Einsatz von Web Services im Semantic Web am Beispiel der RACER Engine und OWL-QL

Jan Galinski

Eingereicht zur Erlangung des Titels
Diplom-Ingenieur Elektrotechnik

Betreut von Prof. Dr. Ralf Möller
M.Sc. Atila Kaya

Technische Universität Hamburg-Harburg [TUHH]
Arbeitsbereich Software Systems Group [STS]

Danksagung

Ich danke Prof. Ralf Möller, für das interessante Thema und die Betreuung und Unterstützung während der Durchführung dieser Diplomarbeit, und den Mitarbeitern des Arbeitsbereichs, insbesondere Sebastian Boßung und Michael Wessel für ihre Hilfe.

Mein besonderer Dank gilt Atila Kaya für seine intensive persönliche Betreuung und die unzähligen anregenden Diskussionen, Gespräche und Schriftwechsel, die die Umsetzung dieser Arbeit erst möglich gemacht haben.

Nicht unerwähnt lassen möchte ich an dieser Stelle meine Kommulitonen Tejas und Jog: Danke für die gute Zusammenarbeit bei der Realisierung der Teilprojekte.

Darüber hinaus danke ich meiner Familie und meiner Freundin für ihre Ratschläge, Hinweise und ihre Geduld.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Aufgabenstellung und Rahmenbedingungen . . . 2
- 1.3 Anmerkungen zum Aufbau der Arbeit . . . 2

2 Grundlagen **5**
- 2.1 Semantic Web . . . 5
  - 2.1.1 Komponenten des Semantic Web . . . 6
  - 2.1.2 Repräsentation . . . 7
  - 2.1.3 Sicherheit . . . 8
- 2.2 Beschreibungslogik-Systeme . . . 9
  - 2.2.1 Wissensbasen . . . 9
  - 2.2.2 Formale Definition . . . 11
  - 2.2.3 Inferenz-System . . . 12
- 2.3 Web Services . . . 13
  - 2.3.1 Komponenten von Web Services . . . 13
  - 2.3.2 SOAP . . . 15
  - 2.3.3 Message Styles . . . 16

3 Analyse **19**
- 3.1 RACER . . . 19
  - 3.1.1 Der RACER-Proxy . . . 20
  - 3.1.2 Das JRacer-API . . . 21
  - 3.1.3 RICE . . . 22
  - 3.1.4 Architektur des Systems . . . 22
- 3.2 Die new RACER Query Language . . . 23
  - 3.2.1 Die Syntax einer nRQL-Query . . . 23
  - 3.2.2 Die Syntax einer nRQL-Antwort . . . 24
- 3.3 Die OWL Query Language . . . 25
  - 3.3.1 Der Dialog . . . 25
  - 3.3.2 Die Syntax des Frage-Dokuments . . . 26
  - 3.3.3 Die Syntax des Antwort-Dokuments . . . 31

3.3.4 Zusammenfassung . . . 33
3.4 Vergleich von OWL-QL und nRQL . . . 35
3.4.1 Zusammenfassung . . . 37
3.5 Resultierende Aufgabenstellung . . . 37

4 Entwurf **39**
4.1 Auswahl eines Web Service-Frameworks . . . 39
4.1.1 Stand-Alone Lösung . . . 40
4.1.2 Eingebettet in Server . . . 40
4.2 Modellierung von OWL-QL Eigenschaften . . . 41
4.2.1 Unterstützung von Prämissen . . . 42
4.2.2 Unterstützung von Don't-Bind-Variablen . . . 43
4.3 Architektur-Entwurf . . . 44
4.3.1 Definition der Schnittstellen . . . 45
4.3.2 Interaktion der Komponenten . . . 46

5 Realisierung **49**
5.1 Vorstellung der verwandten Teilprojekte . . . 49
5.1.1 RACER Server . . . 50
5.1.2 RACER Proxy . . . 51
5.1.3 DialogManager . . . 52
5.2 Das Paket *Syntax* . . . 52
5.2.1 Das Paket *NrqlSyntax* . . . 53
5.2.2 Das Paket *OwlQlSyntax* . . . 57
5.2.3 Translator . . . 57
5.3 Das Paket Service . . . 60
5.3.1 NrqlService . . . 61
5.3.2 OwlQlService . . . 61
5.4 Zusammenführung der Teilprojekte . . . 62

6 Zusammenfassung und Ausblick **63**
6.1 Zusammenfassung . . . 63
6.1.1 nRQL-Service . . . 63
6.1.2 OWL-QL Service . . . 64
6.2 Offene Probleme . . . 64
6.3 Kritische Betrachtung von OWL-QL . . . 65
6.4 Ausblick . . . 66
6.4.1 Objekt-Orientierte API für nRQL . . . 66
6.4.2 Unterstützung von OWL-QL . . . 67
6.5 Fazit . . . 67

A Eingesetzte Werkzeuge **69**
A.1 Apache Axis . . . 69
A.1.1 Architektur von Axis . . . 70
A.1.2 Einsatz von Axis . . . 71
A.2 Jena Semantic Web Framework . . . 72
A.2.1 Einsatz des Frameworks . . . 73
A.3 XMLBeans . . . 77
A.3.1 Einsatz von XMLBeans . . . 78

B Ausgewählter Quellcode **79**
B.1 OWL QL-Syntax . . . 79
B.1.1 Owl-QL Syntax . . . 79
B.1.2 Owl-QL Variables . . . 85
B.1.3 Query . . . 86
B.1.4 AnswerBundle . . . 87
B.2 Verwendete Beispiel-Knowledgebase . . . 89
B.3 XMLBeans Ant-Task . . . 90
B.4 Java Quellcode . . . 92
B.4.1 Die Klasse *OwlQlSimpleService* . . . 92
B.4.2 Die Klasse *Translator* . . . 94

C Inhaltsverzeichnis der Daten CD **101**
C.1 Dokumente . . . 101
C.2 Software . . . 101
C.3 Referenzen . . . 101

Glossar **103**

Literaturverzeichnis **107**

Abbildungsverzeichnis

2.1 Semantic Web Architektur 6
2.2 RDF Graph Model 8
2.3 Architektur eines Beschreibungslogik-Systems 9
2.4 Ontologie 10
2.5 Aufbau 14
2.6 Aufbau einer SOAP-Nachricht 15

3.1 Alte Architektur 22
3.2 OWL-QL Dialog Schema 25

4.1 Ablauf Prämisse 43
4.2 Architektur Entwurf 45
4.3 Sequenz-Diagramm des neuen RACER Systems 47

5.1 Überblick über die Paket-Struktur 50
5.2 Architektur des neuen RACER Proxy 52
5.3 Architektur des Connection-Pakets 53
5.4 Paketdiagramm *racer.syntax* 54
5.5 Interfaces des Pakets *syntax.nrql* 54
5.6 Klassen-Diagramm Atom 55
5.7 Klassendiagramm Axiom 55
5.8 Klassendiagramm Assertion 56
5.9 Klassendiagramm syntax.owlql 58

A.1 Architektur der AXIS-Engine 70
A.2 Einsatz von WSDL mit Axis 71
A.3 Architektur von Jena2 [11] 73

Quellcode-Verzeichnis

3.1 OWL-QL Premise Example 26
3.2 QueryPattern Example 27
3.3 ServerContinuation mit ProcessHandle 28
3.4 Definition von MustBind-Variablen 29
3.5 Definition einer AnswerKBReferenz 30
3.6 Festlegung der Anzahl gewünschter Antworten 31
3.7 BindingSet Example 32
3.8 AnswerPatternInstance Example 32
3.9 AnswerBundle ServerContinuation 33
A.1 Erzeugung von RDF mit Jena, Java-Klasse 74
A.2 Erzeugung von RDF mit Jena, RDF-Output 75
A.3 Parsen von RDF mit Jena, Java-Klasse 75
A.4 Parsen von RDF mit Jena, Console Output 77
B.1 OWL-QL Schema 79
B.2 OWL-QL Variables Schema 85
B.3 OWL-QL Query 86
B.4 OWL-QL AnswerBundle 87
B.5 Knowledge Base Example 89
B.6 XMLBeans Ant-Task 90
B.7 Quellcode der Klasse service.OwlQLSimpleService 92
B.8 Quellcode der Klasse syntax.Translator 94

Tabellenverzeichnis

3.1 Aufbau Query-Document 34
3.2 Aufbau AnswerBundle-Document 35

Kapitel 1

Einleitung

Das Semantic Web ist eine Vision, die den Umgang mit dem Internet verändern wird [27]. Es wird Maschinen erlauben aufgrund logischer Zusammenhänge Informationen zu erkennen, zu verarbeiten und inhaltlich zu analysieren.

Das Wissen und das Wesen der Welt kann in Wissensdomänen klassifiziert und durch intelligente Suchdienste, die auch Zusammenhänge zwischen Informationen berücksichtigen können, sie erfassbarer und somit leichter verfügbar machen.

Die Technik hinter dieser Entwicklung bilden Reasoner oder Inferenz-Maschinen, die aufgrund von Beschreibungslogiken die semantische Bedeutung von Begriffen erfassen und daraus logische Rückschlüsse ziehen können.

Für die Kommunikation von Anbietern und Nutzern solcher Dienste ist eine gemeinsame Sprache sowie ein Protokoll erforderlich.

Die stetig zunehmende Vernetzung von heterogenen Systemen in offenen IT-Landschaften bewirkte in den letzten Jahren eine Abkehr von proprietären Entwicklungen hin zu offenen, freien Standards. Diese ermöglichen die freie Wahl des Kommunikationspartners unabhängig von der jeweiligen Plattform, Implementierungssprache und Transportprotokoll.

Diese Arbeit ist ein Beitrag zur Realisierung der Kommunikation über offene Standards mit einem bestehenden Reasoner.

1.1 Motivation

Das Ziel dieser Arbeit ist die Entwicklung einer Architektur, die es ermöglicht den Reasoner RACER über offene Web-Standards nutzbar zu machen und somit die möglichen Einsatzbereiche der Software zu vergrößern.

Die zugrunde liegende Software-Version unterstützt nativ die Kommunikation über TCP und HTTP/DIG, einem XML-Standard. Durch den Einsatz von Web-Services, bzw. des

Protokolls SOAP für die Kommunikation zwischen Kommunikationspartnern kann ein weitgehend transportunabhängiger Dialog ermöglicht werden.

Die Unterstützung des kommenden Standards OWL-QL, wird es ermöglichen, dem Client eine plattform-, sprach- und systemübergreifende Anfragesprache für den RACER-Server anzubieten.

1.2 Aufgabenstellung und Rahmenbedingungen

Die konkrete Aufgabenstellung dieser Arbeit unterteilt sich in drei Unteraufgaben.

1. Die Entwicklung eines Web-Service, der Anfragen in der system-eigenen Sprache nRQL unterstützt.
2. Die Untersuchung der Konzepte von OWL-QL und nRQL im Hinblick auf Gemeinsamkeiten und Unterschiede mit dem Ziel eine konkrete Untermenge beider Konzepte zu definieren, die als Basis für die Entwicklung eines Übersetzers zwischen beiden Sprachen dienen kann.
3. Die prototypische Realisierung einer Architektur, die Anfragen an einen RACER-Server über einen Service mit der Sprache OWL-QL ermöglicht.

Dabei stellt die Entwicklung eine Grundlagen-Arbeit dar, die Web-Services als Standard zur Kommunikation zwischen verteilten Systemen verwendet. Die Lösung bestehender Probleme mit der Web-Service Technologie als solcher[1] oder die Anwendung der Konzepte der *Semantic Web enabled Web Services* (SWWS),wie OWL-S, sind nicht Teil dieser Arbeit.

Diese Arbeit ist Teil einer Reihe von Projekten zur Erweiterung des RACER-Systems am Arbeitsbereich STS. Auf den Zusammenhang mit verwandten Arbeiten wird in späteren Kapiteln eingegangen.

1.3 Anmerkungen zum Aufbau der Arbeit

Die vorliegende Ausarbeitung gliedert sich in einen Hauptteil mit sechs Kapitel und einen Anhang, bestehend aus drei Teilen.

Kapitel 1: Einleitung Dieses Kapitel.

Kapitel 2: Grundlagen In Kapitel 2 werden Begriffe und Konzepte vorgestellt, die für das Verständnis der weiteren Ausführung vorausgesetzt werden müssen.

[1] zum Beispiel Interoperabilität, Sicherheit, Performance

Kapitel 3: Analyse Eine Betrachtung des Status-Quo und die Abgrenzung des Umfangs der Arbeit.

Kapitel 4: Entwurf Entwurf der Lösung.

Kapitel 5: Realisierung Beschreibung der konkreten Umsetzung des Designs und implementierungsspezifischer Details.

Kapitel 6: Zusammenfassung Abschließende Betrachtung der Arbeit, Ausblick und Probleme.

Zusätzliche Informationen sind im Anhang zusammengefasst:

Anhang A: Eingesetzte Werkzeuge Vorstellung der für die praktische Realisierung verwendeten Werkzeuge und Frameworks

Anhang B: Quellcode Ausgewählte Code-Fragmente, Klassen oder XML-Dateien.

Anhang C: Daten CD Dokumente, Software und Quellcode auf CD.

Im Anschluss an die Arbeit werden verwendete Begriffe, Abkürzungen und Akronyme in einem Glossar zusammengefasst, sowie die verwendeten Quellen im Literaturverzeichnis aufgelistet.

Anmerkung Die Arbeitssprache der Informatik, insbesondere die der Web-Technologien, ist Englisch. Aufgrund der Internationalität des Themas, der Vergleichbarkeit von Begriffen und der Klarheit der Aussagen wird im Rahmen dieser Arbeit darauf verzichtet, gängige Begriffe, die auch in deutschsprachigen Veröffentlichungen überwiegend in ihrer englischen Form verwendet werden, ins Deutsche zu übersetzen.

Kapitel 2

Grundlagen

Für das Verständnis der weiteren Kapitel dieser Arbeit ist eine gemeinsame Vorstellung der verwendeten Grundkonzepte unabdingbar. Dieses Kapitel gibt einen kurzen Überblick über verwendete Begriffe.
Das Ziel ist dabei nicht die umfassende Darstellung aller Sachverhalte, zur vertiefenden Lektüre seien die Referenzen empfohlen.

2.1 Semantic Web

Die semantische Klassifizierung von Information ist seit jeher ein Grundproblem der Informatik. Durch die rasante Entwicklung des Internet und der zunehmenden digitalen Speicherung von Information gewinnt sie zunehmend an Bedeutung.

Der Großteil der heutigen Inhalte des World Wide Web (WWW) ist für die Ansicht durch menschliche Betrachter ausgerichtet. Suchmaschinen finden unzählige Ergebnisse auf Anfragen, von denen die meisten jedoch irrelevant sind, weil sie nur nach der Schreibweise der gesuchten Begriffe (Syntax), nicht jedoch nach ihrer Bedeutung (Semantik) suchen.

Das Semantic Web bildet eine Erweiterung des bestehenden Webs, mit dem Ziel diese Probleme zu lösen. Durch die konsequente Bereitstellung semantischer Informationen und der Verwendung von Beschreibungslogik bzw. ihrer Auswertung entsteht so ein neues Netz von Informationen, dass von Software-Systemen und Agenten genutzt werden kann. Die W3C definiert den Begriff *Semantic Web* folgendermaßen:

> „The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework, which integrates a variety of applications using XML for syntax and URIs for naming.“ [39]

Grundlage für die Nutzung des Semantic Web als Anbieter und Verwerter von Informationen ist ein gemeinsames Verständnis der Architektur und der Begriffe. Aus diesem Grund werden die Komponenten des Semantic Web in den folgenden Unterabschnitten beschrieben.

2.1.1 Komponenten des Semantic Web

Das W3C hat eine Architektur für die Komponenten des Semantic Web entworfen. Ziel ist die Realisierung einer modularen, auf offenen Standards basierenden Plattform. Dies ermöglicht die Teilname unterschiedlicher Anbieter und Implementierungen an einem gemeinsamen Netzwerk.

Diese Architektur wurde von Tim Berners-Lee im Jahr 2000 vorgestellt [20] und ist in Abbildung 2.1 dargestellt. Zur Erörterung soll sie in drei Bereiche unterteilt werden: Präsentation, Beschreibungslogik-Systeme und Sicherheit.

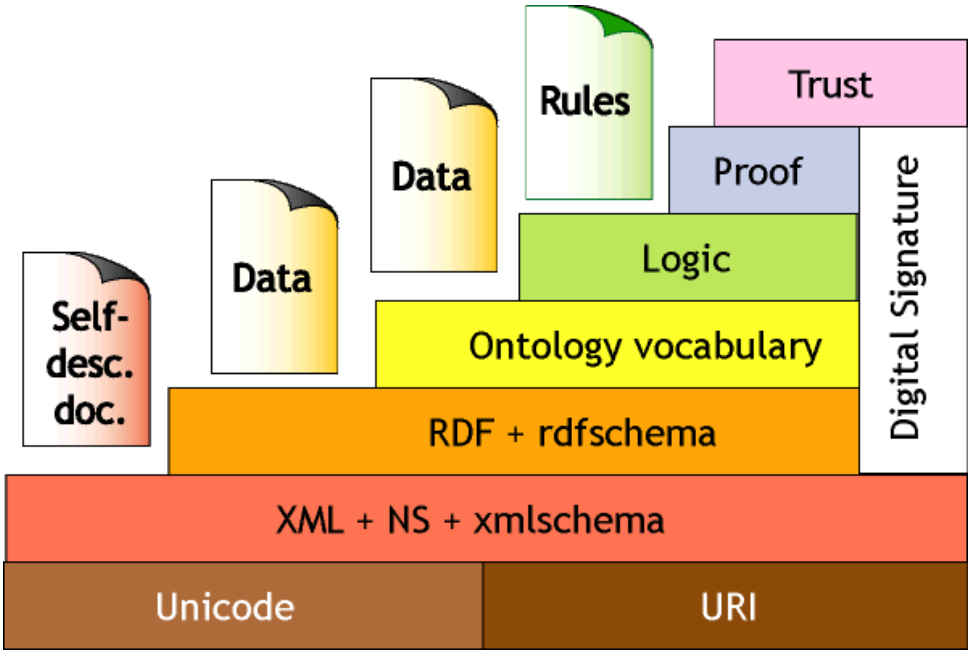

Abbildung 2.1: Semantic Web Architektur

Die Präsentationsschicht wird durch die Basis-Technologien URI, Unicode, XML und RDF gebildet. Sie bilden die Grundlage für die Darstellung von Datenstrukturen.
Die Bearbeitung und Verwertung dieser Daten erfolgt durch Beschreibungslogik-Systeme, bestehend aus Ontologien- und Logik-Schicht.
Die Konzepte Proof, Signature und Trust werden unter dem Begriff Sicherheit zusam-

mengefasst, wobei sich Sicherheit nicht nur auf Autorisierung und Authentifizierung bezieht, sondern im Sinne von Vertrauen und Beweisbarkeit verstanden werden kann.

Die einzelnen Bereiche werden in den folgenden Unterabschnitten kurz erläutert. Der Bereich Beschreibungslogik-Systeme wird aufgrund seiner Bedeutung für diese Arbeit in einem eigenen Abschnitt besprochen (2.2).

2.1.2 Repräsentation

Das Fundament der Architektur bilden Technologien, die bereits als Standard etabliert sind: URI, Unicode und RDF über XML. Diese Sprachen des Semantic Web werden in diesem Abschnitt kurz vorgestellt.

Unicode Der Standard „Unicode“[1] weist jedem in einer natürlichen Sprache verwendeten Zeichen eine Nummer zu und schafft so eine syntaktisch eindeutige, sprach-, plattform- und programmunabhängige Basis für die globale Anwendbarkeit des Sematic Web [28].

URI Die Uniform Resource Identifier sind ein Adressierungs-Standard des W3C und ermöglichen die eindeutige Identifikation einer Quelle. Alle Begriffe bzw. Objekte verfügen über eine spezifische Adresse, über die sie angesprochen werden können. Die heute im Internet verwendeten URLs [2] stellen eine Untermenge der URIs dar.

XML Die Extensible Markup Language (XML) ist ein Standard zur Darstellung von hierarchischen Datenstrukturen in Dokumenten. XML definert dabei den Aufbau dieser Dokumente durch eine maschinenlesbare Baumstruktur. Seine einfache Bedienung, der universelle Einsatz und die gute Werkzeugunterstützung für diverse Systeme und Programmiersprachen haben XML zu einer wichtigen Sprache für den Austausch von Informationen werden lassen.

RDF Die Verwendung von XML alleine reicht jedoch nicht aus um eine einheitliche Semantik bereitzustellen. Durch die Verwendung von Unicode und URI, sowie Schemata und Namespaces lassen sich eindeutig identifizierbare und lesbare Dokumente erzeugen. Ohne ein übergeordnetes Sprachkonzept, das die Universalität von XML hinreichend einschränkt, kann allerdings keine gemeinsame Basis erzeugt werden. Jedem Akteur im

[1] Internationale Norm ISO/IEC 10646
[2] Uniform Resource Locators

Semantic Web wäre es ansonsten freigestellt, eine eigene Darstellungsform von Informationen zu wählen.

Die erforderliche Einschränkung wird durch das *Resource Description Framework* (RDF) erreicht. RDF setzt auf XML auf, und unterstützt eine einfache *Statement*-Syntax zur Modellierung der Beziehung zwischen Entitäten, das RDF-Graph-Model (Abbildung 2.2, [29]).

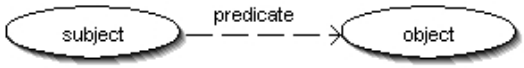

Abbildung 2.2: RDF Graph Model

Ein RDF-Statement besteht aus Subjekt, Prädikat und Objekt, wobei Subjekt und Objekt die Knoten und das Prädikat die Kante des Graphen darstellt. Mit RDF-Schemata (RDFS) lässt sich das zu verwendende Vokabular dieser Beziehung definieren.

2.1.3 Sicherheit

Der Aspekt Sicherheit bezieht sich wie bereits angedeutet auf die Konzepte Verification, Proof und Trust.

Digitale-Signaturen Die Grundlage aller Sicherheitskonzepte ist die digitale Signatur. Sobald es um den Zugriff auf Beschreibungslogik-Systeme geht, in denen die Aussagekraft und Identifikation der Quelle eine besondere Bedeutung erlangt, ist es von entscheidender Bedeutung für die höheren Konzepte Beweisbarkeit und Vertrauen, dass die Herkunft von Informationen nachvollziehbar und beweisbar ist.

Da sich das Modell des W3C auf die Technologien XML und RDF stützt, und die Signatur und Verschlüsselung von XML-Dokumenten technisch keine Hürde mehr darstellt, ist dieser Teil der Architektur bereits heute erfüllbar. Es kann dadurch sichergestellt werden, dass nur autorisierte Akteure an kritischen Teilen einer Kommunikation teilnehmen und die Berechtigung zum Zugriff auf Systeme und Informationen nachprüfbar ist.

Proof Entscheidend für das Vertrauen in die vom Beschreibungslogik-System gefolgerten Antworten ist die Nachprüfbarkeit der gelieferten Ergebnisse.

Die Systeme sollen daher auf Anfrage nicht nur Ergebnisse sondern auch Beweisketten liefern können, anhand derer sich nachvollziehen lässt, welche Beziehungen und welche Aussagen der Wissensbasis zu den Schlussfolgerungen geführt haben.

Trust Die Kombination von Proof und Signatur ergibt ein „Web of Trust“. Das Vertrauen basiert zum einen auf der Sicherheit, dass die Überbringer der Information identifiziert werden können und zum anderen auf Beweisen, die sicherstellen dass die Informationen auf transparentem Wege aus benannten und einsehbaren Quellen gewonnen wurde.

2.2 Beschreibungslogik-Systeme

Beschreibungslogik-Systeme (Description Logic-Systems, DL) bilden die Grundlage logischer Entscheidungs-Prozesse im Sematic Web. Sie sind eine Untergruppe von Wissensrepräsentations-Systemen.

Sie bestehen im Wesentlichen aus den Komponenten Schnittstelle, Inferenz-System und Wissenbasis (siehe Abbildung 2.3).

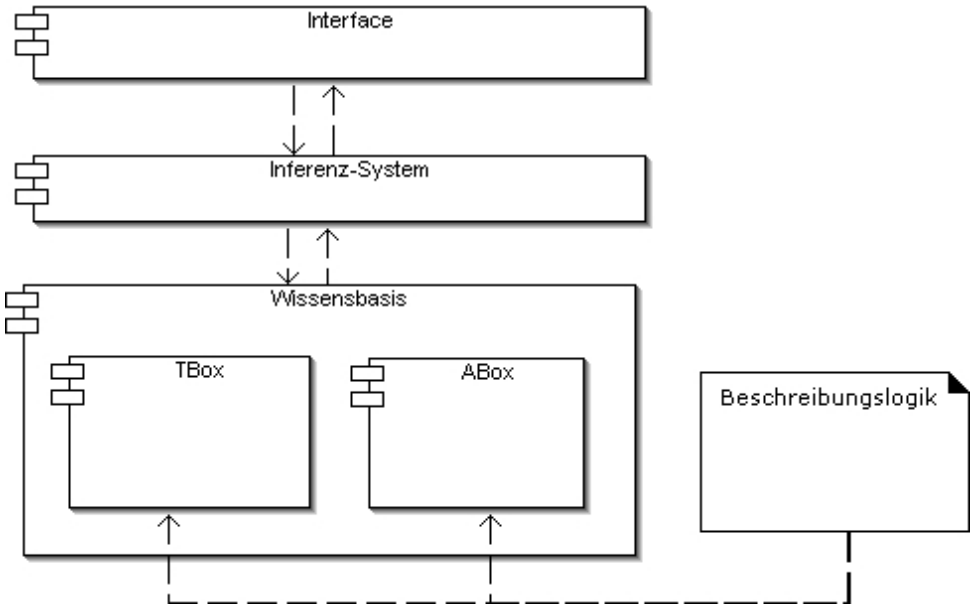

Abbildung 2.3: Architektur eines Beschreibungslogik-Systems

2.2.1 Wissensbasen

Wissenbasen oder auch Knowledge-Bases (KB) speichern die Informationen die das Beschreibungslogik-System zur Arbeit benötigt. Eine Wissensbasis besteht aus einem Tupel (T, A) aus einer TBox T und einer ABox A.

Dabei beschreibt die TBox intensionales Wissen durch eine Menge von terminologischen Axiomen, und die ABox extensionales Wissen durch eine Menge von Aussagen über Individuen. Anders ausgedrückt bedeutet dies, dass in der TBox einer Wissenbasis die allgemeinen Konzepte und Relationen einer Domäne dargestellt werden (Klassen), während in der ABox konkrete Instanzen dieser Konzepte realisiert werden, die untereinander durch Rollen in Beziehung stehen können (Objekte).

2.2.1.1 Ontologien

Der Begriff Ontologie stammt ursprünglich aus der Philosophie. Er bezeichnet die Lehre vom Sein, und beschäftigt sich mit dem Wesen fundamentaler Entitäten. In die moderne Informatik fand er im Rahmen der KI-Forschung und in den letzten Jahren besonders im Rahmen der Semantic Web Entwicklung Einzug.

Im Sinne der Informatik bezeichnet eine Ontologie die Beziehungen zwischen Symbolen und Objekten. Diese Beziehung ist in Abbildung 2.4 [28] dargestellt.

