Implementierung eines Sichtenmechanismus für ontologiebasierte Metadaten

Daniel Oberle, Raphael Volz
{oberle, volz}@aifb.uni-karlsruhe.de

Inhalt

Anhang

1 Einleitung

Kardinalziel dieser Arbeit ist die Realisierung eines Sichtenmechanismus für Ontologien basierend auf der proprietären KAON Semantic Web Infrastruktur des hiesigen Instituts (vgl. Kapitel 2.5). Innerhalb der Informatik dienen Ontologien der formalen Spezifikation einer Konzeptualisierung [Gru93]. Dabei ordnet man den Entitäten einer interessierenden Miniwelt Begriffe[1] zu, strukturiert diese und setzt sie einander in Beziehung. Ergebnis ist eine formale Wissensrepräsentation der betrachteten Domäne, zumeist in einer Beschreibungslogik. Eine Sicht auf ein solches konzeptuelles Modell selektiert nur eine gewisse Teilmenge der Begriffe und Relationen, was einer Restrukturierung gleichkommt. In der Datenbankwelt ist derartiges bereits etablierte Technik, innerhalb der Wissensrepräsentation sind Publikationen dazu allerdings rar.

1.1 Verwandte Arbeiten

Wie bereits erwähnt, kommt heute kein Datenbanksystem ohne Sichten (*Views*) aus. Das gilt insbesondere für die derzeit dominierende relationale Technologie. Eine View ist dort verwirklicht als benannte Anfrage, auf die in anderen Bezug genommen werden kann. Man erreicht damit eine einfache Anpassung zur Integration heterogener und verteilter Systeme, z.B. Data Warehouses. Ein zweiter Aspekt ist das Verstecken der Eigenheiten mehrerer Datenbasen zur einheitlichen Konsolidierung sowie Interoperabilität in verteilten Systemen. Eine solche Funktionalität findet man typischerweise bei Mediatoren. Im wesentlichen ist die Sichtenproblematik hier erforscht. Neuere Publikationen zu relationalen Sichten behandeln meist das Problem der Zugriffszeit. Soll die korrespondierende Anfrage jedesmal neu ausgewertet werden? Ist es nicht effizienter das Ergebnis dauerhaft zu speichern? Was geschieht aber dann mit Änderungen auf den ursprünglichen Daten? Dieser Problematik begegnet man in der Literatur unter den Schlagwörtern *View-Update-* oder *View-Materialization-Problem* [GMS93].

Mag der Sichtenmechanismus im relationalen Datenmodell schon etabliert sein, steckt er bei den objektorientierten Datenbanksystemen noch in den Kinderschuhen. Grund dafür ist u.a., daß in der OO-Welt das Verhalten einer Klasse (in Form von Methoden) berücksichtigt werden muß. Seit mehreren Jahren sind OO-Sichten Gegenstand der Forschung und aufgrund der Verwandtschaft zu Ontologien hier relevant. Eine Sicht, in diesem Kontext auch virtuelles Schema genannt, besteht aus reellen und virtuellen Klassen, neuen Attributen und Methoden. Sie basiert auf einem Basisschema, das wiederum virtuell sein kann, und importiert davon die gewünschten Ressourcen. Ein Zitat aus [SAC] verdeutlicht diesen Ansatz, an dem sich auch das vorliegende Dokument orientiert:

> *„A view is ... a special kind of schema to which some constraints on its use and definitions are imposed. We will use the term virtual schema as a synonym of view in what follows and the term real schema will be used in contrast to virtual schema to avoid confusion.*

[1] In Wissensmanagament und Datenmodellierung verwendet man synonym „Konzept“. Es handelt sich dabei um eine falsche Übersetzung des englischen „concept“, zu deutsch „Begriff“. Daher rührt auch das Substantiv Konzeptualisierung, das hier passender „Begriffsbildung“ oder „Begriffszuordnung“ lauten sollte.

Like a real schema, a view includes definitions of classes, methods, types, functions and named objects. It may also import and export definitions from other schemas, although these mechanisms will be given a slightly different semantics. In addition, virtual defnitions can be included in a virtual schema.“ [SAC]

Eine Lösung definiert die Population virtueller Klassen beispielsweise mittels Anfrage [AB91]. Dadurch entstehen sogenannte imaginäre Instanzen. Sehr ähnliche Ansätze finden sich in [Kuno], [Tec94]. Dabei wird die objektorientierte Theorie mehr oder weniger stark formalisiert.

Eine weitere Gruppe von Arbeiten definiert eine Sicht einfacher als virtuelle Klasse, die von einer oder mehreren Klassen des Basisschemas abgeleitet ist (cf. [Ber92], [BFN95]). eXoT/C hingegen behandelt eine Sicht als externes Schema, das zunächst nur aus abgeleiteten Klassen besteht. Virtuelle Klassen werden in einem separierten Schema definiert (cf. [Dob97]). Dieser Ansatz findet sich auch in UniSQL [KK95].

Wie auch im relationalen Datenmodell kommen mit einem Sichtenmechanismus Effizienzfragen auf. Es stellt sich hier ebenfalls das View-Materialisierungs-Problem. [GMS93], [GGMS] und [SLT] zeigen Lösungen dafür.

1.2 Motivation

Verwandtschaften zwischen Objektorientierung und Ontologien in der Wissensrepräsentation sind nicht zu leugnen. Allerdings berücksichtigt OO das Verhalten von Entitäten in Form von Methoden, das bei Ontologien gänzlich unter den Tisch fällt. Viele Probleme, wie beispielsweise Interface-Auflösung, erledigen sich damit. Statt dessen bieten viele wissensbasierte Systeme die Möglichkeit mit Hilfe einfacher Axiome neues Wissen zu generieren. Diese Arbeit betritt mit seinem Sichtenmechanismus auf Ontologien trotz der Ähnlichkeiten Neuland. Die Situation im Semantic Web zeigt nämlich einige Besonderheiten auf. Zum einen multiple Instantiierung – eine Instanz kann ohne weiteres mehreren Konzepten angehören (cf. [RDFS00]). Vererbung zwischen Konzepten folgt einer Mengeninklusionssemantik und ist nicht strukturell wie in der objektorientierten Welt (cf. [PH01]). Dem OO-Paradigma widerspricht ebenfalls die Tatsache, daß Relationen und Konzepte grundsätzlich getrennt zu betrachten sind (cf. [RDFS00]). Die Arbeit kann dennoch grundlegende Ansätze aus der OO-Welt nutzen, wie bereits in 1.1 geschildert.

Ausgehend von einer Basisontologie möchte man Konzepte und Relationen ausblenden und statt dessen neue virtuelle hinzunehmen. Der angestrebte Mechanismus muß dabei garantieren, daß die vom Benutzer definierte Sicht stets mit der Faktenbasis konsistent bleibt. Dazu ist es notwendig das zugrundeliegende Modellierungsvokabular auf abstrakter Ebene zu betrachten und Axiome zur Korrektheit zu definieren. Eine Liste konkreter Anforderungen findet sich in Subkapitel 1.3.

Die grundsätzliche Motivation für Sichten unterscheidet sich aber im wesentlichen nicht von der bei relationalen und objektorientierten Datenbanken. Das virtuelle Restrukturieren soll es dienstnehmenden Applikationen ermöglichen mittels Sichten gemeinsam genutzte Datenobjekte auf ihre Bedürfnisse anzupassen ohne andere Anwendungen dabei zu tangieren. Zusammenfassend ergeben sich allgemein die folgenden Gründe [AHV95]:

- *Flexibilität*
  Im Vordergrund steht primär die flexible Anpassung eines Datenhaltungssystems auf die Bedürfnisse einer Anwendung.
- *Größe des globalen Schemas*
  Aus Gründen der Übersichtlichkeit und Verwaltbarkeit ist man evtl. nur an einer Teilmenge des globalen Schemas interessiert.
- *Autorisierung*
  Man möchte nur relevante Daten zur Verfügung stellen, d.h. eine Regulierung des Zugriffs durch Visibilität.
- *Integration*
  In heterogenen Umgebungen ist es u.U. erstrebenswert, Schemata für unterschiedliche Agenten anzupassen.

Ein Beispiel soll den avisierten Sichtenmechanismus auf Ontologien motivieren und den Leser durch das Dokument begleiten. Die in Abbildung 1 gezeigte Ontologie strukturiere das Wissen eines mittelständischen Unternehmens, welches sowohl als Hersteller für End- als auch für Zwischenprodukte agiere. Natürlich sprengt eine tatsächliche Unternehmensontologie hier den Rahmen des Dokumentierbaren. Ein solch stark vereinfachtes Exempel dient lediglich der Übersichtlichkeit.

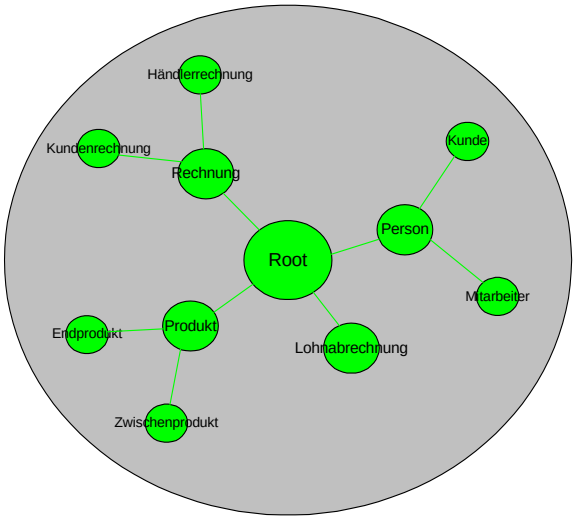


Abbildung 1: Beispielontologie

Verschiedensten Benutzerkreisen soll der Zugriff auf die korrespondierende Faktenbasis gewährt sein, allerdings nicht immer im selben Umfang. So plant das Unternehmen beispielsweise einen Netzauftritt mit eCommerce-Funktionalität für Endkunden und auch im Business-to-Business-Bereich (B2B). Für die Mitarbeiter sind diverse Applikationen im eigenen Intranet angestrebt und schließlich möchte die Personalabteilung Stammdaten und Lohnabrechnungen verwalten. Ein Knowledge Warehouse soll darüber hinaus eine Wissensgewinnung per Business Intelligence Applikationen ermöglichen. Statt nun für jedes Szenario eine neue Ontologie zu definieren, bietet sich die Formulierung folgender Sichten an:

- Sicht 1: Intranet für Mitarbeiter
- Sicht 2: Web-Shopping für Endkunden
- Sicht 3: eCommerce B2B
- Sicht 4: Buchhaltung
- Sicht 5: Knowledge Warehouse

1.3 Anforderungen

- *Robustheit*
  Die Sicht sollte gegenüber Änderungen ihrer ursprünglichen Ontologie so resistent wie möglich sein. Insbesondere in einem dynamischen Umfeld wie dem WWW sind Schemaänderungen an der Tagesordnung. Diese Anforderung zieht nach sich

- *Dynamik*
  Eine Sichtendefinition ist nicht statisch zu speichern, angemessener wäre die Information wie sie sich aus der ursprünglichen Ontologie errechnet. Das führt zu einer

- *Sichtendefinitionssprache*
  Ähnlich den Sichten in der objektorientierten Welt soll auch hier eine einfache Sprache zur Verfügung stehen. Diese leistet die oben genannten Anforderungen. Man kann sich dabei anlehnen an Sprachen zur Definition von Wrappern (wie bsp. OEM), bei denen sehr ähnliche Verhältnisse herrschen.

- *Faktenbasis*
  Im speziellen Falle der KAON-Infrastruktur sollen sowohl Ontologie als auch Sicht auf dieselbe Faktenbasis zugreifen. Replikation, Inkonsistenzen, Materialisierungsfragen oder View-Update-Problemen geht man damit aus dem Wege.

- *Konsistenzaxiomatik*
  Eine Ontologie strukturiert das Wissen in einer Faktenbasis. Axiome müssen dafür Sorge tragen, daß eine Sicht dieses in konsistenter Weise erledigt.

- *GUI*
  Dem Endbenutzer soll es später möglich sein, eine Sicht komfortabel mittels graphischer Benutzeroberfläche zu definieren.

- *Transparenz*
  Es soll einer dienstnehmenden Applikation stets transparent bleiben, daß sie auf einer Sicht operiert. D.h. gleich ob Ontologie oder Sicht, die Schnittstelle in Form des KAON-API bleibt dieselbe.

- *Kaskadierung*
  Ausgehend von einer Ontologie müssen beliebig viele Sichten darauf definierbar sein. D.h. eine Sicht kann als Basis nicht nur das ursprüngliche Schema, sondern wiederum eine Sicht haben.

- *Autorisierung*
  Die Möglichkeit den Zugriff auf die Faktenbasis zu beschränken ist inhärent durch Visibilität gegeben. Eine Rechteverwaltung auf Benutzerebene wird hier nicht diskutiert.

- *Virtualität*
  Der Sichtenmechanismus sollte die Definition von virtuellen Attributen, Assoziationen und Klassen ermöglichen.

1.4 Aufbau des Dokuments

Zunächst schildert Kapitel 2 sämtliche Technologien, die der Arbeit zugrundeliegen. Allesamt sind im Bereich „Semantic Web“ einzuordnen, in welchem sich der Sichtenmechanismus auch niederschlagen wird. Kapitel 3.1 listet zunächst die Elemente einer Sicht und definiert dazu ein korrespondierendes Vokabular. Darauf baut die Konsistenzaxiomatik in Kapitel 4 auf, die einen bedeutenden Teil der Arbeit in Anspruch nimmt. Es folgt schließlich die Dokumentation zur Integration des Sichtenmechanismus in die bereits vorhandene Infrastruktur (Kapitel 5). Der Ausblick listet zuletzt einige Aspekte und offene Fragen, die im Rahmen dieser Arbeit nicht beantwortet werden konnten.

1.5 Zusammenfassung

Von Beginn an stand die Forderung eines tatsächlich funktionierenden Sichtenmechanismus inmitten der proprietären KAON Semantic Web Infrastruktur [HMSV01]. Orientiert man sich an vergleichbaren, meist theoretischen Arbeiten aus der OO-Welt, keine Selbstverständlichkeit. Aus diesem Grunde ist die hier vorgestellte Lösung, genannt KAON-Views, sehr zielgerichtet. Die o.g. Karlsruhe Ontologie (KAON) Semantic Web Infrastruktur verwaltet und speichert Ontologien mit Hilfe des Resource Description Framework [RDF]. Aussagen, auch Statements genannt, mit einer simplen Tripelstruktur (subject, predicate, object) dienen dabei der Annotierung von Web-Ressourcen. RDF ist bereits vom World Wide Web Consortium (W3C) abgesegnet und wird als Basistechnologie des Semantic Web zum Einsatz kommen.

Der Begriff „Sicht“ ist auf der Schemaebene anzusiedeln. Demnach kommt automatisch RDF Schema als Modellierungssprache ins Visier – ein RDF-Vokabular vorgeschlagen vom W3C zur Formulierung einfacher Ontologien [RDFS00]. Bei näherer Betrachtung kristalliert sich allerdings dessen Untauglichkeit in Bezug auf Sichtenformulierung heraus. Grund dafür ist seine flexible Modellierungsarchitektur; sie führt letztlich zu einer Reihe von Problemen (siehe 2.4). Die Alternative heißt RDFS Fixed Architecture (RDFS(FA)), welche sämtliche Probleme elegant umgeht [PH01].

Ausgehend von RDFS(FA) formalisiert das Dokument zunächst, was man unter einem Schema, oder hier synonym unter einer Ontologie, versteht. Darauf aufbauend schließt sich die Formalisierung einer Subontologie, d.h. einer Sicht, an. Es handelt sich dabei ebenfalls um eine Ontologie, allerdings muss sie eine Menge von Konsistenzaxiomen erfüllen, deren Formulierung einen Großteil dieser Arbeit ausmacht. Sämtliche Formalisierungen stehen im formalen Modell von RDF. Alternativen wären XML-Serialisierungen von RDF oder dessen Graphmodell, beide sind aber weniger gut geeignet. Erstere sind nicht eindeutig und das Graphmodell dient vorallem der Visualisierung. Der Leser mag sich wundern, warum man nicht Modelltheorie o.ä. dazu heranzog. Der Grund dafür ist Pragmatismus – wie bereits erwähnt, basiert die KAON-Infrastruktur auf RDF, so daß früher oder später eine Umsetzung nötig würde. Die Konsistenzbedingungen an eine Subontologie hantieren also im wesentlichen mit Tripeln (subject, predicate, object) $\in$ *Statements*. Im Sinne der Mathematik werden nun Allquantor, Existenzquantor, sowie die üblichen logischen Junktoren herangezogen. Damit bleiben die Axiome auf abstrakter Ebene, fernab von Entwurfs- und Implementierungsfragen.

Nach der umfassenden Formulierung der Konsistenzbedingungen schließt sich die Erweiterung der KAON Infrastruktur an. Die objektorientierte Schnittstelle ist in ihrer Funktionalität entsprechend zu ergänzen. Dieser Schritt umfaßt die Realisierung von OO-Repräsentanten zu Subontologie, virtuellen Konzepten, virtuellen Relationen usw. Ein Dienstnehmer sollte dabei von Virtualität möglichst unberührt bleiben, d.h. er sollte transparent auf einer Sicht arbeiten können.

Schließlich stellt sich die Frage, wie die Axiome letztendlich operationalisiert werden. Auf der Hand liegt zunächst in der objektorientierten Welt zu bleiben und die verwendeten Quantoren implizit zu codieren. Die Entwurfsentscheidung fiel jedoch auf die flexiblere und interessantere Lösung mit Inferenzmaschine. Kernidee dabei ist die Transformation eines Tripels (subject, predicate, object) $\in$ *Statements* in ein logisches Prädikat statement(subject, predicate, object) [WR00]. Zum Einsatz kommt die auf F-Logic-beruhende com.ontoprise.inference.Evaluator Inferenzmaschine, auch Ontobroker genannt [DBSA]. Bei der Umkodierung in F-Logic kristallisieren sich zwei Typen von Axiomen heraus: zum einen sind das Hornregeln, die neue Statements, beispielsweise zur Definition der Klassenhierarchie, einfügen. Auf der anderen Seite stehen Konsistenzbedingungen. Bei derartigen Hornregeln steht ein violation-Prädikat im Kopf, dessen Erfülltheit entsprechend widerlegt werden muß.

KAON-Views, wie der Sichtenmechanismus insgesamt genannt wird, greift an verschiedenen Stellen in der Infrastruktur ein und umfaßt folgende Punkte: zunächst die Erweiterung des KAON-API um neue Klassen sowie Änderungen an vorhandenen Klassen. Nicht zu vergessen die Konsistenzaxiomatik im formalen Modell von RDF bzw. die korrespondierenden Regeln in F-Logic. Der Axiomcheck per com.ontoprise.inference.Evaluator, womit diese Inferenzmaschine zwingender Bestandteil wird, und die Schnittstelle dazu. Angedacht, jedoch aus Zeitgründen nicht realisiert, sind darüber hinaus ein KAON-Views Plug-In zur komfortablen Konfiguration der Axiome, sowie die Erweiterung von KAON-SOEP, dem graphischen Ontologieeditor, zur Sichtendefinition.

2 Technologien

Dieses Kapitel gibt einen kurzen Abriß über die der Arbeit zugrundeliegenden Technologien, den sich der erfahrene Leser selbstverständliclh sparen kann. Dargelegt wird die Chronologie der Entwicklung im WWW, angefangen bei SGML und dessen wichtigste Applikation HTML, über XML bis hin zum Semantic Web und den korrespondierenden W3C-Standards RDF und RDFS. Ein nächstes Subkapitel schildert, daß insbesondere RDFS nicht als Basis für einen Sichtenmechanismus taugt und statt dessen RDFS(FA) den Vorzug bekommt. Es folgt eine Erläuterung der Semantic Web Infrastruktur des hiesigen Instituts.

2.1 XML

Die ISO-genormte *Standard Generalized Markup Language* (*SGML*) entstammt den 80‘er Jahren und war motiviert durch das US-amerikanische Verlagswesen. Es handelt sich bei SGML im wesentlichen um eine Grammatik zur Formulierung von sogenannten Dokumenttypdefinitionen (DTD). Eine solche beschreibt ihrerseits Regeln wie eine Instanz, d.h. ein konkretes Dokument dieses Typs, auszusehen hat. Markups (auch Tags genannt) strukturieren den Text in derartigen Ressourcen. Die wohl bekannteste DTD ist die *Hypertext Markup Language* (*HTML*) im WWW.

Aufgrund seiner mehrere hundert Seiten umfassenden Dokumentation hat SGML nicht zu Unrecht den Ruf einer Geheimwissenschaft. Das World Wide Web Consortium (W3C) extrahierte deshalb eine Teilmenge, die mit kleinen Änderungen in den 1999 definierten Standard *eXtensible Markup Language* (*XML*) mündete. Motivation dafür waren die Unzulänglichkeiten von HTML, welches einen Text ausschließlich mit Präsentations-spezifischen Markups umkleidet. Die Kernidee von XML ist die Separation von Struktur, Präsentation und Inhalt für zukünftige Sprachen im WWW. HTML wird dabei in bestimmten Bereichen ersetzt durch eine domänenspezifische DTD plus passenden Klienten. So gibt es beispielsweise eine Chemical Markup Language (CML) zur Beschreibung von Molekülen zusammen mit dem zugehörigen Browser JUMBO, der aus den Dokumenten grafische Modelle kreiert. Folgendes Diagramm soll einen ersten Überblick geben.

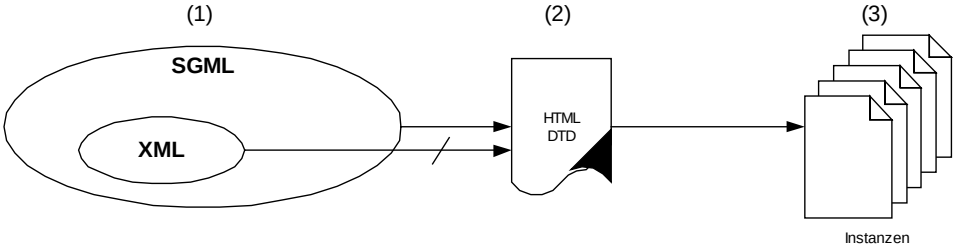


Abbildung 2: SGML, XML, DTD und Instanzen

Die formale Grammatik (N_{SGML}, T_{SGML}, P_{SGML}, S_{SGML}) definiert wie eine DTD auszusehen hat (1); genauer: mit ihren Produktionen generiert sie die Menge aller möglichen DTDs. Exakt ein Element dieser Menge entspricht der Dokumenttypdefinition von HTML. Geregelt wird im wesentlichen welche Markups es gibt, wie sie geschachtelt sein können und über welche Attribute sie verfügen. Um den Leser auf die Problematik zu sensibilisieren und ihm eine Vorstellung zu vermitteln, skizziert die folgende, bewußt abgekürzte und unvollständige Grammatik XML $\subset$ SGML (cf. [XML]):

```
N_XML = doctypedecl, markupdecl, elementdecl, ...
T_XML = ISO/IEC 10646
P_XML = doctypedecl     ::= '<!DOCTYPE' S Name (S ExternalID)? S?        ? = optional
                            ('[' (markupdecl | PEReference | S)* ']' S?)? '>'
        markupdecl      ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment
        elementdecl     ::= '<!ELEMENT' S Name S contentspec S? '>'
        contentspec     ::= 'EMPTY' | 'ANY' | Mixed | children
        AttlistDecl     ::= '<!ATTLIST' S Name AttDef* S? '>'
        AttDef          ::= S Name AttType S DefaultDecl
                        ...
        S               ::= (#x20 | #x9 | #xD | #xA)+                    Leerraum
        Name            ::= (Letter | '_' | ':') (NameChar)*
                        ...
S_XML = doctypedecl
```

Abbildung 3: XML Grammatik (cf. [XML])

In Schritt (2) kann nun die DTD wiederum als Grammatik (N_{DTD}, T_{DTD}, P_{DTD}, S_{DTD}) aufgefaßt werden. Sie generiert die Menge aller gültigen Dokumente zu einem speziellen Dokumenttyp. Als Beispiel, aus Gründen der Übersichtlichkeit wiederum gekürzt, dient hier HTML (cf. [HTML]). Es sei an dieser Stelle darauf hingewiesen, daß die Mächtigkeit von XML nicht ausreicht zur Definition der HTML-DTD. Die Wurzel diesen Übels liegt bei den sogenannten Bachelor-Tags, das sind alleinstehende Tags wie
 oder <HR>. Abhilfe schafft *XHTML*, das versucht mit einer XML-konformen DTD seinem Vorbild so nahe wie möglich zu kommen.

```
N_HTML = html, head, body, title, meta...
T_HTML = ISO/IEC 10646
P_HTML = html           ::= '<HTML' (S lang)? (S dir)? '>' head body '</HTML>'
         lang           ::= 'LANG=' Letter*
         dir            ::= 'DIR=' ( 'ltr' | 'rtl' )
         head           ::= '<HEAD' (S lang)? (S dir)? (S profile)? '>'
                                title? base? meta? '</HEAD>'
                        ...
         body           ::= '<BODY' ... '>' (div | address | table | form ...)+ '</BODY>'
         address        ::= '<ADDRESS' ...  '>' (#PCDATA | fontstyle | phrase | ...)*
                                '</ADDRESS>'
                        ...
         S              ::= (#x20 | #x9 | #xD | #xA)+                    Leerraum
S_HTML = html
```

Abbildung 4: HTML Grammatik (cf. [HTML])

Dokumente, die Element der aus einer DTD generierten Instanzenmenge sind, werden im XML-Jargon *gültig* (*valid*) genannt. Demgegenüber stehen *wohlgeformte* (*well-formed*) Dokumente, die einer weiteren Grammatik gehorchen und nicht Instanz eines Dokumenttyps sein müssen. Im wesentlichen wird gefordert, daß der Name im End-Tag eines Elements identisch sein muß mit dem im Start-Tag, kein Attributname darf mehr als einmal in demselben Start-Tag erscheinen, Zeichen, auf die mittels einer Zeichenreferenz verwiesen wird, müssen zur Produktion für Char passen usw. Diese Grammatik (N_{WF}, T_{WF}, P_{WF}, S_{WF}) stellt wesentlich niedrigere Anforderungen und ist unten kurz skizziert (cf. [XML]). Jedes valide Dokument ist auch wohlgeformt, die Umkehrung gilt nicht.

N_{WF} =	document, prolog, element, Misc, EmptyElemTag, ...	
T_{WF} =	ISO/IEC 10646	
P_{WF} =	document	::= prolog element Misc*
	prolog	::= ...
	element	::= EmptyElemTag | STag content ETag
	STag	::= '<' Name (S Attribute)* S? '>'
	Attribute	::= Name Eq AttValue
	content	::= (element | CharData | Reference | CDSect | PI | Comment)*
	ETag	::= '</' Name S? '>'
		...
	Name	::= (Letter | '_' | ':') (NameChar)*
	S	::= (#x20 | #x9 | #xD | #xA)+ Leerraum
		...
S_{WF} =	document	

Abbildung 5: Grammatik zur Wohlgeformtheit (cf. [XML])

Entsprechend der Zweiteilung in gültige und wohlgeformte Dokumente existieren auch validierende und nichtvalidierende Parser. Beide verbuchen auf der Habenseite die Wiederverwendbarkeit, was ein bedeutendes Argument für den Einsatz von XML darstellt. Ein XML-Dokument ist auch als Baum zu begreifen, wobei Knoten den Tags entsprechen, Blätter dem Inhalt eines Elements und Kanten den Attributen. Eine XML-Datei oder –Serialisierung entsteht durch DFS-Traversierung über diesen Baum. Mit den Attributtypen ID und IDREF ist ein Verweisen auf Elemente möglich – der Baum wird dadurch zum Graph. Die Illustration unten zeigt ein einfaches Beispiel in XML-Syntax und als Baum.

```
<Adresse>
      <Name>Daniel Oberle</Name>
      <Strasse>Mozartstr. 15</Strasse>
      <Stadt>76448 Durmersheim</Stadt>
</Adresse>
```

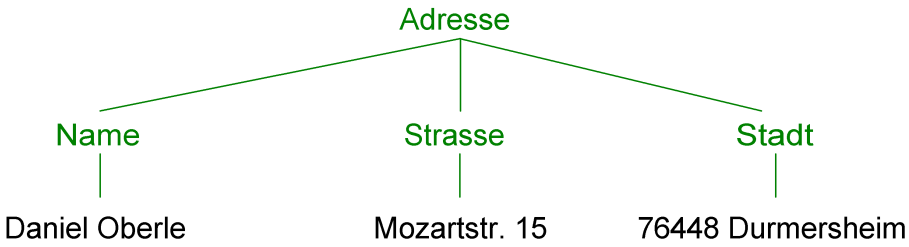


Abbildung 6: XML Beispiel

Um den XML-Standard herum sind in der Zwischenzeit allerlei Extensionen entstanden. Bemerkenswert sind vor allem die *XML-Style Sheets* (*XSL*), die die avisierte Trennung von Inhalt, Struktur und Layout perfekt machen, sowie Mechanismen zur Transformation (*XSLT*). Daneben einige Spezifikationen zur Referenzierung und Selektion: *XLink*, *XPath*, *XPointer*, *XQL*, *XML-QL* uvm.

