Institut für Softwaretechnik

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Vergleich der Effizienz von Anfragen an OWL-Ontologien und TGraphen

Studienarbeit

vorgelegt am 8. November 2011 von

Olga Haubrich

Betreuer:
Prof. Dr. Jürgen Ebert
Hannes Schwarz

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.

Koblenz, den 8. November 2011 . . . . . .

Inhaltsverzeichnis

Kapitel 1

Ziel der Studienarbeit

Im Rahmen dieser Studienarbeit soll die Effizienz von Anfragen an OWL-Ontologien und Anfragen an TGraphen untersucht werden. Dabei sollen als Grundlage eine oder mehrere OWL-Ontologien dienen, die über die zur Verfügung stehende API in TGraphen transformiert werden. Die OWL-Ontologien sollen auf zwei verschiedene Weisen in TGraphen transformiert werden. Für Transformationen werden die zwei an der Universität Koblenz-Landau entwickelten Transformationsverfahren Schema-Aware und Simple Mapping verwendet [SE10].

Für Anfragen an OWL-Ontologien wird die Anfragesprache SPARQL 1.0 verwendet. Es soll aber auch die letztere Version SPARQL 1.1 in der Arbeit beschrieben werden. Die beiden Versionen SPARQL 1.0 und SPARQL 1.1 sollen auf Gemeinsamkeiten und Unterschiede untersucht werden.

Anfragen an TGraphen werden mit der Sprache GReQL gestellt. SPARQL-Anfragen[1] werden entsprechend der Transformation der OWL-Ontologien in GReQL-Anfragen transformiert. Wird zum Beispiel die OWL-Ontologie über Schema-Aware Mapping in den TGraphen transformiert, wird auch gleichzeitig die SPARQL-Anfrage über Schema-Aware Mapping in die GReQL-Anfrage überführt. Dementsprechend werden die SPARQL-Anfragen auch wie die OWL-Ontologie auf zwei verschiedene Arten in GReQL-Anfragen transformiert.

Gleichzeitig können die transformierten GReQL-Anfragen über ein an der Universität Koblenz-Landau entwickeltes Modifikationsverfahren modifiziert werden [Sch11]. Das Modifikationsverfahren soll die transformierten GReQL-Anfragen bei Vorhandensein bestimmter Eigenschaften so modifizieren, dass auf den Einsatz eines Reasoners verzichtet werden kann. Mit dem Modifikationsverfahren können die Property-Eigenschaften „transitiv“, „symmetrisch“, „äquivalent zu“, „invers zu“ und die `subPropertyOf`-Beziehungen zwischen Properties berücksichtigt werden.

Das Ziel dieser Studienarbeit ist, diese Verfahren auf ihre Effizienz zu prüfen und mit der Effizienz der unmittelbaren Ausführung von SPARQL-Anfragen an OWL-Ontologien zu vergleichen. Dabei sollen die Laufzeit einer Anfrage und der benötigte Speicherplatz zur Ausführung einer Anfrage ermittelt werden.

Im Mittelpunkt der Evaluierung steht das Modifikationsverfahren. Es wird vermutet, dass durch das Modifikationsverfahren und den gleichzeitigen Verzicht auf einen Reasoner die Ausführung von Anfragen effizienter sein kann als mit dem Einsatz eines Reasoners.

Es sollen SPARQL-Anfragen entwickelt werden, die Properties mit den oben genannten Eigenschaften verwenden. Es sind also Property-Eigenschaften, die vom Modifikationsverfahren behandelt werden können.

[1] Mit den zur Verfügung stehenden APIs können zurzeit nicht alle mögliche Sprachelemente transformiert werden. Es können keine alternativen (UNION) oder optionalen (OPTIONAL) Graphmuster und keine Filter-Bedingungen behandelt werden.

Insgesamt ergeben sich die in Tabelle 1.1 aufgelisteten Testfälle.

	Anfrage		**Datenstruktur**	
Nr	**Anfrageart**	**Transformationsart**	**Art der Datenstruktur**	**Transformationsart**
1	direkte SPARQL-Anfrage	keine	OWL-Ontologie	keine, Ausführung der OWL-Ontologie mit Reasoning
2	transformierte modifizierte GReQL-Anfrage	Schema-Aware Mapping	TGraph	Schema-Aware Mapping ohne Reasoning
3	transformierte GReQL-Anfrage	Schema-Aware Mapping	TGraph	Schema-Aware Mapping mit Reasoning
4	transformierte modifizierte GReQL-Anfrage	Simple Mapping	TGraph	Simple Mapping ohne Reasoning
5	transformierte GReQL-Anfrage	Simple Mapping	TGraph	Simple Mapping mit Reasoning

Tabelle 1.1: Testfälle

Unabhängig von oben genannten Verfahren soll die Transformation von GReQL-Anfragen in SPARQL-Anfragen auf ihre Effizienz untersucht werden. Die Transformation wird über die API `greql2sparql` [Sch11] realisiert. Dabei soll die Transformationszeit von GReQL-Anfragen in SPARQL-Anfragen unterschiedlicher Komplexität ermittelt werden.

Kapitel 2

Einführung in RDF

Das Resource Description Framework (RDF) - eine Empfehlung des W3C - ist eine formale Sprache zur Beschreibung von Ressourcen im Web [MM04].

RDF-Dokumente können mit der RDF/XML- oder mit der für Menschen besser lesbaren Turtle-Syntax erstellt werden. Zum besseren Verständnis wird in der Studienarbeit Turtle-Syntax verwendet.

Das Listing 2.1 demonstriert ein RDF-Dokument im Turtle-Format. Das Dokument beginnt mit dem Schlüsselwort `@prefix`; damit werden Abkürzungen für Namensräume deklariert. Nach dem Schlüsselwort `@prefix` folgt ein beliebiger Bezeichner, der mit einem Doppelpunkt enden muss. Anschließend wird in eckigen Klammern eine URI[1] geschrieben. Der Bezeichner `pe` ist also eine Abkürzung für die URI `http://www.uni.de/person/`. Der Bezeichner `pe` kann im ganzen Dokument an der Stelle der URI `http://www.uni.de/person/` verwendet werden. RDF-Dokumente in Turtle-Format können auch ohne solche Abkürzungen erstellt werden. Der Zweck der Abkürzungen ist die Lesbarkeit eines RDF-Dokuments zu erleichtern. Nach der Deklaration von Namensräumen folgen Tripel, die voneinander mit einem Punkt getrennt werden.

2.1 RDF-Tripel

Daten werden in RDF-Dokumenten mit Hilfe von RDF-Tripeln modelliert. Ein RDF-Tripel besteht aus einem Subjekt, einem Prädikat und einem Objekt (die Reihenfolge ist dabei wichtig). Ein Subjekt ist über ein Prädikat mit einem Objekt verbunden. Die Zeile 6 im Listing 2.1 enthält das Tripel `pe:student1 pe:name pe:Alice`, dabei ist `student1` ein Subjekt, `name` ein Prädikat und Alice ist ein Objekt. Ein Tripel ist eine Aussage. In dem Beispiel sagt es folgendes aus: „`Student1` heißt `Alice`“.

```
1  @prefix pe: <http://www.uni.de/person/>.
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
3  @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
4
5  pe:student1 rdf:type pe:student.
6  pe:student1 pe:name  pe:Alice.
7  pe:student1 pe:matrNumber "1020".
8  pe:student1 pe:age "20"^^xsd:int.
9  pe:student1 pe:phone  _:p1.
10 _:p1        pe:homePhone "0234".
11 _:p1        pe:mobilePhone "0176".
```

[1] URI (Uniform Resource Identifier) ist eine Zeichenfolge, die einer abstrakten oder physischen Ressource identifiziert.

```
pe:student1  pe:friends  pe:student2.
pe:student1  pe:friends  pe:student3.

pe:student2 rdf:type pe:student.
pe:student2 pe:name  pe:Tom.
pe:student2 pe:matrNumber "1001".
pe:student2 pe:age "22"^^xsd:int.
pe:student2 pe:libraryCard "1718".
pe:student2 pe:phone  _:p2.
_:p2        pe:homePhone "0634".
pe:student2  pe:friends  pe:student3.

pe:student3 rdf:type pe:student.
pe:student3 pe:name  pe:Bob.
pe:student3 pe:matrNumber "1011".
pe:student3 pe:age "20"^^xsd:int.
pe:student3 pe:libraryCard "1756".
pe:student3 pe:phone  _:p3.
_:p3        pe:homePhone "0614".
```

Listing 2.1: RDF-Dokument im Turtle-Format

2.2 RDF-Graph

Ein RDF-Dokument beschreibt einen gerichteten Graphen: ein Subjekt ist ein Anfangsknoten und ein Objekt ein Endknoten eines Prädikats, welches durch eine Kante dargestellt wird [PHRS08]. Das Tripel `pe:student1 pe:name pe:Alice` ist in der Abbildung 2.1 graphisch dargestellt. `pe:student1` und `pe:Alice` sind Knoten, `pe:name` ist eine Kante, die die beiden Knoten verbindet.

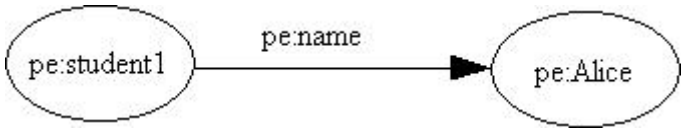


Abbildung 2.1: RDF-Graph

2.3 RDF-Terme

Es werden drei Typen von RDF-Termen unterschieden: RDF-Ressourcen, Literale und Leere Knoten (Blank Nodes).

2.3.1 RDF-Ressourcen

Alle Objekte, die in RDF beschrieben werden nennt man Ressourcen [BG04]. Die Namen von Ressourcen müssen eindeutig sein, deswegen werden sie durch URIs identifiziert.

2.3.2 Literale

Datenwerte in RDF werden Literale genannt. Nur Objekte können in RDF-Tripel Literale sein. Ein Literal-Objekt hat keine weiteren Verbindungen, genauer es kann nicht ein Ausgangspunkt einer Kante sein. Es

wird zwischen getypten und ungetypten Literalen (Plain literals) unterschieden. Ein getyptes Literal besteht aus zwei Teilen: einem String und einem Datentyp. Beispielsweise repräsentiert „20"^^xsd:int die Zahl 20 vom Datentyp Integer. Ungetypte Literale werden ohne Datentyp geschrieben, sie entsprechen dem getypten String-Literal. Die ungetypten Literale können mit Sprachangaben (Language Tags) versehen werden. Eine Sprachangabe definiert, in welcher natürlichen Sprache ein Literal geschrieben wurde.

2.3.3 Leere Knoten (Blank Nodes)

Mit leeren Knoten werden anonyme Subjekte oder Objekte deklariert. Leere Knoten werden nicht durch eine URI, sondern durch eine ID (Identifikator) bezeichnet. In der Turtle-Syntax werden leere Knoten durch Verwendung eines Unterstriches an der Stelle einer Namensraum-Abkürzung dargestellt. Zum Beispiel _:p1 ist ein leerer Knoten mit der ID p1.

2.4 RDF-Schema

RDF-Schema ist eine Erweiterung von RDF. Mit RDF-Schema wird das Vokabular für eine konkrete Domäne festgelegt. Im RDF-Schema können Klassenhierarchie und Untermengen von Beziehungen beschrieben werden, was die Modellierung einfacher Ontologien ermöglicht.

2.4.1 Klassen und Instanzen

Die Gruppen von Ressourcen bilden Klassen in RDF. Instanzen einer Klasse repräsentieren Elemente einer Klasse.

Im RDF-Schema stehen folgende vordefinierte Klassen zur Verfügung: rdfs:Resource, rdfs:Class, rdfs:Literal und rdfs:Datatype. Das Präfix rdfs sei dabei eine Abkürzung der URI <http://www.w3.org/2000/01/rdf-schema#>.

Alle RDF-Ressourcen sind Instanzen der Klasse rdfs:Resource.

Mit dem vordefinierten Property rdf:type aus RDF und rdfs:Class können bestimmte Klassen definiert werden, zum Beispiel pe:Student rdf:type rdfs:Class, damit wird die Ressource pe:Student als eine Klasse deklariert. Mit rdf:type werden auch die Instanzen einer Klasse festgelegt. Durch pe:Alice rdf:type pe:Student wird beispielsweise pe:Alice der Klasse pe:Student zugeordnet, damit ist pe:Alice eine Instanz der Klasse pe:Student.

rdfs:Literal ist die Klasse aller Literalwerte und rdfs:Datatype die Klasse aller Datentypen.

2.4.2 Properties

Properties werden Bezeichner für Prädikaten genannt, also Ressourcen die anstelle der Prädikaten stehen. Prädikaten sind damit Vorkommen von Properties.

Die wichtigsten neuen Properties in RDF-Schema sind rdfs:subClassOf, rdfs:domain, rdfs:range und rdfs:subPropertyOf.

Die Property rdfs:subClassOf definiert Unterklassen einer Klasse. Zum Beispiel C2 rdfs:subClassOf C1 heißt C2 ist eine Unterklasse der Klasse C1.

Mit der Property `rdfs:subPropertyOf` werden Unterproperties einer Property definiert. Zum Beispiel `P2 rdfs:subPropertyOf P1` heißt `P2` ist eine Unterproperty von Property `P1`.

Die Property `rdfs:domain` definiert die Klasse von Subjekten einer Property. Zum Beispiel `P rdfs:domain S`, wobei `S` als ein Subjekt der Property `P` festgelegt wird.

Mit der Property `rdfs:range` wird die Klasse von Objekten einer Property definiert. Beispielsweise das Tripel `P rdfs:range O` legt fest, dass alle Objekte in Tripeln mit `P`, Instanzen von `O` sind.

Kapitel 3

Einführung in OWL 2

Die Web Ontology Language (OWL) ist eine formale Sprache für Beschreibungen von Ontologien. Mit OWL-Ontologien werden Ressourcen (Objekte) einer Domäne und deren Beziehungen zueinander formal beschrieben. Dabei ist es wichtig die Semantik der beschriebenen Beziehungen für Computer lesbar („verständlich“) zu machen. OWL ist eine für das Semantic Web entwickelte Ontologiesprache.

Die erste Version von OWL wurde im Februar 2004 vom W3C als Ontologiesprache standardisiert [PSHH04]. Die aktuelle erweiterte Version von OWL ist OWL 2, die seit Oktober 2009 verfügbar ist [MPSP09].

In Anlehnung an [MPSP09] wird in diesem Kapitel OWL 2 erläutert.

OWL 2-Dokumente können mit fünf verschiedenen Syntaxen erstellt werden. Es sind die Manchester-, RDF/XML-, OWL/XML-, Turtle-Syntax und funktionale Syntax.

Die funktionale Syntax erleichtert die Einsicht in die formale Struktur der OWL 2-Ontologien. Mit der funktionalen Syntax werden die OWL 2-Ontologien in kompakter Form beschrieben. Aus diesen Gründen wird in dieser Studienarbeit für Beispiele und Erläuterungen die funktionale Syntax verwendet.

OWL 2-Ontologien bestehen aus drei verschiedenen syntaktischen Kategorien: Entitäten, Ausdrücke (Expressions) und Axiome.

3.1 Entitäten

Die Entitäten sind grundlegende Bausteine von OWL 2-Ontologien. Zu den Entitäten zählt man Klassen, Properties (auch Rollen genannt) und Individuen.

3.1.1 Klassen und Individuen

Individuen repräsentieren Objekte der realen Welt. Klassen sind Kategorien von den Individuen. Die Individuen werden auch Instanzen einer Klasse genannt. Zum Beispiel gibt es eine Klasse `Person`, die eine Instanz `Alice` hat. In OWL 2 gibt es die zwei vordefinierten Klassen `Thing` und `Nothing`. Die `Thing`-Klasse ist eine Klasse, die alle Individuen enthält. Sie ist sozusagen eine Superklasse von allen Klassen, die von Benutzern definiert werden. Die `Nothing`-Klasse ist eine leere Klasse, die keine Elemente enthält.

3.1.2 Properties

In OWL 2 gibt es die zwei Arten von Properties Objekt- und Daten-Properties. Eine Objekt-Property verbindet zwei Individuen miteinander. Eine Daten-Property ist eine Verbindung zwischen einem Individuum und einem Literal (Datenwert). Ähnlich wie in RDF/RDF-Schema werden die Beziehungen von Elementen als Tripel: Individuum-Property-Individuum bzw. Individuum-Property-Literal dargestellt. Jede OWL 2-Ontologie kann in einen RDF-Graphen transformiert werden, dabei sind Individuen und Literale Knoten und Properties gerichtete Kanten.

3.1.3 Literale

Literale sind Datenwerte in OWL 2. Ein Literal besteht aus zwei Teilen: einem String und einem Datentyp. Zum Beispiel das Literal „`17"^^xsd:integer` repräsentiert die Zahl 17 von Datentyp Integer.

3.1.4 Ausdrücke

In OWL 2 können Entitäten mit Hilfe von Konstruktoren in einen Ausdruck zusammengesetzt werden. Die Konstruktoren sind Bezeichner von den Ausdrücken, zum Beispiel: `InverseObjectProperty`, `ObjectUnionOf`. Es wird zwischen Propertyausdrücken und Klassenausdrücken unterschieden.

Propertyausdrücke

Die Propertyausdrücke sind in Objekt-Propertyausdrücken und Daten-Propertyausdrücken unterteilt. Es gibt die zwei Arten von Objekt-Propertyausdrücken Objekt-Property und inverse Objekt-Property. Die Grammatik der Objekt-Propertyausdrücke sieht in BNF-Notation wie folgt aus [MPSP09]:

```
ObjectPropertyExpression := ObjectProperty | InverseObjectProperty
```

Die Daten-Property ist der einzige Daten-Propertyausdruck:

```
DataPropertyExpression := DataProperty
```

Klassenausdrücke

Im Gegensatz zu Propertyausdrücken sind Klassenausdrücke zahlreich und komplex. Die folgende Grammatik beschreibt die Klassenausdrücke:

```
ClassExpression :=
     Class |
     ObjectIntersectionOf | ObjectUnionOf | ObjectComplementOf |
     ObjectOneOf | ObjectSomeValuesFrom | ObjectAllValuesFrom |
     ObjectHasValue | ObjectHasSelf | ObjectMinCardinality |
     ObjectMaxCardinality | ObjectExactCardinality |
     DataSomeValuesFrom | DataAllValuesFrom | DataHasValue |
     DataMinCardinality | DataMaxCardinality | DataExactCardinality
```

In OWL 2 werden Klassen und Propertyausdrücke benutzt, um Klassenausdrücke zu beschreiben. Mit den Klassenausdrücken kann man beispielsweise eine Menge von Individuen beschreiben, die besondere Charakteristiken aufweisen. Die Klassenausdrücke können wie folgt aufgeteilt werden: logische Verknüpfungen, Aufzählung von Individuen, Kardinalitätseinschränkung von Properties und sonstige Einschränkungen von Properties.

Die Konjunktion (ObjectIntersectionOf), Disjunktion (ObjectUnionOf) und Negation (ObjectComplementOf) sind mögliche logische Verknüpfungen in OWL 2.

Der Klassenausdruck ObjectIntersectionOf($CE_1 \ldots CE_n$) besteht mindestens aus einem Klassenausdruck $CE_{i,i=1..n}$ und umfasst alle Individuen, die gleichzeitig Instanzen von allen Klassenausdrücken CE_i sind. Der Klassenausdruck ObjectUnionOf($CE_1 \ldots CE_n$) besteht auch aus einem oder mehreren Klassenausdrücken. Das Ergebnis dieses Ausdrucks sind Individuen, die Instanzen von mindestens einem Klassenausdruck CE_i sind.

OWL 2-Ontologien basieren auf der sogenannten Open World Assumption (Offene-Welt-Annahme). Bei der Open World Assumption wird alles, was nicht modelliert wurde, als unbekannt interpretiert. Das heißt die Nicht-Existenz von Fakten muss explizit definiert werden.

Mit dem Klassenausdruck ObjectComplementOf(CE) kann die Negation des Klassenausdrucks CE explizit definiert werden. Mit dem Axiom - auf Axiome wird später eingegangen - ClassAssertion(a:Fish a:Nemo) wird der Klasse Fish die Instanz Nemo zugefügt. Wenn man die Frage stellt, ob Nemo eine Katze ist, wird die Frage nicht mit Nein beantwortet, da es nicht explizit definiert wurde, dass die Instanz Nemo keine Instanz der Klasse Cat ist. Fügt man dagegen die Aussage ClassAssertion(ObjectComplementOf(a:Cat) a:Nemo) hinzu, wird die Frage „Ist Nemo eine Katze?“mit Nein beantwortet.

Der Klassenausdruck ObjectOneOf(a_1 ... a_n) besteht aus einem oder mehreren Individuen. Mit dem Ausdruck kann man zum Beispiel einer Klasse eine Liste von Individuen zuweisen, damit wird die Klasse als geschlossen definiert. Die geschlossene Klasse kann wie folgt definiert werden: EquivalentClasses(a:GreenFamilyMember ObjectOneOf(a:Mary a:Bill a:Tom a:Alice)), das heißt die Klasse GreenFamilyMember besteht nur aus vier verschieden Instanzen Mary, Bill, Tom und Alice.

ObjectSomeValuesFrom, ObjectAllValuesFrom, ObjectHasValue und ObjectHasSelf sind Konstruktoren, mit denen Ausdrücke über die Einschränkungen von Objekt-Property erstellt werden.

Der Klassenausdruck ObjectSomeValuesFrom(OPE CE) ist eine Existenzquantifikation über eine Objekt-Property und beinhaltet Individuen, die mindestens einmal über die Objekt-Property OPE mit den Instanzen des Klassenausdrucks CE verbunden sind.

Der Klassenausdruck ObjectAllValuesFrom(OPE CE) steht für die Allquantifikation über eine Objekt-Property und umfasst Individuen, die über die Objekt-Property OPE nur mit Instanzen des Klassenausdrucks CE verbunden sind. Hier soll auch betont werden, dass der Klassenausdruck ObjectAllValuesFrom auch Individuen enthält, die keine Beziehungen über die Objekt-Property OPE haben.

Der Klassenausdruck ObjectHasValue(OPE a) besteht aus einem Objekt-Propertyausdruck OPE und einem Individuum a. Der Klassenausdruck umfasst alle Individuen, die über den Objekt-Propertyausdruck OPE mit Individuum a verbunden sind.

Der Klassenausdruck ObjectHasSelf(OPE) besteht aus einem Objekt-Propertyausdruck OPE und enthält alle Individuen, die mit sich selbst über OPE verbunden sind.

In OWL 2 ist es möglich die Kardinalität von Objekt-Properties festzulegen. Mit den Klassenausdrücken ObjectMaxCardinality, ObjectMinCardinality und ObjectExactCardinality kann maximale, minimale und exakte Kardinalität von Objekt-Properties bestimmt werden. Die Klassenausdrücke ObjectMaxCardinality(n OPE CE), ObjectSomeValuesFrom(OPE CE) und ObjectExactCardinality(n OPE CE) bestehen jeweils aus einer natürlichen Zahl n, einem Objekt-Propertyausdruck OPE und einem Klassenausdruck CE.

Der Ausdruck ObjectMaxCardinality(n OPE CE) umfasst alle Individuen, die über OPE mit höchstens n verschiedenen Instanzen von CE verbunden sind.

Der Ausdruck `ObjectMinCardinality(n OPE CE)` umfasst alle Individuen, die über `OPE` mit mindestens n verschiedenen Instanzen von `CE` verbunden sind.

Der Ausdruck `ObjectExactCardinality(n OPE CE)` enthält alle Individuen, die über `OPE` mit genau n verschiedenen Instanzen von `CE` verbunden sind.

Mit `ObjectMinCardinality` kann beispielsweise Folgendes beschrieben werden: `ObjectMinCardinality(1 a: hasChild a:Person)`- „Jemand ist ein Elternteil, wenn er mindestens ein Kind hat„.

Auf ähnliche Weise werden maximale, minimale und exakte Kardinalität von Daten-Properties entsprechend mit den Klassenausdrücken `DataMaxCardinality`, `DataMinCardinality` und `DataExactCardinality` definiert.

3.2 Axiome

Der wichtigste Bestandteil von OWL 2 sind Axiome. Axiome sind Aussagen (Statement) innerhalb einer Ontologie, die immer erfüllt sind. OWL 2 verfügt über zahlreiche Axiome. Die wichtigsten Axiome sind Axiome über Klassen, Axiome über Properties und Axiome über Individuen.

3.2.1 Axiome über Klassen

In OWL 2 gibt es die vier Axiome über Klassen `SubClassOf`, `EquivalentClasses`, `DisjointClasses` und `DisjointUnion`.

Mit dem Axiom `SubClassOf` wird die Hierarchie von Klassen gebildet. Beispielsweise `SubClassOf(a:Baby a:Person)` ist ein Axiom über Klassen, das Folgendes aussagt: "Jedes Baby ist eine Person„. Die Klasse `Baby` ist eine Unterklasse der Klasse `Person`.

Das Axiom `EquivalentClasses(`CE_1 `...` CE_n`)` besteht aus zwei oder mehreren Klassenausdrücken und die Klassenausdrücke sind äquivalent zueinander.

Das Axiom `DisjointClasses(`CE_1 `...` CE_n`)` besteht aus mindestens zwei Klassenausdrücken, die paarweise disjunkt sind. Die Instanzen eines Klassenausdrucks können nicht die Instanzen des anderen Klassenausdrucks sein.

Das Axiom `DisjointUnion(C `CE_1 `...` CE_n`)` besteht aus einer Klasse und mindestens zwei Klassenausdrücken. Die Klasse `C` entspricht der Vereinigung der Klassenausdrücke $CE_{i,\ i=1..n}$ und die Klassenausdrücke $CE_{i,\ i=1..n}$ sind paarweise disjunkt. Ein Beispiel ist das Axiom `DisjointUnion(a:Person a:Woman a:Man)`: die Klasse `Person` besteht aus den Instanzen der Klassen `Woman` und `Man`, dabei sind die Klassen `Woman` und `Man` disjunkt. Das heißt die Instanzen der Klasse `Woman` können nicht die Instanzen der Klasse `Man` sein.

3.2.2 Axiome über Properties

In OWL 2 ist es möglich mit dem Axiom `SubObjectPropertyOf` eine Hierarchie von Properties zu bilden. Es gibt zwei Formen des Axioms `SubObjectPropertyOf`, eine einfache und eine komplexe Form.

Das einfache Axiom `SubObjectPropertyOf(`OPE_1 OPE_2`)` besteht aus zwei Objekt-Propertyausdrücken OPE_1 und OPE_2 und sagt aus, dass der Objekt-Propertyausdruck OPE_1 eine Unterproperty von OPE_2 ist.

Wenn $\texttt{OPE}_1$ das Individuum $\texttt{a}_1$ mit dem Individuum $\texttt{a}_2$ verbindet, dann verbindet auch $\texttt{OPE}_2$ das Individuum $\texttt{a}_1$ mit dem Individuum $\texttt{a}_2$.

Die zweite Variante des Axioms, `SubObjectPropertyOf(ObjectPropertyChain(`$\texttt{OPE}_1$ `...` $\texttt{OPE}_n$`)` `OPE)`, besteht aus einer Verkettung von Properties - eine neue Eigenschaft in OWL 2 - und einer Property. Dieses Axiom kann wie folgt erklärt werden: Verbindet eine Sequenz der Objekt-Propertyausdrücke $\texttt{OPE}_{i,\ i=1..n}$ das Individuum $\texttt{a}_1$ mit dem Individuum $\texttt{a}_2$, dann verbindet auch der Objekt-Propertyausdruck `OPE` das Individuum $\texttt{a}_1$ mit dem Individuum $\texttt{a}_2$. Das folgende Beispiel „Alice hat Vater Bill, Bill hat Bruder Willy, dann hat Alice den Onkel Willy“ ist in Abbildung 3.1 graphisch dargestellt:

```
SubObjectPropertyOf(ObjectPropertyChain(a:hasFather a:hasBrother)a:hasUncle)
ObjectPropertyAssertion(a:hasFather a:Alice a:Bill)
ObjectPropertyAssertion(a:hasBrother a:Bill a:Willy)
```

Die graphische Darstellung der Entitäten in der Abbildung ist keine formale Syntax von OWL 2, sie dient nur zur Veranschaulichung von Beispielen.

Es kann vorkommen, dass Properties mit unterschiedlichen Namen die gleiche semantische Bedeutung haben. Mit dem Axiom `EquivalentObjectProperties(`$\texttt{OPE}_1$ `...` $\texttt{OPE}_n$`)` können solche Properties als äquivalent definiert werden. Disjunkte Objekt-Properties bzw. Objekt-Propertyausdrücke werden mit `DisjointObjectProperties(`$\texttt{OPE}_1$ `...` $\texttt{OPE}_n$`)` definiert.

In OWL 2 können auch die Hierarchie, die Äquivalenz und die Disjunktion von Daten-Properties definiert werden.

Die Objekt-Properties können bestimmte Eigenschaften besitzen, die auch mit Axiomen deklariert werden. Die möglichen Eigenschaften von Properties sind in Tabelle 3.1 aufgeführt.

Jede Property kann eine passende inverse Property haben. Wie schon erwähnt, haben die Properties eine bestimmte Richtung. Die inverse Property hat die umgekehrte Richtung bezüglich seiner Property. Ein Beispiel ist das Tripel: `Bill-hasChild-Alice`. Die inverse Property von `hasChild` ist beispielsweise `hasParent`. Das Tripel mit der inversen Property ist dann wie folgt aufgebaut: `Alice-hasParent-Bill`. Der geeignete Graph dazu ist in Abbildung 3.2 dargestellt.

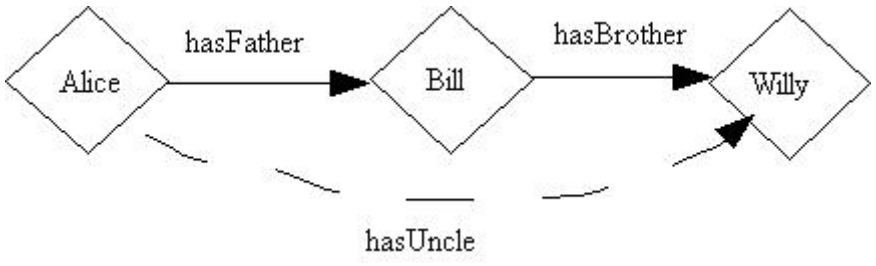


Abbildung 3.1: Verkettung von Properties

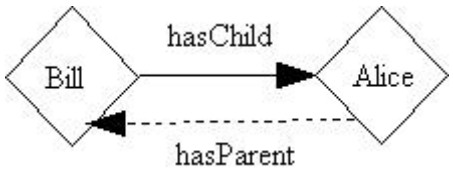


Abbildung 3.2: Inverse Property: hasParent

Da für diese Studienarbeit transitive und symmetrische Properties von besonderer Bedeutung sind, wird hier auf die beiden Eigenschaften detaillierter eingegangen. Die Symmetrie von Properties kann man fol-

Eigenschaften von Property	Deklaration mit der funktionalen Syntax
invers	`InverseObjectProperties(:hasParent :hasChild)`
symmetrisch	`SymmetricObjectProperty(:hasSpouse)`
asymmetrisch	`AsymmetricObjectProperty(:hasChild)`
disjunkt	`DisjointObjectProperties(:hasFather :hasMother)`
transitiv	`TransitiveObjectProperty(:hasAncestor)`
reflexiv	`ReflexiveObjectProperty(:knows)`
irreflexiv	`IrreflexiveObjectProperty(:parentOf)`
funktional	`FunctionalObjectProperty(:hasHusband)`
Invers funktional	`InverseFunctionalObjectProperty(:hasHusband)`

Tabelle 3.1: Eigenschaften von Properties.

gendermaßen definieren: Falls eine Property `P` symmetrisch ist und das Individuum a mit dem Individuum `b` verbindet, dann verbindet die Property `P` das Individuum `b` mit dem Individuum `a`. Zum Beispiel „`Alice` hat Geschwister `Tom`, `Tom` hat Geschwister `Alice`“. In Abbildung 3.3 ist das Beispiel graphisch dargestellt.

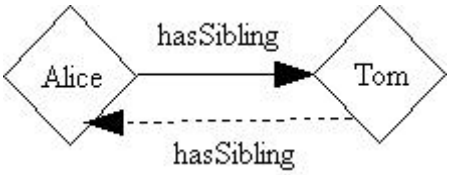


Abbildung 3.3: Symmetrische Property: hasSibling

Transitive Properties können wie folgt erklärt werden: Falls eine Property `P` transitiv ist und das Individuum `a` mit dem Individuum `b` und das Individuum `b` mit dem Individuum `c` verbindet, dann verbindet die Property `P` das Individuum `a` mit dem Individuum `c`, zum Beispiel „Bob ist ein Vorfahre von `Bill`, `Bill` ist ein Vorfahre von `Alice`, dann ist `Bob` auch ein Vorfahre von `Alice` “. Die Abbildung 3.4 demonstriert das Beispiel.

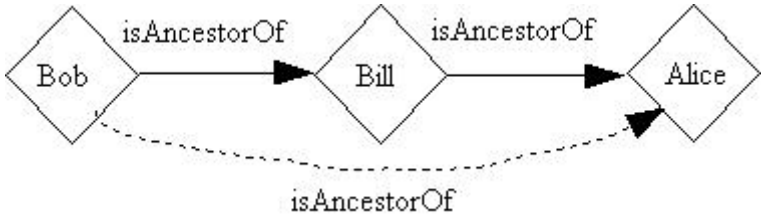


Abbildung 3.4: Transitive Property: isAncestorOf

3.2.3 Axiome über Individuen

In OWL 2 stehen die Axiome über Individuen `SameIndividual`, `DifferentIndividuals`, `ClassAssertion`, `ObjectPropertyAssertion`, `NegativeObjectPropertyAssertion`, `DataPropertyAssertion` und `NegativeDataPropertyAssertion` zur Verfügung.

Es kann vorkommen, dass mehrere Individuen dieselbe Bedeutung haben, wie zum Beispiel der Name `Caroline` und `Caro`. Mit dem Axiom `SameIndividual(`a_1 `...` a_n`)` können die Individuen $a_{i,\ i=1..n}$ gleich gesetzt werden. Für das Beispiel mit dem Namen `Caroline` wird das Axiom wie folgt definiert: `SameIndividual(a: Caroline a: Caro)`.

Mit Hilfe des Axioms `DifferentIndividuals(`a_1 `...` a_n`)` wird die Ungleichheit von Individuen a_i, $_{i=1..n}$ definiert.

Mit dem Axiom `ClassAssertion(CE a)` wird das Individuum `a` dem Klassenausdruck `CE` zugewiesen, dementsprechend ist `a` eine Instanz von `CE`. Zum Beispiel „`Bill` ist eine Person“ kann wie folgt definiert werden: `ClassAssertion(a:Person a:Bill)`.

Es ist auch wichtig die Quelle und das Ziel einer Property festzulegen. Das Axiom `ObjectPropertyAssertion(OPE` a_1 a_2`)` legt das Individuum a_1 als die Quelle und das Individuum a_2 als das Ziel des Propertyausdrucks `OPE` fest.

Es können auch negative Relationen zwischen zwei Individuen definiert werden. Das Axiom `NegativeObjectPropertyAssertion(a:hasFather a:Bill a:Tom)` sagt aus, dass `Tom` nicht der Vater von `Bill` ist.

Mit den Axiomen `DataPropertyAssertion` und `NegativeDataPropertyAssertion` können positive und negative Relationen zwischen einem Individuum und einem Literal festgelegt werden.

Kapitel 4

Einführung in die Anfragesprache SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) ist eine Anfragesprache für RDF. Zurzeit wird zwischen den zwei Versionen SPARQL 1.0 und SPARQL 1.1 unterschieden.

In Anlehnung an [PS08] wird in diesem Kapitel SPARQL 1.0 erläutert. SPARQL 1.0 wurde im Januar 2008 vom W3C als Anfragesprache für RDF standardisiert.

Die erweiterte Version SPARQL 1.1 hat den Status Recommendation noch nicht erreicht und befindet sich zurzeit im Bearbeitungsprozess (Working Draft) [HS10]. Am Ende dieses Kapitels wird auf neue Features der Version 1.1 eingegangen. Alle vorgestellten Anfragen in dem Kapitel werden an den RDF-Datensatz in Listing 2.1 gestellt.

4.1 Anfrageformen in SPARQL

SPARQL bietet vier verschiedene Anfrageformen: SELECT, CONSTRUCT, ASK und DESCRIBE.

4.1.1 SELECT

Die SELECT-Anfrage ist die am häufigsten verwendete Anfrageform. Die Anfrage ist wie folgt aufgebaut:

```
PREFIX  <Namensraum>
SELECT  <Ergebnis>
WHERE   <Graphmuster>
```

In der folgenden Anfrage soll nach den Namen der Studenten gesucht werden, die einen Bibliotheksausweis besitzen:

```
PREFIX pe: <http://www.uni.de/person/>
SELECT  ?studentName
WHERE { ?s  pe:name    ?studentName
        ?s  pe:libraryCard  ?cardNr }
```

Die Anfrage liefert das folgende Ergebnis zurück:

```
@PREFIX pe: <http://www.uni.de/person/> .
studentName
-----------
pe: Tom
pe: Bob
```

In <Namensraum> werden Abkürzungen für Namensräume deklariert, um die Lesbarkeit der Anfragen zu erleichtern. In dem Beispiel ist pe eine Abkürzung für <http://www.uni.de/person/>.

In <Ergebnis> werden die Variablen ausgewählt, die im Ergebnis projiziert werden. Die Variablen werden in SPARQL mit ?- oder $-Zeichen beginnend deklariert, dabei sind ? und $ nicht der Teil des Variablennamens. Die Symbole ? und $ sind äquivalent und können beide zur Deklaration von Variablen verwendet werden. Es können die Variablenwerte von Subjekten, Objekten und Prädikaten als Ergebnis zurückgeliefert werden.

In <Graphmuster> werden Graphmuster innerhalb der WHERE-Klausel definiert. In SPARQL wird es zwischen einfachen und gruppierenden Graphmustern unterschieden. Ein einfaches Graphmuster besteht aus einer Menge von Tripelmustern, die mit einem Punkt voneinander getrennt werden. Die Tripelmuster sind ähnlich wie die Tripel in RDF aufgebaut: Subjekt-Prädikat-Objekt. Ein gruppierendes Graphmuster besteht aus einem oder mehreren einfachen Graphmustern und ist in geschweiften Klammern eingeschlossen.

Die Graphmuster sind Schablonen, nach denen Untergraphen in RDF-Graphen für das Ergebnis gesucht werden. Unter Erfüllung der anderen in der Anfrage angegebenen Voraussetzungen wird dann das Ergebnis aus den gefundenen Untergraphen extrahiert bzw. kombiniert. In dem Beispiel besteht das Graphmuster aus zwei Tripelmustern ?s pe:name ?studentName und ?s pe:libraryCard ?cardNr. Die Subjekte und Objekte sind in diesem Fall Variablen. In SPARQL ist es erlaubt, dass an der Stelle von Subjekten, Objekten und Prädikaten Variablen vorkommen. Der Punkt zwischen den Tripelmustern gilt als logisches Und. In einem einfachen Graphmuster muss jedes Tripelmuster für das Ergebnis erfüllt sein. Dementsprechend wird im Beispiel das Ergebnis nur dann zurückgeliefert, wenn alle Variablen der beiden Tripelmuster gebunden sind.