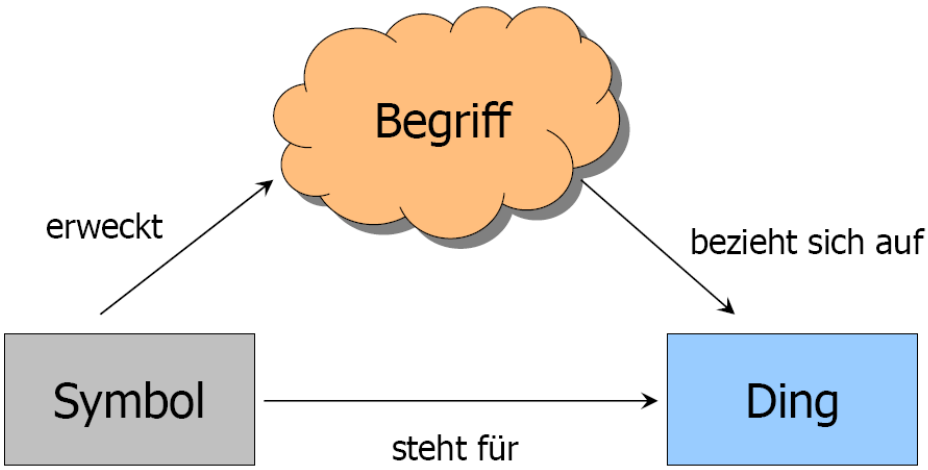

Abbildung 2.4: Ontologie

Dabei wird davon ausgegangen, dass ein Symbol, also zum Beispiel ein Wort oder ein Zeichen, sowohl eine syntaktische als auch eine semantische Bedeutung besitzt. Im Sinne der Syntax stellt ein Symbol einen Platzhalter für ein Objekt („Ding“) dar. Der syntaktische Bezug ist unmittelbar, enthält jedoch in sich kein konzeptionelles Verständnis für den mit dem Symbol auszudrückenden Inhalten.

Der Begriff oder das Konzept, für das ein Symbol steht, erschließt sich erst durch das semantische Verständnis des Umfeldes, in dem dieses Symbol verwendet wird. Der Wider-

spruch zwischen syntaktischer und semantischer Bedeutung eines Symbols wird umso größer, je höher der Abstraktionsgrad des verwendeten Symbols ist.

Beispiel Die realitätsgetreue Abbildung eines Gegenstandes beinhaltet einen niedrigen Abstraktionsgrad, Symbol, Begriff und Objekt sind annähernd deckungsgleich. Ein geschriebenes Wort dagegen besitzt einen hohen Abstraktionsgrad, da es aufgrund seiner Gestalt keine Rückschlüsse auf das beschriebene Objekt zulässt. Die Vorstellung vom Begriff, für den es steht, erschliesst sich dem Betrachter nur über sein Hintergrundwissen.

Menschen sind den Umgang mit dem Widerspruch zwischen Symbol und Objekt gewohnt und haben im Alltag nur selten Probleme, die Abweichung von Konzept und Objekt umzusetzen. Fallstricke bieten hier aber Synonyme und Homonyme, bei denen sich die Bedeutung erst aus dem größeren Zusammenhang der Verwendung des Symbols erschließt.

Maschinen dagegen fehlt das implizite Verständnis für solche Zusammenhänge, sie arbeiten mit explizit angegebenen Beziehungen und sind daher auf die syntaktische Bedeutung eines Symbols beschränkt.

Dieses Defizit soll durch die Entwicklung von Wissensrepräsentations-Systemen verringert werden.

2.2.2 Formale Definition

In der Wissenschaft existieren mehrere Definitionen des Begriffs „Ontologie“ bei seiner Verwendung in der Informatik. Eine der am häufigsten verwendeten ist die von Gruber:

> „An ontology is an explicit specification of a conceptualization.“ [18]

Wie bereits im vorigen Abschnitt erläutert, müssen Spezifikationen für Maschinen explizit angegeben werden, da ihnen unser implizites Verständnis nicht zur Verfügung steht. Die verwendeten Konzepte müssen eindeutig definiert werden.

Der Begriff Konzeptualisierung bezieht sich auf ein abstraktes Modell eines bestimmten Ausschnittes der realen Welt [21]. Da die Modellierung von Ontologien sehr anspruchsvoll ist und nicht umfassend sein kann, wird davon ausgegangen, dass von vornherein nur einige Aspekte in einer Ontologie abgebildet werden (Domäne).

Um die Verwendung in Wissensrepräsentations-Systemen zu ermöglichen, muss die Spezifikation in einer formalen und maschinenlesbaren Sprache erfolgen.

2.2.3 Inferenz-System

Die Kern-Funktionalität eines Beschreibungslogik-Systems stellt das Inferenz-System (auch *Reasoner*)zur Verfügung. Das Inferenz-System ermöglicht den Zugriff auf die in der Wissensbasis gespeicherten Datenstrukturen und erlaubt das Ziehen impliziter Schlussfolgerungen aus explizit formulierten Aussagen.

Die drei wichtigsten Inferenzdienste sind:

Konsistenz Durch Konsistenzprüfung lässt sich die Frage nach der Erfüllbarkeit von Konzepten oder Aboxen beantworten. Der Widerspruch in der folgenden einfachen KB wird so aufgedeckt.

```
(disjoint C1 C2)
(instance a C1)
(instance a C2)
```

Da definiert ist, dass die Konzepte `C1` und `C2` disjunkt sind, darf keine Instanz von `C1` gleichzeitig Instanz von `C2` sein.

Subsumption Durch Subsumption lässt sich feststellen, ob ein Konzept `C1` allgemeiner ist als ein Konzept `C2`. Daraus folgt unter anderem, dass alle Instanzen von `C2` automatisch Instanzen von `C2` sind.

```
(implies C1 C2)
(instance a C2)
```

Instanz Der Dienst „Instanz“ kann die Frage beantworten, ob ein ABox-Objekt `a` eine Instanz eines TBox-Konzepts `C` ist.

Bezogen auf das Beispiel unter „Subsumption“ kann die Frage für `a` und `C1` positiv beantwortet werden.

Diese drei Inferenzdienste werden bei Molitor [30] als Standardinferenzen bezeichnet, da sie „von (fast) allen Beschreibungslogik-Systemen bereitgestellt werden, unerlässlich für die Strukturierung einer Wissensbasis sind und ihre Untersuchung seit Einführung der ersten BL-Systeme einen Kernpunkt der Arbeit im Bereich Beschreibungslogiken bildet.“

2.3 Web Services

Web-Services haben sich in den letzten Jahren von einer Idee zu einer einsetzbaren Basis-Technologie für die Kommunikation von Software-Systemen entwickelt.

Der Standard hat sich trotz Defiziten in den Bereichen Interoperatibilität und Sicherheit für die Kommunikation von Software-Systemen etabliert und wird im Hinblick auf die Service orientierte Architektur (SOA) zunehmend für die Realisierung modularisierter, spezialisierter Teilsysteme eingesetzt.

Da in den sich in den ersten Jahren der Entwicklung Marketing und Technik gegenseitig mit Erwartungen und Ankündigungen übertroffen haben und eine Unzahl von Definitionen für Web Services kursieren [36], sei an dieser Stelle die Definition des W3C als Grundlage für den Begriff Web Service zu Hilfe genommen.

> A web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems.
>
> These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.“ [22]

Es handelt sich um Software-Systeme mit klar definierten Schnittstellen zur Kommunikation mit anderen Systemen. Die Schnittstellen werden durch ein XML-Dokument beschrieben. Die *Web Service Description Language* (WSDL) ist standardisiert, ebenso wie das eingesetzte XML-Nachrichtenformat SOAP.

Das Protokoll zur Kommunikation wird durch den Standard nicht definiert, klassischerweise erfolgt die Kommunikation über HTTP, jedoch sind auch Alternativen wie E-Mail (SMTP), File-System (IO) oder Messaging (JMS) möglich.

Die grundlegenden Eigenschaften von Web Services, der Ablauf der Kommunikation zwischen Akteuren und die verwendeten Standards, werden in diesem Abschnitt erklärt.

2.3.1 Komponenten von Web Services

An der Kommunikation mittels Web Services sind drei Akteure beteiligt (Abbildung 2.5). Diese Akteure sind der Dienste-Anbieter, der Dienste-Konsument, und die Registry ([2]).

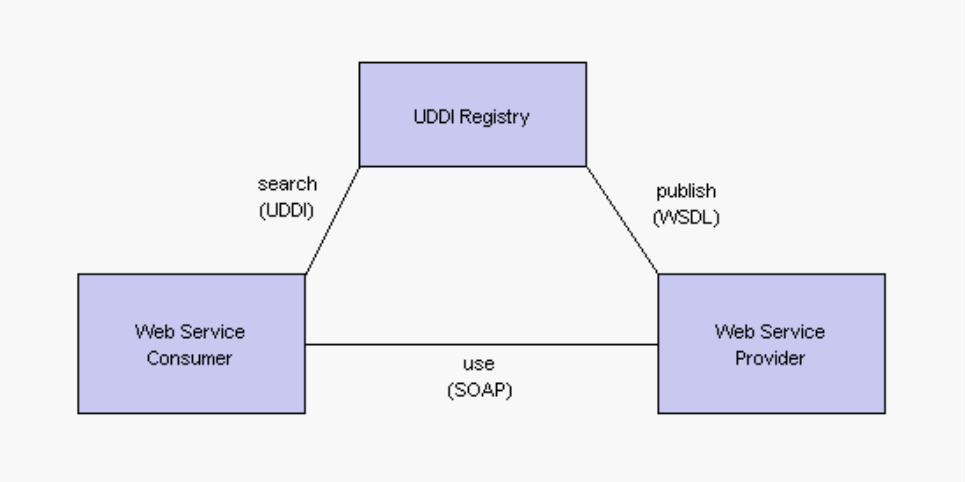

Abbildung 2.5: Aufbau

Der Dienste-Anbieter Der Anbieter eines Services entwickelt eine Softwarekomponente mit klar definierten Schnittstellen. Diese Schnittstellen und den URI, unter der der Dienst erreichbar sein wird, veröffentlicht er in einem standardisierten Format (WSDL).

Die UDDI Registry Eine „Universal Definition, Discovery and Integration“ (UDDI) Registry ist ein Standard für Suchdienste für Web Services. Sie liefert dem Benutzer mögliche Dienste-Anbieter und deren Angebote in Form der registrierten WSDL-Datei. Eine Registry kann öffentlich sein, etwa für Dienstleistungen wie Börsenkurse, die weltweit angeboten werden, oder sie kann auf einen Teil des Netzwerkes (Intranet) beschränkt sein, und so beispielsweise Web Services unternehmensintern anbieten.

Der Dienste-Konsument Der Konsument sucht den benötigten Web Service in der UDDI Registry. Als Antwort auf die Suchanfrage erhält er die Web Service-Beschreibung des Anbieters mit den Informationen für den Zugriff auf seinen Dienst.

Der dynamische Zugriff auf Services zur Laufzeit der Client-Software ist derzeit nur eingeschränkt möglich.

Ist dem Konsumenten die URI des Dienstes bereits bekannt, kann der Zugriff auch ohne den Umweg über die Registry erfolgen. Dies ist insbesondere im Bereich SOA gegeben, da die einzelnen Module eines Systems zwar über Web Services kommunizieren aber nicht unbedingt öffentliche Schnittstellen für Clients außerhalb des Systems anbieten.

2.3.2 SOAP

SOAP ist ein leichtgewichtiges Protokoll für den Informationsaustausch in dezentralen, verteilten Systemen. Die Kommunikation der Systeme erfolgt durch den Austausch von XML-Nachrichten.

Eine SOAP Nachricht besteht aus drei Teilen: dem SOAP Envelope, einem SOAP Header und dem SOAP Body. Dieser Aufbau ist in Abbildung 2.6 dargestellt.

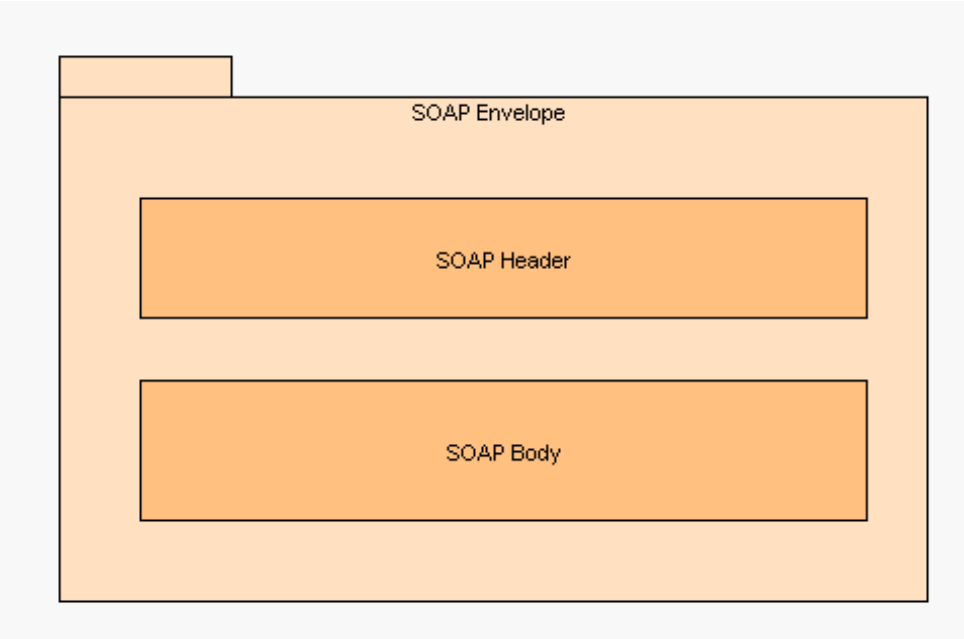

Abbildung 2.6: Aufbau einer SOAP-Nachricht

Envelope Der SOAP Envelope ist der „Briefumschlag“ der Nachricht. Er bildet einen strukturierten Rahmen, der den Inhalt der enthaltenen Nachricht beschreibt. Insbesondere legt er den Adressaten fest und implementiert ein einfaches Prioritätenmodell, das die Dringlichkeit des Inhaltes beschreibt.

Header Der Header der Nachricht ist optional. Durch die Informationen im SOAP Header können Nachrichten um Zusatzinformationen erweitert werden, die für die Kommunikation zweier Systeme nicht zwingend erforderlich sind. Der Adressat kann die Nachricht auch ohne Kenntnisse der speziellen Header-Informationen bearbeiten. Für typische Anwendungsfälle solcher Zusatzinformationen, wie Autorisierung, Transaktionsmanagement oder Sitzungsverwaltung, können die Attribute im SOAP Header auch als verpflichtend gekennzeichnet werden. Der Empfänger muss dann in der Lage sein, die Header-Informationen zu verstehen, ansonsten kann er die Nachricht nicht verarbeiten.

Body Der SOAP Envelope enthält ein oder mehrere Body-Elemente. Im SOAP Body befindet sich die eigentliche Information der Nachricht. Dies können Dateninhalte in Form von Strings sein, aber auch Aufrufe von Methoden auf dem Server. Der genaue innere Aufbau des Body-Elements wird durch die Verwendung von Message Styles bestimmt. Die am häufigsten verwendeten Stile werden im folgenden Abschnitt erläutert.

2.3.3 Message Styles

Der Nachrichtenstil beschreibt, wie der Body einer SOAP-Nachricht aufgebaut ist und wie die zu übertragenden Daten repräsentiert werden [1]. Grundsätzlich sind zwei Stile zu unterscheiden: Remote Procedure Calls (RPC) und Messaging.

Bei RPC erfolgt wie der Name bereits impliziert der Aufruf einer Methode, der SOAP Body enthält die Bezeichnung der verwendeten Methode, ggf. zu verwendende Parameter und Informationen über den Typ des Rückgabewerts der aufgerufenen Funktion.
Die Verwendung des RPC-Stils erfordert eine synchrone Kommunikation zwischen Client und Service.

Beim Messaging-Stil besteht der SOAP Body aus einem XML-Dokument, das den Inhalt der Nachricht beschreibt. In diesem Fall kann, muss jedoch nicht, auch eine asynchrone Kommunikation realisiert werden.

Mit diesen grundlegenden Nachrichtenstilen lassen sich unter Verwendung von verschiedenen Nachrichtenformaten und Kodierungsregeln verschiedene Typen von SOAP Service Aufrufe erstellen. Die in der Praxis wichtigsten sind RPC/Encoded, Document/-Literal und Wrapped. Welches Verfahren zum Einsatz kommt bestimmt der Dienste-Anbieter durch Festlegung in der WSDL-Datei zu seinem Service.

RPC/Encoded Ein Aufruf vom Typ *RPC/Encoded* stellt einen Methodenaufruf dar, wie er weiter oben für RPC bereits vorgestellt wurde. Die Verwendung ist der von verwandten Konzepten wie CORBA oder RMI sehr ähnlich. Die Kodierung der Nachricht und der Parameter erfolgt nach der SOAP Spezifikation, der Einsatz eigener Datentypen ist nicht möglich.
RPC/Encoded ist der Standard bei der Erstellung von Web Services mit Apache Axis.

Document/Literal Der SOAP Body besteht beim Typ *Document/Literal*[3]aus einem XML-Dokument, dessen Format über ein Schema definiert werden kann. Das Dokument kann verwendet werden, um einen RPC zu imitieren, bei der Anwendung kann dann die Definition eigener Datentypen als Parameter im Schema erfolgen.
Die XML-Nachrichten müssen jedoch keinen Methodenaufruf realisieren, sondern

[3] oft auch nur doc/lit

können auch die Verarbeitung des SOAP Bodys als Dokument ermöglichen. Dies erlaubt einen sehr flexiblen Einsatz, der allerdings auf Kosten der Kompatibilität gehen kann.

Wrapped Der Typ *Wrapped* ist eine Mischform der beiden zuvor genannten Nachrichtenformate. Er implementiert einen Methodenaufruf, verpackt jedoch die Parameter in ein Top-Level Element (wrapped) und versendet die Nachricht im *Document/-Literal*-Stil als XML Nachrícht. *Wrapped*-Services sind die Standard-Einstellung bei Services die mit dem .NET-Framework erstellt werden.

Kapitel 3

Analyse

In Kapitel 2 wurde eine Einführung in die allgemeinen Grundlagen der verwendeten Konzepte gegeben. Das folgende Kapitel beschäftigt sich konkret mit den im vorliegenden Szenario auftretenden Technologien.

Dabei erfolgt insbesondere eine Beschreibung der bestehenden Systemarchitektur und -komponenten sowie die Vorstellung der Sprachkonzepte. Basierend auf den dabei gewonnenen Erkenntnissen schließt sich eine Anforderungsanalyse für die zu erstellende Architektur an. In diesem Zusammenhang werden gegebenenfalls erforderliche Einschränkungen im Bezug auf die Realisierung des Prototypen vorgenommen und die Aufgabenstellung aus Kapitel 1 konkretisiert.

3.1 RACER

Die RACER-Engine[1] ist ein Beschreibungslogik-System, bzw. Wissensrepräsentations-System, wie in Abschnitt 2.2 beschrieben. Die Weiterentwicklung des Kern-Systems wird von Prof. Ralf Möller, Prof. Volker Haarslev und Michael Wessel betrieben.

Eigenschaften von RACER RACER implementiert die Beschreibungs-Logik $ALCQHI_{R^+}$[2]. Dadurch können neben atomaren Rollen und Konzepten auch Zahlenbeschränkungen (constraints) und hierarchische, inverse sowie transitive Rollenbeziehungen realisiert werden.

Die Engine beinhaltet eine Reihe von Optimalisierungs-Algorithmen, die performante Inferenz-Dienste über mehrere Wissensbasen ermöglichen. Wissensbasen können in der

[1] Renamed ABox and Concept Expression Reasoner, [34]
[2] auch: $SHIQ$

system-eigenen, am Standard KRSS[3] orientierten Beschreibungssprache RDL[4], oder in OWL bzw. DAML/OIL modelliert werden.

RACER verwendet bei der Beantwortung von Fragen die *Open-World-Assumption* (OWA). Bei dieser Annahme wird davon ausgegangen, dass eine Wissensbasis nicht die gesamten Konzepte, Beziehungen und Individuen einer Welt beinhalten kann. Daraus folgt, dass eine eindeutige Antwort aus eine Frage nur möglich ist, wenn sich die Wahrheit der verwendeten Aussage beweisen lässt. Eine explizite Aussage über die Unwahrheit einer Aussage erfolgt nicht.

Ein weiteres Kern-Konzept von RACER ist die *Unique-Name-Assumption* (UNA). Sie fordert, dass sich verschiedene Namen in einer ABox auch auf verschiedene Individuen beziehen.

Einsatz der Software Die Implementierungssprache der RACER-Engine ist LISP. Das System kann wahlweise mit File-I/O oder als Server-Dienst betrieben werden. Der Einsatz als Stand-Alone-Anwendung ist möglich, in diesem Fall erfolgt der Zugriff von Clients entweder über TCP oder HTTP/DIG bzw. interaktiv auf Kommandozeilen-Ebene.

Bei der Verwendung als Server bildet RACER in Verbindung mit zusätzlichen Komponenten ein mehrschichtiges Software-System, bestehend aus der Engine, einem Proxy und einem Client-API.

Diese Komponenten werden in den folgenden Unterabschnitten beschrieben. Abbildung 3.1 skizziert den Aufbau des Systems. Den Zugriff auf die Wissensbasen und die Inference-Engine ermöglicht eine umfangreiche API, die in Abschnitt 3.2 vorgestellt wird.

3.1.1 Der RACER-Proxy

Der RACER Server selbst unterstützt kein Multithreading, das heisst die Verbindung zu einem Server ist während der Bearbeitung von Anfragen blockiert. Um dennoch den Zugriff mehrerer Benutzer performant bewältigen zu können, wird ein Proxy vorgeschaltet.

Dieser RACER-Proxy verwaltet die Kommunikation zwischen den Clients und einem oder mehreren RACER-Servern. Zusätzlich stellt er erweiterte Funktionen für die Clients bereit (vgl. [13]):

[3] Knowledge Representation System Specification (KRSS)
[4] RACER Description Language

Weiterleitung von Anfragen Der Proxy nimmt Anfragen von Clients entgegen und leitet sie an einen Server weiter. Der Zugriff ist dabei für den Client transparent, das heisst für den Zugriff auf den Server entwickelter Client kann ohne Umstellungen auch über einen RACER-Proxy kommunizieren.

Unterstützung von Mehrbenutzerbetrieb Beliebig viele Clients können gleichzeitig auf den RACER-Proxy zugreifen. Die Anfragen werden nicht gleichzeitig bearbeitet[5], sondern werden mit Hilfe einer Queue[6] verwaltet und auf die mit dem Proxy verbundenen Server verteilt.

Unterstützung mehrerer Anfragen eines Clients Stehen genug freie Server zur Verfügung, kann der Proxy Anfragen des Clients asynchron verarbeiten. Der Client schickt eine Liste von Anfragen, die dann auf mehrere Server verteilt werden. Dies führt zu einer schnelleren Auswertung der Anfragen.

Publish/Subscribe-Mechanismus Clients können sich vom Proxy per TCP oder Mail über Änderungen an den KBs der Server informieren lassen. Dies ist nützlich, wenn Anfragen gestellt wurden, deren Ergebnisse sich durch spätere Manipulationen verändern. In diesem Fall wird der Client benachrichtigt.

Alle Server die der RACER-Proxy verwaltet, müssen über identische Knowledge-Bases verfügen. Aus diesem Grund muss der Proxy die Handhabung von Queries und Statements unterscheiden und im letzteren Fall mittels Broadcast alle beteiligten Server über die verändernden Zugriffe an der KB informieren. Dies hat zur Folge, dass Server, die vom Proxy verwaltet werden, auch nur über diesen Proxy angesprochen werden dürfen.

3.1.2 Das JRacer-API

Mit JRacer steht dem Java-Entwickler ein API für den Zugriff auf RACER-Server (oder Proxies) zur Verfügung. Das Interface implementiert die Verbindung über TCP und den Transfer von nRQL-Queries und Befehlen.

Komplexere RACER-Funktionen werden durch die API gekapselt und lassen sich als Java-Methoden aufrufen, ohne dass dafür Kenntnisse der nRQL-Syntax erforderlich wären.

[5] siehe fehlende Unterstützung im Server
[6] FIFO-Speicher

3.1.3 RICE

RICE[7] ist ein grafischer Client für den Zugriff auf einen RACER-Server, bzw. den Zugriff auf einen Proxy.

Der Client ist eine Java-Anwendung und wurde mit dem Ziel entwickelt, ein einfach zu benutzendes Interface zur Manipulation von DL-basierten Knowledge-Bases bereitzustellen [12]. Neben der Kontrolle der Verbindung zu RACER über das JRacer-API[8] und einer interaktiven Kommandozeilen-Implementierung, zeichnet sich das Programm durch die grafische Darstellung der in einer KB enthaltenen Datenstrukturen aus.

3.1.4 Architektur des Systems

Die in den vorangehenden Abschnitten vorgestellten Komponenten bilden wie bereits erwähnt das RACER-System. Die Architektur dieses Systems lässt sich wie in Abbildung 3.1 dargestellt skizzieren.

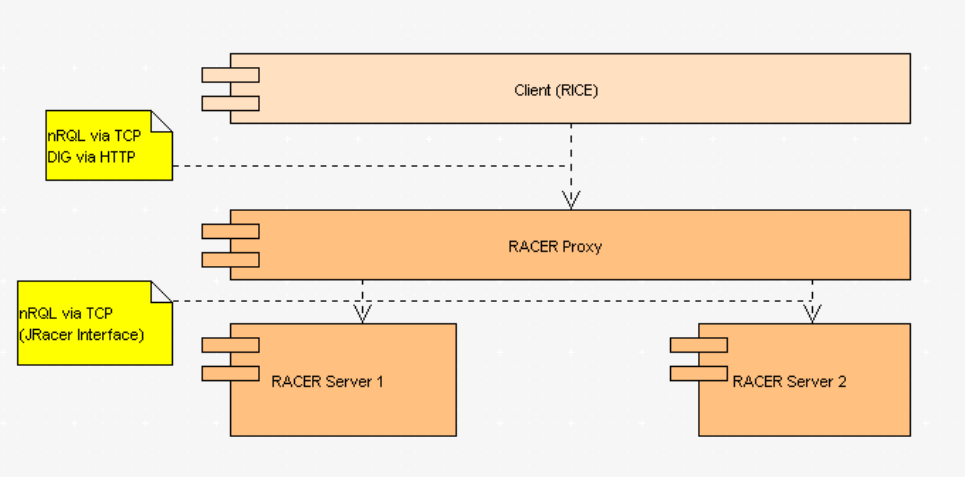

Abbildung 3.1: Alte Architektur

Die Komponenten Client, Proxy und Server sind, wie erwähnt, jeweils eigenständige Programme, die ausschließlich über HTTP oder TCP kommunizieren.

[7] RACER Interactive Client Environment
[8] Die in RICE verwendete Version von JRacer unterscheidet sich von der vorgestellten, aktuellen Version auf den RACER-Seiten

3.2 Die new RACER Query Language

Im Folgenden soll die *new RACER Query Language* (nRQL)[9] vorgestellt werden. Bezogen auf die Darstellung des Aufbaus von Beschreibungslogik-Systemen (Abbildung 2.3) stellt nRQL die Schnittstellenschicht des Systems dar. Sie setzt auf den internen APIs von RACER auf und stellt Anfrage-Methoden zur Verfügung.

nRQL erlaubt die Verwendung von Variablen in Queries und ermöglicht komplexe Anfragen an die Wissensbasis des RACER-Systems. Eine komplette Beschreibung der Sprache und die detaillierte Erläuterung der Befehle liefert [5]. An dieser Stelle erfolgt nur ein kurzer Überblick über die Hauptelemente der Syntax.

3.2.1 Die Syntax einer nRQL-Query

Eine nRQL-Query besteht aus einem Befehl, einem Head und einem Body. Die Syntax folgt dabei der LISP-Syntax. Das heisst, eine Query ist eine Zeichenfolge, deren einzelne Elemente durch Klammerung von einander getrennt werden. Der gesamte Ausdruck wird ebenfalls durch Klammern eingeschlossen.

```
(<command> (<head>) (<body>))
```

Der Head kann leer bleiben oder ein oder mehrere Objekte enthalten. An diese können die Antworten gebunden werden. Eine Query mit leerem Head-Element ist eine boolesche Frage, auf die mit wahr oder falsch geantwortet wird (s.u.). Objekte im Head können Variablen oder ABox-Individuen sein, wobei Variablen durch ein dem Bezeichner vorangestelltes Fragezeichen gekennzeichnet werden („?x“).