2.2 RDF

Auch mit XML ändert sich nichts an der Tatsache, daß das WWW immer noch auf den Menschen ausgerichtet ist. Die schier unendlichen Ressourcen sind zwar maschinenlesbar, lassen aber jegliche maschinenverständliche Semantik vermissen. Eine Domäne kann per XML mit beliebig vielen DTDs kodiert werden; die Kommunikationspartner sollten sich deshalb vor dem Datenaustausch auf eine DTD geeinigt haben. Um diesen

Unzulänglichkeiten zu begegnen, definierte das W3C zur Jahrtausendwende das sogenannte *Resource Description Framework* (*RDF*). Es soll o.g. Problematik mit Hilfe von Metadaten, in diesem Sinne „Daten über Daten“, entsprechender Annotierung von Ressourcen sowie Separierung von Inhalt und Struktur lösen. Damit wurde der Grundstein für die Verarbeitung von Metadaten im WWW gelegt. Es profitieren davon insbesondere Suchmaschinen, intelligente Softwareagenten, digitale Signaturen, Inhaltsbeschreibungen uvm. Die Spezifikation definiert ein formales Modell, das wie folgt aussieht (cf. [RDF]):

Trägermengen
Menge *Resources*
Menge *Literals*
Teilmenge $Properties \subset Resources$
Relation $Statements \subset Properties \times Resources \times (Resources \cup Literals)$
dadurch entstehen Tripel (<predicate>, <subject>, <object>)

Typisierung
RDF:type $\in$ *Properties*
(RDF:type, sub, obj) $\in$ *Statements* $\rightarrow$ sub $\in$ *Resources* und obj $\in$ *Resources*

Reifikation
RDF:Statement $\in$ *Resources\Properties*
{RDF:predicate, RDF:subject, RDF:object} $\subset$ *Properties*
Die Reifikation eines Tripels (pred, sub, obj) $\in$ *Statements* ist ein Element r $\in$ *Resources* und
s1: (RDF:predicate, r, pred) $\in$ *Statements*
s2: (RDF:subject, r, subj) $\in$ *Statements*
s3: (RDF:object, r, obj) $\in$ *Statements*
s4: (RDF:type, r, [RDF:Statement]) $\in$ *Statements*

Kollektionen
{RDF:Seq, RDF:Bag, RDF:Alt} $\subset$ *Resources\Properties*
Ord = {RDF:_1, RDF:_2, RDF:_3, ...} $\subset$ *Properties*

Description
$d \in 2^{Statements}$

Abbildung 7: RDF Model (cf. [RDF])

Zentral ist die Rolle des *Statements*. Ein solches Tripel besteht aus einer Ressource, die es zu beschreiben gilt (Subjekt der Aussage eines URIs), sowie einer Eigenschaft (Property) mit zugehörigem Wert (Objekt der Aussage). Letzterer kann Literal oder wiederum Ressource sein. Die Reifikation eines Statements erlaubt „Aussagen über Aussagen“.

Neben dieser formalen Repräsentation gibt es auch eine graphbasierte und, insbesondere für den Austausch gedacht, eine XML-DTD deren Syntax unten skizziert ist (cf. [RDF]). Bezüglich des Graphen kann man RDF als Generalisierung von XML ansehen. Wie bereits erwähnt, ist ein XML-Baum per „IDREF“ zwar zum Graphen erweiterbar, die Kantenbezeichnung bleibt allerdings auf eben diese Zeichenkette festgelegt. RDF hingegen erlaubt beliebige Referenzen mit beliebigen Benennungen.

Abbildung 8 entstammt [RDF] und zeigt das Statement “Ora Lassila“ ist Creator von http://www.w3.org/Home/Lassila, in formaler Schreibweise also (Creator, “Ora Lassila“, http://www.w3.org/Home/Lassila). Zunächst steht die XML-Serialisierung des Statements, gefolgt von der graphbasierten Darstellung. Dabei sind URIs visualisiert als runde Knoten und Literale als Rechtecke.

```
<?xml version="1.0"?>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description about="http://www.w3.org/Home/Lassila">
      <Creator>Ora Lassila</Creator>
    </rdf:Description>
  </rdf:RDF>
```

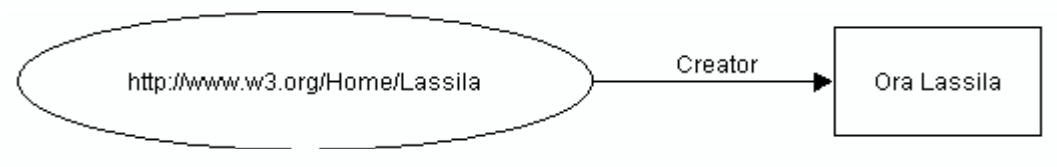


Abbildung 8: RDF Beispiel (cf. [RDF])

RDF	::=	['<rdf:RDF>'] obj* ['</rdf:RDF>']
obj	::=	description | container
description	::=	'<rdf:Description' idAboutAttr? bagIdAttrb? propAttr* '/>' |
		'<rdf:Description' idAboutAttr? bagIdAttrb? propAttr* '>' |
		propertyElt* '</rdf:Description>' | typedNode
container	::=	sequence | bag | alternative
idAboutAttr	::=	idAttr | aboutAttr | aboutEachAttr
idAttr	::=	'ID="' IDSymbol '" '
aboutAttr	::=	'about="' URI-Reference '" '
aboutEachAttr	::=	'aboutEach="' URI-Reference '" ' | 'aboutEachPrefix="' string '" '
	...	
propAttr	::=	propName '="' string '" '
typedNode	::=	'<' typeName idAboutAttr? propAttr* '/>' |
		'<' typeName idAboutAttr? propAttr* '>' property* '</' typeName '>'
propertyElt	::=	'<' propName '>' value '</' propName '>' | '<' propName resourceAttr '/>'
	...	
propName	::=	Qname
value	::=	description | string
resourceAttr	::=	'resource="' URI-reference '" '
Qname	::=	[NSprefix ': '] name
	...	

Abbildung 9: RDF Syntax (cf. [RDF])

2.3 RDFS

Die Semantik in RDF kommt erst durch Referenz auf ein Schema zustande. Ein Schema ähnelt einem OO-Klassendiagramm bzw. einer Ontologie ohne Axiome, darin definiert sind Klassen, Eigenschaften und Beziehungen[2]. Diese Möglichkeit war in RDF nicht gegeben, weshalb das W3C eine neue Initiative namens *RDF Schema* (*RDFS*) startete, die genau dieses realisiert. Der Kern von RDFS definiert ein Typensystem, einige Klassen (die Wurzel rdfs:Resource, sowie rdfs:Class), Eigenschaften (rdf:type, rdfs:subClassOf, rdfs:subPropertyOf, ...) und schließlich Constraints (rdfs:ConstraintResource, rdfs:ConstraintProperty, ...).

Bemerkenswert ist die zirkuläre Verwendung beider Spezifikationen: Für RDF gibt es ein Schema in RDFS und RDFS ist definiert mit Hilfe von RDF (cf. [RDFS00]). Abbildung 10 zeigt dieses Phänomen und illustriert erneut die graphbasierte Repräsentation von RDF.

[2] „Schema“ und „Ontologie“ werden fortan synonym verwendet.

Ein Statement besteht dabei aus zwei Knoten verbunden mit einer Kante; als Tripel geschrieben würde sich beispielsweise ergeben (rdfs:subClassOf, rdfs:class, rdfs:Resource) für die Aussage, daß rdfs:class eine Subklasse von rdfs:Resource ist. Die Serialisierung per XML-DTD dieses Statements sieht wie folgt aus:

```
<rdf:RDF>
     <rdfs:Class rdf:ID=Class>
      <rdfs:subClassOf
       rdf:resource=http://www.w3.org/2000/01/rdf-schema#Resource/>
     </rdfs:Class>
</rdf:RDF>
```

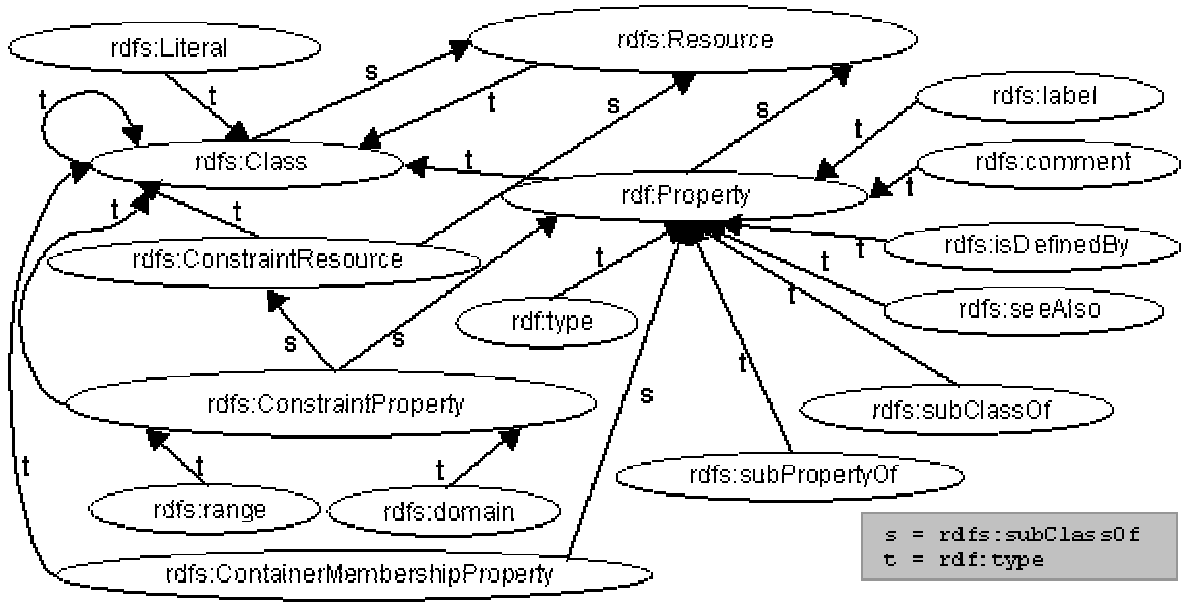


Abbildung 10: RDFS in RDF (cf. [RDFS00])

RDF und RDFS sind Teil einer W3C Initiative namens *Semantic Web*: Metadaten im WWW so angewandt und vernetzt, daß sie für den Rechner verständlich werden und nicht nur der Präsentation für den Menschen dienen; dadurch aber verwendbar für Automation, Integration und Wiederverwendung. Die sogenannte funktionale Architektur des Semantic Web umfaßt drei Schichten: Metdadata, Schema und Logic Layer. RDF ist auf der untersten Ebene einzuordnen – ein einfaches Modell für semantische Zusicherungen, das lediglich auf Ressourcen und deren Eigenschaften operiert. Ein Kandidat für die Schemaebene ist RDFS, welches ein Vokabular zur Definition simpler Ontologien zur Verfügung stellt. Unter einer *Ontologie* versteht man konkrete Modellierungen bestimmter Anwendungsdomänen, auf die sich eine Benutzergruppe geeinigt hat. Eine solche Formalisierung ist sowohl für den Menschen, als auch für die Maschine verständlich und leicht kommunizierbar. Es handelt sich um eine hierarchische Anordnung von relevanten Konzepten, zusammen mit deren Eigenschaften und Beziehungen. Schließlich finden sich auf der Logikebene mächtigere Vokabulare mit reichhaltigeren Modellierungsprimitiven, die meist von Beschreibungslogiken herrühren. Eine Abbildung auf ein Logikkalkül ist deshalb zumeist möglich, was auch einen Inferenzmechanismus erlaubt. Die *DARPA Agent Markup Language* (*DAML*) in Kombination mit der *Ontology Inference Layer* (*OIL*) stellt einen wichtigen Vertreter dar (cf. [CHHGS]).

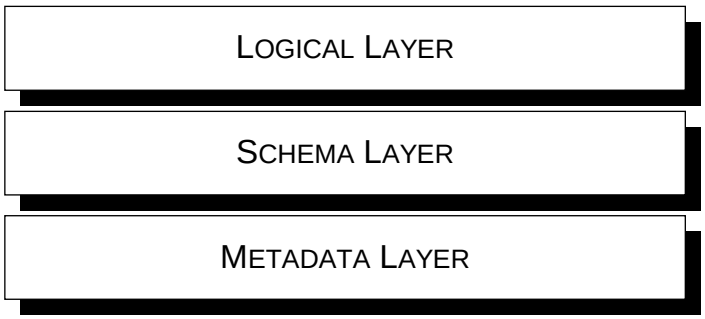


Abbildung 11: Funktionale Architektur des Semantic Web

2.4 RDFS(FA)

Die derzeitige Version von RDFS schießt deutlich am Kardinalziel Maschinenverständlichkeit vorbei. Ursache ist das Fehlen einer formalen Semantik. In der Logik wird eine solche durch Testen auf Enthaltensein in der Extension zu einem Prädikat erreicht. Moderne Ontologiesprachen wie DAML+OIL zählen zu den Beschreibungslogiken und realisieren dieses ebenso. Erschwerend hinzu kommt die unsaubere Abgrenzung von RDFS bezüglich der sogenannten *Metamodeling Architecture*. Diese legt die Definition von Modellierungsprimitiven (Vokabular) fest, die in einer Modellierungssprache verwendet werden können. Um diese Problematik zu verdeutlichen, wird im folgenden zurückgegriffen auf UML, wo die Lösung zu einem sehr ähnlichem Dilemma aus einer strikten Vierteilung besteht (cf. [PH01] et [CAKBC]):

- *Meta-metamodel Schicht*
  Steht an oberster Stelle und dient zum Definieren einer Sprache zur Spezifikation eines Metamodels. Beispiele für Entitäten in dieser Schicht sind *MetaClass* oder *MetaAttribute*.
- *Metamodel* (*Repräsentationsvokabular*)
  Ein Metamodell ist Instanz eines Meta-metamodells und verantwortlich für die Definition einer Sprache zum Spezifizieren eines Benutzermodells. Objekte hier sind etwa *Class* oder *Attribute*.
- *Model* (*Benutzermodell*, *Ontologie*)
  Erst hier wird eine Domäne in gewohnter Art und Weise durch spezielle Klassen wie bsp. *Person* kodiert. Es handelt sich um eine Instanz eines Metamodells.
- *User Objects* (*Instanzen*, *User Data*)
  Konkrete Instanzen der oben erwähnten Klassen finden sich schließlich hier, so würde man z.B. eine modellierte Person „xy“ auf dieser Ebene wiederfinden.

RDFS ist nun anzusiedeln sowohl auf der Meta-metamodel Schicht als auch im Repräsentationsvokabular. Konkrete Klassen in einer Ontologie wären beispielsweise Instanzen von rdfs:Class - gleichzeitig dient rdfs:Class aber auch als Meta-metamodel-Objekt z.B. zum Definieren von rdf:Property, das wiederum zum Metamodel gehört. Damit nicht genug, handelt es sich bei rdfs:Class gar um eine Instanz von rdfs:Resource und um eine Instanz seiner selbst. RDFS nähert sich deshalb empfindlich der Russell’schen Antinomie, wenn man einmal eine Klasse als Menge auffaßt. Diese Antinomie läßt sich gemäß [Mau] auf das Halteproblem zurückführen, was die Enthaltensein-Relation unentscheidbar macht. Das wiederum hat zur Folge, daß es keine vernünftige Abbildung des RDFS-Vokabulares auf eine Logik und damit keine formale Semantik geben kann. Das Testen auf Wahrheit eines Prädikates *p(x)* hat immer eine Prüfung auf Enthaltensein von *x* in der p-Extension zur Folge. In der englischsprachigen Literatur begegnet man all diesen Unzulänglichkeiten häufig unter

der Bezeichnung „Layer Mistake“. RDFS befindet sich derzeit im Stadium einer sogenannten Candidate Recommendation - ein Vorschlag, genannt *RDFS Fixed Architecture* (*RDFS(FA)*), umgeht die o.a. Nachteile durch eine starre vierschichtige Modellierungsarchitektur mit formaler Semantik.

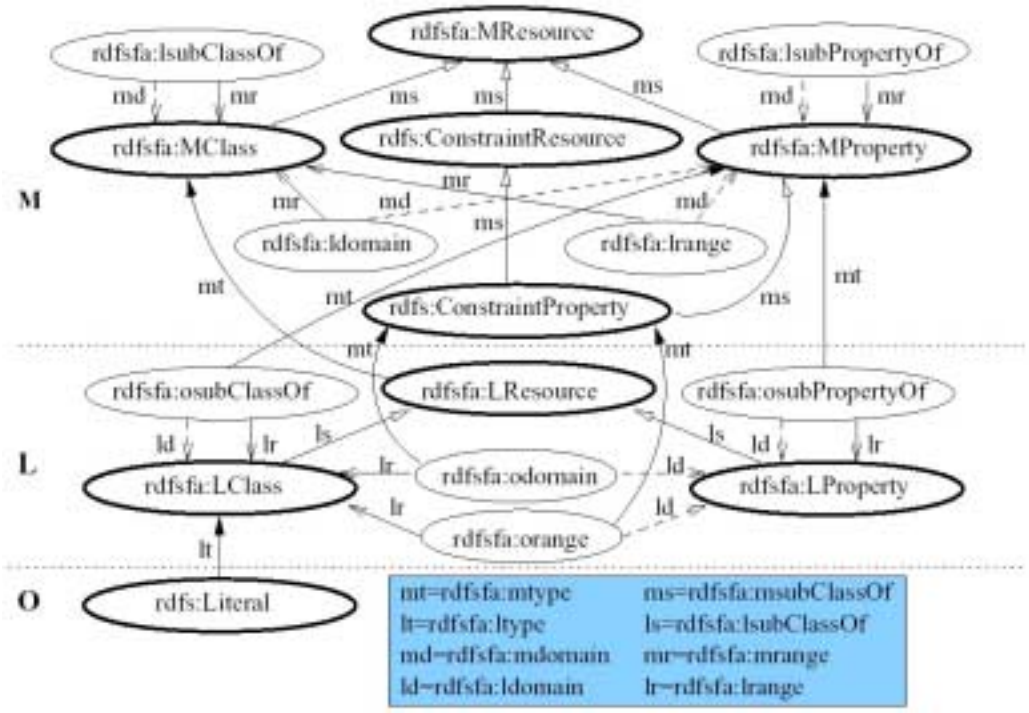


Abbildung 12: RDFS(FA) Überblick (cf. [PH01])

Die Primitiven von RDFS werden soweit notwendig erweitert und auf vier Schichten verteilt, die den o.g. entsprechen. Die oberste und grundlegendste Ebene ist der *Metalanguage Layer* (M) zur Definition des *Language Layer* (L). rdfs:Class spiegelt sich nun sauber getrennt wieder sowohl in M, als auch in L als rdfsfa:MClass bzw. rdfsfa:LClass. Erstere Primitive dient zum Definieren von Ressourcen in L und rdfsfa:LClass zum Definieren von Ressourcen in der nächsttieferen Schicht O (*Ontology Layer*). Erst dort wird eine konkrete Ontologie spezifiziert, um damit eine bestimmte Domäne zu beschreiben. Wie zu erwarten war, findet sich schließlich noch der *Instance Layer* (I), der in Abbildung 12 nicht wiedergegeben ist. Die Primitiven spiegeln sich jeweils mit einem Präfix wider zur Indikation der Schicht. So wird aus rdfs:range sowohl rdfsfa:mrange, als auch rdfsfa:lrange und rdfsfa:orange usf. Eine derart fixierte Modellierungsarchitektur hat neben all den bisherigen geschilderten Vorzügen auch Schwachstellen. Gegenüber dem herkömmlichen RDFS, das beliebig viele Modellierungsebenen zuläßt, ist hier vor allem die Flexibilität zu nennen.

Neuland betritt dieses Dokument bezüglich der Einordnung des RDF-Container-Modells und der Reifikation in RDFS(FA). In [PH01] sind diese Konstrukte nicht erwähnt, weshalb sich eine exakte Modellierung an dieser Stelle findet. Die RDF-Spezifikation (cf. [RDF]) definiert rdfs:Container zusammen mit seinen Subklassen rdf:Bag, rdf:Seq und rdf:Alt. Damit ist man in der Lage auf Instanzenebene eine Menge von Ressourcen zu referenzieren. Abbildung 13 zeigt wie sich diese Primitiven in der Fixed Architecture widerspiegeln. Jeder Container verfügt laut Spezifikation implizit über prinzipiell unendlich viele Relationen rdf:_1, rdf:_2 usw. zur Definition der einzelnen Elemente. Bild dieser Relationen ist das neu hinzugekommene Primitiv rdfsfa:ContainerElement. Das allseits bekannte Beispiel (cf. [RDF]) instantiiert ein Bag-Objekt mit zwei Studenten als Inhalt.

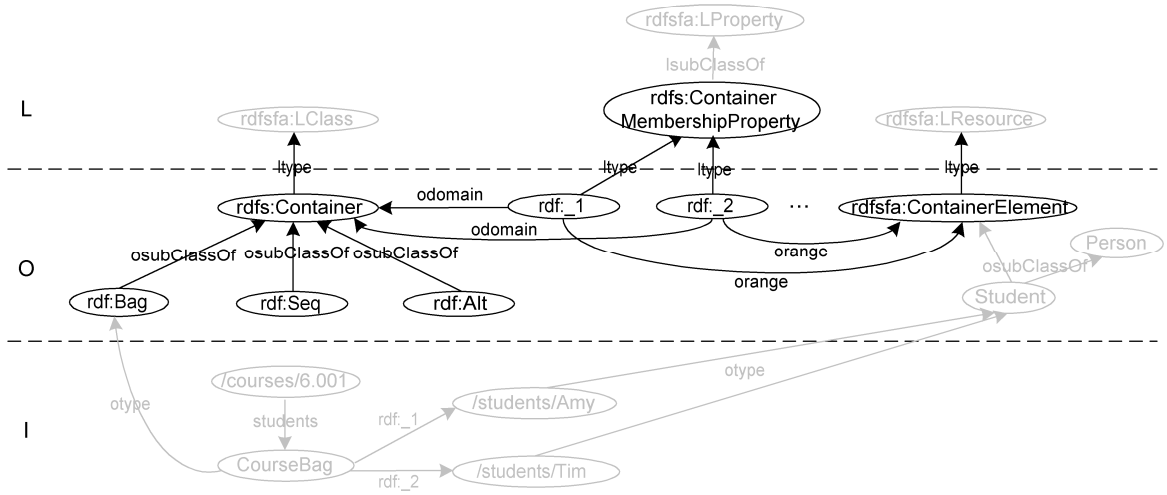


Abbildung 13: Container in RDFS(FA)

Reifikation dient als Mechanismus um Aussagen über Aussagen zu ermöglichen. Das RDF-Vokabular definiert dazu eine Klasse rdf:Statement mitsamt zugehörigen Relationen rdf:subject, rdf:object, rdf:predicate, wobei letztere das zu reifizierende Tripel repräsentieren. Wie sich diese Primitiven in RDFS(FA) verkörpern, ist in Abbildung 14 illustriert. Schließlich erzwingt auch hier eine saubere Modellierung die Einführung von rdfsfa:ReificationElement als Bild der o.g. Relationen. Das Beispiel in Abbildung 14 ist [RDF] entnommen. Sowohl Reifikation, als auch Container-Modell spielen im folgenden keine Rolle, da beide Konstrukte auf der I-Ebene angesiegelt sind. Sichten jedoch, werden ausschließlich auf Schema-, also O-Ebene relevant.

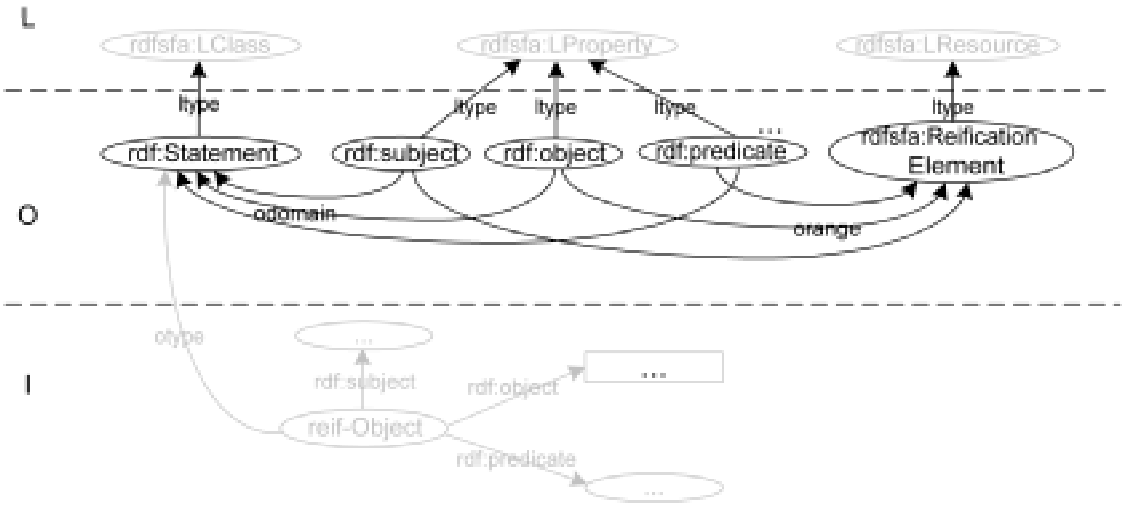


Abbildung 14: Reifikation in RDFS(FA)

Primitive wie rdfs:comment, rdfs:label, rdfs:seeAlso und rdfs:isDefinedBy dienen rein der Dokumentation und tangieren nicht die formale Semantik von RDFS(FA), so daß sie in [PH01] nicht weiter berücksichtigt sind. Korrekterweise müßte man hier ebenfalls aufsplitten in rdfsfa:mcomment, rdfsfa:lcomment und rdfsfa:ocomment; da es sich aber lediglich um Attribute, d.h. Relationen mit rdfs:Literal als Bild handelt, ist eine solche Differenzierung nicht erstrebenswert. Sämtliche in diesem Subkapitel aufgeführten Modellierungskonstrukte werden im folgenden als implizit vorhanden vorausgesetzt.

Im Laufe dieser Arbeit entstanden weitere Formalisierungen von RDFS und damit Alternativen zu RDFS(FA), die nicht weiter berücksichtigt wurden. [WHM01] beispielsweise demonstrieren einen Vorschlag, der auf O-TELOS [Jeu92] basiert und Gebrauch von Datalog macht. Dabei wird nur eine Basisrelation für sämtliche Entitäten des Datenmodells verwendet. [Tol00] zeigt einige Ungereimtheiten und offene Fragen in der RDF(S)-Spezifikation. Der Arbeit entspringt ein validierender Parser, der auf einer Menge von Validierungsregeln basiert. [WR00] schlagen eine Hornlogik-basierte Interpretation vor, die einen Beweis des Entailment Lemmas mit Hilfe von Fixpunkt-Semantik erlaubt. Eine weitere Publikation wählt den Weg zur Formalisierung mit Hilfe von Modelltheorie [Hay01]. Schließlich vergleichen [CK01] die modelltheoretische [Hay01] und die logikbasierte Formalisierung [WR00]. Das Ergebnis bringt ans Licht, daß sich jeweils eine unterschiedliche Semantik ergibt.

2.5 KAON

Um das Semantic Web nun tatsächlich zu operationalisieren, bedarf es einer umfassenden technischen Infrastruktur. *KAON*, die auf Java basierende *Karlsruhe Ontologie und Semantic Web Infrastruktur*, leistet genau dieses (cf. [HMSV01]). Einen ersten Überblick gibt Abbildung 15, indem sie grob die dreigeschichtete Architektur skizziert. Auf dem Client-Layer ist insbesondere das Integrated Developer's Environment zu erwähnen; es umfaßt Werkzeuge zum Ontology Engineering in einheitlicher Art und Weise. So zum Beispiel *KAON-SOEP* (Simple Ontology Editor Plug-In) zur Definition von Ontologien oder *Ontomat* zur semiautomatischen Annotierung von Web-Ressourcen. Daneben finden sich auf dieser Schicht sämtliche Dienstnehmer, die nicht notwendigerweise Java-Applikationen sein müssen.

