Kleinschreibung als Teil ihres Subjektes und/oder Prädikates und/oder Objektes beinhalten.

7.3 Verknüpfung mittels OR (Disjunktion)

Mehrere mittels des Operators `OR` verknüpfte Teilabfragen $X_1,\ldots,X_n$ werden zunächst separat zu einer Menge von Fundstellen ausgewertet, welche wiederum jeweils eine Menge von Aussagen enthalten. Bei jeder dieser Teilabfragen kann es sich z. B. um ein einfaches Literal handeln, aber auch um eine weitere, verschachtelte Abfrage.

Jede Teilabfrage wird zunächst separat ausgewertet. Das Resultat wird erzeugt, indem die Ergebnismengen aller dieser Abfragen vereinigt werden.

Algorithmische Beschreibung der Vorgehensweise

1. Erzeuge einen leeren Puffer für die Rückgabemenge
2. Werte jede der *n* (per `OR` verknüpften) Teilabfragen aus
3. Füge jede Rückgabemenge der *n* durchgeführten Abfragen dem Puffer hinzu
4. Gebe den Puffer zurück

Anmerkungen

Dieses Vorgehen führt unter Umständen mehr Operationen durch, als tatsächlich zwingend nötig sind. Sofern die Abfrage eine Disjunktion von Literalen und/oder URIs entspricht, welche mittels `OR` verknüpft sind, könnte eine einzige RQL-Abfrage zum Einsatz kommen, welche eine entsprechend umfangreiche Selektionsklausel besitzen müsste, und z. B. wie folgt formuliert sein könnte:

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}
WHERE
     s LIKE "*X1'*" OR @p LIKE "*X1'*" OR o LIKE "*X1'*" OR …
     OR s LIKE "*Xn'*" OR @p LIKE "*Xn'*" OR o LIKE "*Xn'*"
```

Für eine Disjunktion von *n* Literalen müsste so lediglich eine RQL-Abfrage durchgeführt werden, statt *n* RQL-Abfragen, wie dies im zuvor beschriebenen Verfahren der Fall ist.

Um dieses Vorgehen anzuwenden, welches im Sinne einer Laufzeitoptimierung implementiert werden kann, ist es nötig, zunächst sicherzustellen, dass einerseits alle Teilabfragen URIs und Literale sind, und andererseits der Abfragemodus für alle URIs bzw. Literale identisch ist (denn die durch den Abfragemodus bewirkten Veränderungen an der Rückgabemenge würden nach Durchführung dieser Abfrage auf das gesamte Abfrageergebnis angewendet werden müssen).

Problematisch ist in diesem Zusammenhang auch, dass die RQL-Referenzimplementierung [RQL-Demo] derartige Abfragen ab einer bestimmten Menge von Quantoren fehlerhaft ausführt, d. h. keine Rückgabe mehr liefert. Ein Grund dafür, warum die aktuelle eRQL-Implementierung (siehe 8 eRqlEngine – ein eRQL-Prozessor) keinen Gebrauch von dieser Optimierungsmöglichkeit macht, und statt dessen möglichst einfache RQL-Abfragen verwendet.

7.4 Verknüpfung mittels AND (Konjunktion)

Genau wie im Fall der Disjunktion werden auch mittels des Operators `AND` verknüpfte Teilabfragen X_1, …, X_n zunächst separat zu einer Menge von Fundstellen ausgewertet, welche wiederum jeweils eine Menge von Aussagen enthalten.

Das Resultat wird erzeugt, indem für jede Kombination von *n* Fundstellen (jeweils eine aus dem Resultat jeder Abfrage) ermittelt wird, ob sie sich *überlappen* (siehe Seite 23). Falls eine Überlappung besteht, werden alle Aussagen dieser *n* Fundstellen in die Rückgabe übernommen. Anschließend wird mit der nächsten Kombination von *n* Fundstellen fortgefahren.

Algorithmische Beschreibung der Vorgehensweise

1. Erzeuge einen leeren Puffer für die Rückgabemenge

2. Werte jede der *n* (per `AND` verknüpften) Teilabfragen aus, und erhalte pro Teilabfrage eine Menge von Fundstellen zurück
3. Für jede Kombination von *n* Fundstellen (jeweils eine aus der Rückgabe jeder Abfrage) führe durch:

   Teste, ob die Schnittmenge aller Ressourcen aller Aussagen dieser *n* Fundstellen nicht leer ist. Falls ja: Füge alle Aussagen dieser *n* Fundstellen dem Puffer hinzu.
4. Gebe den Puffer zurück

7.5 Point Of Interest-Operator {…}

Der POI-Modus ist der Vorgabemodus und gilt daher für jede URI und jedes Literal, für welches nicht explizit ein anderer Modus erzwungen wird. Es gilt, zu einer Menge von Aussagen die *Umgebung* dieser Aussagen zu ermitteln und hinzuzufügen.

Algorithmische Beschreibung der Vorgehensweise

1. Erzeuge einen Puffer für die Rückgabemenge, der mit der Eingabemenge initialisiert wird
2. Iteriere durch alle Aussagen der Eingabemenge, und führe für jede einzelne dieser Aussagen durch:

   Füge dem Puffer alle Aussagen des Modells hinzu, welche sich mit dieser Aussage überlappen, d. h. welche eine Ressource oder ein Literal mit dieser Aussage gemeinsam haben. Führe auch diesen Vergleich ohne Berücksichtigung der Groß-/Kleinschreibung durch, siehe 7.1 Groß- und Kleinschreibung.
3. Gebe den Puffer zurück

7.6 Aussagemodus-Operator […]

Der Aussagemodus-Operator `[…]` erzwingt das Zurückgeben der einzelnen Aussagen, und somit die Verhinderung der POI-Bildung. Die Eingabemenge an Aussagen wird also unverändert zurückgegeben.

Algorithmische Beschreibung der Vorgehensweise

Gebe alle Aussagen der Eingabemenge unverändert zurück

7.7 Dokumentmodus-Operator <…>

Der Dokumentmodus kann mittels des Dokumentmodus-Operators <…> erzwungen werden, und bewirkt das Zurückgeben sämtlicher Aussagen der betroffenen Dokumente. Es gilt, zu einer Menge von Aussagen zunächst die betroffenen Dokumente zu bestimmen, und anschließend alle Aussagen dieser Dokumente zu ermitteln und hinzuzufügen. Siehe auch 7.8.1 Dokumentmodus.

Algorithmische Beschreibung der Vorgehensweise

1. Erzeuge einen leeren Puffer für die Rückgabemenge
2. Iteriere durch alle Aussagen der Eingabemenge, und führe für jede einzelne dieser Aussagen durch:
   - Ermittele die Menge derjenigen Dokumente, welchen diese Aussage entstammt
   - Füge für jedes der ermittelten Dokumente sämtliche enthaltenen Aussagen dem Puffer hinzu
3. Gebe den Puffer zurück

7.8 Implementierung

In diesem Abschnitt werden Aspekte der Implementierung eines eRQL-Prozessors angesprochen, welche generell und insbesondere unabhängig von der Beispielimplementierung *eRqlEngine* (siehe 8 eRqlEngine – ein eRQL-Prozessor) auftreten.

7.8.1 Dokumentmodus

Die Implementierung des Dokumentmodus' (siehe 6.1.2 Umgebung und Abfragemodus) ist problematischer als die Implementierung des POI- oder Aussagemodus'.

Im Gegensatz zum POI- und Aussagemodus erfordert der Dokumentmodus ein Mehr an Informationen gegenüber dem Datenmodell von RDF: die Zugehörigkeit von Aussagen zu *Dokumenten* (siehe Seite 23). Im Dokumentmodus gilt es, eine gegebene Aussage um sämtliche Aussagen desselben Dokumentes zu erweitern. Dazu sind Informationen nötig, welche zu einer gegebenen Aussage das Dokument, und umgekehrt zu einem gegebenen Dokument die Menge der enthaltenen Aussagen ermitteln lassen. RDF kennt den Begriff des *Dokumentes* nicht, und stellt entsprechende Informationen daher auch nicht zur Verfügung.

Da RDF den Begriff des *Dokumentes* nicht kennt, werden Dokumente auch nicht explizit von RQL unterstützt. Selbiges gilt für alle zur Verfügung stehenden RDF-Parser (insbesondere auch nicht von dem in der eRQL-Referenzimplementierung zum Einsatz kommenden RDF-Parser *VRP* (siehe 9.6 Auswertung von Abfragen mittels VRP)).

7.8.2 Optimierung des POI-Modus'

Im Rahmen einer Optimierung könnte eine performantere Behandlung von *Points Of Interest* implementiert werden, als in 7.5 Point Of Interest-Operator {…} beschrieben ist. Dadurch könnten die zur Zeit nötigen zahlreichen RQL-Abfragen (zunächst die Ermittlung von Aussagen, anschließend die Ausweitung jeder gefundenen Aussage auf deren Umgebung) auf eine einzige RQL-Abfrage reduziert werden.

Zur Optimierung stehen mehrere Möglichkeiten zur Verfügung, derer drei im Folgenden aufgelistet sind. Von diesen drei Möglichkeiten lässt sich derzeit lediglich die erste mittels *RqlEngine*

auswerten, die weiteren können z. B. mittels der Referenzimplementierung [RQL-Demo] ausgewertet werden.

Die derzeitige Implementierung von *eRqlEngine* erzeugt keine optimierten *POI*-Abfragen. Obwohl dieses Vorgehen sehr performant wäre, und von *RqlEngine* teilweise auch bereits unterstützt wird, gibt es Gründe, welche gegen diesen Einsatz sprechen. Die Vor- und Nachteile der optimierten *POI*-Abfragen sind im Einzelnen:

+ Die Ausführung einer einzelnen RQL-Abfrage ist *wesentlich* performanter als die Ausführung mehrerer RQL-Abfragen.

– Komplexe Abfragen (basierend auf Pfadausdrücken mit mehr als einer Aussage), Quantoren oder gar Unterabfragen (*Sub-Selects*) stellen höhere Ansprüche an den verwendeten RQL-Prozessor, und schränken die Auswahl der verwendbaren Prozessoren tendenziell ein.

– Derzeit erzeugt *eRqlEngine* lediglich Pfadausdrücke, welche genau eine einzige Aussage beinhalten (`{s}@p{o}`). Zur Realisierung des optimierten *POI*-Modus' sind Pfadausdrücke mit mindestens zwei Aussagen notwendig, z. B. `{s}@p{o}.{o}@p2{o2}`, `{s}@p{o}.{o}@p2{o2}.{o2}@p3{o3}` – je nach Größe der zu ermittelnden Umgebung (d. h. je nach Anzahl der verschachtelten *POI*-Operatoren). Bei Gebrauch von Pfadausdrücken liefert RQL jedoch nur dann eine Fundstelle zurück, wenn der Pfadausdruck vollständig gefunden wird. Um also z. B. das Resultat einer dreifachen *POI*-Bildung (z. B. `~~Picasso`) zu ermitteln, sind bereits drei dieser Pfadausdrücke notwendig, deren Rückgabemengen vereinigt werden müssen: je einer für die Umgebungen der Größe eins, zwei und drei. Diese Tatsache erhöht die Ansprüche an den RQL-Prozessor, und reduziert den Vorteil hinsichtlich der Ausführungsgeschwindigkeit. Um diesen Nachteil auszuschließen, können also nur einfache *POIs* (d. h. *POIs* der Größe eins) optimiert werden.

– Bei der Auswertung von *POIs* ist es unabdingbar, die Information darüber zu erhalten, welche Aussagen der Rückgabe jeweils zu einer bestimmten Fundstelle gehören. Der Grund dafür ist die Disjunktion von Teilabfragen, bei welcher auf Basis dieser Fundstellen geprüft wird, ob eine Überlappung vorliegt – nicht etwa auf Basis der zu den Fundstellen gehörigen Aussagen. Siehe dazu auch 6.1.3 Boolesche Verknüpfungen und Klammerung. Diese Information geht bei allen aufgeführten Optimierungsansätzen verloren.

Möglichkeit 1

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}, {a}@b{c}
WHERE
     (s LIKE "*X'*" OR o LIKE "*X'*" OR
      a LIKE "*X'*" OR c LIKE "*X'*")
AND
     (s=a OR s=c OR o=a OR o=c)
```

Beispiel

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}, {a}@b{c}
WHERE
     (s LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*" OR
      o LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*" OR
      a LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*" OR
      c LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*")
AND
     (s=a OR s=c OR o=a OR o=c)
```

s	@p	o
www.culture.net/picasso132	paints	http://www.museum.es/guernica.jpg
-"-	-"-	http://www.museum.es/woman.qti
-"-	first_name	"Pablo"
-"-	last_name	"Picasso"
http://www.museum.es/guernica.jpg	exhibited	http://www.museum.es
-"-	technique	"oil on canvas"
http://www.museum.es/woman.qti	-"-	-"-

Anmerkungen

Dieses Beispiel lässt sich mittels *RqlEngine* nachvollziehen. Die Vorgehensweise basiert auf den mächtigen Pfad-Funktionen von RQL, welche in *RqlEngine* (teilweise) implementiert sind. Die Pfad-Funktionen von RQL erlauben die Definition von Pfaden innerhalb des RDF-Graphen und somit auch die Bindung von Variablen an benachbarte Aussagen. Für die Bildung von *POIs* erscheinen sie auf den ersten Blick ideal, und wie obiges Beispiel demonstriert, sind sie dafür in zumindest einigen Fällen auch geeignet.

Dieses Vorgehen ist zwar in *eRqlEngine* implementiert (Klasse `eworks.eRQL.model.Literal`), wird jedoch aus den oben genannten Gründen heraus zur Zeit nicht verwendet.

Möglichkeit 2

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}
WHERE
  s IN (SELECT a FROM {a}@b{c} WHERE c LIKE "*X'*") OR
  o IN (SELECT a FROM {a}@b{c} WHERE c LIKE "*X'")
```

Beispiel

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}
WHERE
     s IN (
          SELECT a FROM {a}@b{c}
          WHERE c LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*"
     ) OR o IN (
          SELECT a FROM {a}@b{c}
          WHERE c LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*"
     )
```

s	@p	o
www.culture.net/picasso132	creates	http://www.museum.es/guernica.jpg
-"-	-"-	http://www.museum.es/woman.qti
-"-	paints	http://www.museum.es/guernica.jpg
-"-	-"-	http://www.museum.es/woman.qti
-"-	first_name	"Pablo"
-"-	last_name	"Picasso"

Anmerkungen

Aus den oben genannten Gründen heraus ist dieses Vorgehen in *eRqlEngine* zur Zeit nicht implementiert.

Möglichkeit 3

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}
WHERE (
     exists d IN (
          SELECT a FROM {a}@b{c}
          WHERE c LIKE "*s*"
     ) SUCH THAT (d=s OR d=o)
)
```

Beispiel

```
SELECT DISTINCT
     s,@p,o
FROM
     {s}@p{o}
WHERE (
     exists d IN (
          SELECT a FROM {a}@b{c}
          WHERE c LIKE "*[Pp][Ii][Cc][Aa][Ss][Ss][Oo]*"
     ) SUCH THAT (d=s OR d=o)
)
```

s	@p	o
www.culture.net/picasso132	creates	http://www.museum.es/guernica.jpg
-"-	-"-	http://www.museum.es/woman.qti
-"-	paints	http://www.museum.es/guernica.jpg
-"-	-"-	http://www.museum.es/woman.qti
-"-	first_name	"Pablo"
-"-	last_name	"Picasso"

Anmerkungen

Aus den oben genannten Gründen heraus ist dieses Vorgehen in *eRqlEngine* zur Zeit nicht implementiert.

8 eRqlEngine – ein eRQL-Prozessor

eRqlEngine ist die prototypische Implementierung eines eRQL-Prozessors, und wurde im Rahmen dieser Arbeit erstellt. Einen schnellen visuellen Eindruck von *eRqlEngine* vermittelt 8.6 Screenshots

eRqlEngine benötigt einen RQL-Prozessor, und verwendet zu diesem Zweck den ebenfalls im Rahmen dieser Arbeit implementierten RQL-Prozessor *RqlEngine* (siehe 9 RqlEngine – ein RQL-Prozessor, 7 Umwandlung von eRQL- in RQL-Abfragen).

eRqlEngine wurde mittels Java [JAVA-HOME] in der Version 1.4.2 entwickelt und kann frei unter [DBIS_ERQL] oder [WLEKLINSKI] bezogen werden.