4.1.2 CONSTRUCT

Die Anfrageform CONSTRUCT ist ähnlich wie die SELECT-Anfrage aufgebaut, das Schlüsselwort SELECT wird durch CONSTRUCT ersetzt.

```
PREFIX     <Namensraum>
CONSTRUCT  <Ergebnis>
WHERE      <Graphmuster>
```

Der PREFIX-Bereich und die WHERE-Klausel sind analog zur SELECT-Anfrage strukturiert.

In <Ergebnis> wird eine Schablone für einen RDF-Graphen in geschweiften Klammern definiert. Diese Schablone besteht auch, wie ein einfaches Graphmuster, aus Tripelmustern. Mit der Anfrageform CONSTRUCT wird ein RDF-Graph gebildet, dementsprechend wird als Ergebnis ein einzelner RDF-Graph zurückgeliefert. Die Variablen der CONSTRUCT-Klausel werden entsprechend durch die in der WHERE-Klausel gefundenen Variablenbelegungen ersetzt.

Im folgenden Beispiel werden mit der WHERE-Klausel Belegungen für die Variablen ?n und ?nr anhand des Graphmusters gesucht.

```
PREFIX pe: <http://www.uni.de/person/>
CONSTRUCT { ?n pe:matriculationNumber ?nr.}
WHERE     { ?x pe:name   ?n.
            ?x pe:matrNumber  ?nr. }
```

Die gefundenen Werte der Variablen `?n` und `?nr` ersetzen die entsprechenden Variablen in der CONSTRUCT-Schablone `?n un:matriculationNumber ?nr`, wenn die beiden Beziehungen `?x pe:name ?n` und `?x pe:matrNumber ?nr` existieren.

Die CONSTRUCT-Anfrage liefert das folgende Ergebnis:

```
@PREFIX pe: <http://www.uni.de/person/> .
pe:Alice  pe:matriculationNumber    "1020" .
pe:Tom    pe:matriculationNumber    "1001" .
pe:Bob    pe:matriculationNumber    "1011" .
```

4.1.3 ASK

Die ASK-Anfrage ist wie folgt aufgebaut:

```
PREFIX     <Namensraum>
ASK
WHERE      <Graphmuster>
```

Der Aufbau der ASK-Anfrage ist ähnlich der SELECT-Anfrage mit der Ausnahme, dass nach dem Schlüsselwort ASK keine Variablen folgen. Der PREFIX-Bereich und die WHERE-Klausel sind wie bei der SELECT-Anfrage aufgebaut.

Die ASK-Anfrage liefert einen Wahrheitswert zurück. Es wird geprüft, ob das in der WHERE-Klausel angegebene Graphmuster einen entsprechenden Untergraphen besitzt. Wenn es einen entsprechenden Untergraphen gibt, wird der Wahrheitswert `true` zurückgeliefert, wenn nicht, dann `false`. Das folgende Beispiel demonstriert eine ASK-Anfrage. In der Anfrage wird geprüft, ob es einen Studenten gibt, der die Matrikelnummer `1020` besitzt.

```
PREFIX pe: <http://www.uni.de/person/>
ASK
WHERE {?x pe:matrNumber "1020"}
```

4.1.4 DESCRIBE

Die DESCRIBE-Anfrage ist wie folgt strukturiert:

```
PREFIX     <Namensraum>
DESCRIBE   <Ergebnis>
WHERE      <Graphmuster>
```

In <Ergebnis> werden Variablen oder URIs/IRIs[1] deklariert. Der PREFIX-Bereich und die WHERE-Klausel haben dieselbe Struktur wie die anderen drei Anfragen.

[1]IRI (Internationalized Resource Identifier) ist eine Erweiterung der URI um fast allen Zeichen des Universal Character Set (Unicode).

Eine DESCRIBE-Anfrage ist dann hilfreich, wenn der Benutzer keine Vorstellung vom Aufbau des RDF-Graphen hat bzw. davon, wie die Daten in einem RDF-Dokument modelliert sind. Diese Anfrage liefert einen RDF-Graphen zurück, der gefundene Objekte und ihre Beziehungen beschreibt, dabei ist nicht festgelegt, was genau zurückgeliefert wird. Das Ergebnis der DESCRIBE-Anfrage ist in SPARQL nicht genau spezifiziert.

Das folgende Beispiel demonstriert eine DESCRIBE-Anfrage:

```
PREFIX pe: <http://www.uni.de/person/>
DESCRIBE ?s
WHERE { ?s pe:name pe:Alice }
```

In der Anfrage wird nach Beziehungen von `Alice` und nach anderen potentiell wichtigen Informationen über `Alice` gesucht.

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
@prefix pe: <http://www.uni.de/person/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
pe:student1       rdf:type        pe:student
pe:student1       pe:name         pe:Alice
pe:student1       pe:matrNumber   "1020"
pe:student1       pe:age          "20"
pe:student1       pe:phone        _:autos1
pe:student1       pe: friends     pe:student2
pe:student1       pe: friends     pe:student3
_:autos1          pe:homePhone    "0234"
_:autos1          pe:mobilePhone  "0176"
```

4.2 Filter

Mit dem Schlüsselwort FILTER können zusätzliche Bedingungen definiert werden, die nicht mit Graphmustern dargestellt werden können. FILTER wird innerhalb der WHERE-Klausel verwendet. Die Filter-Bedingungen folgen nach dem Schlüsselwort FILTER und sind in runden Klammern eingeschlossen. Arithmetische Operationen, Vergleichsoperationen, reguläre Ausdrücke und weitere spezielle Operationen können mit FILTER verwendet werden. SPARQL 1.0 verfügt über folgende arithmetische Operatoren: + (Addition), - (Subtraktion), * (Multiplikation) und / (Division). Die Vergleichsoperatoren sind = (gleich), <= (kleiner/gleich), < (kleiner) und > (größer). Die weiteren Filter-Operatoren sind in Tabelle 4.1 aufgeführt. Das folgende Beispiel demonstriert die Verwendung von FILTER:

```
PREFIX pe:<http://www.uni.de/person/>
SELECT ?studentName
WHERE { ?student pe:name ?studentName.
?student pe:age ?a.
FILTER (?a < 21)
}
```

Die Anfrage liefert die Namen von Studenten, die jünger als 21 Jahre alt sind. Die Anfrage liefert das folgende Ergebnis zurück:

```
StudentName
-----------
pe:Alice
pe:Bob
```

Operator	Beschreibung
BOUND(A)	Liefert true, wenn A eine gebundene Variable ist.
isURI(A) / isIRI	Liefert true, wenn A eine URI/IRI ist.
isBLANK(A)	Liefert true, wenn A ein leerer Knoten ist.
isLiteral(A)	Liefert true, wenn A ein RDF-Literal ist.
STR(A)	Liefert RDF-Literale und URIs als ein xsd:string zurück.
LANG(A)	Liefert Sprachangabe des Literals A als xsd: string oder einen leeren String, wenn keine Sprachangabe vorhanden ist.
sameTERM(A, B)	Liefert true, wenn A und B dieselben RDF-Terme sind.
langMATCHES(A, B)	Liefert true, wenn A und B die gleichen Sprachangaben haben.
REGEX(A, B)	Liefert true, wenn im String A, der reguläre Ausdruck B gefunden wird.

Tabelle 4.1: Spezielle Filteroperatoren [PHRS08]

4.3 Optionale und alternative Graphmuster

In der WHERE-Klausel können optionale und alternative Graphmuster definiert werden.

4.3.1 OPTIONAL

Mit dem Schlüsselwort OPTIONAL werden in SPARQL optionale Graphmuster definiert. Nach dem Schlüsselwort OPTIONAL folgt in geschweiften Klammern ein Graphmuster, genauer ein gruppierendes Graphmuster, das als ein optionales Graphmuster bezeichnet wird. Die folgende Anfrage veranschaulicht die Verwendung eines optionalen Graphmusters, das aus zwei Tripelmustern besteht:

```
PREFIX pe: <http://www.uni.de/person/>
SELECT    ?name ?mobile
WHERE { ?student  pe:name  ?name.
        OPTIONAL
        {?student  pe:phone ?phoneNr.
        ?phoneNr  pe:mobilePhone ?mobile}
      }
```

Mit den optionalen Graphmustern wird die optional gesuchte Information definiert. Im Beispiel wird nach Namen von Studierenden und ihren Handynummern gesucht. Im Vordergrund stehen die Namen von Studierenden und die Handynummern sind optional. Die Anfrage liefert alle Namen von Studenten zurück, die mit oder ohne eine Handynummer gespeichert wurden. Die Anfrage gibt das folgende Ergebnis zurück:

```
Name        mobile
--------   --------
pe:Alice    0176
pe:Bob
pe:Tom
```

Wird die Anfrage ohne OPTIONAL formuliert, werden nur die Namen von Studierenden zurückgegeben, die mit den Handynummern gespeichert wurden. Dementsprechend werden nicht alle Namen der Studierenden im Ergebnis angezeigt.

4.3.2 UNION

Mit dem Schlüsselwort UNION werden alternative Graphmuster definiert. Vor und nach dem Schlüsselwort UNION stehen in geschweiften Klammern Graphmuster, die alternative Graphmuster genannt wer-

den. UNION vereinigt zwei Graphmustern und entspricht dem logischen `oder`, das heißt für das Ergebnis reicht es schon, wenn eines der Graphmuster erfüllt ist. Die Variablen der beiden Graphmuster beeinflussen sich nicht gegenseitig. Das heißt die gleichnamigen Variablen der beiden Graphmuster stehen in keiner Beziehung zueinander. Die folgende Anfrage stellt die Verwendung von UNION dar:

```
prefix pe: <http://www.uni.de/person/>
SELECT  ?studentName ?phoneNr
WHERE { ?student  pe:name  ?studentName.
        ?student  pe:phone ?phoneNum.
        {?phoneNum pe:homePhone ?phoneNr}
       UNION
        {?phoneNum pe:mobilePhone ?phoneNr}
      }
```

Die Anfrage liefert Namen von Studenten mit ihren Telefonnummern (Festnetznummern und/oder Handynummern) zurück. Die Anfrage hat das folgende Ergebnis:

```
studentName    phoneNr
 ---------     --------
 pe:Bob         0614
 pe:Tom         0634
 pe:Alice       0234
 pe:Alice       0176
```

4.4 Modifikatoren

In SPARQL 1.0 stehen folgende Modifikatoren zur Verfügung: ORDER BY, LIMIT, OFFSET und DISTINCT. Die Modifikatoren modifizieren die Ergebnismenge und stehen in der Anfrage nach der WHERE-Klausel.

4.4.1 ORDER BY

Für die Sortierung der Ergebnismenge wird ORDER BY verwendet. ORDER BY oder ORDER BY ASC sortiert die Ergebnismenge aufsteigend. Mit ORDER BY DESC wird die Ergebnismenge in absteigende Reihenfolge gebracht.

4.4.2 LIMIT und OFFSET

Mit dem Modifikator LIMIT wird die Anzahl der Ergebniselemente begrenzt. Der Modifikator bestimmt die Anzahl der ersten Ergebniselemente. Verwendet man den Modifikator LIMIT in der Kombination mit dem Modifikator OFFSET, kann die Stelle, ab der die Ergebnismenge zurückgeliefert wird, festgelegt werden. Der Modifikator LIMIT ist nur mit der Verwendung des Modifikators ORDER BY sinnvoll. Ohne Sortierung der Ergebniselemente kann es nicht garantiert werden, dass immer die gleiche Reihenfolge der Ergebniselemente zurückgegeben wird.

4.4.3 DISTINCT

Duplikate in der Ergebnismenge werden mit dem Modifikator DISTINCT eliminiert.

4.5 Leere Knoten (Blank Nodes)

Leere Knoten (Blank Nodes) können auch in SPARQL-Anfragen verwendet werden. Das Verhalten von leeren Knoten ist mit Variablen vergleichbar, mit der Ausnahme, dass sie mit SELECT nicht ausgewählt werden können. Es gibt zwei Möglichkeiten [ ] oder _:KnotenID (analog zur Turtle-Syntax) leere Knoten zu deklarieren. In einem einfachen Graphmuster können gleichnamige leere Knoten mehrmals verwendet werden, dürfen aber nicht in mehreren Graphmustern innerhalb einer Anfrage vorkommen.

4.6 Neue Features in Version 1.1

Ausdrücke in der SELECT-Klausel, Aggregationsfunktionen, Negation, Unteranfragen und Property Pfade sind neue Features in SPARQL 1.1. Diese werden in den folgenden Abschnitten beschrieben [HS10].

4.6.1 Ausdrücke in SELECT-Klausel

Es ist jetzt möglich, in der SELECT-Klausel neue Variablen zu definieren und arithmetische Ausdrücke und Aggregationsfunktionen zu schreiben. Zum Beispiel: `SELECT ?title (?p*(1-?d) AS ?price)`, hier wird der arithmetische Ausdruck `?p*(1-?d)` berechnet und in einer neuen Variable `?price` gespeichert. Nach dem Schlüsselwort `AS` folgt eine neue Variable, die nicht im Graphmuster vorkommen muss. Die Variable beeinflusst Variablen in Graphmustern nicht, deswegen kann der Variablenname mit den Variablennamen in den Graphmustern übereinstimmen. Die Ausdrücke in der SELECT-Klausel erleichtern die Gestaltung der Anfragen und bieten neue Möglichkeiten bei der Erstellung der Anfragen.

4.6.2 Aggregation

GROUP BY ist eine neue Aggregationsfunktion in SPARQL 1.1, die Gruppierungen von Werten nach bestimmten Parametern ermöglicht. GROUP BY befindet sich am Ende der Anfrage außerhalb der WHERE-Klausel. Nach dem Schlüsselwort GROUP BY folgen eine oder mehrere Variablen, die mit einem Leerzeichen voneinander getrennt sind.

Mit dem Schlüsselwort HAVING können zusätzliche Bedingungen über Gruppen definiert werden. Der Schlüsselwort HAVING folgt nach der Funktion GROUP BY.

Die weiteren neuen Funktionen COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT und SAMPLE sind in Tabelle 4.2 beschrieben. Die Funktionen können in der SELECT-Klausel und/oder im HAVING-Bereich definiert werden. Die folgende Anfrage verwendet die Aggregationsfunktionen COUNT:

```
PREFIX pe: <http://www.uni.de/person/>
SELECT   ?studentName
WHERE {  ?s  pe:name     ?studentName.
         ?s  pe:friends  ?friendName  }
GROUP BY ?studentName
HAVING (COUNT(?friendName)>1)
```

Zuerst werden mit GROUP BY `?studentName` Gruppen nach Studentennamen gebildet, dann werden mit `COUNT(?friendName)` Freundennamen in jeder Gruppe gezählt. Die Gruppen (Studentennamen), die mehr als einen Freundennamen enthalten, werden im Ergebnis angezeigt. Die Anfrage liefert folgendes Ergebnis zurück:

```
studentName
------------
pe:Alice
```

Funktion	Beschreibung
COUNT(E)	Gibt die Anzahl der Werte zurück, die durch den Ausdruck E bestimmt werden.
SUM(E)	Gibt die Summe der Werte zurück, die durch den Ausdruck E bestimmt werden.
MIN(E)	Gibt den Minimalwert der Werten zurück, die durch den Ausdruck E bestimmt werden.
MAX(E)	Gibt den Maximalwert der Werten zurück, die durch den Ausdruck E bestimmt werden.
AVG(E)	Gibt den Durchschnittswert der Werten zurück, die durch den Ausdruck E bestimmt werden.
GROUP_CONCAT(E)	Gibt eine Verkettung der Werten als einen String zurück, die durch den Ausdruck E bestimmt wird.
SAMPLE(E)	Gibt einen beliebigen Wert der Werten zurück, die durch den Ausdruck E bestimmt werden.

Tabelle 4.2: Aggregationsfunktionen in SPARQL 1.1 [HS10]

4.6.3 Negation

In SPARQL 1.0 ist es nicht möglich Tripelmuster direkt zu negieren. Es gibt dafür eine andere Möglichkeit die Tripelmuster auszuschließen. Das folgende Beispiel demonstriert diese Möglichkeit der Negation:

```
PREFIX pe: <http://www.uni.de/person/>
SELECT ?name
WHERE {  ?s pe:name ?name
         OPTIONAL
         { ?s pe:libraryCard ?cardNr }
         FILTER (!BOUND(?cardNr))
      }
```

Es werden zuerst alle Studenten - mit dem und ohne den Bibliotheksausweis - gefunden, dann werden Studenten die Bibliotheksausweise haben entfernt. Im Ergebnis werden Studenten angegebenen, bei denen die Variable `?cardNr` nicht gebunden ist bzw. kein Tripelmuster `?s pe:libraryCard ?cardNr` im Untergraphen vorkommt. Die Anfrage hat folgendes Ergebnis:

```
studentName
------------
 pe:Alice
```

Mit der Einführung des neuen Feature NOT EXISTS ist es einfacher in SPARQL 1.1 Graphmuster zu negieren. NOT EXISTS ist ein Filterausdruck, der direkt nach dem Schlüsselwort FILTER folgt. Nach dem FILTER NOT EXISTS werden in geschweiften Klammern Tripelmuster definiert. Das folgende Beispiel demostriert die Verwendung von NOT EXIST:

```
PREFIX pe: <http://www.uni.de/person/>
SELECT   ?name
WHERE {  ?s pe:name ?name.
         FILTER NOT EXISTS { ?s pe:libraryCard ?cardNr }
       }
```

Die Anfrage liefert Studenten zurück, die über das Prädikat `pe:libraryCard` keine Verbindungen besitzen. Das heißt das Ergebnis zeigt alle Studenten an, bei denen im Untergraphen kein Tripelmuster `?s pe:libraryCard ?cardNr` vorhanden ist.

4.6.4 Unteranfragen

In einigen Fällen ist es notwendig das Ergebnis einer Anfrage als Parameter einer anderen Anfrage zu übergeben. Das kann mit Hilfe von Unterprogrammen bzw. Unteranfragen realisiert werden. Eine Unteranfrage wird innerhalb der WHERE-Klausel einer Hauptanfrage definiert. Die folgendde Anfrage enthält eine Unteranfrage:

```
PREFIX pe: <http://www.uni.de/person/>
SELECT   DISTINCT ?ffn
WHERE { ?f   pe:friends ?ff.
        ?ff  pe:name   ?ffn.
        SELECT ?f
        WHERE {?student   pe:name pe:Alice.
               ?student   pe:friends  ?f }
        ORDER BY DESC(?f)
        LIMIT 2
      }
```

Die Anfrage liefert alle Freunde von zwei Personen, die Alice´s Freunde sind. Die Unteranfrage gibt die ersten zwei Freunde (Namen alphabetisch absteigend sortiert) von Alice zurück. Das Ergebnis der gesamten Anfrage sieht wie folgt aus:

```
ffn
------
pe:Bob
```

4.6.5 Property Pfad

Ein Property Pfad ist eine Sequenz von Properties zwischen zwei Elementen (Variable oder RDF-Term). Der Anfang oder das Ende eines Pfades kann ein RDF-Term oder eine Variable sein. Eine Variable kann nicht der Teil eines Pfades sein, nur der Anfang oder das Ende eines Pfades. Ein Tripelmuster ist ein Beispiel für einen einfachen Pfad, der die Länge 1 besitzt.

Die folgende Anfrage verwendet ein Pfadausdruck:

```
PREFIX    pe: <http://www.uni.de/person/>
SELECT    DISTINCT  ?ffName
WHERE {?student   pe:name  pe:Alice.
       ?student   pe:friends/pe:friends ?ff.
       ?ff        pe:name  ?ffName
      }
```

Die Anfrage enthält den Property Pfad `pe:friends/pe:friends`, der aus einer Sequenz von zwei Properties besteht.

Ein Property Pfad stellt einen möglichen Weg zwischen zwei Knoten in einem RDF-Graphen dar. Bei der Auswertung einer Anfrage werden alle möglichen Pfade gematcht und an Subjekte oder Objekte gebunden. In der Beispielanfrage werden Pfade gesucht, die aus zwei aufeinander folgenden Properties `pe:friends`

bestehen und damit alle Freunde von Alice's Freunden zurückliefert. Die Anfrage liefert das folgende Ergebnis zurück:

```
fName
-------
pe:Bob
```

Mögliche Property Pfadausdrücke sind in Tabelle 4.3 beschrieben, dabei ist `elem` ein Pfadelement.

Syntax	**Beschreibung**
uri	Eine URI bzw. ein Prefix-Name.
$\hat{}elem$	Ein inverser Pfad. (vom Objekt zum Subjekt)
$!uri_1$	Eine negierte Property: uri_1 darf nicht im angegebenen Pfad vorkommen.
$!(uri_1\|...\|uri_n)$	Eine negierte Propertymenge: $uri_1...uri_n$ dürfen nicht im angegebenen Pfad vorkommen.
$!\hat{}uri_1$	Eine negierte Property: uri_1 darf nicht als inverse Property im angegebenen Pfad vorkommen.
$!(uri_1\|...\|uri_j\|\hat{}uri_{j+1}\|...\|\hat{}uri_n)$	Eine negierte Propertymenge: $uri_1...uri_j$ dürfen nicht als Properties und $uri_{j+1}...uri_n$ als inverse Properties im angegebenen Pfad vorkommen.
$elem_1/elem_2$	Eine Pfadsequenz: Im angegebenen Pfad muss nach $elem_1$ $elem_2$ folgen.
$elem_1\|elem_2$	Alternative Pfade: Im angegebenen Pfad kommt $elem_1$ oder $elem_2$ vor.
$elem*$	$elem$ kann beliebig oft (auch keinmal) im angegebenen Pfad vorkommen.
$elem+$	$elem$ kommt mindestens einmal im angegebenen Pfad vor.
$elem?$	$elem$ ist optional, es kommt null- oder einmal im angegebenen Pfad vor.
$elem\{n,m\}$	$elem$ muss mindestens n- und darf maximal m-mal im angegebenen Pfad vorkommen.
$elem\{n\}$	$elem$ kommt genau n-mal im angegebenen Pfad vor.
$elem\{n,\}$	$elem$ muss mindestens n-mal im angegebenen Pfad vorkommen.
$elem\{,n\}$	$elem$ darf höchstens n-mal im angegebenen Pfad vorkommen.

Tabelle 4.3: Property Pfadausdrücke [HS10].

Kapitel 5

Einführung in TGraphen

In diesem Kapitel werden der TGraph und seine Eigenschaften angeordnet, typisiert, attribuiert und gerichtet vorgestellt. Im Abschnitt werden Beziehungen zwischen den TGraphen und TGraph-Schemata anhand eines Beispiels erläutert.

Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten. Eine Kante stellt eine Verbindung zwischen zwei Knoten dar.

TGraphen repräsentieren eine allgemeine Art von Graphen. Wie alle Graphen bestehen TGraphen aus Kanten und Knoten.

5.1 TGraph-Elemente

Ein TGraph ist ein angeordneter, typisierter, attribuierter und gerichteter Graph [BE10].

5.1.1 Angeordnet

Die Knoten und die Kanten eines TGraphen können jeweils eine bestimmte Anordnung haben. In Abbildung 5.1 ist ein TGraph abgebildet, in dem die Kanten `e1, e2, e3, e4` und `e5` und die Knoten `v1, v2, v3, v4, v5` und `v6` untereinander angeordnet sind.

Verbindungspunkte zwischen Knoten und Kanten werden Inzidenzen genannt, die auch angeordnet sind. In runden Klammern ist die Anordnung von Inzidenzen im Beispiel dargestellt.

5.1.2 Typisiert

TGraph-Elemente (Kanten und Knoten) und der TGraph selbst sind bestimmten Typen zugeordnet. Im Beispiel sind zwei Typen der Kanten `hasPhone` und `hasFriend` und zwei Typen der Knoten: `Student` und `Phone` dargestellt. Der TGraph `universityMember` ist von dem Typ `University`.

5.1.3 Attribuiert

Sowohl Kanten und Knoten als auch der TGraph selbst können Attribute und Attributwerte besitzen. Der Knoten `v1:Student` hat drei Attribut-Wert-Paare: `name="Alice"`, `matrNr="1020"`, `age=20`. Die Kante `e4:hasFriend` ist eine attributierte Kante, die das Attribut `since` mit dem Wert `"2009"` besitzt.

5.1.4 Gerichtet

Ein TGraph ist ein gerichteter Graph, das heißt, es gibt bezüglich jeder Kante einen Anfangs- und einen Endknoten. In Abbildung 5.1 ist die Richtung durch Pfeile dargestellt, genauer der Knoten `v1:Student` ist der Anfangsknoten und der Knoten `v3:Student` der Endknoten der Kante `e4:hasFriend`. Bezüglich eines Knotens werden eingehende und ausgehende Kanten unterschieden. Zum Beispiel besitzt der Knoten `v3:Student` eine eingehende und zwei ausgehende Kanten. Obwohl ein TGraph ein gerichteter Graph ist, kann er in beide Richtungen (in und gegen die Richtung) traversiert werden.

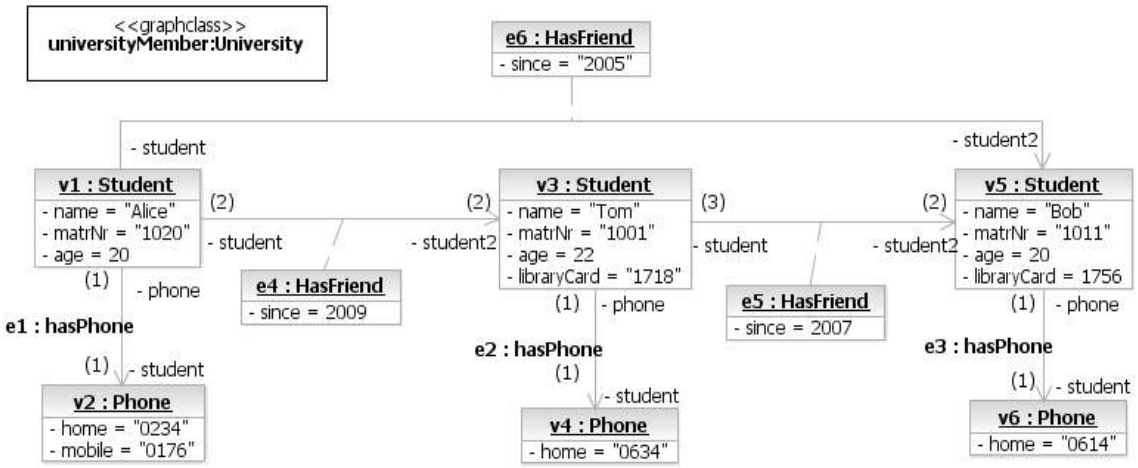


Abbildung 5.1: TGraph

5.2 TGraph und TGraph-Schema

Ein TGraph-Schema dient als ein Metamodell für einen TGraphen. Die Struktur und Bestandteile eines TGraphen werden durch ein TGraph-Schema festgelegt. Typen von Kanten und Knoten sowohl ihre Verbindungen zueinander werden im TGraph-Schema definiert. Der TGraph in Abbildung 5.1 repräsentiert eine Instanz des Schemas in der Abbildung 5.2.

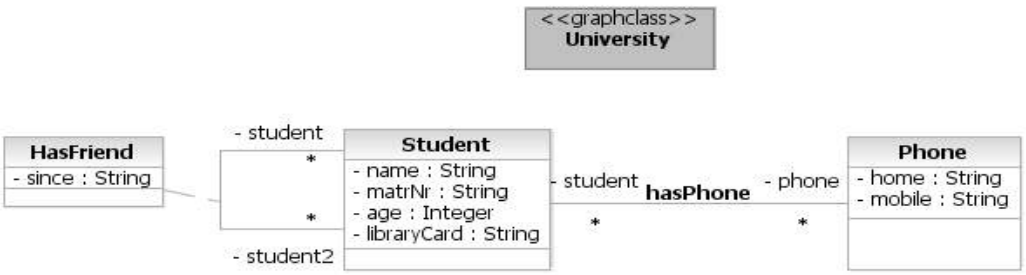


Abbildung 5.2: TGraph-Schema

Die TGraph-Schemata werden mit grUML (Graph UML) modelliert. grUML ist eine Untermenge der Sprache der UML-Klassendiagramme [SE10].

Typen der Knoten werden mittels Klassen und Typen der Kanten mittels Klassen und Assoziationen definiert. Die Klassen `Student` und `Phone` repräsentieren Typen der Knoten. Die Klasse `hasFriend` und die

Assoziation `hasPhone` stellen Typen der Kanten dar.

Wie im UML-Klassendiagramm sind auch hier die Aggregation und die Komposition erlaubt.

Mit Multiplizitäten kann die Anzahl der Nachbarknoten eines Knotens beschränkt werden. Im TGraphen des Beispielschemas können alle Knoten beliebig viele Nachbarknoten besitzen, da alle Multiplizitäten im Schema durch den Stern * gekennzeichnet sind.

Kapitel 6

Einführung in die Anfragesprache GReQL

GReQL (Graph Repository Query Language) ist eine Anfragesprache an TGraphen [BE10].

Alle Anfragen in diesem Kapitel werden an den TGraphen aus Abbildung 5.1 gestellt.

6.1 GReQL-Ausdrücke

Jede GReQL-Anfrage ist ein Ausdruck, der Teilausdrücke enthalten kann. Dementsprechend ist jedes Sprachelement in GReQL ein Ausdruck. In diesem Kapitel findet eine Einführung in GReQL statt. Aus Platzgründen werden hier nicht alle Features von GReQL vorgestellt. Zuerst werden die grundlegenden Sprachelemente Literale, Variablen und Operatoren beschrieben. In den darauffolgenden Abschnitten werden `let`- und `where`-Ausdrücke, bedingte Ausdrücke, Funktionsanwendung, FWR- und quantifizierte Ausdrücke erläutert. Am Ende werden reguläre Pfadausdrücke beschrieben.

6.1.1 Literale und Variablen

In GReQL werden folgende Typen von Literalen unterschieden: Boolesche-, Integer-, Long-, Float- und String-Literale (siehe Tabelle 6.1).

Name	**Beispiele**
Boolesches-Literal	`f`alse, true
Integer-Literal	0, -28.
Long-Literal	1L, -28L
Float-Literal	0d, 2.8
String-Literal	„Eine Zeichenkette“

Tabelle 6.1: Literale [BHG10].

In GReQL werden Variablen an Wertemengen gebunden. Zum Beispiel x:{1,2,3} ist eine Deklaration der Variable x, die an die Integer-Werte 1, 2 und 3 gebunden ist.

6.1.2 Operatoren

GReQL verfügt über boolesche, arithmetische und relationale Operatoren, diese sind in Tabelle 6.2 aufgelistet.

Boolesche Operatoren		**Arithmetische Operatoren**		**Relationale Operatoren**	
Symbol	**Benennung**	**Symbol**	**Benennung**	**Symbol**	**Benennung**
and	logisches Und	+	Addition	=	Gleichheit
or	logisches Oder	-	Subtraktion	<>	Ungleichheit
xor	logisches Exklusiv-Oder	/	Division	<	kleiner
not	logische Negation	-	unäres Minus	<=	kleiner gleich
		%	Modulo	>	größer
		sqrt	Quadratwurzel	>=	größer gleich

Tabelle 6.2: Operatoren [BHG10].

Außer den oben genannten Operatoren gibt es noch Operatoren für Strings. Die Operatoren und dessen Beschreibungen sind in Tabelle 6.3 dargestellt.

Operator	**Beschreibung**
`capitalizeFirst(str)`	Wandelt den ersten Buchstaben des String `str` in einen Großbuchstaben um.
`str1 ++ str2`	Verbindet die zwei Strings `str1` und `str2` zu einem String.
`str1 =~ str2`	Sucht den regulären Ausdruck `str2` in dem String `str1`
`split(str1,str2)`	Teilt den String `str1` nach dem regulären Ausdruck `str2` in Teilstrings auf.
`toString(o)`	Wandelt das Objekt `o` in ein String um.

Tabelle 6.3: Operatoren für Strings [BHG10].

6.1.3 `let`- und `where`-Ausdrücke

Die `let`- und `where`-Ausdrücke sind folgendermaßen aufgebaut:

```
let <Deklaration> in <Ausdruck>
<Ausdruck> where <Deklaration>
```

Im `let`-Ausdruck folgt nach dem Schlüsselwort `let` Variablendeklarationen, die voneinander mit einem Komma getrennt werden. Nach dem Schlüsselwort `in` folgt ein weiterer Ausdruck, in dem die deklarierten Variablen verwendet werden.

Der `where`-Ausdruck beginnt mit einem Ausdruck, nach dem das Schlüsselwort `where` folgt. Am Ende des Ausdrucks werden Variablen deklariert, die in dem Ausdruck verwendet werden.

Die folgenden zwei Beispiele demonstrieren die Verwendung der beiden Ausdrücke. In der ersten Zeile ist ein `let`-Ausdruck dargestellt. In dem Ausdruck werden zuerst die Variablen `x` und `y` definiert, die in dem Ausdruck `x+y` berechnet werden. Die zweite Zeile repräsentiert einen `where`-Ausdruck. Dieser Ausdruck berechnet eine Multiplikation von den Variablen `x` und `y`, die nach dem Schlüsselwort `where` definiert werden.

```
let x:=17, y:=18 in x+y   // Ergebnis: 35
x*y where x:=17,y=2       // Ergebnis: 34
```

6.1.4 Bedingte Ausdrücke

Ein bedingter Ausdruck beginnt mit einem booleschen Ausdruck, danach folgt ein Fragezeichen. Nach dem Fragezeichen folgen zwei Ausdrücke, die voneinander mit einem Doppelpunkt getrennt werden. Der Aufbau des bedingten Ausdrucks sieht wie folgt aus:

```
<Boolescher Ausdruck> ? <Ausdruck1> : <Ausdruck2>
```

Liefert <Boolescher Ausdruck> *true* zurück, wird <Ausdruck1> ausgewertet. Gibt <Boolescher Ausdruck> *false* zurück, wird <Ausdruck2> ausgewertet.

Das folgende Beispiel veranschaulicht die Verwendung der bedingten Ausdrücke. Ist x kleiner als y wird der Ausdruck *y-x* berechnet, wenn nicht, dann der Ausdruck *x-y*.

```
x<y ? y-x : x-y
```

6.1.5 Funktionsanwendung

Eine Funktionsanwendung ist wie folgt aufgebaut:

```
<Funktion> <Typen> <Ausdrücke>
```

Die Funktionsanwendung beginnt mit einem Funktionsnamen (<Funktion>). In <Typen> werden in geschweiften Klammern Typen definiert, diese sind jedoch nicht bei allen Funktionen vorhanden. Danach folgt in runden Klammern ein oder mehrere Ausdrücke. In dem folgenden Beispiel ist `outDegree` der Funktionsname, `HasFriend` der Kantentyp, `v1` der Knotenbezeichner. Die Funktion `outDegree` liefert die Anzahl aller ausgehenden Kanten des Typs `HasFriend` vom Knoten `v1`.

```
outDegree {HasFriend} (v1)   //Ergebnis: 3
```

6.1.6 FWR-Ausdrücke

Die am häufigsten verwendeten GReQL-Anfragen sind `from-with-report` Ausdrücke (FWR-Ausdrücke). Die FWR-Ausdrücke sind wie folgt aufgebaut [Ebe10]:

```
from    <Deklaration>
with    <Bedingung>
report  <Ergebnis>
end
```

In `<Deklaration>` werden Variablen deklariert, die in der Anfrage verwendet werden. Wie schon erwähnt, werden Variablen an Wertemengen gebunden. Außer Basistypen können auch Klassen aus dem Graph-Schema Wertebereiche für die Variablen sein [Mar06]. In der `from`-Klausel aus dem folgenden Beispiel wird die Variable `s` deklariert, die die Knoten (V) vom Typ `Student` repräsentiert. Bei mehreren Deklarationen werden die Deklarationen durch ein Komma getrennt.

```
from   s:V{Student}
with   hasAttribute(s,"libraryCard")
report s.name
end         // Ergebnis: Tom, Bob
```

In <Bedingung> werden Bedingungen definiert, die für das Ergebnis erfüllt sein müssen. Die Bedingungen werden für jede Variablenbelegung ausgewertet. Wird die Bedingung erfüllt, wird für die Variablenbelegung die report-Klausel ausgewertet [Mar06]. Im Beispiel wird eine Bedingung durch die boolesche Funktion hasAttribute(s, „libraryCard") definiert. Diese Funktion prüft, ob ein Knoten vom Typ Student das Attribut libraryCard besitzt.

In <Ergebnis> wird die Struktur des Ergebnisses angegeben. <Ergebnis> besteht aus Ausdrücken, die durch ein Komma voneinander getrennt werden. Der report liefert auch eine Multimenge zurück, das heißt mehrfache Vorkommen der Elemente sind möglich [Mar06]. Bei report kann für jedes Element der Ergebnisliste ein String-Ausdruck angegeben werden. Diese String-Ausdrücke dienen dann als Bezeichner für Wertemengen im Ergebnis. Im Beispiel wird über s.name auf die Werte des Attributs name zugegriffen, das Ergebnis sind also die Werte des Attributs name aller Knoten vom Typ Student.

Anstelle von report können auch reportSet oder reportBag auftreten. Wird anstelle von report reportSet verwendet, liefert der FWR-Ausdruck eine Menge mit eindeutigen Werten als Ergebnis zurück. Mit reportBag wird eine Multimenge zurückgegeben. Der Unterschied eines einfachen report von reportSet liegt darin, dass bei reportBag keine String-Ausdrücke für Elemente der Ergebnisliste angegeben werden können.

6.1.7 Quantifizierte Ausdrücke

In GReQL gibt es drei Arten von quantifizierten Ausdrücken: exists (Existenzquantor), exists! (Eindeutigkeitsquantor) und forall (Allquantor). Alle drei Ausdrücke haben die gleiche syntaktische Struktur. Der Rückgabewert der Ausdrücke ist ein boolescher Wert *false* oder *true*. Die Struktur eines exists-Ausdrucks sieht wie folgt aus:

```
exists <Deklaration> @ <Bedingung>
```

In <Deklaration> werden Variablen analog zum FWR-Ausdruck deklariert. Nach <Deklaration> folgt das Zeichen @, das die Deklaration von Bedingungen trennt. In <Bedingung> werden Bedingungen an Variablen gestellt.

Mit exists wird geprüft, ob es mindestens ein Element gibt, das bestimmte Bedingungen erfüllt. Mittels exists! wird geprüft, ob es genau ein Element gibt, das definierte Bedingungen erfüllt. forall prüft, ob alle Elemente definierte Bedingungen erfüllen.

Die Verwendungen von exists, exists! und forall sind im folgenden Listing dargestellt.

