Mark Schneider

SPARQLAS

Implementation von SPARQL Anfragen in OWL Syntax

Betreuer: Fernando Silva Parreiras

19. Mai 2010

Erklärung

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

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den 19. Mai 2010

Mark Schneider

Abstract. SPARQL can be employed to query RDF documents using RDF triples. OWL-DL ontologies are a subset of RDF and they are created by using specific OWL-DL expressions. Querying such ontologies using only RDF triples can be complicated and can produce a preventable source of error depending on each query. SPARQL-DL Abstract Syntax (SPARQLAS) solves this problem using OWL Functional-Style Syntax or a syntax similar to the Manchester Syntax for setting up queries. SPARQLAS is a proper subset of SPARQL and uses only the essential constructs to obtain the desired results to queries on OWL-DL ontologies implying least possible effort in writing. Due to the decrease in size of the query and having a familiar syntax the user is able to rely on, complex and nested queries on OWL-DL ontologies can be more easily realized. The Eclipse plugin EMFText is utilized for generating the specific SPARQLAS syntax. For further implementation of SPARQLAS, an ATL transformation to SPARQL is included as well. This transformation saves developing a program to directly process SPARQLAS queries and supports embedding SPARQLAS into running development environments.

Kurzdarstellung. Mittels SPARQL können Anfragen in Form von RDF Tripeln auf RDF Dokumente gestellt werden. OWL-DL Ontologien sind eine Teilmenge von RDF und können über spezifische OWL-DL Ausdrücke erstellt. Solche Ontologien über RDF Tripel anzufragen kann je nach Anfrage kompliziert werden und eine vermeidbare Fehlerquelle darstellen. Die SPARQL-DL Abstract Syntax (SPARQLAS) löst dieses Problem indem Anfragen mittels OWL Functional-Style Syntax oder einer der Manchester Syntax ähnlichen Syntax gestellt werden. SPARQLAS ist eine echte Teilmenge von SPARQL und verwendet nur die nötigsten Konstrukte, um mit möglichst wenig Schreibaufwand schnell die gewünschten Ergebnisse zu Anfragen auf OWL-DL Ontologien zu erhalten. Durch die Verringerung des Umfangs einer Anfrage und der Verwendung einer dem Nutzer bekannten Syntax lassen sich komplexe und verschachtelte Anfragen auf OWL-DL Ontologien einfacher realisieren. Zur Erstellung der spezifischen SPARQLAS Syntax wird das Eclipse Plugin EMFText verwendet. Die Implementation von SPARQLAS beinhaltet zudem noch eine ATL Transformation zu SPARQL. Diese Transformation erspart die Entwicklung eines Programms zur direkten SPARQLAS Verarbeitung und erleichtert so die Integration von SPARQLAS in bereits laufende Entwicklungsumgebungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AMMA	ATLAS Model Management Architecture
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ATL	ATLAS Transformation Language
EBNF	Extended Backus Naur Form
EMF	Eclipse Modeling Framework
HTTP	Hypertext Transfer Protocol
HUTN	Human-Usable Textual Notation
IDE	Integrated Development Environment
INRIA	Institut National de Recherche en Informatique et en Automatique
IRI	Internationalized Resource Identifier
LINA	Laboratoire d'Informatique de Nantes Atlantique
MOST	Marrying Ontology and Software Technology
NeOn	NeOn: Lifecycle Support for Networked Ontologies
OMG	Object Management Group
OMG MOF	OMG Meta Object Facility
OMG MOF/QVT RFP	OMG MOF Query/Views/Transformations Request For Proposal
OWL	Web Ontology Language
OWL-DL	OWL - Description Logic
PHP	PHP: Hypertext Preprocessor
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
SPARQL-DL	SPARQL - Description Logic
SPARQLAS	SPARQL-DL Abstract Syntax
TwoUse	Transforming and Weaving Ontologies and UML in Software Engineering
URI	Uniform Resource Identifier

W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

1

Einleitung

SPARQL ist eine vom W3C empfohlene Anfragesprache für RDF, welche verschieden komplexe Anfragen an Ontologien unterstützt. Mit SPARQL lassen sich somit Daten erfragen, die in Ontologien enthalten sind. Eine solche Ontologie kann mittels der Web Ontology Language (OWL) ausgedrückt werden und wird auch so vom W3C empfohlen. OWL besitzt verschiedene Ausprägungen, die sich nach Ausdrucksfähigkeit unterscheiden. Eine dieser Ausprägungen nennt sich OWL-DL und folglich können sich SPARQL Anfragen auch auf OWL-DL Ontologien beziehen. Um OWL-DL Ontologien zu erstellen, können verschiedene Syntaxen verwendet werden, u.a. die vom W3C empfohlene OWL Functional-Style Syntax. SPARQL jedoch unterstützt nur RDF als Syntax, so dass mit OWL Functional-Style Syntax erzeugte OWL-DL Ontologien über gewisse RDF Tripel angefragt werden müssen. Diese vom Benutzer zusätzlich zu bewerkstelligende Übersetzung zwischen diesen beiden Syntaxen kann je nach Komplexität der Anfrage nicht nur zeitaufwendig, sondern auch äußerst fehleranfällig sein.

Eine Lösung zu diesem Problem bietet die hier vorgestellte SPARQL-DL Abstract Syntax (SPARQLAS). SPARQLAS bietet die Möglichkeit, SPARQL Anfragen auf OWL-DL Ontologien in einer einfachen und übersichtlichen Art und Weise zu stellen. Dies wird dadurch erreicht, dass SPARQLAS die OWL Functional-Style Syntax unterstützt, was dem Nutzer zum einen die Übersetzung zwischen zwei Syntaxen erspart und zum anderen das Erstellen von Anfragen durch eine ihm vertraute Syntax erleichtert. Die übrigen Rahmenangaben einer SPARQL Anfrage werden von SPARQLAS auf das Nötigste reduziert, was die Übersichtlichkeit einer Anfrage erhöht. Außerdem wird eine weitere, der Manchester Syntax ähnlichen Syntax unterstützt, so dass Anfragen noch kürzer gestaltet werden können.

SPARQLAS beruht auf den theoretischen Konzepten von SPARQL-DL, welche jedoch um noch nicht beschriebene Axiome, die mit OWL-DL ausdrückbar sind, erweitert werden. Die Schwierigkeit in dieser Umsetzung liegt somit darin, dass die komplette Sprache zu SPARQLAS von Grund auf erstellt werden muss. Dies beinhaltet die Erarbeitung eines Metamodells zu

SPARQLAS und die Generierung einer textuellen Syntax mittels EMFText. Zudem wird eine ATL Transformation von SPARQLAS zu SPARQL entwickelt. Diese Transformation ist deshalb notwendig, da zu SPARQL genügend Programme existieren, die solche Anfragen verarbeiten können. SPARQLAS ist jedoch eine neue Sprache ohne bereits existierenden Programmen zur Anfragenverarbeitung. Da SPARQL und SPARQLAS artverwandt sind, ist es daher sinnvoll, eine Transformation durchzuführen anstatt ein Programm zur speziellen SPARQLAS Verarbeitung zu entwerfen. Außerdem wird so die Integration von SPARQLAS in laufende Umgebungen erleichtert. In den folgenden Kapiteln wird der Entwicklungsprozess zu SPARQLAS wiedergegeben.

In Kapitel 2 wird zuerst auf die Grundlagen eingegangen, was sind SPARQL und SPARQLAS genau und wo liegen die Unterschiede. Danach wird EMFText erläutert, womit die textuelle Syntax zu SPARQL und SPARQLAS realisiert wird. Abgeschlossen wird dieses Kapitel durch die Vorstellung von ATL, was die Transformation von SPARQLAS zu SPARQL umsetzt. Kapitel 3 befasst sich mit der geleisteten Arbeit, bevor die textuelle Syntax und die Transformation angegangen werden können. Es werden hier die einzelnen Metamodelle zu SPARQL und SPARQLAS vorgestellt, die beide für die jeweilige Syntax und für die Transformation unabdingbar sind. Die Hauptarbeit ist Bestandteil von Kapitel 4 und verdeutlicht die Entwicklung der textuellen Syntax von SPARQLAS und die Transformation zwischen SPARQLAS und SPARQL. Zusätzlich werden die verschiedenen Anwendungsmöglichkeiten dargelegt, also als Eclipse Plugin, als Standalone Version oder API und als Webservice. Abgerundet wird dieses Kapitel durch eine Evaluation bzgl. SPARQL und SPARQLAS Anfragen, wonach im letzten Kapitel 5 eine Zusammenfassung mit Ausblick und Danksagung geliefert wird.

Zuvor soll noch kurz die während dieser Studienarbeit verwendete Beispielanfrage erläutert werden. Als beispielhafte Ontologie gilt die hinlänglich bekannte Pizza Ontologie [DHS+07] und es sollen alle Pizzen erfragt werden, die einen Käsebelag als Belag besitzen.

2

Grundlagen

2.1 SPARQL und SPARQLAS

2.1.1 Erläuterung von SPARQL

SPARQL Protocol and RDF Query Language (SPARQL)[1] ist wie schon erwähnt eine Anfragesprache für RDF[2] und hat am 15. Januar 2008 die W3C[3] Recommendation erhalten. Momentan wird an der Version 1.1 gearbeitet, so dass im Folgenden die Version mit der W3C Empfehlung verwendet wird. SPARQL an sich ist in drei Kategorien unterteilt. Zum einen besteht es aus dem SPARQL Protocol for RDF [CFT08], was das Zusammenspiel von SPARQL Anfragen zwischen Query Clients und Query Processors regelt, einerseits im abstrakten Sinne und andererseits auch konkret anhand von HTTP[4] und SOAP Bindings. Zum zweiten beinhaltet es die SPARQL Query Language for RDF [PS08], welches der eigentlichen Anfragesprache entspricht und die Syntax der einzelnen Konstrukte definiert. Zuletzt besteht SPARQL noch aus dem SPARQL Query Results XML Format [BB08], welches XML[5] als Query Result Form für Ergebnisse von ausgeführten SPARQL Anfragen vorgibt. Da für die Studienarbeit nur der zweite Teil zur Anfragesprache relevant ist, wird diese im Folgenden näher erläutert. Zur Erläuterung der anderen beiden Kategorien sei daher an dieser Stelle auf die angegebene Literatur hingewiesen.

Im Dokument zur SPARQL Query Language for RDF wird aufgezeigt, wie Anfragen aufgebaut sind und wie man konkrete Anfragen stellen kann. Eine SPARQL Anfrage lässt sich in zwei Teile unterteilen, zum einen in den Prolog und zum anderen in die Anfrage selbst.

---

[1] http://www.w3.org/2009/sparql/wiki/Main_Page
[2] Resource Description Framework, http://www.w3.org/RDF/
[3] World Wide Web Consortium
[4] Hypertext Transfer Protocol
[5] eXtensible Markup Language

Eigentlich ist der Prolog für eine SPARQL Anfrage nicht unbedingt notwendig, er vereinfacht es allerdings SPARQL Anfragen zu stellen. Im Prolog lassen sich nämlich so genannte Präfixe definieren, um Internationalized Resource Identifiers (IRIs) [DS05] abzukürzen. Denn die grundlegenden Daten, welche man mit SPARQL abfragen möchte, liegen als RDF Graph vor und die einzelnen Elemente müssen für SPARQL referenzierbar sein. Dies geschieht in SPARQL über absolute oder relative IRIs. Letztere bestehen aus einem Präfix und dem eigentlichen Element, wobei der entsprechende Präfix im Prolog definiert werden muss. Wird eine bestimmte grundlegende IRI besonders häufig in der Anfrage verwendet, so lässt sich im Prolog auch eine Basis IRI definieren, um auf Präfixe zu verzichten. Zwar sind in der eigentlichen Anfrage auch absolute IRIs zulässig, doch sind diese oftmals lang und so erspart man sich durch die Definitionen im Prolog etwas Schreibarbeit und erhöht zugleich die Übersichtlichkeit der Anfrage.

SPARQL Anfragen gibt es in vier verschiedenen Formen. Zum einen gibt es die SELECT Anfrage, welche zu gegebenen Variablen mögliche Zuordnungsmöglichkeiten ausgibt. Zum anderen liefert eine ASK Anfrage einen booleschen Wert zurück, welcher angibt, ob die Anfrage einem Teil des RDF Graphen entspricht oder auch nicht. Andererseits lässt sich mittels einer CONSTRUCT Anfrage mit Hilfe von vorhandenen RDF Graphen ein neuer, über die Anfrage spezifizierter RDF Graph erstellen. Zuletzt gibt es noch die DESCRIBE Anfrage, welche einen RDF Graphen zurück liefert, der ein durch die Anfrage gegebenes Element näher beschreibt. Wie man sieht, lassen sich mit SPARQL verschiedene Arten an Anfragen realisieren. Für die Erläuterung des Zusammenhangs zu SPARQLAS sind aber nur die SELECT und ASK Anfrage bedeutend, weshalb für nähere Informationen zur CONSTRUCT und DESCRIBE Anfrage auf die weitergehende Literatur zu verweisen ist.

Eine SELECT Anfrage wird durch den Term „SELECT“ eingeleitet, worauf die einzelnen zu erfragenden Variablen folgen. Alternativ lässt sich durch ein Sternchen (*) kennzeichnen, dass alle in der Anfrage verwendeten Variablen erfragt werden sollen. Danach kann man über „FROM“ oder „FROM NAMED“ den grundlegenden Datensatz spezifizieren. Dies ist jedoch in der Anfrage nicht notwendig, da der Datensatz auch im SPARQL Protocol for RDF angegeben werden kann. Darauf folgt nun der Hauptteil der Anfrage, in der die Bedingungen zur Anfrage definiert werden. Dieser Teil kann durch den Term „WHERE“ eingeleitet werden, muss aber nicht, da dieser Teil der Anfrage in geschweiften Klammern eingegrenzt wird. Die Bedingungen werden in einer für RDF typischen Tripel-Form angegeben und es besteht die Möglichkeit, optionale oder alternative Bedingungen zu formulieren. Weiter ist es auch möglich, die Anfrage auf bestimmte Graphen zu beschränken und verschiedene Filtervariationen zu verwenden. Je nach Anfrage kann es vorkommen, dass mehrere gleiche Resultate zurück geliefert werden. Dies lässt sich durch die Angabe der Terme „DISTINCT“ oder „REDUCED“ nach dem „SELECT“ Term verhindern. Außerdem kann die Ergebnismenge nach gewissen Kriterien geordnet oder auf eine bestimmte Anzahl beschränkt werden.

Eine ASK Anfrage ist ähnlich der SELECT Anfrage aufgebaut und wird analog dazu mit dem Term „ASK“ eingeleitet. Die FROM bzw. FROM NAMED und WHERE Teile gestalten sich genau wie bei einer SELECT Anfrage. Die einzigen Unterschiede im Aufbau liegen in der verschiedenen Art der Anfrage. Da eine ASK Anfrage überprüft, ob die Anfrage einem Teil des RDF Graphen entspricht, werden keine Variablen angegeben. Auch kann die Ergebnismenge weder geordnet noch beschränkt werden, da nur ein boolescher Wert zurück geliefert wird.