Der Body der Query enthält die Bedingungen der Frage, ähnlich der WHERE-Klausel einer SQL-Query. Er besteht aus Query-Atomen, die Rollen, Konzepte, Individuen und Variablen enthalten können. Komplexe Anfragen werden durch die Verknüpfung mit den bekannten logischen Operatoren AND, OR, NOT und INV[10] ermöglicht.

Beispiel Das folgende Beispiel demonstriert den Einsatz der nRQL-Syntax unter Verwendung des `retrieve`[11]-Befehls.

```
(retrieve (?x ?y) (AND (?x ?y has-child) (?x woman))))
```

Die Query sucht alle in der ABox der KB benannten Mütter und ihre Kinder (die Variablen ?x und ?y sind durch die Rollenbeziehung „has-child“ verbunden, durch die weitere

[9] gesprochen: *nercle*
[10] Inverse
[11] vergleichbar mit `SELECT` in SQL

Forderung, dass ?x eine Instanz des Konzepts „Frau“ sein soll, werden nur weibliche Elternteile berücksichtigt). Dieses Beispiel setzt voraus, dass die entsprechenden Konzepte und Rollen in der TBox der KB definiert sind.

3.2.2 Die Syntax einer nRQL-Antwort

Bei den nRQL-Antworten müssen zwei Fälle unterschieden werden. Eine Antwort kann entweder aus einem AnswerSet oder einem Symbol bestehen.

Symbole werden verwendet, wenn es sich bei der Frage um eine boolesche Frage handelt, wenn also keine Bindung erfolgt sondern die Erfüllbarkeit der Bedingung beantwortet wird.

Die verwendeten Symbole von nRQL sind `T` und `NIL`. `T` steht dabei für „wahr“ (true), `NIL` für „nicht beweisbar“. Wie weiter oben im Abschnitt bereits erläutert, ist wegen der *Open-World-Assumption* keine Unterscheidung zwischen „falsch“ und „nicht beweisbar“ möglich.

Bezieht sich die Antwort auf eine Frage, die Bindungen zwischen Objekten erfordert, so besteht die Antwort der Engine aus einer Liste von Listen von Bindungen. Es sei denn, die Ergebnismenge ist leer, in diesem Fall wird wieder `NIL` zurückgegeben.

Eine Bindung zwischen zwei Objekten wird in der nRQL-Syntax durch Klammerung erzeugt. Ein Antwortelement besteht aus einer oder mehreren solcher Bindungen und wird ebenfalls in Klammern eingeschlossen, ebenso wie die gesamte Liste von Antworten.

Beispiel Das folgende AnswerSet

```
( ((?x P1) (?y C1)) ((?x P2) (?y C2)) )
```

beinhaltet als Antwort auf die Frage `(retrieve (?x ?y) (?x ?y has-child))` zwei Eltern-Kind-Paare: (P1,C1) und (P2,C2). Dabei ist der Elternteil jeweils an die Variable ?x, das Kind an die Variable ?y gebunden.

Weitere Beispiele für die nRQL-Syntax finden sich in [5] und in den späteren Kapiteln dieser Arbeit.

3.3 Die OWL Query Language

Die *Web Ontology Query Language* (OWL-QL) ist der Entwurf eines standardisierten Zugriffs auf Wissensbasen und Reasoner Systeme. Die zugrunde liegende *Web Ontology Language* (OWL) wird in [24, 25] beschrieben. Zum Zeitpunkt des Entstehens dieser Arbeit hat die Spezifikation den Status eines W3C Standard Kandidaten.

Die OWL-QL Spezifikation definiert zwei Bereiche der Client-Server Kommunikation: das Nachrichtenformat und ein Kommunikationsprotokoll. Zum einen wird die Struktur von Frage- und Antwort-Dokumenten in XML-Schemata ([40]) definiert, zum anderen wird ein Protokoll für den Dialog-Prozess zwischen Client und Server vorgeschlagen.

Dabei ist OWL-QL ein schwacher Standard, da die Spezifikation keine spezifische externe Syntax sondern nur die Struktur vorgibt.

Diese Bestandteile, also der Aufbau des Protokolls und die Struktur der Nachrichten, werden in diesem Abschnitt erläutert.

3.3.1 Der Dialog

Eine Besonderheit von OWL-QL ist die Unterstützung iterativer Abfragen. Mit einen definierten Dialog ist es möglich, nur eine begrenzte Anzahl von Antworten sofort vom Server zu erhalten. Mit einem Continuation Token, das eine Referenz auf eine vorher gestellte Anfrage darstellt, lassen sich später weitere Ergebnisse zu einer Frage abrufen.

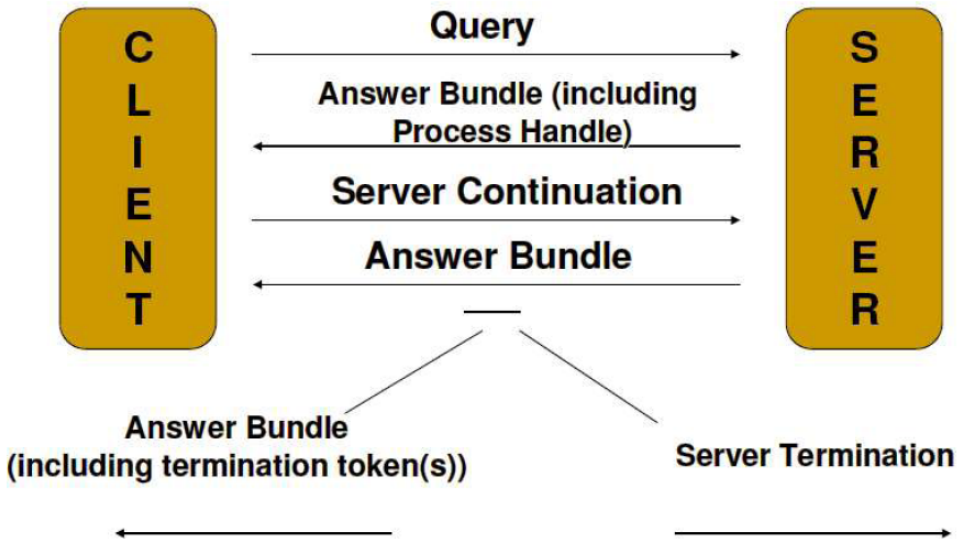

Abbildung 3.2: OWL-QL Dialog Schema

Abbildung 3.2 zeigt das Schema eines OWL-QL Dialogs. Die Kommunikation beginnt mit der Anfrage des Clients an den Server (Query, siehe 3.3.2). Der Server liest die Informationen aus und sendet eine Antwort (AnswerBundle, siehe 3.3.3). Zusammen mit dem AnswerBundle wird ein ContinuationToken gesendet, das ein Prozess-Handle (analog: Session-ID) enthält, mit dem der Client die Kommunikation fortsetzen kann.

Der Dialog ist beendet, wenn eine der Seiten ein TerminationToken sendet. Im Falle des Clients bedeutet dies, dass momentan keine weiteren Antworten benötigt werden, im Falle des Servers, dass für die Frage keine weiteren Antworten vorliegen.

3.3.2 Die Syntax des Frage-Dokuments

OWL-QL definiert ein Query Dokument für das Senden von Anfragen an den Reasoner. Dieses Dokument besteht aus den Elementen Premise, QueryPattern, ServerContinuation, ServerTermination, AnswerPattern, MustBindVars, MayBindVars, DontBindVars, AnswerKBRef und AnswerSizeBound. Die einzelnen Teile werden in den nächsten Unterabschnitten erläutert. Als durchgehendes Beispiel dient eine exemplarische Query (B.1.3)

3.3.2.1 Premise

Die Premise (Prämisse) ermöglicht, es Fragen unter einer oder mehreren Annahmen zu stellen. Die umgangsprachliche Formulierung wäre. „Unter der Voraussetzung, dass diese Annahme zutrifft, wie wäre die Antwort auf folgende Frage?“.

Für die Query bedeutet dies, dass vor Evaluierung der eigentlichen Frage die angenommenen Aussagen in die Wissensbasis eingespielt und dann bei der Beantwortung berücksichtigt werden.

```
<premise>
    <rdf:RDF>
        <rdf:Description
            rdf:about="file://family-1#kathrin">
            <rdf:type rdf:resource="file://family-1#woman" />
        </rdf:Description>
        <rdf:Description
            rdf:about="file://family-1#kathrin">
            <fam:has-child rdf:resource="file://family-1#
               petra" />
```

```
        </rdf:Description>
    </rdf:RDF>
</premise>
```

Quellcode 3.1: OWL-QL Premise Example

Das Beispiel 3.1 zeigt eine Prämisse in XML-Syntax. Prämissen sind OWL Sätze, die in RDF Syntax angegeben werden.

Die im Beispiel formulierte Prämisse beinhaltet zwei solcher Sätze. Im ersten wird definiert, dass das benannte Objekt „kathrin“ eines Instanz des Konzepts „woman“ ist (`instance kathrin woman`), und das „kathrin“ über die Rollenbeziehung „has-child“ mit dem Objekt „petra“ in Verbindung steht (`related kathrin petra has-child`). Die Aussage dieser Prämisse lautet also: „Kathrin ist eine Frau und hat ein Kind namens Petra“[12].

3.3.2.2 QueryPattern

Bei der ersten Anfrage im Rahmen des Dialogs wird das QueryPattern übergeben. Dieses Pattern legt die Suchbegriffe fest. Die Syntax ist wie zuvor die einer OWL Wissenbasis, dass heisst eine Reihe von Statements in Form von RDF Triples. Im Gegensatz zur OWL Wissenbasis werden die Teile der Aussage, nach denen gesucht werden soll durch Variablen-Bezeichner ersetzt. Beispiel 3.2 zeigt den Aufbau des QueryPatterns.

```
<queryPattern>
    <rdf:RDF>
        <rdf:Description
            rdf:about="http://www.w3.org/2003/10/owl-ql-
               variables#x">
            <rdf:type rdf:resource="file://family-1#woman" />
        </rdf:Description>
        <rdf:Description
            rdf:about="http://www.w3.org/2003/10/owl-ql-
               variables#x">
            <fam:has-child rdf:resource="http://www.w3.org
               /2003/10/owl-ql-variables#y" />
        </rdf:Description>
    </rdf:RDF>
</queryPattern>
```

Quellcode 3.2: QueryPattern Example

[12] Anmerkung: Die implizite Annahme, die sich beim Lesen aufdrängt ist: Petra ist die Tochter. Diese Folgerung kann eine Maschine jedoch ohne weitere, eventuell in der Ziel-KB bereits getroffene, Aussagen nicht schliessen.

Der Inhalt dieses Beispiels soll ebenfalls in verständlicherer Form forumliert werden. Das QueryPattern enthält zwei RDF-Statements. In beiden werden Variablen verwendet. Die Variablen haben den Namespace `http://owl/variables#` und werden im Schema OWL-QL Variables (B.1.2) definiert.

Die beiden Sätze lauten in nRQL-Syntax: (`?x woman`) und (`?x ?y has-child`). Beide Sätze werden durch ein logisches „und“ verknüpft. Die entstehende Frage sucht Objekte der ABox, die in einer „hat-Kind“-Beziehung stehen, und von denen das erste eine Frau[13] ist. Vereinfacht ausgedrückt sollen also Mutter-Kind-Paare unter den Individueen der Wissensbasis gesucht werden.

Zu beachten ist, dass entgegen der Darstellung im verwendeten Beispiel eine Variable für jeden Ausdruck eines RDF Triples verwendet werden kann. So ist die Suche über Instanzen ebenso möglich wie über Rollen oder Konzepte. Variablen werden bei OWL-QL in Must-, May- und Don't-Bind Variablen unterschieden, mehr dazu unter Bindung von Variablen (3.3.2.6).

3.3.2.3 ServerContinuation

Alternativ zum QueryPattern kann der Client das ServerContinuation-Element verwenden. Das Schema erlaubt in diesem Fall nur einen einzigen String als Parameter, das ProcessHandle. Dieses repräsentiert einen offenen Dialog auf der Seite des Servers und ermöglicht diesem die Weiterführung auch ohne explizite Angabe einer Frage. Ein Beispiel für die Wiederaufnahme eines Dialogs unter Verwendung des ProcessHandles ist in 3.3 angegeben.

```
<serverContinuation>
  <processHandle>20228056#23965177</processHandle>
</serverContinuation>
```

Quellcode 3.3: ServerContinuation mit ProcessHandle

3.3.2.4 ServerTermination

Statt zur Fortsetzung des Dialogs kann das oben beschriebene ProcessHandle auch zum clientseitigen Beenden der Kommunikation verwendet werden. Im Kontext der ServerTermination kann der Server den offenen Dialog ermitteln und beenden.

[13] Genauer: Eine Instanz des Konzepts „woman“

3.3.2.5 AnswerPattern

Das AnswerPattern ermöglicht dem Client die Ausgabeform der Antworten zu definieren. Es handelt sich um eine Option, denn wird kein AnswerPattern angegeben, kann alternativ auch das QueryPattern verwendet werden.
Der Zusammenhang zwischen der Antwort und dem mitgegebenen Pattern wird im Unterabschnitt 3.3.3.2 erläutert.

3.3.2.6 VariableBindings

Ziel einer OWL-QL Anfrage ist die Bindung von Elementen an Variablen. Für die Definition dieser Variablen stehen drei Typen zur Verfügung: *Must-Bind*, *Don't-Bind* und *May-Bind* Variablen. Der Typ und der Name der zu bindenden Variablen wird im Query-Dokument wie aus Beispiel 3.4 ersichtlich übergeben.

```
34 <mustBindVars>
       <var:x/>
       <var:y/>
   </mustBindVars>
```

Quellcode 3.4: Definition von MustBind-Variablen

Die Definition der May- und Don't-Bind Variablen erfolgt analog. Die Bedeutung der möglichen Bindungstypen soll im Folgenden erläutert werden. Zur Verdeutlichung wird das von Fikes, Hayes und Horrocks in [4] verwendete Beispiel `(hasFather ?p ?f)` herangezogen.

MustBindVariables MustBind-Variablen erfordern zwingend eine Bindung an ein Antwort-Element. Das bedeutet für das Ergebnis, dass nur eine Antwort seitens des Servers erfolgt, wenn mindestens ein Element gefunden wurde, das die Bedingungen im QueryPattern erfüllt. Das OWL-QL-Syntax Schema legt fest, dass jede Frage mindestens eine Variable vom Typ MustBind enthalten muss.[14]

Es wird ausgegangen von der Fragestellung: `(hasFather ?p ?f)`. Dieser Term bezeichnet das Verhältnis Vater-Sohn zwischen zwei Personen P und F. In der zur KB gehörenden TBox ist definiert, dass jede Person einen Vater hat, unabhängig ob dieser als benannte Instanz in der ABox vorhanden ist. Die Person P (das Kind) ist dabei eine zwingend gebundene Variable, dass heisst es wird maximal so viele Antworten geben wie Personen-Instanzen in der Wissensbasis vorkommen.

[14] siehe 6.3 und 3.4

Ist nun die Variable `?f` ebenfalls eine zwingend gebundene Variable, so wird die Antwortmenge die Anzahl der bekannten Väter umfassen, da jede Person genau einen Vater besitzt und keine Aussagen getroffen werden, wenn nicht für `?p` und `?f` Instanzen gefunden werden, die in der Beziehung `?f` ist Vater von `?p` stehen. In der Antwort werden sowohl `?p` als auch `?f` an Instanzen gebunden.

MayBindVariables MayBind-Variablen werden schwach gebunden. Das bedeutet, dass die Bindung einer Instanz aus der Wissensbasis an die Variable erfolgt, wenn eine Antwort gefunden werden kann, diese Bindung für das Ausliefern der Antwort aber keine Bedingung darstellt.
Bezogen auf das oben genannte Beispiel bedeutet das, das die Antwortmenge die Menge aller Personen-Instanzen aus der KB umfasst. Dort wo der Vater einer Person (siehe 3.3.2.6) identifiziert werden kann, wird die Bindung von `?f` vollzogen, in den anderen Fällen nicht. Die zurückgegebene Menge von Antworten besteht folglich aus einer inhomogenen Menge von komplett gebundenen Tupeln (`?p` und `?f` identifiziert) und nur teilweise gebundenen Tupeln (`?p` identifiziert, `?f` bleibt Variable).

DontBindVariables Don't-Bind-Variablen werden bei der Beantwortung der Frage überhaupt nicht gebunden. Für das Beispiel bedeutet der Einsatz von `?f` als Don't-Bind Variable, dass die Antwort die Menge aller bekannten Personen-Instanzen der KB umfasst (da von jeder Person bekannt ist, dass sie einen Vater hat, s.o.), alle jedoch ungebunden als Tupel `(?f Instanz )` zurückgegeben werden.

Ein detailliertes Beispiel für die Verwendung von Don't-Bind- und Must-Bind-Variablen in nRQL-Syntax folgt in Abschnitt 4.2.2.

3.3.2.7 AnswerKBRef

Eine Query enthält eine Referenz auf die zu verwendene Wissensbasis (siehe Beispiel 3.5). Dieses Element kann je nach Belegung verschiedene Bedeutungen für den Server haben.

```
<answerKBPattern>
    <kbRef rdf:resource="file://family-1#"/>
</answerKBPattern>
```

Quellcode 3.5: Definition einer AnswerKBReferenz

OWL Knowledge-Base Es ist möglich im AnswerKBRef-Element eine OWL-QL Knowledge-Base in Form von RDF-Statements zu übergeben. In diesem Fall wird genau diese KB bei der Beantwortung der Frage verwendet.

Referenz (URL) Wird eine URL übergeben, so wird die entsprechende OWL-KB, die über die Adresse erreicht wird, zur Beantwortung herangezogen.

Variable Wird statt einer KB eine Variable aus dem Namensraum *owl-ql-variables* übergeben, wählt der Server die zu verwendene KB aus. In der Antwort wird die Variable substituiert. Dadurch erhält der Client die Information, welche Wissensbasis verwendet wurde.

3.3.2.8 AnswerSizeBound

Der Vorteil des Dialog-Modells von OWL-QL besteht in der Möglichkeit, iterative Anfragen zu stellen und sich jeweils nur eine begrenzte Menge von Antworten zurückliefern zu lassen (Codebeispiel 3.6).

```
<answerSizeBound>
5
</answerSizeBound>
```

Quellcode 3.6: Festlegung der Anzahl gewünschter Antworten

Die Information, wieviele Antworten mit dem nächsten AnswerBundle geschickt werden sollen, wird durch einen positiven Integer-Wert definiert.

3.3.3 Die Syntax des Antwort-Dokuments

Die Antwort des Reasoners wird ebenfalls als XML Nachricht dargestellt. Der Aufbau dieses AnswerBundle-Dokuments wird im OWL-QL Syntax Schema (B.1.1) beschrieben.

Die Elemente des AnswerBundles sind das QueryPattern, die Antworten in Form eines Arrays von Antwort-Elementen und Informationen über den Zustand des Dialogs (ServerContinuation).

3.3.3.1 QueryPattern

Das QueryPattern, das dem AnswerBundle mitgegeben wird, ist inhaltlich eine Kopie des QueryPatterns der Frage (siehe 3.2). Die redundante Wiederholung dieses Elements ist optional, ermöglicht jedoch, z.B. bei der Speicherung des AnswerBundle-Dokuments später nachzuvollziehen, auf welche Fragestellung sich die Antwort bezog.

3.3.3.2 AnswerArray

Das AnswerArray enthält die Antworten, die der Reasoner anhand der Query ermittelt hat. Eine Antwort besteht aus einem Binding-Set und optional einem Answer-Pattern.

Binding-Set Das Binding-Set enthält die Bindung der MustBind Variablen an die Antwort-Terme aus der Wissensbasis. Ein Beispiel ist im Code-Ausschnitt 3.7 zu dargestellt.

```
<binding-set>
    <var:x rdf:resource="file://family-1#betty"/>
    <var:y rdf:resource="file://family-1#eve"/>
</binding-set>
```

Quellcode 3.7: BindingSet Example

AnswerPatternInstance Optional kann ein AnswerPattern mitgegeben werden. Für das AnswerPattern werden in dem in der Fragestellung mitgegebenen OWL-Term die Variablen mit den gefundenen Bindungen substituiert. Wird in der Query nicht explizit ein AnswerPattern angegeben, so kann für die Variablen-Substitution auch das Query-Pattern verwendet werden.

```
<answerPatternInstance>
    <rdf:RDF>
        <rdf:Description rdf:about="file://family-1#betty
            ">
            <rdf:type rdf:resource="file://family-1#woman
                "/>
            <fam:has-child rdf:resource="file://family-1#
                eve"/>
        </rdf:Description>
    </rdf:RDF>
</answerPatternInstance>
```

Quellcode 3.8: AnswerPatternInstance Example

Die so generierte Antwort stellt eine vollwertige OWL-KB dar und kann vom Client verwendet werden, um Antworten in eigene Bestände zu importieren oder weiterzuverarbeiten.

3.3.3.3 ServerContinuation

Zur Unterstützung des Dialog-Prozesses verfügt auch der Server in seinem Answer-Bundle über ein Element, mit dem der Zustand des Dialogs beschrieben werden kann.

Im Gegensatz zum Query-Dokument werden im AnswerBundle jedoch Fortführung und Beendigung des Dialogs unter dem Element ServerContinuation zusammengefasst.

```
<continuation>
    <continuation-token>
        <processHandle>20228056#23965177</processHandle>
        <answerBundleSize>2</answerBundleSize>
    </continuation-token>
</continuation>
```

Quellcode 3.9: AnswerBundle ServerContinuation

Das Code-Beispiel 3.9 zeigt die Übergabe eines serverseitig generierten Process-Handles an den Client. Zusätzlich zum Process-Handle sendet der Server die Anzahl der im Dokument gelisteten Antwort-Elemente als Kontrollwert mit.

Alternativ zu diesen Elementen kann der Server den Dialog beenden, indem er ein Termination-Token sendet. Durch die Verwendung dieses Symbols zeigt er dem Client an, dass er keine weiteren Antworten für die laufende iterative Anfrage bereitstellen kann. Die Terminierung des Dialogs kann auch bereits in der ersten Antwort des Servers erfolgen, für den Fall, dass er überhaupt keine gültigen Antworten ermitteln konnte.

Das OWL-QL-Syntax Schema definiert drei gültige Termination Token: end, none und rejected[15].

End Die Verwendung von „End“ zeigt dem Client an, dass keine weiteren Ergebnisse auf die Frage geliefert werden, weil alle gefundenen Antworten bereits gesendet worden sind.

None Bewirkt im Prinzip das gleiche wie „End“, jedoch mit dem Unterschied, dass der Server eine Aussage darüber trifft, dass er für die Frage überhaupt keine Ergebnisse finden konnte.

Rejected Wenn der Server eine Frage aus formalen Gründen nicht beantworten kann, z.B. weil Sprachelemente in der XML-Nachricht verwendet wurden, die nicht unterstützt werden können, kann er die Frage unter Verwendung von „Rejected“ zurückweisen.

3.3.4 Zusammenfassung

In den vorangegangen Abschnitten erfolgte eine detaillierte Darstellung der einzelnen Elemente von Query und AnswerBundle. Abschließend soll als Zusammenfassung der

[15] Das ist der neueste Stand, dass verwendete Schema kannte noch kein „rejected“

Ausführungen eine kurze tabellarische Übersicht geliefert werden, im den Aufbau leichter verständlich zu machen.

Bei den Tabellen beziehen sich die Bezeichner in der Spalte „Use“ auf die Entität der Elemente, die Syntax entspricht der Regular Expression Syntax, eine „1“ steht für ein obligatorisches Element[16], ein „?“ für ein optionales Element[17] und ein „*“ für ein mengenmäßig unbestimmtes Element[18].

Query-Document Ein Query-Document wird durch das Wurzel-Tag *query* erzeugt. Innerhalb dieses umschließenden Elements sind die in Tabelle 3.1 aufgelisteten Unter-Elemente zugelassen bzw. gefordert.

Element	Parent	Use	Beschreibung
Query	Root	1	Hauptelement
Premise	Query	?	Aussagen (Axiome)
QueryPattern	Query	? (1)	OWL-KB und Variablen
ServerContinuation	Query	? (1)	mit ProcessHandle
ServerTermination	Query	? (1)	mit ProcessHandle
AnswerPattern	Query	?	OWL KB
MustBindVars	Query	1	mit mind. einer gebundenen Variable
MayBindVars	Query	?	Liste der May-Bind Variablen
DontBindVars	Query	?	Liste der Don’t-Bind Variablen
AnswerKBPattern	Query	1	zu verwendene KB
AnswerSizeBound	Query	?	gewünschte Anzahl von Antworten

Tabelle 3.1: Aufbau Query-Document

Zu beachten ist, dass die Elemente QueryPattern, ServerContinuation und ServerTermniation einander ausschließen. Das Schema legt fest, dass genau eins dieser Elemente vorhanden sein muss.

AnswerBundle Das AnswerBundle-Element umfasst die in Tabelle 3.2 gelisteten Elemente.

Die Beschreibung des Aufbaus der OWL-QL Dokumente ist leider in verschiedenen Quellen nicht konsistent. Außerdem ergeben sich aus dem Schema und dem Protokoll einige offene Fragen. Eine Erörterung der Probleme erfolgt in der Zusammenfassung (Abschnitt 6.3).

[16] genau ein Element
[17] kein oder ein Element
[18] kein, ein oder mehrere Elemente

Element	Parent	Use	Beschreibung
AnswerBundle	Root	1	Hauptelement
QueryPattern	AnswerBundle	?	Wiederholung aus Query
Answer	AnswerBundle	*	Array von Antworten
Binding-Set	Answer	1	Bindungen von Must-Bind Var.
AnswerPatternInstance	Answer	1	Frage mit ersetzten Variablen
ServerContinuation	AnswerBundle	?	Dialog
ProcessHandle	ServerContinuation	?	ID für die Fortsetz. des Dialogs
Termination Token	ServerContinuation	?	Beenden des Dialogs
AnswerSizeBound	AnswerBundle	?	Anz. der Antworten im Dok.

Tabelle 3.2: Aufbau AnswerBundle-Document

3.4 Vergleich von OWL-QL und nRQL

Da eine native Unterstützung der OWL-QL Sprachelemente durch den RACER Server weder geplant, noch beabsichtigt ist, muss ein Übersetzer entwickelt werden, der Fragen in OWL-QL Syntax in nRQL Befehle umwandelt und die ermittelten Antworten wieder in OWL-QL Sytanx ausgibt.
Die Aufgabe dieses Abschnitts der Analyse ist es, die Sprachen hinsichtlich ihrer Funktion und Syntax zu untersuchen und eine Schnittmenge zu definieren.

Dabei hat die Sprache nRQL die Sonderstellung, dass sie speziell für den RACER-Server entwickelt wurde. Sie unterstützt nicht alle Funktionen, die der Server-Kern bereitstellen würde, bietet jedoch mit ihren angebotenen Befehlen eine umfassende API.

Damit unterscheidet sie sich hinsichtlich ihrer Entwicklungsgeschichte und ihrem Einsatzzweck von OWL-QL. nRQl unterstützt Befehle zum verändernden Zugriff auf die Knowledge-Base, während OWL-QL als reine Anfragesprache für OWL-basierte Wissensbasen entwickelt wurde. Die erste Einschränkung, die sich ergibt, ist also die Konzentration auf Queries.

Die komplexe Syntax von nRQL-Queries erlaubt darüber hinaus innerhalb einer Query Disjunktionen (or) und Negationen (not) sowie das Bilden der Inverse eines Ausdrucks. Dies lässt sich mit OWL-QL nicht abbilden. Es erfolgt eine Beschränkung auf konjunktive Anfragen [19].