8.1 Einbindung

Die Einbindung und Ansprechung von *eRqlEngine* ist mit den folgenden Codezeilen getan. Die Klasse `eRqlEngine` wird instanziiert, anschließend werden eine Datenquelle zugewiesen und die Abfrage durchgeführt:

```
eRqlEngine eRQL = new eRqlEngine();
eworks.RDF.model.Tuples result = null;
try {
     result = eRQL.query(new java.io.File( "C:\input.rdf" ), "Picasso");
} catch(Exception e) {
     System.out.println(e);
}
```

8.2 Programmstruktur

Die Funktionalität von *eRqlEngine* ist auf die folgenden vier Java-Packages verteilt:

- `eworks.eRQL.engine`

  Beinhaltet die Anwendung *eRqlEngine* an sich, d. h. insbesondere auch die Klasse `eRqlEngine`, welcher der Dreh- und Angelpunkt dieser Anwendung ist.

- `eworks.eRQL.gui`

  Beinhaltet eine grafische Benutzungsschnittstelle der Anwendung *eRqlEngine*, insbesondere auch deren Hauptklasse `Gui`.

- `eworks.eRQL.model`

  Beinhaltet alle Klassen, welche für die Repräsentation einer eRQL-Abfrage benötigt werden.

- `eworks.eRQL.parser`

  Beinhaltet alle Klassen, welche für das Parsen und Validieren einer eRQL-Abfrage benötigt werden.

Zusätzlich verwendet *eRqlEngine* die Klasse `edu.stanford.ejalbert.BrowserLauncher`, welche eine Funktion zum Öffnen einer URL in dem clientseitig definierten Standardbrowser zur Verfügung stellt (diese Funktion wird innerhalb der grafischen Benutzungsschnittstelle verwendet).

eRqlEngine verwendet die Klassen von *RqlEngine* (`eworks.RQL.*`), insbesondere auch bzgl. des RDF-Modells (`eworks.RDF.*`).

8.3 Parsen von Abfragen

Das Parsen der Abfragen überlässt *eRqlEngine* dem Parser-Lexer-Gespann *CUP* [CUP] und *JFlex* [JFLEX], der dazu notwendige Programmcode befindet sich innerhalb des Packages `eworks.eRQL.parser`.

Eine ausführliche Gegenüberstellung verfügbarer Parser und Lexer findet sich in [TOLLE]. Neben den dort aufgeführten Argumenten spricht für den Einsatz von *CUP/JFlex* vor allem auch, dass *eRqlEngine, RqlEngine* und *VRP* somit denselben Parser verwenden – in Hinblick auf den eventuellen zukünftigen Einsatz innerhalb von mobilen Agenten (z. B. im Rahmen von *SIMAT* [SIMAT]) kann sich dies vorteilhaft auf die Größe des Programmcodes auswirken.

Die exakte Grammatik von eRQL, sowie die entsprechenden Definitionsdateien für den Lexer *JFlex* und den Parser *CUP* finden sich im Anhang, siehe Anhang D – eRQL Syntax, Anhang E – eRQL Grammatikdefinition für CUP und Anhang F – eRQL Lexerdefinition für JFlex.

8.4 Datenspeicherung und -sammelung in Informationsportalen

Ein Informationsportal stellt dem Besucher den Zugang zu Informationen zur Verfügung. Dazu bietet es dem Benutzer Werkzeuge wie Suchfunktionen, Themenregister, Kategorisierungen und ähnliches an, mittels derer er den Datenbestand durchforsten und die gesuchten Informationen erlangen kann.

Für die Realisierung einer Informationssuche, gleichgültig ob es sich dabei um eine Suchfunktion, ein Themenregister oder eine andere Art von Suche handelt, bieten sich verschiedene Möglichkeiten an, die im Folgenden erläutert werden.

8.4.1 Datenspeicherung

Die Datenspeicherung ist die Art und Weise, wie bzw. wo die Informationen gespeichert sind, auf denen das Informationsportal seine Suche durchführt. Für die Datenspeicherung kommen prinzipiell eine zentrale und eine dezentrale Lösung in Frage:

- **Zentrale Datenspeicherung**

  … findet z. B. bei Internetsuchmaschinen wie Google[38] oder *alltheweb*[39] statt, welche auf riesigen Rechnerfarmen die zur Suche notwendigen Informationen über den Datenbestand vorhalten.

  Großer Vorteil der zentralen Datenspeicherung ist die extrem hohe Geschwindigkeit bei der Datenabfrage, da die zu durchsuchenden Daten für den Abfrageprozessor lokal vorliegen, und nicht zunächst zeitaufwändig über das Netzwerk übertragen werden müssen. Daher verwenden die gängigen Internetsuchmaschinen eine zentrale Datenspeicherung.

- **Dezentrale Datenspeicherung**

  … findet z. B. bei der gewöhnlichen Dateisuche am eigenen Arbeitsplatzrechner statt[40], welche bei Vorhandensein eines Netzwerkes auch entfernte Dateien durchsucht, indem diese Daten zunächst auf das lokale System übertragen, und anschließend durchsucht werden.

  Vorteil der dezentralen Datenspeicherung ist die hohe Aktualität der Daten, da stets aktuelle Daten durchsucht und zurückgegeben werden – anders als bei der zentralen Datenspeicherung, wo auf lokalen Kopien dieser Daten operiert wird. Ein weiterer Vorteil ist der eher niedrige Bedarf an lokalem Speicherplatz, der sich in einfacherer und kostengünstigerer Installation und Wartung solch eines Systems niederschlägt.

8.4.2 Datensammelung

Die Datensammlung ist die Art und Weise, auf welche das Informationsportal an die Informationen gelangt, welche es im Auftrag des Benutzers durchsucht und ggf. auch präsentiert.

Unabhängig von der Datenspeicherung, gleich ob zentral oder dezentral, kommen für die Sammlung der Daten zwei weitere, orthogonale Möglichkeiten in Frage:

- **Datensammlung durch einen zentralen Server**

  … findet beispielsweise sowohl bei Internetsuchmaschinen als auch bei der gewöhnlichen Dateisuche statt, also in beiden, oben aufgeführten Beispielen.

  Der größte Vorteil der serverseitigen Datensammlung dürften die niedrigen Anforderungen an die einzubeziehenden Systeme sein: auf diesen Systemen muss keinerlei Software installiert

[38] http://www.google.com
[39] http://www.alltheweb.com
[40] abgesehen von spezieller Indizierungssoftware wie z. B. dem *Index Server* unter *MS Windows*

werden, lediglich der Lesezugriff auf die zu durchsuchenden Dokumente muss möglich sein (beispielsweise via *HTTP* und *TCP/IP*). Daher verwenden die gängigen Internetsuchmaschinen eine serverseitige Suche, auch „Crawler“ genannt.

- **Datensammlung durch mobile Agenten (mobiler Code)**

  … wie z. B. mit Agentenplattformen wie [LANA] oder [AMETAS] ist derzeit eher die Seltenheit, wird aber in verschiedenen Projekten erprobt, unter anderem in [SIMAT].

  Der Vorteil der Datensammlung durch mobile Agenten ist die Möglichkeit, Daten vor der Versendung über das Netzwerk vorzuverarbeiten. So könnten in einem gefundenen Textdokument beispielsweise Füllwörter entfernt, Verben normalisiert, und binäre Objekte entfernt werden – die Netzwerklast wird dadurch erheblich reduziert.

Die Kombination beider Aspekte ergibt die folgenden Realisierungsmöglichkeiten:

Datensammelung durch… **Informationssuche in…**	**…mobile Agenten:** **+** Datenaufbereitung **– Installation nötig**	**…Serverapplikation:** **+** keine Installation **– hohe Netzlast**
…zentraler Datenbank: schnell ⇒ Online hoher Bedarf an Speicherplatz niedrige Aktualität	???	z. B. Google
…verteilten Datenbanken: hohe Aktualität niedriger Bedarf an Speicherplatz langsam ⇒ Offline	z. B. SIMAT	z. B. Dateisuche

8.4.3 Resümee

Im Rahmen dieser Arbeit habe ich mich dafür entschieden, die Datenspeicherung zentral zu halten.

Der Aspekt der Datensammlung ist nicht Bestandteil dieser Arbeit, daher wird bei der Implementierung davon ausgegangen, dass die Daten bereits in einer zentralen Ablage vorliegen, und nicht zunächst gesammelt werden müssen.

8.5 Optimierungen

Optimierungen sollen die Ausführungsgeschwindigkeit von *eRqlEngine* erhöhen, und/oder den benötigten Speicherplatz reduzieren. Bei den Optimierungen handelt es sich entweder um Opti-

mierungen der RQL-Abfrage, vergleichbar mit der Optimierung einer SQL-Abfrage durch ein RDBMS, oder um generelle Optimierungen des Programmcodes.

Die folgenden Optimierungen sind derzeit in *eRqlEngine* implementiert:

- Die abstrakte Klasse `eworks.eRQL.model.Value` führt die Methode `compact()` ein, welche in abgeleiteten Klassen (z. B. `Disjunction`) überschrieben werden kann, um eine optimierte Variante der Instanz zurückzuliefern, oder den speziellen Wert `null`, wenn auf die Instanz vollständig verzichtet werden kann. Diese Methode liefert – wenn sie nicht überschrieben wird – die Instanz selber zurück (`this`). Abgeleitete Klassen welche eine Teilabfrage beinhalten, sollten den Aufruf dieser Methode entsprechend an die beinhaltete Teilabfrage weiterleiten.

- Die Klasse `eworks.eRQL.model.Junction` (als Basisklasse von `Dicjunction` und `Conjunction`) überschreibt die ererbte Methode `compact()`, welche bei Vorhandensein von mindestens zwei Operanden sich selbst unverändert zurückgibt, bei Vorhandensein nur eines Operanden diesen Operanden zurückgibt, und anderenfalls `null` zurückgibt. Dadurch wird die entsprechende Instanz aus dem Objektmodell der eRQL-Abfrage entfernt, wenn sie nicht unbedingt notwendig ist. Weiterhin reicht diese Klasse zuvor den `compact()`-Aufruf an alle Operanden weiter, welche ggf. ebenfalls eine Optimierung durchführen.

- Die Klasse `eworks.eRQL.engine.eRqlEngine` ruft in der Methode `query()` die `compact()`-Methode der Klasse `eworks.eRQL.model.Query` auf, um die eRQL-Abfrage zu optimieren.

- Im Datenmodell von eRQL (siehe 6.3 Formale Semantik) ist das Ergebnis einer Operation oftmals eine Menge von *Fundstellen*, d. h. eine Menge von Mengen von Aussagen. Diese relativ komplizierte und speicherintensive Struktur wird durch die Klasse `eworks.RDF.model.GroupedTuples` abgebildet.

  Für die folgenden beiden Spezialfälle existieren daher spezialisierte Klassen, welche mit erheblich weniger Speicherplatz auskommen als `GroupedTuples`, da sie keine Listen über die jeweilige Gruppenzugehörigkeit der enthaltenen Tupel verwalten:

  - **Eine einzige Fundstelle mit vielen Aussagen**

    Die Klasse `UngroupedTuples` repräsentiert eine Fundstelle mit beliebig vielen Aussagen, oder, wie der Name nahe legt: eine Menge nicht gruppierter Tupel.

  - **Mehrere Fundstellen mit jeweils einer einzigen Aussage**

    Die Klasse `SingleGroupedTuples` repräsentiert eine Menge von Fundstellen mit jeweils genau einer Aussage, oder, wie der Name nahe legt: eine Menge von Tupeln, derer jeder eine eigenständige Gruppe darstellt.

Die Implementierung dieser Klasse ist weiterhin dadurch optimiert, dass die Schnittstelle `Tuple` die Schnittstelle `Tuples` erweitert, und somit die implementierenden Klassen `TupleImpl` und `Statement` ebenfalls als `Tuples` angesprochen werden können (d. h. ein Tupel bzw. eine Aussage kann programmatisch auch als eine Menge einer Fundstelle, welche aus eben jenem Tupel bzw. jener Aussage besteht, angesprochen werden).

Alle drei Klassen `GroupedTuples`, `UngroupedTuples` und `SingleGroupedTuples` sind von der abstrakten Klasse `AbstractTuples` abgeleitet, und implementieren die Schnittstelle `Tuples`. So verfügen alle drei Klassen über dieselbe Schnittstelle, und sollten (bis auf Ausnahmen) gegen das Interface `Tuples` angesprochen werden, um so transparent im Sinne der *Polymorphie*[41] verwendet werden können.

- Die Klasse `eworks.eRQL.model.Conjunction` bricht die Auswertung ihrer Operanden ab, und gibt eine leere Menge von Aussagen zurück, wenn einer der Operanden nicht mindestens eine Fundstelle ergibt.

- Die Klassen `eworks.eRQL.model.DocumentQuery` und `StatementQuery` überschreiben die ererbte Methode `compact()`, und überprüfen, ob die jeweilige Instanz als Wert ebenfalls eine Instanz von `DocumentQuery` bzw. `StatementQuery` enthält. Wenn dieser Fall eintritt, wird die enthaltene Instanz zurückgegeben, denn die mehrfache Anwendung des Dokumentmodus- bzw. Aussagemodus-Operators erzielt dieselbe Wirkung wie die einfache Anwendung.

- Die Klasse `eworks.eRQL.model.DocumentQuery` bricht die Ausführung der Methode `query()` ab, sofern sämtliche Aussagen des Dokumentes der Rückgabe hinzugefügt worden sind, da der gegenwärtige Stand von *eRqlEngine* nicht mehr als ein Dokument als Datenquelle zulässt. Weiterhin gibt diese Methode eine Instanz von `UngroupedTuples` (statt `GroupedTuples`) zurück, um Speicherplatz einzusparen (denn die Rückgabe ist derzeit stets nur eine einzige Fundstelle mit beliebig vielen Aussagen).

- Die `query()`-Methode der Klasse `eworks.eRQL.model.Literal` unterstützt eine optimierte Abfragefunktion für *Points Of Interest*, siehe auch 7.8.2 Optimierung des POI-Modus'. Diese optimierte Abfragefunktion ist aber aus den in 7.8.2 genannten Gründen auskommentiert. Weiterhin gibt diese Methode eine Instanz von `SingleGroupedTuples` (statt `GroupedTuples`) zurück, um Speicherplatz einzusparen (denn die Rückgabe ist stets eine Menge von Fundstellen mit exakt einer enthaltenen Aussage).

- Die `query()`-Methode der Klasse `eworks.eRQL.model.PoiQuery` speichert die Umgebungen von Literalen und Ressourcen zwischen, um sie bei weiteren Vorkommen innerhalb derselben Abfrage nicht erneut durch den RQL-Prozessor ermitteln zu müssen („Caching").

[41] http://de.wikipedia.org/wiki/Polymorphie

- Die `query()`-Methode der Klasse `eworks.eRQL.model.StatementQuery` gibt eine Instanz von `SingleGroupedTuples` (statt `GroupedTuples`) zurück, um Speicherplatz einzusparen (denn die Rückgabe ist stets eine Menge von Fundstellen mit exakt einer enthaltenen Aussage).

8.6 Screenshots

In diesem Abschnitt werden Screenshots (d. h. Bildschirmfotos) von *eRqlEngine* gezeigt. Dies ist möglich, da *eRqlEngine* über eine grafische Benutzungsschnittstelle verfügt, siehe 8.2 Programmstruktur.

Die Bedienung der Anwendung wird durch englischsprachige Beschriftungen beschrieben und sollte selbsterklärend sein. Es ist zunächst eine Quelldatei mit RDF-Daten auszuwählen, anschließend eine eRQL-Abfrage einzugeben, und auf „Execute!“ zu klicken. Das Ergebnis der Abfrage wird in dem Textfeld im unteren Bereich der Anwendung präsentiert. Alternativ kann durch Klicken auf „RQL“ in den RQL-Modus gewechselt werden, in welchem direkt RQL-Abfragen eingegeben werden können, siehe 9.7 Screenshots. Die Anwendung merkt sich sämtliche Einstellungen bei Programmbeendigung und stellt diese bei erneutem Programmstart wieder her.