```
exists  s:V{Student} @ s.matrNumber="1020"        // Ergebnis: true
exists! s:V{Student} @ s.name="Tom"              //  Ergebnis: true
forall  s:V{Student} @ degree{HasFriend}(s)>0   //   Ergebnis: true
```

Mit dem exists-Ausdruck wird geprüft, ob es einen Studenten gibt, der die Matrikelnummer „1020" besitzt. Der exists!-Ausdruck prüft, ob es genau einen Studenten gibt, der den Namen „Tom" hat. Der forall-Ausdruck enthält die Funktion degree, die die Anzahl aller inzidenten Kanten (ausgehende und eingehende) des Knotens s zurückliefert. Der Ausdruck prüft, ob jeder Student mindestens einen Freund hat.

6.1.8 Reguläre Pfadausdrücke

Ein regulärer Pfadausdruck beschreibt die Struktur eines Pfades in einem Graphen [BE10].

Mittels - -> (ausgehende Kante), <- - (eingehende Kante) und <-> (unbestimmte Richtung) wird die Richtung einer Kante definiert. Ein einfacher Pfadausdruck ist dann wie folgt aufgebaut:

```
-->{<Kantentyp>, <Rollenname> @ <Bedingung>}
```

In geschweiften Klammern können durch `<Kantentyp>`, `<Rollenname>` und `<Bedingung>` Einschränkungen für Kanten definiert werden. In `<Kantentyp>` werden Kantentypen angegeben. `<Rollenname>`: hier werden Rollennamen der Kanten definiert. In `<Bedingungen>` werden Bedingungen an Kanten gestellt. Bei der Definition mehrerer Kantentypen und Rollennamen werden sie jeweils mit einem Komma voneinander getrennt.

Die Einschränkungen für Knoten sind ähnlich wie die Einschränkungen für Kanten aufgebaut. Die Einschränkungen von Knoten werden auch in geschweiften Klammern geschrieben. Der Unterschied liegt darin, dass bei Knoten das Zeichen `&` eingesetzt wird und die Rollennamen entfallen. Wird das Zeichen `&` nach geschweiften Klammern geschrieben, wird damit die Einschränkung für Startknoten definiert. Das bedeutet, dass ein Pfadausdruck mit dem definierten Startknoten beginnen muss. Steht das Zeichen `&` vor geschweiften Klammern, wird damit die Einschränkung für Endknoten beschrieben. Diese Einschränkung bedeutet, dass ein Pfadausdruck mit dem Endknoten enden muss. Der Aufbau solcher Einschränkungen sieht dann folgendermaßen aus:

```
{<Knotentyp> @ <Bedingung>}&
&{<Knotentyp> @ <Bedingung>}
```

Die erste Zeile definiert Einschränkungen für Startknoten, die zweite Zeile für Endknoten.

Verwendung der regulären Pfadausdrücke

Die regulären Pfadausdrücke sind keine eigenständigen Ausdrücke [SE10]. Sie können

- zwischen zwei Knoten (Path Existence Expression): <Knoten1> <Pfad> <Knoten2>,
- mit einem Startknoten (Forward Vertex Set Expressions): <Knoten1> <Pfad> oder
- mit einem Endknoten (Backward Vertex Set Expressions): <Pfad> <Knoten2>

verwendet werden[1].

Mit der Verwendung <Knoten1> <Pfad> <Knoten2> wird nach Existenz des Pfades <Pfad> zwischen den Knoten <Knoten1> und <Knoten2> gesucht. Wird der Pfad gefunden, liefert der Ausdruck *true* zurück, wenn dieser Pfad nicht existiert wird *false* zurückgegeben.

Die Verwendung <Knoten1> <Pfad> liefert, alle Knoten zurück, die von <Knoten1> über den Pfad <Pfad> erreichbar sind.

Die letzte Verwendung <Pfad> <Knoten2> gibt alle Knoten zurück, von denen der Knoten <Knoten2> über <Pfad> erreichbar ist. Das folgende Listing demonstriert jeweilige Beispiele dazu.

```
1 v1 -->{HasFriend} v2       //Ergebnis: false
2 v1 -->{HasFriend}         // Ergebnis: v3, v5
3 -->{HasFriend} v5        //  Ergebnis: v1,v3
```

In der Zeile 1 ist die Verwendung des regulären Pfadausdrucks zwischen zwei Knoten dargestellt. Der Ausdruck gibt *false* zurück, da die Knoten `v1` und `v2` nicht über eine Kante vom Typ `HasFriend` miteinander

[1] Es gibt noch Pfadsystem Verwendungen [BE10].

verbunden sind. Die Zeile 2 soll alle Knoten zurückliefern, mit denen der Knoten `v1` über eine Kante vom Typ `HasFriend` verbunden ist, also das Ergebnis sind die Knoten `v3` und `v5`. In der Zeile 3 wird nach Knoten gesucht, von denen der Knoten `v5` über `hasFriend` erreichbar ist. Als Ergebnis werden zwei Knoten `v1` und `v3` zurückgegeben.

Pfadarten

In GReQL können optionale, alternative, sequenzielle und iterierte Pfade definiert werden. Die Syntax und die Beschreibung dieser Pfade ist in Tabelle 6.4 dargestellt, dabei ist `elem` ein Pfadelement.

Name	**Syntax**	**Beschreibung**
Optionaler Pfad	[`elem`]	`elem` kommt kein- oder einmal vor.
Alternativer Pfad	`elem1 | elem2`	Es kommt `elem1` oder `elem2` vor.
Sequenzieller Pfad	`elem1 elem2`	Nach `elem1` muss `elem2` folgen.
Iterierter Pfad	`elem+`	`elem` kommt mindestens einmal vor.
	`elem*`	`elem` kann beliebig oft (auch keinmal) vorkommen.
	`elem^n`	`elem` kommt genau n-mal vor.

Tabelle 6.4: Reguläre Pfadausdrücke

Im folgenden Listing ist die Verwendung eines iterierten Pfades dargestellt.

```
{Student @ thisVertex.name= "Alice"}& --> {hasFriend}^2   // Ergebnis: v5
```

Mit dem Schlüsselwort `thisVertex` wird der jeweilige Knoten bezeichnet. Der Ausdruck besteht aus einem Knoten vom Typ `Student`, bei dem das Attribut `name` den Wert „`Alice`“ hat, und einer ausgehenden Kante `hasFriend`, die genau 2 mal (`^2`) im Pfad vorkommen muss. Das Ergebnis des Ausdrucks ist der Knoten `v5`.

Kapitel 7

Testdaten

Als Grundlage für die Generierung der Testdaten wurden die zwei OWL-Ontologien Requirements-Ontologie und Family-Ontologie ausgewählt. Die Ontologien werden in folgenden Abschnitten vorgestellt.

Mit den Ontologien sollen die Eigenschaften der Objekt-Properties „transitiv“, „symmetrisch“, „äquivalent zu“ und die `subPropertyOf`-Beziehungen zwischen Properties getestet werden. Auf die Eigenschaften wird in Abschnitt 8.3 näher eingegangen.

Damit man die oben genannten Eigenschaften von Properties nicht bei jeder Verwendung aufzählen muss, bekommen sie den Namen „besondere Eigenschaften“.

7.1 Requirements-Ontologie

Die Requirements-Ontologie wurde an der Technischen Universität Dresden von Katja Siegemund entwickelt. Die Ontologie beschreibt Anforderungen und ihre Zusammenhänge im Software-Engineering. Sie enthält die sieben Hauptklassen `Artifact`, `Attribute`, `Exception`, `Refinement`, `Source`, `Stakeholder` und `hasCost`. Da die Hierarchie der Klassen umfangreich ist, wurde die Darstellung der Hierarchie auf die drei Abbildungen 7.1, 7.2 und 7.4 verteilt. In Abbildung 7.1 ist die Hierarchie der sieben Klassen bis auf die Unterklasse `RequirementArtifact` dargestellt. Die Hierarchie der Klasse `RequirementArtifact` ist bis auf ihre Unterklasse `NonFunctionalRequirement` in Abbildung 7.2 abgebildet. Die Hierarchie der Klasse `NonFunctionalRequirement` repräsentiert die Abbildung 7.4.

Die Ontologie verfügt über Objekt-Properties, deren Hierarchie in Abbildung 7.3 dargestellt ist. Die Properties sind mit ihren Domänen und Wertebereichen in Tabelle 7.1 aufgelistet.

Außer Objekt-Properties verfügt die Ontologie noch über Daten-Properties, die mit ihren Domänen und Wertebereichen in Tabelle 7.2 dargestellt ist.

Insgesamt beinhaltet die sogenannte TBox[1] der Requirements-Ontologie

- 100 Klassen,
- 32 Objekt-Properties,
- 7 Daten-Properties,

[1] Die TBox enthält terminologisches Schemawissen und die ABox assertionales Instanzenwissen [PHRS08]

- 226 Klassen-Axiome und

- 8 Objekt-Property-Axiome

Property	**Domäne**	**Wertebereich**
belongsToUseCase	Scenario	UseCase
describesRequirement	UseCase	Requirement
hasAbstractRequirement	undefiniert	AbstractFunctionalRequirement
hasCost	hasCost	Cost
hasFailurePostcondition	Requirement	Requirement
hasGoal	Requirement	Goal
hasObstacle	RequirementArtifact	Obstacle
hasRefinementSource	Refinement	Requirement
hasRefinementTarget	Refinement	Requirement
hasRelationship	Requirement	Requirement
hasRick	RequirementArtifact	Risk
hasScenario	UseCase	Scenario
hasSoftMetric	Requirement	SoftMetric
hasSource	Artifact	Source
hasTestCase	Requirement	TestCase
hasTrigger	Requirement	Trigger
hasVerificationMethod	undefiniert	undefiniert
isAlternativeTo (symmetrisch)	Requirement	Requirement
isAuthoredBy	Artifact	Stakeholder
isCoexistentWith (symmetrisch)	Requirement	Requirement
isConnectedToUseCase	Goal	UseCase
isDescribedByUseCase	Requirement	UseCase
isExclusionOf (symmetrisch und äquivalent zu isInConflicWith)	Requirement	Requirement
isGeneralizationOf (transitiv und inverse von isSpecializationOf)	Requirement	Requirement
isInConflicWith (symmetrisch und äquivalent zu isExclusionOf)	Requirement	Requirement
isNegativeContributionTo	RequirementArtifact	RequirementArtifact
isRefinementOf (transitiv)	Requirement	Requirement
isSpecializationOf (transitiv und inverse von isGeneralizationOf)	Requirement	Requirement

Tabelle 7.1: Objekt-Properties der Requirements-Ontologie

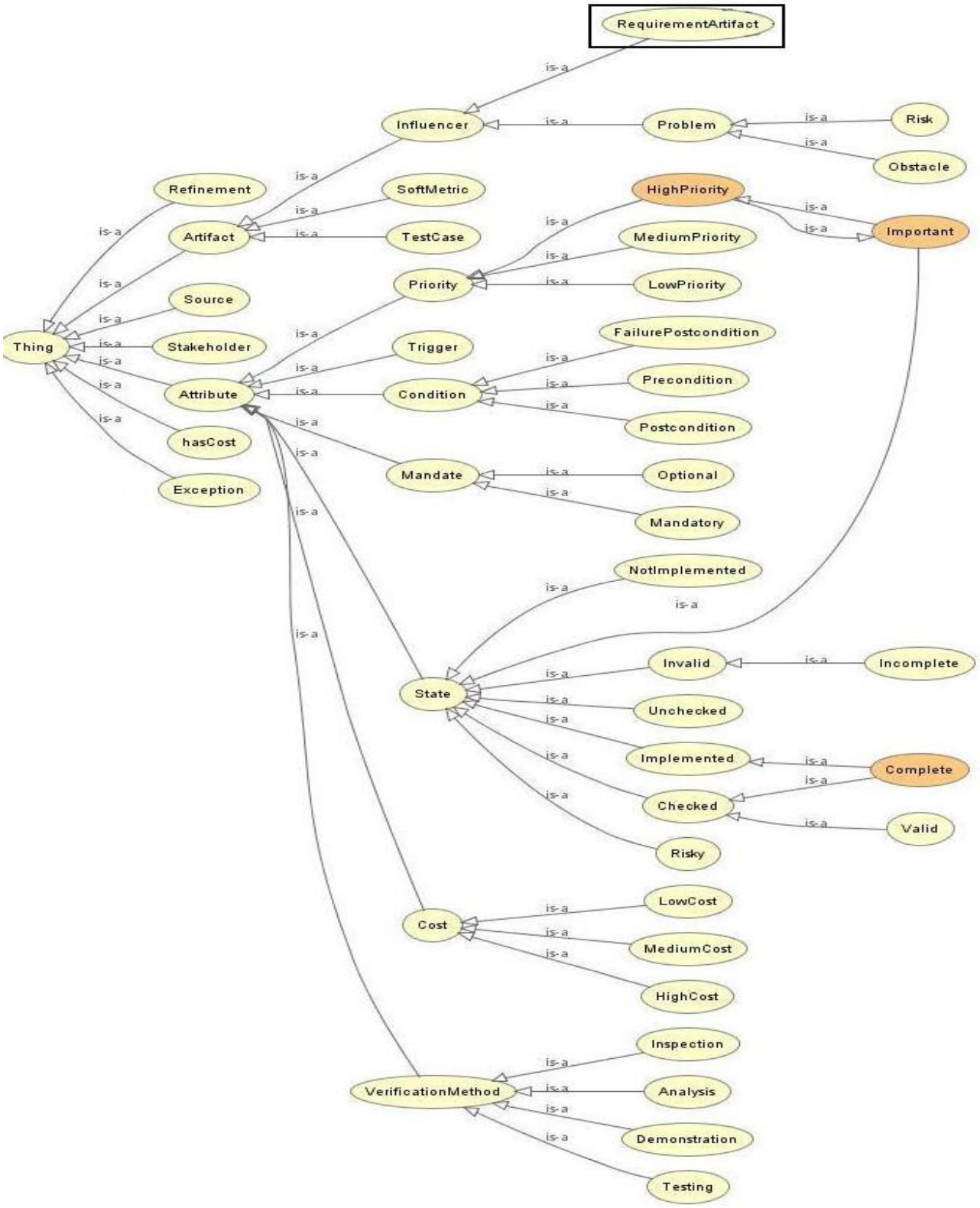


Abbildung 7.1: Hierarchie der Requirements-Ontologie bis fünfte Stufe

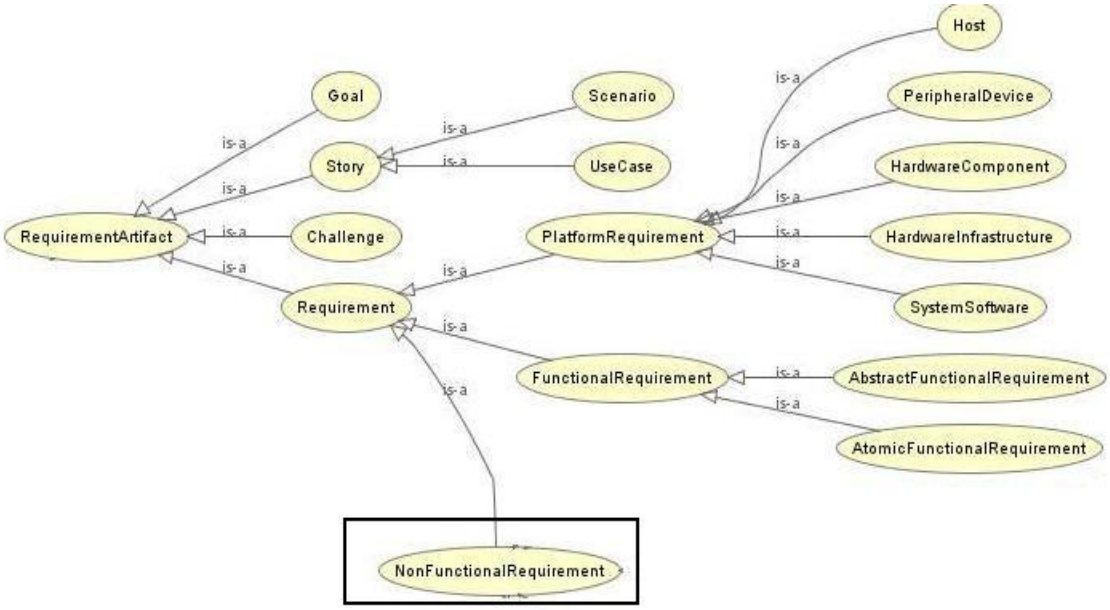


Abbildung 7.2: Hierarchie der Klasse `RequirementArtifact` der Requirements-Ontologie

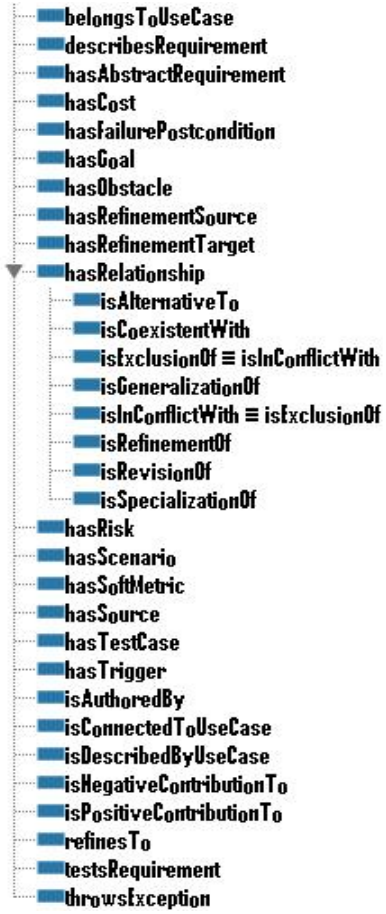


Abbildung 7.3: Hierarchie der Objekt-Properties der Requirement-Ontologie

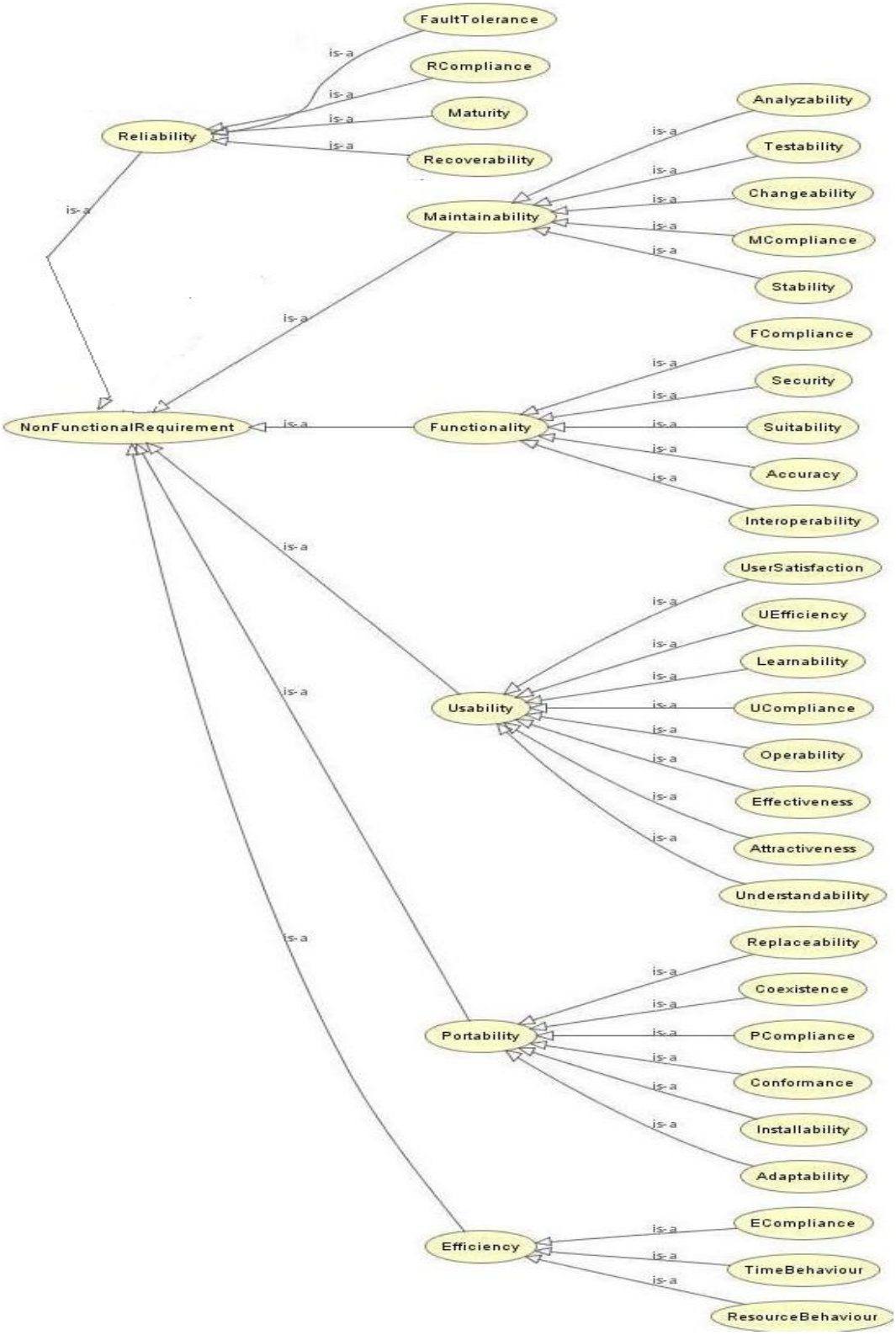


Abbildung 7.4: Hierarchie der Klasse `NonFunctionalRequirement` der Requirements-Ontologie

Property	Domäne	Wertebereich
HasCost	undefiniert	Integer
hasID	RequirementArtifact	String
hasRefinementReason	Refinement	String
hasResponseTime	Requirement	Integer
hasThroughput	Requirement	Integer
hasTimingMetric	Requirement	Integer
hasWaitingTime	Requirement	Integer

Tabelle 7.2: Daten-Properties der Requirements-Ontologie

7.2 Family-Ontologie

Da mit der Requirements-Ontologie nicht alle vorgesehenen Testfälle[2] realisiert werden können, wird für die Generierung der Testdaten noch die Family-Ontologie verwendet.

Die Family-Ontologie ist eine einfache Ontologie, die Familienverhältnisse beschreibt.

Die Hierarchie der Klassen ist in Abbildung 7.5 und die Hierarchie der Properties in Abbildung 7.6 dargestellt.

Die Objekt-Properties der Ontologie sind in Tabelle 7.3 mit ihren Domänen und Wertebereichen aufgelistet. Für Testzwecke enthält die Family-Ontologie die zwei künstlichen Properties `hasArtificialTransitiveRelationshipWith` und `hasArtificialSymmetrricRelationshipWith`, die nur in der Anfrage 8 verwendet werden.

Insgesamt beinhaltet die TBox der Family-Ontologie

- 13 Klassen,
- 28 Objekt-Properties,
- 18 Klassen-Axiome und
- 21 Objekt-Property-Axiome

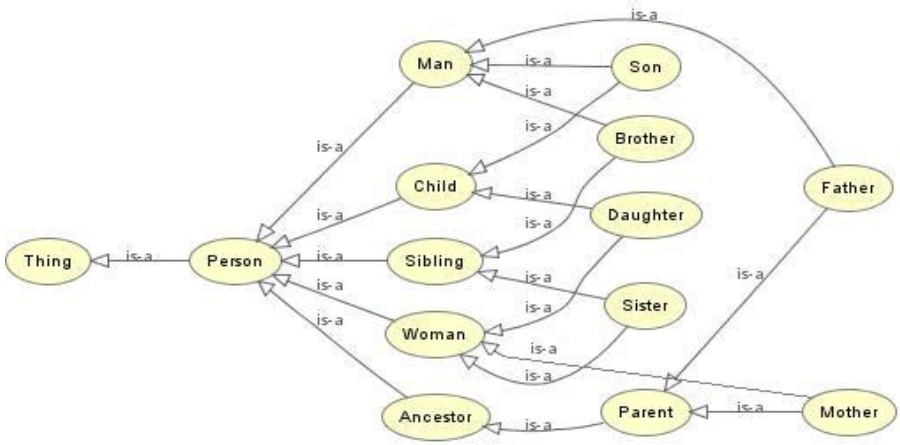


Abbildung 7.5: Klassenhierarchie der Family-Ontologie

[2]Die Testfälle werden im Kapitel 9 vorgestellt

Property	Domäne	Wertebereich
hasAdoptiveChild	Person	Person
hasAdoptiveDaughter	Person	Woman
hasAdoptiveSon	Person	Man
hasAdoptiveGrandchild (äquivalent zu isAdoptiveGrandparentOf)	Person	Person
hasAdoptiveGranddaughter	Person	Woman
hasAdoptiveGrandson	Person	Man
hasAdoptiveSibling (symmetrisch)	Person	Person
hasAdoptiveBrother	Person	Man
hasAdoptiveSister	Person	Woman
hasArtificialSymmetricRelationshipWith (symmetrisch)	Person	Person
hasArtificialTransitiveRelationshipWith (transitiv)	Person	Person
hasBloodRelationship (symmetrisch)	Person	Person
hasAncestor (transitiv)	Person	Person
hasFemaleAncestor (transitiv)	Person	Woman
hasMaleAncestor (transitiv)	Person	Man
hasParent (invers zu hasChild)	Person	Person
hasDad (äquivalent zu hasFather)	Person	Man
hasFather (äquivalent zu hasDad)	Person	Man
hasMother	Person	Woman
hasChild (invers zu hasParent)	Person	Person
hasDaughter	Person	Woman
hasSon	Person	Man
hasCousin (symmetrisch)	Person	Person
hasSpouse (symmetrisch)	Person	Person
hasHusband	Person	Man
hasWife	Person	Woman
isAdoptiveGrandparentOf (äquivalent zu hasAdoptiveGrandchild)	Person	Person

Tabelle 7.3: Objekt-Properties der Family-Ontologie

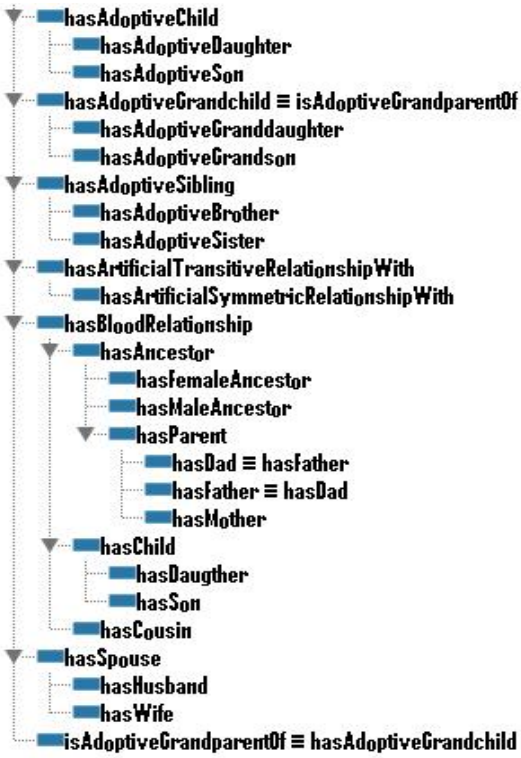


Abbildung 7.6: Hierarchie der Objekt-Properties der Family-Ontologie

7.3 Generierung der Testdaten

Um die Ergebnisse der Anfragen besser kontrollieren zu können, wurde entschieden, die ABoxen der Ontologien nicht nach dem Zufallsprinzip zu generieren. Es wurden so genannte Basis-Ontologien erstellt, die beliebig oft vervielfältigt werden können.

Die Basis-Ontologie der Requirements-Ontologie ist in Abbildung 7.7 und Basis-Ontologie der Family-Ontologie in Abbildung 7.9 dargestellt.

7.3.1 Requirements-Ontologie

Da „besondere Eigenschaften“ von Properties getestet werden sollen, sollen die zu generierenden Daten Properties (Beziehungen) die diese Eigenschaften aufweisen besitzen. Solche Beziehungen sind in der Requirements-Ontologie nur zwischen den Instanzen der Klasse `Requirement` bzw. ihrer Unterklassen möglich. Deswegen wurde für die Generierung der Daten die Klasse `Requirement` gewählt.

Um die entsprechenden Beziehungen zwischen den Instanzen darzustellen, wurden die Unterproperties `isAlternativeTo`, `isInConflictWith`, `isRevisionOf` und `isRefinementOf` von `hasRelationship` verwendet, die diese „besonderen Eigenschaften“ besitzen. Außer diesen Properties wurden die Properties `isDescribedByUseCase` und `hasRefinement` verwendet. Die Property `isDescribedByUseCase` besitzt die inverse Property `describesRequirement`.

Um die „besonderen Eigenschaften“ zu testen, wurden die Instanzen der Klasse `Requirement` entsprechend miteinander verbunden. Der Sinn der Verbindungen zwischen den Instanzen wird im Kapitel 9 bei

der Erläuterung der Anfragen beschrieben.

Die in Abbildung 7.7 dargestellte Basis-Ontologie besteht aus neun Requirements (`Requirement0` - `Requirement8`) und einem `UseCase`. Da für die Evaluierung der Anfragen beliebig große Ontologien verwendet werden sollten, kann die Basis-Ontologie beliebig oft vervielfältigt werden. Es werden so genannte Duplikate der Basis-Ontologie generiert. Die zweifache Vervielfältigung der Basis-Ontologie ist in Abbildung 7.8 dargestellt.

Bei der Generierung der Ontologien werden Requirements durchlaufend nummeriert. Die Duplikat-Ontologien werden am letzten Requirement der jeweiligen Ontologie mit dem ersten Requirement der weiteren Duplikat-Ontologie über `isRefinementOf` nacheinander verbunden. Bei der Generierung mehrerer Duplikat-Ontologien entsteht dadurch eine lange Kette von Requirements, die ununterbrochen über die Vorkommen der transitiven Property `isRefinementOf` nacheinander verknüpft sind. Damit wird die Anfrage 3 (im Kapitel 9) bei größeren Ontologien eine relativ große Ergebnismenge zurückliefern.

Außerdem wird jedes sechste Requirement der Duplikat-Ontologien mit `Requirement5` über das Vorkommen der symmetrischen Property `isAlternativeTo` verknüpft. Damit wird in der Anfrage 2 (im Kapitel 9) die Anzahl der Requirements getestet, die mit `Requirement5` über das Vorkommen der symmetrischen Property verbunden sind.

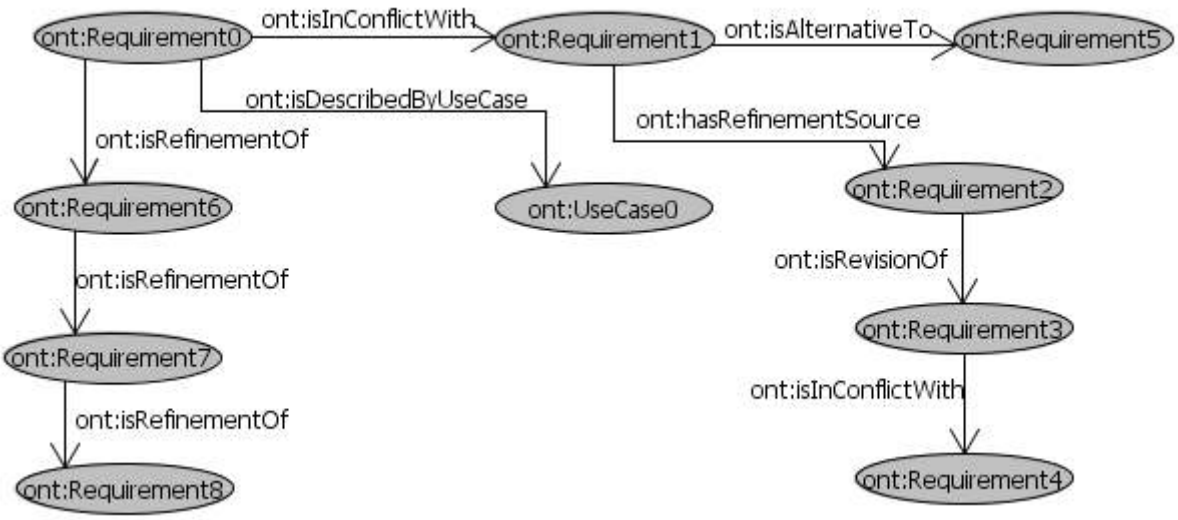


Abbildung 7.7: Basis-Ontologie der Requirements-Ontologie

Die generierten Daten der Requirements-Ontologie sind für die Anfragen 1-5 und 11-13 vorgesehen.

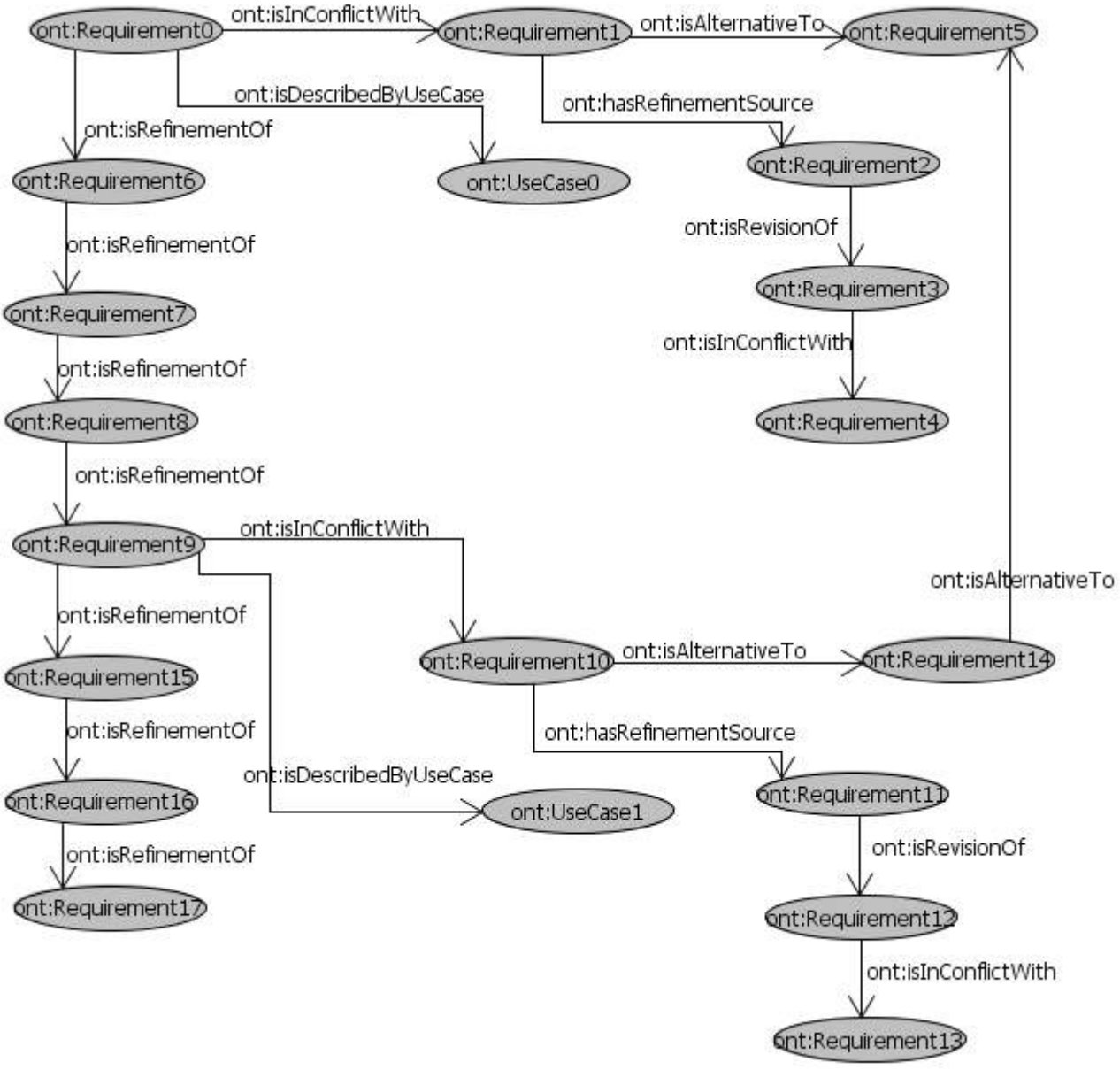


Abbildung 7.8: Zweifache Vervielfältigung der Requirements-Ontologie

7.3.2 Family-Ontologie

Mit der Family-Ontologie sollen auch wie mit der Requirements-Ontologie „besondere Eigenschaften“ von Properties getestet werden. Die Basis-Ontologie der Family-Ontologie ist in Abbildung 7.9 dargestellt. Sie besteht insgesamt aus 17 Personen (`Person0` - `Person16`), die über Vorkommen der jeweiligen Properties miteinander verbunden sind. Der Sinn der Verbindungen zwischen den Individuen wird bei der jeweiligen Anfrage im Kapitel 9 erläutert.

Die Family-Ontologie wird nach dem gleichen Prinzip wie die Requirements-Ontologie vervielfältigt. Die zweifache Vervielfältigung der Family-Ontologie ist in Abbildung 7.10 dargestellt.

Die Individuen der Family-Ontologie werden bei der Generierung durchlaufend nummeriert. Die fünfzehnte Person der jeweiligen Duplikat-Ontologie wird mit der ersten Person der weiteren Duplikat-Ontologie über `hasAncestor` verbunden. Damit wird eine Kette von Personen generiert, die über die transitive Property `hasAncestor` nacheinander verbunden sind. Diese Art der Verbindungen wird bei den unterschiedlichen Ontologiegrößen in der Anfrage 8 im Kapitel 9 getestet. Jede zehnte Person der Duplikat-Ontologie wird mit `Person9` über das Vorkommen der symmetrischen Property `hasAdoptiveSibling` verbunden. Je größer die Ontologie, desto mehr symmetrische Verbindungen wird `Person9` mit anderen Personen besitzen. Damit wird bei der Anfrage 7 die Anzahl der symmetrischen Verbindungen eines Individuums getestet.

Die generierten Daten der Family-Ontologie sind für die Anfragen 6 - 10 und 14 - 19 vorgesehen.

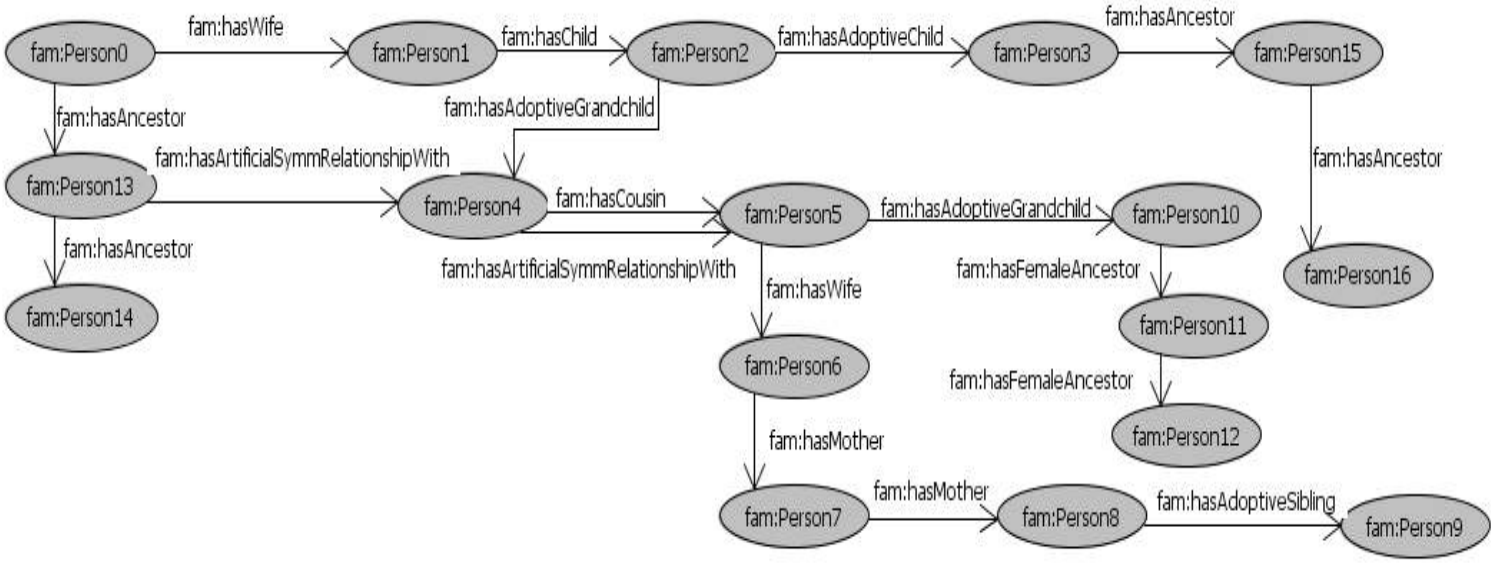


Abbildung 7.9: Basis-Ontologie der Family-Ontologie

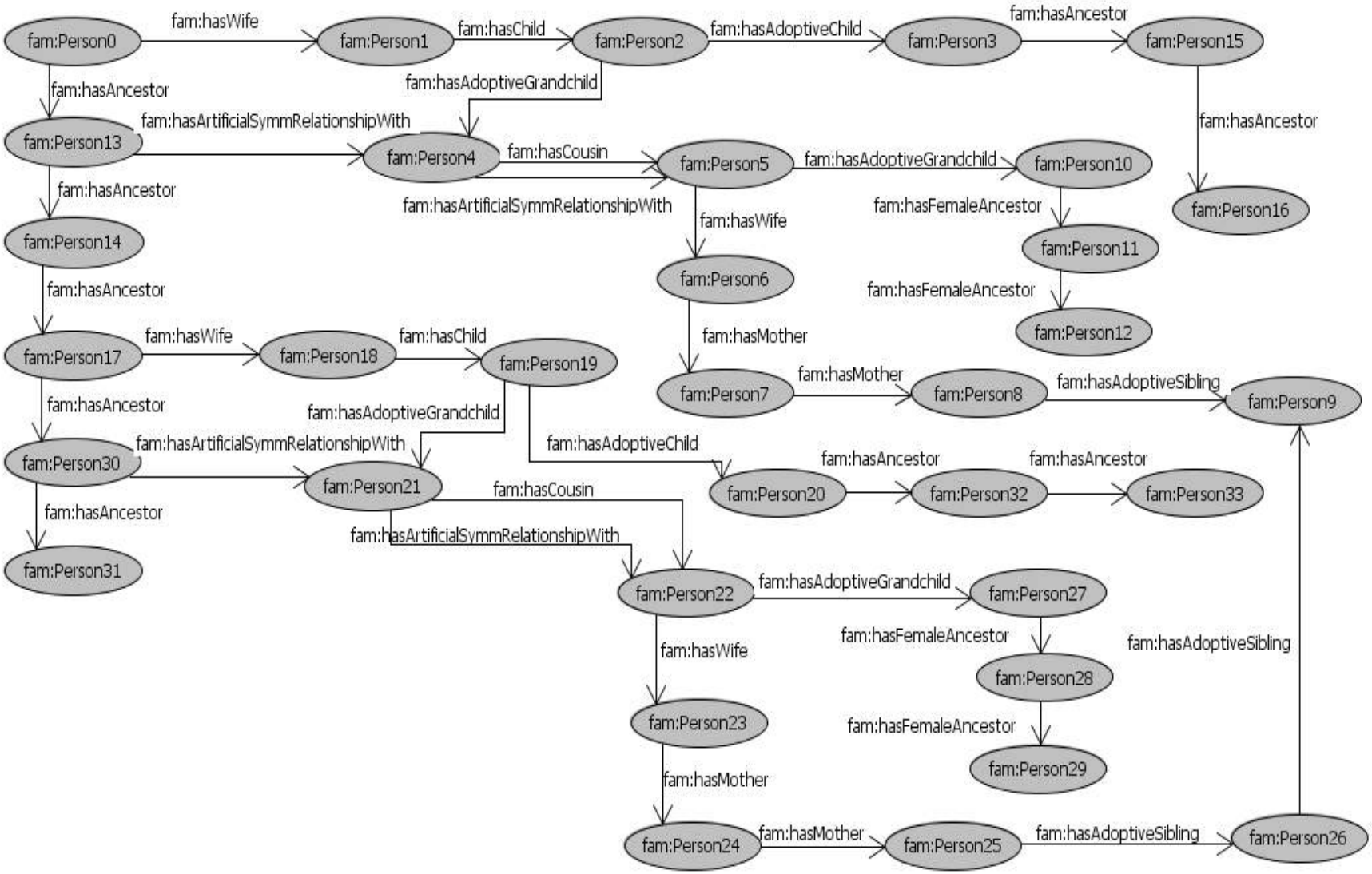


Abbildung 7.10: Zweifache Vervielfältigung der Famiy-Ontologie

Kapitel 8

Transformationsverfahren

In dieser Studienarbeit wird die an der Universität Aberdeen entwickelte Tool-Suite TrOWL zur Anfrage von und zum Reasoning auf OWL-Ontologien verwendet [TPR10]. Das Tool wurde an der Universität Koblenz-Landau um die Transformation der OWL-Ontologien in TGraphen und die Transformation von SPARQL-Anfragen in GReQL-Anfragen erweitert.

In diesem Kapitel werden die zwei Transformationsverfahren Schema-Aware Mapping und Simple Mapping und ein Modifikationsverfahren von Anfragen vorgestellt. Die beiden Transformationsverfahren werden in der Arbeit zur Transformation von OWL-Ontologien in TGraphen, SPARQL-Anfragen in GReQL-Anfragen und GReQL-Anfragen in SPARQL-Anfragen verwendet. Zusätzlich ist es noch möglich die GReQL-Anfragen über ein Modifikationsverfahren zu modifizieren.

Die Implementierung der Verfahren befindet sich noch im Entwicklungszustand, deswegen können noch nicht alle Sprachelemente transformiert bzw. modifiziert werden. Hier wird nur auf die in der Implementierung realisierten Transformations- und Modifikationsmöglichkeiten eingegangen.

8.1 Transformation von OWL-Ontologien in TGraphen

Die OWL-Ontologien werden auf zwei verschiedene Arten in TGraphen transformiert. Eine Variante ist Schema-Aware Mapping, die andere Simple Mapping.

8.1.1 Schema-Aware Mapping

Bei der Transformation der OWL-Ontologien über Schema-Aware Mapping werden ein TGraph-Schema und ein TGraph erzeugt. Wie die einzelnen Elemente der OWL-Ontologien in das TGraph-Schema und den TGraphen transformiert werden, ist in Tabelle 8.1 beschrieben. In der Tabelle kommt bei der Beschreibung der Literale und Daten-Properties die Klasse `Data` vor. Dies ist eine spezielle Klasse, die für die Speicherung der Literale generiert wird. Die Klasse enthält das Attribut `value`, dessen Wert das jeweilige Literal ist.

Um die Transformation über Schema-Aware Mapping zu verdeutlichen, wird hier die Transformation eines kleinen Ausschnittes der Requirements-Ontologie aus Abbildung 8.1 in einen TGraphen vorgeführt.

Die dargestellte Ontologie enthält die sieben Klassen `Thing`, `Artifact`, `Influencer`, `Requirement-Artifact`, `Requirement`, `Story` und `UseCase`. Dabei ist die Klasse `Thing`, wie schon im Kapitel 3

beschrieben, die vordefinierte Superklasse von allen Klassen in den OWL-Ontologien. Die Ontologie besitzt die drei Objekt-Properties `hasRelationship`, `isGeneralizationOf` und `isSpecializationOf`. `HasRelationship` hat als Domäne und Wertebereich die Klasse `Requirement` und besitzt die zwei Unterproperties `isGeneralizationOf` und `isSpecializationOf`, die als transitiv definiert sind. Außerdem ist `isGeneralizationOf` invers zur Objekt-Property `isSpecializationOf`.

Die Ontologie verfügt über vier Instanzen der Klasse `Requirement`, die über die Vorkommen der Property `isGeneralizationOf` nacheinander verbunden sind.

Im folgenden wird die Transformation über Schema-Aware Mapping schrittweise erläutert. Zuerst werden die Klassen und Properties der Ontologie in ein TGraph-Schema transformiert. Dieses TGraph-Schema ist in Abbildung 8.2 dargestellt. Danach werden die Instanzen der Klassen und die Vorkommen der Properties entsprechend dem TGraph-Schema in einen TGraphen überführt (siehe Abbildung 8.3).

Die Klasse `Thing` wird in die Knotenklasse `Thing` transformiert, die das Attribut `uriRef` besitzt, das die URI-Referenzen von Resourcen trägt.

RDF Element	**Beschreibung der Transformation in TGraphen**
Die Klasse `owl:Thing`	Die Klasse `owl:Thing` wird in die Knotenklasse `Thing` transformiert, deren Attribut `uriRef` die URI-Referenzen der Ressourcen trägt.
Klassen	Jede Klasse wird in eine Knotenklasse transformiert. Der Name der Klasse wird von der URI-Referenz abgeleitet.
Objekt-Properties	Jede Object-Property wird in eine Kantenklasse transformiert. Der Name der Klasse wird von der URI-Referenz abgeleitet
Daten-Properties	Jede Daten-Property wird in eine Kantenklasse transformiert, die an der Klasse Thing beginnt und an der Klasse `Data` endet. Der Name der Klasse wird von der URI-Referenz abgeleitet.
Vorkommen von `rdfs:subClassOf` oder `rdfs:subPropertyOf`	Jedes Vorkommen von `rdfs:subClassOf` oder `rdfs:subPropertyOf` werden in Generalisierungsbeziehungen zwischen zwei Knotenklassen oder Kantenklassen entsprechend dem Tripel transformiert. Das Subjekt wird zu einer Unterklasse, das Objekt zu einer Oberklasse.
Ressourcen	Jede Ressource, die keine Klasse, Objekt-Property oder Daten-Property ist, wird in einen Knoten transformiert. Der Typ (Klasse) des Knoten entspricht dem Typ der Ressource. Ressourcen ohne Typen werden Instanzen der Klasse `Thing`.
Vorkommen der Objekt-Property	Jedes Vorkommen der Objekt-Property, das kein Vorkommen von `rdfs:SubClassOf` oder `rdfs:subPropertyOf` ist, wird in eine Kante transformiert. Die Kante ist eine Instanz einer Kantenklasse, die der Objekt-Property entspricht.
Vorkommen der Daten-Property	Jedes Vorkommen der Daten-Property wird in eine Kante entsprechend der Daten-Property transformiert, die an einem Knote der Klasse `Data` endet.
Literale	Jedes Literal wird zu einem Wert des Attributs `value` einer Instanz der speziellen Klasse `Data`.

Tabelle 8.1: Transformation über Schema-Aware Mapping von OWL-Ontologien in TGraphen [SE10]

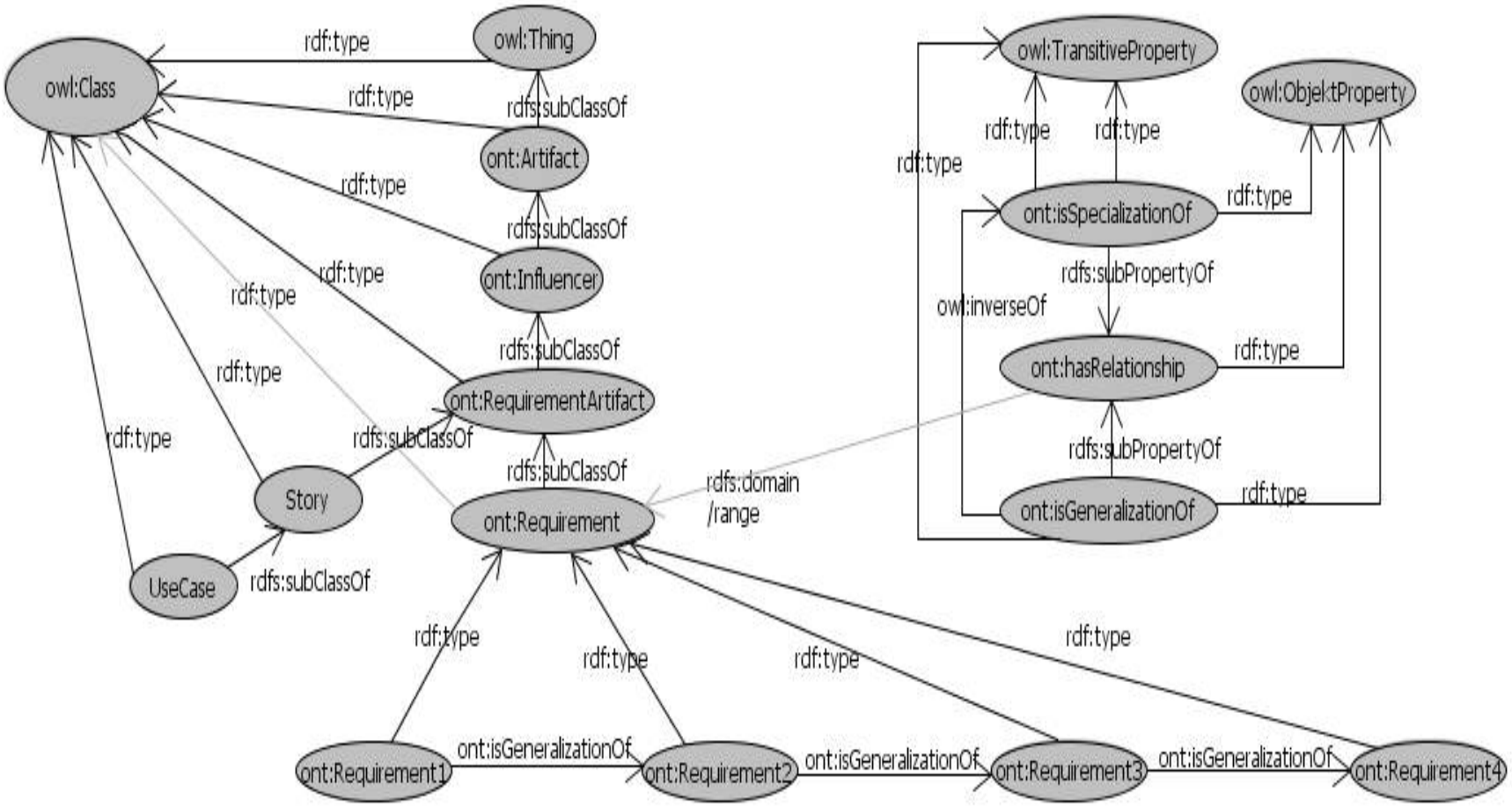


Abbildung 8.1: Ausschnitt aus der Requirements-Ontologie

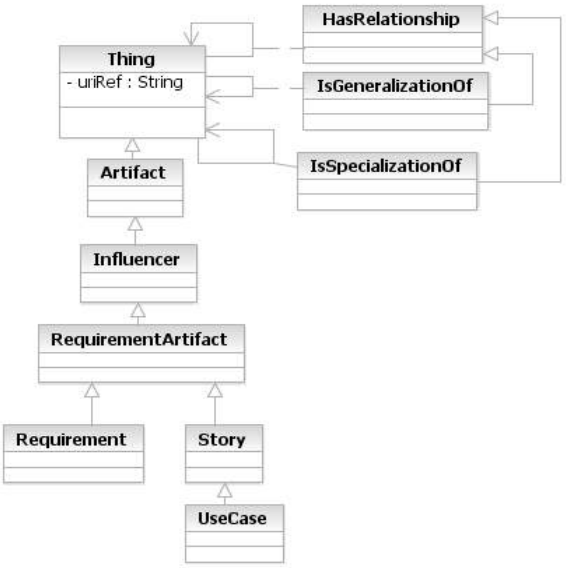


Abbildung 8.2: Der über Schema-Aware Mapping transformierte TGraph-Schema

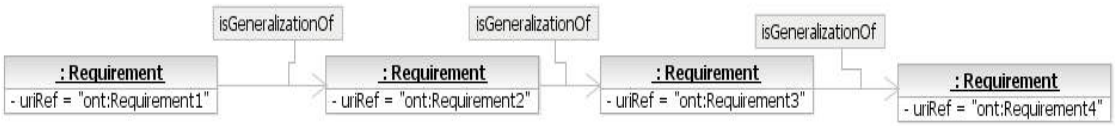


Abbildung 8.3: Der über Schema-Aware Mapping transformierte TGraph

Da die Klassen `Artifact` und `Influencer` über `subClassOf` miteinander verbunden sind, werden sie entsprechend in die zwei Knotenklassen `Artifact` und `Influencer` transformiert, die über Generalisierungsbeziehungen miteinander verbunden sind. Analog werden die anderen Klassen miteinander verbunden.

Die Objekt-Properties `hasRelationship`, `isGeneralizationOf` und `isSpecializationOf` werden in den jeweiligen Kantenklassen transformiert, die an der Klasse `Thing` beginnen und enden. Die Beziehung `subPropertyOf` zwischen `hasRelationship` und `isGeneralizationOf` bzw. zwischen `hasRelationship` und `isSpecializationOf` werden im TGraph-Schema als Generalitätsbeziehungen dargestellt. Dementsprechend sind `isGeneralizationOf` und `isSpecializationOf` die Unterklassen der Knotenklasse `hasRelationship`. Damit ist das TGraph-Schema komplett transformiert.

Jetzt soll die Transformation in den TGraphen erläutert werden. Wie schon beschrieben enthält die Ontologie vier Instanzen der Klasse `Requirement`, die über die drei Vorkommen der Property `isGeneralizationOf` nacheinander verbunden sind.

Bei der Darstellung des TGraph-Schemas und TGraphen aus Abbildungen 8.2 und 8.3 wurde einfacherweise nicht die implementierte Schreibweise der Klassennamen (Typen) gewählt. Der Klassenname `Requirement` entspricht dem implementierten Klassennamen `Ont_numbersign_Requirement1`, `UseCase` dem `Ont_numbersign_UseCase1`, `isGeneralizationOf` dem `Ont_numbersign_isGeneralizationOf` und analog gilt für weitere Klassen.

8.1.2 Simple Mapping

Im Gegensatz zum Schema-Aware Mapping, wo für jede OWL-Ontologie ein TGraph-Schema erzeugt wird, stellt Simple Mapping ein allgemeines TGraph-Schema zur Verfügung, das für alle OWL-Ontologien anwendbar ist. Dieses Schema ist in Abbildung 8.4 dargestellt. Das TGraph-Schema enthält die Knotenklassen `Node`, `Resource`, `Property`, `BlankNode`, `Literal`, `PlainLiteral` und `TypedLiteral` und die Kantenklasse `Arc`, deren Instanzen Instanzen der Knotenklassen miteinander verbinden. Dieses TGraph-Schema ist für jeden über Simple Mapping erzeugten TGraphen kompatibel.

Wie die OWL Elemente in den TGraphen transformiert werden, ist in Tabelle 8.2 beschrieben.

Die Transformation über Simple Mapping wird auch wie Schema-Aware Mapping an einem Beispiel erläutert. Es soll dieselbe Ontologie wie bei der Transformation über Schema-Aware Mapping, aus Abbildung 8.1 über Simple Mapping in einen TGraphen transformiert werden. Dieser TGraph ist in Abbildung 8.5 dargestellt. In dem Fall wird kein TGraph-Schema erzeugt, da Simple Mapping es schon bereit stellt. Die Ontologie wird direkt in den TGraphen überführt.

Alle sieben Klassen der Ontologie werden in die Instanzen der Knotenklasse `Resource` und ihre URI-Referenzen in das Attribut `uriRef` der jeweiligen Instanz transformiert. Die Vorkommen der Property `subClassOf` werden in die Instanzen der Klasse `Arc` übersetzt.

Die drei Properties `hasRelationship`, `isGeneralizationOf` und `isSpecializationOf` werden in die Instanzen der Klasse `Property` und ihre URI-Referenzen in das Attribute `uriRef` transformiert. Die vier Instanzen der Klasse `Requirement` werden auch wie die Klassen als Instanzen der Klasse `Resource` dargestellt.

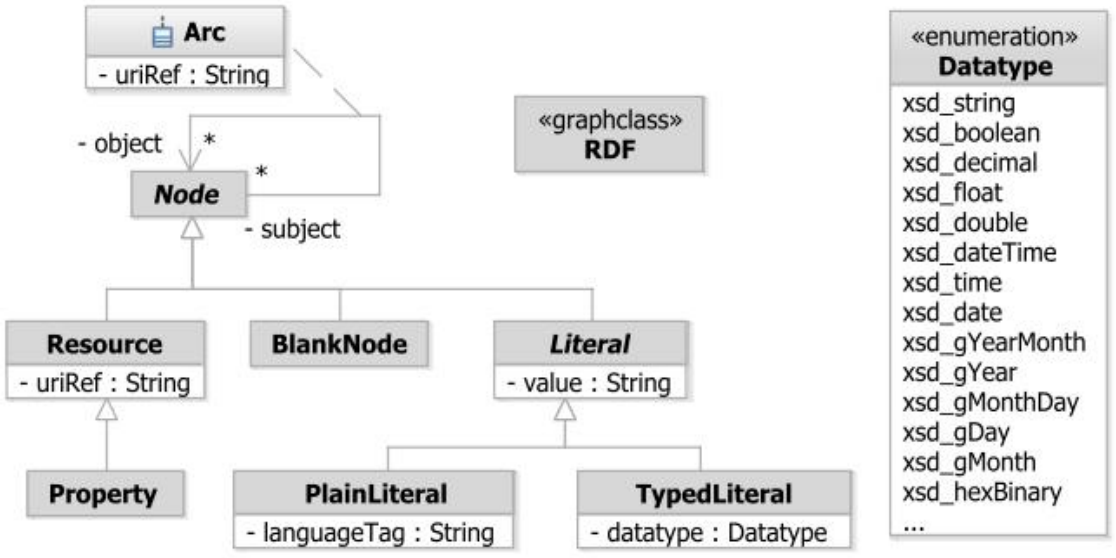


Abbildung 8.4: TGraph-Schema (Simple Mapping) für alle OWL-Ontologien [SE10].

OWL Element	**Beschreibung der Transformation in TGraph**
Ressourcen	Ressourcen werden in eine Instanz der Klasse `Resource` oder `Property` transformiert, deren Attribut `uriRef` die URI-Referenzen der Ressourcen trägt.
leere Knoten (Blank Nodes)	Leere Knoten werden in Instanzen der Klasse `BlankNode` transformiert.
ungetypte Literale	Ungetypte Literale werden in Instanzen der Klasse `PlainNode` transformiert, deren Attribut `value` das Literal und `languageTag` die Sprachangabe dieses Literals trägt.
getypte Literale	Getypte Literale werden in Instanzen der Klasse `Typed-Literal` transformiert, deren Attribute `value` das Literal und `datatype` den Typ dieses Literals trägt.
Vorkommen von Objekt-Property oder Daten-Property	Vorkommen von Objekt-Property oder Daten-Property werden in Instanzen der Kantenklasse `Arc` transformiert, deren Attribut `uriRef` die URI-Referenzen trägt.

Tabelle 8.2: Transformation über Simple Mapping von OWL-Ontologien in TGraphen [SE10]

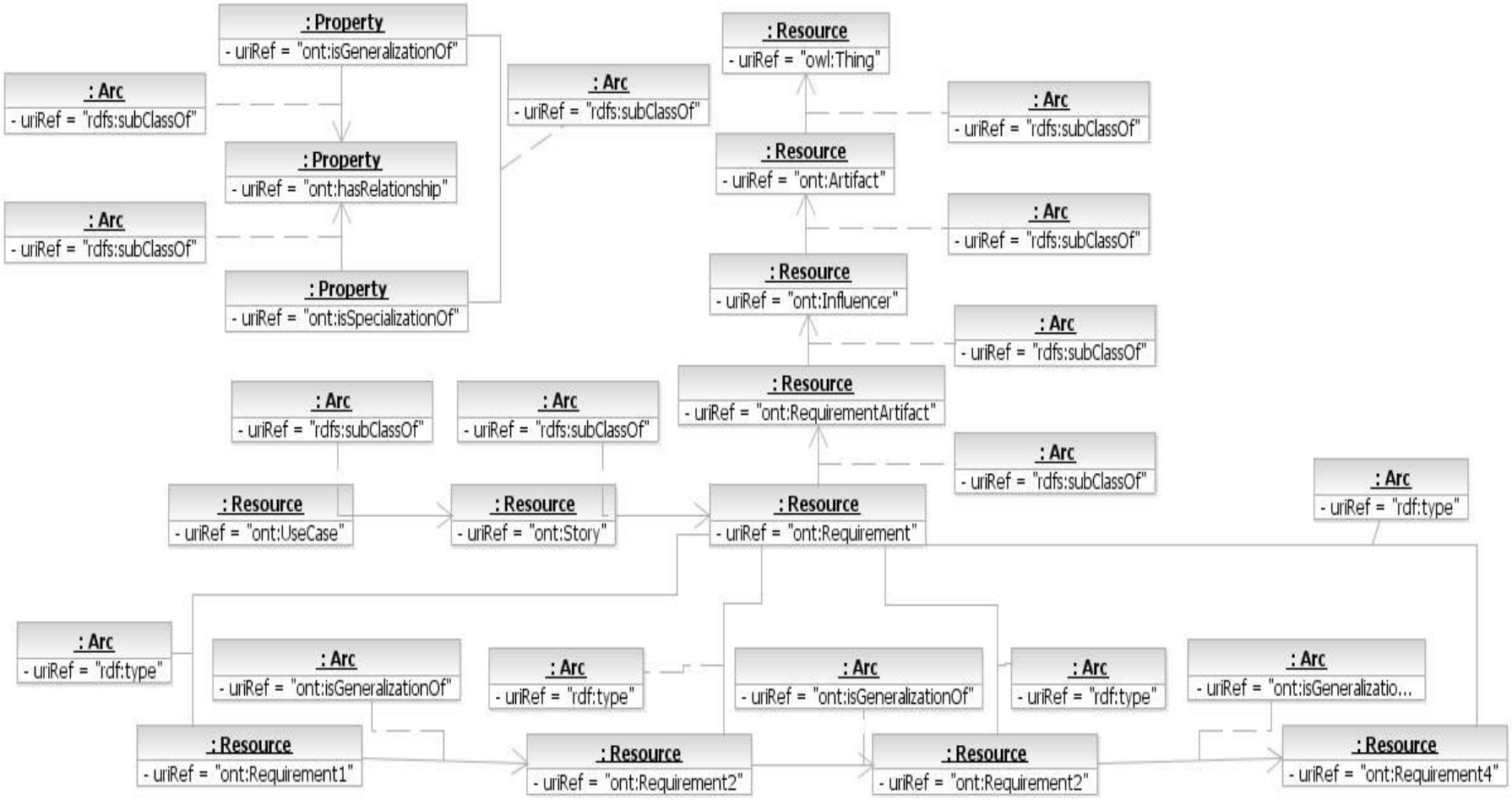


Abbildung 8.5: Der über Simple Mapping transformierte TGraph

8.2 Transformation von Anfragen

Die Transformation der Anfragen wird auch entsprechend der Transformation der OWL-Ontologien über Schema-Aware Mapping oder über Simple Mapping durchgeführt. Wird die OWL-Ontologie über Schema-Aware Mapping transformiert, wird entsprechend die SPARQL-Anfrage über Schema-Aware Mapping transformiert. Wenn die OWL-Ontologie über Simple Mapping in einen TGraphen transformiert wird, wird auch die SPARQL-Anfrage über Simple Mapping transformiert. Die API zur Transformation der Anfragen befinden sich noch im Entwicklungszustand, deswegen können nicht alle möglichen Transformationen der Sprachelemente durchgeführt werden. Die Operatoren UNION, OPTIONAL und FILTER von SPARQL können noch nicht in GReQL transformiert werden, weil TrOWL SPARQL-Anfragen mit den Elementen noch nicht parsen kann.

In diesem Abschnitt werden Transformationen erläutert, die in der zur Verfügung stehenden API vorhanden sind.

8.2.1 Transformation von SPARQL-Anfragen in GReQL-Anfragen

Wie schon in Kapitel 4 beschrieben, verfügt SPARQL über die vier Arten der Anfragen SELECT-, ASK-, DESCRIBE- und CONSTRUCT-Anfragen. Nur SELECT- und ASK-Anfragen können in GReQL-Anfragen übersetzt werden. Mit der zur Verfügung stehenden API können zur Zeit SELECT-Anfragen nur teilweise transformiert werden.

Die SELECT-Anfragen von SPARQL entsprechen den FWR-Ausdrücken von GReQL. Wie die einzelnen Teile der SELECT-Anfrage in den FWR-Ausdruck überführt werden, ist in Tabelle 8.3 dargestellt.

Teil der SELECT-Anfrage	**Teil des FWR-Ausdrucks**	**Beschreibung**
PREFIX	-	Für PREFIX gibt es keinen gleichwertigen GReQL-Ausdruck.
SELECT-Klausel	`report`	Variablen in der SELECT-Klausel werden in den `report`-Teil transformiert.
WHERE	`from` und `with`	Variablen in der WHERE-Klausel werden in die Deklaration des `from`-Teils transformiert. Tripelmuster der WHERE-Klausel werden in den `with`-Teil transformiert.

Tabelle 8.3: Transformation der SELECT-Anfragen in GReQL-Anfragen [SE10]

Transformation über Schema-Aware Mapping

Anhand des folgenden Beispiels wird die Transformation über Schema-Aware Mapping der SELECT-Anfrage in den FWR-Ausdruck erläutert.