Gemäß der Schema-Definition von OWL-QL (siehe 3.3) darf das Must-Bind-Element nicht leer sein. Dies hätte zur Folge, dass boolesche Anfragen wie sie nRQl zum Beispiel mit `retrieve () (?x woman)` unterstützt, nicht realisierbar wären. Die neueren Publikationen zu OWL-QL widersprechen in diesem Punkt jedoch der Schema-Definition, so dass davon ausgegangen werden kann, dass diese Einschränkung keinen Bestand hat.

OWL-QL bietet aber auch eine Reihe von Features, die nRQL nicht direkt abbilden kann. Dies sind zum Beispiel die definierten Dialoge zwischen Client und Server, die iterative Abfragen ermöglichen. Das Konzept der Prämissen ist in nRQL ebenfalls nicht implementiert.
OWL-QL unterscheidet außerdem drei Typen von Variablen-Bindungen, während nRQL nur Must-Bind-Variablen unterstützt.

Eine weitere Einschränkung bilden die möglichen Frage-Typen in nRQL. Eine OWL-QL Query ist grundsätzlich eine OWL Wissensbasis, bestehend aus RDF-Statements, bei denen Subjekt, Prädikat oder Objekt durch Variablen ersetzt werden. Dieses Konstrukt erlaubt Queries, die beispielsweise nach der Beziehung zwischen zwei benannten Instanzen fragen. Eine solche Frage kann in nRQL nicht ausgedrückt werden. Die Bindung von Variablen erfolgt stets an ein eine Instanz eines Konzeptes, nicht jedoch an eine Rolle.

Referenzierung von Knowledge Bases Die Syntax von OWL-QL erfordert die Übergabe einer Referenz auf eine oder mehrere Wissensbasen, die zur Beantwortung der Query herangezogen werden sollen. Diese Referenz kann jedoch ebenfalls durch eine Variable ersetzt werden und erlaubt so dem Server die Auswahl einer geeigneten KB zur Beantwortung der Frage. Die Variable wird im Antwort-Dokument ersetzt und liefert dem Client die Information, welche KB verwendet wurde. Diese Funktion ist in nRQL nicht möglich. Ein RACER Server kann zwar mehrere KBs verwalten, die Auswahl, welche KB verwendet werden soll, erfolgt jedoch explizit in der Query.

Behandlung der UNA Die Unique-Name-Assumption, die in RACER vorgenommen wird, erlaubt keine doppelte Verwendung eines Individuums in einer Variablen-Bindung. Dies führt dazu, dass bei der Annahme einer AnswerKB A, die nur eine ABox-Instanz enthält, eine leere Menge als Antwort auf die folgende Frage zurückgegeben wird.

```
(instance p1 person)
(retrieve (?x) (and (?x person) (?y person)))
> NIL
```

RACER erlaubt keine Bindung von p1 an ?x und ?y im Body. OWL-QL erwartet jedoch eine Antwort „p1“.

Die OWL-QL Antwort bildet eine KB $B(Q)$, bestehend aus dem QueryPattern Q (Statements des Bodys) und dem BindingSet B (Bindung von ?x an p1) [4]. Für eine gültige Ergbnismenge ist gefordert, dass $B(Q)$ in A enthalten ist. Nicht gebundene Variablen (?y) werden dabei als „existential blanks“ behandelt, was bedeutet, dass etwas in der KB existiert, was nicht näher benannt wurde.

$B(Q)$ hätte dann die folgende Gestalt:

```
p1 rdf:type person
!y rdf:type person
```

Die Forderung wäre erfüllt, denn `(?x p1)` ist eine gültige Antwort auf die Fragestellung. Um dieses Ergebnis mit nRQL zu erreichen, muss die Anwendung der UNA unterbunden werden. Dies geschieht durch Voranstellen des „$“-Zeichens vor eine Variable.

Die nRQL-Query

```
(retrieve ($?x) (and ($?x person) ($?x person)))
> (($?x p1))
```

liefert das gewünschte Ergebnis.

3.4.1 Zusammenfassung

Zum Abschluss des vorherigen Unterkapitels kann die Schnittmenge zwischen den Sprachen wie folgt zusammen gefasst werden.

1. Beschränkung auf ABox-Queries
2. Arbeiten mit einer vom Client benannten KB
3. Bindung nur an Instanzen von Konzepten, keine Fragen nach Beziehungen und Rollen
4. Beschränkung auf Must-Bind-Variablen
5. Beschränkung auf konjunktive Verknüpfung von Query-Body-Elementen

Zusätzlich müssen einige von der OWL-QL-Spezifikation geforderten Eigenschaften für die Verwendung mit nRQL realisiert werden. Dabei handelt es sich um Mechanismen, die in nRQL oder RACER nicht als atomare Funktionen bereitstehen, sich jedoch als Erweiterung modellieren lassen. Dies erfolgt in Abschnitt 4.2.

3.5 Resultierende Aufgabenstellung

Ausgehend von der in diesem Kapitel erfolgten Analyse des bestehenden Systems und der Untersuchung der Anforderungen für das zu entwickelnde System, ergibt sich die folgende Definition der Aufgabenstellung für diese Arbeit.

Es soll ein System entwickelt werden, dass den Zugriff auf einen RACER-Server über die Query-Language OWL-QL erlaubt. Dabei wird die in 3.4 herausgearbeitete Schnittmenge von OWL-QL und nRQl umgesetzt und eine Unterstützung von iterativen Abfragen (Dialoge) implementiert werden.

Als Einschränkung für den Funktionsumfang des Prototypen werden nur ABox-Queries berücksichtigt, die OWL-QL Anfragen müssen also konjunktive `retrieve` Queries[19] darstellen. Fragen nach Konzepten der TBox bleiben unberücksichtigt.

Die Schnittstelle für die Kommunikation soll als Web Service realisiert werden, der ein OWL-QL Query-Dokument entgegen nimmt und mit einem AnswerBundle-Dokument antwortet.

Zusätzlich wird im Rahmen dieser Arbeit noch ein einfacher nRQL-Web Service realisiert, der alternativ zur Kommunikation über TCP die Kommunikation über SOAP ermöglicht.

[19] bzw. `retrieve-under-premise`, siehe 4.2.1

Kapitel 4

Entwurf

Ausgehend von der Analyse der bestehenden Strukturen und der daraus abgeleiteten Anforderungen an die zu erstellende Lösung, wird in diesem Kapitel der Entwurf beschrieben.

Wie bei größeren Software-Projekten üblich, haben sich auch während der Entstehung dieser Arbeit die Anforderungen und Rahmenbedingung verändert und mussten an neue Erkenntnisse und Abläufe angepasst werden.

Der Entwurf gliedert sich in mehrere Teile. An seinem Ende steht die Architektur des zu entwickelnden Gesamtsystems. Auf dem Weg zu diesem Ziel müssen einige Design-Entscheidungen bezüglich der einzusetzenden Komponenten und Frameworks getroffen werden und die in Kapitel 3 Eigenschaften von OWL-QL und nRQL modelliert werden.

Die Realisierung des Systems wurde auf mehrere Teilprojekte aufgeteilt, der Entwurf muss also auch die Modularisierbarkeit der Architektur berücksichtigen, um die Implementierungsbereiche klar trennen zu können.

4.1 Auswahl eines Web Service-Frameworks

Die resultierende Lösung soll über einen Web Service (siehe 2.3) erreichbar sein. Implementierungen für Web Services, bzw. das zugrunde liegende Nachrichtenprotokoll SOAP, existieren für alle gängigen Sprachen und Plattformen. Da die Implementierung des SOAP Protokolls nicht Bestandteil dieser Arbeit sein soll und sein kann, muss ein Framework ausgewählt werden, das den Service bereitstellt.

Die Wahl dieses Frameworks stellt eine Designentscheidung dar, da abhängig von ihr eine grundlegende Weiche für die Systemarchitektur gestellt wird.

Zu unterscheiden sind grundsätzlich zwei Möglichkeiten:

1. Stand-Alone Implementierung des Web-Service
2. Web Service eingebettet in Anwendungs-Server bzw. Servlet-Container

Die beiden Ansätze, mögliche Umsetzungen und die jeweiligen Vor- und Nachteile beider Lösungen sollen kurz betrachtet werden.

4.1.1 Stand-Alone Lösung

Vor der Entwicklung der Erweiterungen bestand ein RACER-System aus zwei Stand-Alone Anwendungen: Dem RACER Server und dem RACER Proxy.

Es wäre ein naheliegender Ansatz, auch die Web-Service Anwendung als eigenständiges Programm zu implementieren und eine lose Koppelung zwischen den Komponenten über TCP zu realisieren.

Die Vorteile dieser Lösung beruhen darauf, dass die bestehenden Komponenten unverändert übernommen werden können und bei laufenden Installationen eines RACER Systems nur ein zusätzliches Programm installiert werden muss.

Der Nachteil dieser Art der Implementierung besteht in der losen Koppelung der Java-Komponenten. Durch die Trennung der Komponenten in eigenständige Programme ist die Interaktion nur durch vergleichsweise „teure“ Kommunikation mittels Nachrichten möglich, bzw. durch die aufwändige Verwaltung von Objekten im Speicher des Servers. Ein direkter Zugriff auf Objektebene ist nicht möglich.

Ein weiteres wesentliches, wenngleich nicht ausschlaggebendes Argument gegen die Stand-Alone Lösung ist das Fehlen eines frei verfügbaren Frameworks. Die Entwickler der Java SOAP Referenz Implementierung Apache Axis[1] haben die Entwicklung der Stand-Alone-Version eingestellt.

4.1.2 Eingebettet in Server

Die Alternative zum Einsatz einer Stand-Alone-Software ist die Einbettung in einen bestehenden Server bzw Servlet-Container.
Dies bedeutet bei der Installation und Wartung des Systems zwar serverseitig mehr Aufwand, da ein zusätzlicher Server auf dem Host-Rechner installiert werden muss, bringt jedoch eine Reihe von Vorteilen mit sich:

- Benötigte Funktionen wie Sessionverwaltung, Speichermanagement, Nachrichtentransport usw. werden vom Server bereitgestellt und müssen nicht implementiert werden.

[1] Das Open Source Framework Apache Axis wird in Abschnitt A.1 besprochen

- Durch die Verwendung eines weit verbreiteten Servers entsteht eine gut dokumentierte Basis für spätere Weiterentwicklungen des Systems.

Für den Betrieb eines Web Service-Frameworks innerhalb eines Servlet-Containers, stehen mit Apache Axis Jakarta Tomcat stabile Implementierungen zur Verfügung, was die Bereitstellung von Web Services erheblich vereinfachen kann. Darüber hinaus sind beide Anwendungen kostenlos verfügbar und lassen sich auch in kommerziellen Systemen lizenzfrei nutzen.

Alle zu entwickelnden Komponenten können sich die Laufzeitumgebung des Servlet-Containers teilen. Durch die gemeinsame Objekt- und Speicherverwaltung ist die Interobjekt-Kommunikation zwischen den Modulen möglich, ein Kommunikations-Overhead zwischen Web-Service und Proxy/Manager entfällt.

4.1.2.1 Entscheidung

Aufgrund der vorangegangenen Untersuchungen fiel die Entscheidung zugunsten der zweiten Lösungsvariante aus.

Die Komponenten sollen so entwickelt werden, dass die gesamte Programm-Logik innerhalb des Tomcat Servlet Containes lauffähig ist. Die Kommunikation zwischen den Komponenten erfolgt über Java-APIs und mit Anwendungen außerhalb des Containers über Servlets, Web Services bzw. TCP und HTTP.

4.2 Modellierung von OWL-QL Eigenschaften

Der Vergleich von OWL-QL und nRQL (vgl. 3.4) hat die Schnittmenge der Sprachen eingegrenzt, die sich durch einen Übersetzer direkt verwenden lässt.

Er zeigte jedoch auch Aspekte von OWL-QL auf, die sich zwar prinzipiell in Verbindung mit RACER einsetzen lassen, jedoch nicht, ohne zusätzliche Funktionen im RACER-System zu implementieren.

Die wichtigsten Punkte dabei waren:

- Unterstützung von iterativen Anfragen
- Definition eines Dialogs zwischen Client und Server
- Anfragen unter Verwendung einer Prämisse
- Unterstützung von Don't-Bind Variablen

Die Funktionalität der Unterstützung von iterativen Anfragen im Dialog zwischen Client und Server war Gegenstand begleitender Arbeiten im Fachbereich. Auf die Ergebnisse wird an späterer Stelle eingegangen werden.

Die Frage, wie die übrigen offenen Punkte im zukünftigen RACER-System realisiert werden können, ist Bestandteil dieses Abschnitts.

4.2.1 Unterstützung von Prämissen

Eine Anfrage an einen RACER Server unter Verwendung einer Annahme, wie sie OWL-QL mit dem Element Premise (siehe 3.3.2.1) unterstützt, stellt eine Transaktion dar.

Vor der Bearbeitung der eigentlichen Frage werden Annahmen in Form von Aussagen in die aktuelle ABox des Servers eingespielt. In nRQL Syntax ausgedrückt, bedeutet das, dass die Ausführung von Befehlen wie in folgendem Beispiel dargestellt erfolgt.

```
(instance betty mother)
(related betty doris has-child)
```

Durch diese Befehle wird verändernd auf die ABox zugegriffen. Verändernde Zugriffe (Statements) sind in der Aufgabenstellung eigentlich ausgeschlossen. Der Zwischenschritt, der bei der Prämisse notwendig ist, erfordert also eine gesonderte Behandlung.

Der Entwurf der Transaktion für das Stellen einer Anfrage unter einer Prämisse umfasst die in Abbildung 4.1 dargestellten Schritte.

Bevor sich das System überhaupt mit dem Problem des verändernden Zugriffs beschäftigt, muss geprüft werden, ob die Aussagen der Prämisse bereits explizit oder implizit in der verwendeten KB erfüllt sind. Ist dies der Fall, kann die Anfrage normal ausgeführt werden.

Enthält die Prämisse zusätzliche Aussagen, müssen diese vor Ausführung der Anfrage in die KB eingespielt werden. Damit der Zustand der KB für nachfolgende Queries nicht verändert wird, geschieht diese Änderung auf einer temporären Kopie, die nach dem Zugriff verworfen werden kann.

Die RACER-Engine bearbeitet aufgrund ihrer Struktur nur eine Anfrage zur Zeit. Weitere Anfragen an diesen Server werden vom Proxy in einer Queue gesammelt. Eine zusätzliche Absicherung gegen konkurrierende Zugriffe ist nicht erforderlich, weshalb auf Locking an dieser Stelle verzichtet werden kann.

Nachdem die Änderungen vorgenommen und die Anfrage bearbeitet wurde, kann die Kopie der KB gelöscht werden.

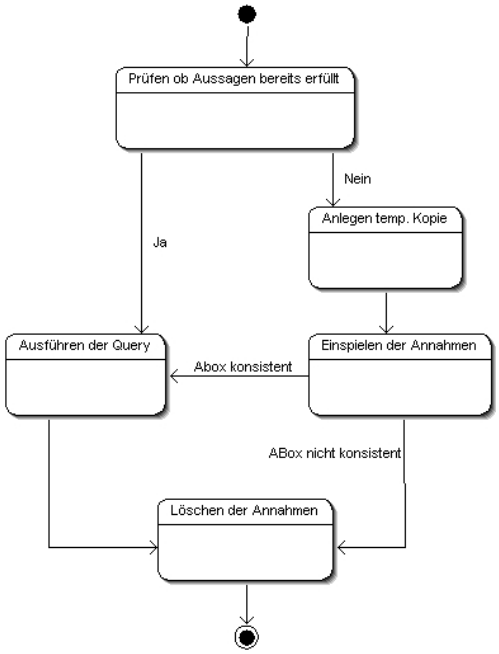

Abbildung 4.1: Ablauf Prämisse

4.2.2 Unterstützung von Don't-Bind-Variablen

Anfragen, in denen Don't-Bind Variablen verwendet werden, können von der RACER-Engine nicht direkt unterstützt werden. Dies liegt an der Open-World-Assumption, die RACER bei der Berechnung von logischen Schlüssen verwendet.

Der Einsatz von Don't-Bind Variablen wurde in 3.3.2.6 bereits vorgestellt, soll an dieser Stelle jedoch durch das folgende Beispiel veranschaulicht werden.

Dazu wird der in allen RACER-Beispielen dieser Arbeit verwendeten „Family"-Knowledge-Base (siehe B.2) eine neue ABox-Instanz hinzugefügt.

```
(instance marta parent)
```

Eine Suche nach allen Eltern in der Wissensbasis liefert nun:

```
(retrieve (?x) (?x parent))
( ((?X MARTA)) ((?X BETTY)) ((?X ALICE)) )
```

Das Konzept „Parent" ist in der TBox der KB als die Rollenbeziehung „hat-Kind" zwischen zwei Personen definiert.

Die Frage `(retrieve (?x) (?x ?y has-child))` findet alle in der KB bekannten Eltern-Kind-Paare und gibt den Elternteil als Antwort aus. Da die Elternrolle der neu

eingefügten Person zwar explizit angegeben wurde, jedoch keine Kind-Instanz in der KB existiert, ist die Ausgabe `(((?X BETTY)) ((?X ALICE))).`

Unter Verwendung einer Don't-Bind Variable[2] könnte die Frage nach allen Elternteilen wie folgt formuliert werden:

```
retrieve (?x) (?x !y has-child))
```

In dieser Query wird die Existenz eines Kindelements „y" nicht gefordert, es wird nach Instanzen von Konzepten gesucht, die in ihrer TBox-Definition eine hat-Kind Beziehung enthalten. Die Antwort müsste aufgrund der vorliegenden Tbox identisch mit der von `(retrieve (?x) (?x parent))` sein.

RACER kann diese Frage jedoch wegen der fehlenden Unterstützung von Don't-Bind-Variablen nicht beantworten.

4.2.2.1 Die Rolling-Up-Technik

Es gibt allerdings einen Lösungsansatz, der für eine bestimmte Gruppe von Fragen mit Don't-Bind-Variablen trotzdem eine Beantwortung zulässt, indem die Query in eine äquivalente, lösbare Form gebracht wird.

Dieses Verfahren wird als „Rolling Up" bezeichnet und wurde von Horrocks und Tessaris [3] vorgestellt. Eine gute Zusammenfassung findet sich in [6].

Nach dieser Methode würde die Beispiel-Query aus dem vorherigen Abschnitt zu `(retrieve (?x) (and (?x (and person (some has-child person))))))` umgeschrieben. Dabei folgt die Umformung der Forderung, dass das unbekannte „!y" mit einer bekannten Person „?x" in einer „hat-Kind"-Beziehung steht. Für „!y" wird in diesem Fall das Top-Level Element „Some" verwendet, das das Super-Konzept in einer RACER-TBox darstellt.

Die Antwort auf diese Frage ist wie bei der Suche nach allen Eltern `(((?X MARTA)) ((?X BETTY)) ((?X ALICE))).`

Die „Rolling-Up"-Technik funktioniert jedoch nur, wenn sich die Frage durch einem azyklischen Graph ausdrücken lässt.

4.3 Architektur-Entwurf

Aus den vorstehenden Betrachtungen ergibt sich die in Abbildung 4.2 dokumentierte Architektur für das neu zu entwickelnde RACER-System.

[2] Don't-Bind Variablen werden im Gegensatz zu Must-Bind-Variablen mit dem Prefix „!" bezeichnet

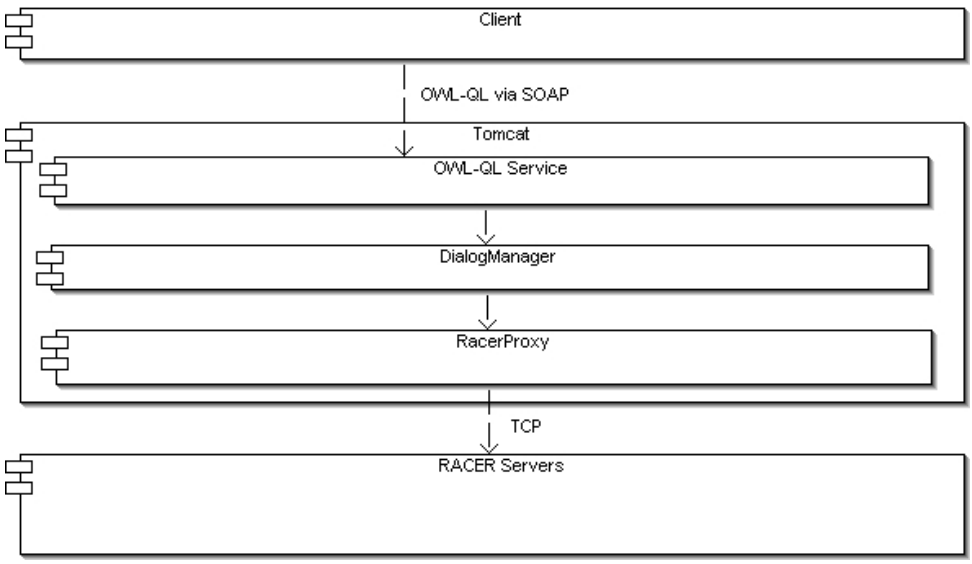

Abbildung 4.2: Architektur Entwurf

Der auffälligste Unterschied zur Ausgangsarchitektur (Abbildung 3.1) ist die Kapselung aller Komponenten im Servlet-Container Tomcat. Durch den Einsatz dieses Servers werden die Erweiterungen gebündelt und lassen sich in einer gemeinsamen Laufzeitumgebung betreiben und administrieren.

Die Architektur ist mehrschichtig aufgebaut. Dabei stellt der OWL-QL-Service die Schnittstelle für die Client-Interaktion zur Verfügung und verbirgt die Implementierung der Prozesse im DialogManager und im Proxy.

Der DialogManager ist zuständig für die Verwaltung des Dialog-Schemas von OWL-QL. Er beinhaltet zusätzlich ein Session-Management zur Bindung von Benutzer-Sitzungen an Dialoge.

Der Proxy organisiert die Kommunikation mit einem oder mehreren RACER Servern, wie in Abschnitt 3.1.1 beschrieben. Seine Aufgaben in der neuen Architetur werden um die Unterstützung iterativer Abfragen erweitert.

Zusätzlich zu den im Diagramm aufgeführten Komponenten wird ein Übersetzer benötigt, der die OWL-QL-Syntax in nRQL-Syntax umwandeln kann.

4.3.1 Definition der Schnittstellen

Mehrschichten-Architekturen zeichnen sich durch eine klare Definition der Kommunikation zwischen den Ebenen aus. Jede Schicht kommuniziert dabei nur mit direkten Nachbarn.

Um dies auch in der vorliegenden Architektur umzusetzen, werden in diesem Abschnitt die Schnittstellen zwischen den Ebenen entworfen.

Client/Web Service Der Client greift über einen Web Service auf das System zu. Das Nachrichtenformat für diese Kommunikation ist SOAP, das in Abschnitt 2.3.2 vorgestellt wurde. Die Anfragen und Antworten innerhalb des Client/Server-Dialogs sind OWL-QL-Dokumente.

Da die OWL-Dokumente bereits alle benötigten Informationen zum Dialog und zur jeweiligen Anfrage beinhalten, benötigt der Web Service nur eine einfache Schnittstelle. Er nimmt ein OWL-QL *Query*-Dokument entgegen und liefert ein OWL-QL *AnswerBundle*-Dokument zurück an den Client.

Web Service/DialogManager Die Kommunikation zwischen Service und Manager kann direkt über ein Interface erfolgen, ein Nachrichten-Transport ist nicht erforderlich. Der Service generiert ein Query-Objekt aus der Client-Anfrage und übergibt es dem DialogManager. Als Antwort erhält er ein Answer-Objekt zurück.

DialogManager/Proxy Der DialogManager sendet die übersetzte Anfrage an den Proxy. Dazu übergibt er einen nRQL-String. Die Antwort erfolgt in Form einer Liste von nRQL-Strings.

Proxy/Server Der Zugriff des Proxys auf einen RACER-Server erfolgt auch in dem neu gestalteten System über eine TCP-Verbindung.

4.3.2 Interaktion der Komponenten

Die in den vorherigen Abschnitten beschriebenen Komponenten und Schnittstellen führen zu dem in Abbildung 4.3 skizzierten Sequenz-Diagramm des Systems.

1. Der Client sendet eine OWL-QL-Query über SOAP an den Service.
2. Der Service deserialisiert die Nachricht und erzeugt ein Request-Objekt, das die in der Query gesendeten Informationen enthält.
3. Das Request-Objekt wird an den DialogManager übergeben.
4. Anhand der Informationen über den Status des OWL-QL-Dialogs, die der Manager aus dem Request gewinnen kann, wird entweder eine bestehende Kommunikation fortgesetzt oder ein neuer Dialog begonnen.
5. Der Dialog fordert beim Request-Objekt eine Übersetzung von OWL-QL-Syntax in nRQL-Syntax an. Die nRQL-Query wird gespeichert und ermöglicht bei folgenden Anfragen die Identifizierung bereits bestehender Dialoge.

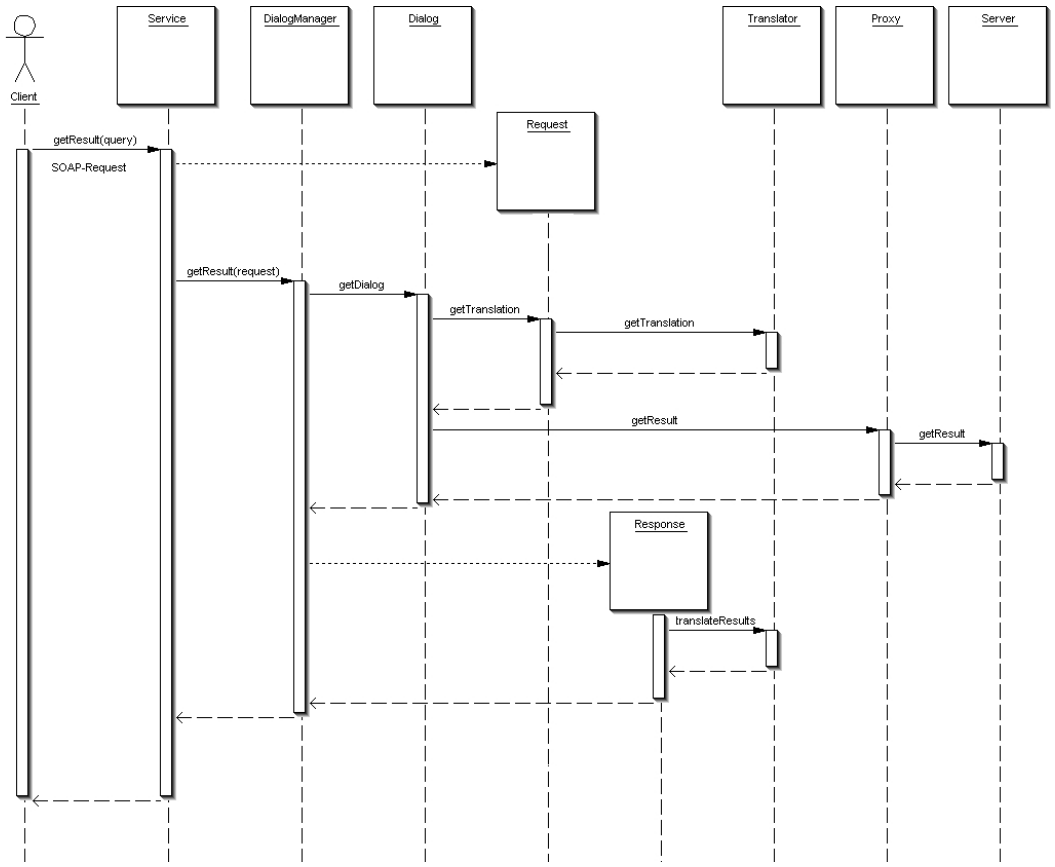

Abbildung 4.3: Sequenz-Diagramm des neuen RACER Systems

6. Die Übersetzung wird vom Translator geliefert und als String an den Dialog zurückgegeben.
7. Die nRQL-Query wird an den Proxy übergeben.
8. Der Proxy sendet die Query über TCP an einen RACER-Server und empfängt die Antwort. Bei mehreren angebundenen Servern und einem bestehenden Dialog wird der entsprechende zuvor verwendete Server identifiziert.
9. Der Manager erzeugt ein Response-Objekt mit der gelieferten Antwort des Servers.
10. Das Response-Objekt übersetzt die Antworten in OWL-QL-Syntax.
11. Der Service generiert ein OWL-QL AnswerBundle-Dokument aus dem Response-Objekt und sendet es über SOAP zurück an den Client.