Nach Durchführung der Abfrage `~~~ pablo` präsentiert sich *eRqlEngine* wie folgt:

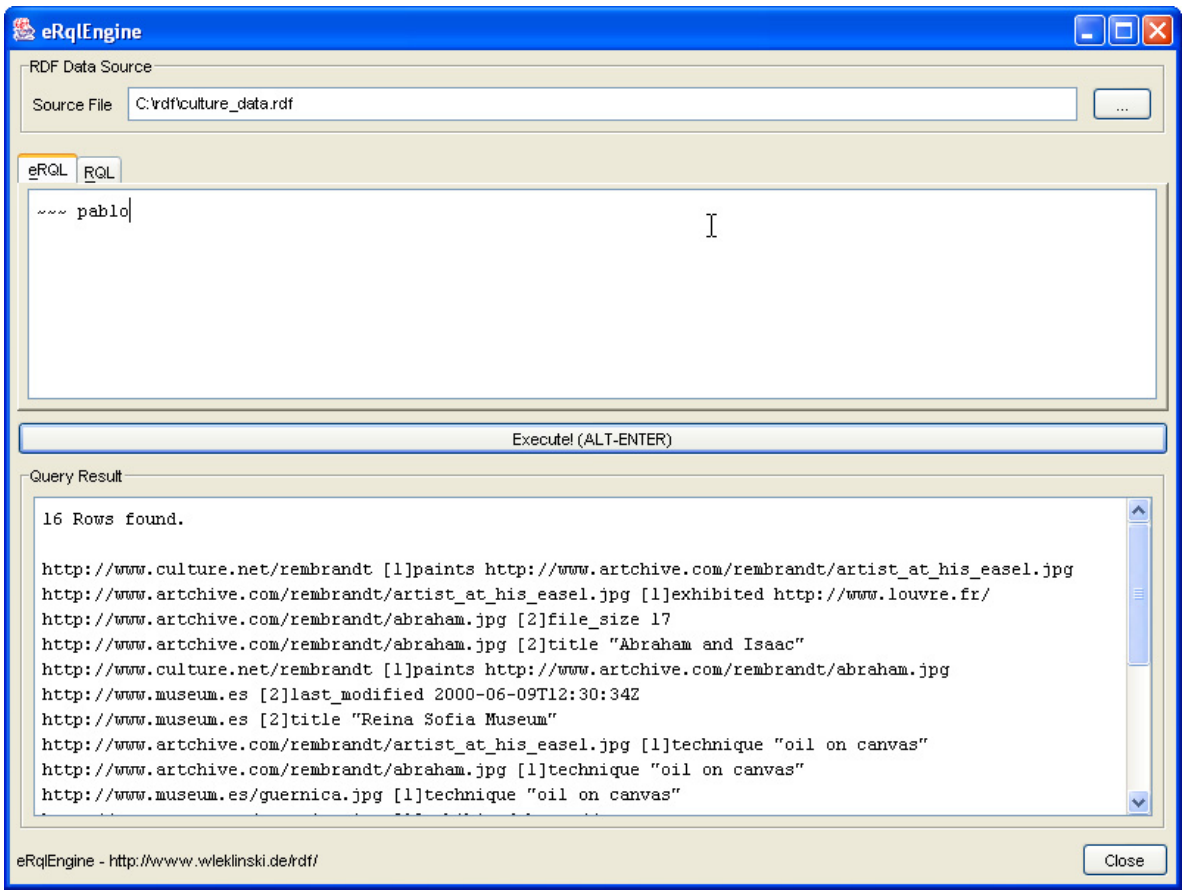

Abbildung 11 – Screenshot eRqlEngine (1)

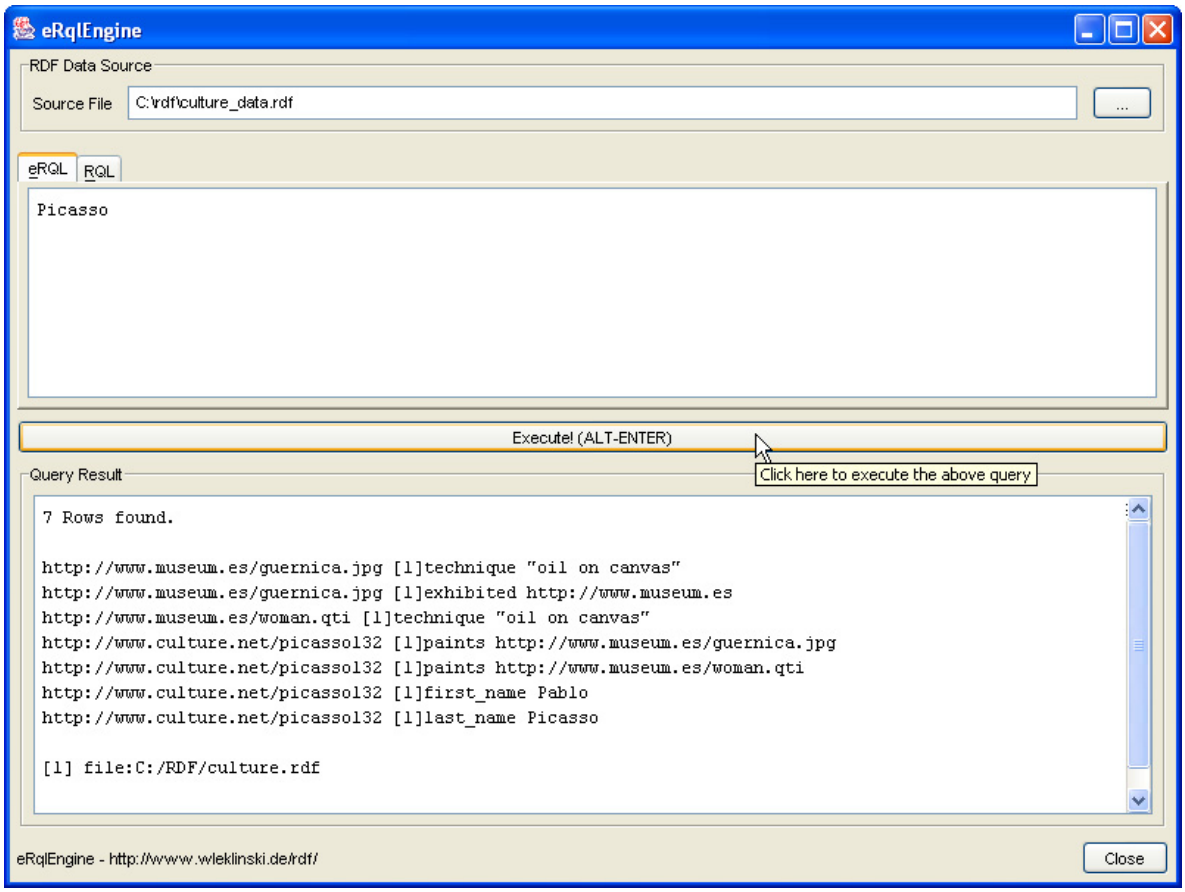

Abbildung 12 – Screenshot eRqlEngine (2)

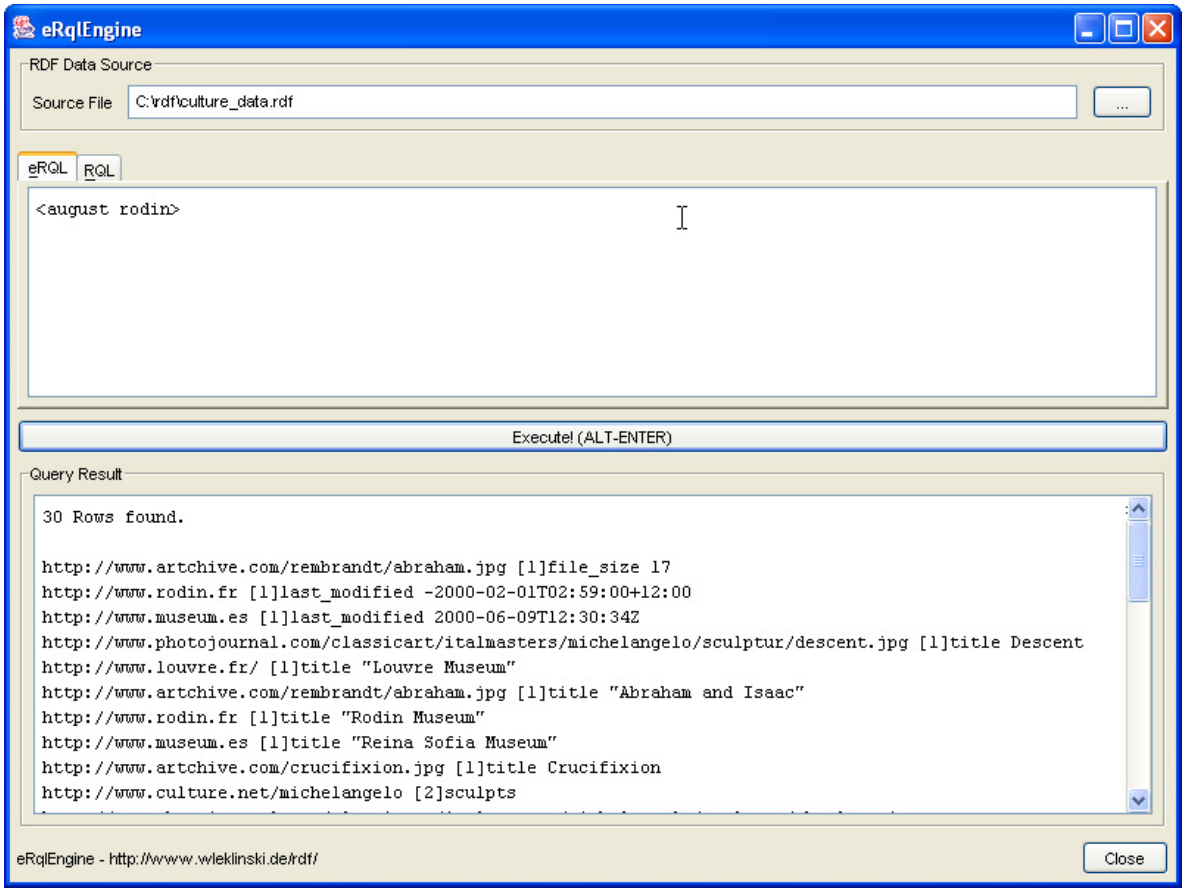

Abbildung 13 – Screenshot eRqlEngine (3)

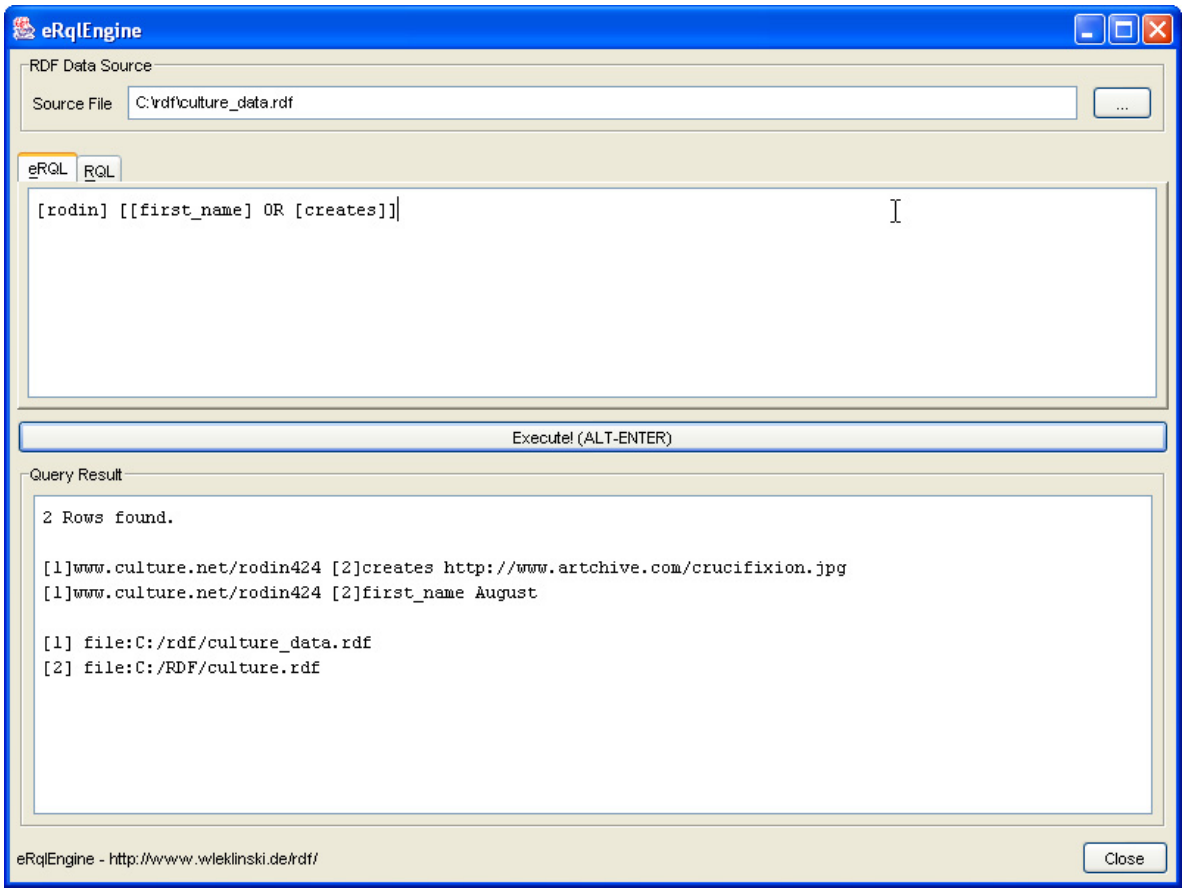

Abbildung 14 – Screenshot eRqlEngine (4)

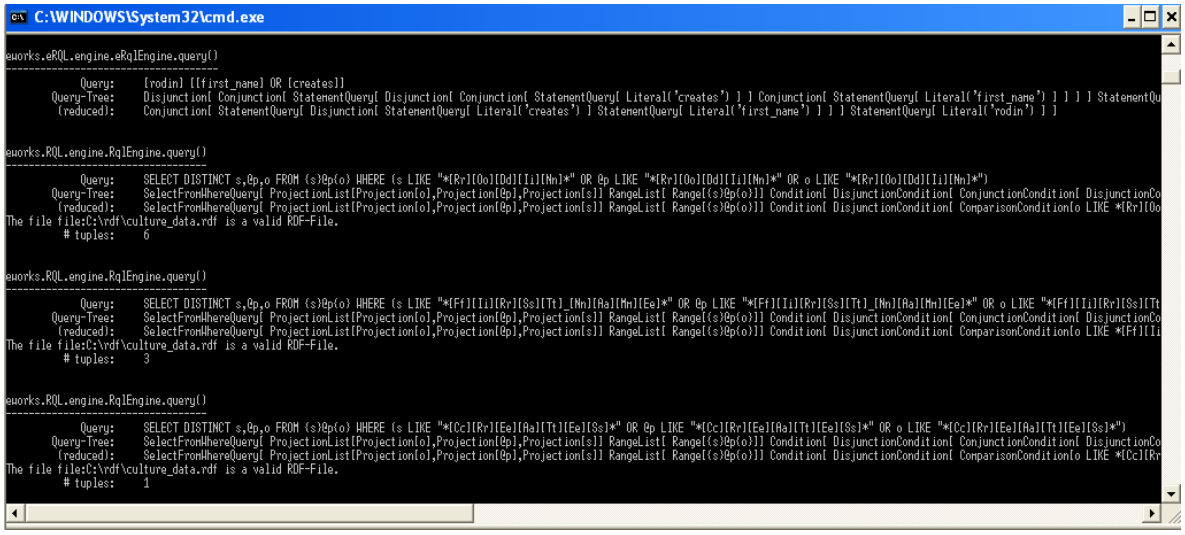

Abbildung 15 – Screenshot eRqlEngine RQL-Abfragen

9 RqlEngine – ein RQL-Prozessor

RqlEngine ist die rudimentäre Implementierung eines Prozessors für die RDF-Abfragesprache RQL [RQL-OVW], welche im Rahmen dieser Arbeit angefertigt wurde.

Der Schwerpunkt der Implementierung von *RqlEngine* liegt nicht darauf, einen möglichst vollwertigen RQL-Prozessor zu implementieren, sondern vielmehr auf der der Unterstützung der minimalen Teilmenge an Funktionen, welche für die Auswertung von eRQL-Abfragen benötigt werden (siehe 7 Umwandlung von eRQL- in RQL-Abfragen, 8 eRqlEngine – ein eRQL-Prozessor).

RqlEngine basiert für den Zugriff auf RDF-Dokumente auf dem RDF-Parser *VRP* [VRP-HOME]. Prinzipiell kann jedoch ein beliebiger RDF-Parser verwendet werden, sofern eine entsprechende Schnittstelle implementiert wird. *RqlEngine* wurde mittels Java [JAVA-HOME] in der Version 1.4.2 entwickelt und kann frei unter [DBIS_ERQL] oder [WLEKLINSKI] bezogen werden.