Somit lässt sich die Beispielanfrage in SPARQL wie folgt ausdrücken:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.co-ode.org/ontologies/pizza/pizza.owl#>

SELECT DISTINCT ?x
WHERE {
 ?x rdfs:subClassOf [
  rdf:type owl:Restriction ;
  owl:onProperty :hasTopping ;
  owl:someValuesFrom :CheeseTopping
  ]
 }
```

Diese Aspekte der SPARQL Query Language for RDF sollen genügen, um die Beziehung zu SPARQLAS zu verdeutlichen. Daher wird nun auf SPARQLAS näher eingegangen.

2.1.2 Erläuterung von SPARQLAS

Die SPARQL-DL Abstract Syntax (SPARQLAS) ist eine für OWL-DL[6] [SWM04] konzipierte Anfragesprache, welche auf den theoretischen Grundlagen von SPARQL-DL [SP07] aufbaut. Wie der Name es schon vermuten lässt, ist SPARQLAS eine Teilmenge der SPARQL Query Language for RDF, in welcher SPARQL-DL Konstrukte durch eine abstrakte Syntax implementiert sind. Wenn man mit SPARQL Anfragen stellen möchte, die OWL-DL Ausdrücke umsetzen, wird man feststellen, dass diese recht komplex ausfallen. Dies liegt daran, dass SPARQL Anfragen in RDF Tripeln gestellt werden und zu OWL-DL Ausrücken äquivalente RDF Tripel meist länger ausfallen. Daher verwendet SPARQLAS OWL-DL Konstrukte, um Anfragen kürzer, einfacher und übersichtlicher zu gestalten. Dabei ist die Syntax so gewählt, dass sich diese eindeutig auf RDF Tripel abbilden lässt. Das ist für eine Transformation

[6] Web Ontology Language - Description Logic

von SPARQLAS zu SPARQL essentiell, worauf in Abschnitt 3.2 und 4.3 näher eingegangen wird.

SPARQLAS ist eine Teilmenge von SPARQL als Anfragesprache, was sich vor allem im syntaktischen Aufbau einer Anfrage widerspiegelt. Wie SPARQL lässt sich SPARQLAS auch in zwei Abschnitte unterteilen. Der erste Teil befasst sich mit der Definition der Namensräume, was der Präfix Deklaration in SPARQL entspricht. Dieser Teil ist in SPARQLAS optional, da auch hier in der eigentlichen Anfrage absolute IRIs verwendet werden können. Mit SPARQLAS lassen sich SELECT und ASK Anfragen realisieren. Im Aufbau einer Anfrage sind beide allerdings identisch und werden somit beide durch den Term „Query“ eingeleitet. Den Unterschied zwischen SELECT oder ASK Anfrage erkennt man daran, ob in der Anfrage Variablen verwendet werden oder nicht. Das Vorhandensein von Variablen deutet auf eine SELECT Anfrage, sonst auf eine ASK Anfrage. Die eigentliche Anfrage wird dann über OWL-DL Ausdrücke formuliert.

Um diese OWL-DL Ausdrücke adäquat umzusetzen, bietet es sich an, die vom W3C empfohlene OWL Functional-Style Syntax [MPSP09] zu verwenden. Alternativ bietet SPARQLAS auch noch die Möglichkeit eine verkürzte Syntax ähnlich der Terme der Manchester Syntax [HPS09] zu verwenden, um die Anfragen noch kürzer und übersichtlicher zu gestalten. Einen kompletten Überblick zur Grammatik von SPARQLAS findet man auf der entsprechenden Homepage[7] innerhalb der Online-Repräsentation des TwoUse Projekts.

Die Beispielanfrage lässt sich mit SPARQLAS folgendermaßen umsetzen:

```
Namespace ( = <http://www.co-ode.org/ontologies/pizza/
     pizza.owl#> )
Query ( SubClassOf ( ?x Some ( hasTopping CheeseTopping ) ) )
```

2.1.3 Unterschiede zwischen SPARQL und SPARQLAS

Wie bereits erwähnt gilt SPARQLAS als Teilmenge von SPARQL. Mit den Unterschieden zwischen diesen beiden Sprachen beschäftigt sich dieser Abschnitt, welche am Ende durch Tabelle 2.1 nochmals zusammengefasst werden.

Ein offensichtlicher Unterschied ist die Unterstützung von nur zwei von vier Anfrageformen durch SPARQLAS, nämlich SELECT und ASK. Dies liegt darin begründet, dass eine CONSTRUCT oder DESCRIBE Anfrage einen RDF Graphen zurück liefert, der entweder durch die Anfrage neu erstellt worden ist oder ein gegebenes Element näher beschreibt. Da SPARQLAS allerdings zum Ziel hat, über OWL-DL Ausdrücke Anfragen auf OWL-DL Ontologien zu vereinfachen, werden diese beiden Anfragetypen nicht unterstützt.

In SPARQL ist es möglich, innerhalb des Prologs Präfixe und/oder eine Basis IRI zu definieren. In SPARQLAS hingegen gibt es nur die Möglichkeit

[7] http://code.google.com/p/twouse/wiki/SPARQLASGrammar

Namensräume zu deklarieren, die den Präfixen aus SPARQL entsprechen. Eine Basis IRI wird nicht verwendet, da sie im Prinzip einer relativen IRI mit leeren Präfix gleichzustellen ist und sich nur in der konkreten Syntax unterscheidet. Da in SPARQLAS die Definition eines leeren Namensraumes gestattet ist, wäre die Verwendung einer speziellen Basis IRI redundant und wird daher nicht unterstützt. An dieser Stelle soll auch noch darauf hingewiesen werden, dass sich die Syntax einer relativen IRI mit leeren Präfix bzw. Namensraum in SPARQL bzw. SPARQLAS unterscheiden. In SPARQL wird *:element* verwendet, in SPARQLAS indes nur *element*, was die Lesbarkeit der Anfrage erhöhen soll.

Nach dem Prolog steht bei beiden Sprachen die eigentliche Anfrage an nächster Stelle, jedoch wird diese in SPARQLAS mit „Query“ und in SPARQL entweder mit „SELECT“ oder „ASK“ eingeleitet. In SPARQL können darauf die Terme „DISTINCT“ oder „REDUCED“ mit der Angabe der zu erfragenden Variablen folgen. In SPARQLAS ist dies nicht notwendig, so dass die Anfrage direkt gestellt werden kann. Bei der Transformation von SPARQLAS zu SPARQL wird jedoch standardmäßig ein „DISTINCT“ mit der Angabe jeder verwendeten Variable hinzugefügt, falls es sich um eine SELECT Anfrage handelt. Auf die Spezifizierung des grundlegenden Datensatzes, in SPARQL durch „FROM“ bzw. „FROM NAMED“ realisiert, wird in SPARQLAS verzichtet. Dies wird von SPARQLAS nicht benötigt, da dies auch in einer SPARQL Anfrage optional ist und durch das SPARQL Protocol for RDF umgesetzt werden kann. Die aus SPARQLAS resultierende SPARQL Anfrage gestaltet sich daher dementsprechend. Die eigentliche Anfrage unterscheidet sich dann in der Verwendung von RDF Tripeln in SPARQL und von OWL-DL Ausdrücken in SPARQLAS.

Eine SPARQLAS Anfrage ist an dieser Stelle abgeschlossen, eine SPARQL Anfrage kann indessen noch weiter verfeinert werden. So können dort noch optionale, alternative oder auf spezielle Graphen beschränkende Bedingungen nebst verschiedenen Filtermöglichkeiten angegeben werden. Auch lässt sich die Ergebnismenge ordnen und auf eine Anzahl an Elementen beschränken. In SPARQLAS werden diese Möglichkeiten nicht angeboten, da diese Verfeinerungen einerseits die Übersichtlichkeit der Anfrage teilweise beeinträchtigen und andererseits für OWL-DL spezifische Anfragen nicht unbedingt notwendig sind. Auf das Ordnen und Beschränken der Elemente einer Ergebnismenge wird bewusst verzichtet, da dies für ASK Anfragen nicht möglich ist und daher der Aufbau einer SELECT und ASK Anfrage in SPARQLAS nicht mehr identisch wäre.

Dies sind die Unterschiede zwischen SPARQL und SPARQLAS und verdeutlichen, dass SPARQLAS eine Teilmenge von SPARQL darstellt, aber im Aufbau und Syntax einer Anfrage übersichtlicher und einfacher gestaltet ist. Allerdings ist SPARQL ausdrucksstärker und bietet mehr Anfragemöglichkeiten, so dass SPARQLAS eine echte Teilmenge ist.

Da man nun einen guten Eindruck hat, wie SPARQLAS aufgebaut ist, wird im folgenden Abschnitt EMFText näher beschrieben, womit die SPARQLAS Syntax und der zugehörige Editor entwickelt worden sind.

	SELECT	**ASK**	**CONSTRUCT**	**DESCRIBE**
SPARQL	ja	ja	ja	ja
SPARQLAS	ja	ja	nein	nein
	Prolog	**FROM**	**Variablenangabe**	**Anfragesyntax**
SPARQL	Präfix, Basis IRI	ja	ja	RDF Tripel
SPARQLAS	Namensräume	nein	nein	OWL-DL Ausdrücke
	Bedingungen	**Filter**	**Ordnen**	**Beschränkung**
SPARQL	ja	ja	ja	ja
SPARQLAS	nein	nein	nein	nein

Tabelle 2.1. Unterschiede zwischen SPARQL und SPARQLAS

2.2 EMFText

EMFText[8] ist ein Plugin zur quelloffenen integrierten Entwicklungsumgebung (IDE[9]) Eclipse[10] und ermöglicht das Erstellen von textueller Syntax für Sprachen, die durch ein Ecore [SBPM09, S. 17 ff] Metamodell beschrieben sind. Dieses Plugin wurde an der Technischen Universität Dresden im Institut für Software- und Multimediatechnik im Zusammenhang des Lehrstuhls Softwaretechnologie entwickelt. Es wird mittlerweile als eigenständiges Tool angeboten, obwohl es ursprünglich Bestandteil des Reuseware Composition Framework[11] war. Um EMFText und dessen Funktionalitäten besser zu verstehen, werden vorerst Eclipse und Ecore bzw. EMF näher erklärt.

Mit Eclipse bezeichnet man einerseits eine IDE, andererseits auch die Open Source Gemeinschaft, die sich um diese IDE aufgebaut hat. Das Eclipse Project wurde im November 2001 von IBM und einem Konsortium weiterer Hard- und Softwarehersteller wie z.B. Borland und SuSe gegründet. Im Januar 2004 wurde die Eclipse Foundation als unabhängige Not-For-Profit-Gesellschaft gegründet, welche jegliche Verwaltungsaufgaben bzgl. der Eclipse Gemeinschaft übernommen hat. Diese Gemeinschaft lebt von der aktiven Teilnahme ihrer Mitglieder und besitzt momentan über 60 Open Source Projekte. Eclipse als IDE baut auf Java auf und wird meistens auch als Java IDE verwendetet. Da

[8] http://www.emftext.org/index.php/EMFText
[9] engl. Integrated Development Environment
[10] http://www.eclipse.org/
[11] http://www.emftext.org/index.php/Main_Page

Eclipse quelloffen vorliegt und das Entwickeln von Plugins ein Hauptaugenmerk der Gemeinschaft darstellt, kann Eclipse theoretisch als IDE für jegliche Programmiersprache dienen, solange entsprechende Plugins vorhanden sind. Aktuell liegt Eclipse in der Version 3.5.2 vor und wird Galileo genannt. Dies ist auch gleichzeitig die Version, die zur Entwicklung innerhalb dieser Studienarbeit verwendet worden ist. Für EMFText wäre allerdings Version 3.4 schon ausreichend gewesen.

Ecore ist ein Metamodell um weitere Metamodelle zu beschreiben, also ein Meta-Metamodell, und Bestandteil des Eclipse Modeling Framework (EMF)[12]. EMF wiederum ist ein Teil des Eclipse Modeling Project[13], was, wie der Name schon andeutet, ein Eclipse Projekt darstellt. Ecore basiert auf einem EMF Modell und beschreibt folglich auch weitere EMF Modelle. Diese EMF Modelle beruhen auf einem XMI[14] [Obj07] Dokument, welche mit jeglichen Texteditoren erstellt werden können. Alternativ bietet EMF auch die Möglichkeit, ein EMF Modell mittels einem graphischen Baumeditor zu erzeugen, was sich im Vergleich einfacher gestaltet. EMF bietet zudem die Möglichkeit aus einem EMF Modell Java Code zu generieren, der dem gegebenen EMF Modell entspricht [SBPM09, S. 23 ff]. Sowohl das EMF Modell wie auch der daraus erzeugte Java Code wird von EMFText verwendet, um daraus eine textuelle Syntax zu erzeugen. Zur Zeit wird EMF in der Version 2.5 angeboten, zur Verwendung von EMFText reicht allerdings schon Version 2.4 vollkommen aus.

Wie bereits erwähnt, benötigt EMFText bei der Erzeugung einer neuen textuellen Syntax zuerst ein Metamodell in Ecore Notation. Aus diesem Metamodell wird per EMF ein so genanntes GenModel erstellt, woraus im späteren Verlauf auch die zugehörige Repräsentation in Java Code generiert wird. Dieses GenModel wird benötigt, da die zur Syntaxerstellung benötigte Concrete Syntax Datei (**.cs*) auf das GenModel verweist. Diese Elemente, Metamodell, GenModel und Concrete Syntax, bilden den Grundstein für das Erstellen einer textuellen Syntax und sind die einzigen Dateien, die vom Benutzer während der Erstellung editiert werden müssen. Zur schnellen Generierung einer Syntax bietet EMFText die Option an, aus einem GenModel eine Repräsentation in HUTN[15] [Obj04] Syntax abzuleiten. HUTN bietet zu jedem nicht abstrakten Element eines Metamodells eine textuelle Repräsentation in EBNF[16] [Int96], welche auch von der Concrete Syntax verwendet wird. Aus dieser Concrete Syntax leitet EMFText eine kontextfreie Grammatik ab, die an ANTLR[17] weitergeleitet wird, um Parser und Lexer zu generieren. Parser und Lexer werden deshalb benötigt, um die einzelnen Elemente einer textuellen Syntax den entsprechenden Elementen des Metamodells zu-

[12] http://www.eclipse.org/modeling/emf/
[13] http://www.eclipse.org/modeling/
[14] XML Metadata Interchange
[15] Human-Usable Textual Notation
[16] Extended Backus Naur Form
[17] ANother Tool for Language Recognition, http://www.antlr.org/

ordnen zu können. EMFText erstellt aus der Concrete Syntax auch einen Printer, der aus einer EMF Resource, also einem speziellen Modell basierend auf dem Metamodell, eine entsprechende textuelle Syntax erzeugt. Zuletzt stellt EMFText auch einen textuellen Editor zur Verfügung, mit dem man die erstellte Syntax testen und verwenden kann. Einerseits lassen sich in diesem Editor bestimmte Schlüsselwörter farblich hervorheben und andererseits werden Fehlermeldungen angezeigt, wenn die Syntax nicht mit der definierten Grammatik übereinstimmt. Außerdem lässt sich zu jedem textuellen Element die zugehörige Repräsentation in der EMF Resource anzeigen.

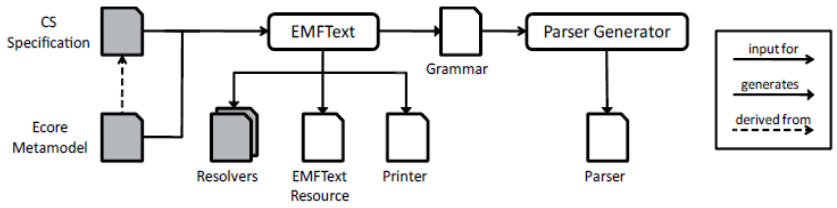

Abb. 2.1. Entwicklungsprozess von EMFText [EMF09b]

Während dem Entwicklungsprozess (siehe Abb. 2.1) generiert EMFText drei verschiedene Plugins, mit denen der Benutzer die textuelle Syntax mit Editor verwenden kann. Dazu muss der Benutzer das eigentliche EMFText Plugin nicht installiert haben, da die erzeugten Plugins unabhängig davon funktionieren. Zum einen wird ein Plugin erstellt, dass die Concrete Syntax, das Metamodell und den daraus abgeleiteten Java Code enthält. Ein zweites Plugin enthält verschiedene EMFText spezifische Klassen, um den Parser, Lexer, Printer und Editor verwenden zu können. Da der Parser und Lexer mit ANTLR verknüpft sind, wird ein drittes Plugin benötigt, welches bestimmte ANTLR Klassen bereitstellt. Dieses ANTLR Plugin ist bei jedem mit EMFText erstellten Plugin identisch, so dass bei mehreren EMFText Plugins immer darauf zurück gegriffen werden kann. Momentan wird von EMFText ANTLR in der Version 3.1.1 verwendet, obwohl bereits eine Version 3.2 angeboten wird. EMFText selbst liegt in der Version 1.2.3 vor, diese erschien allerdings erst am 05. März 2010 und Version 1.2.2 am 11. Februar 2010, so dass innerhalb dieser Studienarbeit noch die Version 1.2.1 verwendet worden ist.

Im Bezug auf die Transformation von SPARQLAS zu SPARQL wird EMFText an zwei Stellen verwendet. Zuerst wird zu einer textuellen SPARQLAS Anfrage eine EMF Resource erzeugt, welche als Eingabe für die Transformation dient. Dann wird aus der Ausgabe der Transformation, wiederum eine EMF Resource, eine textuelle SPARQL Anfrage erstellt. Diese besagte Transformation wird mittels ATL realisiert, welches im Folgenden näher erklärt wird.

2.3 ATL

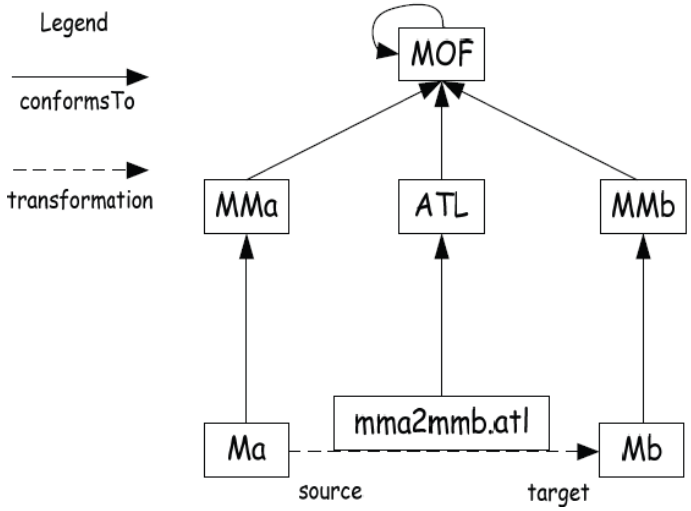

Abb. 2.2. Allgemeines Zusammenspiel von ATL und Metamodellen [Gro06]

Die ATLAS Transformation Language (ATL)[18] ist eine Sprache zur Modelltransformation und ein weiteres Eclipse Plugin. ATL wurde vom ATLAS[19] Forschungsteam in Kooperation von INRIA[20] und LINA[21] entwickelt und ist Bestandteil der ATLAS Model Management Architecture (AMMA) Plattform[22]. Weiter ist ATL Teil des Eclipse Projekts Model-to-Model Transformation (M2M)[23], was wiederum dem Eclipse Modeling Project zuzuordnen ist. Ursprünglich wurde ATL als Antwort auf die OMG MOF/QVT RFP[24] entwickelt und hat im Verlauf auch dessen Spezifikation beeinflusst. Mittlerweile wird ATL innerhalb der ATL-Pro Initiative[25] durch Obeo[26] weiterentwickelt, welche das Ziel verfolgt, eine industrielle Version von ATL zu entwickeln.