```
1 select ?r
2 where { ?r rdf:type ont:Requirement
3         ont:Requirement1  ont:isGeneralizationOf ?r
4       }
```

Listing 8.1: SELECT-Anfrage

Die SELECT-Anfrage verwendet die Klasse `Requirement` und die Objekt-Property `isGeneralizationOf`. Die Anfrage soll alle Requirements zurückliefern, mit denen `Requirement1` über `isGeneralizationOf` verbunden ist.

Die Schema-Aware Mapping API transformiert die Anfragen in den FWR-Ausdruck wie folgt:

```
1 from r_var:V
2 with hasType(r_var, "Ont_numbersign_Requirement1")
3      and not isEmpty({@hasAttribute(thisVertex, "uriRef")
4          ? thisVertex.uriRef = "http://purl.org/ro/ont#Requirement1" : false}&
5          -->{Ont_numbersign_isGeneralizationOf1} r_var)
6 report hasType(r_var, "Data") ? r_var.value : r_var.uriRef
7 end
```

Die abgebildete Anfrage entspricht genau der implementierten Schema-Aware Transformation. Die Anfrage wird an den TGraphen aus Abbildung 8.3 gestellt. Da aus Einfachkeitsgründen die Klassennamen im TGraphen von den implementieren Namen abweichen, wird die Anfrage, um der Darstellung des TGraphen gerecht zu werden, wie folgt umgeschrieben:

```
1 from  r_var:V
2 with  hasType(r_var, "Requirement")
3       and not isEmpty({@hasAttribute(thisVertex, "uriRef")
4           ? thisVertex.uriRef = "http://purl.org/ro/ont#Requirement1": false}&
5           -->{isGeneralizationOf} r_var)
6 report hasType(r_var, "Data") ? r_var.value : r_var.uriRef
7 end
```

In dem `from`-Teil wird die Variable aus der SELECT-Klausel `r` als `r_var` der Kantenklasse (V) deklariert.

Die Zeile 2 der SELECT-Anfrage entspricht der Zeile 2 des FWR-Ausdrucks. Das Tripelmuster mit dem Prädikat `rdf:type` wird in die Funktionsanwendung `hasType` transformiert. Mit der Funktionsanwendung `hasType(r_var, „Requirement“)` wird geprüft, ob die Variable `r_var` vom Typ `Requirement` ist.

Das Tripelmuster `ont:Requirement1 ont:isGeneralizationOf ?r` in der Zeile 3 der SELECT-Anfrage entspricht dem gesamten Ausdruck `not isEmpty`, also die Zeilen 3-5 des FWR-Ausdrucks. Zum besseren Verständnis wird dieser Ausdruck hier etwas detaillierter beschrieben. Das Tripelmuster `ont:Requirement1 ont:isGeneralizationOf ?r` könnte als der folgende GReQL-Ausdruck dargestellt werden:

```
1  {@ thisVertex.uriRef = "http://purl.org/ro/ont#Requirement1"}&
2  --> {isGeneralizationOf} r_var
```

Dieser Ausdruck liefert aber keinen booleschen Wert zurück, sondern eine Menge von Knoten bzw. Requirements. Da in der `with`-Klausel nur boolesche Ausdrücke vorkommen dürfen, wird der Ausdruck mit der Funktion `isEmpty` in einen booleschen Ausdruck umgewandelt. Die Funktion liefert `true` zurück, wenn die Menge leer ist, sonst `false`. Wird der Funktion ein `not`-Operator vorangestellt, gibt die Funktion bei einer nicht leeren Menge `true` zurück. Da bei der Transformation der Prädikaten nicht zwischen der Daten-Property und der Objekt-Property unterschieden wird, und die `Data`-Knoten im TGraphen kein `uriRef`-Attribut besitzen, wird vor dem Zugriff auf den Wert des Attributs `uriRef` geprüft, ob dieses existiert. Nach diesen zusätzlichen Bedingungen wird der Ausdruck wie folgt ergänzt:

```
1 not isEmpty({@hasAttribute(thisVertex, "uriRef")
2       ? thisVertex.uriRef = "http://purl.org/ro/ont#Requirement1": false}&
3       -->{isGeneralizationOf} r_var)
```

Zuerst wird mit dem Ausdruck `hasAttribute(thisVertex, üriRef")` geprüft, ob der jeweilige Knoten das Attribut `uriRef` besitzt. Falls der Knoten das Attribut besitzt, wird geprüft, ob sein Wert gleich `"http://purl.org/ro/ont#Requirement1"` ist, sonst wird `false` zurückgegeben.

Die SELECT-Klausel entspricht dem `report`-Teil im FWR-Ausdruck. In dem `report`-Teil wird erstmal geprüft, ob die Variable `r_var` ein Literal bzw. `Data`-Knoten ist. Repräsentiert die Variable einen `Data`-Knoten, wird auf den Wert des Attributs `value` zugegriffen, sonst auf den Wert des Attributs `uriRef`.

Transformation über Simple Mapping

Die zweite Art der Transformation ist Simple Mapping. Die Transformation wird auch anhand der SELECT-Anfrage aus Listing 8.1 erläutert. Wird die OWL-Ontologie über Simple Mapping transformiert, wird entsprechend die SPARQL-Anfrage auch über Simple Mapping transformiert. Die SELECT-Anfrage wird wie folgt über Simple Mapping in die GReQL-Anfrage transformiert.

```
1  from r_var:V
2  with not isEmpty(
3           r_var -->{@thisEdge.uriRef = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"}&
4           {@hasAttribute(thisVertex, "uriRef") ?
5           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement" : false})
6       and not isEmpty(
7           {@hasAttribute(thisVertex, "uriRef") ?
8           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement1" : false}&
9           -->{@thisEdge.uriRef = "http://purl.org/ro/ont#isGeneralizationOf"} r_var)
10 report hasType(r_var, "Resource") ? r_var.uriRef :
11        hasType(r_var, "TypedLiteral") ? r_var.value + "^^" + r_var.datatype :
12        hasType(r_var, "PlainLiteral") ? r_var.value + r_var.languageTag :
13        hasType(r_var, "BlankNode") ? r_var : "error"
14 end
```

Die dargestellte GReQL-Anfrage wird an den über Simple Mapping transformierten TGraphen aus Abbildung 8.5 gestellt. Die Zeilen 3-5 des FWR-Ausdrucks entsprechen dem Tripelmuster `?r rdf:type ont:Requirement`. In den Zeilen werden Knoten gesucht, die mit dem Knoten verbunden sind, dessen Attributwert `uriRef="http://purl.org/ro/ont#Requirement"` ist. Dabei muss die verbindende Kante den Attributwert `uriRef="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"` besitzen.

Die Ausdrücke in den Zeilen 6-9 sind nach dem gleichen Prinzip, wie beim Schema Aware Mapping aufgebaut. Die Zeilen stellen das in GReQL transformierte Tripelmuster `ont:Requirement1 ont:isGeneralizationOf ?r` dar. Es werden Knoten gesucht, mit denen der Knoten mit `uriRef="http://purl.org/ro/ont#Requirement1"` über die Kante mit `uriRef="http://purl.org/ro/ontisGeneralizationOf"` verbunden ist.

Der `report`-Teil der Anfrage ist im Vergleich zur über Schema-Aware Mapping transformierten Anfrage relativ umfangreich. Es liegt daran, dass es vier Arten der Knoten `Resource`, `TypedLiteral`, `PlainLiteral` und `BlankNode` gibt, die unterschiedliche Attribute besitzen. Bevor auf die Attribute zugegriffen wird, muss erst geprüft werden, ob diese existieren. Dementsprechend wird zuerst der Typ des Knotens mit Hilfe der Funktion `hasType` geprüft, erst dann erfolgt der Zugriff auf den Attributwert. Also zuerst wird geprüft, ob die Variable `r_var` den `Resource`-Knoten repräsentiert, falls ja wird auf den Wert des Attributs `uriRef` zugegriffen, sonst wird geprüft, ob die Variable vom Typ `TypedLiteral` ist und so weiter.

Die detaillierte Beschreibung der Transformationen ist in [SE10] zu finden.

8.3 Modifikation von GReQL-Anfragen

Wie schon beschrieben, werden bei der Auswertung der GReQL-Anfragen die zwei unterschiedlichen Transformationsverfahren eingesetzt. Zusätzlich zu diesen Transformationsverfahren kann noch ein Modifikationsverfahren verwendet werden, das die GReQL-Anfragen so modifizieren soll, dass auf den Einsatz eines Reasoners[1] verzichtet werden kann. Zurzeit können mit dem Modifikationsverfahren die Eigenschaften der Objekt-Properties „transitiv“, „symmetrisch“, „äquivalent zu“ und die `subPropertyOf`-Beziehungen zwischen Properties behandelt werden, also die sogenannten „besonderen Eigenschaften“.

Besitzen in der Anfrage verwendete Objekt-Properties einer der oben genannten Eigenschaften, werden diese in der GReQL-Anfrage entsprechend modifiziert.

Die Modifikation der GReQL-Anfragen ist wie folgt definiert:

```
Gegeben sei der GReQL-Ausdruck -->{P},

falls die Objekt-Property P transitiv:
-->{P} => -->{P}+

falls die Objekt-Property P symmetrisch:
-->{P} => <->{P}

falls die Objekt-Property P äquivalent zu Q:
-->{P} => -->{P,Q}

falls die Objekt-Property P invers zu Q:
-->{P} => -->{P} | <--{Q}

falls Q Unterproperty der Objekt-Property P:
-->{P} => -->{P} | -->{Q}
```

Wie das folgende Beispiel zeigen wird, ist die Modifikation der GReQL-Anfragen nur sinnvoll, wenn die OWL-Ontologie ohne Einsatz eines Reasoners ausgeführt wird. Die folgende GReQL-Anfrage wird an den über Schema-Aware Mapping mit und ohne Reasoner transformierten TGraphen aus Abbildungen 8.3 und 8.6 gestellt.