9.1 Einbindung

Die Einbindung und Ansprechung von *RqlEngine* ist mit den folgenden Codezeilen getan. Die Klasse `RqlEngine` wird instanziiert, anschließend werden eine Datenquelle zugewiesen und die Abfrage durchgeführt:

```
RqlEngine RQL = new RqlEngine();
eworks.RDF.model.Tuples result = null;
try {
     RQL.setDataSource(new java.io.File("C:\input.rdf"));
     result=this.RQL.query("SELECT s,@p,o FROM {s}@p{o}");
} catch(Exception e) {
     System.out.println(e);
}
```

9.2 Programmstruktur

Die Funktionalität von *RqlEngine* ist auf die folgenden fünf Java-Packages verteilt:

- `eworks.RDF.model`

  Beinhaltet alle Klassen, welche für die Repräsentation eines RDF-Modells benötigt werden. Wird auch von der Anwendung *eRqlEngine* benötigt, siehe 8 eRqlEngine – ein eRQL-Prozessor.

- `eworks.RDF.util`

  Beinhaltet zusätzliche Klassen für die Arbeit mit RDF-Modellen, welche nicht unmittelbar der Repräsentation eines RDF-Modells dienen. Wird auch von der Anwendung *eRqlEngine* benötigt, siehe 8 eRqlEngine – ein eRQL-Prozessor.

- `eworks.RQL.engine`

  Beinhaltet die Anwendung *RqlEngine* an sich, d. h. insbesondere auch die Klasse `RqlEngine`, welcher der Dreh- und Angelpunkt dieser Anwendung ist.

- `eworks.RQL.model`

  Beinhaltet alle Klassen, welche für die Repräsentation einer RQL-Abfrage benötigt werden.

- `eworks.RQL.parser`

  Beinhaltet alle Klassen, welche für das Parsen und Validieren einer RQL-Abfrage benötigt werden.

9.3 Konformität mit der RQL-Spezifikation

In diesem Abschnitt werden die Abweichungen der *RqlEngine*-Implementierung von der RQL-Spezifikation [RQL-FUNC] beschrieben. Bei der im Rahmen dieser Arbeit entstandenen RQL-Implementierung handelt es sich nicht um eine RQL-Implementierung im eigentlichen Sinne – zu zahlreich sind die Abweichungen von der RQL-Spezifikation [RQL-FUNC]. Absicht war auch nicht die Erstellung eines universellen RQL-Prozessors, sondern alleine die Demonstration, dass eRQL-Abfragen letztlich tatsächlich in RQL-Abfragen umgewandelt, und durch einen RQL-Prozessor ausgewertet werden können:

Da derzeit nur eine vollständige RQL-Implementierung existiert [RQL-Demo], und diese nicht die Java-Plattform unterstützt (sondern C++ verwendet), habe ich mich für eine minimale RQL-Implementierung entschieden, welche exakt die Teilmenge von RQL unterstützt, welche für die Auswertung von eRQL-Abfragen relevant sind.

Im Folgenden die Aufzählung aller RQL-Eigenschaften, welche von *RqlEngine* unterstützt bzw. nicht unterstützt werden:

Durch RqlEngine unterstützte RQL-Funktionen

- SELECT-FROM-WHERE-Abfragen (`SELECT … FROM … WHERE …`)
- `DISTINCT`-Operator zur Unterdrückung doppelter Zeilen
- Abfragen in Kurzschreibweise (`Sculptor`)
- Variablenbindung an Daten-, Klassen und Prädikatvariablen
- Pfadausdruck zur Ermittlung von vollständigen Aussagen (`{s}@p{o}`)

- Pfadausdruck zur Ermittlung von Instanzen einer Klasse (`Sculptor{C}`)
- `AND`- und `OR`-Operator für die Verknüpfung von Selektionsbedingungen

Durch *RqlEngine* nicht unterstützte RQL-Funktionen

- Verschiedene Pfadausdrücke
- Verkettung von Pfadausrücken (`{s}@p{o}.{a}@b{c}`)
- Schemanavigation
- Funktionen (`domain()`, `range()`, `topclass()`, `subPropertyOf()`, …)
- Namensräume (`using …`)
- Verschachtelte Abfragen („nested queries“)
- Mengenoperationen (`union`, `intersect`, `minus`)
- Quantoren (`exists…in…such`, `forall…in…such`)
- Konstruktoren (`bag`, `seq`)
- Keine Erzeugung eines neuen RDF-Modells für die Rückgabe

9.4 Parsen von Abfragen

Das Parsen der Abfragen überlässt *RqlEngine* genau wie *eRqlEngine* dem Parser-Lexer-Gespann *CUP* [CUP] und *JFlex* [JFLEX], der dazu notwendige Programmcode befindet sich innerhalb des Packages `eworks.RQL.parser`.

Eine ausführliche Gegenüberstellung verfügbarer Parser und Lexer findet sich in [TOLLE]. Neben den dort aufgeführten Argumenten spricht für den Einsatz von *CUP*/*JFlex* vor allem auch, dass *RqlEngine* und *VRP* somit denselben Parser verwenden – in Hinblick auf den eventuellen, zukünftigen Einsatz innerhalb von mobilen Agenten (z. B. im Rahmen von *SIMAT* [SIMAT]) kann sich dies vorteilhaft auf die Größe des Programmcodes auswirken.

Die exakte Grammatik der unterstützten Teilmenge von RQL, sowie die entsprechenden Definitionsdateien für den Lexer *JFlex* und den Parser *CUP* sind im Anhang befindlich, siehe Anhang A – RQL-Syntax, Anhang B – RQL Grammatikdefinition für CUP und Anhang C – RQL Lexerdefinition für JFlex.

9.5 Optimierungen

Optimierungen sollen die Ausführungsgeschwindigkeit von *RqlEngine* erhöhen, und/oder den benötigten Speicherplatz reduzieren. Bei den Optimierungen handelt es sich entweder um Optimierungen der RQL-Abfrage, vergleichbar mit der Optimierung einer SQL-Abfrage durch ein RDBMS, oder um generelle Optimierungen des Programmcodes.

Die folgenden Optimierungen sind derzeit in *RqlEngine* implementiert:

- Die Klasse `eworks.RQL.model.AlikenessCompareOperator` verwendet eine `Hashtable`, um kompilierte Reguläre Ausdrücke für weitere Verwendungen zwischenzuspeichern, und eine erneute Kompilierung desselben Ausdrucks zu vermeiden („Caching").

- Die abstrakte Klasse `eworks.RQL.model.Condition` führt die Methode `compact()` ein, welche in abgeleiteten Klassen (z. B. `CompositionCondition`) überschrieben werden kann, um eine optimierte Variante der Instanz zurückzuliefern, oder den speziellen Wert `null`, wenn auf die Instanz vollständig verzichtet werden kann. Diese Methode liefert – wenn sie nicht überschrieben wird – die Instanz selber zurück (`this`). Abgeleitete Klassen welche eine Teilabfrage beinhalten, sollten den Aufruf dieser Methode entsprechend an die beinhaltete Teilabfrage weiterleiten.

- Die Klasse `eworks.RQL.model.CompositionCondition` (als Basisklasse von beispielsweise `DisjunctionCondition` und `ConjunctionCondition`) überschreibt die ererbte Methode `compact()`, welche bei Vorhandensein von mindestens zwei Operanden sich selber unverändert zurückgibt, bei Vorhandensein nur eines Operanden diesen Operanden zurückgibt, und anderenfalls `null` zurückgibt. Dadurch wird die entsprechende Instanz aus dem Objektmodell der RQL-Abfrage entfernt, wenn sie nicht unbedingt notwendig ist. Weiterhin reicht diese Klasse zuvor den `compact()`-Aufruf an alle Operanden weiter, welche ggf. ebenfalls eine Optimierung durchführen.

- Die Klasse `eworks.RQL.model.DisjunctionCondition` bricht die Auswertung in der Methode `matches()` ab, sobald einer der Operanden zu *wahr* ausgewertet wird. Entsprechend bricht die Klasse `eworks.RQL.model.ConjunctionCondition` die Auswertung in der Methode `matches()` ab, sobald einer der Operanden zu *falsch* ausgewertet wird.

- Die Klasse `eworks.RQL.engine.RqlEngine` ruft in der Methode `query()` die `compact()`-Methode der Klasse `eworks.RQL.model.Query` auf, um die RQL-Abfrage zu optimieren.

- Die Klasse `eworks.RDF.util.UriFactory` stellt einen Zwischenspeicher für erzeugte Instanzen der Klasse `java.net.URI` gemäß des *Factory-Musters*[42] bereit. Dazu wird der statischen Methode `getURI()` eine URI als Literal übergeben, welches von der Methode zunächst normalisiert wird. Anschließend erstellt die Methode ggf. eine entsprechende `URI`-Instanz, legt diese im Zwischenspeicher ab, und gibt sie zurück. Im Rahmen der Normalisierung werden Leerzeichen in „%20" umgewandelt und Vorkommen von \ durch / ersetzt.

- RQL arbeitet im Allgemeinen weniger mit Aussagen (d. h. Tripeln), als vielmehr mit n-Tupeln. Diese werden in der *RqlEngine* durch die Schnittstelle `eworks.RDF.model.Tuple` abgebildet, welche implementiert wird durch die Klasse `eworks.RDF.model.TupleImpl`. Da ein Großteil der verarbeiteten Tupel dennoch Tri-

[42] http://en.wikipedia.org/wiki/Factory_method_pattern

pel sind, existiert die spezialisierte Klasse `eworks.RDF.model.Statement`, welche für den Spezialfall von 3-Tupeln eine ressourcenschonendere Implementierung der `Tuple`-Schnittstelle als `TupleImpl` darstellt.

- Die Klasse `eworks.RDF.model.UriLiteralValue` verwendet die statische Klasse `eworks.RDF.util.UriFactory` zur Erzeugung von `java.net.URI`-Instanzen für die URIs und Namensräume der verschiedenen RDF-Ressourcen.

9.6 Auswertung von Abfragen mittels VRP

RqlEngine basiert für den Zugriff auf RDF-Dokumente auf dem validierenden RDF-Parser *VRP* [VRP-HOME]. Prinzipiell kann jedoch ein beliebiger RDF-Parser verwendet werden, sofern eine entsprechende Schnittstelle implementiert wird. Die von *RqlEngine* benötigten Funktionen stellt VRP jedoch nicht direkt zur Verfügung, so dass diese Funktionen durch *RqlEngine* nachgeahmt und auf VRP-Funktionen abgebildet werden. In diesem Abschnitt werden zunächst die durch VRP unterstützten Funktionen, anschließend die nicht unterstützten Funktionen aufgelistet, jeweils mit Hinweisen zur Implementierung in *RqlEngine* versehen.

9.6.1 Durch VRP unterstützte Abfrageoperationen

Die folgende Tabelle listet elementare Abfrageoperationen auf, wie sie für die Durchführung von RQL-Abfragen notwendig sind, und stellt die derzeitige Unterstützung durch den RDF-Parser *VRP* in der Version 2.5 dar. *VRP* und diesbezügliches Informationsmaterial sind unter [VRP-HOME] erhältlich.

Beschreibung	VRP Funktion(en)
Überprüfung auf Beinhaltung einer URI	`contains()`
Überprüfung auf Beinhaltung einer Aussage	`contains()`
Abfrage aller Metaklassen	`getAllMetaClasses()`
Abfrage aller Metaklassen von Klassen	`getMetaClasses()`
Abfrage aller Metaklassen von Prädikaten	`getMetaProperties()`
Abfrage aller Klassen	`getClasses()`
Abfrage aller Container	`getContainers()`
Abfrage aller Ressourcen der Datenebene	`getDataRessources()`
Abfrage aller Knoten	`getNodes()`
Abfrage aller Prädikate	`getProperties()`

Beschreibung	VRP Funktion(en)
Abfrage aller „vergegenständlichten“ Aussagen (siehe Seite 22)	`getReifiedStatements()`
Abfrage aller Aussagen	`getStatements()`
Abfragen der URIs aller Ressourcen.	`getNodes()`[43]
Abfrage aller Instanzen einer Klasse (sowie optional auch alle Instanzen davon abgeleiteter Klassen)	./. (siehe Folgeabschnitt)
Abfrage aller Verwendungen eines Prädikates (sowie optional auch die Verwendungen dessen Nachfahren)	./. (siehe Folgeabschnitt)
Abfrage aller Klassen, welche direkt (oder indirekt) von einer anderen Klasse abgeleitet sind	./. (siehe Folgeabschnitt)
Abfrage aller Prädikate, welche direkt (oder indirekt) von einem anderen Prädikat abgeleitet sind	./. (siehe Folgeabschnitt)

9.6.2 Durch VRP nicht unterstützte Abfrageoperationen

Während im vorangegangenen Abschnitt die Funktionen betrachtet wurden, welche der RDF-Parser VRP zur Verfügung stellt, geht es in diesem Abschnitt speziell um die Nachbildung derjenigen Funktionen, welche zur Realisierung eines RQL-Prozessors fehlen.

Weiterhin wird in diesem Abschnitt für jede der fehlenden Funktionen eine Schnittstelle vorgeschlagen, welche im Rahmen einer Abstraktionsschicht für den RDF-Parser implementiert werden könnte.

9.6.2.1 Instanzen einer Klasse

Es gilt, alle Ressourcen zu finden, welche von einer gegebenen Schema- oder Metaschemaklasse mit dem Namen *X* instanziiert sind, d. h. derer Instanzen zu ermitteln. Optional sind ebenfalls alle Instanzen der davon abgeleiteten Klassen zu ermitteln. Die folgende Vorgehensweise führt zu diesem Ziel und ist innerhalb von `eworks.RQL.model.InstancesOfClassRange` implementiert:

1. **Klasseneigenschaft sicherstellen und URI(s) ermitteln**

   Mittels der VRP-Funktion `getClasses()` wird die Liste aller Klassen ermittelt, was nicht erst zum Zeitpunkt der Auswertung geschieht (`nextHit()`), sondern bereits während der Initialisierung (`reset()`). Anschließend werden daraus all jene Klassen übernommen, deren URIs mit dem Namen *X* enden (um anführende Namensräume zu ignorieren) – alle anderen Klassen werden für den folgenden Schritt separat aufbewahrt.

[43] Am ehesten durch `getNodes()` ermittelbar, dann werden allerdings auch alle Literale zurückgegeben.

2. **URI(s) der abgeleiteten Klassen ermitteln**

   Die in dem vorangegangenen Schritt aufbewahrten Klassen werden durchlaufen. Dabei werden alle Klassen, welche mittels `rdfs:subClassOf` mit einer der bereits übernommenen Klassen verbunden sind, ebenfalls übernommen, und gleichzeitig aus der Menge der aufbewahrten Klassen entfernt.

   Dieser Schritt findet ebenfalls während der Initialisierung statt, und wird so oft wiederholt, solange während jeder Wiederholung mindestens eine Klasse übernommen wird.

3. **Suche nach Instanzen**

   Zum Zeitpunkt der Auswertung (innerhalb von `nextHit()`) wird mittels `getDataResources()` durch die Liste aller Ressourcen des Modells iteriert. Alle Ressourcen, welche mittels `rdf:type` mit einer RDFS-Klasse verbunden sind, die im vorangegangenen Schritt übernommen worden ist, werden zurückgegeben.

Schnittstelle für Parserabstraktion

Die folgenden Methoden werden zur Implementierung benötigt:

- `getInstancesOfSchemaProperty()`
- `getInstancesOfMetaProperty()`
- `getInstancesOfProperty()`

  (als Zusammenfassung der beiden vorangegangenen Methoden)

- `getInstancesOfSchemaClass()`
- `getInstancesOfMetaClass()`
- `getInstancesOfClass()`

  (als Zusammenfassung der beiden vorangegangenen Methoden)

- `getInstancesOf()`

  Diese Methode ist eine Zusammenfassung aller vorangegangenen Methoden und ermöglicht es, Instanzen beliebiger Ressourcen zu ermitteln, sofern vorhanden.