Die vorgestellte Architektur und der skizzierte Kommunikationsverlauf zwischen ihren Komponenten bilden die Grundlage für die Realisierung des Systems, die im folgenden Kapitel besprochen wird.

Kapitel 5

Realisierung

Dieses Kapitel beschäftigt sich mit den Details der Implementierung. Als Basis dient der Architektur-Entwurf, der in Kapitel 4 ausgearbeitet wurde (vgl. Abbildung 4.2). Wie dort bereits erwähnt, unterteilt sich die Weiterentwicklung der RACER Architektur zur Unterstützung von OWL-QL auf die Teilprojekte: *Server*, *Proxy*, *DialogManager* sowie *Translator* und *Web Service*.

Die Entkopplung der Schichten der Architektur erfolgte in Java durch die Unterteilung in vier Pakete: *Service*, *Syntax*, *Connection* und *Proxy*. Die Kommunikation der Schichten bzw. Pakete erfolgt dabei über Java-Schnittstellen, die im Rahmen des Entwurfs definiert wurden.

Einen ersten Überblick über die Verteilung der Pakete und ihre Abhängigkeiten untereinander liefert Abbildung 5.1[1].

Das Paket *Service* beinhaltet die Web Services zur Kommunikation mit den Clients. Die Umsetzung des OWL-QL Dialogs und die Speicherung status-abhängiger Informationen (Ergebnis-Mengen, Benutzer-Sessions, Dialog-Daten usw.) erfolgt im Paket *Connection*.

Der *Proxy* realisiert die Verbindung mit einem oder mehreren RACER-Servern. Die Übersetzung von OWL-QL Nachrichten in nRQL-Syntax und zurück erfolgt im Paket *Syntax*.

5.1 Vorstellung der verwandten Teilprojekte

Die nicht im Rahmen dieser Arbeit implementierten Teilprojekte *Server*, *Proxy* und *DialogManager* werden in diesem Abschnitt kurz beschrieben.

[1] Der gemeinsame Paket-Namespace für alle Projekte ist *de.tuhh.sts.racer*. Zur besseren Lesbarkeit wird er bei den Diagrammen mitunter ausgelassen

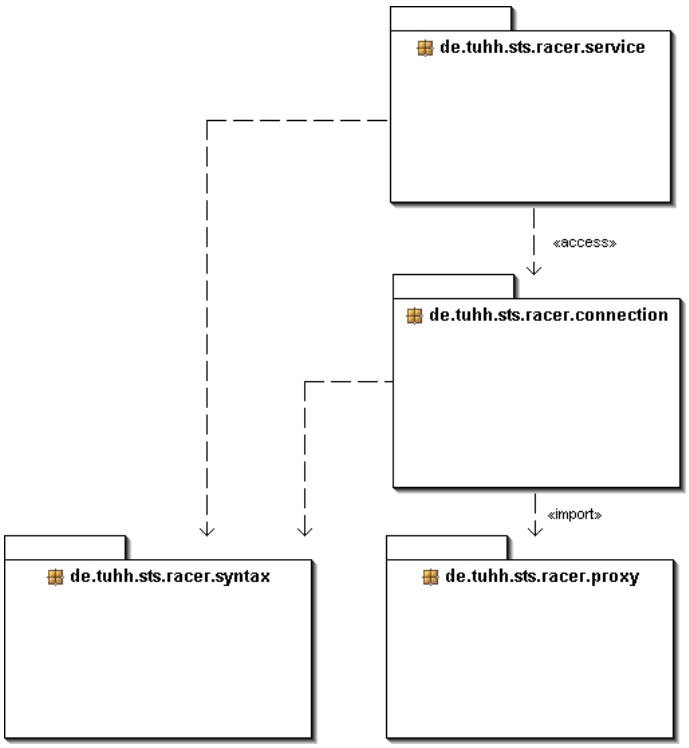

Abbildung 5.1: Überblick über die Paket-Struktur

5.1.1 RACER Server

Zur Unterstützung der im Prototypen angestrebten Teilmenge von OWL-QL (3.4.1) wurden zwei Erweiterungen am RACER-Server vorgenommen.

Iterative Anfragen Mit der Weiterentwicklung des Servers auf die Version 1.8, die bei der Erstellung dieser Arbeit noch nicht öffentlich war, wurde der RACER Server um die Fähigkeit erweitert, iterative Anfragen zuzulassen. Dies bedeutet, dass Ergebnismengen zwischengespeichert werden können und nur bei Bedarf eine zu spezifizierende Menge von Elementen als Antwort-Set zurückgeliefert wird.

Dieser neue Betriebsmodus heißt *TUPLE-AT-A-TIME-Mode*[2].

Nachdem der RACER-Server in diesen Modus versetzt wurde, liefert er nicht mehr automatisch alle Antworten aus, sondern speichert sie in einer Liste. Eine Referenz auf diese Liste wird an den Client (bzw. Proxy) übergeben. Der Client kann dann mit den Befehlen (`GET-NEXT-TUPEL id`) und (`GET-REMAINING-N-TUPELS id num`) Antwortmenge von der Liste anfordern, ohne die Query erneut zu schicken.

[2] Alter Betriebsmodus: *SET-AT-A-TIME-Mode*

Die Implementierung bringt auch ohne die Anbindung der höheren Schichten der Architektur den Vorteil, dass bei großen Wissensbasen, die normalerweise sehr umfangreiche Ergebnismengen erzeugen, der Daten-Transfer zwischen Client und Server auf eine begrenzte Anzahl von Ergebnissen reduziert werden kann.

Unterstützung von Prämissen Zusätzlich wurde die nRQL-Syntax erweitert, um Anfragen unter einer Prämisse (3.3.2.1) zuzulassen. Die Implementierung folgt dabei dem in Abschnitt 4.2.1 entworfenen Ablauf.

Der neue Befehl in nRQL heisst `RETRIEVE-UNDER-PREMISE`. Seine Syntax wird durch das folgende Beispiel dokumentiert.

```
retrieve-under-premise
    (
        (instance C1 Seafood-Course )
        (related C1 W1 drink)
    )
    ( ?x )
    ( W1 ?x has-color)
```

Der Befehl fügt erst die Aussagen „C1 ist ein Fischgericht.“ und „W1 ist ein Getränk für C1.“ in die ABox der Wissensbasis ein und beantwortet im Anschluss die Frage „Welche Farbe hat W1?“.

Die Zusammenfassung der beiden Schritte zu einem nRQL Befehl verbirgt den Prozess vor den höheren Schichten.

5.1.2 RACER Proxy

Der bestehende RACER Proxy wurde von Jog Ratna Maharjan im Rahmen seiner Studienarbeit am Arbeitsbereich STS erweitert, um mit den neuen Betriebsmodi des Servers zusammenarbeiten zu können.

Die Details seiner Implementierung sind in [8] dokumentiert. Einen Überblick über die interne Architektur liefert Abbildung 5.2. Die Integration des Proxys in den Servlet Container Tomcat war nicht Bestandteil seiner Arbeit und wird später in diesem Kapitel gesondert behandelt.

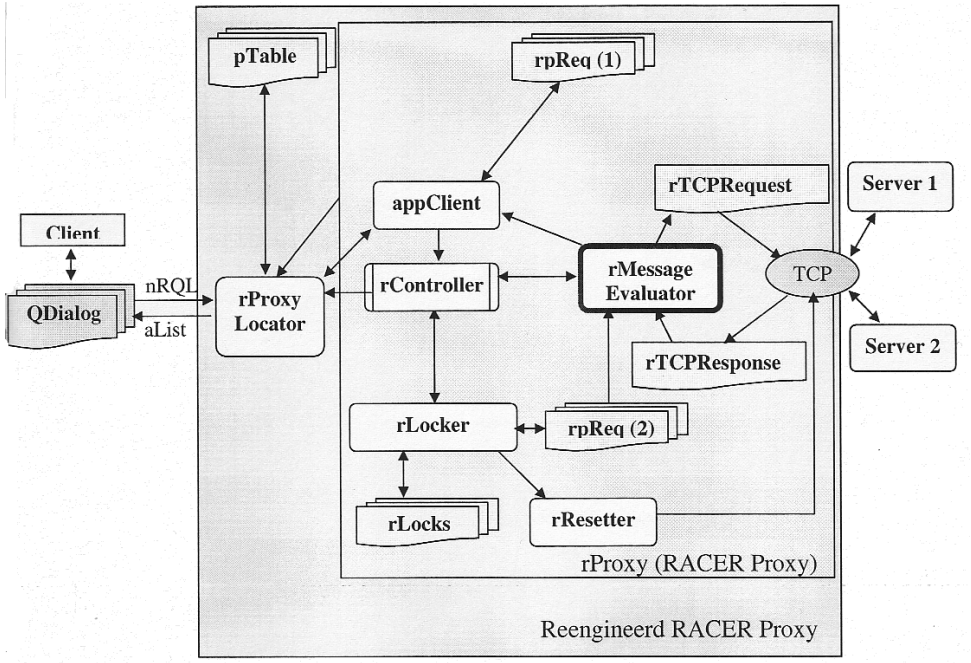

Abbildung 5.2: Architektur des neuen RACER Proxy

5.1.3 DialogManager

Das Paket *Connection* implementiert die Behandlung von OWL-QL-Dialogen. Es beinhaltet einen *DialogManager*, einen *SessionManager* und die Hauptkomponente *RacerOWLManager*, über die die Kommunikation mit dem Service erfolgt. Der genaue Ablauf der Kommunikation wird in Abschnitt 5.3 detailliert beschrieben.

Dieses Paket wurde von Tejas Doshi im Rahmen seiner Studienarbeit am Arbeitsbereich entwickelt. Für die detaillierte Ausführung der Funktionsweise und der Realisierung sei daher auf [7] verwiesen.

Einen Überblick über die Verteilung und Funktion der Klassen im Paket liefert die aus der angeführten Ausarbeitung übernommene Abbildung 5.3.

5.2 Das Paket *Syntax*

Die Aufgabe des Pakets *Syntax* ist die Verarbeitung von nRQL bzw. OWL-QL Nachrichten und die Übersetzung zwischen diesen Formaten. Es unterteilt sich in die Unterpakete *NrqlSyntax* und *OwlQlSyntax* (siehe Abbildung 5.4) und stellt diesen ein gemeinsames Interface zur Verfügung, in dem zentrale Funktionen wie String-Operationen, Fehlerbehandlung und Logging realisiert werden.

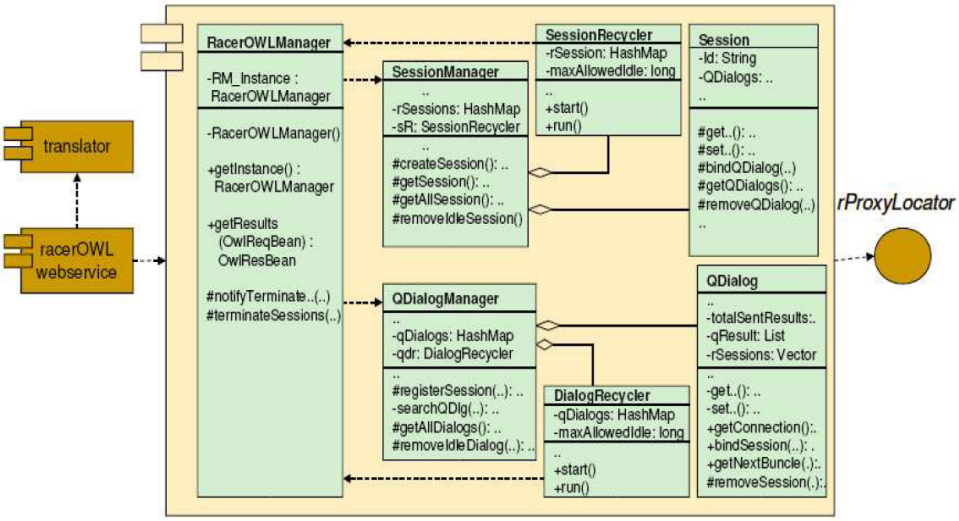

Abbildung 5.3: Architektur des Connection-Pakets

Die Kernklassen dieses Paketverbunds sind die Klassen *Translator*, *OwlQlRequest* und *OwlQlResponse*.

Die Realisierung dieses Pakets erfolgte unter Verwendung von zwei Entwurfsmustern, dem FactoryPattern und dem FacadePattern. Die Methoden der Klassen werden über jeweils ein Interface angesprochen, das eine Factory zur Erzeugung neuer Objekte enthält.

5.2.1 Das Paket *NrqlSyntax*

Die Anfragesprache nRQL arbeitet mit einer String-Syntax und ermöglicht keinen gezielten Zugriff auf einzelne Bestandteile einer Zeichenkette.

Um komfortabel innerhalb von Java auf nRQL Ausdrücke, sowohl bei der Frage als auch bei der Antwort, zugreifen zu können, wurde die nRQL-Syntax in einer Klassenhierarchie nachgebildet, so dass jedes Element einem Typ zugewiesen und objektorientiert bearbeitet werden kann.
Dabei wird Vererbung genutzt, um gemeinsame Eigenschaften in abstrakten oder globalen Klassen zu konzentrieren. Diese Hierarchie der Schnittstellen des Paketes ist in Abbildung 5.5 dargestellt.

Die Grundlagen für den verwendeten Ansatz wurden in der API-Sammlung JRacer2 (vgl. 3.1.2) gelegt, jedoch für diese Arbeit stark erweitert bzw. neu entwickelt.

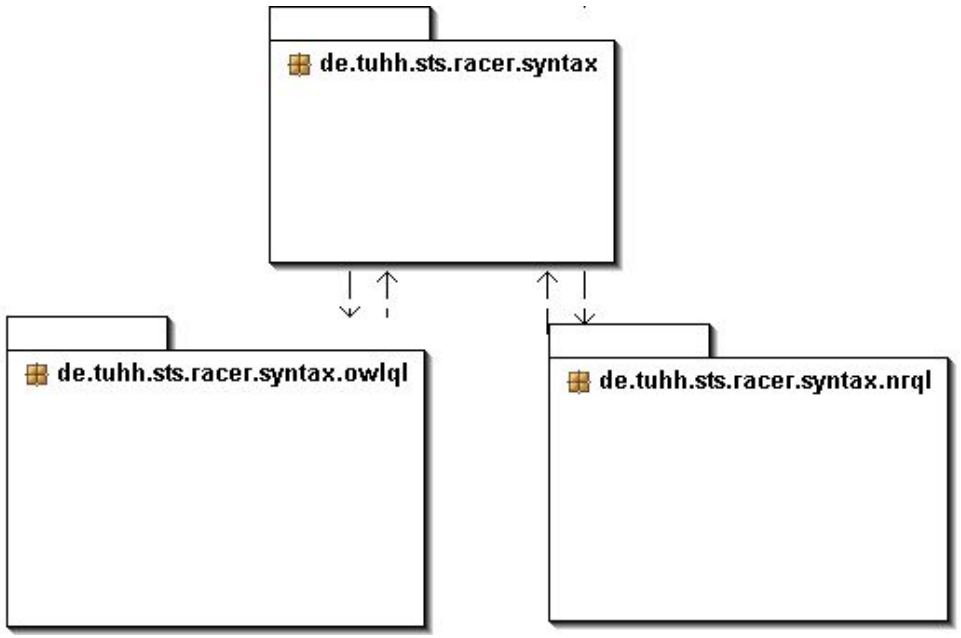

Abbildung 5.4: Paketdiagramm *racer.syntax*

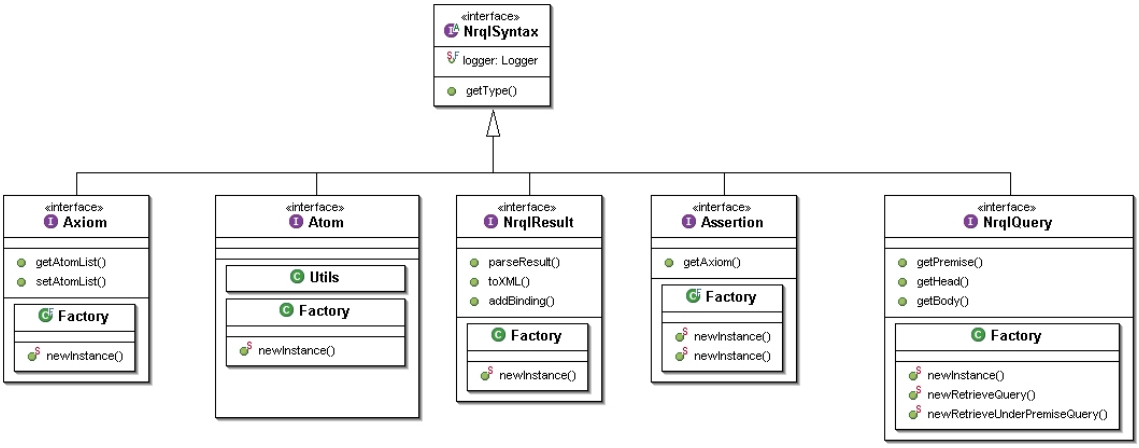

Abbildung 5.5: Interfaces des Pakets *syntax.nrql*

5.2.1.1 Atom

Atome bilden die kleinste Einheit innerhalb der nRQL-Syntax. Ein *Atom* stellt je nach seiner Verwendung im Kontext entweder eine Variable, ein Konzept, eine Rolle oder ein Individuum dar. Abbildung 5.6 zeigt das entsprechende Klassendiagramm.

Der Typ eines *Atom*-Objekts wird beim seiner Erzeugung durch die Factory durch dynamisches Casting festgelegt. Die Verbindung mehrerer Atome zu einem höheren Sprachkonstrukt (siehe *Axiom*, *Assertion*) erfolgt in einer *AtomList*.

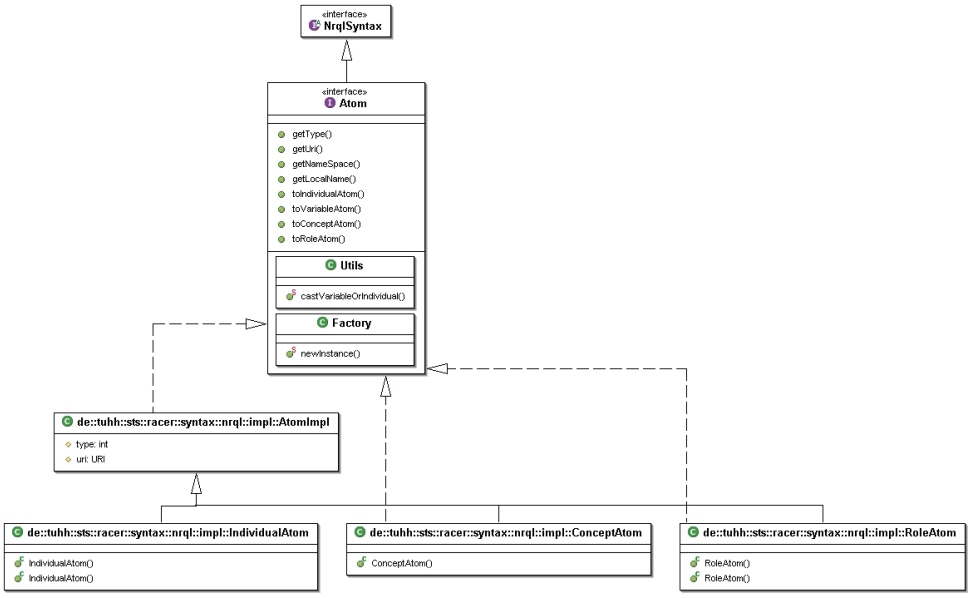

Abbildung 5.6: Klassen-Diagramm Atom

Dies geschieht im Rahmen des Parsens der RDF-Statements in den OWL-QL Nachrichten und wird in 5.2.3 näher erläutert.

5.2.1.2 Axiom

Axiome setzen sich aus mehreren Atomen zusammen und stellen eine Beziehung zwischen diesen her. Dabei wird zwischen den Klassen *ConceptAxiom* und *RoleAxiom* unterschieden.

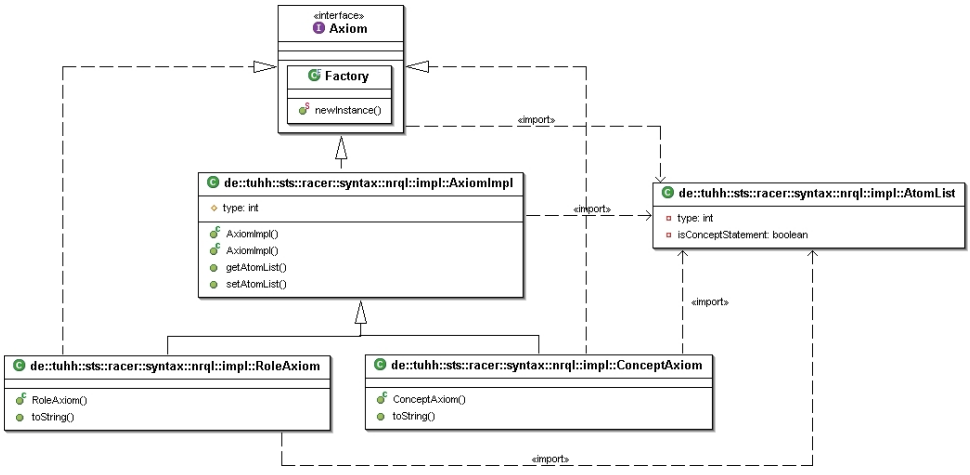

Abbildung 5.7: Klassendiagramm Axiom

Ein *ConceptAxiom* bildet die Beziehung eines Individuums oder einer Variablen zu einem Konzept ab (z.B. `(?x mother)`). *RoleAxiome* dagegen verbinden zwei Individuen (bzw. ein Individuum und eine Variable) durch eine Rollenbeziehung (z.B. `(?x betty has-child)`). Die Beziehungen der realisierten Axiome untereinander zeigt Abbildung 5.7.

5.2.1.3 Assertion

Assertions sind Aussagen, die in die ABox der KB eingestellt werden. Sie enthalten ein *Axiom*, das festlegt, ob es sich um eine *RoleAssertion* (`related betty doris has-child`) oder eine *ConceptAssertion* (`instance peter father`) handelt.

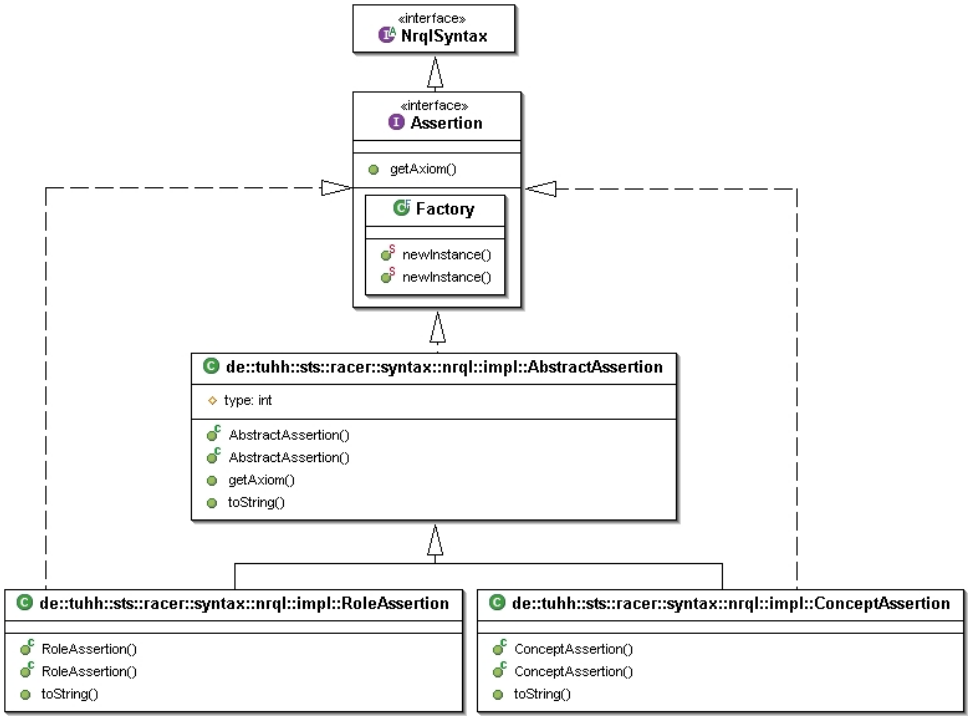

Abbildung 5.8: Klassendiagramm Assertion

Das Klassendiagramm für die implementierten *Assertions* ist in Abbildung 5.8 dargestellt.

5.2.1.4 NrqlQuery

Eine *NrqlQuery* bildet eine Anfrage in nRQL-Syntax ab. Die für den Prototypen implementierten Anfragen sind die *RetrieveQuery* und die *RetrieveUnderPremiseQuery*, die beide Unterklassen der abstrakten *AbstractQuery* darstellen.

Eine Query besteht aus einem Head, in dem die Variablenbindung festgelegt wird, und einem Body, der die Abfragebedingungen enthält. Im Falle der *RetrieveUnderPremise*-Query ist zusätzlich noch eine Prämisse enthalten.

Die Implementierungsdetails dieser Komponenten liefert die folgende Übersicht. Dabei wird auf die weiter oben im Abschnitt eingeführten *Axiome* und *Assertions* zurückgegriffen.

QueryHead Der *QueryHead* enthält eine Liste von Variablen. Im Rahmen der Einschränkungen dieser Arbeit sind Variablen stets als *MustBindVariable* realisiert.

QueryBody Der *QueryBody* besteht aus einem oder mehreren Axiomen. Diese werden bei der Ausgabe durch Konjunktion verknüpft.

QueryPremise Die Prämisse, sofern vorhanden, wird durch eine Liste von *Assertions* realisiert.

5.2.2 Das Paket *OwlQlSyntax*

Das Paket *OwlQLSyntax* enthält zwei Hauptklassen: den *OwlQlRequest* und die *OwlQlResponse*. Die Klassen dienen als Fassade für die XMLBeans-Klassen und stellen Methoden für den Aufruf des *Translators* bereit.

Die Kommunikation zwischen dem Service und dem Manager erfolgt durch Austausch von Request- und Response-Objekten. Abbildung 5.9 zeigt die Interfaces der beiden Klassen.

Der Einsatz des Facade-Patterns zum Verbergen der XMLBeans-Klassen bringt den Vorteil, dass, wenn nach einer Schema-Änderung die XMLBeans neu kompiliert werden müssen, weiterhin eine stabile Schnittstelle für die Kommunikation bereitsteht. Lediglich die Implementierung der Getter- und Setter-Methoden muss ggf. angepasst werden.

Die Methoden *getNrqlQuery()* der Klasse *OwlQlRequest* und *getAnswerBundle()* der Klasse *OwlQlResponse* delegieren den Zugriff auf den *Translator* und liefern nRQL- bzw. OWL-QL-Syntax zurück.

5.2.3 Translator

Der Übersetzer ist eine der wichtigsten Komponenten der Applikation und seine Entwicklung ein Hauptbestandteil dieser Arbeit. Die Aufgabe des *Translators* ist die Entgegennahme der OWL-QL Nachrichten als XML-Dokument und die Umwandlung in nRQL Query-Strings.