```
from r:V{Requirement}
with not isEmpty({@thisVertex.uriRef="ont:Requirement1"}&
     --> {isGeneralizationOf} r)
report r.uriRef
```

Aus Platzgründen fehlen in der Abbildung 8.6 die nicht für die Anfrage relevanten Verbindungen über `hasRelationship`. Im ersten Fall liefert die Anfrage die Requirements 2, 3 und 4 zurück, im zweiten Fall nur das Requirement 2.

Da die Anfrage „transitive Property“`isGeneralizationOf` verwendet, soll sie alle Requirements zurückliefern, mit denen das Requirement1 über `isGeneralizationOf` direkt und indirekt verbunden ist. Dementsprechend ist das Ergebnis im zweiten Fall unvollständig.

Wird das Modifikationsverfahren angewendet, wird die Anfrage wie folgt modifiziert:

```
from r:V{Requirement}
with not isEmpty({@thisVertex.uriRef="ont:Requirement1"}&
     (--> {isGeneralizationOf}|<--{isSpecializationOf})+ r)
report r.uriRef
```

[1] In diesem Rahmen stehen Reasoner für ontologiebasierte Inferenz-Werkzeuge zur Schlussfolgerung des impliziten Wissens

Die transitive Property `isGeneralizationOf` besitzt die inverse Property `isSpecialitzationOf`, die auch transitiv ist. Dementsprechend wird hier die Regel 4 (inverse Properties) und 1 (transitive Properties) angewendet. Der Ausdruck `-->{isGeneralizationOf} r` wird zum Ausdruck `(-->{isGeneralizationOf}|<--{isSpecializationOf})+ r` geändert. Wird die modifizierte Anfrage an den über Schema-Aware Mapping mit und ohne Reasoner transformierten TGraphen gestellt, wird sie in beiden Fällen dasselbe Ergebnis Requirements 2, 3 und 4 zurückliefern. Wie man sieht, bringt die Modifikation der GReQL-Anfragen keinen Vorteil, wenn die OWL-Ontologie in den TGraphen über Reasoner transformiert wird.

Dasselbe gilt für die GReQL-Anfragen, die an die über Simple Mapping mit und ohne Reasoner transformierte TGraphen gestellt werden.

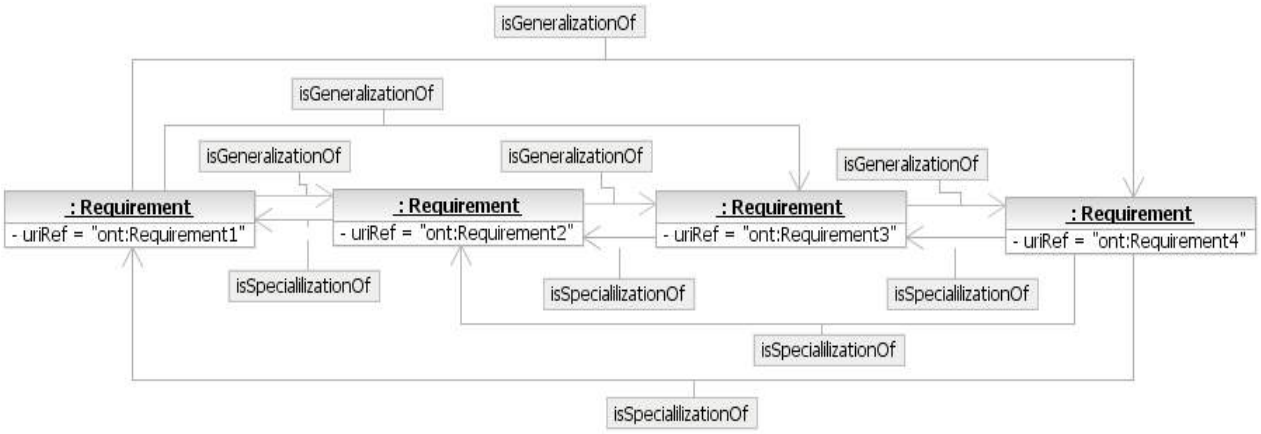


Abbildung 8.6: Der über Schema-Aware Mapping mit Reasoning transformierte TGraph

8.3.1 Transformation von GReQL-Anfragen in SPARQL-Anfragen

In Anlehnung an [SE10] wird hier die Transformation von GReQL in SPARQL erläutert. Um die Transformation zu ermöglichen müssen auch wie bei der Transformation von SPARQL in GReQL bestimmte Einschränkungen eingehalten werden. Es können nur die FWR- und Pfad-Existenz-Ausdrücke (Path Existence Expression) in SPARQL transformiert werden. Für die FWR-Ausdrücke gelten die folgenden Einschränkungen [SE10]:

- Im `from`-Teil von FWR dürfen nur Variablen verwendet werden, die als Domänen Knotenklassen haben.
- Der boolesche Ausdruck im `with`-Teil darf nur die Pfad-Existenz-Ausdrücke enthalten und die einzelnen Funktionsaufrufe müssen über die logischen Operatoren `and` oder `or` miteinander verbunden werden.
- Reguläre Pfad-Ausdrücke dürfen keine Kanten-Beschreibungen und Iterationen enthalten.
- Im `report`-Teil dürfen nur Variablen und Attribut-Zugriffe vorkommen.

Anhand des folgenden FWR-Ausdrucks wird die Transformation verdeutlicht:

```
from r,c:V{Requirement}
with r -->{isInConflictWith} | <->{isExlusionOf} c
      and r.uriRef = "http://purl.org/ro/ont#Requirement0"
report c.uriRef
end
```

Variablen aus dem `from`-Teil werden in Tripelmuster der `WHERE`-Klausel transformiert. Die Variablen werden in Subjekte, die Domänen der Variablen in Objekte transformiert und die Prädikate dieses Tripelmusters sind `rdf:type`.

Der transformierte `from`-Teil des Beispiels sieht wie folgt aus:

```
WHERE {
       ?r rdf:type ont:Requirement.
       ?c rdf:type ont:Requirement.
         ...
```

Zuerst soll der FWR-Ausdruck in die disjunktive Normalform (DNF) gebracht werden, dabei werden die folgenden Substitutionen durchgeführt [SE10]:

```
1. Die Substitution von einfachen Pfad-Beschreibungen mit der unbestimmten Richtung:
    <-> => (-->|<--)

2. Die Substitution von optionalen Pfad-Beschreibungen durch Alternativen:
    seq1 [seq2] seq3 => (seq1 seq3 | seq1 seq2 seq3)

3. Die Anwendung des Distributivgesetzes:
   (seq1 | seq2 |...| seqp) (seqp+1 | seqp+2 |...| seqq)
=> (seq1 seqp+1 | seq1 seqp+2 |...| seq1 seqq
        | seq2 seqp+1 | seq2 seqp+2 |...| seq2 seqq
        |...|
        | seqp seqp+1 | seqp seqp+2 |...| seqp seqq)
```

Dementsprechend wird nach diesen Ersetzungen jeder Pfad-Existenz-Ausdruck wie folgt aussehen, dabei ist $seq_{i,\ i=1..m}$ eine sequenzielle Pfad-Beschreibung, die keine Alternativen und Optionen enthält. v und w sind Variablen.

```
v (seq1 | seq2 |...| seqm) w
```

Jeder in DNF beschriebene Pfad-Existenz-Ausdruck wird durch mehrere Pfad-Existenz-Ausdrücke wie folgt ersetzt [SE10]:

```
(v seq1 w or v seq2 w or ... or v seqm w)
```

Nach den Modifikationen sieht der `with`-Teil des Beispiels wie folgt aus:

```
1. with  r (-->{isInConflictWith}| -->{isExlusionOf}|<--{isExlusionOf}) c
         and r.uriRef = "http://purl.org/ro/ont#Requirement0"

2. with   r (-->{isInConflictWith}c or r-->{isExlusionOf}c or r<--{isExlusionOf}) c
          and r.uriRef = "http://purl.org/ro/ont#Requirement0"
```

Der `with`-Teil wird in die WHERE-Klausel transformiert, dabei werden `and` Operatoren durch join und `or` Operatoren durch UNION ersetzt.

Jeder Pfad-Existenz-Ausdruck wird in ein Tripelmuster wie folgt transformiert [SE10]. `?p`, `?q` und `?r` sind Variablen und `<PreE>` ist eine Instanz von `rdf:Property`, die der jeweiligen Kantenklasse entspricht.

```
1. v --> w                    =>    ?v ?p ?w.
2. v <-- w                    =>    ?w ?p ?v.
3. v <-> w                    =>    ?v ?p ?w. ?w ?q ?v.
4. v -->{EdgeClass} w         =>    ?v <PreE> ?w.
5. v -->{EdgeClass} x <-> w   =>    ?v <PreE> ?x. ?x ?p ?w. ?w ?q ?x.
6. v <->{EdgeClass} <-- w     =>    ?v <PreE> ?r. ?v <PreE> ?r. ?w ?p ?r.
```

Die Pfad-Existenz-Ausdrücke des Beispiels werden wie folgt transformiert:

```
WHERE { ...
          { ?r <isInConflictWith> ?c }
       UNION
          { ?r <isExlusionOf> ?c }
       UNION
          { ?c <isExlusionOf> ?r }
       ...
       }
```

Jeder Funktion-Aufruf wird in einen äquivalenten FILTER-Operator transformiert. Die Attribut-Zugriffe werden wie folgt umgewandelt. <PreA> ist eine Instanz von `rdf:Property`, die dem Attribut `attr` entspricht [SE10].

```
v.attr = value  => v <PreA> ?lit. FILTER(?lit = value)
```

Kommen im `report`-Teil die Attribut-Zugriffe vor, werden entsprechende Tripel-Muster in die WHERE-Klausel eingefügt.

Das Endresultat der transformierten Anfrage sieht wie folgt aus. Die Anfrage entspricht genau der Transformation über die greql2sparql-API.

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ont:<http://purl.org/ro/ont#>

SELECT ?o0
WHERE {
  ?c ont:uriRef ?o0 .
  ?r rdf:type ont:Requirement .
  ?c rdf:type ont:Requirement .
  {
    {
      {
        ?r ont:isInConflictWith ?c .
      }
      UNION
      {
        {
          {
            ?r ont:isExlusionOf ?c .
          }
          UNION
          {
            ?c ont:isExlusionOf ?r .
          }
        }
      }
    }
    FILTER ( ?o1 = "http://purl.org/ro/ont#Requirement0" )
```

```
    ?r ont:uriRef ?o1 .
  }
}
```

Kapitel 9

Anfragen

9.1 Herleitung der SPARQL-Anfragen

Einerseits soll in dieser Studienarbeit geprüft werden, ob Anfragen der OWL-Ontologie über GReQL-Anfragen effizienter als über SPARQL-Anfragen sind. Andererseits soll die effizienteste Variante unter den vier oben beschriebenen Transformationsverfahren Schema-Aware Mapping mit Reasoning, Simple Mapping mit Reasoning, Schema-Aware Mapping mit Modifikationsverfahren oder Simple Mapping mit Modifikationsverfahren ermittelt werden. Wie schon erwähnt, es können nicht alle Sprachelemente von SPARQL evaluiert werden, da das Tool noch nicht alle Elemente einlesen bzw. transformieren kann. Deswegen konzentriert sich die Evaluierung der Anfragen auf die sogenannten „besonderen Eigenschaften“ „transitiv“, „symmetrisch“, „äquivalent zu“ „invers zu“und die `subPropertyOf`-Beziehungen zwischen Properties, die das Modifikationsverfahren ohne Einsatz eines Reasoners behandeln kann. Damit steht das Modifikationsverfahren im Mittelpunkt der Evaluierung. Es wird vermutet, dass durch Modifikationsverfahren und gleichzeitigen Verzicht auf Reasoner die Ausführung der Anfragen effizienter sein kann als mit dem Einsatz eines Reasoners.

In der Tabelle 9.1 sind die Testanfragen kurz beschrieben. Die Anfragen 1-5 verwenden Properties, die keine Superproperties sind. Es sind also Properties oder Unterproperties. Die Anfrage 1 ist die einfachste Anfrage, die eine einfache Property verwendet. Das heißt die Property hat keine der „besonderen Eigenschaften“. Da die Property keine der „besonderen Eigenschaften“ besitzt, bleibt die transformierte GReQL-Anfrage nach der Modifizierung unverändert.

Die Anfragen 2-5 verwenden Properties, die jeweils eine der „besonderen Eigenschaften“ besitzen. Damit werden die transformierten GReQL-Anfragen nach dem Einsatz des Modifikationsverfahrens entsprechend modifiziert.

Die Anfragen 6-10 verwenden Superproperties. Die in den Anfragen 6-8 verwendeten Individuen werden auch in der Original-Ontologie (Ontologie ohne inferierte Daten) über die entsprechenden Superproperties verbunden. In der Anfrage 6 wird eine einfache Superproperty benutzt. Die in den Anfragen 7-10 verwendeten Properties besitzen jeweils eine der „besonderen Eigenschaften“.

Bei den Anfragen 11-13 werden einfache Superproperties benutzt, also Superproperties, die keine „besonderen Eigenschaften“ besitzen. Die in den Anfragen verwendeten Individuen werden in der Ontologie über die Unterproperties dieser Superproperties verbunden. Die Unterproperties sind dabei einfach, symmetrisch oder transitiv.

Die Anfragen 14-16 verwenden symmetrische Superproperties. In der Ontologie werden die entsprechenden Individuen über einfache, symmetrische oder transitive Unterproperties dieser Superproperties verbunden.

In den Anfragen 17-19 werden transitive Superproperties benutzt. In der Ontologie werden die für die Anfragen relevanten Individuen über die Unterproperties dieser Superproperties verbunden. Die Unterproperties sind dabei einfach, symmetrisch oder transitiv.

Die Anfragen 1-5 und 11-13 werden an die Requirements-Ontologie und die Anfragen 6-10 und 14-19 an die Family-Ontologie gestellt.

Im Folgenden werden die Anfragen einzeln vorgestellt und genauer erläutert. Zuerst wird ein Ontologie-Ausschnitt dargestellt, auf den sich die jeweilige Anfrage bezieht. In dem Ontologie-Ausschnitt werden Verbindungen, die durch den Einsatz des Reasoners[1] entstanden sind durch gestrichelte Pfeile dargestellt. Durchgehende Pfeile repräsentieren Verbindungen aus der Original-Ontologie (vor dem Einsatz des Reasoners). Danach folgen eine SPARQL-Anfrage und die über Schema-Aware Mapping transformierten GReQL-Anfragen ohne Modifikation. Anschließend werden der Ontologie-Ausschnitt und die Idee der jeweiligen Anfrage erläutert und die modifizierten GReQL-Anfragen dargestellt.

Die über Simple Mapping transformierten GReQL-Anfragen ohne und mit der Modifikation befinden sich im Anhang B.

Anfrage	**Eigenschaften der in der Anfrage verwendeten Properties**	**Eigenschaften der für die Anfrage relevanten Properties in der Ontologie**
1	Einfache Property 1	Einfache Property 1
2	Symmetrische Property 1	Symmetrische Property 1
3	Transitive Property 1	Transitive Property
4	inverse zu Property 2 Property 1	Inverse zu Property 1 Property 2
5	Property 1	Äquivalente Property 2 zu Property 1
6	Einfache Superproperty 1	Einfache Superproperty 1
7	Symmetrische Superproperty 1	Symmetrische Superproperty 1
8	Transitive Superproperty 1	Transitive Superproperty 1
9	Superproperty 1	Inverse zu Superproperty 1 Property 2
10	Superproperty 1	Äquivalente zu Superproperty 1 Superproperty 2
11	Einfache Superproperty 1	Einfache Unterproperty von Superproperty 1
12	Einfache Superproperty 1	Symmetrische Unterproperty von Superproperty 1
13	Einfache Superproperty 1	Transitive Unterproperty von Superproperty 1
14	Symmetrische Superproperty 1	einfache Unterproperty von Superproperty 1
15	Symmetrische Superproperty 1	Symmetrische Unterproperty von Superproperty 1
16	Symmetrische Superproperty 1	Transitive Unterproperty von Superproperty 1
17	Transitive Superproperty 1	Einfache Unterproperty von Superproperty 1
18	Transitive Superproperty 1	Symmetrische Unterproperty von Superproperty 1
19	Transitive Superproperty 1	Transitive Unterproperty von Superproperty 1

Tabelle 9.1: Beschreibung der Anfragen

Alle im Folgenden erläuterten Anfragen werden an den Ontologien aus Abbildungen 7.7 und 7.9 gestellt.

[1] Zur Ausführung der Ontologien wurde der Pellet-Reasoner (http://clarkparsia.com/pellet/) eingesetzt.

Anfrage 1

Abbildung 9.1: Requirements-Ontologie-Ausschnitt

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement1 ont:hasRefinementSource ?r.
      }
```

```
//Schema-Aware-transfomierte GReQL-Anfrage mit oder ohne Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef")
          ? thisVertex.uriRef = "http://purl.org/ro/ont#Requirement1" : false}&
          -->{Ont_numbersign_hasRefinementSource1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Die Anfrage verwendet die einfache Property `hasRefinementSource` und soll Refinement Sources von `Requirement1` zurückliefern. In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.1 ist `Requirement1` über das Vorkommen dieser Property mit `Requirement2` verbunden. Da die Property `hasRefinementSource` keine Eigenschaften besitzt, werden nach dem Einsatz des Reasoners keine neuen Verbindungen eingefügt. Entsprechend bleibt die GReQL-Anfrage nach der Modifikation unverändert.

Anfrage 2

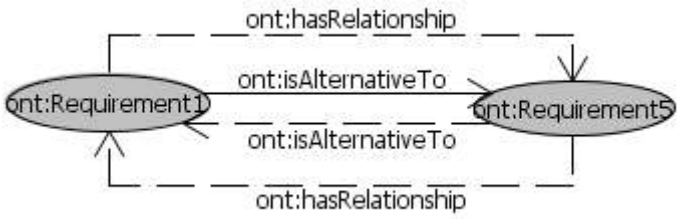


Abbildung 9.2: Requirements-Ontologie-Ausschnitt

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement5 ont:isAlternativeTo ?r.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement5" : false}&
           -->{Ont_numbersign_isAlternativeTo1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.2 ist `Requirement1` mit `Requirement5` über das Vorkommen der symmetrischen Property `isAlternativeTo` verbunden.

Um die Eigenschaften der Symmetrie zu testen, wird in der Anfrage die umgekehrte Verbindung `ont:Requirement5 ont:isAlternativeTo ?r` gesucht. Diese Verbindung entsteht erst nach dem Einsatz des Reasoners, die in Abbildung 9.2 durch gestrichelte Pfeile dargestellt ist. Aufgrund der neuen entstandenen Verbindung in der Ontologie soll die SELECT-Anfrage bzw. GReQL-Anfrage ohne Modifikation das erwartete Ergebnis `Requirement1` zurückliefern.

Die modifizierte GReQL-Anfrage wird an die Original-Ontologie gestellt. Das heißt, die Ontologie wird ohne Einsatz des Reasoners ausgeführt und damit werden keine inferierten Beziehungen in die Ontologie eingefügt. Hier werden Maßnahmen über das Modifikationsverfahren in der Anfrage selbst vorgenommen. Da die Property `isAlternativeTo` symmetrisch ist, wird die Kante `-->{Ont_numbersign_isAlternativeTo1}` in der GReQL-Anfrage zur symmetrischen Kante `<->{Ont_numbersign_isAlternativeTo1}` geändert.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement5" : false}&
           <->{Ont_numbersign_isAlternativeTo1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Anfrage 3

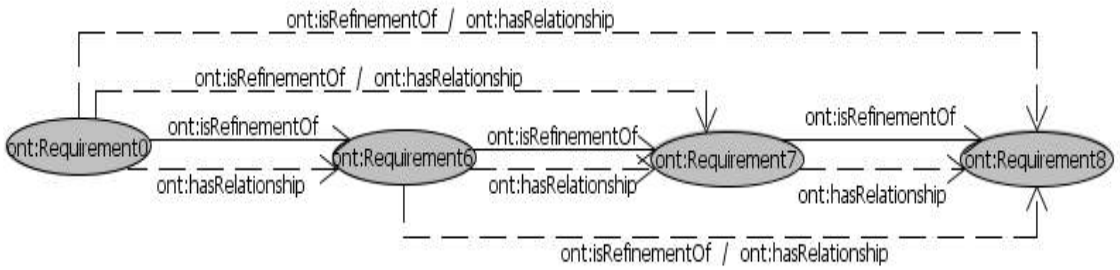


Abbildung 9.3: Requirements-Ontologie-Ausschnitt

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement0 ont:isRefinementOf ?r.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
          thisVertex.uriRef = "http://purl.org/ro/ont#Requirement0" : false}&
          -->{Ont_numbersign_isRefinementOf1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

`Requirement0`, `Requirement6`, `Requirement7` und `Requirement8` wurden in dem Ontologie-Ausschnitt aus Abbildung 9.3 über die Vorkommen der transitiven Property `isRefinementOf` nacheinander verbunden, um die Eigenschaften der Transitivität zu testen.

Die Anfrage 3 gibt alle Requirements zurück, mit denen `Requirement0` über `isRefinementOf` direkt und indirekt verbunden ist.

Da `isRefinementOf` transitiv ist, wird nach dem Einsatz des Reasoners unter anderen die neuen für die Anfrage relevanten Verbindungen über `isRefinementOf` zwischen `Requirement0` und `Requirement7`, `Requirement0` und `Requirement8`, `Requirement6` und `Requirement8` eingefügt.

Bei der Modifikation der GReQL-Anfrage wird die Kante `-->{Ont_numbersign_isRefinementOf1}` entsprechend zu `-->{Ont_numbersign_isRefinementOf1}+` modifiziert.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement0" : false}&
           -->{Ont_numbersign_isRefinementOf1}+
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Anfrage 4

Abbildung 9.4: Requirements-Ontologie-Ausschnitt

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:UseCase0 ont:describesRequirement ?r.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#UseCase0" : false}&
           -->{Ont_numbersign_describesRequirement1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Die Anfrage konzentriert sich auf inverse Properties. Sie verwendet die inverse Property `describesRequirement` von `isDescribedByUseCase`. Es sollen Requirements zurückgegeben werden, mit denen `UseCase0` über `describesRequirement` verbunden ist.

Um die Eigenschaften der inversen Properties zu testen, wurden in der Ontologie aus Abbildung 9.4 `Requirement0` mit `UseCase0` über das Vorkommen der inversen Property `isDescribedByUseCase` von

`describesRequirement` verbunden. Die Verbindung über `describesRequirement` wird erst nach dem Einsatz des Reasoners der Ontologie hinzugefügt.

Nach der Regel der Inversität wird die GReQL-Anfrage bei der Modifikation um den Ausdruck `| <-- {Ont_numbersign_isDescribedByUseCase1}` erweitert.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#UseCase0" : false}&
           (-->{Ont_numbersign_describesRequirement1}
             |<--{Ont_numbersign_isDescribedByUseCase1}
           )
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
```

Anfrage 5

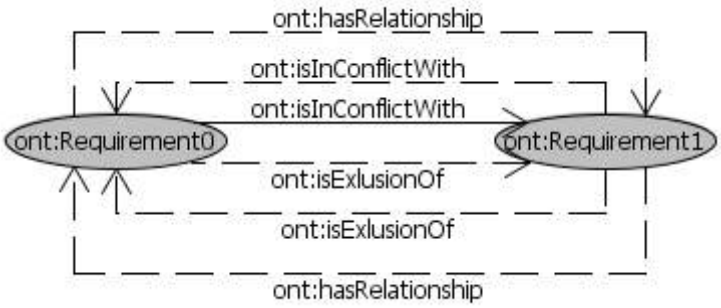


Abbildung 9.5: Requirements-Ontologie-Ausschnitt

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement0 ont:isExclusionOf ?r.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement0" : false}&
           -->{Ont_numbersign_isExclusionOf1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Die Anfrage fokussiert sich auf Eigenschaften der äquivalenten Properties. In der Anfrage wird die zu `isInConflictWith` äquivalente Property `isExclusionOf` verwendet. Die Anfrage soll Requirements zurückliefern, mit denen `Requirement0` über das Vorkommen von `isExclusionOf` oder über das Vorkommen ihrer äquivalenten Property `isInConflictWith` verbunden ist.

In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.5 ist `Requirement0` mit `Requirement1` über `isInConflictWith` verknüpft. Nach dem Einsatz des Reasoners wird aufgrund der Äquivalenz der beiden Properties `isInConflictWith` und `isExclusionOf` die zweite Verbindung über `isExclusionOf`

eingefügt. Da die Properties noch symmetrisch sind werden dieselben Verbindungen in beiden Richtungen definiert.

Nach der Regel der Symmetrie und Äquivalenz wird in der GReQL-Anfrage die Kante `-->{Ont_numbersign_isExclusionOf1}` zu `<->{Ont_numbersign_isExclusionOf1,Ont_numbersign_isInConflictWith1}` modifiziert.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement0" : false}&
           <->{Ont_numbersign_isExclusionOf1, Ont_numbersign_isInConflictWith1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Anfrage 6

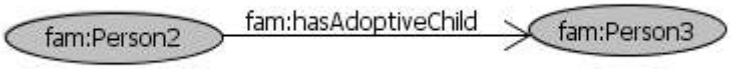


Abbildung 9.6: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person2 fam:hasAdoptiveChild ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person2" : false}&
           -->{Family_numbersign_hasAdoptiveChild1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

In der Anfrage wird die einfache Superproperty `hasAdoptiveChild` verwendet. Die Anfrage soll Personen zurückgeben, mit denen `Person2` über `hasAdoptiveChild` verbunden ist. Auch in der Ontologie aus Abbildung 9.6 ist `Person2` mit `Person3` über das Vorkommen derselben Superproperty `hasAdoptiveChild` verbunden. Nach dem Einsatz des Reasoners werden keine neuen Verbindungen in den relevanten Ontologie-Ausschnitt eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
         thisVertex.uriRef =
```

```
            "http://www.semanticweb.org/ontologies/2011/6/Family#Person2":false}&
         (-->{Family_numbersign_hasAdoptiveChild1}
           | -->{Family_numbersign_hasAdoptiveSon1}
           |-->{Family_numbersign_hasAdoptiveDaughter1}
         )
       p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 7

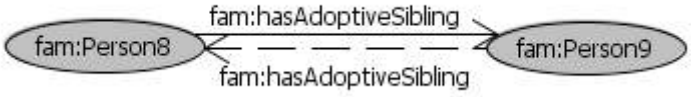


Abbildung 9.7: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person9 fam:hasAdoptiveSibling ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person9" : false}&
           -->{Family_numbersign_hasAdoptiveSibling1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Die Anfrage ist ähnlich der Anfrage 2 mit dem Unterschied, dass die Anfrage 7 eine Property (Superproperty) verwendet, die Unterproperties besitzt. Die Anfrage verwendet die symmetrische Superproperty `hasAdoptiveSibling` und soll alle Adoptivgeschwister von `Person9` zurückgegeben. In dem relevanten Ontologie-Ausschnitt ist `Person8` mit `Person9` über das Vorkommen der Property `hasAdoptiveSibling` verbunden (siehe Abbildung 9.7). Nach dem Einsatz des Reasoners werden zwar neue Verbindungen in die Ontologie eingefügt, die aber für die Anfrage irrelevant sind.

Bei der Modifizierung der GReQL-Anfrage wird die Kante `-->{Family_numbersign_hasAdoptiveSibling1}` als die symmetrische Kante `<->{Family_numbersign_hasAdoptiveSibling1}` definiert.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
            "http://www.semanticweb.org/ontologies/2011/6/Family#Person9":false}&
           <->{Family_numbersign_hasAdoptiveSibling1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 8

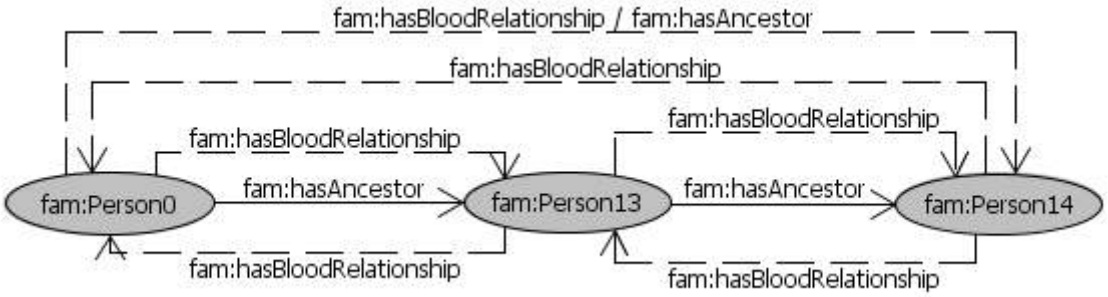


Abbildung 9.8: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person0 fam:hasAncestor ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person0" : false}&
           -->{Family_numbersign_hasAncestor1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

In der Anfrage wird die transitive Property `hasAncestor` verwendet. Die Anfrage soll alle Vorfahren von `Person0` zurückliefern. Um die transitiven Properties zu testen, werden in der Ontologie aus Abbildung 9.8 `Person0`, `Person13` und `Person14` über `hasAncestor` nacheinander verbunden. Nach dem Einsatz des Reasoners wird die Ontologie unter anderen um die Verbindungen zwischen `Person0` und `Person14` über `hasAncestor` ergänzt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
         thisVertex.uriRef =
           "http://www.semanticweb.org/ontologies/2011/6/Family#Person0" : false}&
         (-->{Family_numbersign_hasAncestor1} | -->{Family_numbersign_hasFemaleAncestor1}
           |(-->{Family_numbersign_hasParent1}
             |-->{Family_numbersign_hasFather1, Family_numbersign_hasDad1}
             |-->{Family_numbersign_hasMother1}
             |-->{Family_numbersign_hasDad1, Family_numbersign_hasFather1}
             |<--{Family_numbersign_hasChild1} | -->{Family_numbersign_hasDaugther1}
             | -->{Family_numbersign_hasSon1}
            )
           | -->{Family_numbersign_hasMaleAncestor1}
          )+
        p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
```

```
end
```

Anfrage 9

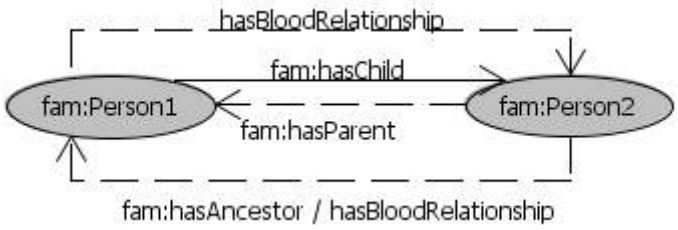


Abbildung 9.9: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person2 fam:hasParent ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person2" : false}&
           -->{Family_numbersign_hasParent1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Die Anfrage ist ähnlich der Anfrage 1, mit dem Unterschied, dass in der Anfrage eine Property (Superproperty) verwendet wird, die Unterproperties besitzt. In der Anfrage wird die inverse Property `hasParent` von `hasChild` benutzt. Die Anfrage soll Eltern von `Person2` zurückgeben.

In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.9 ist `Person1` mit `Person2` über die Property `hasChild` verbunden. Da `hasChild` eine Unterproperty von `hasAncestor` und `hasBloodRelationship` ist und invers zu `hasParent` ist, werden in die Ontologie nach dem Einsatz des Reasoners entsprechende Verbindungen eingefügt.

Die Schema-Aware transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
         thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person2" : false}&
         (-->{Family_numbersign_hasParent1}
           |-->{Family_numbersign_hasFather1, Family_numbersign_hasDad1}
           |-->{Family_numbersign_hasMother1}
           |-->{Family_numbersign_hasDad1, Family_numbersign_hasFather1}
           |<--{Family_numbersign_hasChild1} | -->{Family_numbersign_hasDaugther1}
           |-->{Family_numbersign_hasSon1}
         )
        p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
```

```
end
```

Anfrage 10

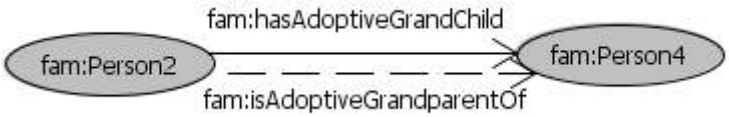


Abbildung 9.10: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person2 fam:isAdoptiveGrandparentOf ?p
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person2" : false}&
           -->{Family_numbersign_isAdoptiveGrandparentOf1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Das Ziel der Anfrage ist äquivalente Superproperties zu testen. Die Anfrage verwendet die zu `hasAdoptiveGrandchild` äquivalente Superproperty `isAdoptiveGrandparentOf`.

In der Original-Ontologie ist `Person2` über `hasAdoptiveGrandchild` mit `Person4` verbunden (siehe Abbildung 9.10). Erst nach dem Einsatz des Reasoners wird in die Ontologie die Verbindung zwischen `Person2` und `Person4` über `isAdoptiveGrandparentOf` eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
            thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person2" :false}&
            -->{Family_numbersign_isAdoptiveGrandparentOf1,
            Family_numbersign_hasAdoptiveGrandchild1}
          p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 11

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement2 ont:hasRelationship ?r.
      }
```

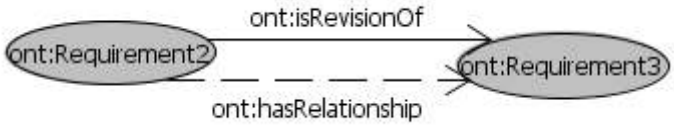


Abbildung 9.11: Requirements-Ontologie-Ausschnitt

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement2" : false}&
           -->{Ont_numbersign_hasRelationship1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Die Anfragen 11-13 sind ähnlich aufgebaut. Sie verwenden dieselbe Superproperty `hasRelationship`. Der Unterschied liegt darin, dass die in den Anfragen verwendeten Individuen in der Ontologie über unterschiedliche Unterproperties von `hasRelationship` verbunden sind.

Die Anfrage 11 soll Requirements zurückliefern, mit denen `Requirement2` über `hasRelationship` oder über die Vorkommen ihrer Unterproperties verbunden ist. In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.11 ist `Requirement2` mit `Requirement3` über `isRevisionOf` verbunden. `isRevisionOf` ist eine einfache Unterproperty von `hasRelationship`, die erst nach dem Einsatz des Reasoners in die Ontologie eingefügt wird.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
            thisVertex.uriRef = "http://purl.org/ro/ont#Requirement2" : false}&
            (-->{Ont_numbersign_hasRelationship1} | <->{Ont_numbersign_isCoexistentWith1}
              |-->{Ont_numbersign_isSpecializationOf1}+ | -->{Ont_numbersign_isRevisionOf1}
              |<->{Ont_numbersign_isInConflictWith1, Ont_numbersign_isExclusionOf1}
              | <->{Ont_numbersign_isAlternativeTo1}
              |<->{Ont_numbersign_isExclusionOf1, Ont_numbersign_isInConflictWith1}
              |-->{Ont_numbersign_isGeneralizationOf1}+
              |-->{Ont_numbersign_isRefinementOf1}
            +)
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Anfrage 12

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement4 ont:hasRelationship ?r.
      }
```

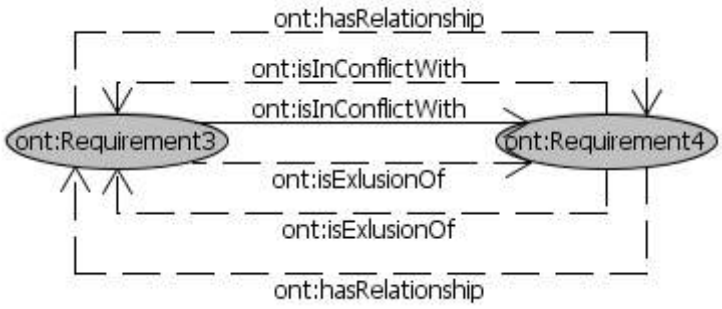


Abbildung 9.12: Requirements-Ontologie-Ausschnitt

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement4" : false}&
           -->{Ont_numbersign_hasRelationship1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Die Anfrage soll Requirements zurückliefern, mit denen `Requirement4` über die Vorkommen von `hasRelationship` oder über die Vorkommen ihrer Unterproperties verbunden ist.

In der Original-Ontologie ist `Requirement3` mit `Requirement4` über `isInConflictWith` verbunden (siehe Abbildung 9.12). `isInConflictWith` ist eine symmetrische Unterproperty von `hasRelationship`, deswegen werden nach dem Einsatz des Reasoners die Verbindungen in beiden Richtungen `ont:Requirement3 ont:hasRelationship ont:Requirement4` und `ont:Requirement4 ont:hasRelationship ont:Requirement3` in die Ontologie eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
            thisVertex.uriRef = "http://purl.org/ro/ont#Requirement4" : false}&
            (-->{Ont_numbersign_hasRelationship1} | <->{Ont_numbersign_isCoexistentWith1}
              |-->{Ont_numbersign_isSpecializationOf1}+ | -->{Ont_numbersign_isRevisionOf1}
              |<->{Ont_numbersign_isInConflictWith1, Ont_numbersign_isExclusionOf1}
              | <->{Ont_numbersign_isAlternativeTo1}
              |<->{Ont_numbersign_isExclusionOf1, Ont_numbersign_isInConflictWith1}
              |-->{Ont_numbersign_isGeneralizationOf1}+
              |-->{Ont_numbersign_isRefinementOf1}
            +)
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Anfrage 13

Abbildung 9.13: Requirements-Ontologie-Ausschnitt

```
select  ?r
where { ?r rdf:type ont:Requirement.
        ont:Requirement6 ont:hasRelationship ?r.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef = "http://purl.org/ro/ont#Requirement6" : false}&
           -->{Ont_numbersign_hasRelationship1}
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Das Ergebnis der Anfrage sind Requirements, mit denen `Requirement6` über `hasRelationship` oder über die Vorkommen ihrer Unterproperties verbunden ist. Die Anfrage bezieht sich auf den Ontologie-Ausschnitt in Abbildung 9.13. In dem Ontologie-Ausschnitt sind Requirements 6, 7 und 8 über `isRefinementOf` nacheinander verbunden.

Da `isRefinementOf` eine transitive Unterproperty von `hasRelationship` ist, werden nach dem Einsatz des Reasoners unter anderem die Verbindungen zwischen 6 und 8 über `isRefinementOf` und `hasRelationship` eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from r_var:V
with hasType(r_var, "Ont_numbersign_Requirement1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
            thisVertex.uriRef = "http://purl.org/ro/ont#Requirement6" : false}&
            (-->{Ont_numbersign_hasRelationship1} | <->{Ont_numbersign_isCoexistentWith1}
              |-->{Ont_numbersign_isSpecializationOf1}+ | -->{Ont_numbersign_isRevisionOf1}
              |<->{Ont_numbersign_isInConflictWith1, Ont_numbersign_isExclusionOf1}
              | <->{Ont_numbersign_isAlternativeTo1}
              |<->{Ont_numbersign_isExclusionOf1, Ont_numbersign_isInConflictWith1}
              |-->{Ont_numbersign_isGeneralizationOf1}+
              |-->{Ont_numbersign_isRefinementOf1}
            +)
         r_var)
report hasType(r_var, "Data") ? r_var.value : r_var.uriRef as "r_var"
end
```

Anfrage 14

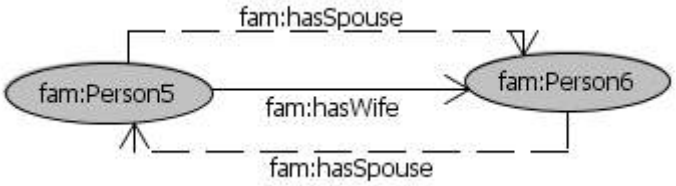


Abbildung 9.14: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person6 fam:hasSpouse ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
            thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person6" : false}&
            -->{Family_numbersign_hasSpouse1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Die Anfrage verwendet die symmetrische Superproperty `hasSpouse` und soll den Ehepartner von `Person6` zurückgeben. In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.14 ist `Person5` mit `Person6` über das Vorkommen der Unterproperty `hasWife` von `hasSpouse` verbunden.

Das Vorkommen der Superproperty `hasSpouse` wird erst nach dem Einsatz des Reasoners in die Ontologie eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person6" : false}&
           (<->{Family_numbersign_hasSpouse1}
             | (<->{Family_numbersign_hasWife1})
             | (<->{Family_numbersign_hasHusband1})
           )
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 15

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person5 fam:hasBloodRelationship ?p.
      }
```

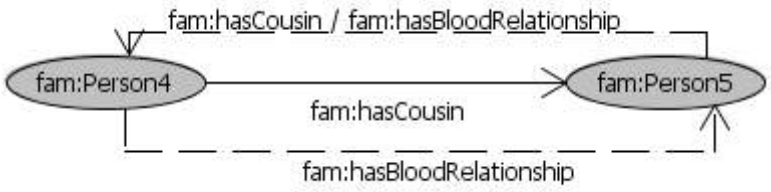


Abbildung 9.15: Family-Ontologie-Ausschnitt

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person5" : false}&
           -->{Family_numbersign_hasBloodRelationship1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

In der Anfrage wird die symmetrische Superproperty `hasBloodRelationship` verwendet. Die Anfrage soll Individuen zurückliefern, mit denen `Person5` über `hasBloodRelationship` oder über die Vorkommen ihrer Unterproperties verbunden ist.