Neben der oben beschriebenen Grundform sind die folgenden Variationen, welche durch „Überladung“ der entsprechenden Funktion implementiert werden können:

- Suche nach den Instanzen von mehr als einer Klasse
- Suche ausschließlich nach den „expliziten“ Instanzen einer Klasse
- Suche ausschließlich nach den Instanzen einer Klasse, und davon abgeleiteten Klassen bis zur Tiefe *n*

9.6.2.2 Ableitungen einer Klasse

Es gilt, alle Klassen zu finden, die von einer gegebenen Klasse mit Namen *X* abgeleitet sind (dieses Vorgehen ist derzeit nicht in *RqlEngine* implementiert, ist jedoch nötig, um weitere eRQL-

Funktionen implementieren zu können). Im Folgenden wird eine Vorgehensweise algorithmisch beschrieben, welche zu diesem Ziel führt:

1. **Klasseneigenschaft sicherstellen und URI(s) ermitteln**

   Mittels der VRP-Funktion `getClasses()` wird die Liste aller Klassen ermittelt, was nicht erst zum Zeitpunkt der Auswertung geschehen muss, sondern bereits während der Initialisierung geschehen kann. Anschließend werden daraus all jene Klassen übernommen, deren URIs mit dem Namen *X* enden (um anführende Namensräume zu ignorieren) – alle anderen Klassen werden für den folgenden Schritt separat aufbewahrt.

2. **URI(s) der abgeleiteten Klassen ermitteln**

   Die in dem vorangegangenen Schritt aufbewahrten Klassen werden durchlaufen. Dabei werden alle Klassen, welche mittels `rdfs:subClassOf` mit einer der bereits übernommenen Klassen verbunden sind, ebenfalls übernommen, und gleichzeitig aus der Menge der aufbewahrten Klassen entfernt.

   Dieser Schritt kann ebenfalls während der Initialisierung stattfinden, und so oft wiederholt werden, solange während jeder Wiederholung mindestens eine Klasse übernommen wird.

Schnittstelle für Parserabstraktion

Die folgenden Methoden werden zur Implementierung benötigt:

- `getSubClassesOfSchemaClass()`
- `getSubClassesOfMetaClass()`
- `getSubClassesOf()`
  (als Zusammenfassung der vorangegangenen Methoden)

Umkehrfunktionen

Zu jeder `getSubClasses…`-Methode ist eine entsprechende `getSuperClasses…`-Methode nützlich.

Variationen

Neben der oben beschriebenen Grundform sind die folgenden Variationen denkbar, welche durch „Überladung“ der entsprechenden Funktion implementiert werden können:

- Suche nach den Ableitungen von mehr als einer Klasse
- Suche ausschließlich nach den direkten Ableitungen einer Klasse
- Suche ausschließlich nach den Ableitungen einer Klasse bis zur Tiefe *n*

9.6.2.3 Ableitungen eines Prädikates

Es gilt, alle Prädikate zu finden, welche von einem gegebenen Prädikat mit dem Namen *X* abgeleitet sind (dieses Vorgehen ist derzeit nicht in *RqlEngine* implementiert, ist jedoch nötig, um wei-

tere eRQL-Funktionen implementieren zu können.) Im Folgenden wird eine Vorgehensweise algorithmisch beschrieben, welche zu diesem Ziel führt:

Alle Aussagen des Modells sind daraufhin zu untersuchen, ob ihr Prädikat `rdfs:subPropertyOf` lautet und ihr Objekt entweder dem gegebenen Prädikat, oder einem der bereits gefundenen Prädikaten entspricht. Für jede Fundstelle handelt es sich beim Subjekt der entsprechenden Aussage um eines der gesuchten Prädikate, welches in die Rückgabemenge übernommen werden muss. Dieser Schritt ist zu wiederholen, bis keine weiteren Prädikate mehr gefunden werden.

Schnittstelle für Parserabstraktion:

Die folgenden Methoden werden zur Implementierung benötigt:

- `getSubPropertiesOfSchemaProperty()`
- `getSubPropertiesOfMetaProperty()`
- `getSubPropertiesOf()`
  (als Zusammenfassung der vorangegangenen Methoden)

Umkehrfunktionen

Zu jeder `getSubProperties`...-Methode ist auch eine entsprechende Methode `getSuperProperties`... nützlich.

Variationen

Neben der oben beschriebenen Grundform sind die folgenden Variationen denkbar, welche durch „Überladung“ der entsprechenden Funktion implementiert werden können:

- Suche nach den Ableitungen von mehr als einem Prädikat
- Suche ausschließlich nach den direkten Ableitungen eines Prädikates
- Suche ausschließlich nach den Ableitungen eines Prädikates bis zur Tiefe *n*

9.7 Screenshots

In diesem Abschnitt werden Screenshots (d. h. Bildschirmfotos) von *RqlEngine* gezeigt. Dies ist möglich, da *RqlEngine* über eine grafische Benutzungsschnittstelle verfügt.

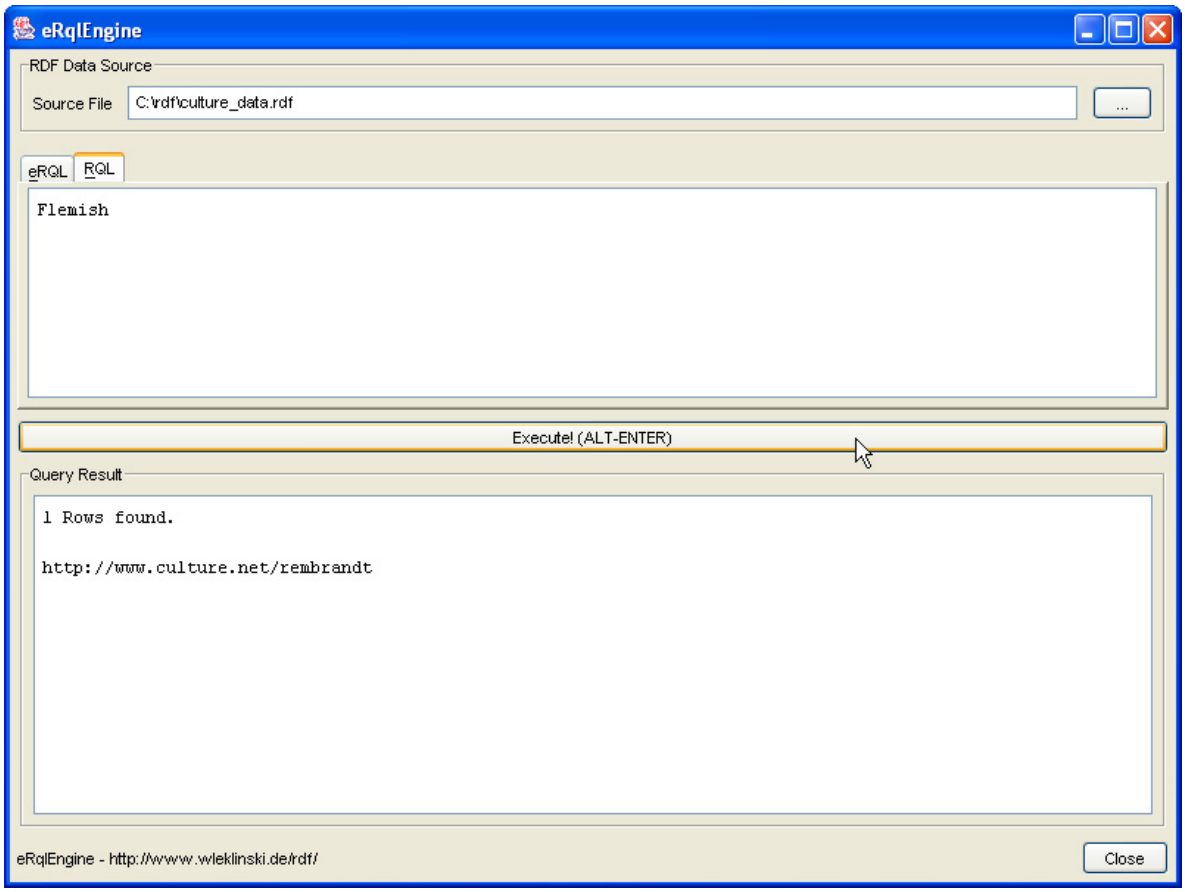

Abbildung 16 – Screenshot RqlEngine (1)

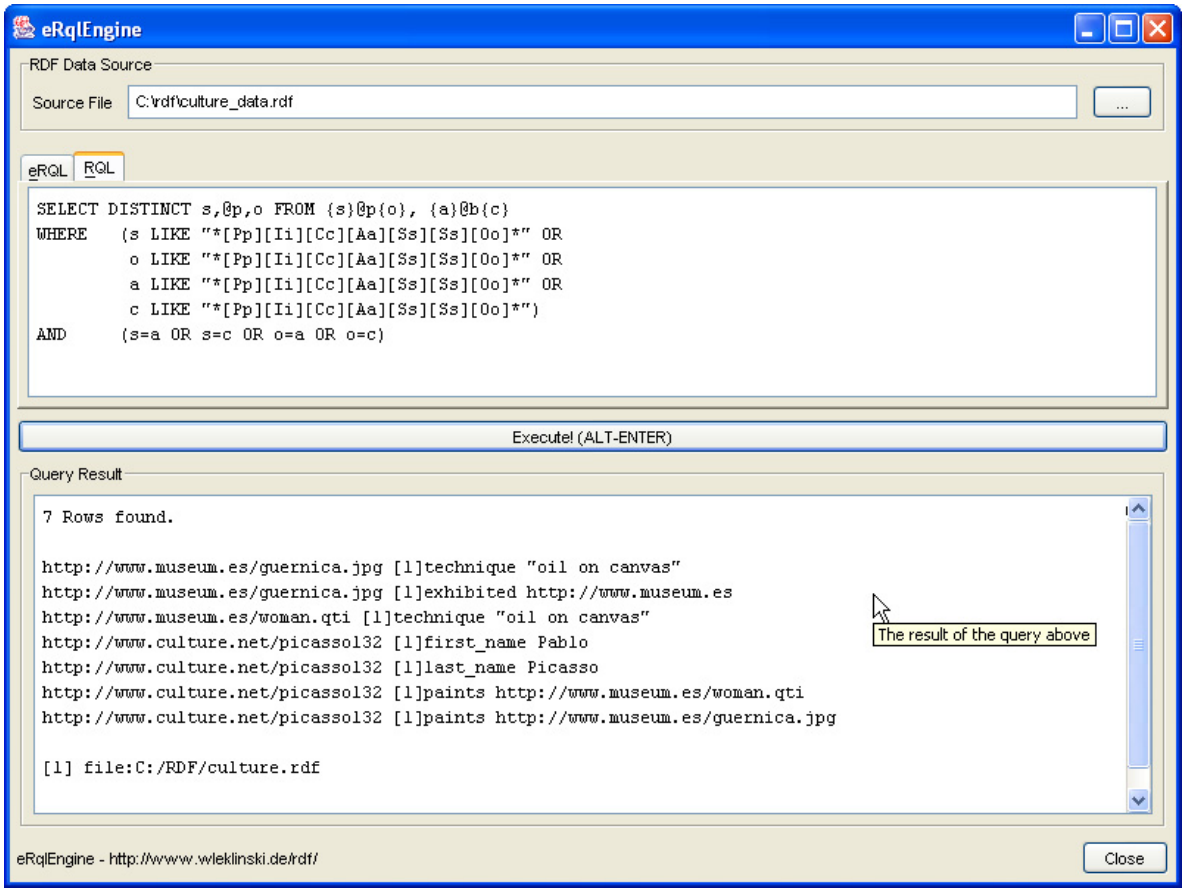

Abbildung 17 – Screenshot RqlEngine (2)

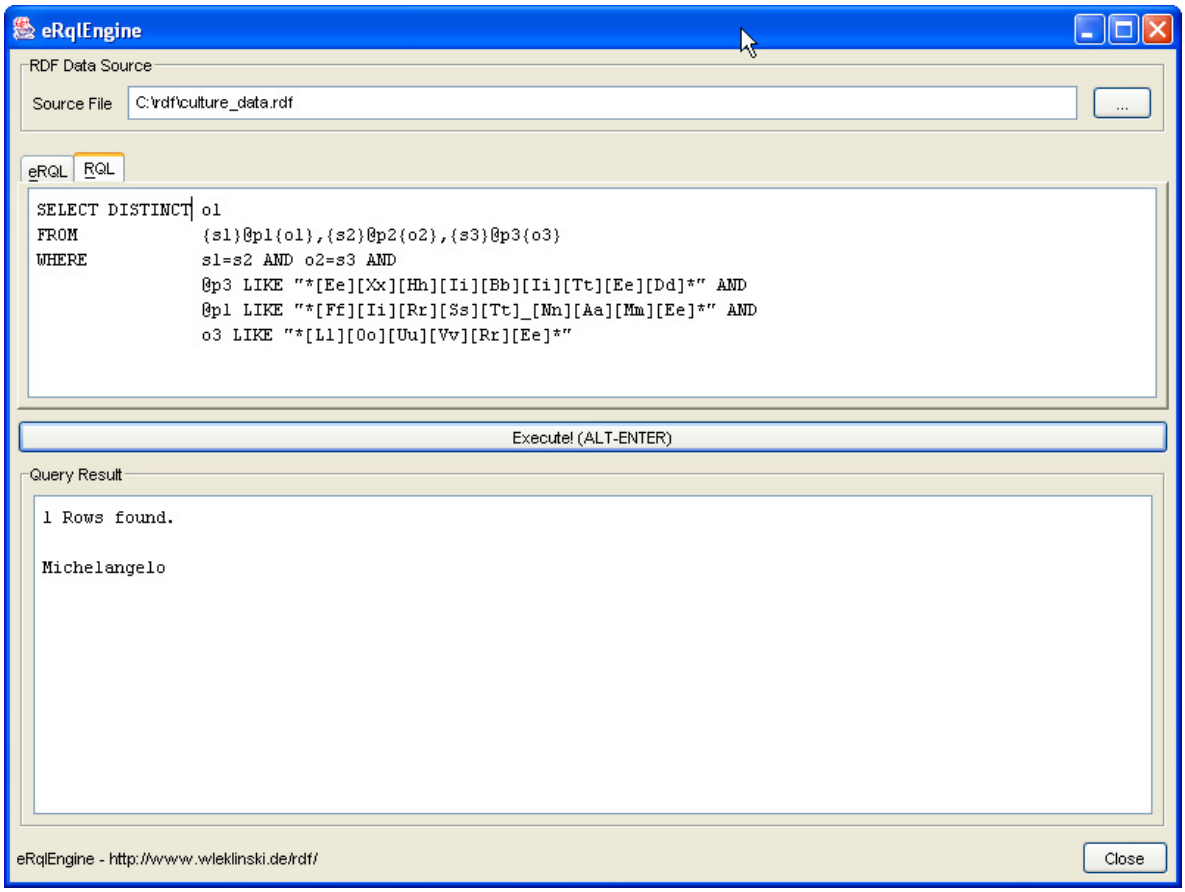

Abbildung 18 – Screenshot RqlEngine (3)

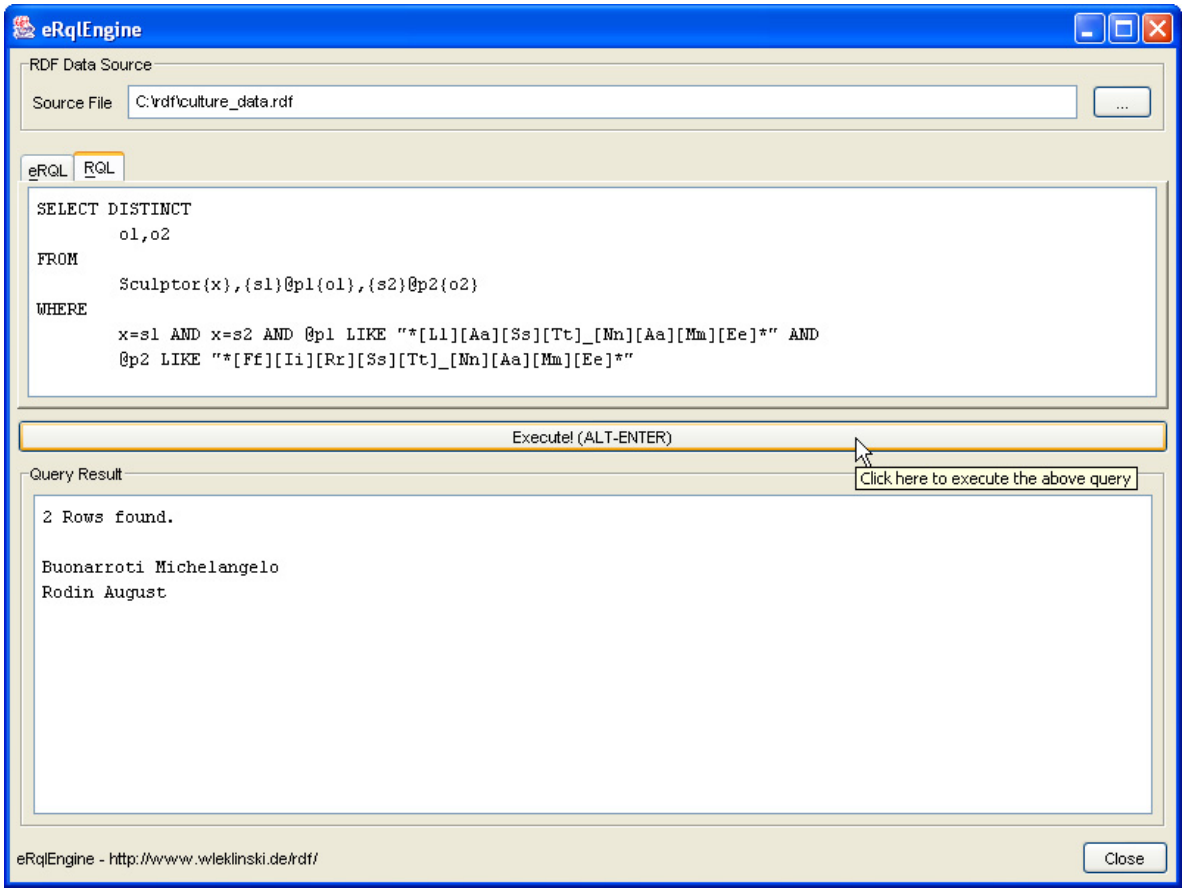

Abbildung 19 – Screenshot RqlEngine (4)

10 Ausblick

10.1 Offene Fragen

In diesem Abschnitt sind offene Aspekte, sowie Kritik und weitere Entwicklungsmöglichkeiten rund um eRQL, *eRqlEngine* und *RqlEngine* zusammengetragen, welche während der Erarbeitung oder Entwicklung entstanden sind. Einige dieser Aspekte wurden bewusst offen gelassen, um die weitere Entwicklung von eRQL nicht durch eine zu frühe Designentscheidung negativ zu beeinflussen.