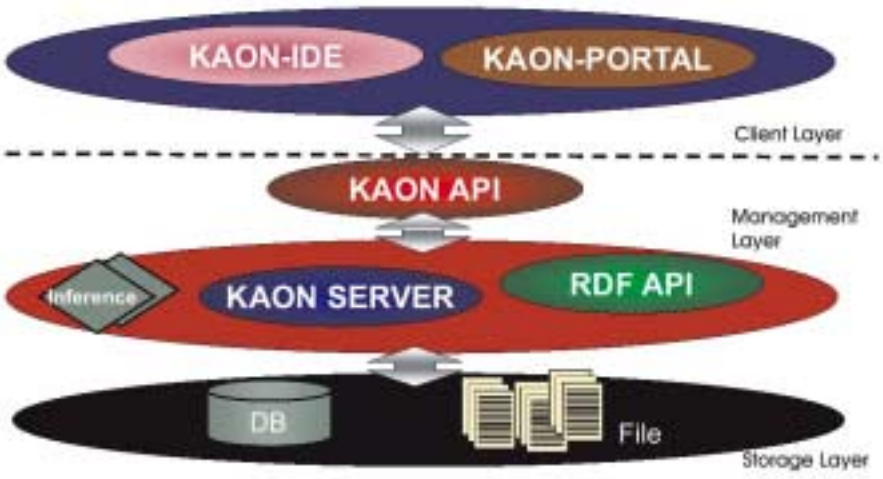


Abbildung 15: KAON Architektur (cf. [HMSV01])

Von herausragender Bedeutung ist das *KAON-API*, die Schnittstelle zum Management Layer. Es ermöglicht einen komfortablen Zugriff auf sämtliche ontologische Ressourcen und definiert dazu eine Hierarchie von Java-Interfaces und abstrakten Klassen, die in Abbildung 16 dargestellt sind. *Entity* dient als Superinterface, davon abgeleitet werden Spezialisierungen wie *NamedEntity*, *HierarchicalEntity* und schließlich *Concept* und *Relation*. Die abstrakte Klasse *Ontology* versteckt schließlich eine komplette Wissensstruktur hinter einer Schnittstelle. Hier kommt nun der Sichtenmechanismus ins Spiel. Ziel der Arbeit wird sein, das KAON-API so zu erweitern, daß ein Dienstnehmer damit auch transparent auf Sichten operieren kann.

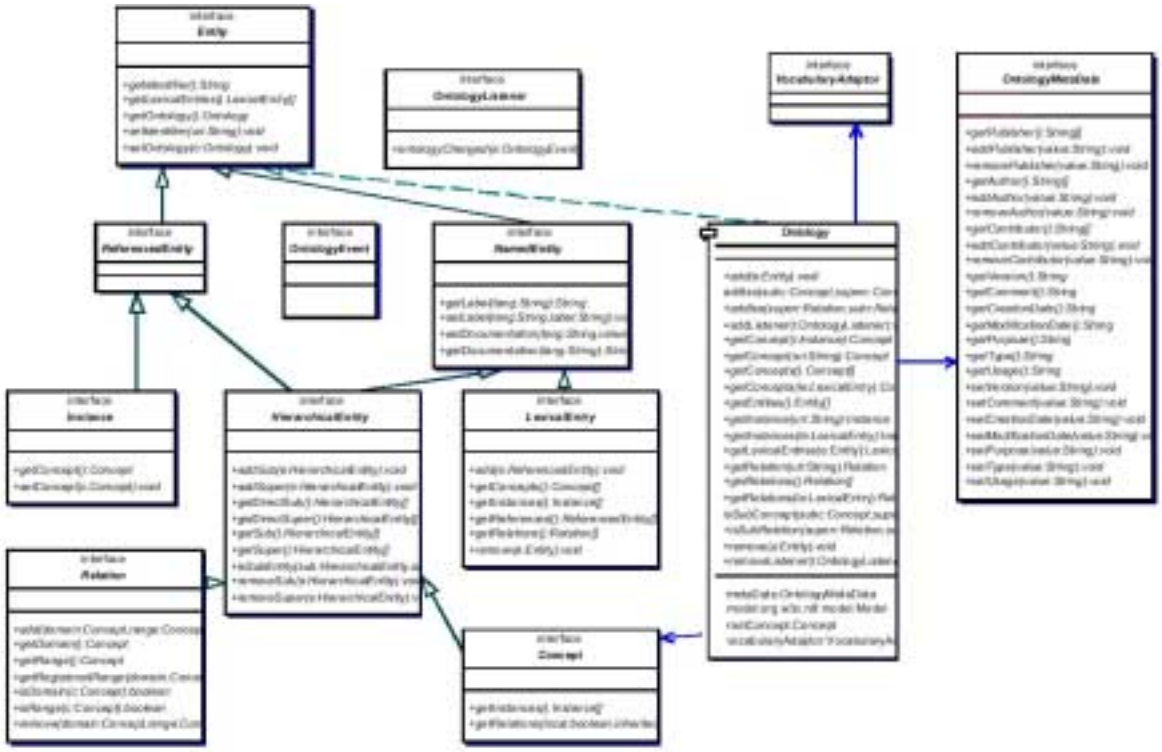


Abbildung 16: KAON-API (cf. [HMSV01])

Zu dieser Interface-Familie existiert eine Implementierung, die sich die Dienste des vom W3C definierten RDF-APIs zunutze macht. Die Standardimplementierung davon arbeitet transient, d.h. sie hantiert im wesentlichen mit Tripeln, die in eine XML-Datei serialisiert oder umkehrt daraus per RDF-Parser gewonnen werden können. Daneben sieht die Architektur eine leistungsfähige persistente Implementierung des RDF-API, genannt *KAON-Server*, vor. Die Anforderungen an ein solches System lauten wie folgt:

- Wie bei jedem Datenhaltungssystem müssen auch hier Persistenz, Nebenläufigkeit und Sicherheit gewährleistet sein.
- Insbesondere in einem hochgradig dynamischen Umfeld wie dem WWW sind Schemaänderungen an der Tagesordnung. Es sollte deshalb die Möglichkeit zur Versionierung bestehen.
- Auch die Option zur Distribution von Metadaten und Kooperation mit anderen Datenhaltungssystemen ist eine Anforderung.
- Sicherheit und Versionierung verlangen nach einem Sichtenmechanismus. Die Realisierung eines solchen ist, wie bereits erwähnt, Ziel dieser Arbeit.

KAON-Server verwendet die Dienste eines J2EE (Java 2 Enterprise Edition) Application Servers, um die Vorteile der Enterprise JavaBeans nutzen zu können. Auf diesem Wege erreicht man eine effiziente und zeitsparende Implementierung, die darüber hinaus mit Netzwerkfähigkeit glänzt. Die Persistenz erledigt KAON-Server mit Hilfe eines objektrelationalen DBMS, damit erklären sich auch die beiden Alternativen des Storage Layers (siehe Abbildung 15). Das KAON-API ermöglicht den Einsatz beliebiger Modellierungsvokabulare. Einige davon, z.B. DAML+OIL, können auf eine formale Logik abgebildet werden. Für diesen Zweck gibt es die Möglichkeit eine passende Inferenzmaschine einzubinden.

3 Sichtendefinitionssprache

In Anlehnung an verwandte Arbeiten objektorientierter Datenbanken spezifiziert dieses Kapitel die Elemente einer Sicht und ein korrespondierendes Vokabular dazu. Wie in den meisten objektorientierten Arbeiten (siehe insbesondere [SAC]), wird eine Sicht später wie ein Schema behandelt werden, jedoch sind an sie gewisse Konsistenzbedingungen geknüpft. Darum kümmert sich Kapitel 4 im besonderen. Die Idee zu virtuellen Konzepten und Attributen, dem Importieren von Ressourcen, sowie implizit dem Verweis auf ein Ursprungsschema stammen aus [AB91], liegen jedoch gleichzeitig auf der Hand, um den Sichtenmechanismus mächtiger zu gestalten. Allerdings divergiert deren Umsetzung im Vergleich zur OO-Welt, da die Situation im Semantic Web eine andere ist (vgl. erneut Kapitel 1.2). Insbesondere im Hinblick auf eine spätere Realisierung inmitten der KAON Infrastruktur müssen die Elemente einer Sicht letztendlich in RDF beschrieben werden können. Neu sind darüber hinaus Filter auf Attributen und Assoziationen sowie virtuelle Assoziationen.

Ein Beispiel verdeutlicht die Elemente einer Sicht und das Vokabular am Ende des Kapitels. Um Mißverständnisse zu vermeiden, definiert die folgende Tabelle die Terminologie vorweg. Einige Begriffe werden erst in Kapitel 4 formal eingeführt.

Begriff	Synonyme	Erläuterung
Schema	Ontologie	Wird definiert in Kapitel 4.2
Subschema	Sicht, Subontologie, View, virtuelles Schema, externe Ontologie	Wird definiert in Kapitel 4.3
Relation	Property	Gemäß UML und OCL der Überbegriff für Attribut und Assoziation.
Attribut	Attribute	Jedes Attribut ist auch Relation mit der Einschränkung, daß das Bild vom Typ rdfs:Literal sein muß.
Assoziation	Association	Jede Assoziation ist auch Relation mit der Einschränkung, daß das Bild auf eine Klasse zeigen muß.
Klasse	Konzept, Concept	Jedes Tripel mit (rdfsfa:ltype, c, rdfsfa:LClass) und Instanzen von Subklassen zu rdfsfa:LClass.
Domäne	Domain	Domäne einer Relation r ist die Klasse c mit (rdfsfa:odomain, r, c).
Bild	Range	Bild einer Relation r ist die Klasse c mit (rdfsfa:orange, r, c)

3.1 Elemente einer Sicht

- *Superontologie*
  Wichtigstes Element einer Sicht ist der Verweis auf ihr Ursprungsschema. In einer ersten Lösung des Sichtenmechanismus wird pro Sicht nur ein solcher Verweis möglich sein. Das Ursprungsschema darf wiederum eine Sicht sein, so daß eine Kaskade entstehen kann.

- *Importierte Ressourcen*
  Vom Ursprungsschema können Ressourcen nach Belieben importiert werden. Darunter Konzepte, Attribute und Assoziationen, auch jeweils virtuelle, wenn die Ursprungsontologie wieder eine Sicht ist. Dasselbe gilt nicht für Attribut- und Assoziationsfilter (s.u.). Nur importierte Ressourcen erscheinen in der Sicht.

- *Filter*
  Eine Sicht kann über Attribut- und Assoziationsfilter verfügen. Wie der Name bereits vermuten läßt, sind erstere auf einem Attribut definiert und dezimieren die Extension einer Klasse entsprechend. Assoziationsfilter belassen nur solche Instanzen sichtbar, die auch Beziehung stehen. Eine ähnliche Funktionalität kann man erreichen durch virtuelle Konzepte (s.u.). Filter sind hier eine flexible Alternative, wenn die Klassenhierarchie unberührt bleiben soll.

- *Virtuelle Attribute*
  Neben oben geschilderten Mechanismen soll auch die Möglichkeit zur Definition virtueller Attribute bestehen. Ein Ontology Engineer ist damit in der Lage Attribute zu spezifizieren für die es keine korrespondierende Instanzen gibt und deren Wert sich dynamisch berechnet.

- *Virtuelle Assoziationen*
  Auch virtuelle Assoziationen sind in dieser Lösung vorgesehen. Solche befinden sich zwischen zwei Klassen wie auch herkömmliche Assoziationen. Hinzu kommt allerdings eine Spezifikation der Extension per Query-Ausdruck, die beschreibt, wie Instanzen in Beziehung stehen sollen.

- *Virtuelle Konzepte*
  Eine sehr mächtige Funktionalität zur individuellen Anpassung der Klassenhierarchie ist die Definition virtueller Konzepte. Insbesondere bei unüberschaubar großen Ontologien mit unzähligen Konzepten mag es sinnvoll sein weitere Klassen zur besseren Strukturierung in die Hierarchie einzufügen. Dieses kann geschehen durch Selektion, Schnitt, Vereinigung und Differenz auf vorhandenen Klassen, welche wiederum virtuell sein können. Auf der anderen Seite soll aber auch die Möglichkeit zur arbiträren Definition bestehen, also neue Klassen, die ihre Extension nach Belieben bestimmen.

3.2 Vokabular

Für die KAON Infrastruktur legt dieses Kapitel ein Vokabular fest, auf dem das weitere Vorgehen beruht. Prinzipiell geschieht nichts anderes als die formale Umsetzung der oben aufgeführten Elemente in RDF-Primitive. Dieses ist eine Notwendigkeit, da das KAON-API auf RDF basiert. Als Metamodellierungsprache kommt dabei RDFS(FA) zum Einsatz. Warum, ist an dieser Stelle nicht ersichtlich, wird aber spätestens in Kapitel 4.2 deutlich werden. Das KAON-API wird die Primitive später interpretieren und die Elemente einer Sicht entsprechend operationalisieren (siehe Kapitel 5).

Allen voran steht die Primitive KAON:View, anhand deren festgemacht wird, daß es sich um eine Sicht handelt. Sie dient dazu, um auf RDF-Ebene zwischen Schema und Subschema differenzieren zu können. (KAON:View, <*name*>, <*ontologie*>) definiert die Bezeichnung

der Sicht und auch gleichzeitig den Verweis auf die Ursprungsontologie. Sollen später mehrere solche zulässig sein, vervielfacht sich die Anzahl der Statements entsprechend (KAON:View, <*name*>, <*ontologie*$_1$>), (KAON:View, <*name*>, <*ontologie*$_2$>) usw.

Konzepte und Relationen, die aus der Ursprungsontologie den Weg in die Sicht finden, werden dort nicht definiert, sondern importiert. Dementsprechend steht in einer Sichtendefinition KAON:importClass, KAON:importProperty statt rdfsfa:ltype. Im Bild solcher Statements steht jeweils, aus welcher Superontologie die Ressource stammt. Abbildung 17 zeigt die Einordnung der Primitive in RDFS(FA).

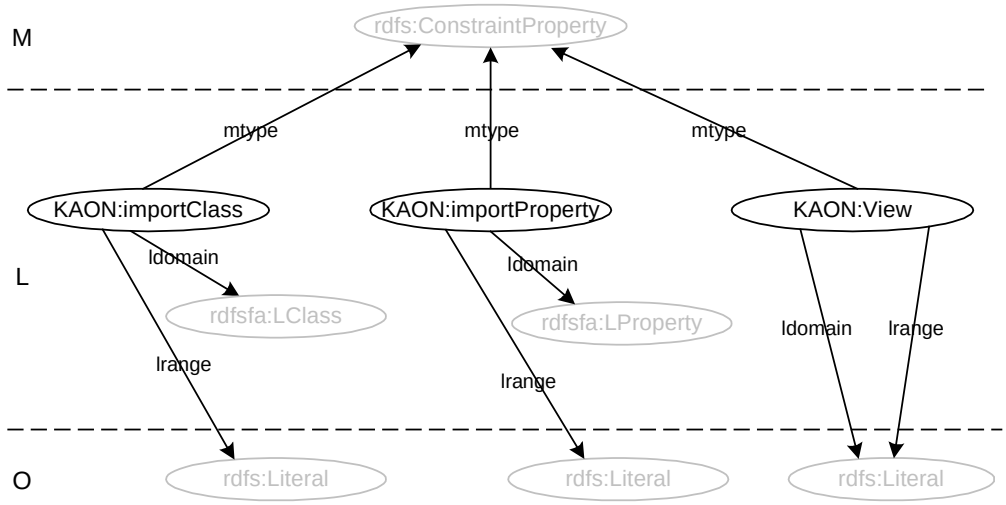


Abbildung 17: KAON Vokabular

Die oben angesprochene Filterfunktionalität spiegelt sich in KAON:Filter und seinen Subklassen wider. Eine Instanz von KAON:Attributefilter ist auf der O-Schicht einzuordnen mit der Domäne auf einer Relation und Literalen als Bild. Es mag zunächst verwirrend erscheinen, daß sich Instanzen von rdfs:Literal auf der Schicht O befinden. Allerdings beinhalten diese Ausdrücke, die einer gewissen Syntax gehorchen, die eigentliche Filterinformation und sind damit korrekterweise auf der Schemaebene einzuordnen. Das KAON-API erkennt später das Vorhandensein solcher Filter und berücksichtigt diese jeweils. Bei einer Anfrage muß dann das Bild des Filters für ein Tripel erfüllt sein, damit es im Ergebnis erscheint. Für Assoziationen existiert ebenfalls ein Filter. Er stellt sicher, daß später nur solche Tripel zugänglich sind, die miteinander in Relation stehen.

Virtuelle Attribute und Assoziationen sind Subklassen zu KAON:virtualProperty, welches seinerseits eine Spezialisierung von rdfsfa:LProperty darstellt. Die Domäne zeigt bei beiden wie gewöhnlich auf ein Konzept. KAON:virtualAttribute besitzt ein Literal stets rdfs:Literal. Die Berechnungsvorschrift befindet sich im Objekt eines KAON:Extension-Statements. KAON:virtualAssociation steht immer zwischen zwei Konzepten und muß ebenfalls über eine KAON:Extension-Kante verfügen, die schließlich in Beziehung stehende Instanzen spezifiziert.

KAON:virtualClass als Subklasse von rdfsfa:LClass, zusammen mit KAON:Extension, sind die Primitive für virtuelle Konzepte. Ähnlich zu virtuellen Relationen, wird auch hier die Population im Objekt eines KAON:Extension-Statements bestimmt. Der Ontology Engineer kann von anderen Ressourcen selektieren, vereinigen, subtrahieren, schneiden oder auch beliebige Extensionen spezifizieren. Zumindest im Fall der Selektion, sind Filter eine Alternative zur Definition virtueller Klassen.

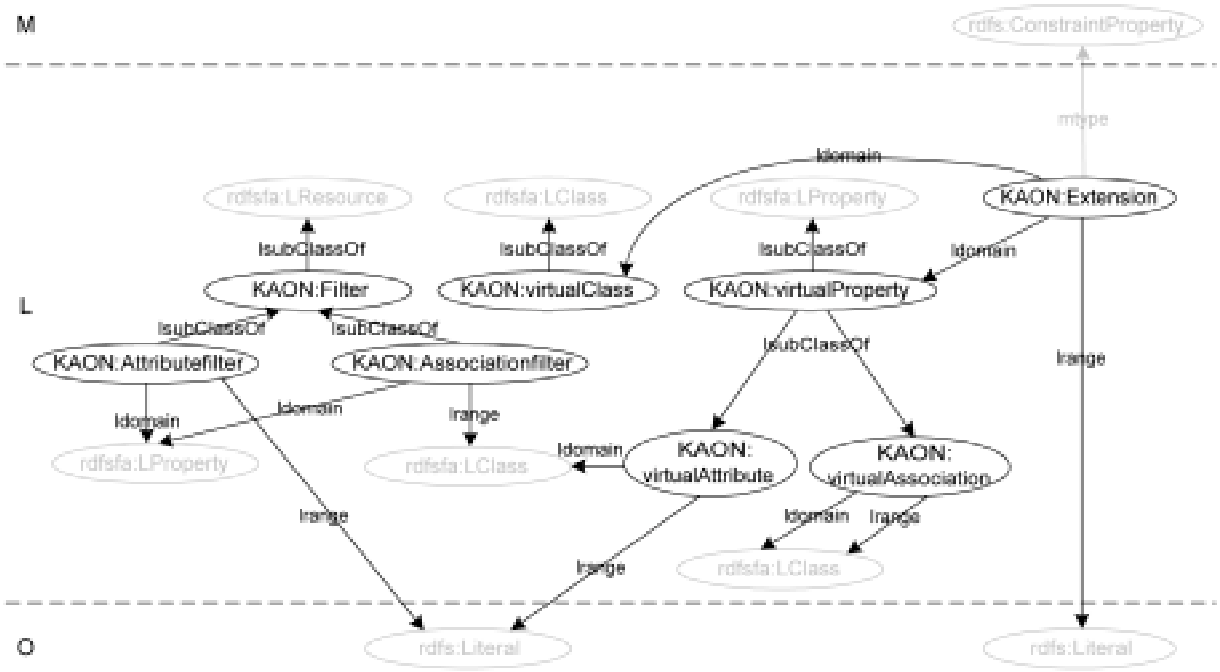


Abbildung 18: KAON Vokabular

3.3 Sichtendefinition per Textdatei

Zur einfacheren Handhabung soll es dem Ontology Engineer später möglich sein, eine Sicht komfortabel als ASCII-Datei zu spezifzieren. Die Syntax dieser Sprache zeigt die Tabelle unten in der linken Spalte. Die triviale Umsetzung in RDF-Statements mit dem Vokabular aus 3.2 ist jeweils in der rechten Spalte gezeigt. Kardinalziel ist jedoch die grafische Definition einer Sicht, die hier nicht weiter besprochen wird.

CREATE SUBONTOLOGY <*URI*>	
BASE <*MODEL_URI*> {BASE <*MODEL_URI*>}	(<*URI*>, KAON:View, <*MODEL_URI*>) (<*URI*>, KAON:View, <*MODEL_URI*>)
{IMPORT CLASS <*URI*>}	(<*URI*>, KAON:importClass, <*MODEL_URI*>), ...
{CREATE VIRTUAL CLASS <*VIEW_URI*> ON <*URI*> SELECTION '<*query*>' }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualClass) (<*VIEW_URI*>, rdfsfa:osubClassOf, <*URI*>), ... (<*VIEW_URI*>, KAON:Extension, <*query*>)
{CREATE VIRTUAL CLASS <*VIEW_URI*> ON <*URI*> DIFFERENCE <*DIFF_URI*> {DIFFERENCE <*DIFF_URI*> } }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualClass) (<*VIEW_URI*>, rdfsfa:osubClassOf, <*URI*>) (<*VIEW_URI*>, KAON:Extension, <*query*>)
{CREATE VIRTUAL CLASS <*VIEW_URI*> ON <*URI*> UNION <*UNION_URI*> {UNION <*UNION_URI*> } }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualClass) (<*URI*>, rdfsfa:osubClassOf, <*VIEW_URI*>) (<*UNION_URI*>, rdfsfa:osubClassOf, <*VIEW_URI*>),... (<*VIEW_URI*>, KAON:Extension, <*query*>)

{CREATE VIRTUAL CLASS <*VIEW_URI*> ON <*URI*> INTERSECT <*ISECT_URI*> {INTERSECT <*ISECT_URI*>} }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualClass) (<*VIEW_URI*>, rdfsfa:osubClassOf, <*URI*>) (<*VIEW_URI*>, rdfsfa:osubClassOf, <*ISECT_URI*>), ... (<*VIEW_URI*>, KAON:Extension, <*query*>)
{CREATE VIRTUAL CLASS <*VIEW_URI*> SUBCLASS OF <*URI*> USE '<*query*>' }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualClass) (<*VIEW_URI*>, rdfsfa:osubClassOf, <*URI*>) (<*VIEW_URI*>, KAON:Extension, <*query*>)
{IMPORT PROPERTY <*URI*>}	(<*URI*>, KAON:importProperty, <*MODEL_URI*>), ...
{CREATE VIRTUAL PROPERTY <*VIEW_URI*> SET DOMAIN <*DOMAIN_URI*> {SET DOMAIN <*DOMAIN_URI*>} SET RANGE <*RANGE_URI*> USE '<*query*>' }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualAttribute) / (<*VIEW_URI*>, rdfsfa:ltype, KAON:virtualAssociation) (<*VIEW_URI*>, rdfsfa:odomain, <*DOMAIN_URI*>) (<*VIEW_URI*>, rdfsfa:odomain, <*DOMAIN_URI*>) (<*VIEW_URI*>, rdfsfa:orange, <*RANGE_URI*>) / (<*VIEW_URI*>, rdfsfa:orange, rdfs:Literal) (<*VIEW_URI*>, KAON:Extension, <*query*>)
{CREATE ATTRIBUTEFILTER <*VIEW_URI*> ON <*ATTR_URI*> USE '<*query*>'}	(<*VIEW_URI*>, rdfsfa:ltype, KAON:Attributefilter) (<*VIEW_URI*>, rdfsfa:odomain, <*ATTR_URI*>) (<*VIEW_URI*>, rdfsfa:orange, <*query*>)
{CREATE ASSOCIATIONFILTER <*VIEW_URI*> ON <*ASSOCIATION_URI*> TARGET <*CLASS_URI*> }	(<*VIEW_URI*>, rdfsfa:ltype, KAON:Associationfilter) (<*VIEW_URI*>, rdfsfa:odomain,<*ASSOCIATIONURI*>) (<*VIEW_URI*>, rdfsfa:orange, <*CLASS_URI*>)

3.4 Beispiel

Dieser Abschnitt exemplarisiert die oben genannten Elemente am Beispiel aus Kapitel 0. Zunächst existiert die Ursprungsontologie gemäß Abbildung 1, deren Konzepthierarchie unten in RDFS(FA) Schicht O dargestellt ist. Die Kanten verkörpern die osubClassOf-Relation; ltype-Relationen sind aus Gründen der Übersichtlichkeit nicht eingezeichnet.

Im folgenden soll nun die Sicht 2 „Web-Shopping für Endkunden“ extrahiert werden. Ein Kunde möchte per WWW Endprodukte einkaufen und eventuell seine alten Rechnungen begutachten. Zu übernehmende Konzepte sind demnach Kundenrechnung, Kunde und Endprodukt. Folgende RDF-Statements ergeben sich dazu, zugunsten der Lesbarkeit wird auf die korrekte Darstellung von URIs und Namespaces verzichtet:

(KAON:View, “Web-Shopping für Endkunden“, “Unternehmensontologie“)
(KAON:importClass, Kundenrechnung, “Unternehmensontologie“)
(KAON:importClass, Kunde, “Unternehmensontologie“)
(KAON:importClass, Endprodukt, “Unternehmensontologie“)

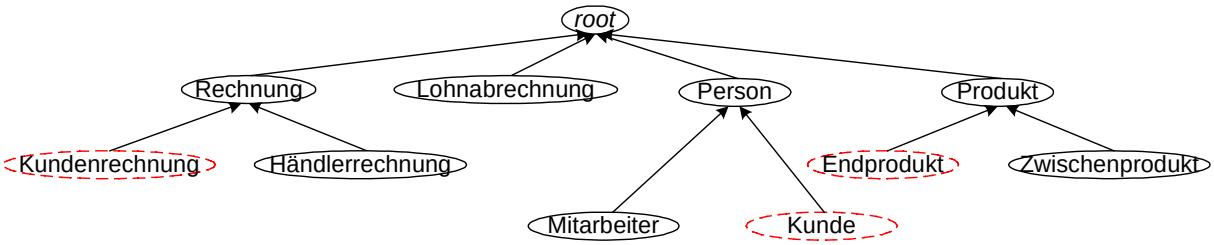


Abbildung 19: Beispielontologie in RDFS(FA)

Die Selektion von Relationen erfolgt unabhängig davon. Relevant sind Vorname, Nachname, Alter, Nummer, Datum, Lieferanschrift, Rechnungsanschrift, Debitor und KR_Position. Das führt zu u.a. Statements. Fehlt in Abbildung 20 die range-Kante, so ist dieses zugunsten der Übersichtlichkeit geschehen und gleichbedeutend mit rdfs:Literal als Bild. Gestrichelte Knoten deuten an, welche Relationen später den Weg in die Sicht finden sollen. Des weiteren wird visuell differenziert zwischen Klassen (dünn umrandet) und Relationen (fett umrandet).