[18] http://www.eclipse.org/m2m/atl/
[19] http://atlas.lina.univ-nantes.fr/atlas/
[20] Institut National de Recherche en Informatique et en Automatique, http://www.inria.fr/
[21] Laboratoire d'Informatique de Nantes Atlantique, http://www.lina.univ-nantes.fr/
[22] http://atlanmod.emn.fr/AMMAROOT/
[23] http://www.eclipse.org/m2m/
[24] Object Management Group Meta Object Facility Query/Views/Transformations Request For Proposal, Spezifikation unter http://www.omg.org/spec/QVT/1.0/
[25] http://www.atl-pro.com/
[26] http://www.obeo.fr/

Mit ATL [Ecl10b] wird es ermöglicht, dass aus einem Eingangsmodell ein Ausgangsmodell transformiert werden kann. Diese beiden Modelle können sich dabei auf unterschiedliche Metamodelle beziehen, vorausgesetzt die Metamodelle beziehen sich auf dasselbe Meta-Metamodell, bspw. MOF[27] oder Ecore. Realisiert wird dies dadurch, dass ATL selbst zu diesem Meta-Metamodell konform ist und als Übersetzer vom Eingangs- zum Ausgangsmetamodell dient (siehe Abb. 2.2). Diese Übersetzung/Transformation lässt sich in ATL über Regeln ausdrücken. In diesen Regeln wird festgehalten, wie ein Element des Eingangsmetamodells zu einem Element des Ausgangsmetamodells transformiert werden soll. Da ATL eine hybride Sprache ist, können die Regeln einerseits deklarativ, andererseits imperativ gestaltet werden, wobei die deklarative Variante bevorzugt verwendet wird. Die Sprache ATL basiert zudem auf OCL[28] [Obj06], so dass verschiedene Datentypen und Operationen zu diesen verwendet werden können.

ATL Regeln gibt es in drei verschiedenen Formen. Die übliche Form ist die *Matched Rule*, welche bei jedem durch die Regel angegebenen Element angewendet wird. Dann existieren noch die *Lazy Rule* und *Called Rule*, welche beide aufgerufen werden müssen. Der Unterschied liegt hauptsächlich in welchem Teil einer Regel diese aufgerufen werden, *Lazy Rule* innerhalb eines deklarativen, *Called Rule* innerhalb eines imperativen Teils. Eine *Matched Rule* kann außerdem abstrakt oder von einer anderen *Matched Rule* abgeleitet sein und eine *Lazy Rule* darf als einzigartig ausgezeichnet sein, wenn diese immer dasselbe Ausgangselement zurück liefert. Zusätzlich zu den Regeln gibt es noch die Möglichkeit Methoden und allgemeine Variablen zu definieren, die *Helper* genannt werden.

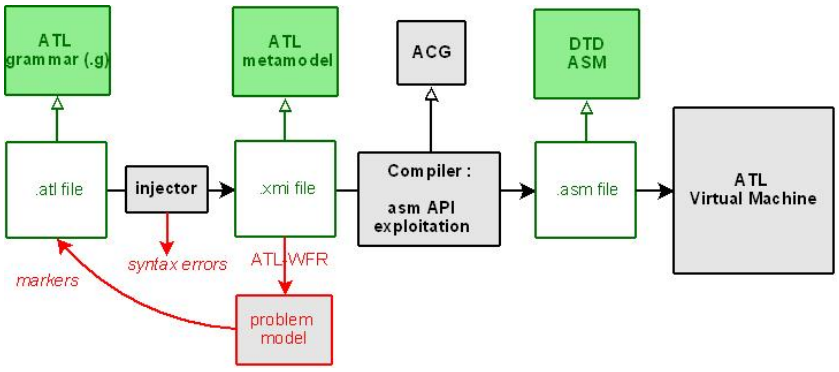

Abb. 2.3. Compilierung einer ATL Transformation [Ecl10a]

[27] Meta Object Facility, http://www.omg.org/spec/MOF/2.0/
[28] Object Constraint Language

Das ATL Plugin entspricht einer IDE und wird daher mit IDE typischen Werkzeugen angeboten. Dazu gehört einerseits eine spezielle ATL Perspektive in Eclipse, in der ein zugehöriger Editor geöffnet wird und bestimmte ATL Informationen über Sichten angegeben werden. Ein Wizard zum Erstellen von ATL Projekten und/oder Dateien gehört genauso wie ein Debugger zum Repertoire. Transformationen an sich werden über die ATL Virtual Machine realisiert, welche allerdings keine ATL, sondern eine ASM Datei erwartet. ASM ist eine Assembler ähnliche Sprache und wird im XML Format angeben. Diese wird aus der ATL Datei mit den Transformationsregeln über einen Parser und Compiler automatisch generiert. Einen graphischen Überblick zu diesem Erstellungsprozess bietet Abb. 2.3. Das aktuelle ATL Plugin liegt in der Version 3.0.1 vor und wird auch in dieser Studienarbeit verwendet. Die nächste ATL Version 3.1 ist für Juni 2010 und Version 3.2 für Juni 2011 angekündigt.

Hiermit ist die Vorstellung und Erläuterung der einzelnen Sprachen, der Entwicklungsumgebung und der verwendeten Plugins beendet. Daher soll nun im folgenden Kapitel auf die Anwendung dieser Plugins und die Entwicklung der SPARQLAS Syntax und der Transformation von SPARQLAS zu SPARQL eingegangen werden.

3

Vorarbeit

3.1 Das SPARQL Metamodell

Um die Transformation einer SPARQLAS Anfrage zu einer SPARQL Anfrage zu realisieren, müssen die entsprechenden Syntaxen unterstützt werden. Diese beiden Syntaxen werden in dieser Studienarbeit mit EMFText umgesetzt, da ATL mit EMF Resourcen als Ein- und Ausgabemodell umgehen kann. Wie zuvor erwähnt beruht eine mit EMFText genierte textuelle Syntax auf einem Ecore Metamodell. An dieser Stelle soll nun das verwendete SPARQL Metamodell näher geschildert werden.

Zuerst soll festgehalten werden, dass die komplette Syntax zu SPARQL nicht während dieser Studienarbeit erstellt worden ist, sondern bereits im EMFText Concrete Syntax Zoo [EMF10] vorhanden ist und bis auf kleine Anpassungen bzgl. der Textformatierung auch so verwendet wird. Diese Syntax unterstützt die gesamte Ausdrucksfähigkeit von SPARQL Anfragen. Genauer beschrieben werden jedoch nur die für die Transformation relevanten Elemente, also z.B. kein CONSTRUCT oder Filter. Dass keine „leichtere“ Version von SPARQL verwendet wird, ist damit begründet, dass im Zusammenhang mit dem TwoUse Toolkit jegliche SPARQL Anfragen unabhängig von SPARQLAS ausgeführt werden können. Somit wird im TwoUse Toolkit auf Redundanz verzichtet und auch in dieser Studienarbeit die gesamte SPARQL Syntax unterstützt.

Jede SPARQL Anfrage wird durch *SparqlQueries* eingeleitet, welche auf den *Prologue* und die *Query* verweist. Im *Prologue* wird nur das optionale *PrefixDecl* benutzt, was wiederum auf das Präfix in *PNAME_NS* und die IRI in *IRI_REF* hinweist. In beiden Fällen lässt sich das Präfix bzw. die IRI direkt als String angeben. Als *Query* kommen *SelectQuery* und *AskQuery* in Frage. Innerhalb einer *SelectQuery* sind nur ein möglicher *SolutionsDisplayNE*, mehrere *Var* und eine *WhereClause* interessant, für die *AskQuery* aus bekannten Gründen nur die *WhereClause*. Als *SolutionsDisplayNE* wird während der Transformation standardmäßig *DistinctNE* ausgewählt, um doppelte Ergebnisse auszublenden. Über *Var* werden Variablen direkt als String wiedergege-

ben und in der *WhereClause* wird der WHERE Teil näher beschrieben. Hier wird das *WhereLiteral* immer gesetzt, so dass in der Anfrage ein „WHERE“ hingeschrieben wird. Darauf folgt ein *GroupGraphPattern* mit einem *TriplesBlock*, in dem verschiedene *TriplesSameSubject*s liegen. Für OWL-DL Anfragen reicht es aus, als weitere Struktur immer *TriplesSameSubjectLeftNE* zu nehmen. Dieses besteht aus einem *VarOrTerm* und einer *PropertyListNotEmpty*.

Ein *VarOrTerm* ist entweder eine Variable (*Var*) oder ein *GraphTerm*. Ein *GraphTerm* kann eine Konstante, also eine absolute (*IRI_REF*) oder relative (*PNAME_LN*) IRI, eine *BlankNode* oder ein *RDFLiteral* sein. Mit *BlankNode* bezeichnet man eine spezielle Variable, über die sich die Zugehörigkeit von Triplen zueinander ausdrücken lässt. Soll eine *BlankNode* benannt werden, so setzt man einen String über *BLANK_NODE_LABEL*, eine anonyme *BlankNode* wird über *ANON* umgesetzt. Ein *RDFLiteral* besteht aus einem *StringLiteral* und einem *LANGTAGOrIRIrefNE*. Letzteres wird über *UpIRIrefNE* als IRI für einen Datentyp zum *StringLiteral* verwendet. Von der Transformation wird *STRING_LITERAL2* benutzt, da SPARQLAS für Literals dieselbe Konvention verwendet, nämlich den eigentlichen String umschlossen von normalen Anführungszeichen (”). Nun ist mit *VarOrTerm* der erste Teil eines Tripels abgeschlossen, der Rest befindet sich in *PropertyListNotEmpty*.

Eine solche *PropertyListNotEmpty* kann aus mehreren *Verb*s und mehreren *ObjectList*s bestehen. In Bezug auf OWL-DL Anfragen ist ein *Verb* immer eine IRI, manchmal stellt es ein abzufragendes Element dar, meistens wird jedoch der angegebene OWL-DL Ausdruck wiedergegeben, z.B. „rdf:type“. Eine *ObjectList* kann aus mehreren *Object*s bestehen, die auf eine *GraphNode* verweisen. Im einfachsten Fall ist eine *GraphNode* auch ein *VarOrTerm* und somit dann das RDF Triple abgeschlossen. Benötigt ein OWL-DL Ausdruck allerdings mehrere RDF Triple, dann ist eine Verschachtelung über eine *BlankNodePropertyList* als *GraphNode* machbar. Alternativ lässt sich dies auch über mehrere *TriplesSameSubject*s mit derselben *BlankNode* als *VarOrTerm* realisieren, aber für die Transformation ist der Weg über eine *BlankNodePropertyList* einfacher und kürzer umzusetzen. Dies liegt daran, dass innerhalb einer *BlankNodePropertyList* der *VarOrTerm* automatisch auf eine bestimmte *BlankNode* gesetzt wird. Deshalb ist nur noch die *PropertyListNotEmpty* anzugeben, die wie zuvor zu gestalten ist. Auf diese Weise sind mehrere Verschachtelungen oder auch Rekursion während der Transformation besser realisierbar.

Das SPARQL Metamodell besitzt *http://www.emftext.org/sparql* als Namespace URI[1] [BLFM05] und wird darüber in der ATL Transformation referenziert. Als Beispiel für die Verwendung dieses Metamodells sieht man in Abb. 3.1 eine einfache SPARQL Anfrage einmal in textueller Syntax und auch als Repräsentation als EMF Resource in der Outline Sicht. In dieser Sicht

[1] Uniform Resource Identifier

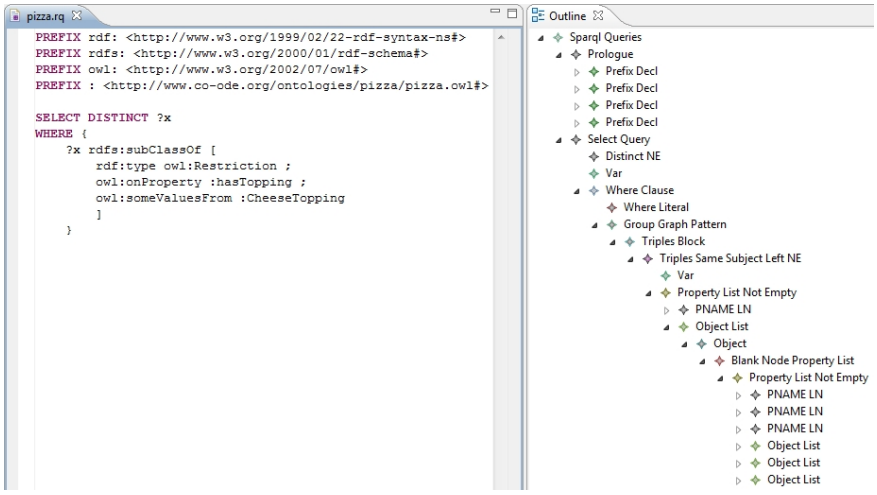

Abb. 3.1. Beipielhafte SPARQL Anfrage mit zugehöriger Outline Sicht

erkennt man gut, wie die einzelnen Elemente des Metamodells miteinander verknüpft sind und die textuelle Syntax widerspiegeln.

Dieser Teil des gesamten SPARQL Metamodells reicht aus, um SPARQLAS Anfragen in SPARQL Anfragen zu transformieren. Wie nun SPARQLAS unter der Oberfläche aussieht, wird im nächsten Abschnitt genauer behandelt.

3.2 Das SPARQLAS Metamodell

Da die Entwicklung einer textuellen Syntax zu SPARQLAS ein wichtiger Bestandteil dieser Studienarbeit darstellt, kann auf nicht so viel Vorarbeit wie im SPARQL Fall zurückgegriffen werden. Trotzdem werden zur Erstellung des SPARQLAS Metamodells einige Quellen zur Hilfe genommen. Dazu gehört im geringem Maße das zuvor beschriebene SPARQL Metamodell im Zusammenhang von Anfragenaufbau und Umsetzung von IRIs. Bei der Realisierung verschiedener Terme und OWL-DL Ausdrücken werden das Paper zu SPARQL-DL [SP07] und das Dokument zur OWL Functional-Style Syntax [MPSP09] berücksichtigt.

Jede SPARQLAS Anfrage ist ein *OntologyDocument*, welches aus mehreren *PrefixDefinition*s bestehen kann und genau eine *Query* besitzt. Innerhalb einer *PrefixDefinition* wird der Präfix eines Namensraums als String direkt und die absolute IRI als String über *FullIRI* angegeben. Eine *Query* kann eine *queryIRI* zur Identifikation, mehrere importierte Dokumente über *directlyImportsDocuments* und verschiedene *Atom*s beinhalten. *queryIRI* und *directlyImportsDocuments* sind *FullIRI*s, werden allerdings bei einer Transformation zu SPARQL nicht berücksichtigt, sind aber für eventuelle zukünftige

Entwicklungen schon einmal implementiert. Ein bestimmter OWL-DL Ausdruck wird durch ein spezielles *Atom* repräsentiert. Zur besseren Übersicht gibt es aber sechs verschiedene Kategorien an *Atom*s, nämlich *Assertion*, *ClassAtom*, *ObjectPropertyAtom*, *DataPropertyAtom*, *HasKey* und *Declaration*. In diesen Kategorien sind dann die Elemente enthalten, die die OWL-DL Konstrukte wiedergeben.

In *Assertion* werden alle ABox [Baa05, S. 15f] Ausdrücke gesammelt, also in welcher Beziehung Individuals zu Class, Data- und ObjectProperty stehen können. Zu *Assertion* gehören folgende Elemente: *ClassAssertion*, *DirectClassAssertion*, *ObjectPropertyAssertion*, *DataPropertyAssertion*, *NegativeObjectPropertyAssertion*, *NegativeDataPropertyAssertion*, *SameIndividual* und *DifferentIndividuals*. In diesen Konstrukten kommen *Individual*, *ClassExpression*, *ObjectPropertyExpression*, *DataPropertyExpression* und *Literal* zum Einsatz. All diese Elemente, mit Ausnahme von *Literal*, können als *Term* vorkommen, also als *Constant* oder *Variable*. Eine *Constant* ist entweder eine absolute (*FullIRI*) oder relative (*AbbreviatedIRI*) IRI und wird wie eine *Variable* als String angegeben. *Individual* bestitzt auch noch die Ausprägung zum *AnonymousIndividual*, womit sich nach der Transformation BlankNodes erzeugen lassen. *ClassExpression* und *ObjectPropertyExpression* haben noch weitere Erweiterungen, auf die später näher eingegangen wird. Ein *Literal* kann auch eine *Variable* sein, ansonsten besteht es aus einem String und einem *Datatype* im IRI Format.