10.1.1 Points Of Interest und Anonyme Ressourcen

Gegenwärtig wird der *Point Of Interest* (*POI*) um eine Aussage unabhängig von der Gestalt der Aussage gebildet, siehe 6.1.2 Umgebung und Abfragemodus. Denkbar wäre, unter Umständen einen größeren *POI* zu bilden, wenn in den erfassten Aussagen Anonyme Ressourcen enthalten sind (siehe Seite 20). Dies könnte geschehen, indem für jede Anonyme Ressource des jeweiligen *POIs* alle Aussagen hinzugefügt werden, welche diese Ressource *berühren*[44]. Alternativ könnte auch der *POI* dieser Ressource gebildet und hinzugefügt werden. Bei iterativer Anwendung ist jedoch zu beachten, dass diese Vorgehensweise nicht zwingend terminiert.

10.1.2 Umgebungsbegriff der Literale

Analog zu der Ausweitung von *POIs* bei Anonymen Ressourcen ist auch im Fall von enthaltenen Literalen zu überlegen, ob ein *POI* um die Umgebungen der Literale erweitert wird.

10.1.3 Prädikate aus RDF & RDF Schema

Undefiniert ist bislang, ob Prädikate der Namensräume RDF[45] und RDFS[46] wie gewöhnliche Prädikate behandelt werden, oder nicht. RQL sieht vor, diese speziellen Prädikate anders als andere Prädikate zu behandeln, d. h. ihre Verwendungen nicht bei der Suche nach Aussagen einzubeziehen.

Da sich die Implementierung *RqlEngine* (siehe 9 RqlEngine – ein RQL-Prozessor) in diesem Punkt an die RQL-Spezifikation hält, und RDF- sowie RDFS-Prädikate ebenfalls nicht zurückgibt, gibt auch *eRqlEngine* (siehe 8 eRqlEngine – ein eRQL-Prozessor) derlei Prädikate nicht zurück. Zu definieren ist, ob dieses Vorgehen beibehalten werden soll. Falls nicht, ist zu klären, wie die RDF- und RDFS-Prädikate aus der RQL-Schicht in die eRQL-Schicht gelangen, da sie nicht automatisch in den zurückgegebenen Aussagen enthalten sind.

---

[44] D. h. mindestens eine Ressource beinhalten, welches der anonymen Ressource entspricht, siehe Seite 23.
[45] http://www.w3.org/1999/02/22-rdf-syntax-ns#
[46] http://www.w3.org/2000/01/rdf-schema#

Die Rückgabe von RDF- und RDFS-Prädikaten würde dem Abfragesteller mehr Informationen verschaffen. Andererseits könnte der Abfragesteller dadurch leicht überfordert werden, da RDF- und RDFS-Prädikate a) sehr zahlreich eingesetzt werden (müssen), b) im Falle von impliziten Prädikaten maschinell erzeugt worden sind und c) eher Metadaten denn Daten aus Sicht des Abfragestellers sind.

Zur Demonstration wird die in 6.1.2.2 Point Of Interest-Modus (POI-Modus) verwendete Abfrage `Picasso` aufgegriffen, welche dort dazu dient, die Wirkung des POI-Operators zu demonstrieren. Sofern *RqlEngine* und *eRqlEngine* RDF/S-Aussagen zurückgäben, wäre das Ergebnis der Abfrage `Picasso` wie folgt:

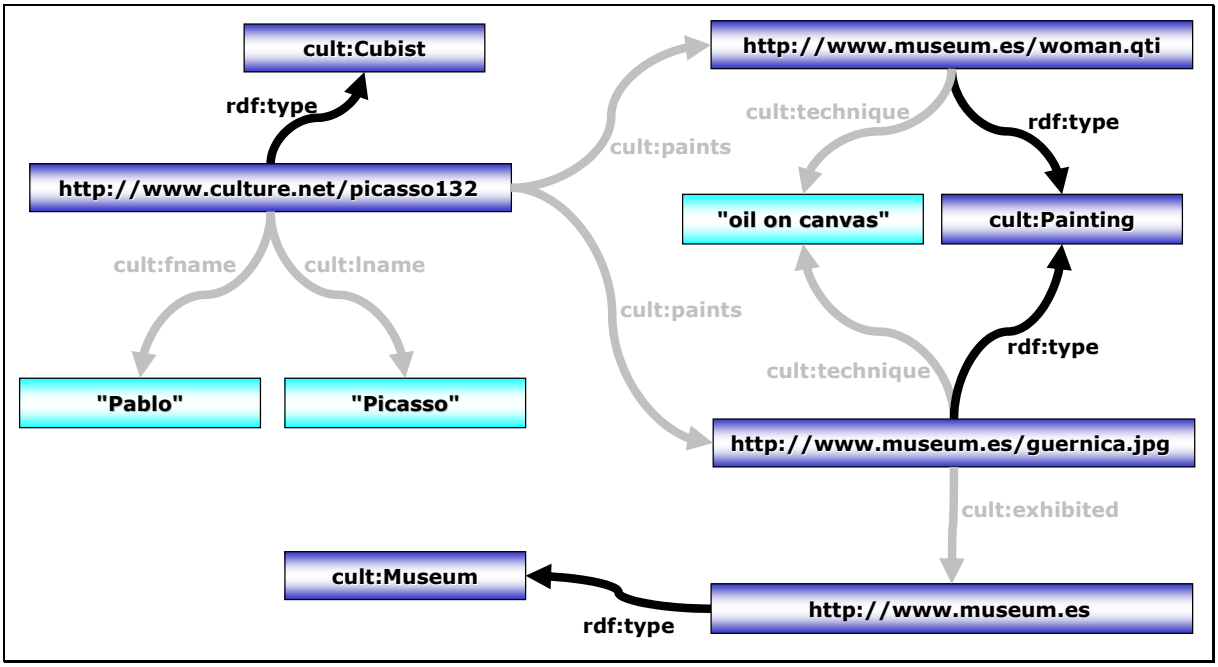

Abbildung 20 – Rückgabe von RDF/S-Aussagen

Die schwarz gezogenen Kanten entsprechen den RDF/S-Aussagen, welche zusätzlich zurückgegeben würden. Die Größe der Ergebnismenge vergrößert sich für den Abfragesteller erheblich (in diesem Fall: > 57%), und liefert ihm lediglich Metainformationen über das Datenmodell – jedoch keinerlei Informationen über Picasso.

10.1.4 Namensräume

eRQL berücksichtigt derzeit keine Namensräume. Dieses Vorgehen ist insofern praktikabel, als dass sich der unwissende Abfragesteller eines Informationsportals mit Nanemsraum-Aspekten sicherlich nicht auskennt.

Es stellt sich die Frage, ob Prädikat-Namen wie `first_name` oder `birth_date` nicht unabhängig von ihrem konkreten Namensraum mit sehr hoher Wahrscheinlichkeit bis Sicherheit dieselbe Bedeutung besitzen, und ob es nicht benutzungsfreundlicher ist, auch zukünftig Namensräume generell zu ignorieren. Zu definieren ist, ob eRQL Namensräume von RDF zukünftig beachten, oder weiterhin ignorieren sollte.

10.1.5 Numerische Literale und Datumswerte

Bislang sieht eRQL lediglich textuelle Vergleiche vor, worin auch ein Teil der Einfachheit von eRQL begründet liegt, da der Abfragesteller über keinerlei Schemakenntnisse verfügen muss, um das Modell durchsuchen zu können.

Nichtsdestotrotz existieren Anwendungsszenarien, in welchen es vorteilhaft wäre, spezielle Operationen auf nicht-textuellen Daten ausführen zu können. Zum Beispiel ist es derzeit nicht mit einer einzigen eRQL-Abfrage möglich, unter allen Ressourcen des Kulturinformationsportals diejenigen zu finden, welche seit mehr als vier Wochen nicht mehr aktualisiert worden sind.

Sollten entsprechende, datentypabhängige Funktionen in eRQL Einzug halten, wird zu diskutieren sein, ob und wie nicht-textuelle Literale (Datumswerte, Wahrheitswerte, XML-Literale, numerische Werte, …) gekennzeichnet werden können respektive müssen, um sie von textuellen Literalen unterscheiden zu können. Sowohl RQL als auch der in *RqlEngine* verwendete RDF-Parser VRP können bereits mit den Datentypen aus XML Schema [XMLSCHEMA] umgehen, *RqlEngine* an sich jedoch nicht.

Das Datenmodell von eRQL unterscheidet derzeit lediglich zwischen URIs und textuellen Literalen, es müsste daher entsprechend erweitert werden um Datentypen für Zeitangaben, Datumsangaben, Zahlen, etc.

10.1.6 Projektion und Selektion

Alle bisherigen Überlegungen haben sich allein auf den Aspekt der Informationssuche beschränkt. Dieser Abschnitt widmet sich der Frage, wie die Rückgabe der gefundenen Informationen beeinflusst werden kann.

Es wurde diskutiert, wie eine Abfrage formuliert wird und welche Aussagen von dieser Abfrage erfasst werden. Außer Acht gelassen wurde die Frage, ob und wie der Abfragesteller beeinflussen kann, welche Informationen über die jeweiligen Fundstellen zurückgegeben werden. Gegenwärtig wird die Menge aller gefundenen Aussagen zurückgegeben, wobei die Reihenfolge keiner definierten Ordnung folgt, und der eRQL-Implementierung freigestellt ist.

Szenario

> Der Besucher eines Informationsportals möchte wissen, wie der Vorname des Künstlers „Rodin“ lautet. Dazu formuliert er eine entsprechende Abfrage, und erwartet als Rückgabe (zu Recht) ausschließlich den Text „August“.
>
> Die bestmögliche Annäherung, welche eRQL zur Zeit erlaubt, wird in 6.2.6 „Vorname von Rodin“ finden dargestellt. Dort ist zu sehen, dass neben dem Vornamen von Rodin auch der Name des entsprechenden Prädikates, und seiner Ressource zurückgegeben wird.
>
> Der Abfragesteller erwartet also eine Information, welche zu granular ist, als dass eRQL sie liefern könnte.

Derartig „präzise“ Abfragen sind mittels eRQL derzeit nicht möglich, da eRQL weder eine exakte Selektion, noch eine Projektion vorsieht, wie sie von Abfragesprachen wie z. B. SQL, RQL, … bekannt ist. Mittels solch einer Projektions-Klausel lässt sich bestimmen, an welchen Aspekten der gefundenen Datensätze Interesse besteht.

eRQL bietet aus Gründen der Einfachheit keine solche Projektionsklausel an, der Abfragesteller ist vielmehr selber dafür zuständig, in den zurückgegebenen Aussagen die für ihn relevanten Aussagen zu erkennen, und ggf. zu entnehmen. Zu diesem Zweck ist eine entsprechende Visualisierung notwendig, siehe 10.2.4 Visualisierung.

10.2 Ausblick

10.2.1 Reguläre Ausdrücke

In einer Abfrage enthaltene Reguläre Ausdrücke müssen speziell gekennzeichnet werden, um sie von gewöhnlichem Text unterscheidbar zu machen. Diese Unterscheidung muss möglich sein, da der gewöhnliche Text speziell aufbereitet werden muss, die Regulären Ausdrücke hingegen nicht.

Wenn sich der Abfragesteller der Schreibweise nicht sicher ist (`*Louvre*`), nach ganzen Wertebereichen suchen möchte (`199[4-7]`) oder nach alternativen Werten suchen möchte (`(http|https|ftp)://www.louvre.fr`), können Reguläre Ausdrücke verwendet werden.

Reguläre Ausdrücke unterstützen den Abfragesteller beispielsweise in den folgenden Situationen:

- **Unsichere Schreibweise**

  ```
  Lou.?re
  ```

- **Bereichssuchen**

  ```
  www.w3.org/200[1-3]/.*
  ```

- **Alternativsuchen**

  ```
  (http|https|ftp):/www.louvre.fr
  ```

Reguläre Ausdrücke könnten an jeder Stelle verwendet werden, wo bislang Literale verwendet werden können. Durch die Vorgaben seitens RQL (siehe 7.1 Groß- und Kleinschreibung) macht eine eRQL-Implementierung ohnehin intensiven Gebrauch von Regulären Ausdrücken, so dass eine Implementierung prinzipiell einfach möglich sein sollte. Problematisch in diesem Zusammenhang ist jedoch die Kennzeichnung der Regulären Ausdrücke, da benötigte Symbole wie „[“, „]“, „(“, „)“, … im eRQL-Zusammenhang bereits eine spezielle Bedeutung tragen.

Aus diesem Grund sieht eRQL bislang keinen Mechanismus zur Kennzeichnung Regulärer Ausdrücke vor.

Es ist zu definieren, ob und wie Reguläre Ausdrücke als Bestandteil von eRQL-Abfragen eingesetzt und gekennzeichnet werden können respektive müssen, und ob es zugunsten einer einfa-

cheren eRQL-Syntax praktikabel ist, lediglich eine Teilmenge der Regulären Ausdrücke zu unterstützen (auf fortgeschrittene Möglichkeiten wie z. B. Gruppierung mittels (…) kann möglicherweise zugunsten einer einfacheren Syntax verzichtet werden). Denkbar wäre beispielsweise eine Art der Markierung wie in PERL[47]: `/regex/`.

Weiterhin ist zu diskutieren, ob von der durch PERL, Grep[48], … bekannten und etablierten Syntax der Regulären Ausdrücke zugunsten einer einfacheren Bedienbarkeit abgewichen werden sollte. Denkbar wäre z. B. die in Anlehnung an MS-DOS umdefinierte Bedeutung der Symbole „*“ und „?“ zu „`.*`“ und „`.`“, wie sie in RQL ohnehin bereits verwurzelt ist. Denkbar wäre weiterhin, symbolische Zeichenklassen für Ziffern, Buchstaben etc. einzuführen.

10.2.2 Rückgabe von Dokumentenverweisen

In allen gezeigten Beispielen wurden stets sämtliche gefundenen Aussagen zurückgegeben. Für die gezeigten Szenarien machte das durchaus Sinn, in dem folgenden Szenario hingegen wäre die Rückgabe ganzer *Dokumente* wünschenswert:

Szenario

Der Besucher eines Informationsportals ist an Dokumenten über Pablo Picasso interessiert, vorzugsweise im HTML- oder PDF-Format.