(KAON:importProperty, Vorname, “Unternehmensontologie“)
(KAON:importProperty, Nachname, “Unternehmensontologie“)
(KAON:importProperty, Alter, “Unternehmensontologie“)
(KAON:importProperty, Nummer, “Unternehmensontologie“)
(KAON:importProperty, Datum, “Unternehmensontologie“)
(KAON:importProperty, Lieferanschrift, “Unternehmensontologie“)
(KAON:importProperty, Rechnungsanschrift, “Unternehmensontologie“)
(KAON:importProperty, Debitor, “Unternehmensontologie“)
(KAON:importProperty, KR_Position, “Unternehmensontologie“)

Zur Einschränkung der zugreifbaren Kunden dient ein Filter auf deren Alter. Der sogenannte Alterfilter ist Instanz der Klasse KAON:Attributefilter, hat als Domäne Alter und als Bild einen Ausdruck. Das System wird diesen später berücksichtigen und dementsprechend nur Kunden liefern, die älter als 18 Jahre sind. Außerdem sollen nur solche Kunden in der Subontologie erscheinen, die überhaupt als solche registriert sind. Als Kriterium dazu dient die Existenz einer Debitor-Relation, d.h. der Kunde hat mindestens eine Rechnung vorliegen. Eine Instanz von KAON:Associationfilter namens Debitorfilter beschreibt genau dieses.

(rdfsfa:ltype, Alterfilter, KAON:Attributefilter)
(rdfsfa:odomain, Alterfilter, Alter)
(rdfsfa:orange, Alterfilter, <Ausdruck>)

(rdfsfa:ltype, Debitorfilter, KAON:Associationfilter)
(rdfsfa:odomain, Debitorfilter, Debitor)
(rdfsfa:orange, Debitorfilter, Kunde)

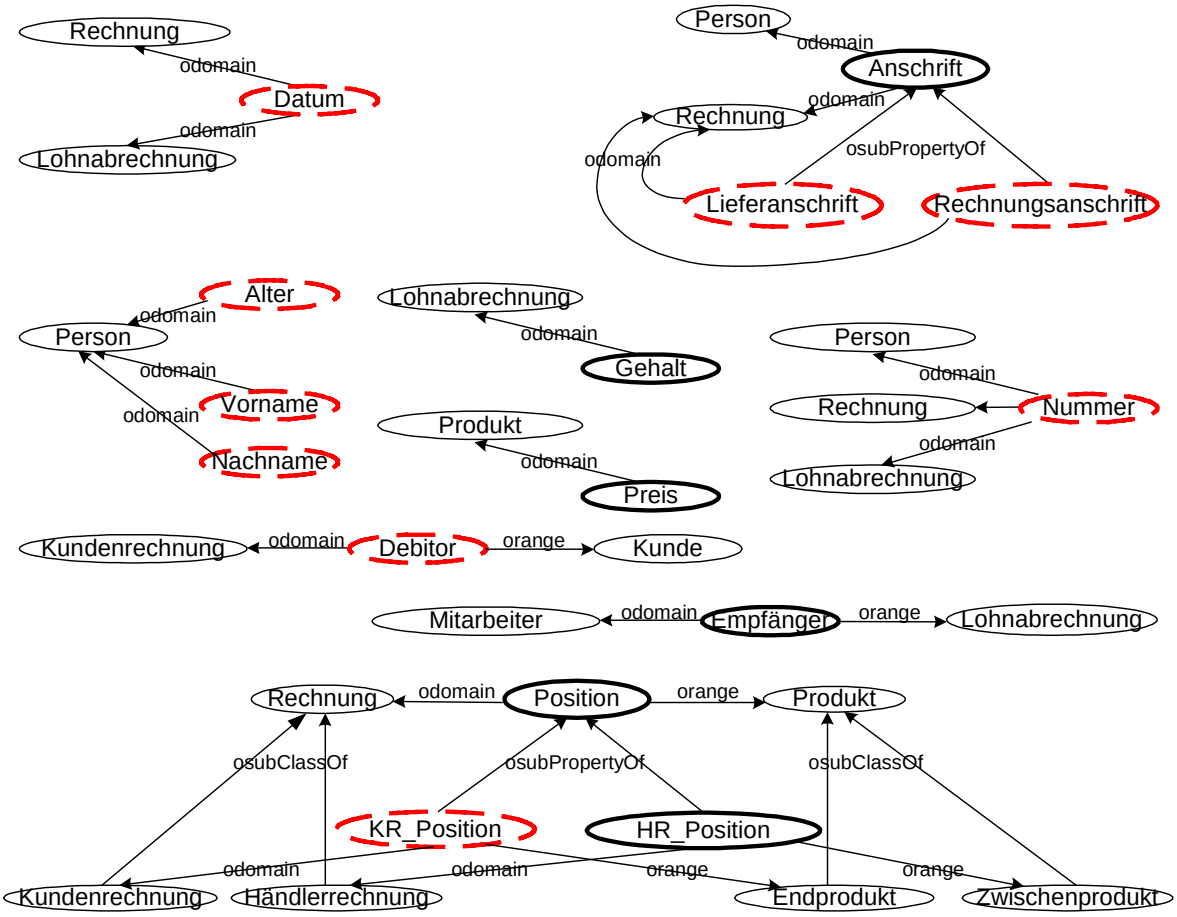


Abbildung 20: Relationenwald

Hinzukommen soll ebenfalls ein virtuelles Attribut namens “Euro“. Es ist Instanz von KAON:virtualAttribute, hat als Domäne Endkunde und als Bild Literale, die sich gemäß dem Query-Ausdruck im Extension-Statement berechnen. Aus vorhandenen Attributwerten “Preis“, läßt sich per einfacher Multiplikation der jeweilige Eurobetrag berechnen.

(rdfsfa:ltype, Euro, KAON:virtualAttribute)
(rdfsfa:odomain, Euro, Endprodukt)
(rdfsfa:orange, Euro, rdfs:Literal)
(KAON:Extension, Euro, “<Query-Ausdruck>“)

Drei virtuelle Klassen Premium, Gold und Normal verfeinern Kunden gemäß ihren bisherigen Einkäufen. Die Extension von Premiumkunden enthält nur solche, die in den letzten beiden Jahren Waren im Wert von mindestens 10000,- Mark gekauft haben. Gold- und Normalkunden schränken ein auf Einkäufe zwischen 5000,- und 10000,- bzw. kleiner 5000,- DM. Das im Vokabular definierte Attribut KAON:Extension enthält als Wert jeweils entsprechende Query-Ausdrücke. Unten abgebildet sind die Statements dazu, sowie ein gesonderter Exkurs zur Demonstration virtueller Assoziationen.

(rdfsfa:ltype, Premium, KAON:virtualClass)
(KAON:Extension, Premium, “<Query-Ausdruck>“)
(rdfsfa:ltype, Gold, KAON:virtualClass)
(KAON:Extension, Gold, “<Query-Ausdruck>“)
(rdfsfa:ltype, Normal, KAON:virtualClass)
(KAON:Extension, Normal, “<Query-Ausdruck>“)

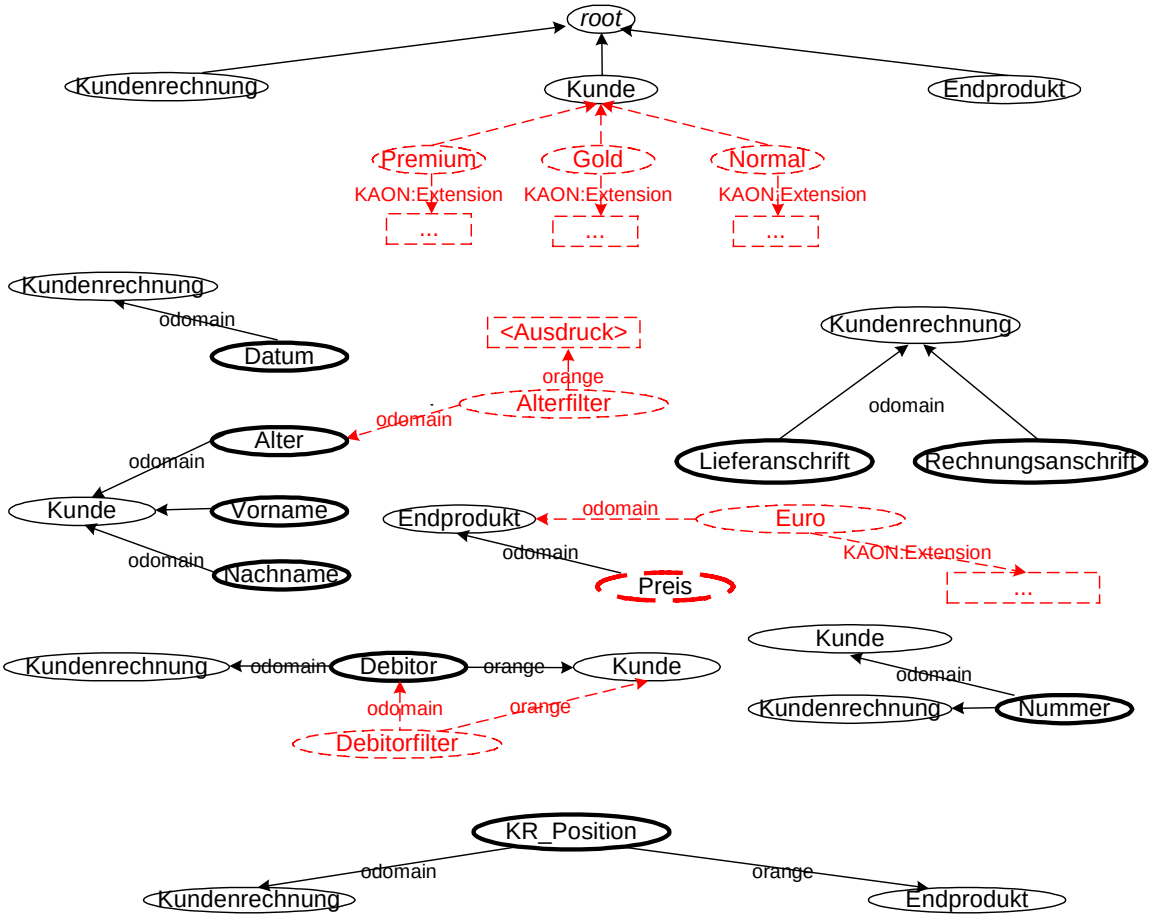
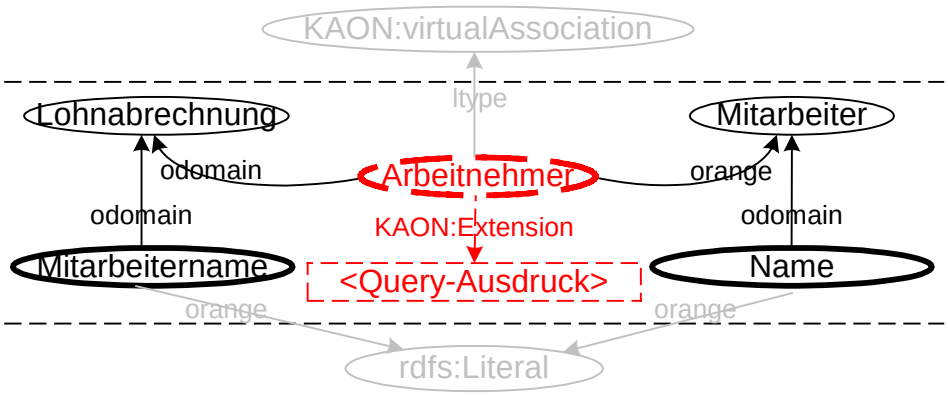


Abbildung 21: Sicht 2

Exkurs:

Angenommen, die ursprüngliche Datenmodellierung berücksichtigte keine Relation zwischen Lohnabrechnung und Mitarbeiter, so kann dieses nun zum Ausdruck gebracht werden. Eine virtuelle Assoziation namens Arbeitnehmer verknüpft beide durch eine Query, die die Gleichheit der beiden Zeichenketten Mitarbeitername und Name fordert.



(rdfsfa:ltype, Arbeitnehmer, KAON:virtualAssociation)
(rdfsfa:odomain, Arbeitnehmer, Lohnabrechnung)
(rdfsfa:orange, Arbeitnehmer, Mitarbeiter)
(KAON:Extension, Arbeitnehmer, “<Query-Ausdruck>“)

Man beachte, daß Konzepthierarchie, Relationenhierarchie, sowie Domänen und Bilder importierter Relationen später automatisch inferiert werden. Die Formalisierung dazu findet sich im nächsten Kapitel. Nichtsdestotrotz sind derartige Kanten in den oben aufgeführten Beispielen jeweils mit notiert um das Verständnis an dieser Stelle zu erleichtern. Der Problematik zu Query-Ausdrücken bzw. den zugrundeliegenden Sprachen widmet sich das Kapitel 5.3.

4 Konsistenzaxiomatik

Dieses Kapitel formalisiert den Begriff der Ontologie, der Subontologie, sowie Konsistenzbedingungen und Regeln zur Inferierung von Klassenhierarchie, Relationenhierarchie, Domänen und Bildern. Ein Exkurs weiter unten wird aufzeigen, daß die Verwendung von RDFS(FA) als Metamodellierungssprache eine Notwendigkeit ist, um überhaupt vernünftige Ergebnisse erzielen zu können.

Die Axiome bewegen sich ausschließlich auf der O-Ebene der Fixed Architecture, d.h. sie kümmern sich um korrekte Schema- und Subschema-Definitionen. Probleme wie „Ist der Wert eines Attributs Instanz der Domäne der Attributdefinition?“ interessieren hier nicht. Die Prüfung auf Validität einer Faktenbasis bezüglich seines Schemas ist Aufgabe eines Parsers wie beispielsweise VRP (cf. [Tol00]). Die Anforderung „Faktenbasis“ aus Kapitel 1.3 verbietet außerdem jedwede persistente Replikation von Instanzen. D.h. nur die Ursprungsontologie, bei Sichtenkaskaden die jeweils oberste Ontologie der Kette, wird später über Instanzen verfügen. Dabei garantieren die unten eruierten Axiome, daß Sicht und Fakten bezüglich des ursprünglichen Schemas konsistent bleiben.

Als Formalismus verwendet die folgende Axiomatisierung das formale Modell für RDF, vorgestellt in Abbildung 7. Wie bereits mehrfach erwähnt, beruht das KAON-API auf RDF, so daß dieser Schritt im Hinblick auf eine spätere Realisierung unabdingbar wird. Alternativen zum formalen Modell wären XML-Serialisierungen oder das Graphmodell von RDF gewesen, beide sind für diesen Zweck freilich ungeeignet. XML-Serialisierungen sind nicht eindeutig und das Graphmodell dient lediglich der Visualisierung.

Ein Axiom besteht aus einem oder mehreren Tripeln (subject, predicate, object) $\in$ *Statements* je nach Bedarf ausgestattet mit Allquantor, Existenzquantor sowie Äquivalenz, Implikation usw. Diese logischen Junktoren werden wie im Sinne der Mathematik verwendet. Zugunsten eines vernünftigen Software-Engineering hat man damit eine abstrakte Beschreibung geschaffen, unabhängig von Entwurfs- und Implementierungsfragen. Erst in Kapitel 5 wird die Entscheidung fallen, wie die Axiome tatsächlich zu prüfen sind.

In Kapitel 3.2 wurde das vorgestellte Vokabular nicht nur aus Gründen der Übersichtlichkeit getrennt dargestellt. Die in 4.3 vorgestellten Konsistenzbedingungen operieren nur auf Primitiven in Abbildung 18. KAON:importClass und KAON:importProperty sind lediglich für Persistenz, also beispielsweise für XML-Serialisierungen, relevant und werden vor der Prüfung auf Konsistenz in rdfsfa:ltype umgesetzt. KAON:View betrifft ebenfalls nicht die Axiomatik, sondern wird erst bei der Umsetzung interessant (siehe Kapitel 5.1.3 und 5.2.2).

Zunächst finden sich einige Hilfsfunktionen, von welchen die Axiome danach ausführlichst Gebrauch machen. Ihren Sinn also im vornherein verstehen zu wollen ist nicht ratsam. Kapitel 4.2 spezifiziert, was man überhaupt unter einem Schema bzw. einer Ontologie in RDFS(FA) versteht. Erst mit dieser Definition kann die Lösung des eigentlichen Problems, nämlich die Festlegung einer Sicht auf einer Ontologie, beginnen (Kapitel 4.3). Das beinhaltet präzise Definitionen der bereits vorgestellten virtuellen Ressourcen sowie Konsistenzbedingungen dazu.

4.1 Hilfsfunktionen

Die folgenden Funktionen verwenden u.a. die Mengen $\boldsymbol{C}$, $\boldsymbol{C'}$, $\boldsymbol{P}$ und $\boldsymbol{P'}$, die allesamt erst in den nächsten Kapiteln eingeführt werden. Um dennoch das Verständnis zu fördern, sei hier kurz auf deren Bedeutung eingegangen. $\boldsymbol{C}$ bezeichnet die Menge aller Konzepte in einer Ontologie und $\boldsymbol{C'}$ die in eine Sicht zu übernehmenden. Ähnliches gilt für $\boldsymbol{P}$ und $\boldsymbol{P'}$, nur daß darin jeweils Properties, also Relationen, enthalten sind. d_S, d_P und d_O bezeichnen jeweils Subjekt, Prädikat und Objekt eines Tripels $d \in$ *Statements*.

SC*: $\boldsymbol{C} \rightarrow 2^{\boldsymbol{C}}$ und SC*: $\boldsymbol{C'} \rightarrow 2^{\boldsymbol{C'}}$

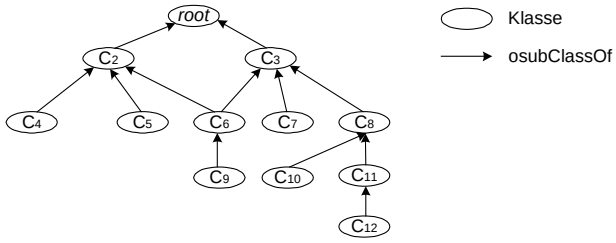
Alle Subkonzepte zu einem Konzept

SC* (für SubConcepts) resultiert in allen Subkonzepten zum Argument. Dazu benötigt werden Hilfsfunktionen SC_i: $2^{\boldsymbol{C}} \rightarrow 2^{\boldsymbol{C}}$, welche jeweils die unmittelbaren Subklassen liefern. Die Definitionen lauten wie folgt:

$SC_0(\{c\}) = \{c\}$ mit c=(rdfsfa:ltype, y, rdfsfa:LClass) $\in \boldsymbol{C}$

$SC_i(SC_{i-1}(...)) = \{ c \mid c=(\text{rdfsfa:ltype, y, rdfsfa:LClass}) \in \boldsymbol{C} \wedge (\text{rdfsfa:osubClassOf, y, z}) \in \boldsymbol{C} \wedge (\text{rdfsfa:ltype, z, rdfsfa:LClass}) \in SC_{i-1}(...)\}$

$$SC^*(c) = \bigcup_i SC_i(SC_{i-1}(...SC_0(\{c\})...)) \setminus \{c\}$$



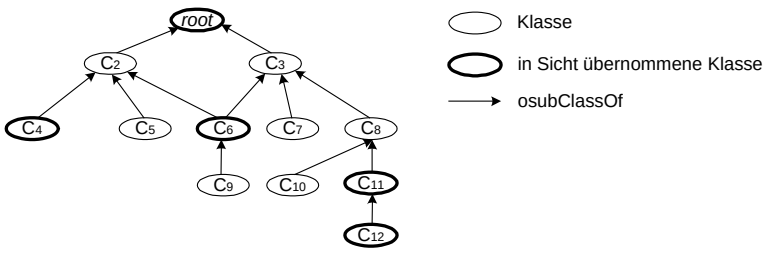
Ein kleines Beispiel soll SC* verdeutlichen: Bei oben gegebener Klassenhierarchie würde SC*(*root*) einfach die Menge {C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11}, C_{12}} liefern. *root* und C_i stehen hier und im folgenden als Abkürzung für Tripel (rdfsfa:ltype, C_i, rdfsfa:LClass). SC*: $\boldsymbol{C'} \rightarrow 2^{\boldsymbol{C'}}$ arbeitet in gleicher Weise auf der Klassenhierarchie in einer Sicht.

SC^0: $\boldsymbol{C} \rightarrow 2^{\boldsymbol{C}}$ und SC^0: $\boldsymbol{C'} \rightarrow 2^{\boldsymbol{C'}}$

Übernommene Subkonzepte zu einem Konzept

Diese Funktion liefert ausgehend von einer Klasse c nur die Subklassen, welche auch in die Sicht übernommen wurden. Nicht dabei sind deren Subklassen, Subsubklassen etc. Zunächst sammelt H:$2^{\boldsymbol{C}} \rightarrow 2^{\boldsymbol{C}}$ alle Subklassen per SC*(c) und selektiert davon die in der Sicht enthaltenen, einschließlich den tieferliegenden. SC^0 soll aber nur die unmittelbaren Subklassen liefern und subtrahiert deshalb H(H(c)).

$H(Cl) = \{ sc \mid x \in Cl \wedge sc \in SC^*(x) \wedge sc \in \boldsymbol{C'}\}$ mit $Cl \subseteq \boldsymbol{C}$
$SC^0(c) = H(\{c\}) \setminus H(H(\{c\}))$

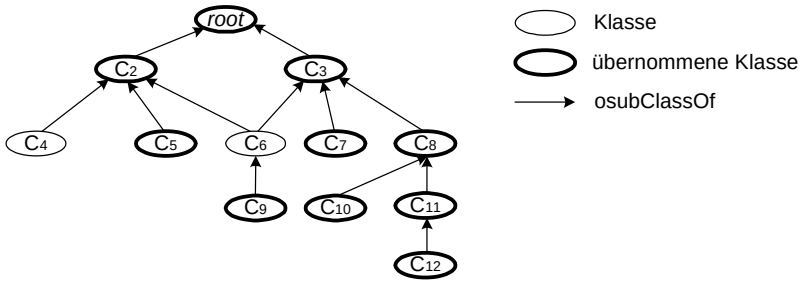


SC^0(*root*) würde in obigem Beispiel also die Menge {C_4, C_6, C_{11}} liefern, H({*root*}) = {C_4, C_6, C_{11}, C_{12}} und H(H({*root*})) = H({C_4, C_6, C_{11}, C_{12}}) = {C_{12}}. Ein Axiom wird später SC^0 nutzen um osubClassOf-Kanten zwischen übernommenen Klassen einzufügen, damit die Hierarchie erhalten bleibt. SC^0: $\boldsymbol{C}' \rightarrow 2^{\boldsymbol{C}'}$ arbeitet in gleicher Weise auf der Klassenhierarchie in einer Sicht.

SC^1: $2^{\boldsymbol{C}} \rightarrow 2^{\boldsymbol{C}'}$ *Nächste übernommene Superklasse*

Um die nächste übernommene Superklasse einer gegebenen Menge von Klassen zu erhalten, wird im folgenden SC^1 verwendet. Im Beispiel liefert diese Funktion das Ergebnis {C_2, C_3} zu {C_9} und {C_2} zu {C_5, C_9}, d.h. es wird eine Art Minimalprinzip realisiert. SC^1 geht top-down vor und beginnt bei der Wurzelklasse. Davon werden sodann alle Subklassen per SC* bestimmt und überprüft, ob das Argument vollends darin enthalten ist. Das Ergebnis sollte so weit unten wie möglich sein, dieselbe Eigenschaft darf also nicht gleichzeitig für jeweilige Elemente von SC* gelten.

$$SC^1(Cl) = \{ c \mid Cl \subseteq SC^0(c) \wedge \forall x \in SC^0(c): Cl \not\subset SC^0(x) \wedge c \in \boldsymbol{C}' \} \text{ mit } Cl \subseteq \boldsymbol{C}$$



SC^2: $2^{\boldsymbol{C}} \rightarrow 2^{\boldsymbol{C}}$ *Nächste Superklasse*

Diese Funktion arbeitet wie SC^1, nur daß das Ergebnis hier nicht Bestandteil der Sicht sein muß. Benötigt wird eine solche Funktionalität, wenn die durch orange referenzierte Klasse nicht übernommen wird (siehe (Ax16), Kapitel 4.3).

$$SC^2(Cl) = \{ c \mid Cl \subseteq SC^*(c) \wedge \forall x \in SC^*(c) : Cl \not\subset SC^*(x) \} \text{ mit } Cl \subseteq \boldsymbol{C}$$

SP*: $\boldsymbol{P} \rightarrow 2^{\boldsymbol{P}}$ *Alle Subproperties zu einem Property*

Sehr ähnlich zu SC*, liefert SP* die Subproperties zum Argument. Hilfsfunktionen $SP_i : 2^{\boldsymbol{P}} \rightarrow 2^{\boldsymbol{P}}$ werden herangezogen um jeweils alle Subrelationen zu berechnen. Die Definitionen lauten wie folgt:

$SP_0(\{p\}) = \{p\}$ mit p= (rdfsfa:ltype, y, rdfsfa:LProperty) $\in \boldsymbol{P}$

$SP_i(SP_{i-1}(...)) = \{ p \mid p=(\text{rdfsfa:ltype, y, rdfsfa:LProperty}) \in \boldsymbol{P} \wedge$
(rdfsfa:osubPropertyOf, y, z) $\in \boldsymbol{P} \wedge$
(rdfsfa:ltype, z, rdfsfa:LProperty) $\in SP_{i-1}(...)\}$

$$SP^*(p) = \bigcup_i SP_i(SP_{i-1}(...SP_0(\{p\})...)) \setminus \{p\}$$

SP^0: $\boldsymbol{P} \rightarrow 2^{\boldsymbol{P}}$ *Übernommene Subproperties zu einem Property*

Diese Funktion liefert, vergleichbar mit SC^0, von einem Property p nur die Subproperties, welche auch in die Sicht übernommen wurden. Einmal mehr sammelt eine Hilfsfunktion $H:2^{\boldsymbol{P}} \rightarrow 2^{\boldsymbol{P}}$ alle Subproperties per SP*(p) und selektiert davon die in die Sicht übernommenen.

$H(Pr) = \{ sp \mid x \in Pr \wedge sp \in SP^*(x) \wedge sp \in \boldsymbol{P}' \}$ mit $Pr \subseteq \boldsymbol{P}$
$SP^0(p) = H(\{p\}) \setminus H(H(\{p\}))$

4.2 Ontologie

In der derzeitigen Version der Resource Description Framework Schema Specification subsumiert man unsauber alle RDF-Tripel, die RDFS-Primitiven enthalten, unter einer Ontologie. Mit der Verwendung von RDFS(FA) stellt sich allerdings die Frage, welche Teilmenge aller Statements denn nun eine Ontologie ausmachen. Der Mensch überblickt die Situation sofort und orientiert sich einfach an den Defintionen auf der Schicht O. Im folgenden soll genau dieses formalisiert werden. Vererbung gemäß dem KAON-Vokabular (Abbildung 18) wird dabei vorausgesetzt.

Definition 4.2 : Eine *Ontologie* in RDFS(FA) ist ein Quadrupel $O = (\boldsymbol{\nu}, \boldsymbol{C}, \boldsymbol{P}, \boldsymbol{A})$ mit

$\boldsymbol{\nu}$ - eine Bezeichnung oder ID der Ontologie. Diese ist identisch mit der Bezeichnung oder ID, wie sie KAON-Server nutzt, um zwischen Schemata in der Faktenbasis zu differenzieren (vgl. Kapitel 2.5) und nicht mit dem Namespace. Eine Ontologie kann damit Ressourcen von verschiedenen Namespaces beinhalten.

$\boldsymbol{C}$ - die Klassenhierarchie (Class-Hierarchy)

$\boldsymbol{C}_1 = \{ (x,y,z) \mid (x,y,z) \in Statements \wedge$
$x=\text{“rdfsfa:ltype“} \wedge z=\text{“rdfsfa:LClass“}\}$
$\boldsymbol{C}_2 = \{ (x,y,z) \mid (x,y,z) \in Statements \wedge x=\text{“rdfsfa:osubClassOf“}\}$
$\boldsymbol{C} = \boldsymbol{C}_1 \cup \boldsymbol{C}_2$

$\boldsymbol{P}$ - die Relationenhierarchie (Property-Hierarchy)

$\boldsymbol{P}_1 = \{ (x,y,z) \mid (x,y,z) \in Statements \wedge$
$x=\text{“rdfsfa:ltype“} \wedge z=\text{“rdfsfa:LProperty“}\}$
$\boldsymbol{P}_2 = \{ (x,y,z) \mid (x,y,z) \in Statements \wedge x=\text{“rdfsfa:osubPropertyOf“}\}$
$\boldsymbol{P} = \boldsymbol{P}_1 \cup \boldsymbol{P}_2$

$\boldsymbol{A}$ - die Menge aller Zusicherungen (Assertions)

$\boldsymbol{A}_x = \{ (x,y,z) \mid (x,y,z) \in Statements \wedge x \in Bz \wedge$
$\exists a,b : (a,y,b) \in \boldsymbol{C} \cup \boldsymbol{P} \}$
$Bz = \{$ rdfsfa:odomain, rdfsfa:orange, rdfs:label, rdfs:seeAlso, rdfs:isDefinedBy, rdfs:comment$\}$
$\boldsymbol{A} = \bigcup_{b \in Bz} A_b$

Erfüllt eine Ontologie $O = (\nu, \boldsymbol{C}, \boldsymbol{P}, \boldsymbol{A})$ die u.a. Axiome (Ax1) bis (Ax8), so nennen wir sie *wohlfundiert*. Die Axiome werden von RDFS(FA) nicht in ihrer Gänze gefordert, sind aber unerläßlich für korrekte Modellierung und die darauffolgende Sichtendefinition.