In der Original-Ontologie aus Abbildung 9.15 ist `Person4` mit `Person5` über das Vorkommen der symmetrischen Property `hasCousin` verbunden. `hasCousin` ist eine Unterproperty der symmetrischen Property `hasBloodRelationship`. Die Verbindungen über `hasBloodRelationship` werden erst nach dem Einsatz des Reasoners in die Ontologie eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person5" : false}&
          (<->{Family_numbersign_hasBloodRelationship1}
            |(<->{Family_numbersign_hasChild1}
               |(<->{Family_numbersign_hasDaugther1})
               |(<->{Family_numbersign_hasSon1})
               |<--{Family_numbersign_hasParent1}
               |(<->{Family_numbersign_hasFather1, Family_numbersign_hasDad1})
               |(<->{Family_numbersign_hasMother1})
               |(<->{Family_numbersign_hasDad1, Family_numbersign_hasFather1})
              )
            |<->{Family_numbersign_hasCousin1}
              |(-->{Family_numbersign_hasAncestor1}
                 |(-->{Family_numbersign_hasFemaleAncestor1})
                 |(<--{Family_numbersign_hasFemaleAncestor1})
                 |(<->{Family_numbersign_hasParent1}
                     |(<->{Family_numbersign_hasFather1, Family_numbersign_hasDad1})
                     |(<->{Family_numbersign_hasMother1})
                     |(<->{Family_numbersign_hasDad1, Family_numbersign_hasFather1})
                     | <--{Family_numbersign_hasChild1}
                     |(<->{Family_numbersign_hasDaugther1})
```

```
                    |(<->{Family_numbersign_hasSon1})
                  )
                | (-->{Family_numbersign_hasMaleAncestor1})
                | (<--{Family_numbersign_hasMaleAncestor1})
              )+
             |(<--{Family_numbersign_hasAncestor1}
               |(-->{Family_numbersign_hasFemaleAncestor1})
               |(<--{Family_numbersign_hasFemaleAncestor1})
               |(<->{Family_numbersign_hasParent1}
                  |(<->{Family_numbersign_hasFather1, Family_numbersign_hasDad1})
                  |(<->{Family_numbersign_hasMother1})
                  |(<->{Family_numbersign_hasDad1, Family_numbersign_hasFather1})
                  |<--{Family_numbersign_hasChild1}
                  |(<->{Family_numbersign_hasDaugther1})
                  |(<->{Family_numbersign_hasSon1})
                )
               |(-->{Family_numbersign_hasMaleAncestor1})
               |(<--{Family_numbersign_hasMaleAncestor1})
              )+
           )
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 16

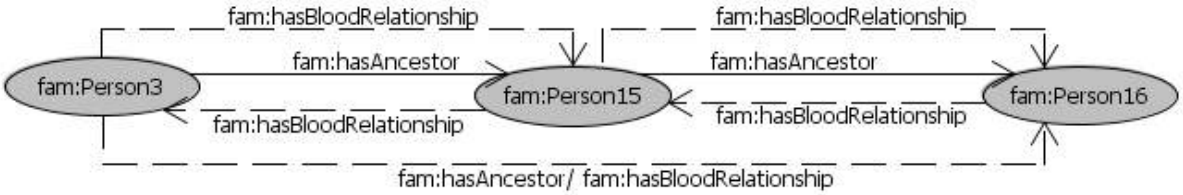


Abbildung 9.16: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person16 fam:hasBloodRelationship ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person16" : false}&
           -->{Family_numbersign_hasBloodRelationship1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Die Anfrage ist ähnlich der Anfrage 15, bezieht sich aber auf einen anderen Ontologie-Ausschnitt aus Abbildung 9.16. Die Anfrage soll alle Personen zurückliefern, mit denen `Person16` über die Vorkommen von `hasBloodRelationship` oder über die Vorkommen ihrer Unterproperties verbunden ist.

In dem relevanten Ontologie-Ausschnitt sind Personen 3, 15 und 16 über die Vorkommen der transitiven

Unterproperty `hasAncestor` von `hasBloodRelationship` nacheinander verbunden. Nach dem Einsatz des Reasoners werden Verbindungen über die Vorkommen von `hasBloodRelationship` eingefügt.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person16" : false}&
          (<->{Family_numbersign_hasBloodRelationship1}
            |(<->{Family_numbersign_hasChild1}
               |(<->{Family_numbersign_hasDaugther1})
               |(<->{Family_numbersign_hasSon1})
               |<--{Family_numbersign_hasParent1}
               |(<->{Family_numbersign_hasFather1, Family_numbersign_hasDad1})
               |(<->{Family_numbersign_hasMother1})
               |(<->{Family_numbersign_hasDad1, Family_numbersign_hasFather1})
             )
            |<->{Family_numbersign_hasCousin1}
              |(-->{Family_numbersign_hasAncestor1}
                 |(-->{Family_numbersign_hasFemaleAncestor1})
                 |(<--{Family_numbersign_hasFemaleAncestor1})
                 |(<->{Family_numbersign_hasParent1}
                     |(<->{Family_numbersign_hasFather1, Family_numbersign_hasDad1})
                     |(<->{Family_numbersign_hasMother1})
                     |(<->{Family_numbersign_hasDad1, Family_numbersign_hasFather1})
                     | <--{Family_numbersign_hasChild1}
                     |(<->{Family_numbersign_hasDaugther1})
                     |(<->{Family_numbersign_hasSon1})
                   )
                 | (-->{Family_numbersign_hasMaleAncestor1})
                 | (<--{Family_numbersign_hasMaleAncestor1})
               )+
              |(<--{Family_numbersign_hasAncestor1}
                |(-->{Family_numbersign_hasFemaleAncestor1})
                |(<--{Family_numbersign_hasFemaleAncestor1})
                |(<->{Family_numbersign_hasParent1}
                   |(<->{Family_numbersign_hasFather1, Family_numbersign_hasDad1})
                   |(<->{Family_numbersign_hasMother1})
                   |(<->{Family_numbersign_hasDad1, Family_numbersign_hasFather1})
                   |<--{Family_numbersign_hasChild1}
                   |(<->{Family_numbersign_hasDaugther1})
                   |(<->{Family_numbersign_hasSon1})
                 )
                |(-->{Family_numbersign_hasMaleAncestor1})
                |(<--{Family_numbersign_hasMaleAncestor1})
               )+
            )
          p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 17

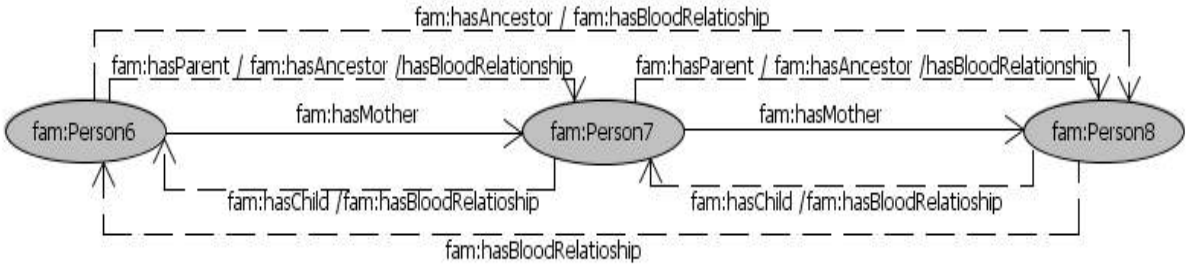


Abbildung 9.17: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person6 fam:hasAncestor ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person6" : false}&
           -->{Family_numbersign_hasAncestor1}
        p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

In der Anfrage wird die transitive Superproperty `hasAncestor` verwendet. Die Anfrage soll Personen zurückliefern, mit denen `Person6` über `hasAncestor` und über die Vorkommen ihrer Unterproperties verbunden ist.

In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.17 werden `Person6`, `Person7` und `Person8` über die Vorkommen der einfachen Unterproperty `hasMother` von `hasAncestor` nacheinander verbunden. Nach dem Einsatz des Reasoners werden die Personen unter anderem auch über `hasAncestor` nacheinander verbunden.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
         thisVertex.uriRef =
            "http://www.semanticweb.org/ontologies/2011/6/Family#Person6" : false}&
         (-->{Family_numbersign_hasAncestor1} |-->{Family_numbersign_hasFemaleAncestor1}
            |(-->{Family_numbersign_hasParent1}
               |-->{Family_numbersign_hasFather1, Family_numbersign_hasDad1}
               |-->{Family_numbersign_hasMother1}
               |-->{Family_numbersign_hasDad1, Family_numbersign_hasFather1}
               |<--{Family_numbersign_hasChild1}
               | -->{Family_numbersign_hasDaugther1}
               |-->{Family_numbersign_hasSon1}
             )
            | -->{Family_numbersign_hasMaleAncestor1}
```

```
          )+
        p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 18

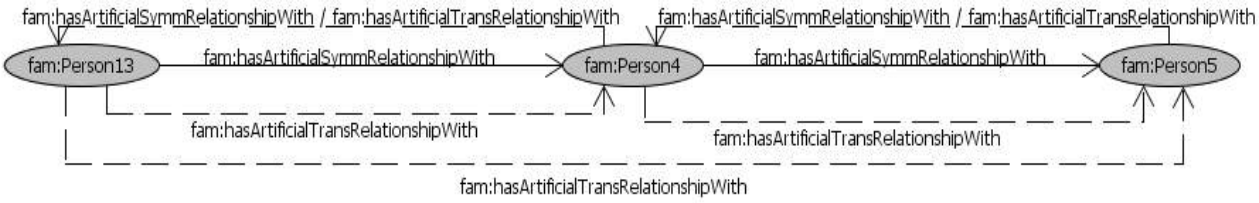


Abbildung 9.18: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person5 fam:hasArtificialTransitiveRelationshipWith ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person5" : false}&
           -->{Family_numbersign_hasArtificialTransitiveRelationshipWith1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Die Anfrage verwendet die transitive Superproperty `hasArtificialTransitiveRelationshipWith` und soll Personen zurückliefern, mit denen `Person5` über `hasArtificialTransitiveRelationship-With` oder die Vorkommen ihrer Unterproperties verbunden ist.

In dem relevanten Ontologie-Ausschnitt aus Abbildung 9.18 sind `Person13`, `Person4` und `Person5` über die Vorkommen der symmetrischen Unterproperty `hasArtificialSymmetricRelationship-With` von `hasArtificialTransitiveRelationshipWith` nacheinander verbunden.

Da `hasArtificialSymmetricRelationshipWith` eine symmetrische Unterproperty von `hasArtificialTransitiveRelationshipWith` ist und `hasArtificialTransitiveRelationshipWith` transitiv ist, werden nach dem Einsatz des Reasoners Personen 13, 4 und 5 über `hasArtificialTransitiveRelationshipWith` nacheinander in beiden Richtungen verbunden.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
              "http://www.semanticweb.org/ontologies/2011/6/Family#Person5" : false}&
           (-->{Family_numbersign_hasArtificialTransitiveRelationshipWith1}
             |<->{Family_numbersign_hasArtificialSymmetricRelationshipWith1}
           )+
```

```
        p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Anfrage 19

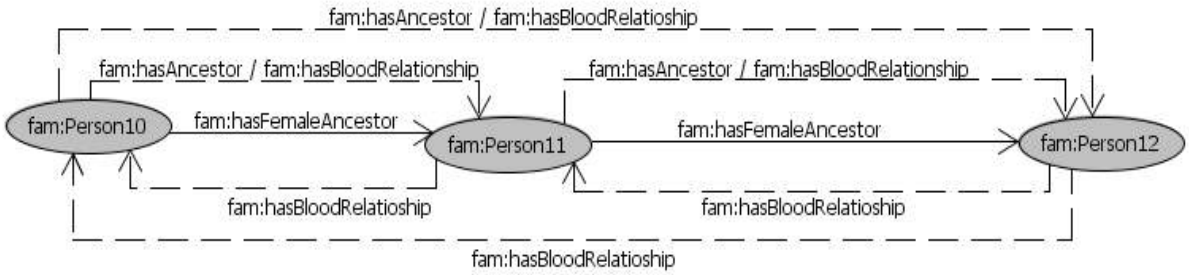


Abbildung 9.19: Family-Ontologie-Ausschnitt

```
select  ?p
where { ?p rdf:type fam:Person.
        fam:Person10 fam:hasAncestor ?p.
      }
```

```
//Schema-Aware-transformierte GReQL-Anfrage
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
           thisVertex.uriRef =
             "http://www.semanticweb.org/ontologies/2011/6/Family#Person10" : false}&
           -->{Family_numbersign_hasAncestor1}
         p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

Die Anfrage ist ähnlich der Anfrage 17, bezieht sich aber auf einen anderen Ontologie-Ausschnitt (siehe Abbildung 9.19). In der Anfrage wird die transitive Property `hasAncestor` verwendet. Die Anfrage soll Personen zurückliefern, mit denen `Person10` über die Vorkommen von `hasAncestor` oder über die Vorkommen ihrer Unterproperties verbunden ist.

In dem relevanten Ontologie-Ausschnitt sind `Person10`, `Person11` und `Person12` über die Vorkommen der transitiven Property `hasFemaleAncestor` nacheinander verbunden.

Da `hasFemaleAncestor` die transitive Unterproperty der transitiven Property `hasAncestor` ist, werden nach dem Einsatz des Reasoners die `Person10`, `Person11` und `Person12` über die Vorkommen von `hasAncestor` nacheinander verbunden.

Die Schema-Aware-transformierte GReQL-Anfrage wird wie folgt modifiziert:

```
//Schema-Aware-transformierte GReQL-Anfrage mit Modifikation
from p_var:V
with hasType(p_var, "Family_numbersign_Person1")
     and not isEmpty({@hasAttribute(thisVertex, "uriRef") ?
         thisVertex.uriRef =
           "http://www.semanticweb.org/ontologies/2011/6/Family#Person10" : false}&
         (-->{Family_numbersign_hasAncestor1} | -->{Family_numbersign_hasFemaleAncestor1}
           |(-->{Family_numbersign_hasParent1}
```

```
            |-->{Family_numbersign_hasFather1, Family_numbersign_hasDad1}
            |-->{Family_numbersign_hasMother1}
            |-->{Family_numbersign_hasDad1, Family_numbersign_hasFather1}
            |<--{Family_numbersign_hasChild1} | -->{Family_numbersign_hasDaugther1}
            | -->{Family_numbersign_hasSon1}
          )
         | -->{Family_numbersign_hasMaleAncestor1}
        )+
      p_var)
report hasType(p_var, "Data") ? p_var.value : p_var.uriRef as "p_var"
end
```

9.2 Herleitung der GReQL-Anfragen

In Rahmen dieser Studienarbeit soll die Effizienz des Transformationsverfahrens `greql2sparql` überprüft werden. Es soll die Transformationszeit von Anfragen unterschiedlicher Komplexität getestet werden.

Es wurden 5 Anfragen mit der unterschiedlichen Komplexität entwickelt. Die Komplexität der Anfragen steigt von einer einfachen Anfrage (Anfrage 1) bis zu einer komplexen Anfrage (Anfrage 5). Welche Sprachelemente die Anfragen verwenden, ist in Tabelle 9.2 beschrieben.

Anfrage	**from**-Teil	**with**-Teil	**report**-Teil
1	zwei Variablen	eine einfache Pfadbeschreibung und ein Attributzugriff	ein Attributzugriff
2	zwei Variablen	eine alternative Pfadbeschreibung und ein Attributzugriff	ein Attributzugriff
3	zwei Variablen	ein optionale Pfadbeschreibung mit einer Option und ein Attributzugriff	ein Attributzugriff
4	vier Variablen	zwei optionale Pfadbeschreibungen eine davon mit einer Option, eine einfache Pfadbeschreibung und zwei Attributzugriffe	zwei Attributzugriffe
5	sechs Variablen	zwei optionale Pfadbeschreibungen eine davon mit einer Option, eine einfache Pfadbeschreibungen, eine sequenzielle Pfadbeschreibung aus drei Kanten und zwei Attributzugriffe	drei Attributzugriffe

Tabelle 9.2: Beschreibung der GReQL-Anfragen

Es wurden die folgenden GReQL-Anfragen erstellt:

Anfrage 1

```
from r,c:V{Requirement}
with r -->{isInConflictWith} c
     and r.uriRef = "http://purl.org/ro/ont#Requirement0"
report c.uriRef
end
```

Anfrage 2

```
from r,c:V{Requirement}
with r -->{isInConflictWith} | <->{isExlusionOf} c
      and r.uriRef = "http://purl.org/ro/ont#Requirement0"
report c.uriRef
end
```

Anfrage 3

```
from r,c:V{Requirement}
with r -->{isInConflictWith} [-->{isAlternativeTo}] | <->{isExlusionOf} c
    and r.uriRef = "http://purl.org/ro/ont#Requirement0"
report c.uriRef
end
```

Anfrage 4

```
from r,c,u,i:V{Requirement}
with r -->{isInConflictWith} [-->{isAlternativeTo}] | <->{isExlusionOf} c
      and r -->{isDescribedByUseCase}|<--{describesRequirement} u
      and r --> {isRefinementOf} i
      and r.uriRef = "http://purl.org/ro/ont#Requirement0"
      and i.uriRef = "http://purl.org/ro/ont#Requirement5"
report c.uriRef, u.uriRef
end
```

Anfrage 5

```
from r,c,u,i,q:V{Requirement}
with r -->{isInConflictWith} [-->{isAlternativeTo}] | <->{isExlusionOf} c
        and r -->{isDescribedByUseCase} |<--{describesRequirement} u
        and r --> {isRefinementOf} i
        and c --> {hasRefinementSource} --> {isRefinementOf} --> {isInConflictWith} q
        and r.uriRef = "http://purl.org/ro/ont#Requirement0"
        and i.uriRef = "http://purl.org/ro/ont#Requirement5"
report c.uriRef, u.uriRef, q.uriRef
end
```

Die jeweiligen transformierten SPARQL-Anfragen befinden sich im Anhang A.

Kapitel 10

Zeit- und Speichermessungen

In Java kann mit der Methode `System.currentTimeMillis()` die aktuelle Zeit in Millisekunden abgefragt werden.

Die Methode wird in der Arbeit zur Ermittlung der Laufzeit von Anfragen verwendet. Mit der Methode wird die Zeit vor Beginn und nach dem Ende der jeweiligen Aktion abgefragt und die Differenz gebildet. Da die Zeitmessungen vom System abhängig sind, sollten sie relativ betrachtet werden. Um ein trotzdem möglichst gutes Ergebnis zu bekommen, werden mehrere Messungen durchgeführt und anschließend der Mittelwert berechnet.

Da man nicht weiß, wann der Garbage Collector seine Arbeit verrichtet, kann der Speicherbedarf zur Ausführung von Anfragen auf ähnliche Weise wie die Laufzeit nicht ermittelt werden. Deswegen wird zur Ermittlung des Speicherbedarfs der Java Profiler VisualVM eingesetzt, der ab Java SE 6 Update 7 fester Bestandteil des JDKs ist [Vis].

Da der Profiler selbst Rechenzeit benötigt und erzeugte Objekte zu Zeit-Messungen Speicher beanspruchen werden Zeitverbrauch- und Speicherverbrauch-Messungen getrennt durchgeführt.

10.1 Zeitmessungen

Für das Messen des Zeitbedarfs wurde die Klasse `Metric` entwickelt, die acht Array-Listen vom Typ `Double` und vier Methoden enthält (siehe Abbildung 10.1). Der Code zu den Klassen `Metric` und `NameOfResultList` befindet sich im Anhang B.

Drei Array-Listen sind für die Speicherung der Laufzeiten von drei Operationen zur Ausführung der SPARQL-Anfrage und fünf Array-Listen für die Speicherung der Laufzeiten von fünf Operationen zur Ausführung der GReQL-Anfragen vorgesehen. Auf die Operationen wird in den Abschnitten 10.1.1 und 10.1.2 genauer eingegangen. Die Klasse `Metric` besitzt die Methoden `add` und `get`, um die Messwerte in den jeweiligen Array-List zu setzen und abzufragen. Außer diesen Methoden verfügt die Klasse noch über die Methode `averageValue`, die den Mittelwert der jeweiligen Messungen aus Array-Listen ausgibt.

Methoden, deren Laufzeit gemessen werden soll, sind in den verschiedenen Klassen verteilt. In den Klassen, wie schon beschrieben, wird vor und nach den relevanten Methoden mit der Methode `System.currentTimeMillis()` die aktuelle Zeit abgefragt und die Differenz gebildet, die mit der Methode `add` in die entsprechende Array-Liste der Klasse `Metric` eingefügt wird.

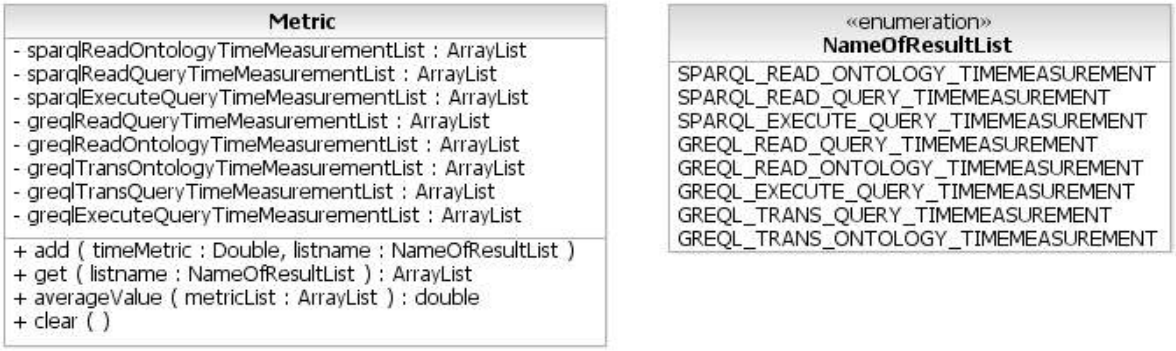


Abbildung 10.1: Klassen `Metric` und `NameOfResultList`

Um möglichst gute Ergebnisse zu bekommen, werden in einem Programmdurchlauf alle Anfragen jeweils 4 mal über ein Test-Verfahren aus Tabelle 1.1 ausgeführt. Nachdem alle Anfragen ausgeführt und die Zeitmessungen in den Array-Listen gespeichert wurden, werden die Mittelwerte der Messungen berechnet. Bei der Berechnung des Mittelwerts werden die ersten Elemente der Array-Listen nicht berücksichtigt, da diese Werte aufgrund von Initialisierung und anderen internen Vorgänge in der Java Virtual Machine stark von den anderen Werten abweichen. Anschließend wird noch die gesamte benötigte Zeit zur Ausführung jeder Anfrage berechnet. Die Mittelwerte der Zeitbedarfsmessungen und der gesamte Zeitbedarf zur Ausführung jeder Anfrage werden in eine Text-Datei geschrieben.

10.1.1 Zeitmessungen zur Ausführung von SPARQL-Anfragen

Die Ausführung[1] einer SPARQL-Anfrage kann in die drei folgenden Basis-Schritte aufgeteilt werden:

1. Einlesen einer OWL-Ontologie
2. Einlesen einer SPARQL-Anfrage
3. Ausführung der SPARQL-Anfrage

Dementsprechend wird der Zeitbedarf jeweils bei diesen drei Operationen gemessen und anschließend der gesamte Zeitbedarf berechnet. Im Folgenden wird detaillierter erläutert, wie der Zeitbedarf gemessen wird. Die oben genannten Operationen werden in den Methoden `readModel` und `executeSPARQLQuery` [Sch11] der Klasse `SPARQLQueryTest` ausgeführt. Die Methoden werden bei der Erläuterung der jeweiligen Operation in Listings dargestellt, dabei ist `timeMetric` eine Instanz der Klasse `Metric`.

1. Der Zeitbedarf zum Einlesen einer Ontologie beinhaltet den Zeitbedarf zur

   - Erzeugung eines Ontologie-Modells (OntModel-Objekt) (siehe Listing 10.1 Zeile 3) und zum
   - Einlesen einer Ontologie aus einer Datei in das Ontologie-Modell (siehe Listing 10.1 Zeile 4).

   Entsprechend wird die Zeit mit `System.currentTimeMillis()` vor (in Zeile 2 aus Listing 10.1) und nach (in Zeile 5 aus Listing 10.1) der Ausführung dieser Methoden gemessen und die Differenz gebildet. Die Differenz der beiden Zeiten wird in Zeile 6 und 7 aus Listing 10.1 anhand der Methode `add` in eine Array-Liste der Klasse `Metric` eingefügt.

[1] Zur Ausführung von SPARQL-Anfragen wird das Jene-Framework verwendet http://jena.sourceforge.net/

```
1 private Model readModel(String url, OntModelSpec spec) {
2   long startTime = System.currentTimeMillis();
3   OntModel m = ModelFactory.createOntologyModel(spec);
4   m.read(url);
5   long stopTime = System.currentTimeMillis();
6   timeMetric.add((stopTime - startTime) / 1000.0,
7     NameOfResultList.SPARQL_READ_ONTOLOGY_TIMEMEASUREMENT);
8   ...
```

Listing 10.1: Code-Ausschnitt aus der Methode `readModel` [Sch11] der Klasse `SPARQLQueryTest`

2. Der Zeitbedarf zum Einlesen einer SPARQL-Anfrage beinhaltet den Zeitbedarf zum

   - Einlesen einer SPARQL-Anfrage in das Anfrage-Objekt (siehe Listing 10.2 Zeile 3-4) und zur

   - Erzeugung einer SPARQL-Anfrage Ausführung (siehe Listing 10.2 Zeile 5).

```
1  private ResultSet executeSPARQLQuery(String query, Model m) {
2    long startTime = System.currentTimeMillis();
3    com.hp.hpl.jena.query.Query q =
4      com.hp.hpl.jena.query.QueryFactory.create(query);
5    QueryExecution qe = QueryExecutionFactory.create(q, m);
6    long stopTime = System.currentTimeMillis();
7    timeMetric.add((stopTime - startTime) / 1000.0,
8       NameOfResultList.SPARQL_READ_QUERY_TIMEMEASUREMENT);
9    startTime = System.currentTimeMillis();
10   ResultSet rs = qe.execSelect();
11   stopTime = System.currentTimeMillis();
12   timeMetric.add((stopTime - startTime) / 1000.0,
13     NameOfResultList.SPARQL_EXECUTE_QUERY_TIMEMEASUREMENT);
14   ...
```

Listing 10.2: Code-Ausschnitt aus der Methode `executeSPARQLQuery` [Sch11] der Klasse `SPARQLQueryTest`

3. Der Zeitbedarf zur Ausführung der SPARQL-Anfrage

   Hier wird die unmittelbare Zeit zur Ausführung einer Anfrage an die OWL-Ontologie ohne die Zeit zum Einlesen der Ontologie und der Anfrage gemessen. Die Ausführung einer SPARQL-Anfrage findet in Zeile 10 aus Listing 10.2 statt.

10.1.2 Zeitmessungen zur Ausführung von GReQL-Anfragen

Die Ausführung einer GReQL-Anfrage kann in die folgenden fünf Basis-Schritte (Operation) aufgeteilt werden:

1. Einlesen einer SPARQL-Anfrage
2. Einlesen einer OWL-Ontologie
3. Transformation der OWL-Ontologie in einen TGraphen
4. Transformation der SPARQL-Anfrage in eine GReQL-Anfrage
5. Ausführung der GReQL-Anfrage

Wie schon beschrieben werden die Transformationen über die zwei verschiedenen Verfahren durchgeführt. In den Schritten 1, 2, 4 und 5 gelten die Messungen sowohl für Schema-Aware Mapping als auch für Simple Mapping. Im Schritt 3 wird unterschieden, ob die Messungen für Schema-Aware Mapping oder für Simple Mapping durchgeführt werden.

Die oben genannten Operationen werden in den Methoden `create` der Klasse `QueryFactory`, `init` der Klasse `GraphLoader`, `run` der Klasse `OWLDLLoaderSchemaAware` (für Schema-Aware Mapping) oder der Klasse `OWLDLLoaderSimple` (für Simple Mapping) und `executeAsGReQL` der Klasse `Query` ausgeführt. Entsprechend finden in den Methoden die Laufzeit-Messungen statt. Im Folgenden werden die Ermittlungen der Zeitmessungen detaillierter erläutert.

1. Der Zeitbedarf zum Einlesen einer SPARQL-Anfrage beinhaltet den Zeitbedarf zur

   - Erzeugung des String-Reader (siehe Listing 10.3 Zeile 3) und zum

   - Einlesen einer SPARQL-Anfrage (siehe Listing 10.3 Zeile 4)

```
1 public static Query create(String sparql) throws QueryException {
2     long startTime = System.currentTimeMillis();
3     StringReader s = new StringReader(sparql);
4     SPARQLParser p = new SPARQLParser(s);
5     long stopTime = System.currentTimeMillis();
6     timeMetric.add((stopTime - startTime) / 1000.0,
7         NameOfResultList.GREQL_READ_QUERY_TIMEMEASUREMENT);
8   ...
```

Listing 10.3: Code-Ausschnitt aus der Methode `create` der Klasse `QueryFactory` [TPR10]

2. Der Zeitbedarf zum Einlesen einer OWL-Ontologie beinhaltet den Zeitbedarf zur

   - Erzeugung des Ontologie-Manager (siehe Listing 10.4 Zeile 4), zur

   - Erzeugung des ReaderDocumentSource-Objekts (siehe Listing 10.4 Zeile 5) und zum

   - Einlesen einer Ontologie aus einer Datei in den Ontologie-Manager (siehe Listing 10.4 Zeile 6)

```
1  public void init() throws LoaderInitException {
2    ...
3    long startTime = System.currentTimeMillis();
4    manager = OWLManager.createOWLOntologyManager();
5    OWLOntologyDocumentSource readerInput = new ReaderDocumentSource(in);
6    ont = manager.loadOntologyFromOntologyDocument(readerInput);
7    importsClosure = manager.getImportsClosure(ont);
8    long stopTime = System.currentTimeMillis();
9    timeMetric.add((stopTime - startTime) / 1000.0,
10     NameOfResultList.GREQL_READ_ONTOLOGY_TIMEMEASUREMENT);
11   ...
```

Listing 10.4: Code-Ausschnitt aus der Methode `init` der Klasse `GraphLoader` [TPR10]

3. Der Zeitbedarf zur Transformation einer OWL-Ontologie in einen TGraphen über Schema-Aware Mapping beinhaltet den Zeitbedarf zur

   - Transformation der TBox einer Ontologie in ein TGraph-Schema (siehe Listing 10.5 Zeile 3) und zur

- Transformation der ABox einer Ontologie in einen TGraphen (siehe Listing 10.5 Zeile 6)

```
1  public void run() {
2    long startTime = System.currentTimeMillis();
3    convertTBox2Schema();
4    schema.compile(CodeGeneratorConfiguration.MINIMAL);
5    graphClass = schema.getGraphClass();
6    convertABox2Graph();
7    long stopTime = System.currentTimeMillis();
8    timeMetric.add((stopTime - startTime) / 1000.0,
9      NameOfResultList.GREQL_TRANS_ONTOLOGY_TIMEMEASUREMENT);
10   ...
```

Listing 10.5: Code-Ausschnitt aus der Methode `run` der Klasse `OWLDLLoaderSchemaAware` [TPR10]

3. Der Zeitbedarf zur Transformation einer OWL-Ontologie in einen TGraphen über Simple Mapping beinhaltet den Zeitbedarf zur

   - Erzeugung eines TGraphen (siehe Listing 10.6 Zeile 3), zur

   - Transformation von Klassen und Individuen einer OWL-Ontologie in den TGraphen (siehe Listing 10.6 Zeile 5) und zur

   - Transformation von Properties der OWL-Ontologie in den TGraphen (siehe Listing 10.6 Zeile 6)

```
1  public void run() {
2    long startTime = System.currentTimeMillis();
3    createGraph();
4    propertiesTransformed = new BooleanGraphMarker(rdfGraph);
5    transformClassesAndIndividuals();
6    transformProperties();
7    long stopTime = System.currentTimeMillis();
8      timeMetric.add((stopTime - startTime),
9        NameOfResultList.GREQL_TRANS_ONTOLOGY_TIMEMEASUREMENT);
10   ...
```

Listing 10.6: Code-Ausschnitt aus der Methode `run` der Klasse `OWLDLLoaderSimple` [TPR10]

4. Transformation der SPARQL-Anfrage in eine GReQL-Anfrage (siehe Listing 10.7 Zeile 4)

```
1  public JValue executeAsGReQL(...)...{
2    ...
3    long transStartTime = System.currentTimeMillis();
4    greqlQuery = toGReQL(loader, mode, reasoner, queryModification);
5    long transStopTime = System.currentTimeMillis();
6    timeMetric.add((transStopTime - transStartTime) / 1000.0,
7      NameOfResultList.GREQL_TRANS_QUERY_TIMEMEASUREMENT);
8    long startTime = System.currentTimeMillis();
9    Graph graph = loader.getGraph();
10    if (eval == null) {
11       eval = new GreqlEvaluator(greqlQuery, graph, usingMap);
12    } else {
13      eval.setQuery(greqlQuery);
14      eval.setDatagraph(graph);
15      eval.setVariables(usingMap);
16      }
17     eval.setEvaluationLogger(null);
18     eval.startEvaluation();
```

```
19      JValue result = eval.getEvaluationResult();
20      long stopTime = System.currentTimeMillis();
21      timeMetric.add((stopTime - startTime) / 1000.0,
22        NameOfResultList.GREQL_EXECUTE_QUERY_TIMEMEASUREMENT);
23      ...
```

Listing 10.7: Code-Ausschnitt aus der Methode `executeAsGReQL` der Klasse `Query` [TPR10]

5. Der Zeitbedarf zur Ausführung der GReQL-Anfrage beinhaltet den Zeitbedarf zur

   - Initialisierung des GReQL-Evaluator (siehe Listing 10.7 Zeile 9-17) und zur
   - Evaluierung (Ausführung) der GReQL-Anfrage (siehe Listing 10.7 Zeile 18-19)

10.1.3 Zeitmessungen zur Transformation von GReQL-Anfragen in SPARQL-Anfragen

Der Zeitbedarf zur Transformation einer GReQL-Anfrage in eine SPARQL-Anfrage beinhaltet den Zeitbedarf zum

- Einlesen einer GReQL-Anfrage (siehe Listing 10.8 Zeile 3), zur
- Transformation der GReQL-Anfrage in DNF (siehe Listing 10.8 Zeile 5) und zur
- Transformation der GReQL-Anfrage in eine SPARQL-Anfrage (siehe Listing 10.8 Zeile 6).