Abfrage: `<Pablo AND Picasso>`

Ergebnis

(alle Aussagen des RDF-Modells)

Im Gegensatz zu den bisherigen Szenarien erwartet der Abfragesteller also nicht die Rückgabe der Menge der gefundenen Aussagen (welche bei umfangreichen Dokumenten beliebig groß werden kann), sondern die Rückgabe von *Dokumentverweisen* (wie von klassischen Internetsuchmaschinen vertraut). Insbesondere in dem Fall, dass im Dokumentmodus operiert wird, kann die zurückgegebene Menge an Aussagen schnell erschlagend sein.

Der Abfragesteller erwartet also eine Information, welche wesentlich weniger granular ist, als eRQL sie liefert.

eRQL könnte um einen Modus erweitert werden, welcher die Rückgabe von Dokumentverweisen (z. B. in Gestalt von URLs) bewirkt. In diesem Modus würden keine einzelnen Aussagen mehr zurückgegeben, sondern lediglich die Menge der URLs aller Dokumente, in welchen Fundstellen lokalisiert worden sind. Gewissermaßen würde sich die kleinste Einheit der Rückgabe damit von der Aussage zum Dokument wandeln!

Damit dieser Modus praktikabel ist, müssten die Dokumente, welchen die Aussagen entstammen, hinreichend granular sein. Für den Fall, dass sämtliche Aussagen in wenigen Dokumenten befindlich sind, wäre dieser Modus gänzlich unpraktikabel (im Extremfall, wenn sämtliche Aussagen in

[47] http://www.perl.com
[48] http://www.gnu.org/software/grep/grep.html

einem einzigen Dokument enthalten sind, würde für jede Abfrage entweder die URI dieses einzigen Dokumentes zurückgegeben werden oder die leere Menge).

Welche Rückgabe eine Abfrage hervorrufen sollte, d. h. eine Menge von Aussagen oder eine Menge von Dokumentverweisen, lässt sich nicht allgemein für alle Szenarien beantworten, sondern ist vielmehr in jedem Einzelfall verschieden. Daher müssten für jede Fundstelle sowohl die betroffenen Aussagen zurückgegeben werden, als auch die URLs der Dokumente, welche sie beinhalten. Es liegt in der Verantwortung des Abfragestellers, bzw. der zum Einsatz kommenden Software, sich für eine der beiden Rückgabeformen zu entscheiden.

Unabhängig von der Art der gestellten Abfrage erhielte der Abfragesteller stets dieselbe Form der Rückgabe: eine Menge von Aussagen, jeweils auf dasjenige Dokument verweisend, dem sie entstammen.

Vorbereitend auf diese doppelte Rückgabe sieht es das eRQL Datenmodell vor, jede Aussage mit Informationen über die enthaltenden Dokumente zu hinterlegen, siehe 6.3.1.4 Aussage. Sowohl diese Hinterlegung, als auch die Rückgabe dieser Informationen sind in der „RQL-Welt“ nicht vorgesehen – abgesehen von der Variante, dass sie ebenfalls mittels RDF kodiert werden. Das verwundert nicht weiter, wenn man berücksichtigt, dass weder RQL noch RDF im Allgemeinen den Begriff des „Dokumentes“ überhaupt kennen.

Zur Realisierung der Dokumentenrückgabe sind zwei Vorgehen denkbar:

1. Hinterlegung der Information darüber, welchem Dokument eine Aussage entstammt, mittels RDF. Dadurch kann (theoretisch) mit RDF-Mitteln operiert werden, um die Anforderung zu erfüllen. Praktisch könnte dieser Weg an der (praktisch vorhandenen) Beschränktheit der RDF- und RQL-Mittel scheitern.
2. Erweiterung der Schnittstelle zwischen eRQL- und RQL-Schicht um die Rückgabe dieser Informationen, dies kann z. B. in Form einer zweiten, zusätzlichen Schnittstelle zum RQL-Prozessor erfolgen.

10.2.3 Schemaoperationen

eRQL unterstützt derzeit keine Operationen unter Einbeziehung von RDFS (RDF Schema). Das heißt, dass Abfragen nicht möglich sind, die z. B. alle Aussagen ermitteln möchten, welche eine Ressource einer bestimmten Klasse enthalten.

Eine Erweiterung von eRQL um Funktionen dieser Art wäre durchaus möglich. Beispielsweise könnte die folgende Syntax verwendet werden, um Aussagen zu ermitteln, welche eine Ressource der RDFS-Klasse `Painting` beinhalten:

Abfrage: `[rdfs:type(Painting)]`

Subjekt	Prädikat	Objekt
`http://www.photojournal.com/classicart/italmasters/michelangelo/sculptur/theslave.jpg`	`exhibited`	`http://www.louvre.fr/`
`http://www.artchive.com/rembrandt/artist_at_his_easel.jpg`	`-"-`	`-"-`

Gegenwärtig ist aus den folgenden Gründen keine derartige Erweiterung um Schemafunktionen in eRQL integriert:

- eRQL wurde vom Start weg als möglichst intuitive Sprache konzipiert, siehe 3 Ziele. Die Einbeziehung von Funktionen und komplizierten Operatoren ist zwar syntaktisch möglich, verletzt aber den Anspruch eines Werkzeuges für Jedermann.
- eRQL fußt auf RQL, und damit auf einer Sprache, welche umfangreiche Mittel zur gezielten Abfrage des RDF Schema und zur Navigation darin bereitstellt. Davon ausgehend, dass jede Implementierung von eRQL das Vorhandensein eines RQL-Prozessors impliziert, könnte dem Besucher des jeweiligen Informationsportals, zusätzlich zu der Durchführung von eRQL-Abfragen, auch die Durchführung von RQL-Abfragen möglich sein.
- In rudimentärem Maße würde eRQL bereits jetzt schon die Einbeziehung von RDFS-Informationen erlauben, wenn der RQL-Prozessor diese zurückgäbe – siehe 10.1.3 Prädikate aus RDF & RDF Schema. Obige Abfrage könnte beispielsweise wie folgt angenähert werden:

  ```
  Louvre [type] [Painting]
  ```

  Die Möglichkeiten von RDFS und auch RQL gehen allerdings weit über diese simple Abfrage hinaus, beispielsweise wird die Klassenhierarchie von RDFS berücksichtigt. Spätestens bei einer Abfrage wie der folgenden führt eRQL gegenwärtig nicht zum Ziel:

  ```
  Louvre [type] [Artifact]
  ```

  Ursache ist, dass eRQL derzeit die Klassenhierarchie nicht beachtet, und von der RDFS-Klasse `Artifact` keinesfalls auf die abgeleiteten Klassen (`Painting`, `Sculpture`, …) schließt. Da im Szenario keine Instanzen von `Artifact` existieren (sondern nur von nachfahrenden Klassen), existiert auch keine Aussage, die das Prädikat `type` und das Objekt `Artifact` enthält.

10.2.4 Visualisierung

Gänzlich unbearbeitet ist bislang die Visualisierung der zurückgegebenen Daten. Da hierzu unter Umständen Metadaten seitens des eRQL-Interpreters von Nöten sind, kann eine ansprechende

Visualisierung des Abfrageergebnisses Zugeständnisse seitens eRQL erfordern. Zu diesem Zweck ist z. B. eine Visualisierung in Graphenform denkbar, eine Realisierung könnte mittels SVG, Java oder ActiveX erfolgen. Die Visualisierung der zurückgegebenen Daten ist jedoch nicht Inhalt dieser Arbeit, siehe 1.1 Aufgabenstellung.

10.2.5 Abstraktionsschicht zum RDF-Parser

Gegenwärtig benutzt der RQL-Prozessor *RqlEngine* den RDF-Parser *VRP* zum Parsen der RDF-Dateien. Um hier eine größere Flexibilität in der Parserwahl zu ermöglichen, könnte eine Abstraktionsschicht eingezogen werden, beispielsweise in Form einer speziellen Klasse. Diese Abstraktionsschicht müsste Funktionen für alle von *RqlEngine* benötigten Funktionen bereitstellen, und diese auf Funktionen des *VRP* abbilden.

Zur Verwendung mit anderen RDF-Parsern müsste die Abstraktionsschicht um eine zweite Klasse erweitert werden, welche dieselbe Menge an Funktionen bereitstellt, diese jedoch auf die entsprechenden anderen Funktionen des Parsers abbildet.

10.2.6 Grafische Abfragenkomposition

Denkbar ist die Unterstützung des Abfragestellers durch grafische Auswahlmöglichkeiten. Dazu könnten dem Abfragesteller z. B. in Form einer Auswahlliste die Namen der RDFS-Klassen präsentiert werden, von denen Instanzen im Datenmodell vorhanden sind. Die folgenden Varianten sind denkbar:

- **Grafische bottom-up Konstruktion der Abfrage**

  Dem Abfragesteller werden elementare Abfrageelemente wie z. B. die im Datenmodell enthaltenen RDFS-Klassen grafisch visualisiert. Per Mausbewegung kann er diese Elemente zu einer vollständigen Abfrage arrangieren. Die Implementierung für den Einsatz im WWW könnte durch JavaScript, ein Java-Applet oder ein ActiveX-Steuerelement erfolgen.

- **Auswahlmöglichkeiten im Zweifelsfall**

  Der Abfragesteller formuliert seine Abfrage mittels eRQL wie gehabt. In bestimmten Zweifelsfällen, z. B. wenn der Abfragesteller nach dem Literal *Java* sucht, aber im Datenmodell mehrere Ressourcen mit der Beschriftung *Java* enthalten sind, wendet sich das System per Dialog zurück an den Abfragesteller, welcher die Uneindeutigkeit auflösen muss. Dieses Vorgehen ist insbesondere zur Eliminierung von Homonymen[49] nützlich.

[49] Siehe S. 14

Anhang

Anhang A – RQL-Syntax

Die Syntax von RQL kann unter [RQL-BNF] eingesehen werden. Die Syntax der durch *RqlEngine* unterstützten Teilmenge von RQL in BNF ist in der Datei `rql.cup` definiert, und lautet wie folgt:

```
         <Query> ::=  <Select><ProjectionList> FROM <RangeList> WHERE <Disjunction> |
                      <Select><ProjectionList> FROM <RangeList> |
                      <RangeList>

        <Select> ::=  SELECT | SELECT DISTINCT
<ProjectionList> ::=  <Projection> | <Projection> , <ProjectionList>
    <Projection> ::=  <Var>
     <RangeList> ::=  <Range> | <Range> , <RangeList>
         <Range> ::=  { <IDENTIFIER> } <PropertyVar> { <IDENTIFIER> } |
                      <IDENTIFIER> { <IDENTIFIER> } |
                      <IDENTIFIER>

   <Disjunction> ::=  <Conjunction> | <Disjunction> OR <Conjunction>
   <Conjunction> ::=  <Condition> | <Conjunction> AND <Condition>

     <Condition> ::=  <Value> <CompOp> <Value> | ( <Disjunction> )

         <Value> ::=  <Var> | <Literal>

           <Var> ::=  <IDENTIFIER> | <ClassVar> | <PropertyVar>
   <PropertyVar> ::=  @ <IDENTIFIER>
      <ClassVar> ::=  $ <IDENTIFIER>

       <Literal> ::=  <StringLiteral> | <UriLiteral>
 <StringLiteral> ::=  " <STRING> "
    <UriLiteral> ::=  & <URI>

        <CompOp> ::=  LIKE | =
```

Anhang B – RQL Grammatikdefinition für CUP

Die *CUP* Grammatikdefinition der durch *RqlEngine* unterstützten Teilmenge von RQL ist in der Datei `rql.cup` definiert, und lautet wie folgt:

```
import eworks.RQL.model.*;

/* Terminals (tokens returned by the scanner). */
          terminal    LCURLYBRACKET,RCURLYBRACKET;
          terminal    LBRACKET,RBRACKET;
          terminal    OROPERATOR, ANDOPERATOR, EQUALITYOPERATOR, LIKEOPERATOR;
          terminal    STRING, URI;
          terminal    SELECT, DISTINCT, FROM, WHERE;
          terminal    IDENTIFIER, AT_IDENTIFIER, DOLLAR_IDENTIFIER;
          terminal    COMMA;

/* Non terminals */
      non terminal    query;
      non terminal    projection, range, condition, conjunction, disjunction;
      non terminal    value;
      non terminal    var, datavar, propertyvar, classvar;
      non terminal    literal, stringliteral, uriliteral;
      non terminal    compop;
      non terminal    projectionlist, rangelist;
      non terminal    select, where;
```

```
/* Precedences */
precedence left DISTINCT;
precedence left LBRACKET, RBRACKET;
precedence left OROPERATOR;
precedence left ANDOPERATOR;
precedence left LCURLYBRACKET, RCURLYBRACKET;
precedence left EQUALITYOPERATOR, LIKEOPERATOR;

/* The grammar */
         query ::=    select:distinct projectionlist:pl FROM rangelist:rangeList
                      WHERE disjunction:d
                      {: RESULT = new SelectFromWhereQuery((ProjectionList) pl,
                      (RangeList) rangeList, (DisjunctionCondition) d,
                      ((Boolean) distinct).booleanValue()); :}
                      |
                      select:distinct projectionlist:pl FROM rangelist:rangeList
                      {: RESULT = new SelectFromWhereQuery((ProjectionList) pl,
                      (RangeList) rangeList, null, ((Boolean) distinct).booleanValue()); :}
                      |
                      rangelist:rangeList
                      {: RESULT = new SelectFromWhereQuery(null,(RangeList) rangeList,
                      null, true); :}
                      ;

        select ::=    SELECT {: RESULT = new Boolean(false); :}
                      |
                      SELECT DISTINCT {: RESULT = new Boolean(true); :}
                      ;

projectionlist ::=    projection:p {: RESULT = new ProjectionList((Projection) p); :}
                      |
                      projection:p COMMA projectionlist:pl
                      {: ((ProjectionList) pl).add( (Projection) p); RESULT = pl; :}
                      ;

    projection ::=    var:v {: RESULT = new Projection((Variable) v); :};

     rangelist ::=    range:r {: RESULT = new RangeList( (Range) r ); :}
                      |
                      range:range COMMA rangelist:rangeList
                      {: ((RangeList) rangeList).add((Range) range); RESULT = rangeList; :}
                      ;

         range ::=    LCURLYBRACKET IDENTIFIER:dv1 RCURLYBRACKET propertyvar:pv
                      LCURLYBRACKET IDENTIFIER:dv2 RCURLYBRACKET
                      {: RESULT = new StatementRange(new DataVariable((String) dv1),
                      (PropertyVariable) pv,new DataVariable((String) dv2)); :}
                      |
                      IDENTIFIER:classname LCURLYBRACKET IDENTIFIER:dv RCURLYBRACKET
                      {: RESULT = new InstancesOfClassRange((String) classname,
                      new DataVariable((String) dv)); :}
                      |
                      IDENTIFIER:classname
                      {: RESULT = new InstancesOfClassRange((String) classname); :}
                      ;

   disjunction ::=    conjunction:c {: RESULT = new DisjunctionCondition((Condition) c); :}
                      |
                      disjunction:d OROPERATOR conjunction:c
                      {: ((DisjunctionCondition) d).add((Condition) c); RESULT = d; :}
                      ;

   conjunction ::=    condition:c {: RESULT = new ConjunctionCondition((Condition) c); :}
                      |
                      conjunction:c1 ANDOPERATOR condition:c2
                      {: ((ConjunctionCondition) c1).add((Condition) c2); RESULT = c1; :}
                      ;

     condition ::=    value:op1 compop:op value:op2
                      {: RESULT = new ComparisonCondition((Value) op1, (CompareOperator) op,
                      (Value) op2); :}
                      |
                      LBRACKET disjunction:d RBRACKET {: RESULT = d; :}
                      ;

         value ::=    var:v {: RESULT = v; :} | literal:l {: RESULT = l; :};
```

```
          var ::=    IDENTIFIER:s {: RESULT = new DataVariable((String) s); :}
                     |
                     propertyvar:s {: RESULT = s; :}
                     |
                     classvar:s {: RESULT = s; :}
                     ;

   propertyvar ::=   AT_IDENTIFIER:s {: RESULT = new PropertyVariable((String) s); :};

      classvar ::=   DOLLAR_IDENTIFIER:s {: RESULT = new ClassVariable((String) s); :};

       literal ::=   stringliteral:s {: RESULT = s; :} | uriliteral:s {: RESULT = s; :};

 stringliteral ::=   STRING:s {: RESULT = new StringLiteralValue((String) s); :};

    uriliteral ::=   URI:s {: RESULT = new UriLiteralValue((String) s); :};

        compop ::=   EQUALITYOPERATOR {: RESULT = CompareOperator.Equality; :}
                     |
                     LIKEOPERATOR {: RESULT = CompareOperator.Alikeness; :}
                     ;
```