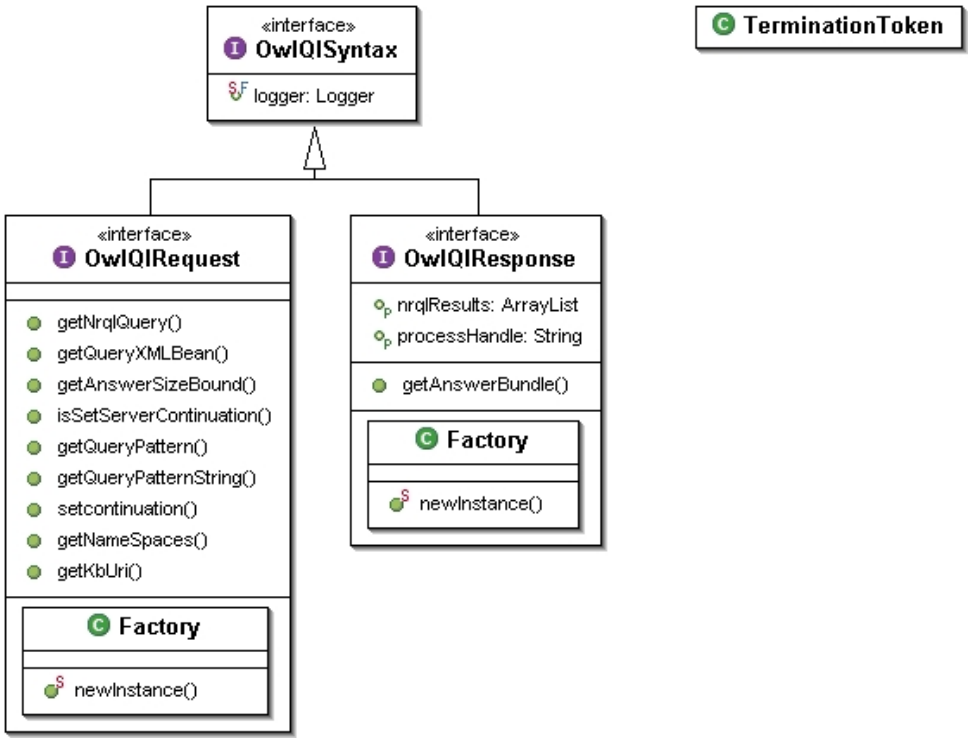

Abbildung 5.9: Klassendiagramm syntax.owlql

Der Aufruf der Übersetzung erfolgt innerhalb der *OwlQlResponse*- und *OwlQlRequest*-Objekte, also nachdem die OWL-QL Nachricht in eine XMLBean umgewandelt wurde, und ist daher unabhängig von der Art des Zugriffs (File, Text, Web Service).

Der *Translator* benutzt das *NrqlSyntax* Paket und ist als Bibliothek in die Anwendung eingebunden. Er enthält keine Status-Information über den Dialog, den Service oder den Kontext der Anwendung.

Der Zugriff erfolgt dabei über die beiden Methoden *getNrqlQuery* und *getAnswerBundle*.

5.2.3.1 Parsen von Prämisse und QueryPattern

Da das OWL-QL Schema bei den Elementen Premise und QueryPattern nur sehr lose spezifiziert, wie die Inhalte dieser Elemente auszusehen haben[3], können diese nicht direkt über die aus dem Schema generierten XMLBeans eingelesen werden, weil keine entsprechende Klasse existiert, die den Inhalt der Elemente abbilden könnte. Sie bleiben daher als RDF/XML-Content erhalten.

[3] RDF, Typ: „xml:any“, siehe [40]

Die Aufgabe des *Translators* ist nun, diese RDF Inhalte zu lesen und in nRQL Syntax umzuwandeln. Dazu bedient er sich der API des Jena Frameworks, das in Anhang A.2 näher vorgestellt wird.

Den umfassenden Rahmen eines RDF-Dokuments bildet bei Jena die Klasse *Model*. Mit Hilfe der Methoden von *Model* lässt sich über die RDF-Statements iterieren.

Jedes einzelne Statement wird in seine Atome zerlegt. Dabei wird abhängig von der Stellung im Dokument (Premise oder QueryPattern) und der Funktion im Statement (Subjekt, Prädikat, Objekt) ermittelt, ob es sich bei dem Atom um eine Variable, ein Individuum oder ein Konzept handelt.

Die *AtomFactory* erzeugt auf Basis dieser Informationen dann ein entsprechendes Atom-Objekt und speichert es in einer *AtomList*. Die *AtomList* wird verwendet, um eine *Assertion* (im Falle der Premise) oder ein *Axiom* (im Falle des QueryPatterns) zu erzeugen.

5.2.3.2 Parsen der Variablen

Das Element MustBindVars muss auf eine andere Weise als die zuvor beschriebene ausgelesen werden. Jena verwirft beim Einlesen der RDF-Statements in ein *Model* die Dokumenten-Struktur, weshalb sich die Reihenfolge der Elemente innerhalb eines Jena-*Models* nicht beeinflussen lässt. Dies stellt bei der Premise und dem QueryPattern kein Problem dar, da RDF-Statements konjunktiv sind und sich die Bedeutung der Aussage durch Vertauschen von Statements nicht ändert.

Im Falle der Variablen-Bindungen wird es jedoch zu einem Problem, da durch Vertauschen von Variablen die Aussage einer Query verfälscht wird. Das folgende Beispiel demonstriert dieses Problem, die beiden Fragen sind nicht äquivalent, sondern vertauschen die Rollen von Kindern und Eltern.

```
 (retrieve (?x ?y) (?x ?y has-child))
(retrieve (?y ?x) (?x ?y has-child))
```

Da sich über das Jena-*Model* kein Ergebnis erzielen lässt, wird zum Parsen der Variablen auf das API *XMLCursor* der XMLBeans-Frameworks zurückgegriffen.

Mit Hilfe der Methoden des Cursors kann zwischen XML-Knoten navigiert werden. Die Must-Bind-Variablen lassen sich dann aus den Attributen der Variable-Elemente auslesen (siehe Beispiel 3.4).

Anschließend wird eine Liste von *VariableAtoms* erzeugt, die im *QueryBody* des *Query*-Objekts gespeichert wird.

5.2.3.3 Ausgabe der nRQL-Query

Da bereits beim Einlesen des OWL-QL-Dokuments im *Translator* eine Übersetzung in die einzelnen Bestandteile einer Query vorgenommen wird, erfolgt die Ausgabe eines nRQL-Query-Strings durch den Aufruf der *toString()*-Methode des entsprechenden Query-Objekts.

An dieser Stelle ist jedoch die Auswirkung der Unique-Name-Assumption von RACER zu beachten, die in 3.4 beschrieben wurde. Die Ausgabe der Variablen im nRQL-String erfolgt deswegen nicht als „?x“, sondern als „$?x“.

5.2.3.4 Ausgabe des AnswerBundle

Bei der Übersetzung von nRQL in ein OWL-QL-*AnswerBundle* Dokument erhält der *Translator* als Parameter ein *OwlQlResponse*-Objekt, das eine Liste von nRQL-Ergebnissen enthält. Diese Ergebnisse sind Axiome, die ein Individuum an eine Variable binden.

Bei der Erzeugung des *AnswerBundles* werden wieder die generierten XMLBeans verwendet, die die Schema-Struktur abbilden.

Für jede Antwort in dieser Liste erzeugt der *Translator* nun ein *Answer*-Element. Dazu wird mit Hilfe des Jena-*Models* für jede Antwort in der Ergebnis-Liste eine Folge von RDF-Statements generiert.

Um dies zu erreichen, werden im *QueryPattern* der OWL-QL-Frage alle Vorkommen von Must-Bind-Variablen durch die entsprechende Individuen-Bindung ersetzt. Analog dazu werden im *BindingSet* die Variablen durch Setzen des Wertes als Attribut aufgelöst.

5.3 Das Paket Service

Das Paket Service stellt die Präsentationsschicht der Architektur dar. Clients, die über SOAP mit dem System kommunizieren, greifen ausschließlich auf die beiden in diesem Paket realisierten Web-Services zu.

Die beiden momentan realisierten Dienste sind der *NrqlSimpleService* und der *OwlQlService*. Sie stellen Axis Service Endpoints dar (vgl. A.1). Der Zugriff erfolgt über RPC via SOAP.

Die Schnittstellen der Klassen sind sehr einfach gehalten. Beide Dienste implementieren jeweils eine Methode *getResult*.

```
1 public String getResult(String query)
```

Die Verarbeitung des Requests und das Generieren der Antwort unterscheidet sich in ihrer Komplexität jedoch erheblich zwischen dem NrqlService und dem QwlQlService.

5.3.1 NrqlService

Der *NrqlService* erhält als Parameter einen String, der einen Befehl in nRQL-Syntax enthält. Da keine weitere Verarbeitung dieses Parameters erforderlich ist, wird eine direkte TCP-Verbindung zu einem Proxy oder Server geöffnet. Diese Verbindung wird über einen Java Socket und entsprechende In- und Output-Streams realisiert. Der nRQL-String wird gesendet und die Antwort des Servers entgegengenommen.

Die Antwort wird an den Web Service Client zurückgesendet.

5.3.2 OwlQlService

Im Falle des *OwlQlService* ist der Parameter ebenfalls ein String, jedoch enthält dieser nicht ein einzelnes Kommando, sondern ein serialisiertes OWL-QL-Dokument.

Auf dem Server liegen mittels XMLBeans generierte Klassen, die alle Elemente des OWL-QL Schemas in Java abbilden. XMLBeans unterstützt das Serialisieren und Deserialisieren von XML-Dokumenten aus einem String.

Die erste Aufgabe des Service ist das Einlesen des *Query*-Dokuments in ein *Query*-XMLBean. Dabei wird gleichzeitig die Syntax des Frage-Dokuments geprüft und eine Fehlermeldung an den Client zurückgesendet und die weitere Bearbeitung abgebrochen, wenn die gesendete XML-Nachricht nicht dem OWL-QL Schema entspricht.

Wurde ein valides Dokument übergeben, liegt es nach dem Einlesen als Java Objekt vor. Mit der Übergabe dieses *Query*-Objekts wird die weitere Verarbeitung an den *RacerOWL-Manager* übertragen.

Dazu wird ein *OwlQlRequest*-Objekt generiert (5.2.2), das als Wrapper für die Query dient und nur die benötigten Methoden nach außen zur Verfügung stellt.

Als Rückantwort vom Manager erhält der *OwlQlService* ein *OwlQlResponse*-Objekt. Die Antwort an den Client erfolgt in Form eines *AnswerBundle*-Objekts (generiert vom *Translator*), das für den Transport über SOAP jedoch wieder zu einem XML-String serialisiert wird.

5.4 Zusammenführung der Teilprojekte

Nach der Realisierung der Komponenten in den beschriebenen Teilprojekten wurden diese im Rahmen des Abschlusses dieser Arbeit zu einem Gesamtsystem zusammengefasst. Die dabei erforderlichen nachträglichen Modifikationen werden hier kurz aufgeführt.

Integration in Tomcat Im Gegensatz zum Architekturentwurf wurden der RACER-Proxy und der *DialogManager* im Paket *Connection* nicht als Tomcat-Komponenten entwickelt. Dies führte dazu, dass die Anwendung zwar als Konsolen-Applikation lauffähig, jedoch nicht als Web Service einsetzbar war.

Die in den beiden Teilprojekten propagierte Lösung durch den Einsatz eines *ProxyLocators*, welcher *DialogManager* und *Proxy* physisch entkoppeln sollte, erwies sich als nicht ausreichend, um diese Probleme zu umgehen.

Um trotzdem einen lauffähigen Web Service bereitstellen zu können, wurde der Proxy in Tomcat integriert. Dazu wurden Klassen, die auf der Konsole als Thread laufen sollten, durch Singletons ersetzt, die in der Laufzeitumgebung des Servlet Containers dauerhaft zur Verfügung stehen.

Gemeinsames Logging Die Logging-Mechanismen des Proxys wurden angepasst, so dass jetzt für alle Komponenten eine gemeinsame, über die zentrale Tomcat-Konfiguration administrierbare Ausgabeplattform für Systeminformationen bereitsteht.

Fehlerbehandlung Die Fehlerbehandlung musste so umgeschrieben werden, dass Exceptions in den unteren Schichten der Architektur zum Service Layer durchgereicht werden, damit die Client/Server-Kommunikation bei Problemen, etwa bei der Verbindung zu einem RACER-Server, korrekt unterbrochen werden kann.

Unterstützung von Prämissen Im Syntax-Parser des Proxys traten Probleme bei der Interpretation von `retrieve-under-premise`-Queries auf. Dadurch wurden Anfragen dieser Art nicht korrekt an den RACER-Server übertragen und die Fragen konnten demzufolge nicht beantwortet werden. Durch Modifikation der entsprechenden Methoden konnte dieses Problem provisorisch umgangen werden.

Das System ist im Rahmen der in dieser Arbeit gemachten Einschränkungen funktionstüchtig und einsatzbereit. Eine Erörterung noch bestehender Probleme und eine Betrachtung möglicher Weiterentwicklungen folgt im nächsten Kapitel.

Kapitel 6

Zusammenfassung und Ausblick

In diesem Kapitel erfolgt eine Zusammenfassung der Arbeit und eine abschließende Betrachtung der Ergebnisse im Bezug auf die in der Einleitung formulierte und in der Analyse konkretisierte Aufgabenstellung.

Im Anschluss daran werden in einem Ausblick mögliche Weiterentwicklungen diskutiert.

6.1 Zusammenfassung

Ziel der Arbeit war die Erweiterung/Neuentwicklung des RACER-Systems um den Zugriff über Web Services. Besondere Berücksichtigung sollte dabei die Erarbeitung eines Konzepts und einer prototypischen Implementierung einer Softwarelösung sein, die es erlaubt, den W3C Standard-Kandidaten OWL-QL als Anfrage Sprache und Protokoll für den RACER-Server nutzen zu können.

6.1.1 nRQL-Service

Der erste Teil dieser Aufgabenstellung konnte einfach und schnell umgesetzt werden. Anstelle der bisherigen Kommunikation über TCP können nun Zeichenfolgen[1] in nRQL über einen SOAP-Service gesendet werden. Die Rückantwort erfolgt ebenfalls über eine Zeichenfolge, indem die Antwort des RACER Servers direkt an den Web-Service Client zurückgegeben wird.

Auf der Client-Seite kommt bei dieser Lösung eine leicht modifizierte JRacer2-API zum Einsatz, bei der die entsprechende Methode zur TCP Kommunikation gegen einen Service Aufruf ausgetauscht wurde.

[1] Befehle oder Anfragen

6.1.2 OWL-QL Service

Der OWL-QL-Service erhält eine Anfrage in Form eines Query-Dokuments. Er erzeugt daraus ein Request-Objekt, das aus dem Dokument in Form einer XMLBean besteht und Schnittstellen für den Zugriff auf dessen Elemente bereitstellt.

Das Request-Objekt dient zum Transport der Query-Informationen zwischen Service- und Manager-Layer. Über den eingebundenen Translator und das Syntax-Paket kann der Manager auf die enthaltende Query in nRQL-Syntax, und ggf. Status-Informationen zum Dialog zwischen Client und Server zugreifen. Der Manager erkennt laufende Dialoge und setzt sie fort. Zusätzlich wird bei neuen Queries untersucht, ob bereits eine Ergebnismenge für diese Anfrage in einem Dialog gespeichert ist. In diesem Fall wird keine neue Anfrage an den Server generiert, sondern auf bestehende Antowrten zurückgegriffen.

Die Antwort des Systems an den Service erfolgt in Form eines Response-Objektes, das eine Liste von nRQL-Antworten und das verwendete Query-Objekt enthält. Aus diesen Daten wird ein OWL-QL AnswerBundle-Dokument generiert und an den Client zurückgesendet.

6.2 Offene Probleme

Nach der Implementierung des Prototypen mussten einige Probleme ungelöst bleiben. Als Grundlage für mögliche Weiterentwicklungen des Systems werden sie an dieser Stelle kurz aufgeführt.

Integration Proxy/Tomcat Die Integration des Proxys in den Tomcat Servlet Container erfolgte prototypisch im Anschluss an die Implementierung des Service und des Translators. Bei der Verwendung als eigenständiges Programm realisiert der Proxy einen Großteil seiner Funktionalität durch das Erzeugen und Überwachen von Threads. Beim Einsatz innerhalb des Servlet Containers sollte das Starten von eigenen Threads jedoch unterbleiben, da die Verwaltung der Abläufe durch den Container realisiert wird.

Im Rahmen dieser Arbeit wurde die Funktionsweise des Proxys nur soweit an die Erfordernisse des Servlet Containers angepasst wie es zum Betrieb des Prototypen erforderlich war. Eine komplette Überarbeitung der Proxy Architektur mit dem Ziel, ihn als Servlet in Tomcat zu betreiben, steht noch aus.

Server-Blockade Ein Dialog speichert neben der gestellten Frage auch eine Referenz auf den Server, der die vorgenerierten Ergebnismengen bereithält. Ein RACER Server blockiert jedoch während der Bearbeitung einer Anfrage alle weiteren Zugriffe von Clients.
Will bei der Fortsetzung eines Dialogs ein Client nun auf einen bestimmten Server zugreifen, der mit einer längeren Anfrage beschäftigt ist, kommt es zum Stau.
Dieses Problem entsteht, weil der Proxy in seiner ursprünglichen Form davon ausgeht, dass alle angebundenen Server dieselben Informationen enthalten, was im Falle der Speicherung von Ergebnislisten für bereits erfolgte Anfragen nicht mehr gegeben ist.

Einen möglichen Ausweg bietet das dynamische Klonen von Servern samt ihrer Statusinformationen zur Laufzeit des Proxys. Entsprechende Überlegungen werden am Arbeitsbereich bereits erörtert.

Doc/Literal-Style Web Services Bedingt durch den beta-Zustand der verwendeten Axis- und der XMLBeans-Version traten Probleme bei der Serialisierung und Deserialisierung von OWL-QL Dokumenten auf. Aufgrund dieser Probleme wird beim momentanen Entwicklungsstand das Dokument auf der Client-Seite in einen String geschrieben und auf der Serverseite wieder aus dem String in ein XMLBeans-Objekt deserialisiert.

Da der OwlQlService im Wesentlichen ein Query-Dokument als XML entgegennimmt, ergänzt höchstens noch um ein User Session-Management, böte sich der Einsatz von Document/Literal-Style Web Services an.
Mit einer Unterstützung dieses Ansatzes durch die verwendeten Frameworks ist in zukünftigen Versionen zu rechnen. Die SOAP/RPC-Methodenaufrufe über einen String-Parameter könnten dann durch Verankerung des OWL-QL-Schemas im WSDL-Dokument des Service entfallen.

6.3 Kritische Betrachtung von OWL-QL

Die Implementierung des Translators geschah in enger Anlehnung an die zur Verfügung stehenden XML-Schemata von OWL-QL. In diesem Zusammenhang traten einige Probleme auf, die in diesem Abschnitt erörtert werden sollen.

Inkompatibilität von Veröffentlichungen und Schema Die Umsetzung des Standards wurde in enger Orientierung am XML-Schema der Frage- und Antwort-Dokumente vorgenommen. Leider sind die Schemata, obgleich als offizielle Referenz für die Syntax angeführt, streckenweise fehlerhaft oder veraltet.

Im Hinblick auf die weite Verbreitung von XML wäre es wünschenswert, wenn die Weiterentwicklung des Standards auch im Schema dokumentiert wird.

Keine Unterstützung von Fehlerbehandlung Ein weiterer Nachteil des Protokolls ist das Fehlen eines Verfahrens zur Fehlerbehandlung. Das System sieht keine expliziten Elemente für den Transport von Meta-Informationen oder Fehlermeldungen vor. Als Workaround wird der Missbrauch des Termination-Tokens als Transporter für Fehler-Codes empfohlen. Während jedoch beim Query-Dokument die Token einen freien Text-Parameter erlauben, sind beim AnswerBundle die möglichen Token[2] fest definiert, so dass hier zur Übergabe von zusätzlichen Informationen in das Schema eingegriffen werden muss.

Wechsel der Prämisse im laufenden Dialog Bei der client-seitigen Fortsetzung eines Dialogs wird statt des QueryPatterns das ContinuationToken übergeben, zusätzlich ist jedoch die Angabe einer neuen Prämisse für die Frage möglich. Dies macht keinen Sinn, wenn von der Tatsache ausgegangen wird, dass der Server bei der Fortsetzung des Dialogs auf eine im Voraus berechnete Ergebnismenge zugreifen kann.

6.4 Ausblick

Während der Entwicklungsphase sind einige Aspekte zu Tage getreten, die das Arbeiten mit dem System in seiner Gesamtheit und dem RACER-Server im Besonderen erschweren. Neben den bereits erwähnten Problemen, die im Rahmen zukünftiger Entwicklungen auf Basis des Systems bearbeitet werden sollten, bieten sich die folgenden Ansätze zur weiteren Untersuchung an.

6.4.1 Objekt-Orientierte API für nRQL

Die Kommunikation mit einem RACER-Server erfolgt über Strings. Dies ist aufgrund der LISP-Syntax von nRQL und der historischen Entwicklung des Servers vertretbar und effektiv, hat jedoch beim Einsatz objekt-orientierter Techniken den Nachteil, dass jede Komponente, die auf Antworten des Servers zugreifen muss[3], einen geeigneten Algorithmus zum Parsen und Verarbeiten der Strings umsetzen muss.

[2] end, rejected usw., nicht alle in der Dokumentation erwähnten Token sind im Schema implementiert
[3] in dieser Arbeit sind das der Proxy, der DialogManager und der Translator

Für weiterführende Arbeiten an der Architektur wäre ein zentrales API, das die Generierung und das Parsen der nRQL-Strings übernimmt, von Vorteil. Mit der JRacer2-Bibliothek und dem in dieser Arbeit entwickelten *Nrql-Syntax*-Paket liegen dazu bereits Ansätze vor, die jedoch weiter verfolgt werden müssen, um universell einsetzbar zu sein.

Die Unabhängigkeit von der Implementierungssprache dieses API, wenngleich auf Kosten der Performance, könnte durch die Verwendung von XML als Eingangs- bzw. Ausgangsdokument erreicht werden.

6.4.2 Unterstützung von OWL-QL

Bei der Realisierung des Standards OWL-QL wurden Einschränkungen des Umfangs in Kauf genommen.
Dies betrifft in erster Linie die unterstützten Anfrage-Methoden. Eine für diese Arbeit gesetzte Einschränkung war die Reduzierung der Queries auf A-Box-Queries. Die Schnittmenge von OWL-QL und nRQL umfasst jedoch auch bestimmte T-BOX-Queries wie die Frage nach Unterklassen, Listen von Klassen usw..

Ein weiterer Punkt, der bewusst nicht vertieft wurde, ist der Umgang mit verschiedenen Knowledge-Bases, insbesondere die Verwendung des AnswerKBRef-Elements als Variable. Dies würde Anfragen ohne explizite Angabe einer KB-URI ermöglichen. Die Antwort des Servers umfasste dann zusätzlich zu den Ergebnissen die Information darüber, unter Verwendung welcher KB die Antworten generiert worden sind.

Dies ist für den Einsatz des Systems im Semantic Web von großer Bedeutung, da nur so die Nachvollziehbarkeit der Ergebnisse gewährleistet werden kann.

6.5 Fazit

Der Ansatz für die Übersetzung der Queries erfolgte dokumenten-zentriert und orientiert sich eng am Schema von OWL-QL. Die konsequente Orientierung an der Schema-Definition und der RDF/XML-Syntax unterscheidet das realisierte System von bestehenden Ansätzen[4].

Daraus entstehende Probleme, verursacht durch Inkonsistenzen und Defizite im Schema, wurden erörtert. Es ist davon auszugehen, dass sie auf dem Weg von OWL-QL zu einem stabilen Standard gelöst werden.

[4](Stanford OWL-QL Services, [42], Manchester DQL Service [43]).

Die Kapselung der Verarbeitung der Dokumente durch offene Frameworks und die Implementierung einer Fassade für den Zugriff auf die entstehenden Objekte entkoppelt die Übersetzung und die Kommunikation innerhalb des Systems vom vorliegenden Schema. Die Architektur kann daher bei späteren Modifikationen des Schemas sehr flexibel an die neuen Anforderungen angepasst werden.

Da bei der Auswahl der Werkzeuge konsequent auf Open Source Projekte gesetzt wurde, die eine große Entwicklergemeinde und starke Partner (Apache, HP) haben, kann erwartet werden, dass zukünftige Weiterentwicklungen im Bereich dieser Frameworks ohne umfangreiche Reengineering-Prozesse auch dem vorliegenden System zu Gute kommen können.

Mit der Entwicklung des neuen RACER-Systems und der erfolgten prototypischen Implementierung wurde ein wichtiger Schritt in Richtung der Unterstützung von OWL-QL-Anfragen durch RACER unternommen.

Anhang A

Eingesetzte Werkzeuge

Für die Realisierung dieser Arbeit wurden verschiedene Werkzeuge und Frameworks eingesetzt. Sowohl in Hinblick auf die Transparenz der Lösung, als auch zur Vermeidung lizenzrechtlicher Probleme bei Einsatz und Weiterentwicklung der Software, wurden dabei ausschließlich freie Open-Source Projekte verwendet.

Die Verwendung offener, generischer Frameworks zur Lösung von spezifischen Problemen bringt zusätzlich den Vorteil, dass spätere Weiterentwicklungen aus dem reichhaltigen Informations- und Dokumentationsangebot der Entwicklergemeinschaft schöpfen können und nicht allein auf die Dokumentation eines Entwicklers angewiesen sind.

Diese Projekte, ihre Funktionsweisen, sowie ihr Einsatz und ihre Konfiguration sollen in diesem Anhang näher vorgestellt werden.

A.1 Apache Axis

Apache Axis [1] ist ein Open Source Java Framework, mit dem auf dem SOAP-Standard basierende Programme in einer einheitlichen Umgebung entwickelt und deployed werden können. Es ist das Nachfolgeprojekt des Apache SOAP Frameworks der Apache Software Foundation.

Axis findet bei der Implementierung von WebService Clients und Server gleichermaßen Verwendung und unterstützt den Entwickler mit einer Reihe von Werkzeugen, um die WSDL Beschreibungen für Services zu erstellen oder Clients zu generieren.

Die in dieser Arbeit verwendete Version ist der Release Candidate 2 der Version 1.2. Die Software kann über die Projekt-Homepage bei der Apache Foundation bezogen werden [31].

[1] Apache eXtensible Interaction System

A.1.1 Architektur von Axis

Axis ist ein modular aufgebautes System. Die Axis-Engine, die auf der Serverseite zum Einsatz kommt, besteht aus spezialisierten Subsystemen. Abbildung A.1 zeigt das Komponentendiagramm der Engine mit dem Nachrichtenpfad einer ein- bzw. ausgehenden SOAP-Nachricht.

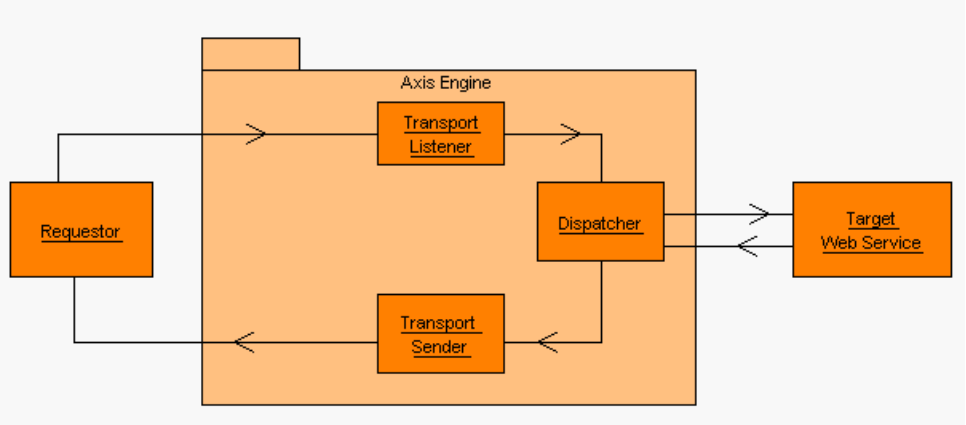

Abbildung A.1: Architektur der AXIS-Engine

Die einzelnen Komponenten der Engine werden durch Handler realisiert. Diese Handler werden Ereignissen beim Parsen der XML-Nachrichten zugordnet. Ein Handler empfängt eine Nachricht, verarbeitet sie und sendet seinerseits eine Nachricht an den folgenden Handler.