Was in der Definition geschieht ist simpel: Aus der Menge *Statements* werden diejenigen Elemente entnommen, die eine Domäne auf der Schicht O formalisieren und danach verteilt auf ***C***, ***P*** und ***A*** – das sind Tripelmengen, die Klassen, Relationen und sonstige Zusicherungen definieren. Es interessieren hier weder Instanzen, noch Primitiven und Definitionen aus Metalanguage- und Language-Layer. ***A*** selektiert einzeln Statements mit rdfsfa:odomain, rdfsfa:orange, rdfs:label, rdfs:seeAlso, rdfs:isDefinedBy, rdfs:comment um sie anschließend zu vereinigen. Der Leser sei an dieser Stelle gleich darauf sensibilisiert, daß es sich hier lediglich um syntaktischen Zucker handelt. Die folgenden Axiome könnte man auch direkt auf *Statements* formulieren; die Aufteilung dient lediglich der besseren Lesbarkeit. Es fließen dabei Ideen und Vorschläge aus [RDFS00] und [Tol00] ein.

> Exkurs
> Daß die Verwendung von RDFS(FA) statt des herkömmlichen RDFS hier unerläßlich ist, soll dieser Exkurs zeigen. Der Leser möge sich erneut Abbildung 10 vor Augen führen, welche ja direkt aus der derzeitigen RDFS-Spezifikation des W3C stammt. Wollte man dort in ähnlicher Manier Klassen einsammeln, bekäme man eine Mixtur aus benutzerdefinierten Domänenklassen und Modellierungsprimitiven wie rdfs:ConstraintResource, rdfs:Resource, rdf:Property etc. und schließlich gar rdfs:Class selbst. Es existiert auf RDF-Tripel-Basis keine Möglichkeit zu differenzieren, da es sich bei allen um Instanzen von rdfs:Class handelt. Dasselbe gilt im übrigen auch für Relationen – dort sind sowohl benutzerspezifische als auch modellierungsbezogene Relationen Instanzen von rdf:Property. Mit den subClassOf- und subPropertyOf-Primitiven läuft man ebenfalls ins offene Messer. RDFS taugt also in keinster Weise als Fundament für einen Sichtenmechanismus. Damit nicht genug, fehlt RDFS eine formale Semantik, so daß eine Maschinenverständlichkeit nicht in Frage kommt. Dieses Problem ist zurückzuführen auf Russells Antinomie, wie in Kapitel 2.4 angesprochen. Potentiell unendlich viele Modellierungsebenen, wie sie in RDFS vorgesehen sind, erlauben darüber hinaus keine effiziente Implementierung.

(Ax1) *Existenz einer Wurzelklasse*

$\exists c \in \boldsymbol{C}\ \forall d \in \boldsymbol{C} \setminus \{c\}: d \in SC^*(c)$

Wie bei DAML+OIL, KAON, RDFS usw. muß auch hier eine Wurzelklasse existieren, d.h. eine Klasse, zu der alle anderen direkt oder indirekt Subklassen sind[3]. Diese nennen wir *root*.

(Ax2) *Zyklenfreie Klassenhierarchie*

$\neg\exists c \in \boldsymbol{C} : c \in SC^*(c)$

Keine Klasse darf in der transitiven Hülle der osubClassOf-Relation selbst enthalten, d.h. Subklasse ihrer selbst, sein. Ist dieses Axiom erfüllt, vermeidet man unerwünschte Zyklen in der Klassenhierarchie. Polymorphismus, also mehrere Superklassen zu einer Klasse, ist erlaubt.

(Ax3) *Zyklenfreie Relationenhierarchien*

$\neg\exists p \in \boldsymbol{P} : p \in SP^*(p)$

Keine Relation darf in der transitiven Hülle bezüglich osubPropertyOf selbst enthalten sein. Ist dieses Axiom erfüllt, vermeidet man unerwünschte Zyklen in einer Relationenhierarchie. Mehrere Superproperties zu einem Property sind allerdings erlaubt. Es ist darüber hinaus von einem Relationenwald auszugehen, also mehreren disjunkten Hierarchien.

[3] In DAML+OIL ist das implizit die Klasse *thing* und in RDFS die Klasse *resource*.

(Ax4) *Korrekte Instantiierung der Subklassen*

$\forall x \forall y[(\text{rdfsfa:osubClassOf}, x, y) \in \boldsymbol{C} \rightarrow (\text{rdfsfa:ltype}, x, \text{rdfsfa:LClass}) \in \boldsymbol{C}]$

Zu jeder Definition einer Subklasse x von y muß auch ein korrespondierendes Tripel zur Klassendefinition vorhanden sein.

(Ax5) *Korrekte Instantiierung der Subrelationen*

$\forall x \forall y[(\text{rdfsfa:osubPropertyOf}, x, y) \in \boldsymbol{P} \rightarrow (\text{rdfsfa:ltype}, x, \text{rdfsfa:LProperty}) \in \boldsymbol{P}]$

Zu jeder Definition einer Subrelation x von y muß auch ein korrespondierendes Tripel zur Instantiierung vorhanden sein.

(Ax6) *Korrekte Instantiierung der Relationen*

$\forall x[(\text{rdfsfa:ltype}, x, \text{rdfsfa:LProperty}) \in \boldsymbol{P} \rightarrow \exists y \exists z((\text{rdfsfa:odomain}, x, y) \in \boldsymbol{A} \wedge$
$(\text{rdfsfa:orange}, x, z) \in \boldsymbol{A} \wedge |\{(\text{rdfsfa:orange}, x, z) \in \boldsymbol{A}\}|=1 \wedge$
$(\text{rdfsfa:ltype}, y, \text{rdfsfa:LClass}) \in \boldsymbol{C} \wedge (\text{rdfsfa:ltype}, z, \text{rdfsfa:LClass}) \in \boldsymbol{C})]$

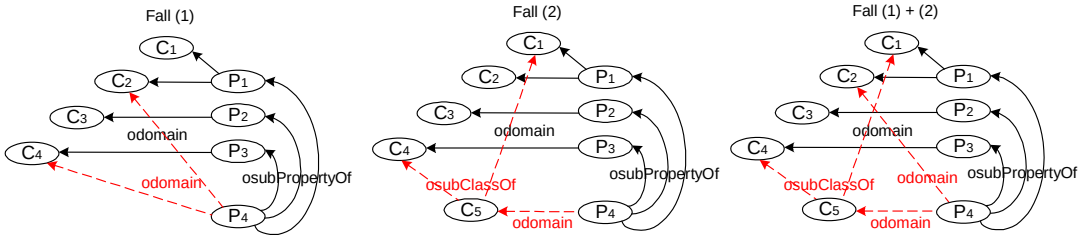
Dieses Axiom sichert, daß zu jeder Relation Domain- und Range-Zusicherungen vorhanden sind. Man beachte, daß multiple Bilder aber nicht multiple Domänen definiert sein dürfen. Die referenzierten Konzepte müssen jeweils in ***C*** sein.

(Ax7) *Domänen von Subrelationen*

(1) $\forall p \forall q \exists c[(\text{rdfsfa:osubPropertyOf}, p, q) \in \boldsymbol{P} \wedge (\text{rdfsfa:odomain}, q, c) \in \boldsymbol{A} \rightarrow$
$(\text{rdfsfa:odomain}, p, c) \in \boldsymbol{A}]$ oder

(2) $\forall p \forall q \forall c[(\text{rdfsfa:osubPropertyOf}, p, q) \in \boldsymbol{P} \wedge (\text{rdfsfa:odomain}, p, c) \in \boldsymbol{A} \rightarrow$
$(\text{rdfsfa:osubClassOf}, c, \text{rdfsfa:odomain.q}) \in \boldsymbol{A}]$

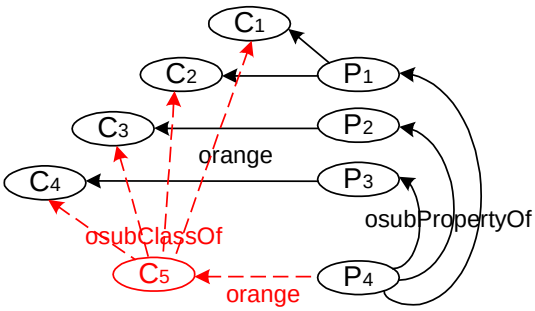
Bei den Domänen der Subrelationen muß differenziert werden. Zum einen kann es mehrere solche Kanten geben, was den Fall hier wesentlich komplexer macht als Axiom (Ax8). Eine Subrelation kann dieselben Domänen haben wie ihre Eltern, es müssen aber nicht alle sein. Dieser Fall ist semantisch korrekt, denn er beschreibt eine Spezialisierung (1). Ähnlich wie in Axiom (Ax8) kann aber auch eine Subklasse der Domänen der Superrelationen zugeordnet werden. Fall (2) beschreibt dieses Szenario, dabei muß c nicht Subklasse aller Domänenklassen sein. Der Ausdruck odomain.q steht dabei für die Domäne des Properties q. Um noch flexibler zu sein, ist auch eine Mixtur aus (1) und (2) zulässig.



(Ax8) *Bilder von Subrelationen*

$\forall p \forall q \forall c[(\text{rdfsfa:osubPropertyOf}, p, q) \in \boldsymbol{P} \wedge (\text{rdfsfa:orange}, p, c) \in \boldsymbol{A} \rightarrow$
$(\text{rdfsfa:osubClassOf}, c, \text{orange.q})]$

Das Bild einer Subrelation muß Subklasse aller Bilder der Superrelationen sein. Im Beispiel ist P_4 eine Spezialisierung von P_1, P_2 und P_3, also sollte auch dessen Range eine entsprechende Verfeinerung sein. Das Axiom erzwingt demzufolge die gestrichelten Kanten und definiert das Bild von P_4 als Subklasse aller Bilder von P_1, P_2 und P_3 (oben bezeichnet als rdfsfa:orange.x). Gemäß Axiom (Ax6) darf jede Relation nur über genau eine orange-Kante verfügen.



4.3 Subontologie

Aufbauend auf der präzisen Definition einer Ontologie in RDFS(FA), läßt sich nun eine Sicht formulieren. Um in der Terminologie zu bleiben, nennen wir eine solche hier Subontologie synonym zu View oder Subschema. Die spezielle Infrastruktur des KAON-API wird keine Replikation einer Datenbasis oder Materialisierung benötigen, so daß einzig und allein Schemainformationen gespeichert werden müssen. Allerdings muß stets ein Bezug zur ursprünglichen Ontologie zur Überprüfung der Korrektheit vorhanden sein, was im folgenden zum Ausdruck kommt. Wenn dort die Rede ist von $\boldsymbol{C}$, $\boldsymbol{P}$ und $\boldsymbol{A}$, so sind damit die gleichnamigen Mengen darin gemeint. Ist die ursprüngliche Ontologie wohlfundiert, so vererbt sich diese Eigenschaft automatisch auf Sichten darauf, solange die u.a. Axiome erfüllt sind. Auch eine Sicht ist wieder ein Schema sobald die Referenz übersehen wird, so daß kaskadiert werden kann.

Definition 4.3 : Eine *Subontologie* in RDFS(FA) ist ein Tupel $SO = (\boldsymbol{\nu}', \boldsymbol{C}', \boldsymbol{P}', \boldsymbol{A}')$ mit

$\boldsymbol{\nu}$ - eine Bezeichnung oder ID der Ontologie. Diese ist identisch mit der Bezeichnung oder ID, wie sie KAON-Server nutzt, um zwischen Schemata in der Faktenbasis zu differenzieren (vgl. Kapitel 2.5) und nicht mit dem Namespace. Eine Subontologie kann damit Ressourcen von verschiedenen Namespaces beinhalten.

$\boldsymbol{C}'$ - die Klassenhierarchie (Class-Hierarchy)

$\boldsymbol{P}'$ - die Relationenhierarchie (Property-Hierarchy)

$\boldsymbol{A}'$ - die Menge aller Zusicherungen (Assertions)

$\boldsymbol{O}^{SO}$ – eine Bezeichnung oder ID der Ursprungs(sub)ontologie

Zusätzlich müssen die Axiome (Ax9) bis (Ax25) für SO erfüllt sein.

(Ax9) $\boldsymbol{C}' \subseteq \boldsymbol{C} \cup \boldsymbol{C}^{neu}$
Axiom (Ax14) fügt eventuell neue osubClassOf-Tripel ein. Hinzu kommen außerdem virtuelle Klassen, allesamt sind hier subsumiert in $\boldsymbol{C}^{neu}$.

(Ax10) $\boldsymbol{P}' \subseteq \boldsymbol{P} \cup \boldsymbol{P}^{neu}$
Axiom (Ax17) fügt eventuell neue osubPropertyOf-Tripel ein. $\boldsymbol{P}^{neu}$ enthält diese zusammen mit virtuellen Relationen, welche allerdings erst in Kapitel 3.2 erläutert werden.

(Ax11) $\boldsymbol{A'} \subseteq \boldsymbol{A} \cup \boldsymbol{A}^{neu}$

Unter Umständen kommen durch (Ax15) und (Ax16) neue Zusicherungen hinzu. Diese sind zusammen mit Filtern und weiteren Primitiven des KAON-Vokabulars (siehe Kapitel 3.2) in $\boldsymbol{A}^{neu}$ berücksichtigt.

(Ax12) *Übernahme Wurzelklasse*

$\exists c \in \boldsymbol{C'} \, \forall d \in \boldsymbol{C'} \setminus \{c\}: d \in SC^*(c)$

Was hier gefordert wird ist einfach, daß der aus der ursprünglichen Ontologie stets vorhandene Wurzelknoten *root* mit übernommen wird. Andernfalls könnte eine disjunkte Menge von Hierarchien entstehen, was modellierungsfremd und nicht konform zu (Ax14) wäre.

(Ax13) *Abgeschlossenheit der Zusicherungen*

$\forall c \forall x \forall y[(\text{rdfsfa:ltype}, c, x) \in \boldsymbol{C'} \rightarrow \forall a \in Dok[(a, c, y) \in \boldsymbol{A} \rightarrow (a, c, y) \in \boldsymbol{A'}]]$

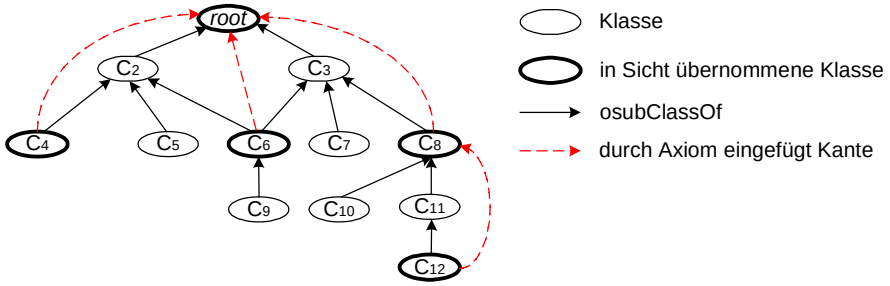
$\forall p \forall x \forall y[(\text{rdfsfa:ltype}, p, x) \in \boldsymbol{P'} \rightarrow \forall a \in Dok[(a, p, y) \in \boldsymbol{A} \rightarrow (a, p, y) \in \boldsymbol{A'}]]$

Für jede in die Sicht übernommene Klasse x, sollten auch die korrespondierenden Zusicherungen wie rdfs:comment, rdfs:seeAlso etc. in $\boldsymbol{A'}$ sein. Man beachte, daß dieses nicht ohne weiteres auch für rdfsfa:odomain und rdfsfa:orange gelten darf (siehe (Ax15) und (Ax16)). Hier gemeint sind lediglich Primitiven zur Dokumentation, nämlich *Dok*={rdfs:comment, rdfs:label, rdfs:seeAlso, rdfs:isDefinedBy}, die auf beliebigen Ressourcen der O-Schicht definiert sein können.

(Ax14) *Klassenhierarchie*

$\forall c \in \boldsymbol{C'} \, \forall sc \in \boldsymbol{C'} [sc \in SC^0(c) \rightarrow (\text{rdfsfa:osubClassOf}, sc_S, c_S) \in \boldsymbol{C'}]$

Durch das Ausblenden von Klassen entstehen Lücken in der Klassenhierarchie, d.h. es ist zu überlegen wie die jeweiligen osubClassOf-Kanten zu übernehmen und zu verändern sind. $SC^0(c)$ liefert zu einer Klasse c alle Subklassen, die auch in der Sicht sein sollen. Das Axiom fordert nun, daß zwischen c und allen $sc \in SC^0(c)$ eine entsprechendes osubClassOf-Statement in $\boldsymbol{C'}$ steht. Sobald virtuelle Klassen in der Sicht enthalten sind wird (Ax14) obsolet. Statt dessen kommt (Ax24) ins Spiel (siehe Kapitel 0 Seite 24).



(Ax15) *Domänen*

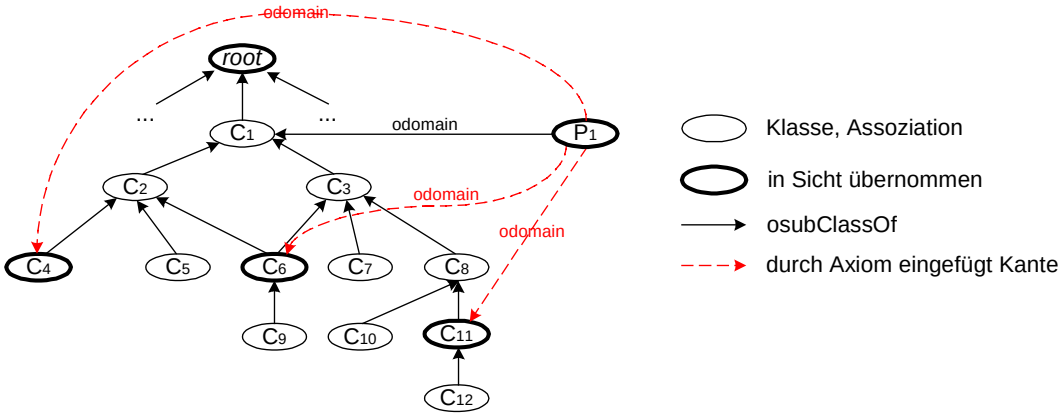
(1) $\forall p \in \boldsymbol{P'} \, \forall c[(\text{rdfsfa:odomain}, p_S, c_S) \in \boldsymbol{A} \wedge c \in \boldsymbol{C'} \rightarrow (\text{rdfsfa:odomain}, p, c) \in \boldsymbol{A'}]$

(2) $\forall p \in \boldsymbol{P'} \, \forall c[(\text{rdfsfa:odomain}, p_S, c_S) \in \boldsymbol{A} \wedge c \notin \boldsymbol{C'} \wedge SC^0(c) \neq \emptyset \rightarrow$
$\forall sc \in SC^0(c): (\text{rdfsfa:odomain}, p_S, sc_S) \in \boldsymbol{A'}]$

(3) $\forall p \in \boldsymbol{P'} \, \exists c \in \boldsymbol{C'} \; (\text{rdfsfa:odomain}, p_S, c_S) \in \boldsymbol{A'}$

Grundsätzlich sind Klassen und Relationen getrennt zu betrachten. Die Attribute und Assoziationen kleben nicht wie bei der Objektorientierung an der Domäne, so daß man hier etwas umdenken muß. Implizit hat jede Subklasse auch die Relationen der Superklasse, es bestehen explizit allerdings keine derartigen

odomain-Kanten in der Ontologie. Wird ein Property übernommen und ist auch dessen Domäne in ***C´***, so hat man keinerlei Probleme: es wird das ursprüngliche odomain-Statement auch in ***A´*** stehen (1). Was soll allerdings passieren, wenn die Domain-Klasse nicht in die Sicht übernommen wurde? (2) regelt diesen Fall und generiert odomain-Kanten zu den Subklassen in der nächsttieferen Ebene. Verfügt ein Property über multiple Domänen in der Ursprungsontologie, so muß zumindest eine odomain-Kante gemäß (1) oder (2) in der Sicht vorhanden sein (3). Im Beispiel findet C_1 nicht den Weg in die Subontologie, so daß die Domain-Kanten von P_1 auf dessen übernommene Subklassen C_4, C_6 und C_{11} zu verlegen sind.



(Ax16) *Ranges*

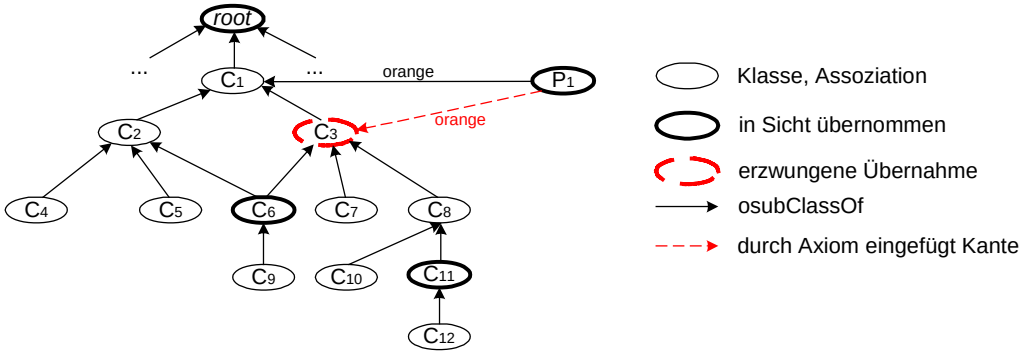
(1) $\forall p \in \boldsymbol{P´}\ \forall c[(\text{rdfsfa:orange}, p_S, c_S) \in \boldsymbol{A} \wedge c \in \boldsymbol{C´} \rightarrow (\text{rdfsfa:orange}, p_S, c_S) \in \boldsymbol{A´}]$

(2) $\forall p \in \boldsymbol{P´}\ \forall c[(\text{rdfsfa:orange}, p_S, c_S) \in \boldsymbol{A} \wedge c \notin \boldsymbol{C´} \wedge |SC^0(c)|=1 \rightarrow$
$\forall sc \in \boldsymbol{C´}\ ((\text{rdfsfa:orange}, p_S, sc_S) \in \boldsymbol{A´} \wedge sc \in SC^0(c))]$

(3) $\forall p \in \boldsymbol{P´}\ \forall c[(\text{rdfsfa:orange}, p_S, c_S) \in \boldsymbol{A} \wedge c \notin \boldsymbol{C´} \wedge |SC^0(c)|>1 \rightarrow$
$\forall sc \in \boldsymbol{C}((\text{rdfsfa:orange}, p_S, sc_S) \in \boldsymbol{A´} \wedge sc \in SC^2(SC^0(c)))]$

(4) $\forall p \in \boldsymbol{P´}\ \forall c[(\text{rdfsfa:odomain}, p_S, c_S) \in \boldsymbol{A} \wedge c \notin \boldsymbol{C´} \rightarrow SC^0(c) \neq \varnothing]$

Die RDFS-Spezifikation erlaubt nur ein orange-Attribut pro Property, was einige Klimmzüge nach sich zieht. Wie auch bei (Ax15) hat man kein Problem, solange die durch orange referenzierte Klasse auch in der Sicht ist (1). Ist dieses nicht der Fall, kann die orange-Kante auf eine übernommene Subklasse umgeleitet werden (2). Gibt es mehrere solche, so wird erzwungen, daß deren nächste Superklasse in der Sicht enthalten ist (3). Letztere liefert der Ausdruck $SC^2(SC^0(c))$. Statt des Erzwingens ist es auch denkbar den Ontology Engineer auf die Alternativen hinzuweisen und ihn letztlich entscheiden zu lassen. Schließlich sichert (4), daß mindestens eine Subklasse in der Sicht sein muß, falls nicht bereits die ursprüngliche orange übernommen wurde. Die Skizze illustriert Fall (3):



(Ax17) *Relationenwald*

$\forall p \in \boldsymbol{P'} \; \forall sp \in \boldsymbol{P'} \; [sp \in SP^0(p) \rightarrow (\text{rdfsfa:osubPropertyOf}, sp_S, p_S) \in \boldsymbol{P'}]$

Wie auch bei der Klassenhierarchie, fristen Relationen zunächst ein disjunktes Dasein in der Sicht. Es muß mit Bedacht entschieden werden, welche subPropertyOf-Kanten zu übernehmen und zu generieren sind. $SP^0(p)$ liefert zu einem Property p alle Subproperties, die auch in der Subontologie sein sollen. Das Axiom fordert nun, daß zwischen p und allen $sp \in SP^0(p)$ eine entsprechendes osubPropertyOf-Statement in $\boldsymbol{P'}$ steht. Im Gegensatz zur Klassenhierarchie, darf hier wiederum ein Wald entstehen.

4.3.1 Filter

Definition 4.3.1 : AF mit (rdfsfa:ltype, AF, KAON:Filter) $\in \boldsymbol{A^{neu}}$ heißt *Attributfilter*, wenn (Ax18) (1) dafür erfüllt ist. Gilt zusätzlich (Ax18) (2) und (3), so nennen wir AF *Assoziationsfilter*.

(Ax18) *Abgeschlossenheit Filter*

(1) $\forall af \forall y[(\text{rdfsfa:ltype}, af, \text{KAON:Filter}) \in \boldsymbol{A^{neu}} \wedge (\text{rdfsfa:odomain}, af, y) \in \boldsymbol{A'} \rightarrow (\text{rdfsfa:ltype}, y, \text{rdfsfa:LProperty}) \in \boldsymbol{P'}]$

(2) $\forall af \forall y \forall p[(\text{rdfsfa:ltype}, af, \text{KAON:Associationfilter}) \in \boldsymbol{A^{neu}} \wedge (\text{rdfsfa:orange}, af, y) \in \boldsymbol{A'} \wedge (\text{rdfsfa:odomain}, af, p) \in \boldsymbol{A'} \rightarrow ((\text{rdfsfa:odomain}, p, y) \in \boldsymbol{A} \vee (\text{rdfsfa:orange}, p, y) \in \boldsymbol{A})]$

Ist ein Filter auf einem Property der Ontologie definiert, so muß diese Relation zwangsweise Bestandteil der Subontologie und reell sein (1). Schließlich regelt Teilaxiom (2), daß das Bild eines Assoziationsfilter mit Domäne oder Bild der korrespondierenden Relation übereinstimmt.

(Ax19) *Attributfilter*

(1) $\forall af \in \boldsymbol{A^{neu}} \; \text{Syntax}_1(af.orange)$

(2) $\forall af \in \boldsymbol{A^{neu}} \; \forall x \; [(\text{rdfsfa:otype}, x, af.odomain.odomain) \in \textit{Ergebnis} \leftrightarrow (af.odomain, x, v) \in \textit{Statements} \wedge \langle af.orange \rangle(v)]$

Zunächst fordert (1) eine korrekte Syntax des Filterausdrucks im orange eines Attributfilters af. Das gleichnamige Prädikat sichert das Enthaltensein des Ausdrucks in einer gewissen Sprache, die erst bei der Implementierung relevant wird. (2) ist auf der Instanzenebene anzusiedeln und beschreibt, welche Tripel das KAON-API bei gegebenem Attributfilter af als Ergebnismenge liefern muß. af.odomain.odomain steht dabei für die Domäne der Domäne des Filters af und <af.orange> ist als Prädikat aufzufassen, das den Filterausdruck realisiert. Demnach sind also nur Tripel im Ergebnis, die Instanz der Domäne des betreffenden Attributs sind und den o.g. Ausdruck, definiert in orange des Filters, erfüllen.