TBox [Baa05, S. 13f] Ausdrücke werden durch *ClassAtom*, *ObjectPropertyAtom*, *DataPropertyAtom*, *HasKey* und *Declaration* wiedergegeben. Zu *ClassAtom* gehören *SubClassOf*, *DirectSubClassOf*, *StrictSubClassOf*, *EquivalentClasses*, *DisjointClasses* und *DisjointUnion*. Hier können verschiedene Beziehungen zwischen *ClassExpression*s ausgedrückt werden. Als eine erweiterte *ClassExpression*, also weder *Constant* noch *Variable*, gilt Folgendes: *ObjectUnionOf*, *ObjectComplementOf*, *ObjectOne*, *ObjectIntersectionOf*, *ObjectAllValuesFrom*, *ObjectSomeValuesFrom*, *ObjectHasValue*, *ObjectMinCardinality*, *ObjectMaxCardinality*, *ObjectExactCardinality*, *DataAllValuesFrom*, *DataSomeValuesFrom*, *DataHasValue*, *DataMinCardinality*, *DataMaxCardinality* und *DataExactCardinality*. In diesen können wiederum *ClassExpression*s auftreten, so dass mehrere Verschachtelungen dieser möglich sind. Je nach Ausdruck können wieder *Individual*s, *ObjectPropertyExpression*s, *DataPropertyExpression*s oder *Literal*s auftreten. Bei den Kardinalitätsrestriktionen wird die Kardinalität als positiven Integer Wert angegeben. Zusätzlich treten bei den *Data*-Elementen noch *DataRange*s auf, welche sich mit *Datatypes* beschäftigen. Weiter gehören zu *DataRange* noch *DataUnionOf*, *DataComplementOf*, *DataOneOf*, *DataIntersectionOf* und *DatatypeRestriction*, welche sich auch untereinander wieder verschachteln lassen.

Mit einem *ObjectPropertyAtom* lassen sich die Beziehungen von ObjectProperties untereinander weiter verdeutlichen. Unter einem *ObjectPropertyAtom* versteht man folgende Elemente: *SubObjectPropertyOf*, *EquivalentObjectProperties*, *DisjointObjectProperties*, *ObjectPropertyDomain*, *ObjectPro-*

pertyRange, *InverseObjectPropertyAtom*, *FunctionalObjectProperty*, *InverseFunctionalObjectProperty*, *ReflexiveObjectProperty*, *IrreflexiveObjectProperty*, *SymmetricObjectProperty*, *AsymmetricObjectProperty* und *TransitiveObjectProperty*. Von diesen Elementen werden *ObjectPropertyExpression*s aufgegriffen, welches neben *Constant* und *Variable* auch ein *InverseObjectProperty* sein kann. Letzteres lässt beliebig oft hintereinander verwenden, da es sich wieder auf eine *ObjectPropertyExpression* bezieht. Innerhalb von *SubObjectPropertyOf* kann anstelle einer einzelnen *subObjectPropertyExpression* auf eine *ObjectPropertyChain* verwiesen werden, was einer Ansammlung von *ObjectPropertyExpression*s entspricht. Ansonsten wird noch auf eine *ClassExpression* von *ObjectPropertyDomain* und *ObjectPropertyRange* Bezug genommen.

Analog zu *ObjectPropertyAtom* befassen sich *DataPropertyAtom*s mit DataProperties, wenn auch mit ein paar weniger möglichen Ausdrücken. Diese sind folgende: *SubDataPropertyOf*, *EquivalentDataProperties*, *DisjointDataProperties*, *DataPropertyDomain*, *DataPropertyRange* und *FunctionalDataProperty*. Hier wird auf *DataPropertyExpression*s weiter verwiesen, welche entweder einer *Constant* oder einer *Variable* entsprechen. Zusätzlich wird von *DataPropertyDomain* eine *ClassExpression* und von *DataPropertyRange* eine *DataRange* verwendet.

Mittels *HasKey* kann man ausdrücken, wie sich eine *ClassExpression* über verschiedene *ObjectPropertyExpression*s und/oder *DataPropertyExpression*s identifizieren lässt.

Über *Declaration* lassen sich wie anzunehmen Elemente deklarieren, die in keiner Beziehung zu anderen Elementen stehen. Dazu gehören *ClassDeclaration*, *ObjectPropertyDeclaration*, *DataPropertyDeclaration* und *IndividualDeclaration*. In diesen wird entweder eine Konstante oder eine Variable angegeben. Somit lassen sich Anfragen konstruieren, die alle Vorkommnisse eines bestimmten Typs, z.B. *Class*, erschließen (SELECT Fall) oder das Vorkommen eines bestimmten Elements bestätigen können (ASK Fall).

Wer sich schon etwas mit OWL beschäftigt hat, wird feststellen, dass Annotations von SPARQLAS nicht unterstützt werden. Dies liegt daran, dass Annotations ein anderes Element mittels eines Strings nur näher beschreiben, ähnlich eines Kommentars. Theoretisch ist eine solche Implementation keine Schwierigkeit, dennoch hat man sich dagegen entschieden, da Annotations nicht zur Übersichtlichkeit und Einfachheit einer Anfrage beisteuern.

Das SPARQLAS Metamodell kann von der ATL Transformation vewendet werden, indem auf die Namespace URI *http://west.uni-koblenz.de/sparqlas* referenziert wird. Ein kleines Beispiel für die Anwendung dieses Metamodells in Zusammenhang mit der erstellten textuellen Syntax findet man in Abb. 3.2. Man erkennt hier gut, dass im Vergleich zum selben Beispiel aus Abb. 3.1 weniger Elemente benötigt werden, um dieselbe Anfrage zu stellen. Eine genauere Evaluation zwischen SPARQL und SPARQLAS Anfragen findet man später in Abschnitt 4.5. Im nächsten Kapitel soll vorerst auf die Erstellung der textuellen Syntax von SPARQLAS mit zugehörigen Editor und danach auf die Transformation von SPARQLAS zu SPARQL eingegangen werden.

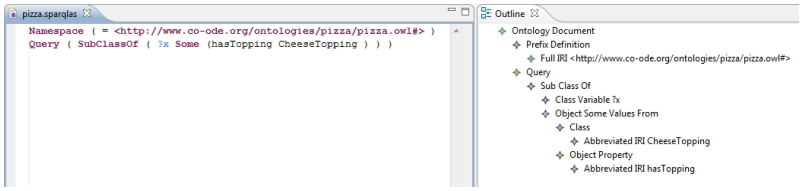

Abb. 3.2. Beipielhafte SPARQLAS Anfrage mit zugehöriger Outline Sicht

4

Realisierung und Evaluation

4.1 Überblick der Realisierungsschritte

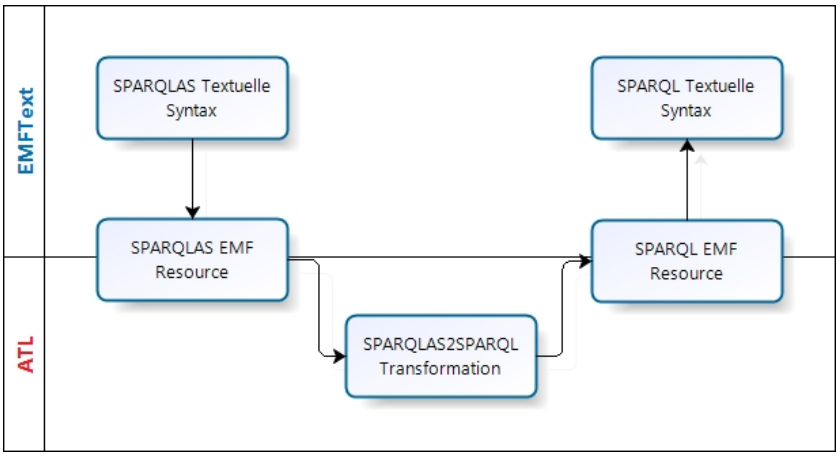

Abb. 4.1. Ablauf einer SPARQLAS zu SPARQL Transformation

Bevor in den Abschnitten 4.2 und 4.3 auf die Erstellung einer textuellen Syntax bzw. auf die ATL Transformation näher eingegangen wird, soll in diesem Abschnitt ein Überblick zu den einzelnen Schritten gegeben werden. Dieser Überblick wird anhand Abb. 4.1 illustriert und zeigt, dass es erwartungsgemäß mit SPARQLAS in textueller Syntax beginnt. An dieser Stelle wird eine Anfrage in SPARQLAS Notation gestellt und im Folgenden weiter verarbeitet. Als nächstes gilt es, die Transformation von SPARQLAS zu SPARQL durchzuführen. Da ATL jedoch SPARQLAS in textueller Syntax nicht verarbeiten kann, wird die zugehörige EMF Resource benötigt, welche von EMFText bereitgestellt wird. Nach Ausführung der ATL Transformation

erhält man eine SPARQL EMF Resource. Diese ist allerdings selbst für Benutzer mit XMI Erfahrung wenig lesbar, so dass aus dieser EMF Resource über den EMFText Printer eine textuelle Syntax in SPARQL Notation erstellt wird. Dieser letzte Schritt schließt zudem den Ablauf ab, so dass im folgenden Abschnitt detaillierter auf die textuelle Syntax zu SPARQLAS eingegangen werden kann.

4.2 Erzeugung der textuellen Syntax von SPARQLAS

Am Ende von 3.2 liegt das SPARQLAS Metamodell in Ecore Notation vor. Als nächster Schritt in der Erstellung einer textuellen Syntax mit EMFText gilt es, ein zugehöriges GenModel zu erzeugen. Dafür bietet das EMF Plugin schon entsprechende Funktionen an, so dass dies schnell umgesetzt werden kann. Man muss nur im Eclipse Projekt das SPARQLAS Metamodell auswählen und eine neue Datei erstellen, in diesem Fall ein *EMF Generator Model*. In den folgenden Schritten des Wizards kann man zuerst den automatisch erzeugten Namen des GenModels bestätigen, dann wird *Ecore Model* als *Model Importer* ausgewählt, woraufhin man schließlich das Metamodell laden muss bevor der Wizard abgeschlossen werden kann. Nun hat man ein GenModel, was als Grundlage für die Java Code Generierung und die Concrete Syntax dient. Am GenModel selbst muss nichts verändert werden, es ist nur notwendig ein *BasePackage* zu definieren, damit der Java Code im richtigen Projekt erzeugt wird. Da das Projekt zur textuellen Syntax *west.twouse.language.sparqlas* heißt, wird als *BasePackage west.twouse.language* angegeben. Der Java Code wird deshalb im richtigen Projekt erstellt, da das Metamodell selbst als *sparqlas* benannt ist. Um das Editieren der Concrete Syntax zu vereinfachen, wird als nächstes eine entsprechende Datei mit HUTN-Syntax erzeugt. Dies wird von EMFText angeboten und geschieht so, dass man die GenModel Datei auswählt, das Kontextmenü öffnet und die Option *Generate HUTN Syntax* verwendet.

Die nun entstandene Concrete Syntax besteht aus fünf Teilen: dem Kopf, den Optionsmöglichkeiten, den Tokendefinitionen, den Tokenformatierungen und den Regeln. Im Kopf wird zuerst angegeben, mit welcher Dateiendung die zu erzeugende textuelle Syntax verknüpft werden soll, hier *sparqlas*. Darauf folgt die Referenzierung des Metamodells über die entsprechende Namespace URI, also *http://west.uni-koblenz.de/sparqlas*, und zum Schluss wird das Startelement des Metamodells angegeben, für SPARQLAS ist dies *OntologyDocument*.

Optionsmöglichkeiten gibt es mehrere, z.B. lässt sich hier angeben, wie das Projekt heißen soll, welches die spezifischen EMFText Klassen wie Printer, Lexer etc. beinhaltet, hier *west.twouse.language.sparqlas.resource.sparqlas*. Eine weitere Option besteht in der Angabe, ob der Java Code zum Metamodell bei der Generierung der textuellen Syntax automatisch neu erstellt werden soll. Falls dies nicht aktiviert ist und etwas am Metamodell geändert wird,

dann kann man den Code auch per Hand über Funktionen des EMF Plugins neu erzeugen. Eine nicht unwichtige Option ist noch die Möglichkeit die Verwendung von vordefinierten Tokens zu Deaktivieren, da sonst der Parser die textuelle Syntax nur nach Standard Tokens analysiert.

In den Tokendefinitionen werden wie anzunehmen spezielle Tokens über reguläre Ausdrücke definiert. Dies ist bei der Erstellung der textuellen Syntax besonders wichtig, da sonst unter Umständen die einzelnen Elemente einer SPARQLAS Anfrage nicht den richtigen Elementen des Metamodells zugeordnet werden können. Einerseits werden für SPARQLAS Standard Tokens definiert, nämlich für Leerräume, Zeilenumbrüche, Kommentare und Integerwerten. Andererseits werden noch fünf für SPARQLAS spezifische Tokens definiert. Dazu gehört zum einen ein Token für den String Teil eines Literals, also dass sich ein Text innerhalb von normalen Anführungszeichen (") befinden muss. Ein weiterer Token gilt für Variablen. Diese müssen entweder mit einem Fragezeichen (?) oder einem Dollarzeichen ($) beginnen, worauf ein Variablenname folgt. Dann gibt es noch ein Token für BlankNodes, welche mit einem Unterstrich und Doppelpunkt (_:) eingeleitet werden. Zuletzt gibt es noch zwei Tokens, um absolute und relative IRIs verwenden zu können.

Bei den Tokenformatierungen lässt sich angeben, wie bestimmte Elemente der textuellen Syntax im Editor hervorgehoben werden sollen. Hier ist angegeben, dass alle Schlüsselwörter wie z.B. *Query* oder *SubClassOf* und absolute IRIs fett gedruckt und in einer violetten Farbe dargestellt werden sollen. Variablen wiederum werden speziell durch eine hellblaue Farbe hervorgehoben. Jeglicher anderer Text wie z.B. relative IRIs oder Strings werden im standardmäßigen Schwarz angezeigt. Diese Wahl an Tokenformatierungen beruhen teilweise auf der von EMFText empfohlenen Formatierung, entsprechen aber auch sonst der subjektiven Einschätzung des Autors und besitzen keinen weiteren Hintergrund.

Die Regeln beschreiben den Aufbau der textuellen Syntax und sind vorerst in HUTN-Syntax wiedergegeben. Diese entspricht wie erwartet nicht der zum Ziel gesetzten SPARQLAS Syntax und muss demzufolge angepasst werden. Zuerst gilt es, die erstellten Schlüsselwörter zu editieren, da zum einen nicht jedes Element ein Schlüsselwort besitzt, z.B. wird eine Variable nur anhand ihres Tokens erkannt, und zum anderen wird von SPARQLAS sowohl die OWL Functional-Style Syntax als auch eine verkürzte Syntax ähnlich der Manchester Syntax unterstützt. Als nächstes werden die geschweiften Klammern an geeigneten Stellen durch runde Klammern ersetzt und sonstige überflüssige Syntax entfernt. Danach wird es etwas aufwendiger, da die Properties der einzelnen Elemente gemäß der richtigen Reihenfolge und der Kardinalitäten umstrukturiert werden müssen. Durch die HUTN-Syntax werden nämlich alle Properties so gestaltet, dass keine Reihenfolge und Kardinalität vorgegeben sind. Außerdem müssen den Properties, die Attribute und keine Referenzen sind, die entsprechenden Tokendefinitionen zugeordnet werden. Besondere Symbole müssen nur an zwei verschiedenen Stellen eingefügt werden. Einmal bei *PrefixDefinition*, da einem Präfix ein Namensraum per Gleichheitszeichen

(=) zugewiesen wird, und ein anderes Mal bei *Literal*, da der String vom *Datatype* durch zwei Zirkumflex Akzente (^^) getrennt wird.