Die Verkettung von Handlern wird als Filterkette bezeichnet. Die Zusammensetzung der Kettenglieder ist konfigurierbar, was dem Entwickler eine flexible Kontrolle über die Prozesse innerhalb der Axis-Engine ermöglicht.

Der Ablauf der Kommunikation zwischen dem Web Service Client und der Axis-Engine erfolgt in den folgenden Schritten (vgl. [2]):

1. Der Requestor (Client) generiert eine SOAP-Nachricht und sendet sie an den Server
2. Der *TransportListener* empfängt die Nachricht. Beim Einsatz innerhalb eines Servlet-Containers ist der Listener typischerweise als Servlet realisiert.
3. Der Listener trennt protokoll-spezifische Daten vom SOAP-Inhalt und erzeugt ein internes MessageObject.
4. Dieses MessageObject wird an den Dispatcher weitergegeben. Dieser enthält einen Provider, der die entsprechende Service Methode aufruft, den Rückgabe-Wert entgegen nimmt und ein ReturnObject generiert.

5. Der TransportSender (ebenfalls als Sevlet realisert) empfängt dieses Objekt und generiert eine SOAP-Nachricht, die wieder um transport-protokoll-spezifischen Information erweitert und an den Requestor zurückgesendet wird.

A.1.2 Einsatz von Axis

Axis bietet dem Entwickler einige nützliche Werkzeuge für das Erstellen von Web Services und Web Service Clients. Dazu gehören vor allem die automatische Generierung von WSDL-Dokumenten auf Basis einer Java-Klasse (Java2WSDL) und die Erzeugung von Client-Code unter Berücksichtigung der Web Service Beschreibung.

Durch die Verwendung dieser Werkzeuge lassen sich Client-Server-Systeme mit SOAP als Nachrichtenprotokoll einfach realisieren. Abbildung A.2 zeigt den schematischen Aufbau eines solchen Systems unter Berücksichtigung der erwähnten Werkzeuge.

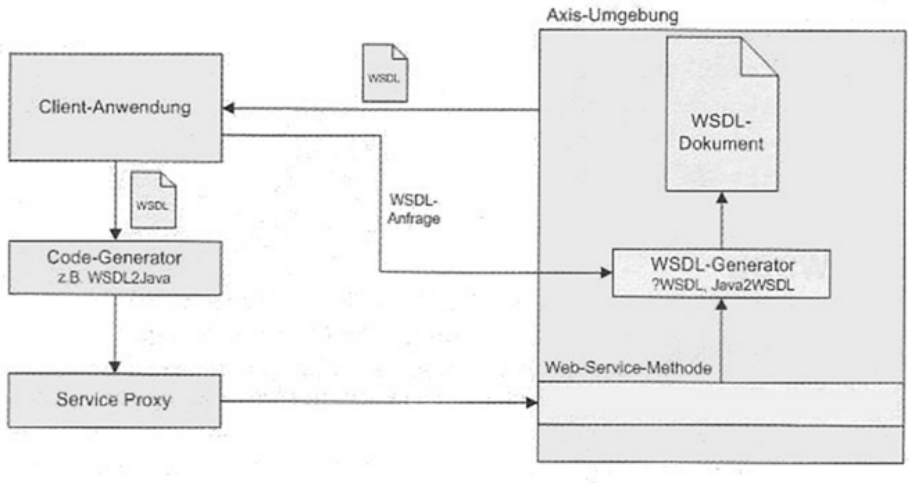

Abbildung A.2: Einsatz von WSDL mit Axis

Die Realisierung findet in folgenden Schritten statt:

1. Der Entwickler des Services schreibt eine normale öffentliche Java-Klasse und generiert mit Java2WSDL eine WSDL-Datei, die den Service beschreibt. Ohne Veränderungen an der Konfiguration werden auf diese Art und Weise alle *public* Methoden der Klasse als Methoden des Service angeboten.

2. Der Entwickler des Clients lädt die WSDL-Datei vom Server des Anbieters und generiert mit dem Programm WSDL2Java einen Satz von Java-Klassen, die den Zugriff auf den Service und seine Methoden realisieren.

3. In der Client-Anwendung erzeugt der Entwickler ein Proxy-Objekt, dass als lokaler Platzhalter für den entfernten Dienst verwendet wird.
4. Der Zugriff auf die Methoden des Service erfolgt über dieses Proxy-Objekt transparent für den Entwickler, die gesamte SOAP-Kommunikation wird von den generierten Klassen und dem Framework gekapselt.

Die automatische Generierung stößt jedoch bei komplexeren Services an ihre Grenzen, etwa wenn umfangreiche Objekte, für die eigene XML-De/Serialisierungs-Methoden geschrieben werden müssen, beteiligt sind. In diesem Fall müssen die Entwickler konkretere Verfahren für die Bereitstellung von Methoden oder deren Konsumierung entwickeln, auch hierbei biete das Framework eine umfangreiche Unterstützung.

Beispiele für Services und Clients finden sich im beiliegenden Quellcode der Arbeit. Zur Vertiefung empfehlen sich die angegebenen Referenzen [2, 1, 15, 31].

A.2 Jena Semantic Web Framework

Das *Jena Semantic Web Framework* wird maßgeblich von den HP Labs [37] im Rahmen ihrer Forschungen im Bereich Semantic Web entwickelt. Es handelt sich bei Jena um Open-Source Software, die über die Projekt-Homepage bei Sourceforge [32] verfügbar ist.

Das Projekt bezeichnet sich als das führende Werkzeug zur Entwicklung von Semantic Web Applicationen mit Java. In dieser Arbeit wird die zur Zeit aktuellste Version Jena2, Version 2.1 verwendet.

Jena stellt dem Entwickler eine Reihe von APIs zum Lesen und Schreiben von RDF-basierten Dokumenten zur Verfügung. Dazu zählen insbesondere Parser und Writer für die Sprachen OWL und DAML+OIL.

Darüber hinaus bietet Jena mit der Anfragesprache RDQL die Möglichkeit Queries in einer SQL-ähnlichen Syntax an Dokumente zu stellen und liefert eine eigene, regelbasierte Inference-Engine mit. Abbildung A.3 zeigt den Aufbau der Jena-Architektur.

Die Architektur ist in drei Schichten realisiert. Die Basis ist der *Graph-layer*. Er bildet einen schlanken Kern, der RDF-Graphen, Triples und Knoten verwaltet und Manipulationen auf ihnen erlaubt.

Der *EnhGraph-Layer*[2] definiert die internen Schnittstellen (SPIs[3]), gegen die die externen APIs entwickelt werden. Die Schicht realisiert die notwendigen Erweiterungen für verschiedene Sprachkonzepte.

[2] Enhanced Graph
[3] System Programmer Interface

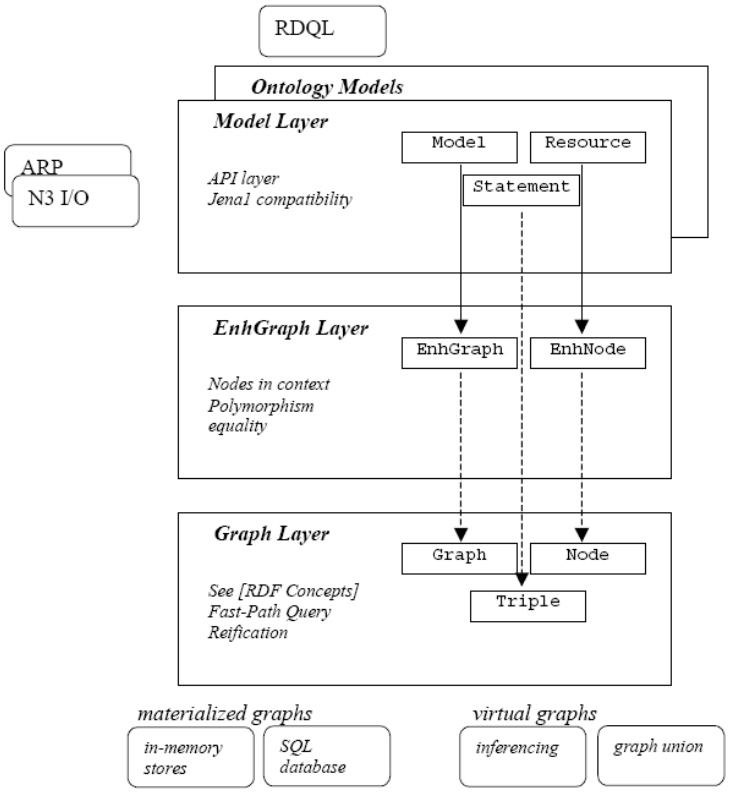

Abbildung A.3: Architektur von Jena2 [11]

Im *Model-Layer* werden die APIs realisiert und dem Anwender zur Verfügung gestellt. Der Zugriff von außen erfolgt dabei stets über ein Model-Objekt, das die weiteren Methoden implementiert. Durch diese Art der Umsetzung wird weitgehende Transparenz für den Anwender gewährleistet.

Alternativ zur Nutzung des *Model-Layers* erlauben die *Ontology Models* die Verwaltung von KB-Präsentationen mit Jena und damit den bereits erwähnten Zugriff auf die eigene Inference-Engine über RDQL.

A.2.1 Einsatz des Frameworks

Für die Realisierung dieser Arbeit wurden lediglich die OWL/RDF APIs aus dem *Model-Layer* benötigt. Sie wurden verwendet, um die durch das offene OWL-QL Schema undefinierten, und daher durch das XML-Beans-Framework nicht zu erfassenen RDF-Elemente innerhalb der OWL-QL Dokument zu parsen und zu erzeugen.

Dieser Unterabschnitt stellt den prinzipiellen Umgang mit dem Jena-Framework anhand von einfachen Beispielen vor, die dem Buch „Practical RDF“ entnommen sind [17].

A.2.1.1 Erzeugen von RDF

Das Beispiel in Quellcode A.1 erzeugt ein Jena „In-Memory“-Model [4] mit einer Ressource $Monsters_1$. Mit der Methode *addProperty* werden dann die zuvor definierten „related“ Beziehungen (*Property*) zu den Objekten $Monsters_2$ und $Monsters_3$ modelliert.

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.common.PropertyImpl;
import java.io.*;

public class pracRdfFirst extends Object {
    public static void main (String args[]) {
        String sURI = "http://burningbird.net/articles/
           monsters1.htm";
        String sPostcon = "http://www.burningbird.net/postcon
           /elements/1.0/";
        String sRelated = "related";

        try {
            // Create an empty graph
            Model model = new ModelMem( );

            // Create the resource
            Resource postcon = model.createResource(sURI);

            // Create the predicate (property)
            Property related = model.createProperty(sPostcon,
                sRelated);

            // Add the properties with associated values (
               objects)
            postcon.addProperty(related, "http://burningbird.
               net/articles/monsters3.htm");
            postcon.addProperty(related, "http://burningbird.
               net/articles/monsters2.htm");

            // Print RDF/XML of model to system output
            model.write(new PrintWriter(System.out));
        } catch (Exception e) {
```

[4] Im Gegensatz zum persistenten Model, das sich bspw. direkt mit einer relationalen DB verbinden ließe

```
29              System.out.println("Failed: " + e);
            }
        }
}
```

Quellcode A.1: Erzeugung von RDF mit Jena, Java-Klasse

Es entstehen die zwei RDF-Statements

- $Monster_1 \rightarrow related \rightarrow Monster_2$
- $Monster_1 \rightarrow related \rightarrow Monster_3$

Das gesamte Model wird über die Methode *write* nach *System.out* geschrieben. Der erzeugte RDF-Code ist in Quellcode A.2 einsehbar.

```
<rdf:RDF
    xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
3   xmlns:NS0='http://www.burningbird.net/postcon/elements
        /1.0/'
>
<rdf:Description rdf:about='http://burningbird.net/articles/
   monsters1.htm'>
    <NS0:related>http://burningbird.net/articles/monsters3.
        htm</NS0:related>
    <NS0:related>http://burningbird.net/articles/monsters2.
        htm</NS0:related>
8 </rdf:Description>

</rdf:RDF>
```

Quellcode A.2: Erzeugung von RDF mit Jena, RDF-Output

A.2.1.2 Parsen von RDF

Im zweiten Beispiel (Quellcode A.3) soll ein RDF-Dokument in ein Model eingelesen werden.

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.burningbird.postcon.vocabulary.POSTCON;

5 public class pracRDFSeventh extends Object {
    public static void main (String args[]) {
        String sUri = args[0];
        String sResource = args[1];
```

```
        try {
            // Create memory model, read in RDF/XML document
            ModelMem model = new ModelMem( );
            model.read(sUri);
            // Find resource
            Resource res = model.getResource(sResource);
            // Find properties
            StmtIterator iter = res.listProperties( );
            // Print out triple - subject | property | object
            while (iter.hasNext( )) {
                // Next statement in queue
                Statement stmt = iter.next( );
                // Get subject, print
                Resource res2 = stmt.getSubject( );
                System.out.print(res2.getNameSpace( ) + res2.
                    getLocalName( ));
                // Get predicate, print
                Property prop = stmt.getPredicate( );
                System.out.print("␣" + prop.getNameSpace( ) +
                     prop.getLocalName( ));
                // Get object, print
                RDFNode node = stmt.getObject( );
                System.out.println("␣" + node.toString( ) + "
                    \n");
            }
        } catch (Exception e) {
            System.out.println("Failed:␣" + e);
        }
    }
}
```

Quellcode A.3: Parsen von RDF mit Jena, Java-Klasse

Es wird wie im Beispiel vorher wieder ein *Model* erzeugt. Über die Methode *read* wird ein RDF Dokument eingelesen.
Über *model.getResource* in Zeile 14 erfolgt der gezielte Zugriff auf ein Subjekt. Die Iteration über alle Statements dieses Subjektes liefert die bekannten Triples, die dann auf der Konsole ausgegeben werden.

Zu beachten ist der Import von *vocabulary.POSTCON* in Zeile 3. Jena stellt dem Entwickler ein Vokabular zur Verfügung, in dem sprachspezifische Beziehungen und Attribute definiert sind. Anhand dieses Vokabulars erlangt die Klasse Informationen über die verwendeten Properties. Eigene Vokabulare lassen sich von den vorgegebenen ableiten.

Die Ausgabe, die sich ergibt wenn das Programm mit dem in A.2.1.1 erzeugten Dokument und dem URI von $Monster_1$ als zweiten Parameter aufgerufen wird, ist in Quellcode A.4 dargestellt.

```
http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/related
http://burningbird.net/articles/monsters2.htm

http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/related
http://burningbird.net/articles/monsters3.htm
```

Quellcode A.4: Parsen von RDF mit Jena, Console Output

Diese Beispiele haben den den einfachen Einsatz des Jena Frameworks demonstriert. Es existieren noch weitere Konzepte zur Verwaltung komplexerer Strukturen, die hier jedoch nicht berücksichtigt werden sollen. Weitere Beispiele und Dokumentation finden sich in den bereits erwähnten Quellen [32, 17], sowie im Quellcode der Klassen dieser Arbeit.

A.3 XMLBeans

XMLBeans ist ein Framework zur Bearbeitung von XML-Dokumenten mit Java. Das Projekt entstand ursprünglich bei Bea im Rahmen ihres WebLogic Application Servers, wurde von dort aus jedoch der Open Source Gemeinde zur Verfügung gestellt und wird seitdem von der Apache Group verwaltet und weiterentwickelt.

Die in dieser Arbeit verwendete Version ist 2.0beta. Die Software, der Quellcode und die Dokumantation sind über die Projekt-Homepage [33] erhältlich.

Es gibt bereits eine Reihe von Ansätzen um XML-Dokumente mit Java zu bearbeiten. DOM[5]-basierte Parser bilden die Elemente eines Dokuments in einer Baumstruktur ab, SAX[6]-Parser implementieren einen event-gesteuerten Mechanismus und reagieren auf ausgelöste Ereignisse.

Der für Java-Entwickler natürliche Ansatz ist jedoch die objekt-orientierte Behandlung von XML-Inhalten.

XMLBeans stellt dem Anwender dafür drei Kern-APIs zur Verfügung [14].

[5] Domain Object Model
[6] Simple API for XML

XML Cursor Mit Hilfe von des XML-Cursor kann durch XML-Daten, die durch ein Java-Objekt repräsentiert werden, in einer DOM-artigen Baumstruktur navigiert werden.

JavaBeans Zu einem vorliegenden XML-Schema wird ein Satz von JavaBean-Klassen generiert. Ein entsprechender Schema-Compiler ist Bestandteil von XMLBeans. Die generierten Klassen unterstützen den Entwickler beim Lesen und Schreiben schema-kompatibler XML-Dokumente und stellen Objekte für die Elemente bereit, die sich direkt aus Java ansprechen lassen.

XQuery XMLBeans unterstützt XQuery-basierte Anfragen an ein XML-Dokument um gezielt auf einzelne Daten zuzugreifen.

A.3.1 Einsatz von XMLBeans

Der zusammen mit XMLBeans ausgelieferte Schema-Compiler `scomp` generiert die Klasses aus einem Schema durch den Aufruf folgenden Kommandozeilenaufruf:

```
scomp schema.xsd
```

Komfortabler ist die Bedienung durch einen Ant-Task. Zur Anschaulichkeit ist der in diesem Projekt verwendete Task in Anhang B.3 dargestellt.

Anhang B

Ausgewählter Quellcode

In diesem Abschnitt werden zum besseren Verständnis der Implementierung ausgewählte Code-Elemente bereitgestellt.

B.1 OWL QL-Syntax

Die Syntax von Owl-QL wird von den Entwicklern unter [40] angeboten. Für die Umsetzung mit dem Java-Framework XMLBeans waren einige Änderungen erforderlich, deshalb sollen die beiden überarbeiteten Schema-Dokumente *owl-ql.xsd* und *owl-ql-variables.xsd* hier dargestellt werden.

B.1.1 Owl-QL Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
    xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
    targetNamespace="http://www.w3.org/2003/10/owl-ql-syntax#
        "
    xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">


    <xs:import namespace="http://www.w3.org/2003/10/owl-ql-
        variables#" schemaLocation="owl-ql-var.xsd" />

    <xs:element name="requestProofsWithURL"     type="owl-
        ql:requestProofsWithURLType" />
```

```
    <xs:element name="premise"                    type="owl-
       ql:premiseType" />
    <xs:element name="queryPattern"               type="owl-
       ql:queryPatternType" />
    <xs:element name="answerPattern"              type="owl-
       ql:answerPatternType" />
    <xs:element name="mustBindVars"               type="owl-
       ql:mustBindVarsType" />
    <xs:element name="mayBindVars"                type="owl-
       ql:mayBindVarsType" />
    <xs:element name="kbRef"                      type="owl-
       ql:kbRefType" />
    <xs:element name="answerKBPattern"            type="owl-
       ql:answerKBPatternType" />
    <xs:element name="answerSizeBound"            type="owl-
       ql:answerSizeBoundType" />
    <xs:element name="serverContinuation"         type="owl-
       ql:serverContinuationType" />
    <xs:element name="serverTermination"          type="owl-
       ql:serverTerminationType" />
    <xs:element name="query"                      type="owl-
       ql:queryType" />
    <xs:element name="answer"                     type="owl-
       ql:answerType" />
    <xs:element name="continuation"               type="owl-
       ql:continuationType" />
    <xs:element name="answerBundle"               type="owl-
       ql:answerBundleType" />
    <xs:element name="proof"                      type="owl-
       ql:proofType" />
    <xs:element name="proofStep"                  type="owl-
       ql:proofStepType" />


    <xs:complexType name="requestProofsWithURLType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:anyAttribute processContents="skip"
                   namespace="http://www.w3.org/1999/02/22-
                   rdf-syntax-ns#"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <xs:complexType name="premiseType">
        <xs:sequence>
```

```
            <xs:any processContents="skip" namespace="http://
               www.w3.org/1999/02/22-rdf-syntax-ns#"
               minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="queryPatternType">
        <xs:sequence>
            <xs:any processContents="skip" namespace="http://
               www.w3.org/1999/02/22-rdf-syntax-ns#"
               minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="answerPatternType">
        <xs:sequence>
            <xs:any processContents="skip" namespace="http://
               www.w3.org/1999/02/22-rdf-syntax-ns#"
               minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="mustBindVarsType">
        <xs:sequence>
            <xs:any namespace="http://www.w3.org/2003/10/owl-
               ql-variables#" minOccurs="1" maxOccurs="
               unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="mayBindVarsType">
        <xs:sequence>
            <xs:any namespace="http://www.w3.org/2003/10/owl-
               ql-variables#" minOccurs="1" maxOccurs="
               unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="kbRefType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:anyAttribute processContents="skip"
                   namespace="http://www.w3.org/1999/02/22-
                   rdf-syntax-ns#"/>
            </xs:extension>
        </xs:simpleContent>
```

```
    </xs:complexType>

    <xs:complexType name="answerKBPatternType">
        <xs:choice>
            <xs:element ref="kbRef" minOccurs="1" maxOccurs="
                unbounded"/>
            <xs:any processContents="skip" namespace="http://
                www.w3.org/1999/02/22-rdf-syntax-ns#"
                minOccurs="1" maxOccurs="1"/>
            <xs:any namespace="http://www.w3.org/2003/10/owl-
                ql-variables#" minOccurs="1" maxOccurs="1"/>
        </xs:choice>
    </xs:complexType>

    <xs:simpleType name="answerSizeBoundType">
        <xs:restriction base="xs:positiveInteger"/>
    </xs:simpleType>

    <xs:complexType name="serverContinuationType">
        <xs:sequence>
            <xs:element name="processHandle" minOccurs="1"
                maxOccurs="1">
                 <xs:simpleType>
                     <xs:restriction base="xs:string"/>
                 </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="serverTerminationType">
        <xs:sequence>
            <xs:element name="processHandle" minOccurs="1"
                maxOccurs="1">
                 <xs:simpleType><xs:restriction base="
                    xs:string"/></xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="queryType">
        <xs:sequence>
```

```
            <xs:element ref="premise" minOccurs="0" maxOccurs
               ="1"/>
            <xs:choice>
118             <xs:element ref="queryPattern" minOccurs="1"
                   maxOccurs="1"/>
                <xs:element ref="serverContinuation"
                   minOccurs="1" maxOccurs="1"/>
                <xs:element ref="serverTermination" minOccurs
                   ="1" maxOccurs="1"/>
            </xs:choice>
            <xs:element ref="answerPattern" minOccurs="0"
               maxOccurs="1"/>
123         <xs:element ref="mustBindVars" minOccurs="0"
               maxOccurs="1"/>
            <xs:element ref="mayBindVars" minOccurs="0"
               maxOccurs="1"/>
            <xs:element ref="answerKBPattern" minOccurs="1"
               maxOccurs="1"/>
            <xs:element ref="answerSizeBound" minOccurs="0"
               maxOccurs="1"/>
        </xs:sequence>
128 </xs:complexType>


    <xs:complexType name="answerType">
        <xs:all>
133         <xs:element name="binding-set" minOccurs="0"
               maxOccurs="1">
                <xs:complexType>
                    <xs:sequence>
                        <xs:any namespace="http://www.w3.org
                           /2003/10/owl-ql-variables#"
                           minOccurs="0" maxOccurs="unbounded
                           "/>
                    </xs:sequence>
138             </xs:complexType>
            </xs:element>
            <xs:element name="answerPatternInstance"
               minOccurs="0" maxOccurs="1">
                <xs:complexType>
                    <xs:sequence>
143                     <xs:any processContents="skip"
                           namespace="http://www.w3.org
                           /1999/02/22-rdf-syntax-ns#"
                           minOccurs="1" maxOccurs="1"/>
                    </xs:sequence>
```

```
                </xs:complexType>
            </xs:element>
        </xs:all>
    </xs:complexType>

    <xs:complexType name="continuationType">
        <xs:choice>
            <xs:element name="termination-token" minOccurs="1
                " maxOccurs="1">
                <xs:complexType>
                    <xs:choice>
                        <xs:element name="none" minOccurs="1"
                            maxOccurs="1"/>
                        <xs:element name="end" minOccurs="1"
                            maxOccurs="1"/>
                    </xs:choice>
                </xs:complexType>
            </xs:element>
            <xs:element name="continuation-token" minOccurs="
                1" maxOccurs="1">
                <xs:complexType>
                    <xs:all>
                        <xs:element name="processHandle"
                            minOccurs="1" maxOccurs="1">
                            <xs:simpleType>
                                <xs:restriction base="
                                    xs:string"/>
                            </xs:simpleType>
                        </xs:element>
                        <xs:element name="answerBundleSize"
                            minOccurs="1" maxOccurs="1">
                            <xs:simpleType>
                                <xs:restriction base="
                                    xs:positiveInteger"/>
                            </xs:simpleType>
                        </xs:element>
                    </xs:all>
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="answerBundleType">
        <xs:sequence>
            <xs:element ref="queryPattern" minOccurs="0"
                maxOccurs="1"/>
```

```
            <xs:element ref="answer" minOccurs="0" maxOccurs=
                "unbounded"/>
            <xs:element ref="continuation" minOccurs="0"
                maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="proofType">
        <xs:sequence>
            <xs:element ref="proofStep" minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="proofStepType">
        <xs:sequence>
            <xs:any processContents="skip" namespace="http://
                www.w3.org/1999/02/22-rdf-syntax-ns#"
                minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
        <xs:anyAttribute processContents="skip" namespace="
            http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    </xs:complexType>


</xs:schema>
```

Quellcode B.1: OWL-QL Schema

B.1.2 Owl-QL Variables

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
    xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
    targetNamespace="http://www.w3.org/2003/10/owl-ql-
        variables#"
    xmlns="http://www.w3.org/2003/10/owl-ql-variables#"
    elementFormDefault="qualified">


    <xs:element name="x" type="var:variableType" />
    <xs:element name="y" type="var:variableType" />
    <xs:element name="z" type="var:variableType" />
```

```
    <xs:complexType name="variableType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:anyAttribute processContents="skip"
                    namespace="http://www.w3.org/1999/02/22-
                    rdf-syntax-ns#"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

</xs:schema>
```

Quellcode B.2: OWL-QL Variables Schema

B.1.3 Query

```
<?xml version="1.0"?>
<query
    xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
    xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:fam="file://famliy-1#">

<premise>
    <rdf:RDF>
        <rdf:Description
            rdf:about="file://family-1#kathrin">
            <rdf:type rdf:resource="file://family-1#woman" />
        </rdf:Description>
        <rdf:Description
            rdf:about="file://family-1#kathrin">
            <fam:has-child rdf:resource="file://family-1#
                petra" />
        </rdf:Description>
    </rdf:RDF>
</premise>

<queryPattern>
    <rdf:RDF>
        <rdf:Description
            rdf:about="http://www.w3.org/2003/10/owl-ql-
                variables#x">
            <rdf:type rdf:resource="file://family-1#woman" />
```

```
        </rdf:Description>
        <rdf:Description
            rdf:about="http://www.w3.org/2003/10/owl-ql-
               variables#x">
            <fam:has-child rdf:resource="http://www.w3.org
               /2003/10/owl-ql-variables#y" />
        </rdf:Description>
    </rdf:RDF>
</queryPattern>

<mustBindVars>
    <var:x/>
    <var:y/>
</mustBindVars>

<answerKBPattern>
    <kbRef rdf:resource="file://family-1#"/>
</answerKBPattern>

<answerSizeBound>
5
</answerSizeBound>

</query>
```