Anhang C – RQL Lexerdefinition für JFlex

Die von *RqlEngine* verwendete *JFlex* Lexerdefinition ist in der Datei `rql.flex` definiert, und lautet wie folgt:

```
package eworks.RQL.parser;
import java_cup.runtime.*;

%%

%class scanner
%cup
%ignorecase
%8bit
%public

%{
  String name;

  StringBuffer string = new StringBuffer();

  static {
        yycmap['\t'] = yycmap[' '];
        yycmap[',']  = yycmap[' '];
        yycmap['=']  = yycmap[' '];
        yycmap['(']  = yycmap[' '];
        yycmap[')']  = yycmap[' '];
        yycmap['{']  = yycmap[' '];
        yycmap['}']  = yycmap[' '];
        yycmap['\''] = yycmap[' '];
        yycmap['\n'] = yycmap[' '];
  }

  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }
%}

LineTerminator = \r|\n|\r\n
WhiteSpace     = {LineTerminator} | [ \t\f]

%state STRING

%%

<YYINITIAL> {
       \"                             { string.setLength(0); yybegin(STRING); }
       {WhiteSpace}                   { if (yytext().equals(","))
```

```
                                        return symbol(sym.COMMA);
                                    else if (yytext().equals("="))
                                        return symbol(sym.EQUALITYOPERATOR);
                                    else if (yytext().equals("("))
                                        return symbol(sym.LBRACKET);
                                    else if (yytext().equals(")"))
                                        return symbol(sym.RBRACKET);
                                    else if (yytext().equals("{"))
                                        return symbol(sym.LCURLYBRACKET);
                                    else if (yytext().equals("}"))
                                        return symbol(sym.RCURLYBRACKET);
                                }
        "SELECT"                    { return symbol(sym.SELECT); }
        "DISTINCT"                  { return symbol(sym.DISTINCT); }
        "FROM"                      { return symbol(sym.FROM); }
        "WHERE"                     { return symbol(sym.WHERE); }
        "AND"                       { return symbol(sym.ANDOPERATOR); }
        "OR"                        { return symbol(sym.OROPERATOR); }
        "LIKE"                      { return symbol(sym.LIKEOPERATOR); }
        @[a-zA-Z][a-zA-Z0-9_]*      { return symbol(sym.AT_IDENTIFIER,yytext()); }
        \$[a-zA-Z][a-zA-Z0-9_]*     { return symbol(sym.DOLLAR_IDENTIFIER,yytext()); }
        [a-zA-Z][a-zA-Z0-9_]*       { return symbol(sym.IDENTIFIER,yytext()); }
        &((http|file):\/\/)?[^ \(\)\\{}<>\"]+
                                    { return symbol(sym.URI,yytext().substring(1)); }
        [^ ()\[\]{}<>~\"]+          { return symbol(sym.STRING,yytext()); }
}

<STRING> {
        \"                      { yybegin(YYINITIAL);
                                  return symbol(sym.STRING,
                                  string.toString()); }
        [^\n\r\"\\]+            { string.append( yytext() ); }
        \\t                     { string.append('\t'); }
        \\n                     { string.append('\n'); }

        \\r                     { string.append('\r'); }
        \\\"                    { string.append('\"'); }
        \\                      { string.append('\\'); }
}

/* error fallback */
.|\n                            { throw new Error("Illegal character <" + yytext() + ">"); }
```

Anhang D – eRQL Syntax

Die Syntax von eRQL ist in der Datei `erql.cup` definiert, und lautet in BNF[50] wie folgt:

```
         <Query> ::=  <Disjunction>

   <Disjunction> ::=  <Conjunction> | <Disjunction> OR <Conjunction>
   <Conjunction> ::=  <SubQuery> |
                      <Conjunction> AND <SubQuery> |
                      <Conjunction> <SubQuery>

      <SubQuery> ::=  <Literal> |
                      { <Disjunction> } | ~ <Disjunction> | { <Literal> } |
                      < <Disjunction> > | < <Literal> > | [ <Disjunction> ] |
                      [ <Literal> ] | ( <Disjunction> )

       <Literal> ::=  " <STRING_LITERAL> " |
                      <STRING_LITERAL> |
                      <URI_LITERAL>
```

[50] http://de.wikipedia.org/wiki/Backus-Naur_Form

Anhang E – eRQL Grammatikdefinition für CUP

Die *CUP* Grammatikdefinition für eRQL ist in der Datei `erql.cup` definiert, und lautet wie folgt:

```
import eworks.eRQL.model.*;

/* Terminals (tokens returned by the scanner). */
terminal             TILDE;

terminal             LANGLEBRACKET,RANGLEBRACKET;
terminal             LCURLYBRACKET,RCURLYBRACKET;
terminal             LSQUAREBRACKET,RSQUAREBRACKET;
terminal             LBRACKET,RBRACKET;

terminal             OROPERATOR, ANDOPERATOR;
terminal             STRING,URI;

/* Non terminals */
non terminal         query;
non terminal         conjunction, disjunction;
non terminal         subquery;
non terminal         literal;

/* Precedences */
precedence left      OROPERATOR;
precedence left      ANDOPERATOR;
precedence left      LBRACKET, RBRACKET;
precedence left      LANGLEBRACKET, RANGLEBRACKET, LSQUAREBRACKET, RSQUAREBRACKET,
                     LCURLYBRACKET, RCURLYBRACKET;
precedence left      TILDE, STRING, URI;

/* The grammar */
         query ::=   disjunction:d
                      {: RESULT=d; :};

   disjunction ::=   conjunction:c
                     {: RESULT=new Disjunction((Conjunction) c); :} |
                     disjunction:d OROPERATOR conjunction:c
                     {: ((Disjunction) d).add((Conjunction) c); RESULT=d; :} ;

   conjunction ::=   subquery:p
                     {: RESULT=new Conjunction((Query) p); :} |
                     conjunction:c ANDOPERATOR subquery:p
                     {: ((Conjunction) c).add((Query) p); RESULT=c; :} |
                     conjunction:c subquery:p
                     {: ((Conjunction) c).add((Query) p); RESULT=c; :} ;

      subquery ::=   literal:l
                     {: RESULT=new PoiQuery((Literal) l); :} |
                     LCURLYBRACKET disjunction:d RCURLYBRACKET
                     {: RESULT=new PoiQuery((Disjunction) d); :} |
                     TILDE disjunction:d
                     {: RESULT=new PoiQuery((Disjunction) d); :} |
                     LCURLYBRACKET literal:l RCURLYBRACKET
                     {: RESULT=new PoiQuery((Literal) l); :} |
                     LANGLEBRACKET disjunction:d RANGLEBRACKET
                     {: RESULT=new DocumentQuery((Disjunction) d); :} |
                     LANGLEBRACKET literal:l RANGLEBRACKET
                     {: RESULT=new DocumentQuery((Literal) l); :} |
                     LSQUAREBRACKET disjunction:d RSQUAREBRACKET
                     {: RESULT=new StatementQuery((Disjunction) d); :} |
                     LSQUAREBRACKET literal:l RSQUAREBRACKET
                     {: RESULT=new StatementQuery((Literal) l); :} |
                     LBRACKET disjunction:d RBRACKET
                     {: RESULT=new Query((Disjunction) d); :} ;

       literal ::=   STRING:s
                     {: RESULT=new Literal((String) s); :} |
                     URI:u
                     {: RESULT=new Literal((String) u); :} ;
```

Anhang F – eRQL Lexerdefinition für JFlex

Die von *eRqlEngine* verwendete *JFlex* Lexerdefinition ist in der Datei `erql.flex` definiert, und lautet wie folgt:

```
package eworks.eRQL.parser;
import java_cup.runtime.*;

%%

%class scanner
%cup
%ignorecase
%8bit
%public

%{
  String name;

  StringBuffer string = new StringBuffer();

  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }
%}

LineTerminator = \r|\n|\r\n
WhiteSpace     = {LineTerminator} | [ \t\f]

%state STRING

%%

<YYINITIAL> {
        \"                              { string.setLength(0); yybegin(STRING); }
        {WhiteSpace}                    { }
        "AND"                           { return symbol(sym.ANDOPERATOR); }
        "OR"                            { return symbol(sym.OROPERATOR); }
        &?[a-zA-Z]{3,10}"://"[^ ]+      { return symbol(sym.URI,yytext()); }
        [^ ()\[\]{}<>~\"\t\n]+          { return symbol(sym.STRING,yytext()); }
        "("                             { return symbol(sym.LBRACKET); }
        ")"                             { return symbol(sym.RBRACKET); }
        "["                             { return symbol(sym.LSQUAREBRACKET); }
        "]"                             { return symbol(sym.RSQUAREBRACKET); }
        "{"                             { return symbol(sym.LCURLYBRACKET); }
        "}"                             { return symbol(sym.RCURLYBRACKET); }
        "<"                             { return symbol(sym.LANGLEBRACKET); }
        ">"                             { return symbol(sym.RANGLEBRACKET); }
        "~"                             { return symbol(sym.TILDE); }
}

<STRING> {
        \"                              { yybegin(YYINITIAL);
                                          return symbol(sym.STRING,
                                          string.toString()); }
        [^\n\r\"\\]+                    { string.append( yytext() ); }
        \\t                             { string.append('\t'); }
        \\n                             { string.append('\n'); }

        \\r                             { string.append('\r'); }
        \\\"                            { string.append('\"'); }
        \\                              { string.append('\\'); }
}

/* error fallback */
.|\n                                    { throw new Error(
                                            "Illegal character <" + yytext() + ">"); }
```

Literaturverzeichnis

Fachbücher

[CMXML] Gunther Rothfuss, Christian Ried: *Content Management mit XML*. Springer, 2001.

Diplomarbeiten

[TOLLE] Karsten Tolle: *Analyzing and Parsing RDF*. Vorgelegt an der Universität Hannover. Neu-Isenburg, Januar 2000.

Internetseiten

[AMETAS] *AMETAS – Good Migrations*. Verfügbar unter http://www.accsis.de/ametas/.

[CARLSON] Dr. Christopher N. Carlson: *Data Smog, Precision und Recall: Retrievalstrategien zur Ballast-Reduzierung bei Internet-Recherchen*. Verfügbar unter http://voeb.uibk.ac.at/odok2003/carlson.pdf.

[CUP] *CUP Parser Generator for Java*. Verfügbar unter http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[DBIS_ERQL] Verfügbar unter http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/.

[DUDEN] *Der Große Duden – Fremdwörterbuch*. 2. Auflage. Bibliographisches Institut AG, Mannheim, 1966.

[ICS-FORTH] *FORTH – ICS Welcome Note by the Director of ICS-FORTH*. Verfügbar unter http://www.ics.forth.gr.

[IOCTL] *ioctl.org : RDF literals*. Verfügbar unter http://ioctl.org/rdf/literals.

[IWI-IUK] *Retrieval auf RDF*. Vortrag beim Seminar an der Deutschen Bibliothek, 2000. Verfügbar unter http://www.iwi-iuk.org/seminarNotes/1/rql2.pdf.

[JAVA-HOME] *The Source for Java Technology*. Verfügbar unter http://java.sun.com.

[JFLEX] *JFlex – The Fast Scanner Generator for Java*. Verfügbar unter http://www.jflex.de/.

[LANA] *LANA – Mobile Object Systems*. Verfügbar unter http://cui.unige.ch/OSG/research/lana.html.

[METALOG] Massimo Marchiori, Samuele Trevisan, Antonio Epifani: *Metalog – The query/logical system for the Semantic Web*. Verfügbar unter http://www.w3.org/RDF/Metalog/.

[METALOG-98] Massimo Marchiori, Janne Saarela*: Metalog – Querying RDF data models*. Verfügbar unter http://www.w3.org/RDF/Metalog/paper980716.html.

[RDF] *Resource Description Framework (RDF) Model and Syntax Specification*. Verfügbar unter http://www.w3.org/TR/REC-rdf-syntax/.

[RDF-FAQ] *Frequently Asked Questions about RDF*. Verfügbar unter http://www.w3.org/RDF/FAQ.

[RDF-HOME] *Resource Description Framework (RDF) – W3C Semantic Web Activity*. Verfügbar unter http://www.w3.org/RDF/.

[RDFAPI] Sergey Melnik: *RDF API Draft*. Stand 19. Januar 2001. Verfügbar unter http://www-db.stanford.edu/~melnik/rdf/api.html.

[RDFDB] R. V. Guha*: rdfDB – An RDF Database*. Verfügbar unter http://www.guha.com/rdfdb/.

[RDFDB-QL] R. V. Guha*: rdfDB Query Language*. Verfügbar unter http://www.guha.com/rdfdb/query.html.

[RDFS] *RDF Vocabulary Description Language 1.0: RDF Schema*. Verfügbar unter http://www.w3.org/TR/rdf-schema/.

[RDFSUITE] *The ICS-FORTH RDFSuite*. Verfügbar unter http://139.91.183.30:9090/RDF/.

[RDQL] *RDQL – RDF Data Query Language*. Verfügbar unter http://www.hpl.hp.com/semweb/rdql.htm.

[RDQL-TUT] Andy Seaborne: *A Programmer's Introduction to RDQL*. Verfügbar unter http://jena.sourceforge.net/tutorial/RDQL/index.html.

[RQL-2002] Sofia Alexaki, Vassilis Christophides, Grigoris Karvounarakis et. al.: *RQL: A Declarative Query Language for RDF*. In Proc. of the Eleventh International World Wide Web Conference (WWW'02), Honolulu, Hawaii, USA, März 2002 Verfügbar unter http://athena.ics.forth.gr:9090/RDF/publications/www2002/www2002.html.

[RQL-BNF] *The BNF grammar of RQL v2.0*. Verfügbar unter http://139.91.183.30:9090/RDF/RQL/bnf.html

[RQL-Demo] *RQL Demo*. Verfügbar unter http://139.91.183.30:8999/RQLdemo/.

[RQL-DOC] *RQL v2.0 Documentation*. Stand 18.7.2003, Verfügbar unter http://139.91.183.30:9090/RDF/RQL/Design.html.

[RQL-MAN] Vassilis Christophides, Grigoris Karvounarakis: *RQL v1.5 User Manual*. Verfügbar unter http://139.91.183.30:9090/RDF/RQL/Manual.html, Heraklion (Griechenland)

[RQL-OVW] *The RDF Query Language (RQL)*, Stand 18.7.2003, Verfügbar unter http://139.91.183.30:9090/RDF/RQL/.

[RQL-FUNC] Sophia Alexaki, Vassilis Christophides, Gregory Karvounarakis et. al.: *RQL: A Functional Query Language for RDF*, Verfügbar unter http://www.dbis.informatik.uni-frankfurt.de/~tolle/Publications/FuncBook.pdf.

[SquishQL] *Inkling: RDF query using SquishQL*. Verfügbar unter http://swordfish.rdfweb.org/rdfquery/.

[SERQL] *Chapter 5. The SeRQL query language*. Version 0.96 (September 2003), Verfügbar unter http://sesame.aidministrator.nl/publications/users/ch05.html.

[SIMAT] *SIMAT – Secure Integration of Middleware and Agent Technologies*. Verfügbar unter http://cui.unige.ch/OSG/research/simat.html.

[VRP-HOME] *The ICS-FORTH Validating RDF Parser (VRP)*. Verfügbar unter http://139.91.183.30:9090/RDF/VRP/.

[W3C] *World Wide Web Consortium*. Verfügbar unter http://www.w3.org/.

[W3C-MEMB] *World Wide Web Consortium (W3C) Members*. Verfügbar unter http://www.w3.org/Consortium/Member/List.

[WLEKLINSKI] *Searching the Semantic Web*. Verfügbar unter http://www.wleklinski.de/rdf/.

[XQUERY] *W3C XML Query (XQuery)*. Verfügbar unter http://www.w3.org/XML/Query.

[XPATH] *XML Path Language (XPath)*. Verfügbar unter http://www.w3.org/TR/xpath.

[XSLT] *The Extensible Stylesheet Language Family (XSL)*. Verfügbar unter http://www.w3.org/Style/XSL/.

[XMLSCHEMA] *W3C XML Schema*. Verfügbar unter http://www.w3.org/XML/Schema.

Stichwortverzeichnis

L

M

N

O

P

Q

R

S

T

U

V

W

X