Nach diesen Veränderungen ist die Concrete Syntax vollständig und es kann nun ein zugehöriger Editor zum Testen und Anwenden der textuellen Syntax erstellt werden. Dies geschieht über eine EMFText Option indem man die Concrete Syntax Datei auswählt und im Kontextmenü *Generate Text Resource* verwendet. Nun erstellt EMFText automatisch den Java Code zum Metamodell, ein weiteres Projekt mit EMFText spezifischen Klassen und, falls noch nicht vorhanden, ein Projekt mit bestimmten ANTLR Klassen unter dem Namen *org.emftext.commons.antlr3_1_1*. Außerdem erzeugt EMFText besondere Plugin Dateien, so dass diese Eclipse Projekte auch als Plugins ausgeführt werden können. Zum lokalen Testen kann man einfach eine *Eclipse Application* starten und eine neue SPARQLAS Datei erzeugen. Daraufhin öffnet sich sofort der zugehörige Editor mit Outline Sicht und SPARQLAS Anfragen können erstellt werden. Ansonsten lassen sich die Plugins noch als Java Archive exportieren, so dass die Plugins auch an weitere Benutzer verteilt werden können.

Um die einzelnen Schritte, wie man von einem Metamodell in Ecore Notation zu einer beliebigen textuellen Syntax gelangt, nochmals bildlich nachzuvollziehen, sei an dieser Stelle auf die detaillierte Videoanleitung zu einer kleinen textuellen Syntax auf der EMFText Webseite [EMF09a] hingewiesen. Außerdem soll festgehalten werden, dass die textuelle Syntax zu SPARQLAS für jedermann über den EMFText Concrete Syntax Zoo unter `http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_SPARQLAS` zugänglich ist.

Nachdem nun diese textuelle Syntax entwickelt worden ist, soll im nächsten Abschnitt die ATL Transformation von SPARQLAS zu SPARQL weiter beleuchtet werden.

4.3 Transformation von SPARQLAS zu SPARQL

Das ATL Dokument zur Transformation von SPARQLAS zu SPARQL besteht aus drei Teilen: dem Kopf, den Helfern und den Regeln, wobei letztere den Hauptanteil bilden. Im Kopf wird zuerst angegeben, wie die Transformation heißt, in diesem Fall *sparqlas2sparql*. Danach werden die verwendeten Metamodelle benannt, so dass die Regeln die zu transformierenden Elemente eindeutig zuordnen kann. Hier sind das Eingangsmodell (*IN*) als *sparqlas* und das Ausgansmodell (*OUT*) als *sparql* bezeichnet. Im Kopf kann man auch noch zu verwendende Hilfsbibliotheken einbinden, dies ist aber für diese Transformation nicht notwendig und wird somit ausgelassen. Nun ist der Kopf auch schon abgeschlossen, so dass die Helfer definiert werden können. Insgesamt werden in dieser Transformation eine Hilfsmethode und eine Hilfsvariable benötigt. Die Hilfsmethode dient dem Zweck, bei einer relativen IRI ohne Präfix einen Doppelpunkt (:) vorzusetzen, da SPARQL dies so verlangt. Die Hilfsvariable ist die Menge aller Variablen innerhalb der Anfrage und wird nur

zur Unterscheidung zwischen SELECT und ASK Anfrage verwendet. Damit sind nun auch die Helfer erläutert, so dass jetzt auf die Transformationsregeln eingegangen werden kann.

Der Übersicht halber sind die Regeln so angeordnet, wie eine SPARQL Anfrage nach Metamodell aufgebaut ist. Daher fängt es mit der Transformation der Startelemente an, also von *OntologyDocument* zu *SparqlQueries*. Zuerst wird der *Prologue* mit seinen Präfixdefinitionen angegeben, worauf die *Query* folgt. Zu den Präfixen gehören zum einen standardmäßig angegebene Präfixe und die Präfixe, die sich aus den Namensraumangaben der SPARQLAS Anfrage ergeben. Die standardmäßigen Präfixe werden immer definiert, auch wenn diese von der speziellen Anfrage nicht alle gebraucht werden, da eine genaue Untersuchung der SPARQLAS Anfrage auf Verwendung von festen Präfixen unverhältnismäßig zum Ertrag einer Nicht-Definition steht. Zu diesen Definition gehören die Präfixe *rdf*, *rdfs*, *owl*, *xsd* und *sparqldl* mit den entsprechenen absoluten IRIs. Um die richtige *Query* auszuwählen, kommt die eben erwähnte Hilfsvariable zum Einsatz. Existieren Variablen in der Anfrage, so wird eine *SelectQuery* gewählt, sonst eine *AskQuery*. Für die *AskQuery* wird die *WhereClause* genauso gestaltet wie für die *SelectQuery*, welche zusätzlich *DistinctNE* als *SolutionDisplayNE* nebst der Angabe der verwendeten Variablen erhält. In der *WhereClause* wird das *WhereLiteral* immer gesetzt und eine Lazy Rule zur *TriplesBlock* Bildung aufgerufen. Dies hat einerseits den Grund, da in SPARQLAS die *Atom*s in die Kategorien *Assertion*, *ClassAtom*, *ObjectPropertyAtom*, *DataPropertyAtom*, *HasKey* und *Declaration* aufgeteilt sind und dies zur besseren Übersicht für die Transformation übernommen wird. Der andere und entscheidende Grund ist der, dass bei der Abbildung der OWL-DL Ausdrücke auf RDF Tripel manche Regeln zwei Eingabeelemente benötigen und diese Regeln speziell aufgerufen werden müssen. Bevor allerdings die Regeln zu diesen Abbildungen umgesetzt werden, sind noch einige Lazy Rules zu definieren, um wiederkehrende Transformationen einfacher bereit zu stellen. So werden in mehreren Transformationsregeln z.B. Variablen erstellt oder es werden verschiedene OWL-DL Terme öfters verwendet wie *owl:onProperty*. Kardinalitäten oder Literals werden genauso von verschiedenen Regeln aufgegriffen wie auch die *ObjectList* Bildung bei Verschachtelungen oder das Erstellen einer Sequenz.

Nachdem all diese Regeln erstellt worden sind, gilt es nun die Regeln zu den Abbildungen von OWL-DL Ausdrücken zu RDF Tripeln umzusetzen. Diese Abbildungsregeln beruhen zum großen Teil auf dem vom W3C empfohlenen Dokument zur Abbildung von OWL auf RDF Graphen [PSM09] und zur OWL Semantik und abstrakten Syntax [PSH04]. Im einfachen Fall gibt es pro Abbildung eine Regel, so dass diese Regeln keine weitere Beschreibung benötigen. Daher wird im Folgenden nur auf die Regeln näher eingegangen, die etwas umfangreicher oder komplizierter gestaltet sind.

Die erste zu erläuternde Regel bezieht sich auf den OWL-DL Ausdruck zu *ObjectPropertyAssertion*. Hierzu gibt es nach den Abbildungsdokumenten zwei verschiedene Ausprägungen. Normalerweise wird bei einer Regel nur

der Typ des Elementes überprüft. Hier wird zusätzlich noch überprüft, ob es sich bei der *ObjectPropertyExpression* um eine *InverseObjectProperty* handelt oder nicht. Falls dem so ist, dann wird die entsprechende Regel gemäß der Abbildungsregel das *SourceIndividual* mit dem *TargetIndividual* vertauschen. Mit den Regeln zu *SameIndividual* und *DifferentIndividuals* treten die ersten Lazy Rules auf, die aus dem *TriplesBlock* aufgerufen werden, da diese zwei Parameter benötigen. Dies liegt an der zugehörigen Abbildungsregel, da ein OWL-DL Term immer zwischen zwei verschiedenen Individuals gelten muss. Diese Regeln schließen zugleich den *Assertion*s Teil der Transformation ab.

Die *ClassAtom* Regeln zu *SubClassOf*, *DirectSubClassOf*, *StrictSubClassOf*, *EquivalentClasses* und *DisjointClasses* verhalten sich etwas anders und liegen jeweils in zwei verschiedenen Version vor. Zuerst soll aber erwähnt sein, dass *EquivalentClasses* und *DisjointClasses* ähnlich zu *SameIndividual* und *DifferentIndividuals* transformiert werden und somit auch aus dem *TriplesBlock* aufgerufen werden. Der Grund für die zwei Versionen liegt darin, dass als erstes angegebenes Element für diese drei Konstrukte eine *ClassExpression* ungleich einer Konstante oder Variable verwendet werden kann. Dies hat zur Folge, dass diese *ClassExpression* nicht direkt als *VarOrTerm* angegeben werden kann und so über eine anonyme BlankNode ($ANON$) transformiert werden muss. Daher werden zu allen Ausprägungen an *ClassExpression*s zusätzliche Transformationsregeln benötigt, die zur *ClassExpression* an sich noch die *SuperClassExpression* bzw. die weiteren *ClassExpression*s der *EquivalentClasses* oder *DisjointClasses* als Parameter übergeben bekommen. Somit wird die *ClassExpression* zuerst umgesetzt und dann wie im einfachen Fall einer *SubClassOf*, *EquivalentClasses* oder *DisjointClasses* Regel weiter transformiert. Diese erwähnten *ClassExpression*s besitzen bis auf die Kardinalitätsrestriktionen jeweils eine Regel, wie es die entsprechende Abbildung verlangt. Bei den Kardinalitätsrestriktionen ist die Angabe einer *ClassExpression* bzw. *DataRange* optional, so dass auf ein Vorhandensein dieser überprüft werden muss und sich so zwei verschiedene Regeln pro Kardinalitätsrestriktion ergeben. Die *DataRange* Regeln verhalten sich wiederum einfach mit einer Regel pro *DataRange* Ausprägung, so dass hiermit die *ClassAtom* Regeln abgeschlossen sind.

Bei allen *ObjectPropertyAtom* Regeln ist es möglich, als erste oder einzige *ObjectPropertyExpression* eine *InverseObjectProperty* anzugeben. Darum gibt es zu jedem *ObjectPropertyAtom* zwei verschieden Ausprägungen, einmal mit einer Konstante oder Variable als *VarOrTerm* und einmal mit einer anonymen BlankNode. Die weitere Transformation wird analog zu den jeweiligen *ClassExpression*s auch für *InverseObjectProperty* fortgesetzt. Zusätzlich zu diesen zwei Ausprägungen, besitzt *SubObjectPropertyOf* noch zwei weitere Regeln, da anstelle einer *SubObjectPropertyExpression* auch eine *SubObjectPropertyChain* verwendet werden kann. In diesem Fall wird die *SuperObjectPropertyExpression* auf eine *InverseObjectProperty* überprüft und danach die dementsprechende Regel verwendet. Von den *ObjectPropertyAtom* Regeln

werden nur *EquivalentObjectProperties* und *DisjointObjectProperties* wegen der Verwendung von zwei Parametern aus dem *TriplesBlock* aufgerufen.

Die *DataPropertyAtom* Regeln sind alle einfach gehalten und benötigen keine weitere Erklärung. Auch hier werden zwei Regeln aus dem *TriplesBlock* aufgerufen und zwar *EquivalentDataProperties* und *DisjointDataProperties*.

HasKey ist sowohl ein Atom als auch ein OWL-DL Ausdruck mit zugehöriger Abbildung. Allerdings ist *HasKey* etwas komplexer aufgebaut, so dass mehrere Arten an Regeln notwendig sind. Zum einen kann die verwendete *ClassExpression* auch keiner Konstanten oder Variablen entsprechen, so dass spezielle *ClassExpression* Regeln wieder benötigt werden. Zum anderen muss unterschieden werden, ob *ObjectPropertyExpresssion*s oder *DataPropertyExpression*s verwendet werden, da beide zwar optional sind, aber mindestens eine der beiden besetzt sein muss. Somit wird überprüft, ob eine der beiden oder beide mindestens eine *Expression* enthält. Demnach werden die weiteren Transformationsregeln gewählt und aufgerufen.

Schließlich gibt es noch die *Declaration* Regeln, die jedoch einfach umgesetzt sind und hier nicht weiter geschildert werden müssen.

Mittels dieser erwähnter Regeln ist es nun möglich, SPARQLAS Anfragen in SPARQL Anfragen zu transformieren, welche man im ATL Projekt *west.twouse.transformation.sparqlas2sparql* findet. Jedoch nützt ein Satz von Transformationsregeln nichts, wenn diese keine Anwendung finden. Daher wird in den nächsten Abschnitten dargelegt, wie und womit man eine Transformation von SPARQLAS zu SPARQL ausführen kann.

4.4 Anwendungsmöglichkeiten der Transformation

Wie zuvor erwähnt, werden im Zuge dieser Studienarbeit drei verschiedene Möglichkeiten angeboten, um die Transformation von SPARQLAS zu SPARQL anzuwenden. Zuerst wird die Anwendung als Eclipse Plugin beschrieben am Beispiel eines Kontextmenüeintrages innerhalb des TwoUse Toolkits. Danach wird eine von Eclipse unabhängige Standalone Version in Form einer API auf Basis von Java vorgestellt. Zuletzt wird auf die Umsetzung der Transformation als Webservice näher eingegangen, welcher im Beispiel über ein PHP[1] Skript aufgerufen wird.

4.4.1 Eclipse Plugin

Um die ATL Transformation programmatisch über Java aufrufen zu können, muss man zuerst verstehen, wie ATL intern eine Transformation ausführt. Einen guten Überblick dazu gewährt Abb. 4.2. Hier sieht man, dass über eine *Launch Configuration* oder einen *Ant Task* die Parameter zu einer Transformation angegeben werden können und diese daraufhin durchgeführt wird.

[1] PHP: Hypertext Preprocessor, http://www.php.net/

Bei einer programmatischen Umsetzung greift man direkt auf den *Core-* und *LauncherService* zu, um die entscheidenden Parameter einer Transformation zu übergeben. Zu diesen Parametern gehören einerseits die Metamodelle zu SPARQLAS und SPARQL sowie die spezielle EMF Resource einer SPARQLAS Anfrage und andererseits die ASM Datei als Repräsentation der Transformationsregeln. Diese Ausführung der Transformation findet man im Eclipse Projekt *west.twouse.backend* im Package *west.twouse.backend.ontology.transformations.sparqlas*.