```
1  public String transform(String greqlQuery, boolean check) {
2      long startTime = System.currentTimeMillis();
3      greqlGraph = GreqlParser.parse(greqlQuery);
4      eval.setDatagraph(greqlGraph);
5      toNormalForm();
6      transformToSPARQLGraph();
7      SparqlUnparser unparser = new SparqlUnparser(sparqlGraph,
8          queriedOntologyPrefix, queriedOntologyUri, prefixes);
9      long stopTime = System.currentTimeMillis();
10     timeMetric.add((stopTime-startTime),
11         NameOfResultList.GREQL_IN_SPARQL_TRANS_TIMEMEASUREMENT);
12     return unparser.unparse();
13   }
```

Listing 10.8: Code-Ausschnitt aus der Methode `transform` der Klasse `Greql2Sparql` [Sch11]

10.2 Speichermessungen

Zum besseren Verständnis der Speicherverbrauch-Messungen wird in diesem Abschnitt erst auf die Speicherverwaltung der JVM eingegangen und der VisualVM Profiler kurz vorgestellt.

10.2.1 Speicherverwaltung der JVM

Der Speicher der JVM ist in die zwei Hauptbereiche Heap und Permanent Generation (PG) aufgeteilt (siehe Abbildung 10.2). Der Heap besteht aus den zwei Bereichen Young Generation (YG) und Old Generation (OG) [Mas06]. Im Bereich Young Generation werden kurzlebige Objekte und im Bereich Old Generation

langlebige Objekte aufbewahrt. Im Bereich Permanent Generation werden unter anderem Meta-Daten von Java-Klassen gespeichert [Mas06].

Sowohl zur Heap-Größe als auch zur PG-Größe gibt es Standardeinstellungen wie die minimale und maximale Speichergröße, die vom Benutzer falls nötig geändert werden können. In Permanent Generation geht jedoch in seltenen Fällen der Speicher aus. Dies könnte geschehen, wenn viele Klassen geladen werden müssen [GCT].

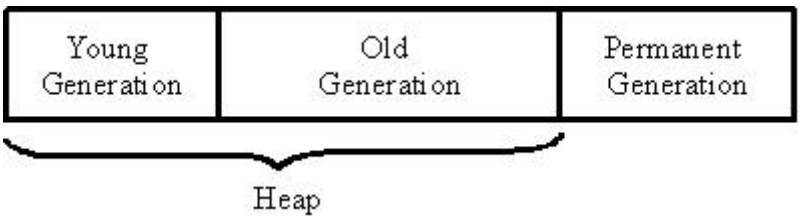


Abbildung 10.2: Seciher der JVM

10.2.2 VisualVM

Der VisualVM Profiler kann bei der Entwicklung von Java-Anwendungen eingesetzt werden, um Speicherlecks zu entdecken, Anwendungen zu überwachen und die Performance zu steigern.

Mit Hilfe von VisualVM können Monitoring einer Anwendung, Profiling von Anwendungen und Heap-Dumping durchgeführt werden.

Monitoring ist ein Prozess, der in gleichmäßigen Abständen das Verhalten während der Ausführung von Anwendungen protokolliert und auswertet [CS06]. Mit Monitoring kann zum Beispiel der Speicherbedarf oder die CPU-Auslastung bei der Ausführung einer Anwendung überprüft werden. Beim Monitoring wird jedoch keine detaillierte Information wie zum Beispiel welche Methode am meisten die CPU belastet geliefert. Das fällt in den Aufgabenbereich des Profiling.

Das Ziel des Profiling ist eine Anwendung zu untersuchen und deren Schwachstellen zu finden. Mit Profiling kann zum Beispiel die Laufzeit von konkreten Methoden oder der Speicherbedarf von Klasseninstanzen ermittelt werden.

Mit Monitoring werden also Probleme von Anwendungen entdeckt und mit Profiling werden die Ursachen dieser Probleme festgestellt.

Ein Dump bezeichnet das Anzeigen eines zusammenhängenden Teils des Speichers [CS06]. Ein Dumping ist ein Prozess, der das realisiert.

Im Folgenden werden die Features des Profiler [Doc] kurz vorgestellt.

Monitoring einer Anwendung

Monitoring einer Anwendung kann für die CPU, Memory, Klassen und Threads durchgeführt werden. Hier wird die allgemeine Information über die genannten Objekte live ermittelt und angezeigt. Die ermittelten Daten werden graphisch in Diagrammen dargestellt. Im Folgenden werden die vier Monitoring-Arten kurz vorgestellt.

- CPU-Monitoring

  Hier kann über ein Diagramm die CPU-Auslastung und die Aktivität des Garbage Collector prozentual betrachtet werden (siehe Abbildung 10.3).

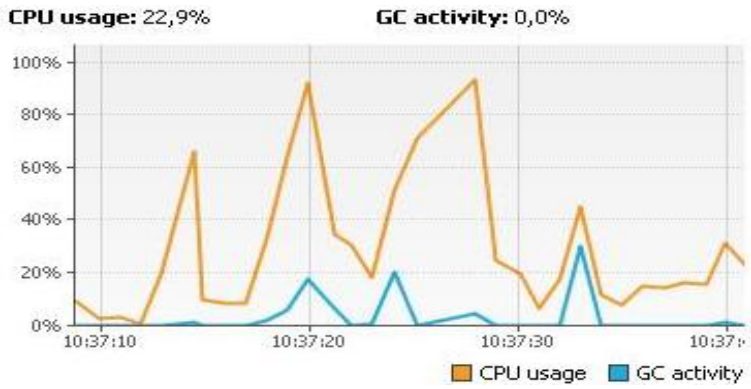


Abbildung 10.3: CPU-Monitoring

- Memory-Monitoring

  Da diese Funktionalität des Tools in der Arbeit verwendet wird, wird sie hier detaillierter erläutert. Unter der Memory-Ansicht lassen sich die zwei weiteren Heap- und Permanent Generation(PG)-Ansichten öffnen. In der Heap-Ansicht werden die gesamte Heap-Größe, die benutzte Heap-Größe und die maximal mögliche Heap-Größe in der Abbildung 10.4 angezeigt. In der PG-Ansicht werden die gesamte PG-Größe, die benutzte PG-Größe und die maximale mögliche PG-Größe in der Abbildung 10.5 dargestellt. Dementsprechend kann hier die maximale Speicherauslastung - Heap-Speicher und Permanent Generation - bei einem Programmdurchlauf ermittelt werden.

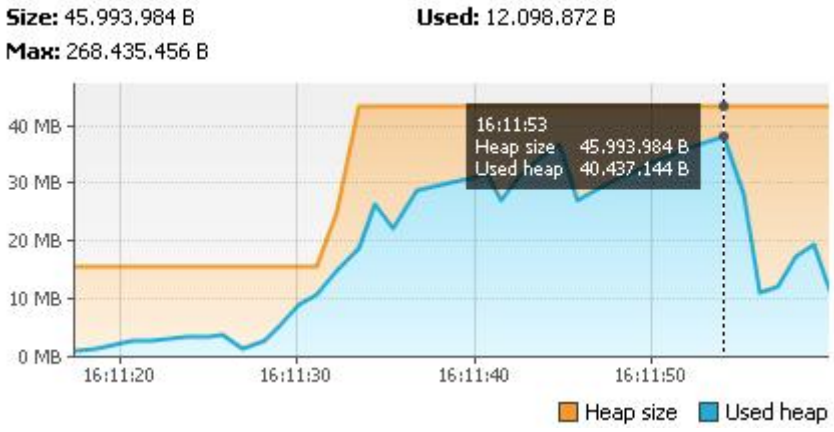


Abbildung 10.4: Heap-Monitoring

- Klassen-Monitoring

  In der Ansicht kann das Diagramm zur Anzahl aller aktuell geladenen Klassen beobachtet werden. Der Verlauf der Kurve, die die Anzahl von Klassen bezüglich der Zeit darstellt ist in Abbildung 10.6 abgebildet.

- Threads-Monitoring

  Hier kann die so genannte Thread-Timeline betrachtet werden. Die Thread-Timeline zeigt den zeitlichen Verlauf der Zustände (laufend, schlafend, wartend) aller Threads farbig an (siehe Abbildung 10.7). Zusätzlich steht noch eine Tabelle zur Verfügung, die die Zeitdauer über den jeweiligen Zustand und die gesamte Laufzeit zu jedem Thread enthält.

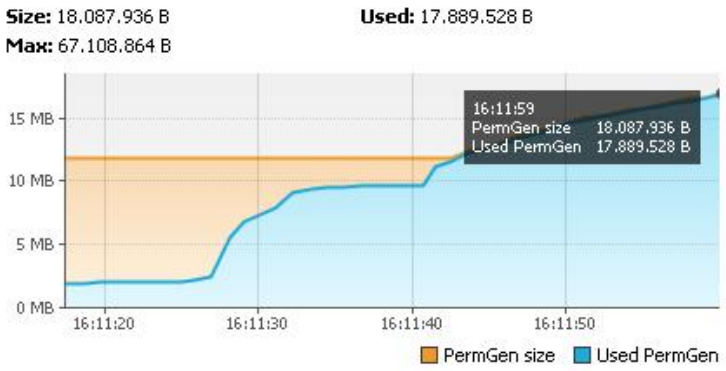


Abbildung 10.5: Permanent Generation-Monitoring

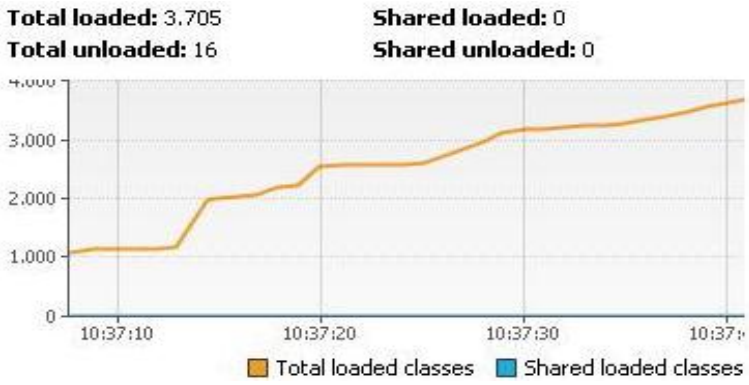


Abbildung 10.6: Klassen-Monitoring

Profiling von Anwendungen

VisualVM bietet auch die detaillierten Untersuchungen der CPU-Auslastung und des Speichers während der Ausführung einer oder mehrerer Java-Anwendungen. Aktuell ermittelte Profiling-Daten werden während der Ausführung von Anwendungen live angezeigt. Zu jedem Zeitpunkt können die momentanen Aufnahmen (Snapshots) von ermittelten Daten erstellt werden, die zum Beispiel später analysiert oder verglichen werden können.

- CPU-Profiling

  Beim CPU-Profiling wird der Zeitverbrauch von Methoden ermittelt (siehe Abbildung 10.8). Zu jeder untersuchten Methode werden die gesamte Ausführungszeit von allen Aufrufen und die Anzahl

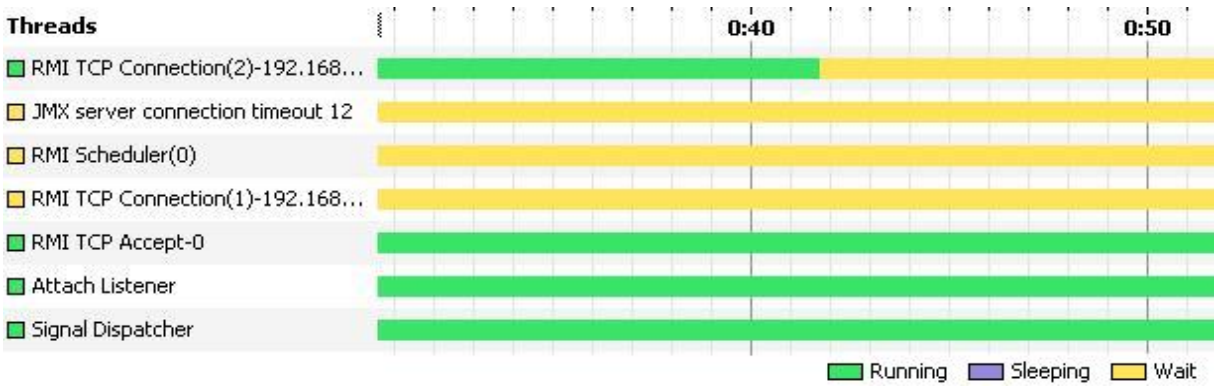
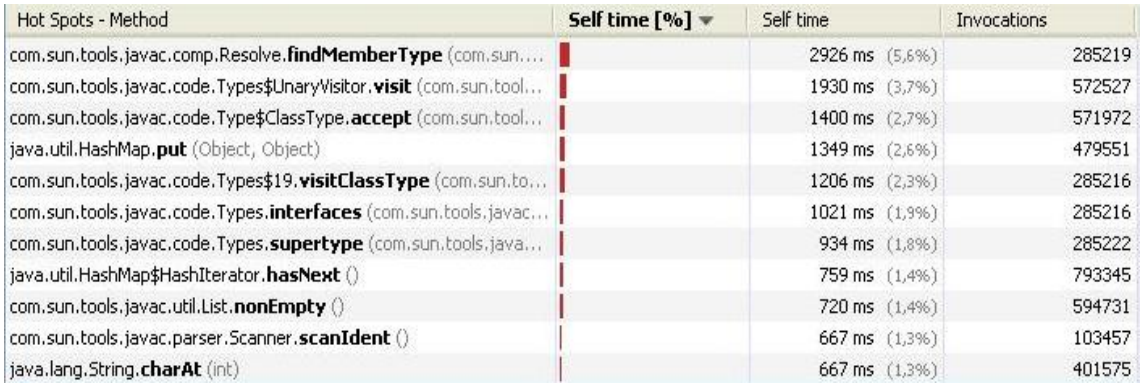


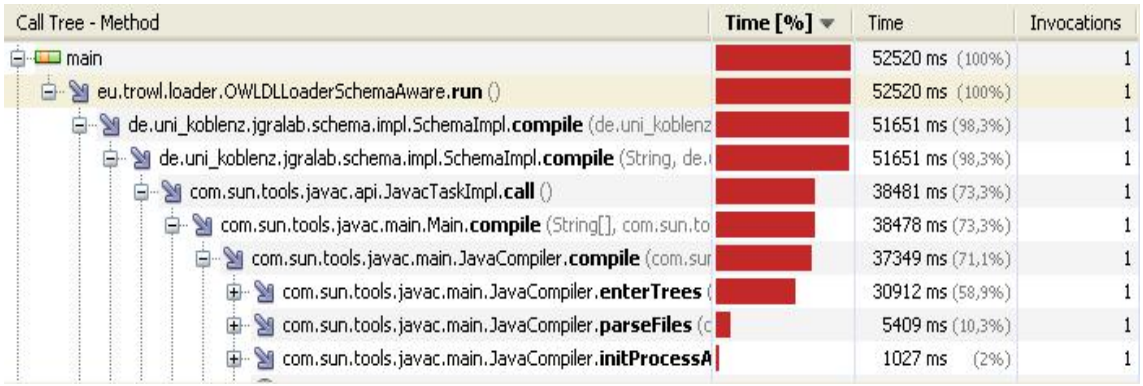
Abbildung 10.7: Thread-Timeline

von Aufrufen angezeigt. Es ist auch möglich den Aufrufbaum von Methoden mit dem jeweiligen Zeitverbrauch anzeigen zu lassen (siehe Abbildung 10.9).



Hot Spots - Method	Self time [%]	Self time	Invocations
com.sun.tools.javac.comp.Resolve.**findMemberType** (com.sun....		2926 ms (5,6%)	285219
com.sun.tools.javac.code.Types$UnaryVisitor.**visit** (com.sun.tool...		1930 ms (3,7%)	572527
com.sun.tools.javac.code.Type$ClassType.**accept** (com.sun.tool...		1400 ms (2,7%)	571972
java.util.HashMap.**put** (Object, Object)		1349 ms (2,6%)	479551
com.sun.tools.javac.code.Types$19.**visitClassType** (com.sun.to...		1206 ms (2,3%)	285216
com.sun.tools.javac.code.Types.**interfaces** (com.sun.tools.javac...		1021 ms (1,9%)	285216
com.sun.tools.javac.code.Types.**supertype** (com.sun.tools.java...		934 ms (1,8%)	285222
java.util.HashMap$HashIterator.**hasNext** ()		759 ms (1,4%)	793345
com.sun.tools.javac.util.List.**nonEmpty** ()		720 ms (1,4%)	594731
com.sun.tools.javac.parser.Scanner.**scanIdent** ()		667 ms (1,3%)	103457
java.lang.String.**charAt** (int)		667 ms (1,3%)	401575

Abbildung 10.8: CPU-Profiling einer Anwendung: Snapshot-Ausscchnitt über den Zeitverbrauch der Methoden

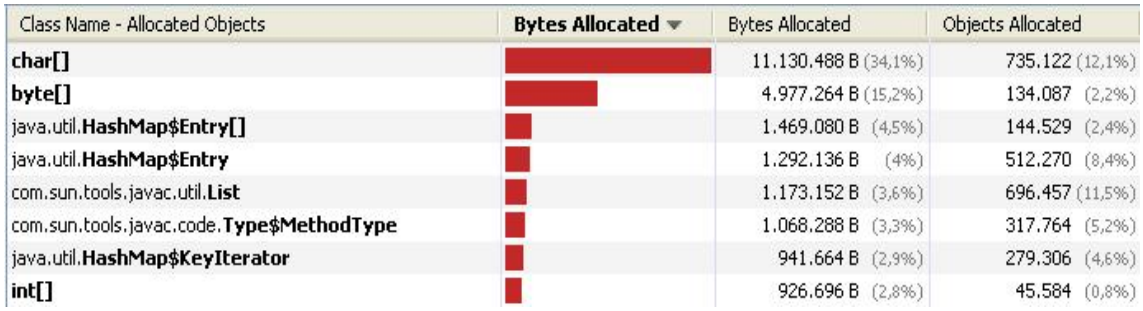


Call Tree - Method	Time [%]	Time	Invocations
main		52520 ms (100%)	1
eu.trowl.loader.OWLDLLoaderSchemaAware.**run** ()		52520 ms (100%)	1
de.uni_koblenz.jgralab.schema.impl.SchemaImpl.**compile** (de.uni_koblenz		51651 ms (98,3%)	1
de.uni_koblenz.jgralab.schema.impl.SchemaImpl.**compile** (String, de.(		51651 ms (98,3%)	1
com.sun.tools.javac.api.JavacTaskImpl.**call** ()		38481 ms (73,3%)	1
com.sun.tools.javac.main.Main.**compile** (String[], com.sun.to		38478 ms (73,3%)	1
com.sun.tools.javac.main.JavaCompiler.**compile** (com.sur		37349 ms (71,1%)	1
com.sun.tools.javac.main.JavaCompiler.**enterTrees** (		30912 ms (58,9%)	1
com.sun.tools.javac.main.JavaCompiler.**parseFiles** (c		5409 ms (10,3%)	1
com.sun.tools.javac.main.JavaCompiler.**initProcessA**		1027 ms (2%)	1

Abbildung 10.9: CPU-Profiling einer Anwendung: Snaphot-Ausschnitt vom Aufrufbaum der Methoden

- Memory-Profiling

  Hier können Klassennamen, deren allokierte Objekte und der Speicherverbrauch (in Bytes) von Klassen-Objekten, wie in der Abbildung 10.10 dargestellt, ermittelt werden. In der Stack-Trace Ansicht aus der Abbildung 10.11 können Methoden betrachtet werden, die Objekte einer bestimmten Klasse erzeugt haben.



Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		11.130.488 B (34,1%)	735.122 (12,1%)
byte[]		4.977.264 B (15,2%)	134.087 (2,2%)
java.util.**HashMap$Entry[]**		1.469.080 B (4,5%)	144.529 (2,4%)
java.util.**HashMap$Entry**		1.292.136 B (4%)	512.270 (8,4%)
com.sun.tools.javac.util.**List**		1.173.152 B (3,6%)	696.457 (11,5%)
com.sun.tools.javac.code.**Type$MethodType**		1.068.288 B (3,3%)	317.764 (5,2%)
java.util.**HashMap$KeyIterator**		941.664 B (2,9%)	279.306 (4,6%)
int[]		926.696 B (2,8%)	45.584 (0,8%)

Abbildung 10.10: Memory-Profiling einer Anwendung: Snaphot-Ausschnitt der allokierten Objekte

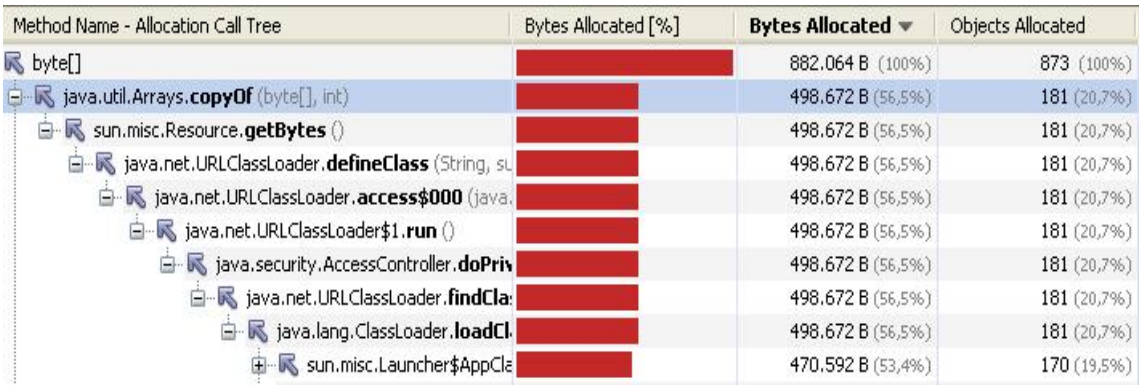


Abbildung 10.11: Memory-Profiling einer Anwendung: Snaphot-Ausschnitt des Methoden-Aufrufbaums

Heap-Dumping

Mit VisualVM ist es möglich, ein Heap-Dump zu erstellen. Hier kann der aktuelle Inhalt des Heap einer Anwendung ausgelesen werden. Es gibt eine Tabelle über die Anzahl der vorhandenen Instanzen jeder Klasse und deren Speicherbedarf. Zusätzlich können einzeln die Instanzen und ihre Felder (Instanzvariablen) angesehen werden (siehe Abbildung 10.12).

Außer oben genannten Funktionen, kann VisualVM mit Plugins um weitere Funktionen erweitert werden.

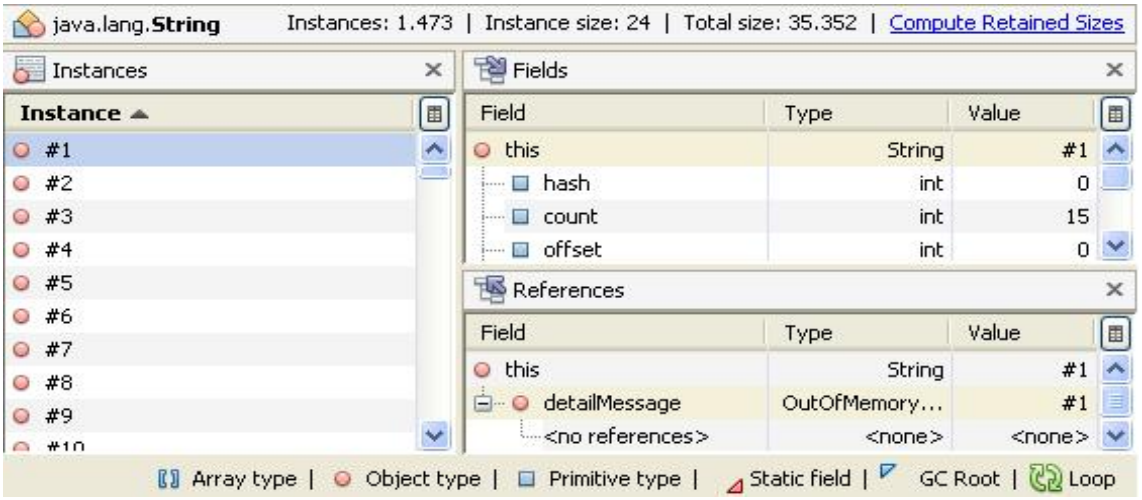


Abbildung 10.12: Heap-Dumping: Ausschnitt aus der Instanzen-Ansicht

10.2.3 Speichermessungen zur Ausführung von SPARQL- und GReQL-Anfragen

Die geeignete Funktion von VisualVM zu den Speicherverbrauch-Messungen der Ausführung von Anfragen ist Heap-Monitoring. Damit kann die maximale Speicherauslastung - Heap-Speicher und Permanent Generation - bei einem Programmdurchlauf ermittelt werden. Um die Speicherauslastung bei der Ausführung jeder einzelnen Anfrage ermitteln zu können, wird im Gegensatz zu Zeitmessungen in einem Programmdurchlauf nur eine Anfrage einmal ausgeführt. Die dabei ermittelten Werte sollten auch als relative Ergebnisse betrachtet werden. Die Werte gelten zu Vergleichszwecken in diesem Rahmen.

Wie schon erwähnt werden die Speicherbedarf-Messungen getrennt von Zeitbedarf-Messungen durchgeführt. Da erzeugte Objekte zur Messung des Zeitbedarfs auch den Speicher beanspruchen, werden die Ob-

jekte und Methoden-Aufrufe zu Zeitmessungen während der Speicherverbrauch-Ermittlungen im Code auskommentiert.

Kapitel 11

Evaluierung

Die Testfälle wurden in der folgenden Testumgebung durchgeführt:

- Pentium Centrino 1.70 GHz
- 1,25 GB RAM
- 80 GB Festplatte
- Windows XP Home Service Pack 2
- Java SDK 1.6._27

Anfragen wurden auf den Ontologien mit 100, 200, 500 und 1000 Duplikaten der Basis-Ontologien (Requirements-Ontologie und Family-Ontologie) ausgeführt. Wie schon erwähnt enthält eine Basis-Ontologie (ABox) der Requirement-Ontologie 10 Individuen und eine Basis-Ontologie der Family-Ontologie 17 Individuen. Dementsprechend besitzt die Requirement-Ontologie mit 100 Duplikaten 1000 Individuen und die Family-Ontologie 1700 Individuen.

Die gesamte Ausführungszeit einer Anfrage wurde maximal auf 10 Minuten beschränkt.

11.1 Zeit-Messergebnisse zur Ausführung von Anfragen

Die ermittelten Zeitmessungen zur Ausführung der Anfragen an Ontologien mit 100 und 200 Duplikaten sind in Tabelle 11.1 und an Ontologien mit 500 und 1000 Duplikaten in Tabelle 11.2 aufgelistet.

Bevor die Tabelle 11.1 betrachtet werden kann sollten hier die aus Platzgründen vorgenommenen Abkürzungen erläutert werden.

In der Spalte `Messungen` bedeuten die Abkürzungen folgendes:

- Einlesen SA ≙ Einlesen der SPARQL-Anfrage
- Einlesen Ont. ≙ Einlesen der OWL-Ontologie
- Trans. Ont. in TG ≙ Transformation der OWL-Ontologie in den TGraphen
- Trans. SA in GA ≙ Transformation der SPARQL-Anfrage in die GReQL-Anfrage

- Ausf. GA / SA ≙ Ausführung einer GReQL-Anfrage oder einer SPARQL-Anfrage (Das hängt vom verwendeten Verfahren ab. Wird eine SPARQL-Anfrage unmittelbar auf der OWL-Ontologie ausgeführt, ist in dem Fall die Ausführung einer SPARQL-Anfrage gemeint, sonst die Ausführung einer GReQL-Anfrage).

Die Zeile `Datensatz` (die obere Zeile) repräsentiert durch die Angabe der Anzahl der Duplikate der Basis-Ontologien die Größen der getesteten Ontologien.

In der Zeile `Verfahren` sind getestete Verfahren die in Tabelle 1.1 genauer beschrieben sind in abgekürzter Form dargestellt. Die Abkürzungen in der Zeile bedeuten folgendes:

- SPARQL ≙ unmittelbare Ausführung einer SPARQL-Anfrage

  (entspricht dem Testfall 1 in Tabelle 1.1)

- S-A M ≙ Schema-Aware Mapping mit Modifikation

  (entspricht dem Testfall 2 in Tabelle 1.1)

- S-A R ≙ Schema-Aware Mapping mit Reasoner

  (entspricht dem Testfall 3 in Tabelle 1.1)

- S M ≙ Simple Mapping mit Modifikation

  (entspricht dem Testfall 4 in Tabelle 1.1)

- S R ≙ Simple Mapping mit Reasoner

  (entspricht dem Testfall 5 in Tabelle 1.1)

Wie schon in Kapitel 10 beschrieben, entfallen beim Testfall 1 unmittelbare Ausführung einer SPARQL-Anfrage die Messungen „Transformation der OWL-Ontologie in den TGraphen“ und „Transformation der SPARQL-Anfrage in die GREQL-Anfrage“. Entsprechend bleiben die jeweiligen Zellen in Tabelle 11.1 leer. Bei den Anfragen, die länger als 10 Minuten zur Ausführung benötigen, werden die entsprechenden Zellen in Tabelle mit `Timeout` markiert.

Für die Tabelle 11.2 gelten die gleichen Abkürzungen wie für die Tabelle 11.1. Zur besseren Übersicht fehlen aber in Tabelle 11.2 die Spalten `S-A R` und `S R`, deren Ergebnisse bei jeder Anfrage mit `Timeout` markiert müssten.

Anfrage	Messungen \ Datensatz & Verfahren	100 Duplikate					200 Duplikate				
		SPARQL	S-A M	S-A R	S M	S R	SPARQL	S-A M	S-A R	S M	S R
1	Einlesen SA	12.0	0.0	0.0	0.0	0.0	12.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	234.4	127.3	130.2	159.2	130.0	402.6	239.5	340.6	290.4	180.333
	Trans. Ont. in TG.		3415.8	5976.4	38.1	3154.333		3836.3	26512.4	94.0	11954.0
	Trans. SA in GA		0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0
	Ausf. GA/SA	0.0	16.1	304.4	37.1	270.667	2.0	44.1	1011.2	44.2	1068.333
1	Insgesamt	246.4	3559.2	6411.0	234.4	3555.0	416.6	4119.9	27864.2	428.6	13202.667
2	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0
	Einlesen Ont.	214.4	113.0	106.0	129.2	143.333	406.6	209.3	340.2	192.0	160.333
	Trans. Ont. in TG.		3318.0	5952.6	34.1	3151.333		3582.0	25761.0	96.0	11887.0
	Trans. SA in GA		0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0
	Ausf. GA/SA	0.0	22.0	146.4	322.4	560.667	4.0	54.2	1103.4	1134.0	2223.333
2	Insgesamt	214.4	3453.0	6205.0	485.7	3855.333	410.6	3848.5	27204.6	1422.0	14270.667

3	Einlesen SA	2.0	0.0	0.0	1.0	0.0	0.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	226.4	101.2	134.4	141.2	131.2	410.6	206.2	timeout	194.0	timeout
	Trans. Ont. in TG.		3479.0	6198.8	33.0	3298.6		3465.0	timeout	120.2	timeout
	Trans. SA in GA		0.0	0.0	2.0	0.0		0.0	timeout	0.0	timeout
	Ausf. GA/SA	0.0	924.2	216080.4	2174.0	156963.8	0.0	3734.4	timeout	8670.8	timeout
3	Insgesamt	228.4	4504.4	222413.6	2351.2	160393.6	410.6	7405.6	timeout	8985.0	timeout
4	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	208.4	113.3	202.4	129.2	137.0	410.6	216.2	344.4	276.4	234.333
	Trans. Ont. in TG.		3319.7	5894.2	53.1	3191.0		3485.2	25632.6	74.2	11950.333
	Trans. SA in GA		0.0	0.0	1.0	0.0		0.0	0.0	4.0	0.0
	Ausf. GA/SA	0.0	11.0	136.2	29.0	267.333	0.0	26.0	993.6	52.0	991.333
4	Insgesamt	208.4	3444.0	6232.8	212.3	3595.333	410.6	3727.4	26970.6	406.6	13176.0
5	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	S0.0	0.0	0.0	0.0
	Einlesen Ont.	212.2	112.1	100.2	126.2	97.0	392.6	244.1	318.4	300.4	180.333
	Trans. Ont. in TG.		3510.2	5750.6	44.1	3234.667		3536.4	26782.6	70.2	11940.667
	Trans. SA in GA		0.0	0.0	2.0	0.0		1.0	6.0	0.0	0.0
	Ausf. GA/SA	0.0	26.0	500.2	45.0	467.0	0.0	38.1	3330.8	64.2	1806.0
5	Insgesamt	212.2	3648.3	6351.0	217.3	3798.667	392.6	3819.6	30437.8	434.8	13927.0
6	Einlesen SA	2.0	0.0	0.0	1.0	0.0	2.0	0.0	0.0	2.0	0.0
	Einlesen Ont.	334.6	193.4	280.4	204.1d	160.0	703.0	413.6	565.0	433.0	270.0
	Trans. Ont. in TG.		1055.3	3999.8	87.2	3622.333		1212.9	24232.8	144.0	13152.333
	Trans. SA in GA		0.0	0.0	2.0	0.0		0.0	0.0	2.0	0.0
	Ausf. GA/SA	4.0	46.2	130.2	65.2	230.0	2.0	S102.0	945.0	132.0	911.667
6	Insgesamt	340.6	1294.9	4410.4	359.5	4012.333	707.0	1728.5	25742.8	713.0	14334.0
7	Einlesen SA	2.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	354.6	203.0	156.2	222.3	150.0	719.0	391.5	579.0	422.6	300.333
	Trans. Ont. in TG.		1055.7	4075.8	87.2	3585.333		1185.7	23633.8	150.0	13049.333
	Trans. SA in GA		0.0	0.0	2.0	0.0		0.0	0.0	0.0	0.0
	Ausf. GA/SA	0.0	39.2	138.2	501.7	707.667	0.0	78.2	1001.6	1862.8	2723.667
7	Insgesamt	356.6	1297.9	4370.2	813.2	4443.0	721.0	1655.4	25214.4	2435.4	16073.333
8	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	362.6	158.1	167.0	179.2	156.667	721.0	404.6	timeout	428.8	timeout
	Trans. Ont. in TG.		1117.8	4423.0	99.2	3692.333		1232.7	timeout	148.0	timeout
	Trans. SA in GA		0.0	0.0	3.0	0.0		0.0	timeout	6.0	timeout
	Ausf. GA/SA	0.0	846.2	109768.0	3435.9	101272.333	2.0	2971.4	timeout	13116.8	timeout
8	Insgesamt	362.6	2122.1	114358.0	3717.3	105121.333	723.0	4608.7	timeout	13699.6	timeout
9	Einlesen SA	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	356.6	213.1	154.0	238.3	156.667	727.2	376.6	557.0	408.4	310.333
	Trans. Ont. in TG.		1126.6	3964.0	67.1	3575.333		1303.8	23828.2	152.4	13345.667
	Trans. SA in GA		0.0	2.0	5.0	0.0		0.0	0.0	10.0	0.0
	Ausf. GA/SA	0.0	30.1	128.2	125.2	256.667	0.0	96.2	937.2	252.4	878.333
9	Insgesamt	356.6	1371.8	4248.2	435.6	3988.667	727.2	1776.6	25322.4	823.2	14534.333
10	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.333
	Einlesen Ont.	336.6	163.4	160.0	277.4	146.667	707.0	381.4	593.0	420.0	267.0
	Trans. Ont. in TG.		1034.4	4024.2	72.1	3572.0		1221.8	23640.0	171.0	13349.667
	Trans. SA in GA		0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0
	Ausf. GA/SA	0.0	24.0	126.0	37.0	237.333	2.0	46.2	939.4	100.0	881.333
10	Insgesamt	336.6	1221.8	4310.2	386.5	3956.0	709.0	1649.4	25172.4	691.0	14501.333
11	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	230.4	108.1	96.2	129.1	87.0	420.6	209.2	timeout	220.0	timeout
	Trans. Ont. in TG.		3496.0	5728.4	43.0	3218.0		3444.1	timeout	70.4	timeout
	Trans. SA in GA		0.0	2.0	0.0	0.0		1.0	timeout	2.0	timeout
	Ausf. GA/SA	0.0	1004.5	222746.2	1338.0	155710.333	0.0	4053.7	timeout	4955.0	timeout
11	Insgesamt	230.4	4608.6	228572.8	1511.1	159015.333	420.6	7708.0	timeout	5247.4	timeout
12	Einlesen SA	2.0	0.0	0.0	0.0	0.0	0.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	208.2	93.2	134.0	128.2	93.333	420.6	195.3	timeout	230.6	timeout
	Trans. Ont. in TG.		3278.4	6170.6	34.1	3194.333		3531.1	timeout	70.0	timeout
	Trans. SA in GA		1.0	0.0	1.0	0.0		0.0	timeout	8.0	timeout
	Ausf. GA/SA	0.0	959.6	221482.6	1312.8	153908.333	0.0	4136.9	timeout	4925.0	timeout
12	Insgesamt	210.2	4332.2	227787.2	1476.1	157196.0	420.6	7863.3	timeout	5233.6	timeout
13	Einlesen SA	0.0	0.0	0.0	0.0	0.0	2.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	228.2	129.1	134.4	127.2	90.0	394.4	205.3	timeout	238.0	timeout
	Trans. Ont. in TG.		3844.7	6036.6	34.0	3348.333		3891.5	timeout	74.2	timeout

	Trans. SA in GA		0.0	0.0	1.0	0.0		0.0	timeout	4.0	timeout
	Ausf. GA/SA	0.0	1131.6	219722.0	2332.5	152729.667	2.0	3911.6	timeout	8945.0	timeout
13	Insgesamt	228.2	5105.4	225893.0	2494.7	156168.0	398.4	8008.4	timeout	9261.2	timeout
14	Einlesen SA	6.0	0.0	0.0	0.0	0.0	0.0	0.0	3.333	0.0	0.0
	Einlesen Ont.	364.6	160.4	208.4	211.2	147.0	703.0	401.0	297.667	396.4	267.0
	Trans. Ont. in TG.		1070.4	4033.8	80.2	3578.667		1445.333	12447.667	160.4	13242.333
	Trans. SA in GA		0.0	0.0	2.0	0.0		0.0	0.0	2.2	0.0
	Ausf. GA/SA	0.0	56.1	204.0	116.2	480.333	0.0	90.0	1442.0	312.2	1846.333
14	Insgesamt	370.6	1936.333	4446.2	409.6	4206.0	703.0	1796.6	14190.667	871.2	15355.667
15	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	timeout	2.0	timeout
	Einlesen Ont.	346.6	162.2	146.2	164.2	153.667	721.0	332.3	timeout	336.6	timeout
	Trans. Ont. in TG.		1068.7	4036.0	66.1	3688.333		1244.0	timeout	184.0	timeout
	Trans. SA in GA		5.0	2.0	18.0	0.0		2.0	timeout	20.0	timeout
	Ausf. GA/SA	2.0	2076.8	210663.0	16938.4	193341.333	2.0	7818.2	timeout	67240.8	timeout
15	Insgesamt	348.6	3312.7	214847.2	17186.7	197183.333	723.0	9396.5	timeout	67783.4	timeout
16	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	334.6	142.3	150.0	164.6	147.0	701.0	315.5	timeout	318.6	timeout
	Trans. Ont. in TG.		1018.6	4030.0	67.0	3628.667		1052.5	timeout	148.2	timeout
	Trans. SA in GA		1.0	0.0	12.0	0.0		0.0	timeout	20.0	timeout
	Ausf. GA/SA	0.0	1991.7	211173.8	17153.4	203799.667	2.0	7215.4	timeout	65938.8	timeout
16	Insgesamt	334.6	3153.6	215353.8	17397.0	207575.333	703.0	8583.4	timeout	66425.6	timeout
17	Einlesen SA	2.0	0.0	0.0	0.0	0.0	2.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	350.6	200.5	146.6	173.3	147.0	699.0	351.7	timeout	366.2	timeout
	Trans. Ont. in TG.		1373.7	4031.6	77.0	3618.333		1085.4	timeout	154.8	timeout
	Trans. SA in GA		1.0	0.0	6.0	0.0		0.0	timeout	6.0	timeout
	Ausf. GA/SA	2.0	932.4	104858.8	2090.0	98892.333	0.0	2840.0	timeout	7847.0	timeout
17	Insgesamt	354.6	2507.6	109037.0	2346.3	102657.667	701.0	4277.1	timeout	8374.0	timeout
18	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0
	Einlesen Ont.	354.6	200.3	150.2	217.2	144.0	725.0	425.7	303.333	352.8	280.333
	Trans. Ont. in TG.		1027.5	4009.8	66.2	3668.333		1088.6	12414.333	144.0	13312.333
	Trans. SA in GA		1.0	0.0	4.0	0.0		0.0	0.0	4.0	0.0
	Ausf. GA/SA	2.0	42.0	1320.0	86.1	944.333	2.0	116.0	3224.667	204.4	3635.0
18	Insgesamt	356.6	1270.8	5480.0	373.5	4756.667	727.0	1633.3	15942.333	705.2	17227.667
19	Einlesen SA	2.0	0.0	0.0	0.0	0.0	0.0	0.0	timeout	0.0	timeout
	Einlesen Ont.	356.6	147.2	162.4	201.4	157.333	705.0	338.2	timeout	368.2	timeout
	Trans. Ont. in TG.		1067.7	4268.2	68.1	3698.333		1132.9	timeout	158.8	timeout
	Trans. SA in GA		0.0	0.0	6.0	0.0		1.0	timeout	10.0	timeout
	Ausf. GA/SA	0.0	765.0	105513.6	2108.0	99366.0	0.0	2871.1	timeout	7869.0	timeout
19	Insgesamt	358.6	1979.9	109944.2	2383.5	103221.667	705.0	4343.2	timeout	8406.0	timeout

Tabelle 11.1: Zeitmessungen (in ms) zur Ausführung von Anfragen an Ontologien mit 100 und 200 Duplikaten

Anfrage	Messungen \ Datensatz & Verfahren	500 Duplikate			1000 Duplikate		
		SPARQL	S-A M	S M	SPARQL	S-A M	S M
1	Einlesen SA	13.333	0.0	0.0	14.0	0.0	0.0
	Einlesen Ont.	964.667	563.7	619.0	1980.8	1162.0	1104.667
	Trans. Ont. in TG.		3513.9	242.4		3675.0	541.0
	Trans. SA in GA		0.0	0.0		0.0	6.667
	Ausf. GA/SA	0.0	68.3	94.0	0.0	157.0	247.0
1	Insgesamt	978.0	4145.9	955.4	1994.8	4994.0	1899.333
2	Einlesen SA	3.333	0.0	0.0	2.0	0.0	0.0
	Einlesen Ont.	928.0	674.9	591.0	1936.8	1098.667	1141.667
	Trans. Ont. in TG.		3112.5	268.2		3561.667	490.667
	Trans. SA in GA		0.0	6.2		0.0	0.0
	Ausf. GA/SA	0.0	164.2	6489.2	2.0	400.333	26907.333
2	Insgesamt	931.333	3951.6	7354.6	1940.8	5060.667	27729.667
3	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	934.667	587.0	596.6	1950.8	1105.333	1084.333
	Trans. Ont. in TG.		3163.5	280.6		3885.333	424.333
	Trans. SA in GA		0.0	0.0		0.0	6.667
	Ausf. GA/SA	3.333	21736.2	54322.0	0.0	85379.333	220954.333

3	Insgesamt	938.0	25486.7	55199.2	1950.8	90370.0	222469.667
4	Einlesen SA	0.0	0.0	0.0	2.0	0.0	0.0
	Einlesen Ont.	1014.667	665.8	602.8	1970.8	1074.667	1165.0
	Trans. Ont. in TG.		3085.3	240.4		3425.0	467.667
	Trans. SA in GA		0.0	0.0		0.0	6.667
	Ausf. GA/SA	0.0	82.5	120.2	0.0	253.667	250.0
4	Insgesamt	1014.667	3833.6	963.4	1972.8	4753.333	1889.333
5	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1038.333	543.6	614.8	2032.6	1095.333	1121.667
	Trans. Ont. in TG.		3218.7	236.2		3371.333	491.0
	Trans. SA in GA		0.0	4.0		0.0	3.333
	Ausf. GA/SA	3.333	97.2	150.4	2.2	300.333	420.333
5	Insgesamt	1041.667	3859.5	1005.4	2034.8	4767.0	2036.333
6	Einlesen SA	0.0	0.0	0.0	2.0	0.0	0.0
	Einlesen Ont.	1679.333	891.2	963.2	3545.0	2280.0	2123.0
	Trans. Ont. in TG.		1348.0	481.0		1465.667	971.333
	Trans. SA in GA		0.0	6.0		0.0	16.667
	Ausf. GA/SA	0.0	160.3	268.2	0.0	256.667	500.667
6	Insgesamt	1679.333	2399.5	1718.4	3547.0	4002.333	3611.667
7	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1812.667	960.4	941.0	3475.0	2089.667	1923.0
	Trans. Ont. in TG.		1222.6	449.0		1442.333	844.333
	Trans. SA in GA		1.0	8.0		0.0	13.333
	Ausf. GA/SA	0.0	215.4	11298.2	2.0	477.0	45795.667
7	Insgesamt	1812.667	2399.4	12696.2	3477.0	4009.0	48576.333
8	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1666.333	894.2	961.4	3487.0	2089.667	2173.333
	Trans. Ont. in TG.		1178.7	480.8		1933.0	907.667
	Trans. SA in GA	0.0	18.0			0.0	36.667
	Ausf. GA/SA	0.0	16438.8	81581.4	2.0	63738.0	325882.0
8	Insgesamt	1666.333	18511.7	83041.6	3489.0	67760.667	328999.667
9	Einlesen SA	0.0	2.0	2.0	0.0	0.0	0.0
	Einlesen Ont.	1713.0	924.3	945.4	3533.2	1833.0	2159.333
	Trans. Ont. in TG.		1209.8	434.6		1689.0	1011.667
	Trans. SA in GA		0.0	12.0		0.0	26.667
	Ausf. GA/SA	0.0	155.2	584.8	0.0	343.667	1018.333
9	Insgesamt	1713.0	2291.3	1978.8	3533.2	3865.667	4216.0
10	Einlesen SA	0.0	0.0	0.0	2.0	0.0	0.0
	Einlesen Ont.	1705.667	937.5	913.2	3487.0	2019.667	1982.667
	Trans. Ont. in TG.		1244.7	516.8		1412.0	982.0
	Trans. SA in GA		0.0	6.0		0.0	6.667
	Ausf. GA/SA	0.0	114.0	190.2	0.0	253.667	396.667
10	Insgesamt	1705.667	2296.2	1626.2	3489.0	3685.333	3368.0
11	Einlesen SA	3.333	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1095.0	480.6	510.6	1942.8	885.333	918.0
	Trans. Ont. in TG.		3265.8	196.6		3611.333	564.0
	Trans. SA in GA		0.0	10.0		0.0	20.0
	Ausf. GA/SA	0.0	23109.2	31317.0	4.0	91428.0	123654.667
11	Insgesamt	1098.333	26855.6	32034.2	1946.8	95924.667	125156.667
12	Einlesen SA	0.0	1.1	0.0	0.0	0.0	0.0
	Einlesen Ont.	948.333	496.9	519.0	1904.8	915.0	951.333
	Trans. Ont. in TG.		3281.7	230.0		3314.333	507.333
	Trans. SA in GA		0.0	10.0		0.0	23.333
	Ausf. GA/SA	0.0	23203.1	31215.0	0.0	89235.0	127059.333
12	Insgesamt	948.333	26982.8	31974.0	1904.8	93464.333	128541.333
13	Einlesen SA	0.0	0.0	0.0	2.0	0.0	0.0
	Einlesen Ont.	914.667	472.8	512.6	1920.6	1037.667	1091.667
	Trans. Ont. in TG.		3319.9	228.4		3662.333	504.0
	Trans. SA in GA		0.0	14.0		0.0	20.0
	Ausf. GA/SA	0.0	23291.3	57308.6	0.0	90924.0	231846.667
13	Insgesamt	914.667	27084.0	58063.6	1922.6	95624.0	233462.333
14	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1726.0	1164.667	1168.333	3481.0	2123.0	2039.333

	Trans. Ont. in TG.		1575.667	400.667		1448.667	1165.0
	Trans. SA in GA		0.0	6.667		0.0	16.667
	Ausf. GA/SA	3.333	130.667	450.667	2.0	330.667	721.333
14	Insgesamt	1729.333	2871.0	2026.333	3483.0	3902.333	3942.333
15	Einlesen SA	0.0	0.0	0.0	0.0	0.0	timeout
	Einlesen Ont.	1839.333	1081.5	998.0	3543.2	1896.0	timeout
	Trans. Ont. in TG.		1218.7	520.667		1769.0	timeout
	Trans. SA in GA		3.0	70.0		0.0	timeout
	Ausf. GA/SA	0.0	40925.8	419950.667	2.0	157483.333	timeout
15	Insgesamt	1839.333	43229.0	421539.333	3545.2	161148.333	timeout
16	Einlesen SA	0.0	0.0	0.0	2.0	0.0	timeout
	Einlesen Ont.	1682.0	1061.6	964.667	3481.0	2039.667	timeout
	Trans. Ont. in TG.		1160.6	540.667		1522.00	timeout
	Trans. SA in GA		1.0	43.333		0.0	timeout
	Ausf. GA/SA	0.0	40898.8	421967.0	2.0	155780.667	timeout
16	Insgesamt	1682.0	43122.0	423515.667	3485.0	159342.333	timeout
17	Einlesen SA	3.333	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1669.0	933.2	918.0	3479.0	2049.667	1956.0
	Trans. Ont. in TG.		1184.9	460.667		1375.333	831.333
	Trans. SA in GA		0.0	16.667		3.333	40.333
	Ausf. GA/SA	0.0	16321.5	48419.667	2.0	64095.333	188193.667
17	Insgesamt	1672.333	18439.6	49815.0	3481.0	67523.667	191021.333
18	Einlesen SA	0.0	0.0	0.0	0.0	0.0	0.0
	Einlesen Ont.	1686.0	977.3	991.333	3571.2	2016.333	1979.333
	Trans. Ont. in TG.		1155.6	520.667		1388.667	1071.667
	Trans. SA in GA		0.0	3.333		0.0	10.0
	Ausf. GA/SA	0.0	242.5	394.333	0.0	433.667	911.333
18	Insgesamt	1686.0	2375.4	1909.667	3571.2	3838.667	3972.333
19	Einlesen SA	0.0	0.0	0.0	2.0	0.0	0.0
	Einlesen Ont.	1685.333	1046.2	984.333	3479.0	1972.667	1886.333
	Trans. Ont. in TG.		1156.9	531.0		1408.667	1135.0
	Trans. SA in GA		0.0	20.0		0.0	40.0
	Ausf. GA/SA	0.0	16472.6	46814	2.0	63431.333	189312.0
19	Insgesamt	1685.333	18675.7	48349.333	3483.0	66812.667	192373.333

Tabelle 11.2: Zeitmessungen (in ms) zur Ausführung von Anfragen an Ontologien mit 500 und 1000 Duplikaten

Nach allen Evaluierungen wurde festgestellt, dass die Ausführung der SPARQL-Anfragen generell effizienter ist, als alle anderen getesteten Verfahren. Bei den Anfragen 1, 4, 5, 6, 10 und 18 konnte jedoch beobachtet werden, dass sie bei Simple Mapping mit Modifikation fast die gleiche oder sogar weniger Zeit zur Ausführung benötigen. Diese Anfragen können als einfache Anfragen charakterisiert werden, da sie entweder Properties, die keine Unterproperties besitzen oder Superproperties mit der einstufigen Hierarchie verwenden.

Wenn man die Tabellen 11.1 und 11.2 betrachtet, wird es ersichtlich, dass die Verfahren Schema-Aware Mapping mit Reasoner und Simple Mapping mit Reasoner am zeitaufwendigsten sind. Bei den Verfahren benötigen die Anfragen 3, 8, 11, 12, 13, 15, 16, 17 und 19 zur Ausführung schon in den Ontologien mit 200 Duplikaten mehr als 10 Minuten. Die Anfragen brauchen generell fast bei allen Verfahren relativ viel Zeit zur Ausführung, da sie im Vergleich zu anderen Anfragen etwas komplexer sind. Sie verwenden Properties, die tiefe Hierarchien besitzen oder liefern relativ große Ergebnismengen zurück. Alle Anfragen, die auf den Ontologien mit 500 und 1000 Duplikaten über Schema-Aware Mapping mit Reasoner oder Simple Mapping mit Reasoner ausgeführt wurden, benötigen mehr als 10 Minuten zur Ausführung.

Wie vermutet, sind die Verfahren Schema Aware Mapping und Simple Mapping mit Modifikation, also ohne Einsatz eines Reasoners, effizienter als mit dem Einsatz eines Reasoners.

Bei den Ergebnissen der Ontologien mit 100 Duplikaten fällt auf, dass Schema-Aware Mapping im Vergleich zu Simple Mapping relativ viel Zeit zur Transformation der Ontologien benötigt. Wenn man jedoch

die weiteren größeren Ontologien in Betracht zieht, kann man beobachten dass, die Transformationszeit der Ontologien über Schema-Aware Mapping nur gering aufsteigt. Die Transformationszeit der Ontologien über Simple-Mapping wächst dagegen fast proportional zur Ontologie-Größe. Es liegt daran, dass die TBox der Ontologien über Schema-Aware Mapping aufwendiger transformiert wird als über Simple Mapping. Wie schon in Kapitel 8 beschrieben, besitzt Simple Mapping ein vordefiniertes TGraph-Schema, das für alle OWL-Ontologien anwendbar ist. Bei jeder Transformation von OWL-Ontologien in TGraphen über Schema-Aware Mapping muss dagegen ein TGraph-Schema erst erzeugt werden, was einige Zeit in Anspruch nehmen kann. Dafür werden aber die Individuen und deren Beziehungen bei Schema-Aware Mapping effizienter transformiert als bei Simple Mapping. Die unmittelbare Ausführung aller GReQL-Anfragen über Schema-Aware Mapping mit Modifikation ist dagegen in jeder Ontologie-Größe effizienter als über Simple-Mapping mit Modifikation.

Als Beispiel wird eine der komplexen Anfragen die Anfrage 19 etwas detaillierter betrachtet. Die Transformation der Ontologien in den TGraphen über Schema-Aware Mapping mit Modifikation hat bei der Anfrage 19 folgende Ergebnisse

- 1067,7 ms (bei 100 Duplikaten),
- 1132,9 ms (bei 200 Duplikaten),
- 1156,9 ms (bei 500 Duplikaten) und
- 1408,6 ms (bei 1000 Duplikaten).

Die Transformation der Ontologien in den TGraphen über Simple Mapping mit Modifikation hat bei der Anfrage 19 folgende Ergebnisse

- 68,1 ms (bei 100 Duplikaten),
- 158,8 ms (bei 200 Duplikaten),
- 531,0 ms (bei 500 Duplikaten) und
- 1135 ms (bei 1000 Duplikaten).

Wenn die Differenz zwischen den zwei Transformationsarten bei den Ontologien mit 100 Duplikaten bei 998,9 ms liegt, beträgt sie bei 1000 Duplikaten nur 273,6 ms.

Die unmittelbare Ausführung der Anfrage 19 über Schema-Aware Mapping mit Modifikation hat folgende Ergebnisse

- 765,0ms (bei 100 Duplikaten),
- 2871,1 ms (bei 200 Duplikaten),
- 16472,6 ms (bei 500 Duplikaten) und
- 63431,3 ms (bei 1000 Duplikaten).

Die unmittelbare Ausführung der Anfrage 19 über Simple Mapping mit Modifikation hat folgende Ergebnisse

- 2108,0 ms (bei 100 Duplikaten),

- 7869,0 ms (bei 200 Duplikaten),
- 46814,0 ms (bei 500 Duplikaten) und
- 189312,0 ms (bei 1000 Duplikaten).

Wie man sieht, dauert die unmittelbare Ausführung der Anfrage 19 über Simple Mapping mit Modifikation wesentlich länger als über Schema-Aware Mapping mit Modifikation. Ein ähnliches Bild kann bei den anderen Anfragen beobachtet werden.

Der Großteil der Anfragen in Tabelle 11.1 wird effizienter über Simple Mapping transformiert als über Schema-Aware Mapping. Bei 100 Duplikaten wurden nur die Anfragen 8, 15, 16 und 19 effizienter über Schema-Aware Mapping mit Modifikation als über Simple Mapping mit Modifikation ausgeführt. Bei 200 Duplikaten sind es die Anfragen 3, 7, 8, 13, 15, 16, 17 und 19, deren Ausführung über Schema-Aware mit Modifikation effizienter ist. Es sind also die schon oben erwähnten Anfragen, die als komplexe Anfragen identifiziert wurden. In der Tabelle 11.2 (500 und 1000 Duplikate) wurden die meisten Anfragen besonderes bei 1000 Duplikaten über Schema-Aware Mapping mit Modifikation effizienter ausgeführt als über Simple Mapping mit Modifikation. Dementsprechend kann folgendes festgestellt werden: bei den Ontologien mit großen ABoxen (mit großen Anzahl an Individuen) und bei komplexen Anfrage ist Schema-Aware Mapping im Vergleich zu Simple Mapping effizienter. Darauf deutet auch der aufgetretene Timeout in den Ontologien mit 1000 Duplikaten bei den über Simple Mapping mit Modifikation transformierten Anfragen 15 und 16.

11.2 Speicher-Messergebnisse zur Ausführung von Anfragen

Die ermittelten Speichermessungen sind in Tabellen 11.3 und 11.4 aufgelistet.

Für die Tabellen gelten außer für Spalte `Messungen` die gleichen Abkürzungen, wie für die Tabellen zu den Zeitmessungen.

In der Spalte `Messungen` bedeuten die Abkürzungen folgendes:

- Heap ≙ Maximal verbrauchter Heap-Speicher
- PermGen ≙ Maximal verbrauchter Speicher im Bereich Permanent Generation

Anfrage	Messungen	100 Duplikate					200 Duplikate				
		SPARQL	S-A M	S-A R	S M	S R	SPARQL	S-A M	S-A R	S M	S R
1	Heap	5673208	19349856	44062848	8050480	62083192	9169992	20840960	92562040	11108840	201376000
	PermGen	14137936	19716736	21678144	15652136	17625080	14138520	19611782	21803568	15603864	17765016
2	Heap	5485312	19552616	45118584	9798544	51219312	9103832	20010056	93581808	11244632	197570464
	PermGen	14137800	19720024	21678320	15652216	17625480	14138488	195663136	21802416	15626080	17760200
3	Heap	5839528	24228280	45865336	6391272	65971248	9175536	24814176	timeout	9596360	timeout
	PermGen	14138168	19725112	21725368	15660480	17657144	14138688	19625784	timeout	15728640	timeout
4	Heap	5631456	21735008	43815984	9544992	65956456	9222664	22792272	91118664	10621528	201897952
	PermGen	14137872	19724096	21683880	15658000	17585288	14138736	19619504	21795296	15631512	17758696
5	Heap	5567064	19798064	39872016	9270464	49064720	9156832	20186912	104439456	10545944	239311608
	PermGen	14134784	19720568	21669240	15652872	17617112	14138424	19613184	21795256	15625824	17749040
6	Heap	5839528	21175392	40438272	10986192	74126232	10262680	2129346	87786488	11305448	222289072
	PermGen	14138168	18770440	20834184	15564520	17658864	14123848	18817200	21132888	15536248	17968184

7	Heap	6875144	21289104	41061336	11279000	61337520	10049304	21898488	88275360	12730792	181144992
	PermGen	14123816	18774584	20834616	15557080	17618360	14123536	18857560	21134920	15537792	17953720
8	Heap	6552728	19892664	41940936	13992432	77739152	9992608	20929688	timeout	14241400	timeout
	PermGen	14123560	18828304	20866696	15582056	17688080	14123864	18889192	timeout	15552008	timeout
9	Heap	6725424	16300736	41387440	9823856	74910528	10103424	17663160	87831392	13455760	224970624
	PermGen	14123648	18770536	20847032	15539040	17654920	14123832	18869296	21135017	15540600	17972736
10	Heap	6230840	19177816	41716864	9728864	75641136	10519648	22982336	89689656	10252936	226795968
	PermGen	14123408	18766920	20846032	15559360	17658448	14124232	18861872	21132808	15532256	17963664
11	Heap	5923208	24016744	48669664	6176888	64769328	9201864	31013920	timeout	11014368	timeout
	PermGen	14138272	19733744	21708480	15672496	17647496	14138720	19920200	timeout	15646192	timeout
12	Heap	5431088	23460008	47714312	5699008	65210568	9326368	23746584	timeout	10911000	timeout
	PermGen	14137856	19733912	21712456	15672176	17663856	14138688	19626688	timeout	15646088	timeout
13	Heap	5550704	24266416	47857984	7378256	57513544	9258680	24874440	timeout	12400390	timeout
	PermGen	14137872	19763328	21722600	15672208	17644152	14138672	19635496	timeout	15676032	timeout
14	Heap	6689072	21399280	39778944	12154872	63339536	10001424	23216032	104984632	13520552	261875424
	PermGen	14123648	18776520	20837304	15564752	17658744	14123552	18717496	21133848	15565496	17962432
15	Heap	6519856	25884560	42201208	17577528	78796376	10719808	26990848	timeout	24256360	timeout
	PermGen	14123408	18797808	20874568	15555624	17676864	14124544	18692536	timeout	15614544	timeout
16	Heap	6745312	25561216	42261008	17578808	78991864	10749064	27837328	timeout	23192048	timeout
	PermGen	14123888	18799072	20874680	15563512	17697200	14122240	18795040	timeout	15718768	timeout
17	Heap	6449472	22169640	42394208	13949384	78322480	10503864	23419120	timeout	13056032	timeout
	PermGen	14123312	18791728	20881544	15578928	17690624	14124424	18780688	timeout	15580904	timeout
18	Heap	6899336	21472560	36392864	11936432	72941760	10421168	22800072	94308848	12954928	259820232
	PermGen	14123856	18778888	20834296	15572608	17649664	14123952	18727424	21233664	15573416	17967944
19	Heap	6875144	19566400	42879112	12687157	79957824	10452872	20174600	timeout	13776568	timeout
	PermGen	14123816	18790728	20866344	15581312	17695800	14123928	18779224	timeout	15580280	timeout

Tabelle 11.3: Speicherbedarf (in Byte) zur Ausführung von Anfragen

Anfrage	**Datensatz & Verfahren / Messungen**	**500 Duplikate**			**1000 Duplikate**		
		SPARQL	**S-A M**	**S M**	**SPARQL**	**S-A M**	**S M**
1	Heap	9269480	30712872	13886072	12416960	34352096	34083248
	PermGen	14110576	19637952	15286528	14110152	19635280	15626048
2	Heap	9715816	28022016	15484296	12337248	34186280	44832496
	PermGen	14107216	19635080	15656352	14139064	19650001	15628136
3	Heap	8546416	34890017	18269240	12392792	42705150	44591864
	PermGen	14110152	19630141	15687552	14139004	19681312	15661232
4	Heap	8944632	28950600	15119976	12719560	32448440	35529123
	PermGen	14110608	19646744	15661104	14139392	19631504	15631832
5	Heap	8697880	30868224	12015832	12473832	40916584	40089128
	PermGen	14110400	19636672	15656192	14139144	19609488	15631152
6	Heap	11041616	27163072	19670056	15278024	50094832	47469528
	PermGen	14094872	18736056	15571640	14124600	18759944	15557712
7	Heap	11309464	27163072	20570512	13820184	57106280	58739312
	PermGen	14095088	18736056	15571088	14124320	18760976	15566432
8	Heap	11023560	35912512	29570688	15067632	63800701	62795904
	PermGen	14124072	18764400	15570904	14124504	18710003	15600256
9	Heap	11134264	32216384	24060272	14101504	60800008	45533624
	PermGen	14120664	18756536	15569640	14125264	18760312	15560664
10	Heap	11052472	26867976	23047160	15380552	50529104	42414536
	PermGen	14123864	18746040	15559800	14125604	18754776	15553472
11	Heap	9083840	34470784	13479336	12351976	41070768	44772264
	PermGen	14139432	19665792	15690816	14138992	19695768	15671664
12	Heap	8755376	27826344	13663328	12559896	42857344	44200280
	PermGen	14139104	19664600	15728640	14139192	19684776	15640323
13	Heap	9037136	31091152	18180536	12616656	44630040	44964824
	PermGen	14139488	19663891	15724256	14139160	19690656	15672560
14	Heap	10871160	30905640	25196224	13802816	62400768	38303280
	PermGen	14123600	18750488	15572360	14124472	18760104	15565972

15	Heap	10087568	45438400	13801680	14861896	70067232	timeout
	PermGen	14122472	18842040	15677360	14122296	18812248	timeout
16	Heap	10970856	32220952	40759816	15515440	71090016	timeout
	PermGen	14123616	18831336	15728640	14124624	18745328	timeout
17	Heap	11276000	36272348	31472472	14692384	67275888	61063448
	PermGen	14123864	18813845	15531373	14121168	18790680	15575410
18	Heap	11202192	32646920	25424304	14940232	42468720	61521672
	PermGen	14123880	18757336	15574520	14124136	18769080	15565744
19	Heap	10996848	37293528	31981480	14809456	70113928	61199840
	PermGen	14123960	18761912	15533800	14124072	18813584	15597992

Tabelle 11.4: Speicherbedarf (in Byte) zur Ausführung von Anfragen

Auch im Speicherverbrauch ist SPARQL am effizientesten. Im Vergleich zu den anderen Verfahren benötigt die Ausführung der SPARQL-Anfragen wesentlich weniger Speicherplatz. Je nach der Ontologie-Größe liegt der Speicherbedarf (Heap) zwischen 5 und 15 MB.

Die Verfahren Schema-Aware Mapping und Simple Mapping mit Reasoner benötigen am meisten Speicher. Der Speicherbedarf (Heap) bei den getesteten Anfragen liegt zwischen 17 und 261 MB.

Im Vergleich zu Schema-Aware Mapping und Simple Mapping mit Reasoner verbrauchen Schema-Aware Mapping und Simple Mapping mit Modifikation wesentlich weniger Speicher. Der Speicherverbrauch bei den Verfahren liegt zwischen 5 und 71 MB.

Bei den Ontologie-Größen 100, 200 und 500 benötigt Simple Mapping mit Modifikation bei allen Anfragen weniger Speicher (Heap) als Schema-Aware Mapping.

Bei den Ontologien mit 1000 Duplikaten ist der Speicherverbrauch bei den über Schema-Aware transformierten Anfragen 1, 2, 3, 4, 5, 7, 11, 12, 13, 15, 16 und 18 niedriger als bei Simple Mapping. Auch wie bei den Zeitmessungen wird Schema-Aware Mapping mit Modifikation bei größeren Ontologien effizienter als Simple Mapping mit Modifikation.

11.3 Ergebnisse von Anfragen

In Bezug auf die zurückgelieferten Ergebnisse aller ausgeführten Anfragen (Anfragen mit der Ausführungszeit weniger als 10 Minuten) gibt es keine Unterschiede zwischen den getesteten Verfahren. Bei allen Verfahren liefert dieselbe Anfrage das gleiche Ergebnis zurück. Zum Beispiel liefert die Anfrage 3 bei der Requirements-Ontologie mit 100 Duplikaten bei allen fünf Verfahren dieselbe Ergebnismenge mit 399 Elementen zurück. Bei der Ontologie mit 200 Duplikaten liefert die Anfrage auch bei allen Verfahren dieselbe Ergebnismenge mit 799 Elementen zurück. Das Gleiche gilt für die Ontologien mit 500 und 1000 Duplikaten.

Die Anzahl der Ergebniselemente aller getesteten Anfragen ist in Tabelle 11.5 dargestellt. Für die Tabelle gelten die gleichen Abkürzungen wie für die Tabellen 11.1 und 11.2.

Anfrage	100 Duplikate					200 Duplikate					500 Duplikate			1000 Duplikate		
	SPARQL	S-A M	S-A R	S M	S R	SPARQL	S-A M	S-A R	S M	S R	SPARQL	S-A M	S M	SPARQL	S-A M	S M
1	1					1					1			1		
2	100					200					500			1000		

3	399	799	timeout	799	timeout	1999	3999
4	1	1				1	1
5	1	1				1	1
6	1	1				1	1
7	100	200				500	1000
8	299	599	timeout	599	timeout	1499	2999
9	1	1				1	1
10	1	1				1	1
11	1	1	timeout	1	timeout	1	1
12	1	1	timeout	1	timeout	1	1
13	398	798	timeout	798	timeout	1998	3998
14	1	1				1	1
15	1	1				1	1 timeout
16	2	2	timeout	2	timeout	2	2 timeout
17	2	2	timeout	2	timeout	2	2
18	3	3				3	3
19	2	2	timeout	2	timeout	2	2

Tabelle 11.5: Anzahl der Ergebniselemente von Anfragen

11.4 Zeit-Messergebnisse zur Transformation von GReQL- in SPARQL-Anfragen

Die Ergebnisse zum Zeitbedarf der Transformation von GReQL- in SPARQL-Anfragen sind in Tabelle 11.6 dargestellt. Wie erwartet, wächst die Transformationszeit mit der Komplexität der Anfragen.

Anfrage	**Zeitmessung der Transformation**
1	2183.333
2	2306.666
3	2406.666
4	2420.333
5	2477.0

Tabelle 11.6: Zeitmessung (in ms) der Transformation von GReQL-Anfragen in SPARQL-Anfragen

Kapitel 12

Zusammenfassung

Im Rahmen dieser Arbeit wurde die Effizienz von Anfragen an OWL-Ontologien und Anfragen an TGraphen evaluiert. Dabei wurden die zwei unterschiedlichen Transformationsverfahren Schema-Aware Mapping und Simple Mapping getestet. Bei der Transformation einer OWL-Ontologie in einen TGraphen über Schema-Aware Mapping werden ein TGraph-Schema und ein TGraph erzeugt. Im Gegensatz zu Schema-Aware Mapping besitzt Simple Mapping ein schon vordefiniertes TGraph-Schema, das für alle OWL-Ontologien anwendbar ist. Damit wird bei der Transformation einer OWL-Ontologie in einen TGraphen über Simple Mapping nur ein TGraph - kein TGraph-Schema - generiert. Mit den Verfahren wurden entsprechend auch SPARQL-Anfragen in GReQL-Anfragen transformiert.

Ein wichtiger Punkt bei der Transformation von Anfragen war die Möglichkeit, Anfragen bei Vorhandensein der Property-Eigenschaften „transitiv“, „symmetrisch“, „äquivalent zu“, „invers zu“ und die `subPropertyOf`-Beziehungen zwischen Properties über ein Modifikationsverfahren zu modifizieren. Dementsprechend wurden 19 Anfragen entwickelt, die diesen Eigenschaften genügen. Die Anfragen wurden an zwei Ontologien gestellt: der Requirements- und der Family-Ontologie.

Insgesamt wurden die in Tabelle 12.1 dargestellten Testfälle evaluiert:

	Anfrage		**Datenstruktur**	
Nr	**Anfrageart**	**Transformationsart**	**Art der Datenstruktur**	**Transformationsart**
1	direkte SPARQL-Anfrage	keine	OWL-Ontologie	keine, Ausführung der OWL-Ontologie mit Reasoning
2	transformierte modifizierte GReQL-Anfrage	Schema-Aware Mapping	TGraph	Schema-Aware Mapping ohne Reasoning
3	transformierte GReQL-Anfrage	Schema-Aware Mapping	TGraph	Schema-Aware Mapping mit Reasoning
4	transformierte modifizierte GReQL-Anfrage	Simple Mapping	TGraph	Simple Mapping ohne Reasoning
5	transformierte GReQL-Anfrage	Simple Mapping	TGraph	Simple Mapping mit Reasoning

Tabelle 12.1: Evaluierte Testfälle

Den ermittelten Messungen - Zeitbedarf und Speicherbedarf - zufolge, ist die Ausführung von unmittelbaren SPARQL-Anfragen an OWL-Ontologien am effizientesten.

Als zeit- und speicheraufwendigsten ergaben sich die Schema-Aware Mapping- und Simple Mapping-transformierten Anfragen ohne Modifikation, also mit dem Einsatz eines Reasoners.

Dazwischen liegen die Schema-Aware Mapping- und Simple Mapping-transformierten Anfragen mit Modifikation (die Ausführung von Ontologien ohne Einsatz eines Reasoners). Dabei haben die komplexeren Anfragen an Ontologien mit größeren ABoxen (vielen Individuen) über Schema-Aware Mapping mit Modifikation bessere Messergebnisse erzielt als über Simple Mapping mit Modifikation. Damit hat sich die Hypothese, dass mit Hilfe des Modifikationsverfahrens und den gleichzeitigen Verzicht auf einen Reasoner die Ausführung von Anfragen effizienter sein soll, teilweise bestätigt. Die Hypothese hat sich also unter den vier Transformationsarten von SPARQL-Anfragen in GReQL-Anfragen bewährt, nicht aber unter allen getesteten Testfällen. Unmittelbare SPARQL-Anfragen an die mit dem Einsatz eines Reasoners ausgeführten Ontologien haben wesentlich bessere Ergebnisse erzielt als modifizierte GReQL-Anfragen an TGraphen, also an die Ontologien, die ohne Einsatz eines Reasoners in den TGraphen transformiert wurden.

Unabhängig von den oben erwähnten Evaluierungen wurde noch die Transformation von GReQL-Anfragen in SPARQL-Anfragen auf ihre Effizienz untersucht, dabei wurde die Transformationszeit von Anfragen ermittelt. Für dieses Ziel wurden fünf unterschiedlicher Komplexität GReQL-Anfragen entwickelt, deren Transformationszeit zwischen 2183,3 und 2477,0 ms liegt.

Ein interessanter Ausgangspunkt für weitere Arbeiten wäre die Ursachen - was nicht das Ziel dieser Arbeit war - zu finden, die die Ausführung der transformierten GReQL-Anfragen verzögern. Außerdem können noch weitere mögliche Anfragen, die optionale (OPTIONAL), alternative (UNION) Graphmuster und Filter-Bedingungen[1] enthalten, getestet werden.

[1]Optionale (OPTIONAL), alternative (UNION) Graphmuster und Filter-Bedingungen konnten zur Zeit der Erstellung dieser Arbeit noch nicht transformiert werden

Anhang A

Die von GReQL transformierten SPARQL-Anfragen

Anfrage 1

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fam:<http://www.semanticweb.org/ontologies/2011/6/Family#>

SELECT ?o1
WHERE {
  ?p fam:uriRef ?o1 .
  ?p rdf:type fam:Person .
  ?a rdf:type fam:Person .
  {
    ?p fam:isAcquainted ?a .
    FILTER ( ?o0 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person13" )
    ?a fam:uriRef ?o0 .
  }
}
```