(Ax20) *Assoziationsfilter*

(1) $\forall af \in \boldsymbol{A^{neu}} \forall x \forall i \; [(\text{rdfsfa:otype}, x, af.orange) \in \textit{Ergebnis} \leftrightarrow (af.odomain, i, x) \in \textit{Statements} \wedge (\text{rdfsfa:otype}, i, af.odomain.odomain) \in \textit{Statements}]$

(2) $\forall af \in \boldsymbol{A^{neu}} \forall x \forall i \; [(\text{rdfsfa:otype}, x, af.orange) \in \textit{Ergebnis} \leftrightarrow (af.odomain, i, x) \in \textit{Statements} \wedge (\text{rdfsfa:otype}, i, af.odomain.orange) \in \textit{Statements}]$

Der Assoziationsfilter af stellt sicher, daß nur solche Instanzen in der Sicht zugänglich sind, die auch in Beziehung stehen. Es ist hier entscheidend, ob das Bild des Filters auf die Domäne oder das Bild der Assoziation gelegt wird (1) bzw. (2). Verschiedene Ergebnisse sind die Folge.

4.3.2 Virtuelle Relationen

Definition 4.3.2: VA mit (rdfsfa:ltype, VA, KAON:virtualAttribute) $\in \boldsymbol{P^{neu}}$ heißt *virtuelles Attribut*, wenn (Ax21) erfüllt ist. VA mit (rdfsfa:ltype, VA, KAON:virtualAssociation) $\in \boldsymbol{P^{neu}}$ heißt *virtuelle Assoziation*, wenn (Ax22) gilt.

(Ax21) *Virtuelle Attribute*

(1) $\forall va \in \boldsymbol{P^{neu}}\ \forall c \forall d$ [
(rdfsfa:ltype, va, KAON:virtualAttribute) $\in \boldsymbol{P^{neu}} \wedge$
(rdfsfa:odomain, va, c) $\in \boldsymbol{A'} \wedge$ (rdfsfa:orange, va, rdfs:Literal) $\in \boldsymbol{A'} \wedge$
(KAON:Extension, va, e) $\in \boldsymbol{A'} \wedge \text{Syntax}_2$(va.Extension)]

(2) $\forall va \exists k$[(rdfsfa:ltype, va, KAON:virtualAttribute) $\in \boldsymbol{P^{neu}} \rightarrow$
$|\{$(Extension, va, k) $\in \boldsymbol{A'}\}|=1$]

(3) $\forall va \in \boldsymbol{P^{neu}} \forall c$[
(rdfsfa:odomain, va, c) $\in \boldsymbol{A'} \rightarrow$ (rdfsfa:ltype, c, rdfsfa:LClass) $\in \boldsymbol{C'}$]

Für ein virtuelles Attribut va berechnet das KAON-API den Wert aus vorhandenen Ressourcen bzw. Instanzen. Die Domäne zeigt wie bei herkömmlichen Attributen auf eine Klasse, das Bild jedoch lautet stets rdfs:Literal. Der Berechnungsvorschrift steckt in einem KAON:Extension-Statement. (1) garantiert dessen Korrektheit bezüglich einer gegebenen Sprache, die erst in Kapitel 5.3 näher besprochen wird. Virtuelle Attribute müssen in die Menge $\boldsymbol{P^{neu}}$ und die Domänen und Bilder in $\boldsymbol{A'}$ (1). Pro virtuellem Attribute darf nur ein Extension-Statement existieren (2). Sämtliche Domänen, auf die va referenziert, müssen auch in der Sicht sein (3).

(Ax22) *Virtuelle Assoziationen*

(1) $\forall va \in \boldsymbol{P^{neu}}\ \forall c \forall d \forall e$ [
(rdfsfa:ltype, va, KAON:virtualAssociation) $\in \boldsymbol{P^{neu}}$
$\wedge$ (rdfsfa:odomain, va, c) $\in \boldsymbol{A'} \wedge$ (rdfsfa:orange, va, d) $\in \boldsymbol{A'} \wedge$
(KAON:Extension, va, e) $\in \boldsymbol{A'} \wedge \text{Syntax}_3$(va.Extension)]

(2) $\forall va \exists k$[(rdfsfa:ltype, va, KAON:virtualAssociation) $\in \boldsymbol{P^{neu}} \rightarrow$
$|\{$(Extension, va, k) $\in \boldsymbol{A'}\}|=1$]

(3) $\forall va \in \boldsymbol{P^{neu}}\ \forall c \forall y$ [(rdfsfa:odomain, va, c) $\in \boldsymbol{A'} \rightarrow$ (rdfsfa:ltype, c, y) $\in \boldsymbol{C'}$]

(4) $\forall va \in \boldsymbol{P^{neu}}\ \forall c \forall y$ [(rdfsfa:orange, va, c) $\in \boldsymbol{A'} \rightarrow$ (rdfsfa:ltype, c, y) $\in \boldsymbol{C'}$]

Ähnlich den virtuellen Attributen, muß auch bei einer virtuellen Assoziation va der Query-Ausdruck, definiert per KAON:Extension, einer gewissen Syntax genügen (1). Davon darf es in der Subontologie nur einen geben (2). Die eigentliche Definition von va sollte in $\boldsymbol{P^{neu}}$ stehen und seine odomain-, orange- und Extension-Zusicherungen in $\boldsymbol{A'}$ (1). Domänen und Bilder, auf die verwiesen wird, sollten in der Sicht sein, vgl. Teilaxiome (3) und (4). Im Gegensatz zu importierten Relationen muß der Ontology Engineer diese spezifizieren.

4.3.3 Virtuelle Klassen

Definition 4.3.3: VC mit (rdfsfa:ltype, VC, KAON:virtualClass) $\in \boldsymbol{C}^{neu}$ nennen wir *virtuelle Klasse*, wenn dafür die Axiome (Ax23) bis (Ax25) erfüllt sind.

(Ax23) *Korrekte Instantiierung von virtuellen Klassen*

(1) $\forall vc \in \boldsymbol{C}^{neu}\ \forall e$ [
(rdfsfa:ltype, vc, KAON:virtualClass) $\in \boldsymbol{C}^{neu} \wedge$
(KAON:Extension, vc, e) $\in \boldsymbol{A}' \wedge Syntax_4$(vc.Extension)]

(2) $\forall vc \exists k$[(rdfsfa:ltype, vc, KAON:virtualClass) $\in \boldsymbol{C}^{neu} \rightarrow$
$|\{$(Extension, vc, k) $\in \boldsymbol{A}'\}|=1$]

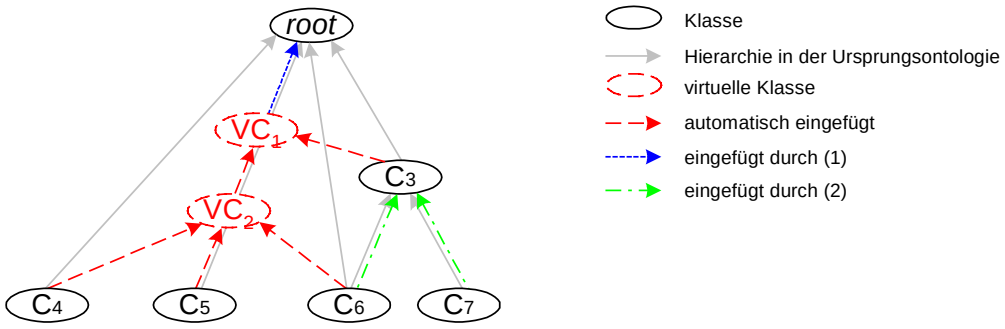
Die Definition einer virtuellen Klasse vc muß sich in $\boldsymbol{C}^{neu}$ befinden (1). Es sollte dazu genau eine Extension-Kante existieren (1), (2). (1) sagt außerdem, daß dieser Query-Ausdruck einer gewissen Syntax gehorchen muß.

(Ax24) *Klassenhierarchie*

(1) $\forall vc \in \boldsymbol{C}^{neu}$ [
$\neg\exists x$(rdfsfa:osubClassOf, x, vc_S) $\wedge$ *Cl* ={ c | c $\in$ SC^0 (vc)} $\wedge$ SC^1(*Cl*)=D
$\rightarrow \forall d \in$ D:(rdfsfa:osubClassOf, vc_S, d_S) $\in \boldsymbol{C}'$]

(2) $\forall a \in \boldsymbol{C}'\ \forall b \in \boldsymbol{C}'\ \forall c \in \boldsymbol{C}\ \forall d \in \boldsymbol{C}$ [
c $\in$ SC*(d) $\wedge$ a $\notin$ SC*(b) $\wedge$ a $\in$ SC^0(b) $\wedge$ a=c $\wedge$ b=d
$\rightarrow$ (rdfsfa:osubClassOf, a_S, b_S) $\in \boldsymbol{C}'$]

Eine virtuelle Klasse vc, deren Extension durch Union spezifziert wird, verfügt zunächst über keine Superklasse. Die Axiomatik sollte diese stattdessen deduzieren. Zu diesem Zwecke sammelt (1) alle Subklassen von vc in der Menge *Cl* und bestimmt per SC^1 deren nächste übernommene Superklasse. Für importierte Klassen, die in der Ursprungs-, nicht jedoch in der Subontologie in Beziehung stehen, vervollständigt (2) die Hierarchie mit entsprechenden Kanten.



Im Beispiel vereinige VC_1 die Klassen C_3 und VC_2, VC_2 die Klassen C_4, C_5, C_6. (1) fügt eine Kante von VC_1 zu *root* ein, da sämtliche Subklassen, also C_3, C_5, C_6, C_7, als nächste übernommene Subklasse *root* vorweisen. D.h. *Cl*={C_3, C_5, C_6, C_7} und SC^1(*Cl*)={*root*}. Dasselbe geschieht nicht für VC_2, da Teilaxiom (1) korrekterweise fordert $\neg\exists$x(rdfsfa:osubClassOf, x, VC_2). Schließlich stehen C_3 und C_7 noch nicht in Beziehung, wie das in der ursprünglichen Ontologie der Fall war. Axiom (2) sorgt hier, wie auch bei C_3 und C_6, für Korrektheit.

(Ax25) *Attributvererbung*

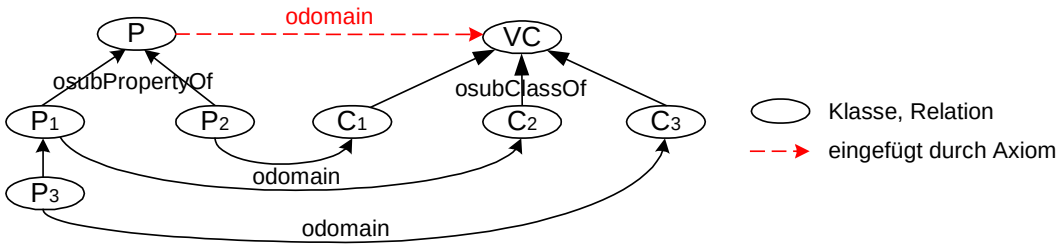
$\forall vc \forall c \forall p' \exists p[$

$(c, \text{rdfsfa:osubClassOf}, vc) \in \boldsymbol{C'} \wedge (\text{rdfsfa:odomain}, p', c) \in \boldsymbol{A'} \wedge$

$p' \in SP^*(p) \cup \{p\} \wedge (\text{rdfsfa:ltype}, p, \text{LProperty}) \in \boldsymbol{A'}$

$\rightarrow (\text{rdfsfa:odomain}, p, vc) \in \boldsymbol{A'}]$

Wie alle Klassen erben auch virtuelle die Relationen ihrer Superklassen. Dieser Vererbungsmechanismus wird expliziert innerhalb des KAON-API. Allerdings besteht auch die Möglichkeit, daß eine virtuelle Klasse Relationen von ihren Subklassen erbt. Das ist dann der Fall, wenn sämtliche Subklassen Domäne einer Relation p sind. Allgemeiner noch kann man argumentieren, daß p auch dann nach oben vererbt wird, wenn Spezialisierungen von p, also Subproperties p´, involviert sind. p muß dann allerdings in die Sicht übernommen werden.



Im Beispiel sind C_1, C_2 und C_3 Subklassen der virtuellen Klasse VC. Alle drei verfügen über Attribute, die zu einer Relationenhierarchie mit P als Wurzel gehören. Dementsprechend fordert dieses Axiom die Übernahme von P und das Vorhandensein einer entsprechenden Domain-Kante auf VC.

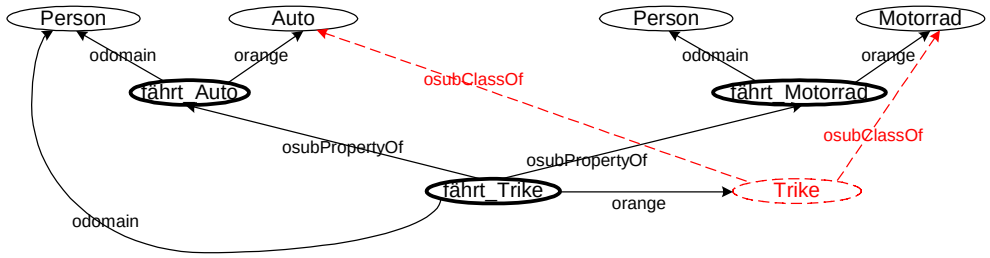
4.4 Beispiel

Die Beispiel greift erneut die Unternehmensontologie auf und führt Kapitel 3.4 fort. Zunächst soll die Ursprungsontologie auf Wohlfundiertheit geprüft werden. Um diese Frage positiv beantworten zu können, ist ein Test auf die Axiome (Ax1) bis (Ax8) vonnöten. Daß es sich um eine zyklenfreie Hierarchie gemäß (Ax2) handelt, läßt sich unschwer erkennen. Die nicht eingezeichneten ltype-Kanten von jeder Klasse zu rdfsfa:LClass führen darüber hinaus zur Gültigkeit von Axiom (Ax4). Schön zu sehen ist außerdem die Forderung nach Vorhandensein der Wurzelklasse *root* (Ax1). Vergleiche jeweils Abbildung 19.

Für die übrigen Forderungen ist zunächst ein Blick auf die Relationen notwendig. In Abbildung 20 sieht man sofort, daß jede der 10 Relationenhierarchien das Axiom (Ax3) der Zyklenfreiheit erfüllt. Im Gegensatz zur Klassenhierarchie muß hier kein eindeutiger Wurzelknoten vorhanden sein. Ltype-Kanten von den einzelnen Knoten zu rdfsfa:LProperty sind wieder wegen der Übersichtlichkeit weggelassen; von einer Erfüllung des Axioms (Ax5) ist deshalb auszugehen. Auch verfügt jede Relation über Domänen und genau ein Bild, so daß man (Ax6) ebenfalls abhaken kann. Interessant sind die Hierarchien zu Anschrift und Position. Bezüglich der Domänen sieht man hier, wie wichtig die Flexibilität von Axiom (Ax7) ist. Die Subproperties Lieferanschrift und Rechnungsanschrift übernehmen nur eine der beiden Domänen der Superrelation, was einer Spezialisierung gleichkommt und somit semantisch korrekt ist. Bei Position sind Domänen und Bilder der Subproperties jeweils Subklassen und damit ebenfalls Spezialisierungen. Axiom (Ax8), das sich um die Bilder von

Subrelationen kümmert, wird nur relevant bei KR_Position und HR_Position. Beide Ranges sind Subklassen des Ranges vom Superproperty Position, so daß das Axiom erfüllt und die Ontologie wohlfundiert ist. Ein spezielles Beispiel zur Problematik der Bilder von Subrelationen zeigt der Exkurs unten.

Exkurs:
Um die Notwendigkeit des Axiomes (Ax8) zu motivieren, hier ein kleiner Exkurs mittels eingehendem Beispiel. Ausgehend von der Forderung, daß jede Relation nur über ein Bild verfügen darf, muß im Falle dieses Beispiels eine neue Klasse „Trike“ eingeführt werden, die Subklasse beider Bilder der Superrelationen ist.



Für die Sicht 2 „Web-Shopping für Endkunden“ sorgt Axiom (Ax15) für die Einfügung von odomain-Kanten zwischen Kunde und Vorname sowie Nachname. Problem ist, daß deren ursprüngliche Domäne, nämlich die Klasse Person, nicht in die Sicht übernommen wurde, Subproperties aber implizit erben. Ähnliches geschieht bei Datum, Lieferanschrift, Rechnungsanschrift und Nummer. Nur bei Debitor und KR_Position kann die ursprüngliche odomain-Kante einfach übernommen werden. Zu den Ranges ist zu sagen, daß ein Großteil (Datum, Lieferanschrift, Rechnungsanschrift, Vorname, Nachname, Nummer) auf rdfs:Literal zeigt, welches ohnehin implizit vorhanden ist. Interessant wird es eher bei den Assoziationen Debitor und KR_Position. Bei beiden sind aber auch die ursprünglichen Bilder (Kunde und Endprodukt) in der Sicht. Axiom (Ax16) ist damit erledigt.

Als trivial kann man Axiom (Ax13) allemal einstufen, es kümmert sich ja bekanntlich um die Abgeschlossenheit der Zusicherungen. In diesem Beispiel finden sich keine annotierenden Zusicherungen à la rdfs:comment etc., so daß sich diese Forderung von selbst erledigt. Die Vervollständigung von Lücken bei Relationenhierarchien gemäß (Ax17) erübrigt sich hier ebenfalls, da jeweils nur ein Attribut bzw. eine Assoziation übernommen wird (KR_Position), mehrere nur auf der gleichen Ebene in die Sicht gelangen (Lieferanschrift, Rechnungsanschrift) oder aber die Hierarchie aus nur einem Attribut besteht (Datum, Vorname, Nachname, Nummer, Debitor).

Beide definierten Filter erfüllen das Axiom (Ax18) der Abgeschlossenheit, d.h. die von den Filtern referenzierten Ressourcen Datum, Debitor und Kunde sind Bestandteil der Sicht. Für das virtuelle Attribut “Preis in Euro“ fordert Axiom (Ax21) nun das Enthaltensein von Preis in der Menge $\boldsymbol{P'}$ und Endprodukt in $\boldsymbol{C'}$. Für Endprodukt gilt das bereits, für Preis allerdings nicht, so daß dieses Attribut neu zu $\boldsymbol{P'}$ stößt. Schließlich gilt $\boldsymbol{P' \subseteq P \cup P^{neu}}$ und damit Axiom (Ax10), weil keine weiteren Properties hinzukommen.

Mit den drei virtuellen Konzepten „Premium“, „Gold“ und „Normal“, die allesamt korrekt instantiiert (Ax23) sind, hat man in diesem Stadium 6 disjunkte Klassen, die alsdann wieder zu einer Hierarchie vervollständigt werden - (Ax24) fügt dazu entsprechende osubClassOf-

Kanten in $\boldsymbol{C}'$ ein. Trivialerweise ist damit (Ax9), nämlich $\boldsymbol{C} \subseteq \boldsymbol{C}' \cup \boldsymbol{C}^{\boldsymbol{neu}}$, erfüllt; außerdem erzwingt Axiom (Ax12) die Übernahme von *root*. Die ursprüngliche Unternehmensontologie genügt also der Wohlfundiertheitsforderung nach Definition. Die Subontologie „Web-Shopping für Endkunden“ ist konsistent dazu, weil die Axiome (Ax9) bis (Ax25) erfüllt sind.

5 Umsetzung

Nach der formalen Konsistenzaxiomatik in Kapitel 4, gilt es nun den Sichtenmechanismus in der KAON-Infrastruktur zu integrieren. Die Gesamtlösung, genannt *KAON-Views*, besteht aus mehreren Teilen. Zum einen der Erweiterung des objektorientierten KAON-API um neue Klassen (siehe Kapitel 5.1). Zweiter Hauptbestandteil ist die Inferenzmaschine *com.ontoprise.inference.Evaluator* (cf. [DBSA]) zur Axiomprüfung und Extensionsbestimmung virtueller Ressourcen (siehe Kapitel 5.2 und 5.3). Komponenten von KAON-Views sind außerdem die eingangs erwähnte Sichtendefinitionssprache sowie die Axiome aus Kapitel 4. Später angedacht sind außerdem ein Plug-In zur komfortablen Definition der Axiome und auch die Erweiterung von KAON-SOEP, dem Simple Ontology Browser and Editor Plug-In, zur graphischen Definition einer Sicht.

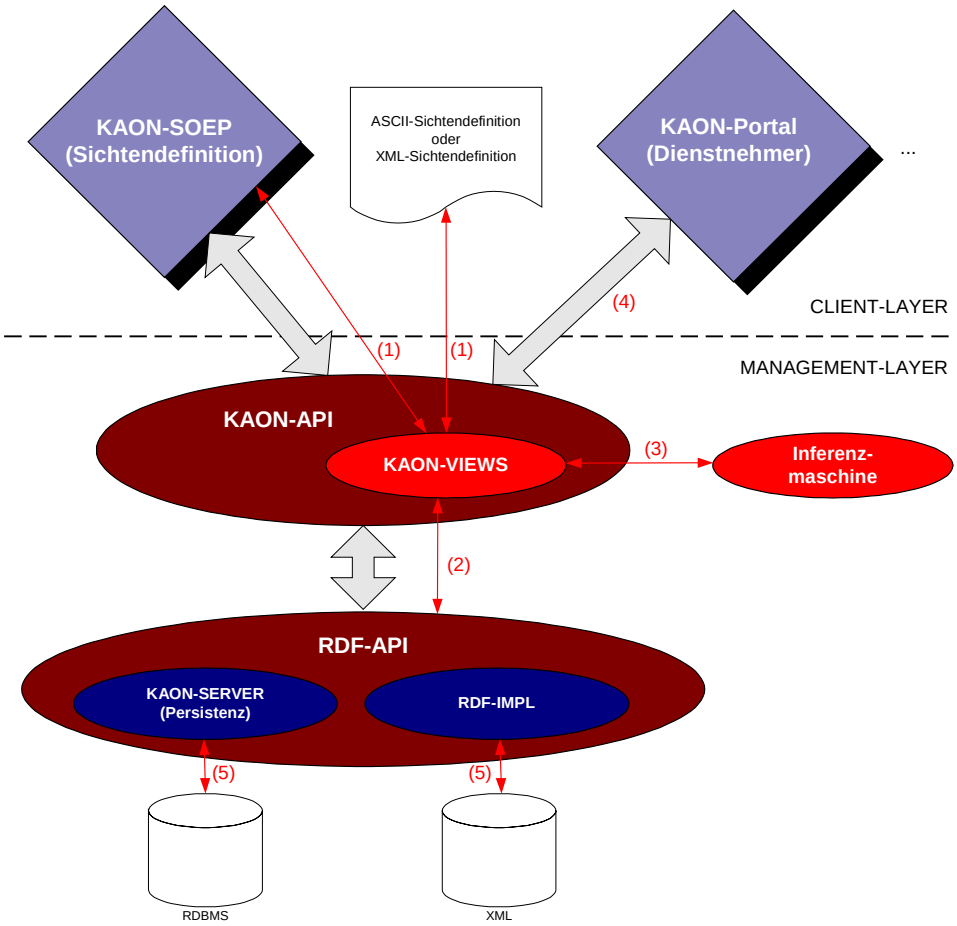


Abbildung 22: KAON-Views Überblick

Abbildung 22 gibt einen Überblick des Zusammenspiels der einzelnen Komponenten. Eine Sicht wird graphisch per KAON-SOEP, per ASCII-Datei oder per XML definiert bzw. geladen (1). Ergebnis ist eine Repräsentation als RDF-Model mit Hilfe von Objekten des RDF-API (2). Ein nächster Schritt besteht aus Konsistenzcheck, sowie Deduktion von Klassenhierarchie, Domänen, Bilder, als auch der Extension virtueller Ressourcen per Inferenzmaschine (3). Erst danach kann ein Dienstnehmer, wie z.B. KAON-Portal, transparent auf einer Sicht arbeiten (4). Persistenz wird erreicht durch KAON-Server oder durch Serialisierung in XML (5).

5.1 KAON-API

Dieses Subkapitel dokumentiert die Eingriffe in das bereits vorhandene KAON-API (siehe auch Kapitel 2.5). Das Klassendiagramm in Abbildung 23 ist zugunsten der Übersichtlichkeit bewußt unvollständig dargestellt. Alle Änderungen sind dabei mit Pfeil markiert bzw. dick umrandet. Diese Neuerungen bilden den wesentlichen Teil von KAON-Views, dazu gehören jedoch auch die Transformation von Subontologie in Sichtendefinition u.u. (Kapitel 5.2.2), sowie Axiomprüfung und Extensionsbestimmung per Inferenzmaschine (Kapitel 5.2 und 5.3).

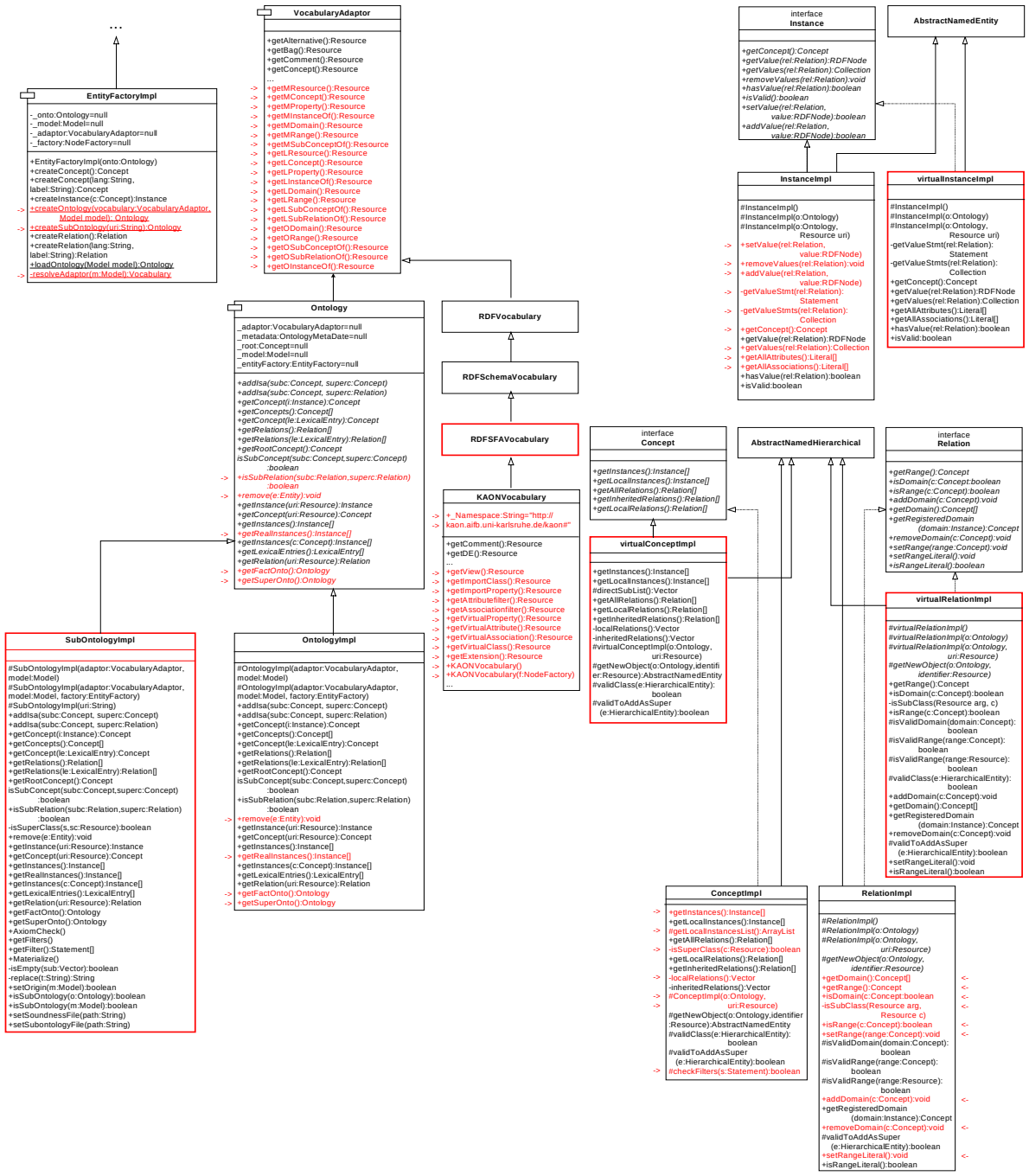


Abbildung 23: KAON-API

Hervorzuheben sind die neuen Klassen SubOntologyImpl, virtualConceptImpl, virtualRelationImpl, virtualInstanceImpl und tiefgreifende Änderungen in EntityFactoryImpl, ConceptImpl und InstanceImpl. Modifikationen an einer Sichtendefinition oder an Instanzen werden unmöglich, wenn das KAON-API auf einer Subontologie operiert. Entsprechende Methoden werfen in diesem Falle eine NotSupportedException. Die Primitiven für RDFS(FA) finden sich nun bereits als abstrakte Methoden in VocabularyAdaptor und implementiert im neuen RDFSFAVocabulary. Schließlich stellt KAONVocabulary die Primitive für das KAON-Views-Vokabular zur Verfügung.