Quellcode B.3: OWL-QL Query

B.1.4 AnswerBundle

```
<?xml version="1.0"?>
<answerBundle xmlns="http://www.w3.org/2003/10/owl-ql-syntax#
   "
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns
           #"
        xmlns:fam="file://famliy-1#"
        xmlns:var="http://www.w3.org/2003/10/owl-ql-variables
           #">

<queryPattern>
    <rdf:RDF>
        <rdf:Description rdf:about="http://www.w3.org
           /2003/10/owl-ql-variables#x">
            <rdf:type rdf:resource="file://family-1#woman"/>
        </rdf:Description>
```

```
        <rdf:Description rdf:about="http://www.w3.org
           /2003/10/owl-ql-variables#x">
            <fam:has-child rdf:resource="http://www.w3.org
               /2003/10/owl-ql-variables#y"/>
        </rdf:Description>
    </rdf:RDF>
</queryPattern>

<answer>
    <binding-set>
        <var:x rdf:resource="file://family-1#betty"/>
        <var:y rdf:resource="file://family-1#eve"/>
    </binding-set>

    <answerPatternInstance>
        <rdf:RDF>
            <rdf:Description rdf:about="file://family-1#betty
               ">
                <rdf:type rdf:resource="file://family-1#woman
                   "/>
                <fam:has-child rdf:resource="file://family-1#
                   eve"/>
            </rdf:Description>
        </rdf:RDF>
    </answerPatternInstance>
</answer>
<answer>
    <binding-set>
        <var:x rdf:resource="file://family-1#betty"/>
        <var:y rdf:resource="file://family-1#doris"/>
    </binding-set>
    <answerPatternInstance>
        <rdf:RDF>
            <rdf:Description rdf:about="file://family-1#betty
               ">
                <rdf:type rdf:resource="file://family-1#woman
                   "/>
                <fam:has-child rdf:resource="file://family-1#
                   doris"/>
            </rdf:Description>
        </rdf:RDF>
    </answerPatternInstance>
</answer>

<continuation>
    <continuation-token>
```

```
        <processHandle>20228056#23965177</processHandle>
        <answerBundleSize>2</answerBundleSize>
    </continuation-token>
</continuation>

</answerBundle>
```

Quellcode B.4: OWL-QL AnswerBundle

B.2 Verwendete Beispiel-Knowledgebase

```
(in-knowledge-base family smith-family)

(signature
    :atomic-concepts (
        human person female male woman man
        parent mother father grandmother
        aunt uncle sister brother only-child)
    :roles (
        (has-descendant :transitive t)
        (has-child :parent has-descendant
            :domain parent
            :range person)
        (has-sibling :domain (or sister brother)
            :range (or sister brother))
        (has-sister :parent has-sibling
            :range (some has-gender female))
        (has-brother :parent has-sibling
            :range (some has-gender male))
        (has-gender :feature t)
    )
)

(implies person (and human (some has-gender (or female male))
   ))
(disjoint female male)
(implies woman (and person (some has-gender female)))
(implies man (and person (some has-gender male)))

(equivalent parent (and person (some has-child person)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))

(equivalent grandmother
```

```
    (and mother
    (some has-child
            (some has-child person))))

(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))

(equivalent brother (and man (some has-sibling person)))
(equivalent sister (and woman (some has-sibling person)))

(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)

(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)

(instance charles brother)
(related charles betty has-sibling)

(related doris eve has-sister)

(related eve doris has-sister)
```

Quellcode B.5: Knowledge Base Example

B.3 XMLBeans Ant-Task

```
<project name="de.tuhh.sts.racer.service" default="rebuild"
   basedir=".">
    <property environment="os"/>
    <property file="servicecontrol.properties"/>
    <property name="webinf.dir" value="WEB-INF"/>
    <property name="classes.dir" value="${webinf.dir}/classes
       "/>
    <property name="lib.dir" value="${webinf.dir}/lib"/>
    <property name="extlib.dir" value="lib"/>
    <property name="src.dir" value="${webinf.dir}/src"/>
    <property name="xbeangen.dir" value="xbeangen"/>
    <property name="schema.dir" value="schemas"/>

    <path id="project.class.path">
    <pathelement location="${classes.dir}"/>
```

```
    <fileset dir="${lib.dir}"/>
    <fileset dir="${extlib.dir}"/>
    </path>

    <!-- XMLBean build -->
    <taskdef name="xmlbeanbuild"
            classname="org.apache.xmlbeans.impl.tool.XMLBean"
               >
    <classpath>
            <path refid="project.class.path"/>
        </classpath>
    </taskdef>

    <target name="clean">
        <delete dir="${classes.dir}"/>
        <delete dir="${xbeangen.dir}" />
    </target>

    <target name="checkxbean">
        <uptodate property="noxbeanrebuild" targetfile="${
           xbeangen.dir}">
            <srcfiles dir="${schema.dir}" includes="*.wsdl,*.
               xsd"/>
        </uptodate>
    </target>

    <target name="xbean" depends="checkxbean" unless="
       noxbeanrebuild">
        <mkdir dir="${classes.dir}"/>
    <mkdir dir="${xbeangen.dir}"/>

    <xmlbeanbuild
        schema="${schema.dir}"
        srcgendir="${xbeangen.dir}"
        classgendir="${classes.dir}"
        failonerror="true">
            <classpath>
                <path refid="project.class.path"/>
            </classpath>
        </xmlbeanbuild>
    </target>

    <target name="rebuild" depends="clean,build"/>

    <target name="build" depends="xbean">
        <mkdir dir="${classes.dir}"/>
```

```
            <javac srcdir="${src.dir}"
        destdir="${classes.dir}"
        classpathref="project.class.path"
        debug="false"/>
    </target>
</project>
```

Quellcode B.6: XMLBeans Ant-Task

B.4 Java Quellcode

Dieser Abschnitt zeigt Auschnitte des Quellcodes wichtiger Klassen der Pakete Syntax und Service. Die detaillierte Beschreibung der Zusammenarbeit wurde in Kapitel 5 vorgenommen.

B.4.1 Die Klasse *OwlQlSimpleService*

```
package de.tuhh.sts.racer.service;

import java.rmi.RemoteException;

import org.apache.xmlbeans.XmlException;
import org.w3.x2003.x10.owlQlSyntax.AnswerBundle;
import org.w3.x2003.x10.owlQlSyntax.Query;

import de.tuhh.sts.racer.connection.ConnectionException;
import de.tuhh.sts.racer.connection.RacerOWLManager;
import de.tuhh.sts.racer.proxy.RacerProxyException;
import de.tuhh.sts.racer.syntax.owlql.OwlQlRequest;
import de.tuhh.sts.racer.syntax.owlql.OwlQlResponse;

public class OwlQlSimpleService {

    /*
     * This is the public Method of the Service,
     * Available to clients.
     * It receives a String serialized Query-Doc.
     * and returns a serialized AnswerBundle
     */
    public String getResult(String request) throws
        RemoteException {
        /*
         * get an Instance of the Manager
```

```
         * that realises proxy-communication, Dialog-Handling
         * and Session Management
         */
29       RacerOWLManager rm = RacerOWLManager.getInstance();

        // init XMLBean-Objects
        Query q = null;
        AnswerBundle a = null;
34
        try {
            // read input document to XMLBean ...
            q = Query.Factory.parse(request);
        } catch (XmlException e) {
39          // ... throw exception if doc is invalid to
                schema
            throw new RemoteException(e.getMessage(), e.
                getCause());
        }

        // create a request object from
44      OwlQlRequest owlReq = OwlQlRequest.Factory.
            newInstance(q);


        OwlQlResponse response;
        try {
49          // Get result from RacerOWLManager, cancel
                communication
            // if connection errors on Proxy Layer occur.
            response = rm.getResults(owlReq);
        } catch (ConnectionException e) {
            throw new RemoteException(e.getMessage()+rm.b.
                toString(), e.getCause());
54      } catch (RacerProxyException e) {
            throw new RemoteException(e.getMessage()+rm.b.
                toString(), e.getCause());
        }
        // receive the AnswerBundle-Bean from Response
        a = response.getAnswerBundle();
59
        // serialize AB to String and return it to the Client
        return a.toString();
    }
}
```

Quellcode B.7: Quellcode der Klasse service.OwlQLSimpleService

B.4.2 Die Klasse *Translator*

```
1  package de.tuhh.sts.racer.syntax.impl;
2
   import java.util.NoSuchElementException;
   import org.apache.xmlbeans.impl.values.XmlComplexContentImpl;

   // import generated XMLBeans for OWL-QL Schema
7  import org.w3.x2003.x10.owlQlSyntax.*;

   import org.w3c.dom.Node;
   import org.w3c.dom.NodeList;

12 // import Jena Framework Classes
   import com.hp.hpl.jena.rdf.arp.*;
   import com.hp.hpl.jena.rdf.model.*;

   // import NrqlSyntax Classes
17 import de.tuhh.sts.racer.syntax.Translator;
   import de.tuhh.sts.racer.syntax.nrql.*;
   import de.tuhh.sts.racer.syntax.owlql.OwlQlRequest;
   import de.tuhh.sts.racer.syntax.owlql.OwlQlResponse;

22 public class TranslatorImpl implements Translator {

       // the nrql Query (command (premise) head body)
       private NrqlQuery nrqlQuery;

27     // OWL-QL Query XMLBean, Request and Response
       // for temporary storage during parsing process
       private Query queryXMLBean;
       private OwlQlResponse owlQlResponse;
       private OwlQlRequest owlQlRequest;
32
       // OWL-QL AnswerBundle XMLBean
       private AnswerBundle answerBundle;

       /*
37      * creates nRQL-String from OWL-QL
        */
       public NrqlQuery getNrqlQuery(Query queryXMLBean)
               throws MalformedURIException,
                  NoSuchElementException {
           this.queryXMLBean = queryXMLBean;
42
```

```
        // generates RetrieveQuery or RetrieveUnderPremise
           Query,
        // depending on OWL-QL-Doc.
        nrqlQuery = NrqlQuery.Factory.newInstance(
           queryXMLBean);

47      // parses premise (see below)
        if (queryXMLBean.getQuery().isSetPremise()) {
            parsePremise(queryXMLBean.getQuery().getPremise()
               );
        }

52      // parsing of QueryPattern (Body) and MustBindVars (
           Head)
        parseQueryPattern(queryXMLBean.getQuery().
           getQueryPattern());
        parseMustBindVars(queryXMLBean.getQuery().
           getMustBindVars());

        // return nRQL-Query Object, nRQL String generated by
            "toString"
57      return nrqlQuery;
    }

    private void parseQueryPattern(QueryPatternType
       queryPattern)
            throws MalformedURIException,
               NoSuchElementException {
62
        // read RDF contents of QueryPattern
        StmtIterator iter = getStatements((
           XmlComplexContentImpl) queryPattern);

        // iterate through Statements
67      while (iter.hasNext()) {
            AtomList stmt = new AtomList(iter.nextStatement()
               , Axiom.T_BODY);

            // create QueryAxiom and add to nRQL-Query-Body
            Axiom axiom = Axiom.Factory.newInstance(stmt);
72          nrqlQuery.getBody().addAxiom(axiom);
        }
    }

    // same as above, but this time Premise-Assertions are
       generated
```

```
77      private void parsePremise(PremiseType premise)
                throws MalformedURIException,
                    NoSuchElementException {

            StmtIterator iter = getStatements((
                XmlComplexContentImpl) premise);
            while (iter.hasNext()) {
82              AtomList stmt = new AtomList(iter.nextStatement()
                    , Axiom.T_PREMISE);
                Assertion ass = Assertion.Factory.newInstance(
                    stmt);
                nrqlQuery.getPremise().addAssertion(ass);
            }
        }
87
        /**
         * reads the MustBind Part of the Owl-Ql Document
         * and extracts the variables
         */
92      private void parseMustBindVars(MustBindVarsType mustBinds
            )
                throws MalformedURIException {
            // get the nodes of mustbind xml as w3 node-list
            NodeList nodeList = mustBinds.getDomNode().
                getChildNodes();

97          // iterate through nodelist and search for "type 1"
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node n = nodeList.item(i);

                if (n.getNodeType() == 1) {
102
                    // get the QueryHead from Query
                    nrqlQuery.getHead().addMustBindVariable(
                            // and add a VariableAtom Instance of
                                Node n
                            new VariableAtom(new URI(n.
                                getNamespaceURI()
107                                 + n.getLocalName())));
                }
            }
        }

112     /**
         * creates a Jena-Model from an XMLBeans Object
         */
```

```
    private Model readXML(XmlComplexContentImpl xml) {
        Model model = ModelFactory.createDefaultModel();
117     model.read(xml.newInputStream(), "");

        return model;
    }

122 /**
     * Delivers an Iterator over the RDF-Statements of a
        XMLBeans Object
     */
    private StmtIterator getStatements(XmlComplexContentImpl
        xml) {
        return readXML(xml).listStatements();
127 }

    // create the AnswerBundle XMLBean
    public AnswerBundle getAnswerBundle(OwlQlResponse
        response) {
        // temp. storage of Request and Response (ease of use
            )
132     owlQlResponse = response;
        owlQlRequest = response.getOwlQlRequest();

        // create new AB with options
        // options contain Namespace Settings
137     answerBundle  = AnswerBundle.Factory.newInstance(
            response.getOptions());

        // add empty container
        answerBundle.addNewAnswerBundle();

142     // copy QueryPattern from OWL-QL Query
        answerBundle.getAnswerBundle().setQueryPattern(
                response.getOwlQlRequest().getQueryPattern())
                    ;

        // reads nRQL Results and adds the Answers (see below
            )
147     answerBundle.getAnswerBundle().setAnswerArray(
            createAnswerArray());

        // generate empty continuation container
        answerBundle.getAnswerBundle().addNewContinuation();
```

```
152         // Server Continuation via Process Handle, if
                appliable
            if (response.isSetProcessHandle()) {
                answerBundle.getAnswerBundle().getContinuation()
                        .addNewContinuationToken();

157             answerBundle.getAnswerBundle().getContinuation()
                        .getContinuationToken().setProcessHandle(
                            response.getProcessHandle());

                answerBundle.getAnswerBundle().getContinuation()
                        .getContinuationToken().
                            setAnswerBundleSize(
162                             new BigInteger(new Integer(
                                    answerBundle
                                        .getAnswerBundle().
                                            sizeOfAnswerArray())
                                        .toString()));
            }

167

            // create Termination Token if no more Answers are
                possible
            if (response.isTerminated()) {
                answerBundle.getAnswerBundle().getContinuation()
                        .addNewTerminationToken();
172             answerBundle.getAnswerBundle().getContinuation()
                        .getTerminationToken().addNewEnd();
            }

            return answerBundle;
177     }

        /*
         * This function iterates through nRQL Results and
         * translates them to OWL-QL Binding set and
            AnswerPatternInstance
182      */
        private AnswerType[] createAnswerArray() {
            // read results
            ArrayList nrqlResults = owlQlResponse.getNrqlResults
                ();

187         // create new typed Array
            AnswerType[] a = new AnswerType[nrqlResults.size()];
```

```
        // iterate through results
        for (int i = 0; i < a.length; i++) {
192         NrqlResult res = (NrqlResult) nrqlResults.get(i);

            // new Answer - Container (options: see above)
            AnswerType abt = AnswerType.Factory.newInstance(
                owlQlResponse.getOptions());
            abt.addNewBindingSet();
197         abt.addNewAnswerPatternInstance();

            // gets the QueryBody of the nRQL-Query
            // and substitutes vars with bindings
            try {
202             abt.getAnswerPatternInstance().set(
                        owlQlRequest.getNrqlQuery().getBody()
                                .getAnswerPatternInstance(res
                                    ,
                                        owlQlRequest.
                                            getNameSpaces()));

207         } catch (RacerSyntaxException e) {
                logger.error(e.getMessage());
            }

            // create bindings between vars and individuals
212         XmlObject xml = res.toXML(owlQlRequest.
                getNameSpaces());
            abt.getBindingSet().set(xml);
        }

        return a;
217     }
}
```

Quellcode B.8: Quellcode der Klasse syntax.Translator

Anhang C

Inhaltsverzeichnis der Daten CD

C.1 Dokumente

1. Dieses Dokument als PDF-Datei (Druck und Internet)
2. Komplette Ausführung als Latex-Projekt
3. Die Zusammenfassung als ASCII-Text Datei

C.2 Software

1. Komplettes System als Web-Archiv
2. Quellcode
3. JavaDoc Dokumentation
4. Verwendete Bibliotheken, Frameworks und Werkzeuge

C.3 Referenzen

1. Gespiegelte Internetseiten als HTML-Dateien

Glossar

ABox	Instanzen, Prädikaten Logik-Formeln deren Prädikate in der TBox definiert sind
Axis	***A**pache e**X**tensible **I**nteraction **S**ystem*, siehe [31]
DAML	***D**ARPA **A**gent **M**arkup **L**anguage* [1], `http://www.daml.org/`
DIG	***D**escription Logic **I**mplementation **G**roup* `http://dl.kr.org/dig/`
FOL	First Order Predicate Logic
Inferenz	Schlussfolgerung; logisches Schließen von Informationen
JRacer	Java API für TCP Zugriff auf Server/Proxy
KB	Knowledge Base, dt. Wissenbasis; Tupel (T, A) aus TBox T und ABox A
KIF	***K**nowledge **I**nterchange **F**ormat* - Lisp-artige Standard-Syntax für Prädikatenlogik.
KRSS	***K**nowledge **R**epresentation **S**pecification **S**ystem* - Standard-Syntax für Description Logic
nRQL	**N**ew **R**ACER **Q**uery **L**anguage
Ontologie	Explizite Spezifikation aller in einer Domäne verwendeten Konzepte
OWL	*Web-Ontology Language*
OWL-QL	*Web-Ontology Query Language*

[1] DARPA: Defense Advanced Research Projects Agency

OWL-S OWL-S (vorm. DAML-S) ist eine OWL basierte Web Service Ontologie zur semantischen Beschreibung von Diensten zur automatisierten Aufindung, Anwendung, Komposition und Interaktion.

Query *(RACER)* ▶Siehe **Statement** *(RACER)* [2]

RACER Akronym für ***R**enamed **A**Box and **C**oncept **E**xpression **R**easoner*, siehe [34]

RDF ***R**esource **D**escription **F**ramework*, Spezifikation siehe [23]

Relationen Beziehungen zwischen den Begriffen

RICE ***R**acer **I**nteractive **C**lient **E**nvironment* - Grafischer Client für Zugriff auf RACER

SOA ***S**ervice **O**riented **A**rchitecture*

SOAP Ursprünglich als Abkürzung für ***S**imple **O**bject **A**ccess **P**rotocol* verwendet. Mit Erscheinen der SOAP Spezifikation 1.2 wurde die Verwendung als eigenständiger Begriff verankert.

Statement *(RACER)* Im Zusammenhang mit Zugriffen auf den RACER Server wird zwischen Queries und Statements unterschieden.
Dabei bezeichnen Queries nicht verändernde Abfragen [3] und Statements, verändernde Zugriffe, bei denen sich der Zustand der ABox oder der TBox nach dem Zugriff von dem vor dem Zugriff unterscheidet[4].

Statement *(RDF)* Ein RDF-Statement ist ein Satz, bestehend aus Subjekt, Prädikat und Objekt. Dabei ist das Subjekt stets ein Individuum, das Prädikat entweder eine Rollen-Bezeichnung oder ein Literal und das Objekt ein anderes Individuum(im Falle einer Rollen-Beziehung) oder ein Konzept.

Taxonomie Systematik; Gliederung von Begriffen in hierarchische Klassen

[2] Keine Veränderungen an der KB
[3] analog SELECT bei SQL
[4] Verändernder Eingriff in die KB

TBox	Ontologien, Konzepte, **Taxonomie** strukturierter Terme, Symboltabelle enthält die Namen der T-Box Terme
UDDI	***U**niversal **D**efinition, **D**iscovery and **I**ntegration* - Standard für die Registrierung von Web Services
URI	***U**niform **R**esource **I**dentifier* `http://www.w3.org/2001/12/URI/`
URL	***U**niform **R**esource **L**ocator* `http://www.w3.org/Addressing/URL/`
W3C	***W**orld **W**ide **W**eb **C**onsortium*, Internationales Konsortium zur Entwicklung von Web-Standards; [35]
WSDL	***W**eb**S**ervice **D**escription **L**anguage*, ein XML Dokument, dass die Schnittstellen eines WebService beschreibt.

Literaturverzeichnis

[1] D. WANG, T. BAYER, T. FROTSCHER UND M. TEUFEL; *Java Web Services mit Apache Axis*; Software & Support Verlag, Frankfurt; 2004;

[2] T. LANGNER; *Web Services mit Java*; Markt&Technik Verlag; München; 2003.

[3] I. HORROCKS UND S. TESSARIS; *Querying the semantic web: a formal approach*; Proceedings of the 13th International Semantic Web Conference; ISWC (Ian Horrocks and J. Hendler, eds.); Lecture Notes in Computer Science; no. 2342; 2002.

[4] R. FIKES, P. HAYES UND I. HORROCKS; *OWL-QL - A Language for Deductive Query Answering on the Semantic Web*; Stanford University; Stanford, California; 2003.

[5] V. HAARSLEV, R. MÖLLER UND M. WESSEL; *RACER User's Guide and Reference Manual*; Version 1.7.19; Software Systems; TU Harburg; April 2004.

[6] B. GLIMM UND I. HORROCKS; *Query Answering Systems in the Semantic Web*; University of Manchester; 2004.

[7] T. DOSHI; *Enhancement of the RacerProxy webservices Interface for Iterative Query Answering*; Arbeitsbereich STS; TU Hamburg-Harburg; 2005.

[8] J.R. MAHARJAN; *Reengineering of the RACER Proxy*; Arbeitsbereich STS; TU Hamburg-Harburg; 2005.

[9] M. ACHATZ UND T. LANGHAMMER; *Einführung in Description Logics*; Universität Passau, Lehrstuhl für Informationsmanagement; 2003.

[10] B. GLIMM; *A Query Language for Web Ontologies*; Bachelorarbeit; Hochschule für Angewandte Wissenschaften Hamburg; Juni 2004.

[11] J.J. CARROLL, I. DICKINSON, ET AL.; *Jena: Implementing the Semantic Web Recommendations*; HP Technical Report HPL-2003-146; Dezember 2003.
http://www.hpl.hp.com/techreports/2003/HPL-2003-146.html

[12] R. CORNET; *RICE-Manual*; Department of Medical Informatics; University of Amsterdam; April 2004.

[13] C. FINCKLER, R. MÖLLER; *RACER-Proxy-Dokumentation*; Fachhochschule Wedel, Hamburg; Dezember 2000.

[14] H. SETH; *XML Beans: The Best of Both Worlds*; XML-Journal; Syscon-Media; 2003. http://dev2dev.bea.com/technologies/xmlbeans/articles/hitesh_seth.jsp

[15] S. RINKE; *WebServices mit Java, Axis, XDoclet und Eclipse*; http://www.stefan-rinke.de/articles/webservices/

[16] V. HAARSLEV, R. MÖLLER AND M. WESSEL; *Querying the Semantic Web with Racer + nRQL*; Proceedings of the KI-2004 International Workshop on Applications of Description Logics (ADL'04); Ulm, Germany; September 2004.

[17] S. POWERS; *Practical RDF*; O'Reilly Verlag, Sebastopol; 2003.

[18] T. GRUBER; *What is an Ontology?*; University of Stanford; 1993. http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[19] V. HAARSLEV, R. MÖLLER, R. VAN DER STRAETEN UND M. WESSEL; *Extended Query Facilities for RACER and an Application to Software-Engineering Problems*; Proceedings of the 2004 International Workshop on Description Logics (DL-2004); Whistler, BC, Canada; Juni 2004.

[20] T. BERNERS-LEE; *Semantic Web - XML2000*; Präsentation; 2000. http://www.w3.org/2000/Talks/1206-xml2k-tbl/

[21] M. FERDINAND; *Ein generativer Ansatz zur semantischen Beschreibung von Geschäftsdokumenten*; Diplomarbeit; Fachbereich Informatik, Universität Hamburg; April 2003.

[22] H. HAAS UND A. BROWN; *Web Services Glossary*; W3C Working Group; February 2004. http://www.w3.org/TR/ws-gloss/;

[23] E. MILLER, R. SWICK UND D. BRICKLEY; *Resource Description Framework Specification*; http://www.w3.org/RDF/;

[24] D. MCGUINNESS UND F. VAN HARMELEN; *OWL Web Ontology Language - Overview*; W3C Recommendation; February 2004. http://www.w3.org/TR/owl-features/;

[25] G. ANTONIOU UND F. VAN HARMELEN; *Web Ontology Language: OWL*; Handbook on Ontologies in Information Systems"; Springer-Verlag; 2003.

[26] J.-H. DEUTSCHER UND T.FROESE; *Erweiterung des Racer-Proxys um eine SOAP-Schnittstelle*; Seminararbeit, AB STS, TU Hamburg-Harburg; September 2003.

[27] T. BERNERS-LEE, J. HENDLER UND O. LASSILA; *The Semantic Web*; Scientific American; May 2001.

[28] H. SACK UND U. KRÜGER; *Semantic Web - WWW 2.0*; Präsentation, Institut für Informatik Friedrich Schiller-Universität Jena; 2003.

[29] G. KLYNE, J.J. CARROLL UND B. MCBRIDE; *Resource Description Framework: Concepts and Abstract Syntax*; W3C Recommendation; February 2004.
http://www.w3.org/TR/rdf-concepts/

[30] R. MOLITOR; *Unterstützung der Modellierung verfahrenstechnischer Prozesse durch Nicht-Standardinferenzen in Beschreibungslogiken*; Dissertation; Fakultät für Mathematik, Informatik und Naturwissenschaften der RTWH Aachen; Aachen; Dezember 2000;

Internet-Quellen

[31] APACHE FOUNDATION; *Homepage des „Apache Axis" Projekts*;
http://ws.apache.org/axis

[32] HP LABS; *Homepage des „Jena Semantic Web"-Frameworks*;
http://jena.sourceforge.net/

[33] APACHE FOUNDATION; *Homepage des „Apache XMLBeans" Projekts*;
http://xmlbeans.apache.org/

[34] R. MÖLLER UND V. HAARSLEV; *RACER-System Homepage*;
http://www.sts.tu-harburg.de/~r.f.moeller/racer/
http://www.racer-systems.de/

[35] W3C; *Homepage des World Wide Web Consortiums (W3C)*;
http://www.w3.org/TR/ws-gloss/;

[36] M. JECKLE; *Web Services*;
http://www.jeckle.de/webServices/

[37] HP LABS; *Homepage der „HP Labs Semantic Web Research" Gruppe*;
http://hpl.hp.com/semweb/

[38] R. FIKES, R. MCCOOL UND D. MCGUINNESS; *OWL-QL Homepage*; Knowledge Systems Laboratory; Stanford University, California;
http://ksl.stanford.edu/projects/owl-ql/

[39] E. MILLER, R. SWICK ET AL.; *W3C: Semantic Web Activity*;
http://www.w3.org/2001/sw/

[40] STANFORD KNOWLEDGE SYSTEMS LABORATORY; *OWL-QL Syntax Schema*;
http://ksl.stanford.edu/projects/owl-ql/owl-ql.xsd

[41] STANFORD KNOWLEDGE SYSTEMS LABORATORY; *OWL-QL Variables Schema*
http://ksl.stanford.edu/projects/owl-ql/owl-qlvar.xsd

[42] STANFORD KNOWLEDGE SYSTEMS LABORATORY; *Java-based Reasoning Agents for the Stanford KSL*; Presentations and Demos;
http://onto.stanford.edu:8080/

[43] B. GLIMM UND I. HORROCKS; *Manchester DQL Service*;
Präsentation: http://www.kw.man.ac.uk/doc/DQL-040928.ppt
Download: http://www.cs.man.ac.uk/~glimmbx/