Anfrage 2

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fam:<http://www.semanticweb.org/ontologies/2011/6/Family#>

SELECT ?o1
WHERE {
  ?p fam:uriRef ?o1 .
  ?p rdf:type fam:Person .
  ?a rdf:type fam:Person .
  {
    {
      {
        ?p fam:isAcquainted ?a .
      }
      UNION
      {
        {
          {
            ?p fam:livesWith ?a .
          }
```

```
          UNION
          {
            ?a fam:livesWith ?p .
          }
        }
      }
    }
    FILTER ( ?o0 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person13" )
    ?a fam:uriRef ?o0 .
  }
}
```

Anfrage 3

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fam:<http://www.semanticweb.org/ontologies/2011/6/Family#>

SELECT ?o0
WHERE {
  ?p fam:uriRef ?o0 .
  ?p rdf:type fam:Person .
  ?a rdf:type fam:Person .
  {
    {
      {
        {
          {
            {
              ?p fam:isAcquainted ?i0 .
              ?i0 fam:hasBloodRelationship ?a .
            }
          }
          UNION
          {
            ?p fam:isAcquainted ?a .
          }
        }
      }
      UNION
      {
        {
          {
            ?p fam:livesWith ?a .
          }
          UNION
          {
            ?a fam:livesWith ?p .
          }
        }
      }
    }
    FILTER ( ?o1 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person13" )
    ?a fam:uriRef ?o1 .
  }
}
```

Anfrage 4

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX fam:<http://www.semanticweb.org/ontologies/2011/6/Family#>

SELECT ?o2 ?o1
WHERE {
  ?p fam:uriRef ?o2 .
  ?agc fam:uriRef ?o1 .
  ?p rdf:type fam:Person .
  ?a rdf:type fam:Person .
  ?b rdf:type fam:Person .
  ?agc rdf:type fam:Person .
  {
    {
      {
        {
          {
            {
              ?p fam:isAcquainted ?i1 .
              ?i1 fam:hasBloodRelationship ?a .
            }
          }
          UNION
          {
            ?p fam:isAcquainted ?a .
          }
        }
      }
      UNION
      {
        {
          {
            ?p fam:livesWith ?a .
          }
          UNION
          {
            ?p fam:livesWith ?a .
          }
        }
      }
    }
    {
      {
        {
          ?a fam:hasCousin ?b .
        }
        UNION
        {
          ?a fam:hasCousin ?b .
        }
      }
       {
        {
        FILTER ( ?o3 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person13" )
        FILTER ( ?o0 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person5" )
        ?a fam:uriRef ?o3 .
        ?b fam:uriRef ?o0 .
        }
        ?agc fam:hasAdoptiveGrandChild ?a .
      }
    }
```

```
  }
}
```

Anfrage 5

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fam:<http://www.semanticweb.org/ontologies/2011/6/Family#>

SELECT ?o0 ?o1 ?o2 ?o5
WHERE {
  ?p fam:uriRef ?o0 .
  ?agc fam:uriRef ?o1 .
  ?s fam:uriRef ?o2 .
  ?c fam:uriRef ?o5 .
  ?p rdf:type fam:Person .
  ?a rdf:type fam:Person .
  ?b rdf:type fam:Person .
  ?agc rdf:type fam:Person .
  ?s rdf:type fam:Person .
  ?c rdf:type fam:Person .
  {
    {
      {
        {
          {
            {
              ?p fam:isAcquainted ?i3 .
              ?i3 fam:hasBloodRelationship ?a .
            }
          }
          UNION
          {
            ?p fam:isAcquainted ?a .
          }
        }
      }
      UNION
      {
        {
          {
            ?p fam:livesWith ?a .
          }
          UNION
          {
            ?p fam:livesWith ?a .
          }
        }
      }
    }
    {
      {
        {
        ?a fam:hasCousin ?b .
        }
        UNION
        {
          ?a fam:hasCousin ?b .
        }
      }
```

```
      {
        {
          {
            {
              {
                ?i4 fam:hasMother ?i5 .
                ?i5 fam:hasSpouse ?s .
              }
              ?i2 fam:hasMother ?i4 .
            }
            ?b fam:hasWife ?i2 .
          }
          {
            {
              {
                ?agc fam:hasAdoptiveChild ?c .
              }
              UNION
              {
                ?agc fam:hasChild ?c .
              }
            }
            {
              FILTER ( ?o4 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person13" )
              FILTER ( ?o3 = "http://www.semanticweb.org/ontologies/2011/6/Family#Person5" )
              ?a fam:uriRef ?o4 .
              ?b fam:uriRef ?o3 .
            }
          }
        }
        ?agc fam:hasAdoptiveGrandChild ?a .
      }
    }
  }
}
```

Anhang B

CD-ROM

Die beiliegende CD-ROM enthält

- diese Arbeit im PDF-Format (siehe Studienarbeit.pdf),
- Simple Mapping transformierte GReQL-Anfragen ohne und mit Modifikation (siehe SimpleMapping-TransformierteAnfragen.pdf),
- Requirements- und Family-Ontologien jeweils mit 100, 200, 500 und 1000 Duplikaten (im Verzeichnis `ontologies/`),
- Klassen `Metric` und `NameOfResultList` (im Verzeichnis `metric/`) und
- Implementierungen
  - zur Generierung der Requirements- und Family-Ontologien (im Verzeichnis `ontologyGenerator/`) und
  - zur Durchführung der Evaluierungen (im Verzeichnis `tests/`)

Literaturverzeichnis

[BE10] Daniel Bildhauer and Jürgen Ebert. Reverse Engineering Using Graph Queries, 2010.

[BG04] Dan Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema W3C Recommendation. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/, Februar 2004. letzter Zugriff: 5.11.2011.

[BHG10] Daniel Bildhauer, Tassilo Horn, and Eckhard Grossman. GReQL-Reference Card, 2010.

[CS06] Volker Claus and Andreas Schwill. *Informatik A-Z - Fachlexikon für Studium, Ausbildung und Beruf*. Dudenverlag, 2006.

[Doc] Oracle: Java SE Dokumentation: Java VisualVM. http://download.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html. letzter Zugriff: 5.11.2011.

[Ebe10] Jürgen Ebert. Mitschrift zur Vorlesung Software Reengineering, 2009-2010.

[GCT] Oracle: Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html?ssSourceSiteId=ocomenpar_gc.ergonomics.default_size. letzter Zugriff: 5.11.2011.

[HS10] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language - W3C Working Draft. http://www.w3.org/TR/2010/WD-sparql11-query-20101014/, Oktober 2010. letzter Zugriff: 5.11.2011.

[Mar06] Katrin Marchewka. GReQL 2. Diplomarbeit, Universität Koblenz-Landau, Campus Koblenz, Juni 2006.

[Mas06] Jon Masamitsu. Weblog: Presenting the Permanent Generation. http://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation, November 2006. letzter Zugriff: 5.11.2011.

[MM04] Frank Manila and Eric Miller. RDF Primer-W3C Recommendation. http://www.w3.org/TR/rdf-primer/, Februar 2004. letzter Zugriff: 5.11.2011.

[MPSP09] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax W3C Recommendation. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/, Oktober 2009. letzter Zugriff: 5.11.2011.

[PHRS08] Markus Krötzsch Pascal Hitzler, Sebastian Rudolph, and York Sure. *Semantic Web*. Springer-Verlag, 2008.

[PS08] Eric Prudhmmeaux and Andy Seaborne. SPARQL Query Language for RDF-W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, Januar 2008.
letzter Zugriff: 5.11.2011.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. Web Ontology Language: Semantics and Abstract Syntax W3C Recommendation.
http://www.w3.org/TR/2004/REC-owl-semantics-20040210/, Februar 2004.
letzter Zugriff: 5.11.2011.

[Sch11] Hannes Schwarz. Erweiterungen des TrOWL-Tools und Implementierung der `greql2sparql`-Applikation, 2011.

[SE10] Hannes Schwarz and Jürgen Ebert. Bridging Query Languages in Semantic and Graph Technologies, 2010.

[TPR10] Edward Thomas, Jeff Z. Pan, and Yuan Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure, 2010.

[Vis] VisualVM. http://visualvm.java.net.
letzter Zugriff: 5.11.2011.