Ein Szenario im unten abgebildeten Sequenzdiagramm verschafft exemplarische Klarheit über die neue Funktionalität. Schön zu sehen ist, wie dem Dienstnehmer transparent bleibt, daß er auf einer Sicht operiert. In einem ersten Schritt generiert er sich mit Hilfe von EntityFactoryImpl eine Ontologie. Darin wird entschieden, ob es sich um eine Sicht oder um eine konkrete Ontologie handelt und dementsprechend eine Instanz von SubOntologyImpl oder OntologyImpl retourniert. Da beide das Interface Ontology implementieren, ist clientseitig keine Differenzierung vonnöten. Dasselbe Spiel wiederholt sich dann mit Konzepten und deren Instanzen. Der Dienstnehmer sendet die Botschaft getConcepts(), woraufhin die Instanz von SubOntologyImpl die Faktenbasis durchsucht und eine Mixtur von ConceptImpl- und virtualConceptImpl-Objekten per Array zurückgibt. Dieses ist möglich, da beide das Interface Concept implementieren. Wiederum muß sich der Client nicht um die Unterschiede kümmern und bekommt von der Differenzierung nichts mit. Sehr ähnlich sieht das Prozedere mit reellen und virtuellen Instanzen aus.

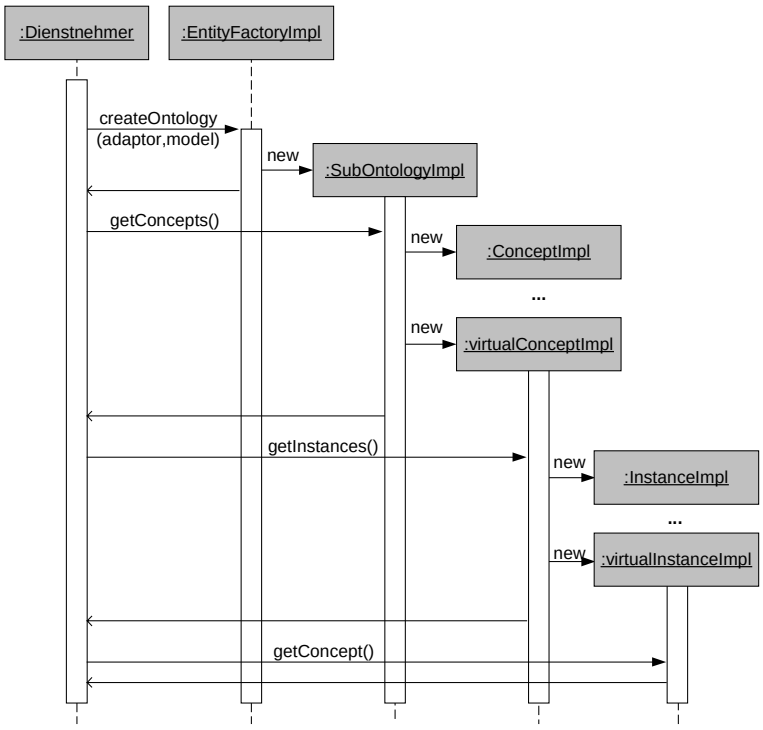


Abbildung 24: Sequenzdiagramm

Abschließend sei noch erwähnt, daß die komplette Erweiterung des KAON-API nur korrekt funktioniert, wenn die Sicht der Konsistenzaxiomatik genügt. Das zugrundeliegende RDF-Model muß also zuvor durch den Axiomcheck gelaufen sein (siehe Kapitel 5.2). Im folgenden werden Filter-Funktionalität, virtuelle Klassen und Relationen näher beleuchtet.

5.1.1 Filter

Die in Kapitel 4.3.1 definierten Assoziationsfilter gehen nicht in neuen Klassen auf, sondern schlagen sich in ConceptImpl nieder. Grund dafür ist die Tatsache, daß ein Dienstnehmer des KAON-API nur mittels ConceptImpl.getInstances() zu Instanzen kommt. Es existiert auch noch der Umweg über Ontology.getInstances(Concept), dabei handelt es sich allerdings nur um eine Fassade. Deshalb muß an dieser Stelle geprüft werden, welche Instanzen den auf das jeweilige Konzept definierten Filtern genügen. Ein Client bekommt demnach nur solche zu Gesicht und kann mit deren OO-Repräsentanten (InstanceImpl) konsistent (man denke an RelationImpl.getValueStmt(Relation)) weiterarbeiten.

Innerhalb des Konstruktors in SubOntologyImpl wird das zugrundeliegende RDF-Model nach vorhandenen Filtern durchsucht. Dazu existiert nun eine öffentliche Methode getFilters(). Für Attributfilter fordert diese Methode alle Relationen, auch die vererbten, die als Domäne das aktuelle Konzept vorweisen. Ein Assoziationsfilter zeigt mittels Range-Kante auf das zu filternde Konzept; demnach ist zu prüfen, ob das aktuelle Konzept diesem entspricht oder aber Subklasse davon ist. getFilters() liefert einen Statement-Array, welcher seinen Weg in eine globale Klassenvariable findet.

Interessanter ist die Prüfung in getLocalInstancesList(): Eine Instanz wird nur retourniert, wenn sie allen Filtern, die auf das aktuelle Konzept definiert sind, genügt. Bei Assoziationsfiltern bereitet die Vererbung von Relationen (Domäne der Assoziation) keine weiteren Umstände, weil in der Faktenbasis direkt auf Vorhandensein einer Assoziationsinstanz geprüft wird. Mindestens eine muß vorhanden sein, damit das aktuelle Konzept sichtbar bleibt – die Semantik der Assoziationsfilter lautet: nur solche Konzeptinstanzen sollen zugänglich sein, die auch eine Beziehung eingegangen sind. Der Programmablaufplan in Abbildung 25 illustriert den Algorithmus.

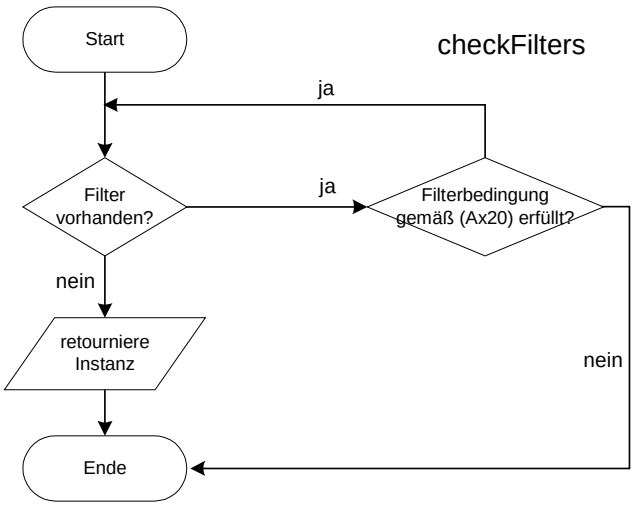


Abbildung 25: checkFilters()

Die Realisierung der Attributfilter geht einen anderen Weg. Problem ist, daß dabei die Filterbedingungen in einer gewissen Sprache zum Ausdruck kommen muß. So möchte man in einer Sicht z.B. nur Personen belassen, deren Alter die Ziffer 18 übersteigt. Aus gutem Grunde findet sich die Lösung nicht an dieser Stelle, sondern erst in Kapitel 5.3, welches sich in erster Linie um die Extensionen virtueller Ressourcen kümmert.

Relationenvererbung bezüglich der Konzepthierarchie ist, wie oben geschildert, in der Filterrealisierung integriert. Diese Eigenschaft soll nicht darüber hinwegtäuschen, daß sich Filter nicht bezüglich der Relationenhierarchie vererben. D.h. ist ein Filter auf Relation r_1 definiert und gilt (rdfsfa:osubPropertyOf, r_2, r_1), so zeigt er in Bezug auf r_2 keine Wirkung[4]. Diese Eigenschaft soll später dem Ontology Engineer den Überblick erleichtern und erhält die Wartbarkeit des Codes. Das Werkzeug zur Sichtendefinition könnte auf Knopfdruck dennoch eine Vererbung ermöglichen, indem es für eine gesamte Relationhierarchie Filter generiert. Erwähnenswert ist außerdem, daß die Kaskadierung nicht für Filter gilt, d.h. basiert eine Subontologie auf einer Sicht, so werden jeweils nur die in der aktuellen Sicht definierten Filter berücksichtigt.

5.1.2 Virtuelle Relationen und Instanzen

VirtualRelationImpl ist die OO-Repräsentation von virtuellen Attributen und Assoziationen. Viel ändert sich jedoch nicht gegenüber RelationImpl. In SubOntologyImpl.getRelations() wird dafür Sorge getragen, daß zwischen reellen und virtuellen Relationen differenziert wird.

Die einzige Möglichkeit per KAON-API an Konzeptinstanzen zu kommen läuft über ConceptImpl.getInstances(). Erst davon ausgehend, kann ein Dienstnehmer an Attributwerte und Assoziationen gelangen. Er benutzt dazu die Methoden InstanceImpl.getValues(Relation rel) und spezifiziert die gewünschte Relation mit dem Argument rel. Retourniert wird eine Collection von Statements, die dem Muster (<Instanz-ID>, <Relation-ID>, ?) entsprechen.

Virtuelle Instanzen entstehen nur an einer Stelle: in virtualConceptImpl.getLocalInstances(). Vorhandene KAON:Extension-Statements zu virtuellen Konzepten müssen ausgewertet und materialisiert werden. Kapitel 5.3 behandelt diese Problematik en détail. Insofern unterscheidet sich die Bestimmung der Instanzen kaum von der reeller. Das KAON-API differenziert dennoch zwischen ConceptImpl und virtualConceptImpl, denn spätestens bei Behandlung von Änderungen werden sich die Unterschiede stärker bemerkbar machen.

Definition 5.1.2 : Eine Instanz i mit (rdfsfa:otype, i, vc) heißt *virtuelle Instanz* zu einem virtuellen Konzept vc, wenn i in der Ergebnismenge des Query-Ausdruckes q in (KAON:Extension, vc, q) enthalten ist.

5.1.3 Kaskadierung und Vererbung

Dieses Kapitel befaßt sich mit der Problematik der impliziten Vererbung und der Kaskadierung von Sichten. Problem ist, daß die in der RDFS-Spezifikation natürlichsprachlich beschriebene Vererbung von Klassen und Relationen im KAON-API expliziert werden muß. Ist eine Klasse C_2 Subklasse von C_1, so zählt auch eine Instanz von C_2 als Instanz von C_1. Dieses Phänomen wird im folgenden präziser definiert:

[4] Innerhalb des RDF-API lautet die Reihenfolge für Tripel (<subject>, <predicate>, <object>) statt (<predicate>, <subject>, <object>) in formalen Modell von [RDF].

Definition 5.1.3 : *Lokale Instanzen einer reellen Klasse* y sind Tupel der Form (rdfsfa:otype, x, y).

Lokale Instanzen einer virtuellen Klasse sind das Ergebnis des Queryausdruckes in KAON:Extension.

Instanzen einer reellen Klasse sind die lokalen Instanzen aller Subklassen. Subklassen können reell oder virtuell sein. In einer Subontologie sind nur die sichtbaren Konzepte zu berücksichtigen.

Instanzen einer virtuellen Klasse sind die lokalen Instanzen aller Subklassen. Subklassen können reell oder virtuell sein. Dementsprechend beinhaltet das Ergebnis eine Mixtur aus reellen und virtuellen Instanzen.

Auf diesen Definitionen und der bisherigen Implementierung von ConceptImpl.getInstances() aufbauend, läßt sich bereits ein Algorithmus angeben. Zunächst sei per Programmablaufplan beschrieben, wie sich die Instanzen zu reellen Konzepten in einer Sicht berechnen (Abbildung 26). Es wird jeweils eine Instanz von ConceptImpl bzw. virtualConceptImpl kreiert und davon wieder getInstances() gefordert[5].

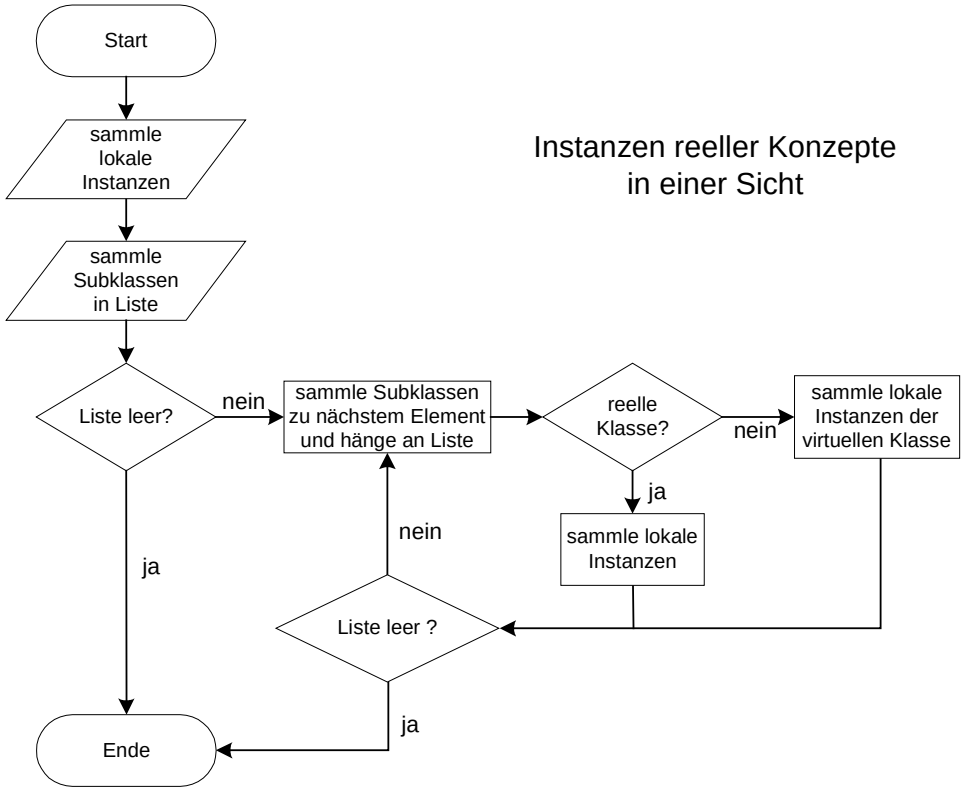


Abbildung 26: ConceptImpl.getInstances()

[5] Methoden in AbstractNamedHierarchical, insbesondere getSub() und getSuper(), funktionieren in beiden Subklassen conceptImpl und virtualConceptImpl, weil nur auf osubClassOf (hierarchyPredicate)-Tripeln operiert wird. Vorsicht ist aber geboten, wenn diese Methoden zum Aufsammeln von Instanzen verwendet werden: getSub() liefert einen Array von HierarchicalEntites, tatsächlich verstecken sich dahinter aber ConceptImpl (siehe AbstractNamedHierarchical.allSubList()), die nicht auf virtualConceptImpl gecastet werden können. D.h. beim Aufruf von getLocalInstances() wird immer ConceptImpl.getLocalInstances() referenziert und nicht etwa virtualConceptImpl.getLocalInstances(). Das führt zu falschem Ergebnis und/oder Exceptions. Deshalb müssen getInstances() in ConceptImpl und virtualConceptImpl neu programmiert werden und dabei rein auf Tripel operieren.

Die Kaskadierung soll Sichten als Ursprungsontologien zulassen und das in beliebiger Tiefe. Diese Anforderung steckt in SubOntologyImpl.setOrigin(), aufgerufen im Konstruktor. Hier wird per Parsing des Models geprüft, ob es sich bei der Superontologie wieder um eine Sicht handelt. Wenn ja, wird auch diese wieder analysiert und auf Vorhandensein eines KAON-View-Statements (siehe Kapitel 3.2) geprüft. Dieses wiederholt sich bis eine reelle Ontologie vorliegt.. Kommt es während dieser Prozedur zu Fehlern, etwa dem Fehlen einer Superontologie, verweigert KAON-Views den Dienst, denn ein korrektes Funktionieren ist in diesem Falle nicht gewährleistet. In der jetzigen Version ist nur eine Kette von Superontologien erlaubt, kein Baum. D.h. jede Sicht darf über nur eine Superontologie verfügen; mehrere würden den Aufwand erheblich steigern und sind in einem nächsten Schritt zu implementieren.

5.2 Axiomprüfung

Nachdem nun das KAON-API mit allerlei virtuellen Ressourcen auf Sichtentauglichkeit getrimmt wurde, stellt sich die Frage, wie die Axiome denn tatsächlich zu überprüfen bzw. zu erfüllen sind. Auf der Hand liegt zunächst eine Operationalisierung mit Hilfe des RDF-API. All- und Existenzquantoren usw. müßten dabei in Java formuliert werden. Die Axiome wären zwar effizient, aber inflexibel implementiert. Interessanter und gleichzeitig wartungsfreundlicher gestaltet sich ein Axiomcheck per Inferenzmaschine. Die Entwurfsentscheidung fiel deshalb letztendlich auf diese Lösung. Zum Einsatz kommt, wie bereits eingangs erwähnt, com.ontoprise.inference.Evaluator (cf. [DBSA]).

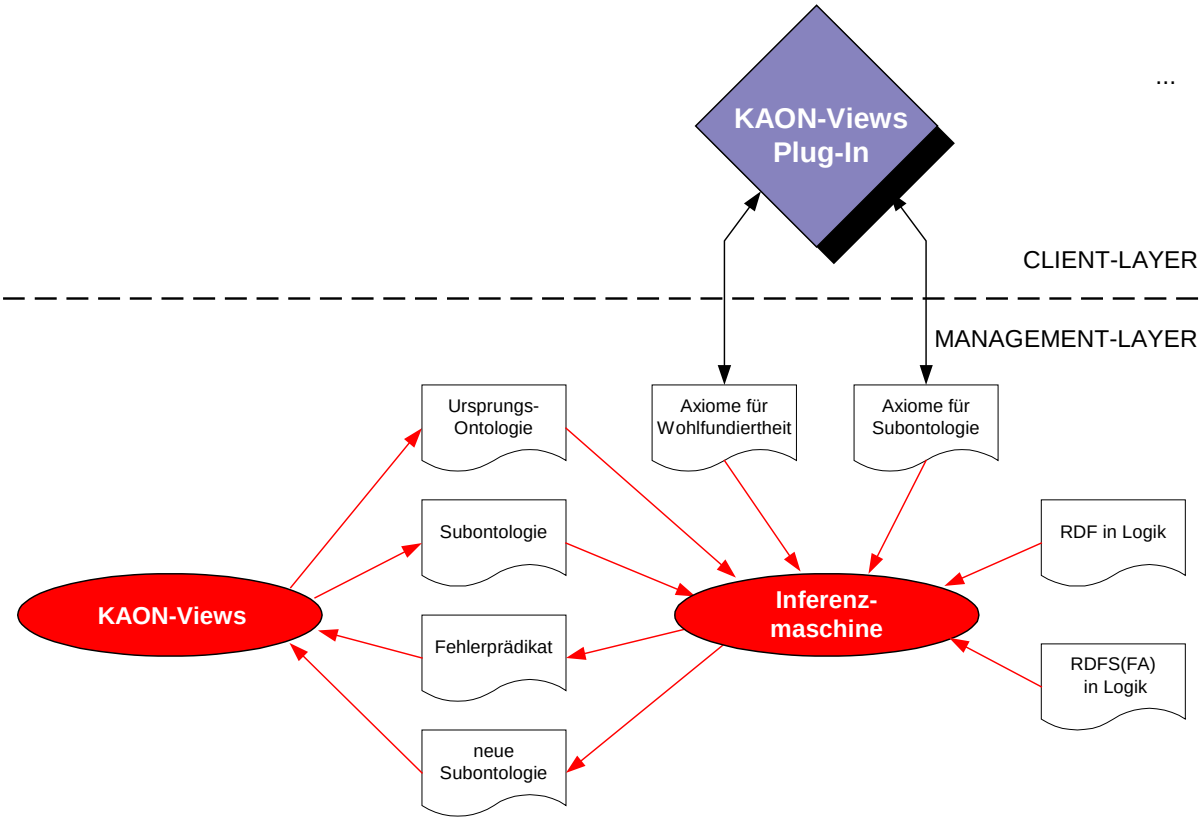


Abbildung 27: Axiomprüfung Überblick

Abbildung 27 gibt einen groben Überblick des Datenflusses. Angeworfen wird der Check sobald ein SubOntologyImpl-Objekt instantiiert wird. Ursprungs- und Subontologie sind in F-Logic-Syntax zu transformieren und dienen als Eingabe für die Inferenzmaschine. Darüber hinaus benötigt man die Axiome zur Wohlfundiertheit und zur Subontologiekonsistenz als Input. Es ist angedacht, daß diese beiden Textdateien später komfortabel per KAON-Views Plug-In konfiguriert werden können. Die Inferenzmaschine deduziert zunächst, ob das Fehlerprädikat erfüllt ist und überprüft damit die Konsistenz der Subontologie. Liegt Konsistenz vor, so inferieren die Axiome Konzept- und Relationenhierarchien, sowie Domänen und Bilder innerhalb der Sicht. SubOntologyImpl liest das Ergebnis schließlich wieder in ein RDF-Model ein. Derselbe Vorgang wiederholt sich für alle Superontologien, wenn die aktuelle Sicht wiederum auf einer Sicht basiert. Sobald dabei nur einmal das Fehlerprädikat erfüllt werden kann, verweigert KAON-Views den Dienst.

Strenggenommen müßte man sämtliche (Sub-)Ontologien in der Kaskade auch auf Korrektheit bezüglich RDF und RDFS(FA) prüfen. Dieses Problem könnte gleichzeitig mit Hilfe des Inferenzprozesses gelöst werden. Für die Validierung einer Menge von Statements auf RDF-Konformität existiert bereits ein Regelwerk in Logik (cf. [WR00]). Dasselbe gilt nicht für RDFS(FA) – hier existiert neben dem Vokabular bisher nur eine modelltheoretische Formalisierung. Eine logische Repräsentation würde es erlauben ein Schema, hier also sowohl Ontologie als auch Subontologie, auf Korrektheit bezüglich RDFS(FA) zu prüfen. Beide Problemstellungen bleiben in dieser Arbeit allerdings außen vor, da solche Fragen das KAON-API insgesamt betreffen, also nicht rein sichtenbezogen sind.

5.2.1 Transformation in Logik

Um die Dienste der Inferenzmaschine nutzen zu können, bedarf es der Transformation eines Model-Objektes in Logik. Die grundlegende Idee dazu stammt aus [WR00], wo ein Tripel (subject, predicate, object) ∈ *Statements* einfach zu einem dreistelligen Prädikat *statement*(subject, predicate, object) wird. Schritt 2 ist die Formulierung der Axiome (Ax1) bis (Ax25) in formaler Logik. Bisher standen die Axiome abstrakt, d.h. fernab jeglicher Entwurfsfragen, im formalen Modell von RDF.

Prädikat	Neues Prädikat
$x \in SC^*(y)$	`isSC(x,y)` `isSC_(x,y)`

$x \in SC^0(y)$	`isSC0(x,y)`
	`isSC02(x,y)`
	`isSC03(x,y)`

$x \in SC^1(\{y\})$	`isSC1(x,y)`
$x \in SC^2(\{y\})$	`isSC2(x,y)`
$x \in SP^*(y)$	`isSP(x,y)` `isSP_(x,y)`
$x \in SP^0(y)$	`isSP0(x,y)`
$(x,y,z) \in \boldsymbol{C}$	`isC(x,y,z)`
$(x,y,z) \in \boldsymbol{P}$	`isP(x,y,z)`
$(x,y,z) \in \boldsymbol{A}$	`isA(x,y,z)`
$(x,y,z) \in \boldsymbol{C}'$	`isC_(x,y,z)`
$(x,y,z) \in \boldsymbol{P}'$	`isP_(x,y,z)`
$(x,y,z) \in \boldsymbol{A}'$	`isA_(x,y,z)`

com.ontoprise.inference.Evaluator ist ursprünglich als F-Logic-Inferenzmaschine konzipiert. Die objektorientierte Semantik von Frame-Logic kommt hier allerdings nicht zum Tragen, es werden lediglich All- und Existenzquantor benötigt. Nötig ist eine Umkodierung der Hilfsfunktionen aus Kapitel 4.1, sowie des impliziten Prädikats ∈ für die Mengen ***C***, ***P***, ***A*** und ***C´***, ***P´***, ***A´***. Die Tabelle oben zeigt, daß letztere nun zu expliziten Prädikaten mit den Bezeichnungen isC, isP, isA und isC_, isP_, isA_ mutieren. Die Hilfsfunktionen werden ebenfalls als Prädikate im Sinne von Enthaltensein realisiert.

Grundsätzlich ist bei Axiomen zu differenzieren zwischen solchen, die neue Statements einfügen, beispielsweise neue osubClassOf-Kanten, und solchen, die eine Konsistenzforderung zum Ausdruck bringen. Neben Syntax und einigen F-Logic-Spezialitäten ändert sich an den erstgenannten prinzipiell nichts. Nicht so bei den übrigen, dort muß umgeschrieben werden, so daß im Falle von Nichterfüllung ein Fehlerprädikat zutrifft. Dieses wird neu eingeführt und trägt die Signatur *violation*(x,y), während x bei Zutreffen des Prädikats jeweils eine Zeichenkette mit Fehlerbeschreibung beinhaltet und y auf die betroffene Ressource verweist, d.h. deren URI liefert. Abgesehen von eventuellen Existenzquantoren in der Prämisse, hat man hier also Hornregeln mit je einem violation- oder is*X*-Prädikat im Kopf. Unten abgebildet sind die beiden Formeltypen, jeweils in PL1- und in F-Logic-Syntax. Eine ausführliche Liste aller Axiome findet sich im Anhang.

Typ(1) ∀x∀y∀z [*is*X(x,y,z) ← Prämisse]
Typ(2) ∀ressource[*violation*(<Fehlerbeschreibung>, ressource) ← Prämisse]

Typ(1)
```
FORALL x,y,z isX(x,y,z) <- <Prämisse>.
```
Typ(2)
```
FORALL Resource
violation("Fehlerbeschreibung",Resource)
<- <Prämisse>.
```

5.2.2 Ablauf

Dieses Kapitel schildert exemplarisch die komplexe Interaktion zwischen KAON-API und Inferenzmaschine. Eine wichtige Rolle dabei spielt die Klasse Trafos, ebenfalls Bestandteil von KAON-Views und verantwortlich für allerlei Umkodierungen. Die Transformation von objektorientiertem RDF-Model in eine F-Logic-Zeichenkette geht wie folgt von statten: Sämtliche Statements der Ursprungsontologie werden zu einem korrespondierenden F-Logic-String “statement(x,y,z).“ (Methode Model2Logic). Bei einer Subontologie differenziert View2Logic(Model) zwischen Statements der Klassenhierarchie, der Relationenhierarchie und Zusicherungen. Dementsprechend wird aus (x,y,z) entweder isC_(x,y,z), isP_(x,y,z) oder isA_(x,y,z).

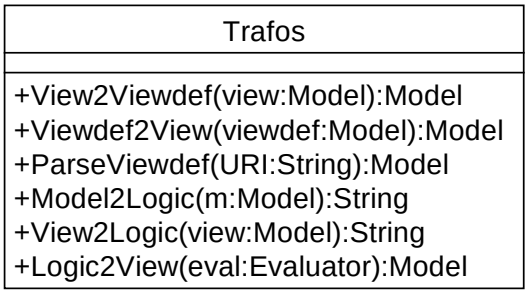


Abbildung 28: Klasse Trafos

Neben KAON-SOEP soll ein Ontology Engineer später in die Lage versetzt werden, eine Sicht mittels einfacher Textdatei zu spezifizieren. ParseViewdef(URI) dient dazu diese in ein RDF-Model zu transformieren. Ergebnis ist eine RDF-Beschreibung der Sichtendefinition. Eine Sichtendefinition kann nicht nur als ASCII-Datei, sondern auch als RDF-Model innerhalb KAON-Server und als XML-Serialisierung davon gespeichert und geladen werden.