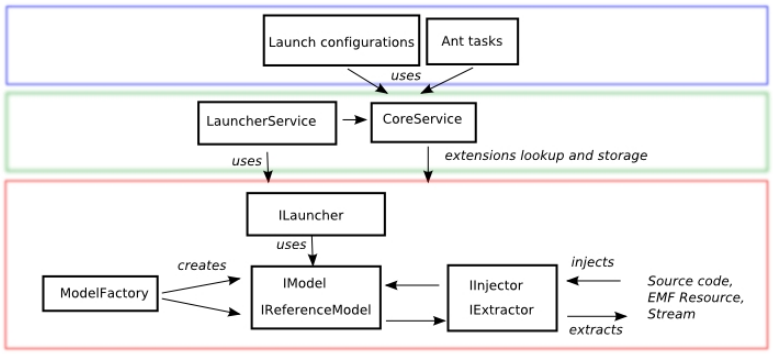

Abb. 4.2. Architektur der Ausführung einer ATL Transformation [Ecl10a]

Diese Transformationsausführung erwartet als einzige Parameter den Pfad zur SPARQLAS Datei und den Pfad, wo die zugehörige SPARQL Datei gespeichert werden soll. Da ATL eine EMF Resource und keine textuelle Syntax von SPARQLAS als Eingabe erwartet, müssen die von EMFText bereitgestellten Parser und Lexer vorher zum Einsatz kommen. Dies wird automatisch bewerkstelligt, wenn man das Metamodell zu SPARQLAS anhand seiner Namespace URI referenziert. Folglich muss das erstellte Plugin zu dieser textuellen Syntax von diesem Eclipse Projekt importiert werden. Für die Transformation selbst wird die ASM Datei über ihren Projektpfad angegeben, so dass auch das ATL Projekt eingebunden sein muss. Das anschließende Erstellen der SPARQL Anfrage geschieht über den Printer des entsprechenden EMFText Plugins und wird auch automatisch angesprochen, wenn das SPARQL Metamodell mittels seiner Namespace URI angegeben wird.

Bevor es jedoch zur eigentlichen Transformation kommt, wird noch überprüft, ob die SPARQLAS Anfrage syntaktisch korrekt ist. Eine syntaktisch korrekte Anfrage ist äquivalent zu einer EMF Resource ohne Fehler. Ist dies nicht der Fall, dann wird an dieser Stelle die Transformation nach Ausgabe einer Fehlermeldung schon abgebrochen. Ist die Anfrage korrekt gestellt, so muss vor der Ausführung der Transformation noch ein so genannter *ILaun-*

cher definiert werden (siehe Abb. 4.2). Wie der Name schon vermuten lässt, wird über diesen *ILauncher* die Transformation gestartet, weshalb dieser auch die jeweiligen Modelle benötigt. Die Metamodelle werden durch *IReferenceModel*s und die Ein- und Ausgabemodelle durch *IModel*s repräsentiert, welche mittels einer *ModelFactory* erzeugt werden. Außerdem ist ein *IInjector* notwendig, um die vorhandenen Metamodelle den *IReferenceModel*s und die EMF Resource zur SPARQLAS Anfrage einem *IModel* zuzuordnen. Nachdem die Transformation durch *ILauncher* erfolgreich beendet worden ist, kann über einen *IExtractor* das Ausgabemodell als SPARQL Anfrage gespeichert werden.

Hiermit ist die Ausführung einer Transformation von SPARQLAS zu SPARQL abgeschlossen und kann so von Eclipse Plugins aufgerufen werden. Als Beispiel hierzu wird der Aufruf der Transformation über einen Kontextmenüeintrag kurz beschrieben und anhand der Integration im TwoUse Toolkit verbildlicht.

Ein Plugin für einen Kontextmenüeintrag lässt sich in Eclipse leicht realisieren. Dazu erstellt man ein neues *Plug-in Project* und wählt im Verlauf des Wizards ein *Plug-in with a popup menu*. Daraufhin kann man einstellen, wie der Menüeintrag heißen und welche *Action Class* aufgerufen werden soll. Die *Action Class* ist die Klasse, in der die Transformation aufgerufen wird und von da aus weiterverarbeitet werden kann. Damit ein Kontextmenüeintrag nur bei SPARQLAS Dateien angezeigt wird, ist es sinnvoll, in der *plugin.xml* unter *Extensions* den *nameFilter* bei *objectContribution* auf „*.sparqlas“ zu setzen. Ein *Submenu* ist für dieses Beispiel nicht notwendig und kann genauso entfernt werden wie der Eintrag bei *menubarPath* in der *action*. Diese wenigen Schritte reichen aus, um ein Plugin für einen Kontextmenüeintrag umzusetzen. Wie dies im TwoUse Toolkit aussieht, verdeutlichen Abb. 4.3 und Abb. 4.4.

4.4.2 Standalone Version in Form einer API

Die Transformation von SPARQLAS zu SPARQL wird auch als Java API angeboten, so dass weitere von Eclipse unabhängige Java Programme diese Transformation verwenden können. Der Hauptunterschied zwischen der Entwicklung dieser API und dem Eclipse Plugin ist der, dass zum einen alle im Verlauf der Transformation über ein Eclipse Plugin verwendeten Plugins einzeln als Java Archiv eingebunden werden müssen. Zum anderen kann man nicht auf den Java Code zur Transformationsausführung zurückgreifen, da ATL außerhalb von Eclipse anders aufgerufen werden muss.

Zu den einzubindenden Java Archiven gehören verschiedene Plugins, die von Eclipse intern, EMF, EMFText, ANTLR und ATL verwendet werden. Zusätzlich müssen noch die Plugins zur textuellen Syntax von SPARQL und SPARQLAS hinzugefügt werden. Für die Transformation an sich reicht es aus, die zugehörige ASM Datei zugänglich zu machen. Die verschiedenen Java

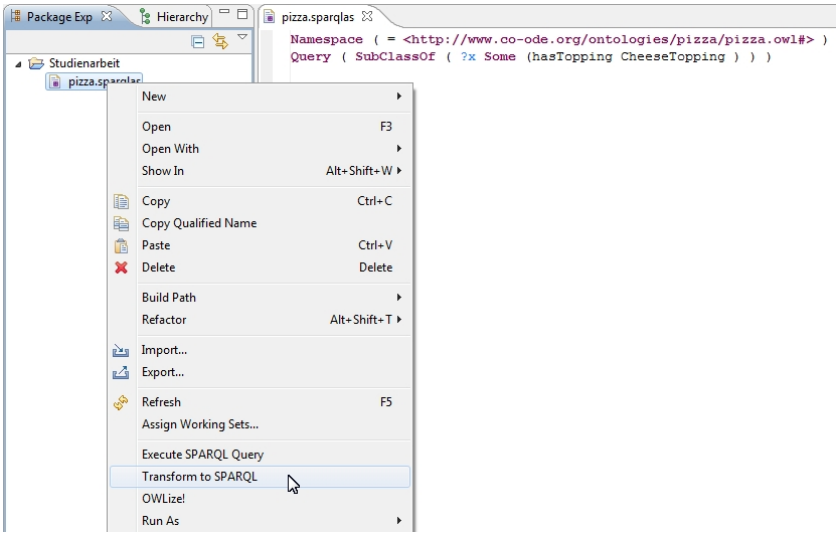

Abb. 4.3. SPARQLAS Anfrage vor Ausführung des Plugins

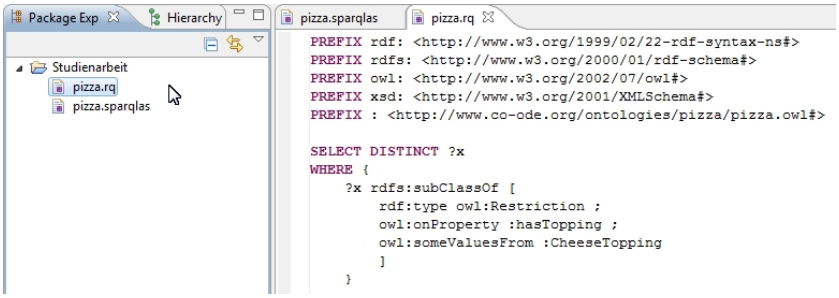

Abb. 4.4. SPARQL Anfrage nach Ausführung des Plugins

Archive werden dann in den *Build Path* aufgenommen und die ASM Datei wird anhand ihres Pfades innerhalb des Java Projekts referenziert.

Die API besteht aus zwei Java Klassen: eine Klasse als Schnittstelle, in der verschiedene Transformationsmethoden und Validierungen angeboten werden, und eine Klasse, die sich mit der Ausführung der Transformation beschäftigt.

Die einzelnen Transformationsmethoden unterscheiden sich durch die verschiedenen Ein- und Ausgaben. Einerseits lässt sich wie beim Eclipse Plugin über die Angabe einer SPARQLAS Datei eine SPARQL Datei transformieren, wobei beide Dateien in der textuellen Syntax vorliegen. Andererseits lässt sich auch eine SPARQLAS Anfrage als String direkt übergeben oder auch eine SPARQL Anfrage als String ausgeben. Da die zweite Klasse zur Transformationsausführung nur *File*s als Parameter erlaubt, muss eine SPARQLAS Anfrage in String Form in eine temporäre Datei geschrieben werden. Möchte

man nun eine SPARQL Anfrage als String zurückgeben, so muss das von der Transformation zurückgelieferte *File* über einen *FileInputStream* zu einem String konvertiert werden. Dazu wird dieser *InputStream* an einen *BufferedReader* übergeben, der als Eingabe zu einem *StringBuilder* dient. So kann dann eine SPARQL Anfrage in String Form erzeugt werden.

Weiter beinhaltet diese Klasse noch Methoden zur Validierung der SPARQLAS Datei oder der Anfrage als String. Hierfür wird auch wie beim Eclipse Plugin auf eine fehlerfreie EMF Resource kontrolliert. Damit eine solche EMF Resource verwendet werden kann, müssen vorerst die einzelnen Modelle für EMF explizit registriert werden. Diese Registrierung ist deshalb notwendig, da kein Eclipse als Grundlage dient. Daher werden sowohl die Metamodelle zu SPARQL und SPARQLAS mit entsprechender Namespace URI als auch so genannte *Factories* mit den SPARQL und SPARQLAS Dateiendungen über den vom GenModel generierten Java Code registriert. Diese *Factories* sind unerlässlich, da sonst die EMFText Parser, Lexer und Printer nicht automatisch eingreifen, wenn eine EMF Resource benötigt wird oder eine textuelle Syntax erstellt werden soll.

Die Ausführung einer ATL Transformation ist vom Ablauf her äußerst ähnlich zur Variante für Eclipse Plugins. Jedoch lassen sich die in Abb. 4.2 erwähnten Klassen nicht außerhalb von Eclipse verwenden, so dass auf grundlegende ATL Klassen zugegriffen werden muss. Diese Art der Transformationsausführung kann man auch für Eclipse Plugins verwenden. Allerdings ist sie auch etwas komplexer und nur außerhalb von Eclipse unverzichtbar, so dass der zuvor beschriebene Weg für Eclipse Plugins gewählt worden ist. Im Folgenden werden somit nun die ATL Klassen beschrieben, die in dieser API zum Einsatz kommen.

Ausgeführt wird die Transformation auch hier über einen Launcher, in dem Fall einen *AtlLauncher*. Auch diesem werden die ASM Datei und die einzelnen Modelle übergeben. Die Metamodelle und Ein- und Ausgabemodelle sind hier jeweils *ASMEMFModel*s, so dass kein Unterschied mehr wie bei *IReferenceModel* und *IModel* besteht. Die verschiedenen Modelle werden den entsprechenden *ASMEMFModel*s per *ModelLoader* zugeordnet, welcher von einem *AtlEMFModelHandler* initialisiert wird. Über diesen *ModelLoader* lässt sich zudem auch das Ausgabemodell nach Vollendung des *AtlLauncher*s abspeichern.

Auf diese Art und Weise kann eine API zur Transformation von SPARQL zu SPARQLAS entwickelt werden. Diese API kann man in Form einer ZIP Datei von der entsprechenden TwoUse Internetpräsenz[2] unter der Bezeichnung *west.twouse.standalone.sparqlas2sparql* herunterladen. Neben der API enthält die ZIP Datei zusätzlich noch ein Beispielprojekt, wie man die API aufrufen und verwenden kann.

[2] http://code.google.com/p/twouse/downloads/list

4.4.3 Webservice

Als dritte Möglichkeit die Transformation von SPARQLAS zu SPARQL auszuführen gilt der angebotene Webservice. Dieser Webservice besteht aus mehreren Komponenten, welche sich mit einem Eclipse Plugin zur Transformation erweitern lässt. Die Vorarbeit zu dem Webservice wurde im Zusammenhang des NeOn[3] Projekts geleistet und kann in einem Arbeitsergebnisdokument [WMRGnd] weiter vertieft werden. Im weiteren Verlauf wird der Aufbau dieses Webservices näher betrachtet und eine beispielhafte Verwendung durch ein PHP Skript dargelegt.

Der Webservice basiert auf einem OSGi[4] Server, speziell einem Equinox[5] Server, und eignet sich deshalb gut für die Umsetzung der Transformation als Webservice, da auch Eclipse auf dem Equinox Framework aufbaut und somit eine Einbindung von Eclipse Plugins erleichtert. Weiter wird Jetty[6] als Servlet, also als Schnittstelle zwischen Client und Server, verwendet, da Jetty mit Java entwickelt worden ist. Apache Axis2[7] wird als Webservice Container mit WSDL[8] [CMRW07] als Nachrichtensprache benutzt. Somit werden Anfragen in WSDL an den Webservice übermittelt und die Antworten in WSDL wieder zurückgeliefert.

Da diese Infrastruktur schon bereit steht, gilt es nun die Transformation als speziellen Webservice zu integrieren. Dazu müssen zum einen die bereits vorhandenen Plugins zu den Syntaxen SPARQL und SPARQLAS, das ANTLR Plugin, das Plugin zur Transformation *west.twouse.backend* und das zugehörige ATL Projekt als Java Archive der Architektur hinzugefügt werden. Zum anderen muss ein spezielles Eclipse Projekt erzeugt und hinzugefügt werden, dass als Schnittstelle zwischen Webservice und Transformation dient. Somit benötigt dieses Projekt eine Java Klasse mit einer Methode, die die Transformation aufruft und das Ergebnis an den Webservice übergibt. Außerdem ist eine Datei namens *services.xml* notwendig, in der der Webservice benannt, die soeben beschriebene Klasse angegeben und Axis2 zum Regeln des Nachrichtenflusses referenziert wird. Es sei zudem zu erwähnen, dass der Webservice so gestaltet ist, dass eine SPARQLAS Anfrage in String Form übergeben wird und die entsprechende SPARQL Anfrage als String zurückgegeben wird. Solch eine Methode ist Bestandteil des *west.twouse.backend* Plugins. Wenn nun all die beschriebenen Java Archive in die Webservice Architektur eingebunden werden, lässt sich nach dem Start des Servers der spezielle Webservice über eine URI ansprechen. Für diesen Webservice zur Transformation von SPARQLAS zu SPARQL gilt `http://twouse.west.uni-koblenz.de:8080/services/sparqlas2sparql?wsdl` als URI.

[3] NeOn: Lifecycle Support for Networked Ontologies, http://www.neon-project.org
[4] http://www.osgi.org/
[5] http://www.eclipse.org/equinox/
[6] http://jetty.codehaus.org/jetty/
[7] http://ws.apache.org/axis2/
[8] Web Services Description Language

Eine Anwendung dieses Webservices bietet das PHP Skript der Homepage http://twouse.west.uni-koblenz.de/services/sparqlas2sparql.php. Diese Seite ist so aufgebaut, dass eine SPARQLAS Anfrage über ein Textfeld eingegeben werden kann und nach Betätigung eines Buttons die SPARQL Anfrage in einem weiteren Textfeld ausgegeben wird. Zusätzlich werden noch Informationen zu SPARQLAS und einige Beispielanfragen angeboten.

Um den Webservice per PHP anzusprechen, werden so genannte SOAP Envelopes [GHM+07] zur Übertragung von WSDL eingesetzt, weshalb serverseitig die SOAP Extension für PHP aktiviert sein muss. Bevor ein SOAP Envelope gesendet werden kann, muss in PHP ein *SoapClient* mit der URI zum Webservice definiert werden. Mittels diesem *SoapClient* wird ein Request in WSDL Syntax an den Webservice gesendet. Ein solcher Request sieht wie folgt aus:

```
$request =
'<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:spar="http://sparqlas2sparql.ws.twouse.west">
  <soapenv:Header/>
  <soapenv:Body>
    <spar:sparqlas2sparql>
      <spar:sparqlasquery><![CDATA['.$query.']]>
      </spar:sparqlasquery>
    </spar:sparqlas2sparql>
  </soapenv:Body>
</soapenv:Envelope>';
```

Man erkennt hier gut, dass ein SOAP Envelope aus einem Header und einem Body besteht. Der Header kann, wie auch in diesem Beispiel, leer sein und der Body enhält die Anfrage. *sparqlas2sparql* lautet hier die zu verwendende Methode, *sparqlasquery* ist der Parameter der Methode und *$query* enthält die SPARQLAS Anfrage aus dem Texteingabefeld. Nachdem diese Anfrage versendet worden ist, erhält man als Antwort einen String eines weiteren SOAP Envelopes in folgender Art:

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns:sparqlas2sparqlResponse
      xmlns:ns="http://sparqlas2sparql.ws.twouse.west">
      <ns:return>[SPARQL Anfrage]</ns:return>
    </ns:sparqlas2sparqlResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Anstelle des Terms `[SPARQL Anfrage]` wird allerdings eine wirkliche SPARQL Anfrage zurückgeliefert, die man anhand von String Operationen aus der gesamten Antwort extrahieren kann. Dieser Teilstring wird dann durch PHP im Textausgabefeld angezeigt, was somit die Transformation von SPARQLAS zu SPARQL über einen Webservice abschließt. Nun lässt sich eine SPARQLAS Anfrage in eine SPARQL Anfrage auf drei verschiedene Weisen transformieren.

Zuvor wurde schon in Abschnitt 2.1.3 auf die Unterschiede beider Metamodelle eingegangen, allerdings noch nicht auf die Vorteile, die SPARQLAS Syntax anstatt der SPARQL Syntax für OWL-DL Anfragen zu verwenden. Daher werden im nächsten Abschnitt anhand von mehreren Testfällen SPARQLAS und SPARQL Anfragen auf Komplexität evaluiert.

4.5 Evaluation

In der Evaluation von SPARQLAS zu SPARQL Anfragen werden 60 verschiedenartige Anfragen untersucht, die alle OWL-DL Ausdrücke von SPARQLAS

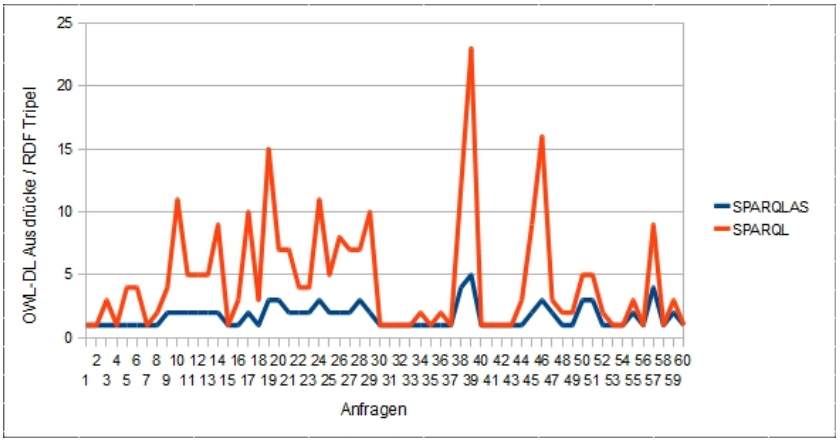

Abb. 4.5. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. OWL-DL Ausdrücke und RDF Tripel

abdecken. Jede Anfrage besteht aus einem *Atom*, d.h. dass zwar mehrere OWL-DL Ausdrücke ineinander verschachtelt vorkommen können, aber keine Anfrage zwei voneinander unabhängige Bedingungen enthält. Alle SPARQLAS und zugehörige SPARQL Anfragen werden auf zwei Weisen evaluiert. Einerseits werden die textuellen Anfragen auf Länge und andererseits die einzelnen EMF Resourcen auf Komplexität untersucht.

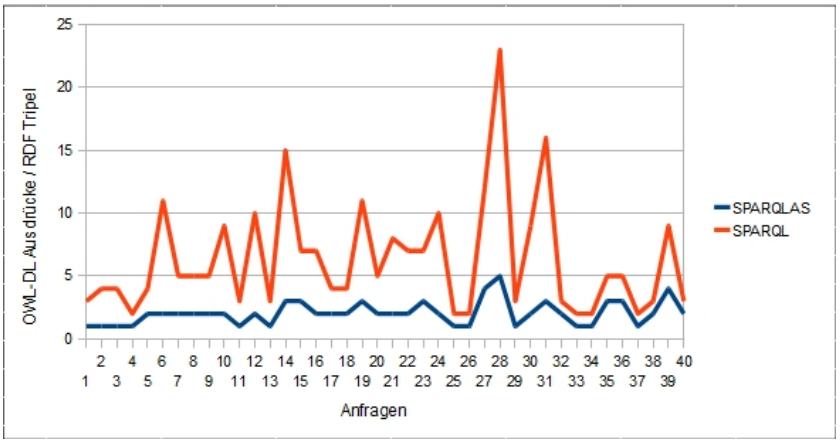

Abb. 4.6. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. OWL-DL Ausdrücke und RDF Tripel (verringerte Testmenge)

Um die textuellen Anfragen auf Länge zu untersuchen, würde sich die Anzahl an verwendeten Zeichen anbieten, was allerdings bei näherer Betrachtung nicht sinnvoll erscheinen wird. Denn ein Hauptgrund für die Verwendung der SPARQLAS anstelle der SPARQL Syntax ist die Möglichkeit, über einen OWL-DL Ausdruck mehrere RDF Tripel zusammenfassen zu können und nicht ob ein Term wie „SELECT“ oder die Anzahl an Präfixen den Unterschied ausmachen. Daher wird zu den 60 Testanfragen notiert, wie viele OWL-DL Ausdrücke die SPARQLAS Anfrage enthält und wie viele RDF Tripel daraus nach der Transformation in der SPARQL Anfrage entstehen. Wie sich SPARQLAS zu SPARQL unter diesem Gesichtspunkt verhält, erkennt man in Abb. 4.5. Man sieht hier einerseits, dass in der Mehrheit der Testfälle SPARQLAS weniger OWL-DL Ausdrücke benötigt als SPARQL RDF Tripel. Andererseits existieren auch Fälle, in denen die Verwendung von SPARQLAS keinen nennenswerten Vorteil schafft. Dies sind jedoch alles einfache Anfragen, in denen ein OWL-DL Ausdruck zu genau einem RDF Tripel übersetzt wird, wie z.B. Anfragen zu einer *Declaration* oder einer *ClassAssertion*. Wenn man nur die Fälle betrachtet, die nicht dieser Kategorie zuzuordnen sind, fallen ein Drittel der Testanfragen weg und man erhält die in Abb. 4.6 gezeigten 40 Anfragen. Dieses Diagramm verdeutlicht, dass besonders umfangreichere

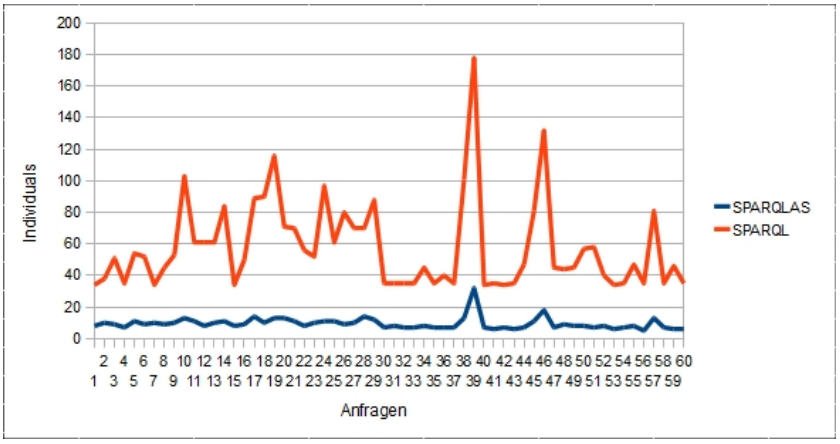

Abb. 4.7. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. Individuals

Anfragen von SPARQLAS wegen der Verwendung von OWL-DL Ausdrücken einfacher gestaltet werden können als es mit SPARQL möglich ist.

Die Komplexität einer EMF Resource lässt sich anhand der Anzahl an Instanzen der einzelnen Klassen des Metamodells ermitteln. Diese Instanzen nennt man auch Individuals und man kann ihre Anzahl im Editor in der Outline Sicht bestimmen, wie z.B. in Abb. 3.1 und Abb. 3.2. Da das Zählen der Individuals bei 60 unterschiedlich komplexen Anfragen, also insgesamt 120 SPARQLAS und SPARQL Anfragen, einiges an Zeit beanspruchen würde, wird für diese Art der Evaluation ein Plugin des TwoUse Toolkits zur Hilfe genommen. Das *OWLizer* Plugin generiert aus jeglicher auf einem Metamodell basierenden Resource eine Ontologie, welches die Beziehungen der verwendeten Klassen untereinander und zu den Individuals erstellt. Diese Ontologie kann man nun mit einem Editor für Ontologien, wie z.B. Protégé[9], öffnen und die Anzahl an Individuals direkt ablesen. Für die 60 Anfragen ergeben sich somit die in Abb. 4.7 dargestellte Anzahl an Individuals getrennt nach SPARQLAS und SPARQL. Man erkennt hier gut, dass die EMF Resourcen zu SPARQL Anfragen deutlich größer ausfallen als bei SPARQLAS Anfragen. Allerdings gilt es noch, die verringerte Testmenge zu begutachten und die SPARQL Anfragen so zu bereinigen, dass nach der Transformation entstandene, überflüssige Präfixdeklarationen entfernt werdenDiese Präfixe können bis zu 15 zusätzliche und im Durchschnitt ca. zehn Individuals ausmachen wie man in Abb. 4.8 sieht.

Die Testfälle sind nun mittels dieser Diagramme gut visualisiert worden, aber sie lassen sich auch noch über statistische Funktionen näher beschreiben.

[9] http://protege.stanford.edu/

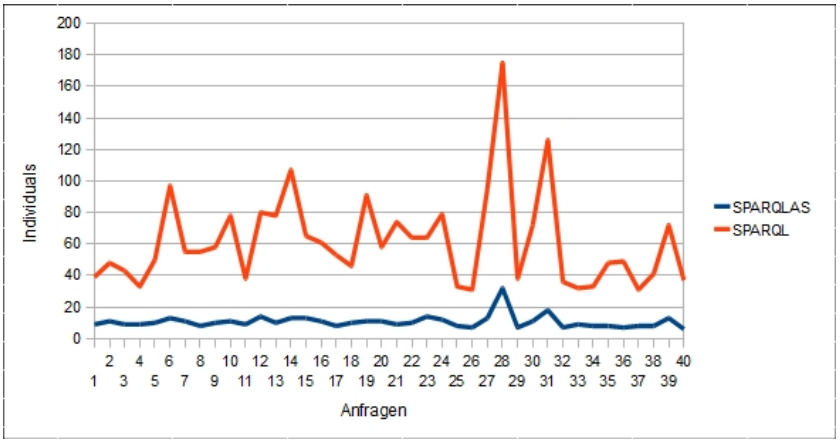

Abb. 4.8. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. Individuals (verringerte Testmenge und Präfix bereinigt)

In Tabelle 4.1 werden zunächst verschiedene Funktionen auf die Grundmenge der 60 Testfälle bzgl. der textuellen Syntax angewandt. Verglichen werden die Kategorien sowohl zur Anzahl an OWL-DL Ausdrücken und RDF Tripel als auch der prozentuale Gewinn durch die Verwendung von OWL-DL Ausdrücken. Also um dies kurz zu verdeutlichen, gilt bei einer Anfrage mit 2 OWL-DL Ausdrücken und 5 RDF Tripel ein prozentualer Gewinn von $1 - \frac{2}{5} = 60$ Prozent. Zu den besagten Funktionen gehören die Bestimmung des Minimal- und Maximalwertes, unterschiedliche Mittelwertfunktionen und die Angabe der Standardabweichung.

An den Werten in dieser ersten Tabelle fällt vor allem auf, dass die Standardabweichung äußerst hoch im Vergleich zu den verschiedenen Mittelwerten liegt. Dies deutet auf viele Werte am oberen und unteren Extrem hin, was auch Abb. 4.5 bestätigt. Aus diesem Grund werden zur weiteren Evaluation dieselben Operationen auf die verringerte Testmenge mit 40 Anfragen wiederholt und in Tabelle 4.2 aufgelistet.

	Min	**Max**	**Arithmet. Mittel**	**Geometr. Mittel**	**Median**	**Standard-abweichung**
OWL-DL	1	5	1,7	1,51	1	0,92
RDF Tripel	1	23	4,57	2,97	3	4,44
Gewinn in %	0	81,82	40,84	49,38	50	31,16

Tabelle 4.1. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. OWL-DL Ausdrücke und RDF Tripel mittels statistischer Funktionen

	Min	Max	Arithmet. Mittel	Geometr. Mittel	Median	Standard-abweichung
OWL-DL	1	5	2,08	1,88	2	0,94
RDF Tripel	2	23	6,46	5,25	5	4,48
Gewinn in %	33,33	81,82	61,55	64,28	60	14,4

Tabelle 4.2. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. OWL-DL Ausdrücke und RDF Tripel mittels statistischer Funktionen (verringerte Testmenge)

Wie zu erwarten verringert sich die Standardabweichung, besonders beim prozentualen Gewinn. Auch diese Werte bestätigen den Eindruck aus Abb. 4.6, dass SPARQLAS Anfragen kürzer gestaltet und dass speziell bei umfangreichen Anfragen dieser Vorteil zum Tragen kommt.

Dasselbe Procedere soll nun auch für die Komplexität der EMF Resourcen durchgeführt werden. Die statistischen Funktionen bleiben erhalten, nur die Kategorien ändern sich entsprechend zum Vergleich der Anzahl an Individuals. Auch für diese Evaluationsart wird ein prozentualer Gewinn angegeben, der darauf hindeutet, wie viel geringer die EMF Resourcen von SPARQLAS im Vergleich zu SPARQL ausfallen. Dazu werden in Tabelle 4.3 wiederum zuerst die gesamten 60 Testanfragen betrachtet und schließend in Tabelle 4.4 die verringerte Testmenge nach Eliminierung überflüssiger Präfixdefinitionen.

	Min	Max	Arithmet. Mittel	Geometr. Mittel	Median	Standard-abweichung
SPARQLAS	5	32	9,48	8,97	8,5	3,87
SPARQL	34	178	57,82	52,75	48,5	28,06
Gewinn in %	70,59	88,89	82,6	82,99	82,35	3,72

Tabelle 4.3. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. Individuals mittels statistischer Funktionen

	Min	Max	Arithmet. Mittel	Geometr. Mittel	Median	Standard-abweichung
SPARQLAS	6	32	10,69	10,14	10	4,24
SPARQL	31	175	62,31	56,96	55	29,08
Gewinn in %	71,87	87,91	81,74	82,19	81,94	4,1

Tabelle 4.4. Vergleich von SPARQLAS zu SPARQL Anfragen bzgl. Individuals mittels statistischer Funktionen (verringerte Testmenge und Präfix bereinigt)

Wie man in diesen beiden Tabellen sieht, ändern sich die Werte nicht so erheblich nach der Bereinigung, wie es auch zuvor in Abb. 4.7 und Abb. 4.8 zu beobachten war. Da die Standardabweichung in beiden Testmengen beim prozentualen Gewinn sehr niedrig ausfällt, deutet dies auf wenige Extrema hin. Daraus lässt sich schließen, dass die EMF Resourcen von SPARQLAS Anfragen erheblich kleiner ausfallen als im SPARQL Fall.

Nach dieser Durchführung verschiedener Evaluationsmethoden lässt sich folgern, dass nicht nur das erstellte Metamodell effizient gestaltet worden ist, sondern auch die Nutzung von OWL-DL Ausdrücken eine sichtliche Verkürzung der Anfragen erzeugen kann. Es ist allerdings auch festzuhalten, dass in einzelnen Fällen die textuelle Syntax von SPARQLAS keinen direkten Vorteil mit sich bringt. Jedoch betrifft dies ausschließlich einfache und kurze Anfragen, so dass der Einsatz von SPARQLAS sich besonders bei verschachtelten und umfangreichen Anfragen lohnt.

5

Zusammenfassung, Ausblick und Danksagung

In dieser Studienarbeit wurde verdeutlicht, wie man SPARQL Anfragen auf OWL-DL Ontologien mittels SPARQLAS übersichtlicher gestalten und vereinfachen kann. Es wurden die Unterschiede zwischen SPARQL und SPARQLAS erläutert und gezeigt, dass SPARQLAS eine echte Teilmenge zu SPARQL darstellt. Mit EMFText und ATL wurden zwei Eclipse Plugins vorgestellt, die während der Entwicklung zum Einsatz kamen. Daraufhin wurden die benötigten Metamodelle zu SPARQL und SPARQLAS in den einzelnen Elementen erklärt und verglichen. Im Hauptteil der Arbeit wurde gezeigt, wie die textuelle Syntax zu SPARQLAS erstellt wurde und wie die Transformation von SPARQLAS zu SPARQL funktioniert. Zudem wurden drei verschiedene Anwendungsmöglichkeiten dieser Transformation wiedergegeben. Zum einem wurde die Anwendung als Eclipse Plugin im Zusammenhang des TwoUse Toolkits verbildlicht, wobei auch der zugehörige Editor zum Einsatz kam. Die zweite Möglichkeit war die Anwendung per Standalone Version in Form einer Java API, welche über die TwoUse Webseite bezogen werden kann. Zuletzt wurde noch die Anwendung als Webservice geschildert, wobei einerseits der Aufbau des Webservices und andererseits eine beispielhafte Anwendung über ein PHP Skript erläutert wurde. Im Anschluss daran zeigte die Evaluation, dass SPARQLAS zwar nicht bei allen Anfragen einen direkten Nutzen bewirkte, aber besonders bei komplexen und umfangreichen Anfragen eine sichtliche Verkürzung der Syntax zu verzeichnen war. Außerdem zeigte die Evaluation, dass die EMF Resourcen zu SPARQLAS Anfragen im Vergleich zu SPARQL Anfragen effizienter ausfielen. Nach dieser kurzen Zusammenfassung der bereits durchgeführten Arbeit sollen an dieser Stelle noch einige Erweiterungsmöglichkeiten in Ausblick gestellt werden.

Zuvor wurde schon erwähnt, das SPARQLAS Anfragen eine *queryIRI* zugeordnet werden kann und über diese IRI weitere Anfragen importiert werden können. Momentan ist dies nur schmückendes Beiwerk und wird bei einer Transformation außer Acht gelassen. Möchte man dies allerdings nun implementieren, gilt es entweder die Transformation so zu erweitern, dass die importierte Anfrage beachtet wird, oder das Anfragen bearbeitende Programm

so zu verändern, dass diese Situation erkannt wird und mehrere Anfragen zusammen ausgeführt werden.

Zum jetzigen Zeitpunkt unterstützt SPARQLAS nur SELECT oder ASK Anfragen. Trotzdem ist es machbar, SPARQLAS um CONSTRUCT oder DESCRIBE Anfragen zu ergänzen. Dazu sind allerdings mehrere Schritte notwendig. Zuerst muss das Metamodell angepasst werden, da so keine Unterscheidung zwischen den vier Typen möglich ist und auch weitere Elemente notwendig sein werden. Außerdem muss die Syntax erweitert werden, da nur ein Term „Query“ für vier Varianten nicht ausreichend ist und bei genauerer Betrachtung evtl. noch weitere Syntaxänderungen nötig werden.

SPARQL ist nicht nur wegen den vier Anfragentypen mächtiger, sondern auch wegen weiterer Verfeinerungsmöglichkeiten bzgl. der Anfrage. Diese weiteren Bedingungen, Filterungen oder die Möglichkeit auf Ordnen und Beschränkung der Ergebnismenge kann auch für SPARQLAS übernommen werden, wenn das Metamodell und die Syntax dementsprechend ausgebaut wird. Bei solchen Erweiterungen ist allerdings ein Kompromiss einzugehen, da SPARQLAS das Ziel verfolgt, Anfragen übersichtlich, einfach und kurz zu halten, was auch für zukünftige Entwicklungen berücksichtigt werden soll.

Diese Studienarbeit wird innerhalb des TwoUse[1] Projekts realisiert, was wiederum Teil des EU geförderten MOST[2] Projekts ist. Die Anwendung des Eclipse Plugins findet man im TwoUse Toolkit.

An dieser Stelle soll der Dank an Fernando Silva Parreiras für jegliche Hilfe Rund um das TwoUse Toolkit und Christian Wende von der Technischen Universität Dresden für die Unterstützung bei der Verwendung des EMFText Plugins ausgedrückt werden.

[1] Transforming and Weaving Ontologies and UML in Software Engineering, http://code.google.com/p/twouse/

[2] Marrying Ontology and Software Technology, http://www.most-project.eu/

Literaturverzeichnis

Baa05. Franz Baader. *The description logic handbook: Theory, implementation, and applications*. Cambridge Univ. Press, Cambridge, 2005.

BB08. Dave Beckett and Jeen Broekstra. SPARQL Query Results XML Format, 16.01.2008. http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/, Letzter Zugriff: 10.03.2010.

BLFM05. T. Berners-Lee, R. Fielding, and L. Masinter. Request for Comments: 3986: Uniform Resource Identifier (URI): Generic Syntax, 01.2005. http://tools.ietf.org/html/rfc3986, Letzter Zugriff: 10.03.2010.

CFT08. Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL Protocol for RDF, 16.01.2008. http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/, Letzter Zugriff: 10.03.2010.

CMRW07. Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 26.06.2007. http://www.w3.org/TR/wsdl20/, Letzter Zugriff: 10.03.2010.

DHS+07. Nick Drummond, Matthew Horridge, Robert Stevens, Chris Wroe, and Sandra Sampaio. Pizza Ontology, 12.02.2007. http://www.co-ode.org/ontologies/pizza/2007/02/12/, Letzter Zugriff: 12.04.2010.

DS05. M. Duerst and M. Suignard. Request for Comments: 3987: Internationalized Resource Identifiers (IRIs), 01.2005. http://tools.ietf.org/html/rfc3987, Letzter Zugriff: 10.03.2010.

Ecl10a. Eclipsepedia. ATL/Developer Guide, 09.02.2010. http://wiki.eclipse.org/ATL/Developer_Guide, Letzter Zugriff: 10.04.2010.

Ecl10b. Eclipsepedia. M2M/Atlas Transformation Language (ATL), 19.02.2010. http://wiki.eclipse.org/ATL, Letzter Zugriff: 10.03.2010.

EMF09a. EMFText. EMFText Getting Started Screencast, 18.06.2009. http://www.emftext.org/index.php/EMFText_Getting_Started_Screencast, Letzter Zugriff: 10.03.2010.

EMF09b. EMFText. EMFText Overview, 18.06.2009. http://www.emftext.org/index.php/EMFText_Overview, Letzter Zugriff: 10.04.2010.

EMF10. EMFText. EMFText Concrete Syntax Zoo SPARQL, 20.02.2010. http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_SPARQL, Letzter Zugriff: 10.03.2010.

GHM+07. Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 27.04.2007. http://www.w3.org/TR/soap12-part1/, Letzter Zugriff: 10.03.2010.

Gro06. ATLAS Group. The ATLAS Transformation Language (ATL) project: Transforming models with ATL, 2006. http://www.atl-pro.com/resource/ATL_Poster.pdf, Letzter Zugriff: 10.04.2010.

HPS09. Matthew Horridge and Peter F. Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax, 27.10.2009. http://www.w3.org/TR/2009/NOTE-owl2-manchester-syntax-20091027/, Letzter Zugriff: 10.03.2010.

Int96. International Organization for Standardization. ISO/IEC 14977:1996: Information technology - Syntactic metalanguage - Extended BNF, 15.12.1996. http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html, Letzter Zugriff: 10.03.2010.

MPSP09. Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax, 27.10.2009. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/, Letzter Zugriff: 10.03.2010.

Obj04. Object Management Group. HUTN, 01.08.2004. http://www.omg.org/spec/HUTN/1.0/, Letzter Zugriff: 10.03.2010.

Obj06. Object Management Group. OCL 2.0, 01.05.2006. http://www.omg.org/spec/OCL/2.0/, Letzter Zugriff: 10.03.2010.

Obj07. Object Management Group. XML Metadata Interchange, 02.12.2007. http://www.omg.org/technology/documents/formal/xmi.htm, Letzter Zugriff: 10.03.2010.

PS08. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, 16.01.2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, Letzter Zugriff: 10.03.2010.

PSH04. Peter F. Patel-Schneider and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax: Section 4. Mapping to RDF Graphs, 06.02.2004. http://www.w3.org/TR/owl-semantics/mapping.html, Letzter Zugriff: 10.03.2010.

PSM09. Peter F. Patel-Schneider and Boris Motik. OWL 2 Web Ontology Language Mapping to RDF Graphs, 28.10.2009. http://www.w3.org/TR/2009/REC-owl2-mapping-to-rdf-20091027/, Letzter Zugriff: 10.03.2010.

SBPM09. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. The eclipse series. Addison-Wesley, Upper Saddle River, NJ, 2nd ed., rev. and updated. edition, 2009.

SP07. Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *3rd OWL: Experiences and Directions Workshop (OWLED2007)*, 2007. PDF Version unter http://clarkparsia.com/files/pdf/sparqldl.pdf, Letzter Zugriff: 10.03.2010.

SWM04. Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL Web Ontology Language Guide, 10.02.2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/, Letzter Zugriff: 10.03.2010.

WMRGnd. Walter Waterfeld, Diana Maynard, Ian Roberts, and Michael Gesmann. Deliverable D6.4.2 NeOn core infrastructure services, 2010 (in Kürze erscheinend).