View2Viewdef(Model) und Viewdef2View(Model) operieren auf RDF-Ebene und erledigen die einfache Transformation von Sichtendefinition in Sicht und umgekehrt. Viewdef2View kommt zum Einsatz nach dem Parsing einer ASCII- oder XML-Sichtendefinition, bzw. nach Laden einer Sichtendefinition aus KAON-Server. Es ist die Vorstufe zum Axiomcheck, bei der lediglich import-Statements in Definitionen umgewandelt werden. View2Viewdef kommt bei der Speicherung zum Einsatz und verwirft sämtliche Statements zu Klassenhierarchie, Relationenhierarchie, sowie Domains und Ranges, da diese später wieder automatisch inferiert werden. Darüber hinaus ändert die Methode rdfsfa:ltype in KAON:importClass bzw. KAON:importProperty.

In welcher Reihenfolge die Methoden der Klasse Trafos zum Einsatz kommen und wie sie mit der Axiomprüfung interagieren, soll im folgenden gezeigt werden. Das Sequenzdiagramm in Abbildung 29 exemplarisiert die Vorgänge anhand eines Szenarios.

Ein Dienstnehmer wendet sich an das KAON-API, genauer an EntityFactoryImpl, mit einer ASCII-Sichtendefinition, spezifiziert durch den Parameter URI. Die Anfrage führt zur Instantiierung eines SubOntologyImpl-Objekts, dessen Konstruktor die Axiomprüfung beinhaltet. Zunächst wird dazu aus der ASCII-Datei per Parsing ein korrespondierendes RDF-Model-Objekt generiert. Die Klasse Trafos stellt dazu die Methode ParseViewDef(URI) bereit. Es schließt sich die Transformation in eine Sicht an. Wie bereits zuvor erwähnt, ist diese sehr simpel und setzt lediglich import-Statements in rdfsfa:ltype um.

SetOrigin() extrahiert den Verweis auf die Ursprungsontologie, referenziert diesen URI und führt darauf wiederum ein Parsing durch. Handelt es sich dabei um eine Sichtendefinition, erkennbar am Vorhandensein eines KAON:View-Statements, wiederholt sich das Spiel, bis die oberste Superontologie gefunden ist. Sämtliche RDF-Models werden dabei in einem Vector gespeichert. Sobald eine Superontologie nicht referenziert werden kann, beim Parsing Fehler auftreten oder Zyklen vorhanden sind verweigert KAON-Views den Dienst.

Im Szenario ist dieses nicht der Fall, so daß mit AxiomCheck() fortgefahren wird. Die Sichtenkaskade durchläuft nun die Prüfung auf Wohlfundiertheit einer Ontologie und Konsistenz einer Subontologie jeweils paarweise abwechselnd von oben nach unten. Dazu ist das oberste RDF-Model in Logik zu transformieren und der Inferenzmaschine zu übergeben. Nach Instantiierung von com.ontoprise.inference.Evaluator geschieht dieses mittels Model2Logic() und compileString(...). Daneben erhält das Evaluator-Objekt die Axiome der Wohlfundiertheit aus einer Datei (compileFile(...)) und die eigentliche Anfrage nach dem violation-Prädikat.

Die Deduktion wird angeworfen per stratify()-, evaluate()- und computeSubstitutions()-Botschaften. Im Szenario ist die Urpsrungsontologie wohlfundiert, so daß nun die Sicht darauf in Logik transformiert wird und mitsamt den Konsistenzaxiomen als zusätzliche Eingabe der Inferenzmaschine dient. Wiederum wird auf Erfüllung des violation-Prädikats geprüft und danach das Ergebnis der Deduktion von Klassenhierarchie, Domänen und Bildern der Relationen usw. in ein RDF-Model überführt (Logic2View(...)). Dieser Vorgang wiederholt sich nun bis an das Ende der Kaskade. Ist dabei nur einmal das violation-Prädikat

erfüllt, verweigert KAON-Views den Dienst und gibt die Substitutionen für violation(x,y) aus. x beinhaltet eine Fehlerbeschreibung und y den URI der Ressource. Schließlich speichert setModel() das nun vervollständigte RDF-Model der eigentlich zu ladenden Subontologie in einer lokalen Variable. Bevor dem Dienstnehmer jedoch die SubOntologyImpl-Instanz retourniert.wird, schließt sich an dieser Stelle die Materialisierung virtueller Instanzen an (siehe Kapitel 5.3).

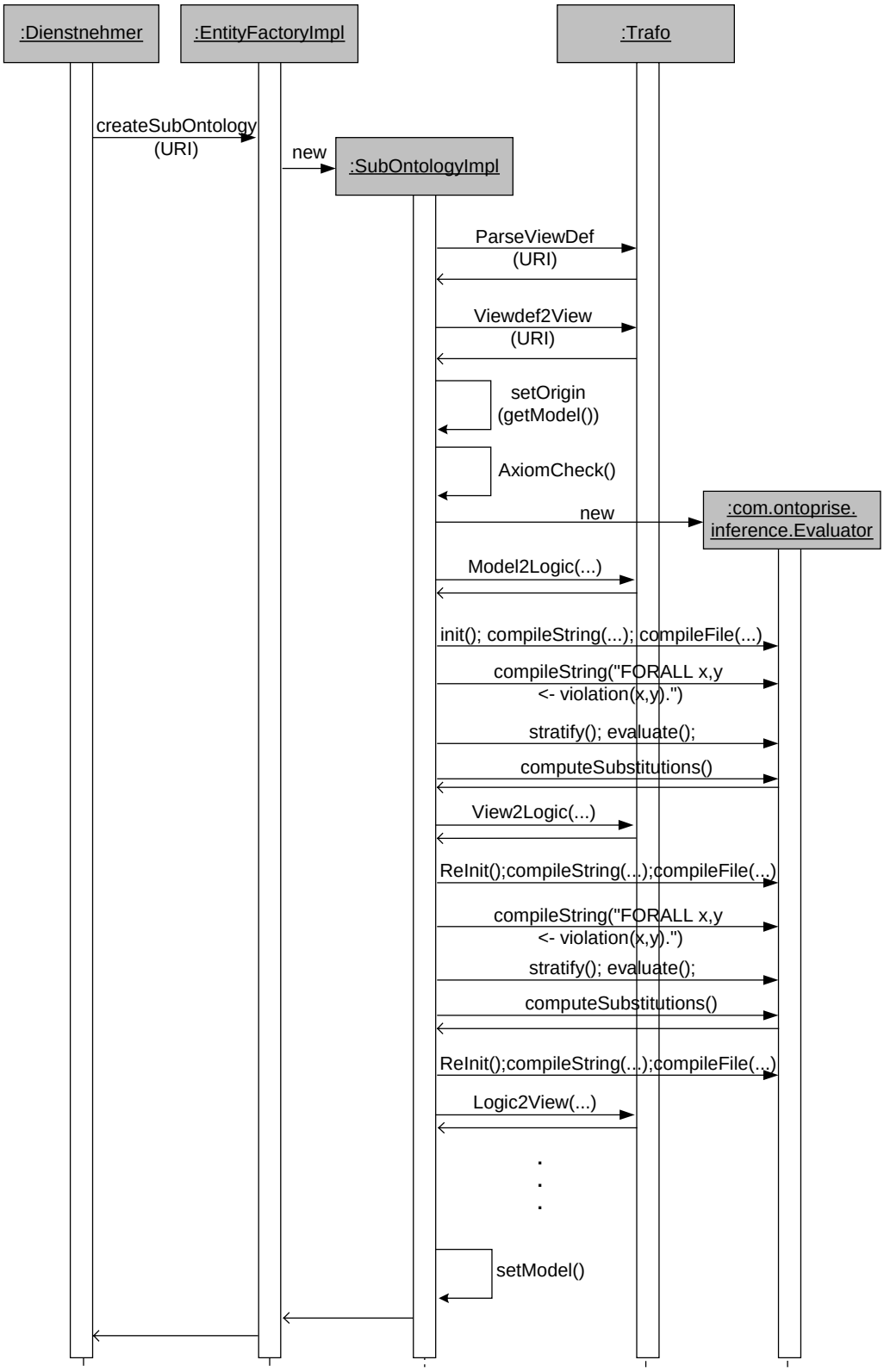


Abbildung 29: Typischer Ablauf

5.3 Extensionen virtueller Ressourcen

Wie, und vorallem mit Hilfe welcher Sprache, die Extensionen virtueller Ressourcen bestimmt werden, blieb bisher ausgeklammert. Die Frage tangiert zunächst $Syntax_1(x)$ bis $Syntax_4(x)$ - das waren Prädikate, die die syntaktische Korrektheit für Query-Ausdrücke spezifizieren. Betroffen sind Attributfilter ($Syntax_1$), virtuelle Attribute ($Syntax_2$), virtuelle Assoziationen ($Syntax_3$) und schließlich virtuelle Klassen ($Syntax_4$). Pragmatismus verbietet jeweils verschiedene Sprachen zu verwenden, so daß ein Prädikat $Syntax(x) = Syntax_1(x) = Syntax_2(x) = Syntax_3(x) = Syntax_4(x)$ anzustreben ist.

Die Entscheidung fiel letztendlich nicht auf eine RDF-Query-Sprache, sondern auf eine Extensionsbestimmung per Inferenz. Wie für den Axiomcheck kommt wieder com.ontoprise.inference.Evaluator zum Einsatz. Syntax(x) muß also erfüllt sein, wenn x der F-Logic-Syntax genügt. Diese Lösung hat den Vorteil, daß die Infrastruktur, d.h. die Transformation von der OO-Welt in Logik und zurück, bereits steht. Die Population virtueller Ressourcen kann damit intensional oder extensional bestimmt werden. Im ersten Falle werden per Horn-Regel neue Instanzen generiert, eine explizite Aufzählung der Instanzen oder eine Mixtur beider Varianten ist ebenfalls möglich. Abbildung 30 gibt einen ersten Überblick.

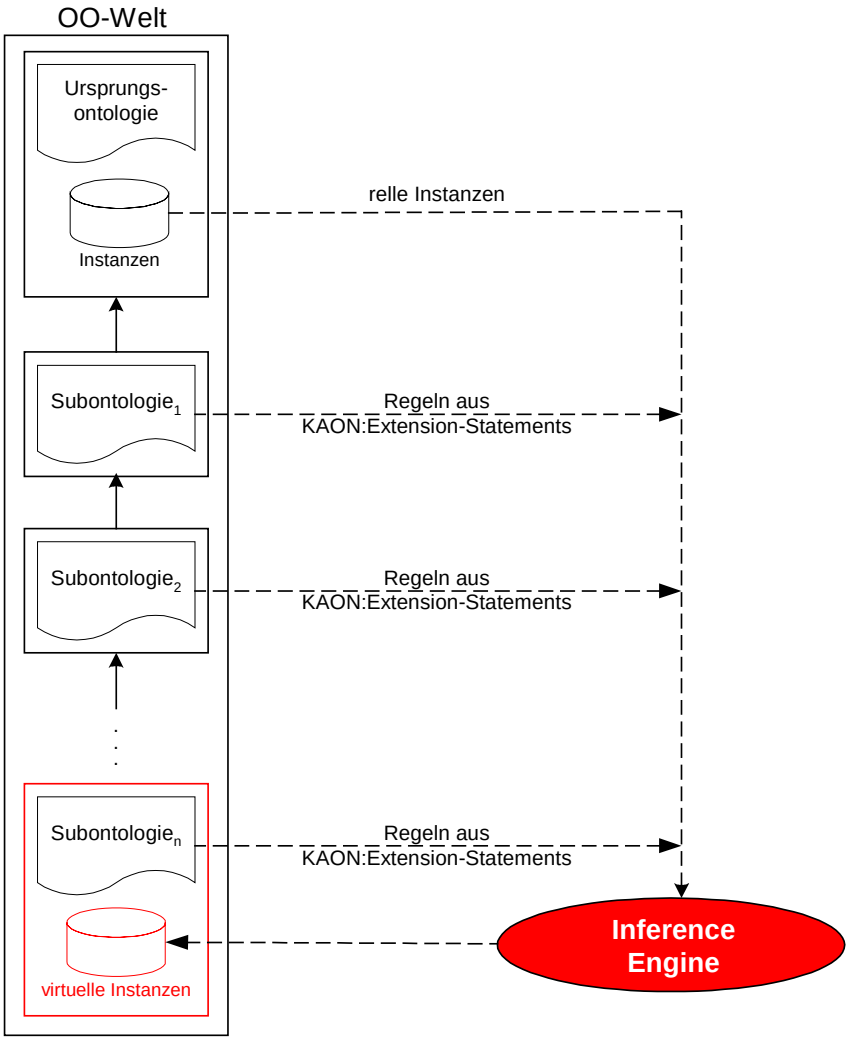


Abbildung 30: Überblick Extensionsbestimmung

KAON-Views materialisiert die virtuellen Instanzen nach dem erfolgreichem Axiomcheck. Dann sind bereits sämtliche Sichten einer Kaskade als RDF-Model vorhanden. Von der obersten Superontologie werden sämtliche reelle Instanzen extrahiert und in Logik transformiert. KAON-Extension-Statements, die später die Horn-Regeln beinhalten werden, aus allen Subontologien finden ebenfalls den Weg in die Inferenzmaschine. Die Deduktion liefert alle reelle Instanzen und neue, virtuelle Instanzen, die allesamt im korrespondierenden SubOntologyImpl-Objekt materialisiert werden. Es handelt sich hier also um eine Snapshot-Semantik. Nur für die Lebensdauer des SubOntologyImpl-Objekts bleiben die Instanzen erhalten. Es kann dabei durchaus vorkommen, daß Instanzen einer nicht importierten Ressource x den Weg in die Subontologie finden. Das ist allerdings nicht weiter schlimm, da dem Dienstnehmer die Ressource x per KAON-API unzugänglich bleibt.

Pluspunkte sammelt die Lösung mit Inferenzmaschine im Hinblick auf die Abgeschlossenheit. Es ist hier ohne weiteres möglich im Regelrumpf abermals virtuelle Ressourcen zu adressieren, die nicht nowendigerweise in der Sicht sein müssen. Es kann beispielsweise ein virtuelles Attribut gefiltert werden mit Domäne auf einer virtuellen Klasse, welche selektiert von einer weiteren, nicht importierten virtuellen Klasse usw. Eine derartige Flexibilität mit einer RDF-Query-Sprache zu erreichen, würde größere Klimmzüge nach sich ziehen. Außerdem erwähnenswert ist die Tatsache, daß mit den Regeln keine neuen URIs generiert werden. Es handelt sich hier also um einen objekterhaltenden Sichtenmechanismus (URI-Preservance).

5.3.1 Virtuelle Klassen

Die Sichtendefinitionssprache in Kapitel 3.3 deutete bereits an, daß man virtuelle Klassen anhand deren Extensionsbestimmung differenzieren kann. Es entstehen virtuelle Klassen durch Selektion, Differenz, Vereinigung, Durchschnitt, Ad-Lib-Abfragen oder durch explizite Aufzählung der Instanzen. Die Auflistung unten skizziert entsprechende Hornregeln zu einer virtuellen Klasse VC, jeweils mit den verschiedenen Möglichkeiten[6]. Dabei darf der Platzhalter <Class> wiederum virtuell sein.

Selection

```
FORALL X  statement(X, "rdfsfa:otype",<VC>) <-
          statement(X, "rdfsfa:otype",<Class>) AND
          <Selection-Teil>.
```

Difference

```
FORALL X  statement(X, "rdfsfa:otype",<VC>) <-
          statement(X, "rdfsfa:otype",<Class>) AND
          NOT EXISTS X0 (
               statement(X0, "rdfsfa:otype",<Class1>)
               AND unify(X,X0)) AND
          NOT EXISTS X1 (
               statement(X1, "rdfsfa:otype",<Class2>)
               AND unify(X,X1)) AND
          ...
```

[6] Innerhalb XML und der ASCII-Querysprache müssen Hochkommata mit Escape-Zeichen versehen sein und „<“ durch sein HTML-Äquivalent < ersetzt werden. Ein XML-Parser erwartet sonst einen End-Tag.

Union

```
FORALL X  statement(X, "rdfsfa:otype",<VC>) <-
          statement(X, "rdfsfa:otype",<Class1>) OR
          statement(X, "rdfsfa:otype",<Class2>) OR
          ...
```

Intersection

```
FORALL X  statement(X, "rdfsfa:otype",<VC>) <-
          statement(X, "rdfsfa:otype",<Class1>) AND
          statement(X, "rdfsfa:otype",<Class2>) AND
          ...
```

Arbitrary

```
FORALL X  statement(X, "rdfsfa:otype",<VC>) <-
          ...
```

Instances

```
statement(<I1>, "rdfsfa:otype",<VC>).
statement(<I2>, "rdfsfa:otype",<VC>).
...
```

Wünscht der Ontology Engineer nicht nur die direkten Instanzen einer Klasse, sondern auch Spezialisierungen davon zu berücksichtigen, so ist in allen obigen Fällen eine zweite Regel hinzuzufügen. Regeln mit gleichem Kopf entsprechen dabei bekanntlich einer Oder-Verknüpfung der Rümpfe.

Erwähnenswert sind außerdem die sogenannten Builtin-Prädikate der Inferenzmaschine. Diese sind insbesondere bei der Selektion, aber auch bei virtuellen Relationen und Filter, interessant. Sollte diese Funktionalität nicht genügen, kann man die Builtins beliebig erweitern. So wurde zum Beispiel das oben angesprochene Prädikat *Syntax*(x) eigens für KAON-Views programmiert.

Mit Vererbung

```
FORALL X       statement(X, "rdfsfa:otype",<VC>) <-
               statement(X, "rdfsfa:otype",<Class1>) ...
FORALL X,Y     statement(X, "rdfsfa:otype",<VC>) <-
               statement(X, "rdfsfa:otype",Y) AND
               subclass(Y,<Class1>) ...
```

Builtins

Arithmetisch:	+, -, *, /, sin, cos, tan, asin, acos, ceil, floor, exp, pow, rint, sqrt, round, max, min,
Vergleich:	is, equal, unify, less, lessorequal, greater, greaterorequal.
Typisierung:	isString, concat, constant2string, string2number, *Syntax*

5.3.2 Virtuelle Relationen

Die Extensionsbestimmung zu virtuellen Attributen und Assoziationen geschieht ähnlich wie im obigen Kapitel. Im Regelkopf stehen hier allerdings Instanzen einer Relation. Die Exempel unten greifen erneut das Beispielszenario aus Kapitel 3.4 auf. Das Attribut Euro berechnet sich aus dem Attributwert von Preis durch Multiplikation mit 0.49. Die virtuelle Assoziation Arbeitnehmer soll zutreffen, wenn Mitarbeitername und Name zweier Lohnabrechnungs- und Mitarbeiter-Instanzen identisch sind.

Virtuelle Attribute

```
FORALL X,Y,Z   statement(X,"Euro",Z) <-
               statement(X,"Preis",Y)
               AND (Z is (Y * 0.49)).
```

Virtuelle Assoziationen

```
FORALL X,Y,U,V statement(X,"Arbeitnehmer",Y) <-
               statement(X,"rdfsfa:otype","Lohnabrechnung")
               statement(Y,"rdfsfa:otype","Mitarbeiter") AND
               statement(X,"Mitarbeitername",U) AND
               statement(Y,"Name",V) AND
               unify(U,V).
```

5.3.3 Filter

Auch die Attributfilter wurden mit der Inferenzmaschine realisiert. Der Leser möge sich erneut das Beispiel aus Kapitel 3.4 vergegenwärtigen. Dort sollten nur erwachsene Personen in der Subontologie sichtbar bleiben. Ein Attributfilter spezifiziert zunächst die gewünschten Statements per Anfrage. KAON-Views nimmt das Ergebnis entgegen und sorgt dafür, daß die restlichen relevanten Instanzen entfernt werden.

Attributfilter

```
     FORALL X,Y,Z  <-
          statement(X,Y,Z) AND unify(Y,"rdfsfa:otype")
          AND EXISTS U statement(X, "Alter", U)
          AND greaterorequal(U,18).
```

Man beachte, daß man eine ähnliche Funktionalität auch durch virtuelle Klassen erreichen kann. Etwa durch Selektion auf Person und Subklassen mit entsprechendem Selection-Teil. Dabei bleibt die Klassenhierarchie in der Subontologie nicht die alte, was unter Umständen unerwünscht sein kann.

Assoziationsfilter benötigen keine F-Logic-Regeln, sondern kommen rein mit KAON-Primitiven zurecht. Dementsprechend konzentriert sich deren Realisierung rein auf die OO-Seite von KAON-Views (siehe Kapitel 5.1.1).

6 Ausblick

Obwohl umfangreich und funktionsfähig, stellt diese Arbeit nur das Fundament für einen umfassenden Sichtenmechanismus dar. Weitere Studien- und Diplomarbeiten werden nötig sein, um das Gesamtwerk abzurunden. Der Ausblick listet abschließend offene Punkte und Probleme auf der Agenda. Dabei wird getrennt zwischen implementierungsrelevanten und konzeptionellen Gesichtspunkten.

6.1 Implementierung

- *GUI*
  Von Beginn an stand der Wunsch nach grafischer Sichtendefinition auf der Agenda. Wie bereits im Entwurf erwähnt, bietet sich dafür KAON-SOEP, das Simple Ontology Browser and Editor Plug-In, an. Es sollte mit virtuellen Ressourcen umgehen und auch den Axiomcheck durchführen können. Abbildung 31 illustriert schematisch die Vorgehensweise der Integration. Der Ontology Engineer übernimmt per grafischer Interaktion Konzepte und Relationen (1) und definiert zusätzlich virtuelle Ressourcen nach Belieben (2). Die Axiomprüfung sichert die Konsistenz und inferiert Klassenhierarchie, sowie Relationenhierarchie, Domänen und Bilder (3). Die Transformation in eine Sichtendefinition sowie Speicherung schließen sich an (4).

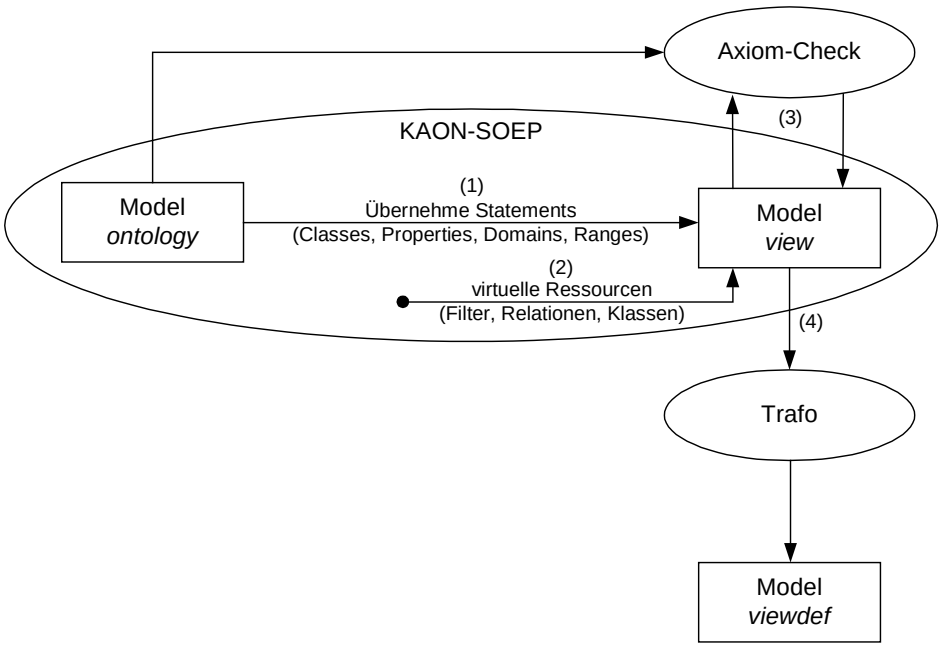


Abbildung 31: KAON-SOEP Integration

- *KAON-Views Plug-In*
  Im Entwurf bereits angesprochen, soll dieses avisierte Plug-In zur komfortablen Konfiguration der Axiome dienen. Die Pfade der F-Logic-Dateien können spezifiert, die Axiome verändert oder erweitert werden. Wünscht der Ontology Engineer, daß Ontologien KAON-weit wohlfundiert sein sollten, so könnte er dieses mit dem Plug-In erledigen.

- *Effizienz*
  Zugunsten eines schnelleren Vorankommens standen effiziente Algorithmen nicht an vorderster Stelle. Sehr häufig anzutreffen sind verschachtelte Suchschleifen auf RDF-Modellen. Das RDF-API führt dazu allerdings Hash-Tabellen, so daß sich der Aufwand in Grenzen hält.

6.2 Konzeptionelles

- *Update-Problematik*
  Gänzlich unter den Tisch fielen Fragen bezüglich Änderungen an einer Sicht. Derzeit bieten SubOntologyImpl und alle anderen Klassen dazu nicht die Möglichkeit. In sämtlichen relevanten Methoden wird geprüft auf (ontology instanceof SubOntologyImpl) und eine NotSupportedException geworfen. Besonders interessant ist das Einfügen von Instanzen, besser bekannt als View-Update-Problem, sowie Änderungen an Schemadaten. Dieselbe Problematik ist selbst bei objektorientierten Datenbanken noch Gegenstand der Forschung, so daß hierfür gewiß noch eine Diplomarbeit zu spendieren ist.

- *Multiple Ursprungsontologien*
  Bisher verfügt KAON-Views zwar über die Möglichkeit zur Kaskadierung, d.h. eine Sicht kann wiederum auf einer Sicht basieren, allerdings ist nur jeweils eine Superontologie erlaubt. Die Mächtigkeit, jedoch in überproportionalen Maße auch der Aufwand, steigt erheblich, sobald man mehrere Ursprungsontologien zuläßt. Aus der bisherigen linearen Kaskade würde also eine Hierarchie oder gar ein Graph. Sowohl die Axiomatik, als auch das KAON-API, müssten dazu neu überdacht und wohl verändert werden.

- *Imaginäre Instanzen*
  Sowohl Extensionen zu virtuellen Klassen und Assoziationen, als auch Werte zu virtuellen Attributen, werden in einer ersten Lösung zur Laufzeit berechnet. Im Sinne der Effizienz sind imaginäre Instanzen, d.h. man wertet die Query-Ausdrücke einmalig, z.B. im Konstruktor zu SubOntologyImpl, aus und materialisiert sie dann im korrespondierenden RDF-Model. Sollen später allerdings Änderungen an einer Sicht möglich sein, vervielfacht sich der Aufwand erneut, denn Instanzen müssten an mehreren Stellen aktualisiert werden.

- *Typisierung von Attributwerten*
  Weder RDFS, noch RDFS(FA) sehen eine Typisierungen für Attributwerte vor. Das wird die Realisierung von virtuellen Attributen erheblich erschweren. In den Berechnungsausdrücken stehen arithmetische Operatoren, Zeichenkettenmodifikation uvm. Eine Typisierung würde hier für mehr Robustheit sorgen. Anstelle des bisher einzigen Typs Literal, sollten gängige Typen aus der Welt der Programmiersprachen treten.

- *Gleichheitsproblematik*
  Die behandelten Vokabulare bieten nicht die Möglichkeit die Gleichheit auszudrücken. Man könnte dieses künstlich erreichen, in dem man Zyklen in der Konzepthierarchie zuläßt. Eine solche Mächtigkeit wäre besonders sinnvoll, wenn mehrere Ursprungsontologien ins Spiel kommen und gleiche Konzepte unterschiedliche URIs besitzen. Per Mengeninklusionssemantik ist dieses leicht zu demonstrieren. Sei Person $\subseteq$ Mensch und Mensch $\subseteq$ Person, so gilt Person = Mensch. Das entspricht aber genau Person rdfsfa:osubClassOf Mensch und Mensch rdfsfa:osubClassOf Person.

- *Andere Vokabulare*
  Die vorliegende Arbeit betrachtet lediglich Ontologien, die mit Hilfe des RDFS(FA)-Vokabulars definiert sind. Unberücksichtigt blieben neben dem untauglichen RDFS andere Vokabulare, allen voran DAML+OIL. Eine notwendige Anforderung an das Vokabular ist eine fixe Modellierungsarchitektur. Der Sichtenmechanimus wäre sonst zum Scheitern verurteilt, wie bereits in Kapitel 4.2 angesprochen. Die Axiomatik muss in jedem Falle an das neue Vokabular angepaßt werden. KAON-Views müßte die Inferenzmaschine in Abhängigkeit des Vokabulars mit verschiedenen Axiom-Files füttern.

- *Unterschiedliche Vokabulare*
  Eine ungeklärte Frage ist ebenfalls die Handhabung von unterschiedlichen Vokabularen innerhalb einer Sichtenkaskade. Im jetzigen Zustand würde eine NotSupportedException geworfen, sobald das Vokabular einer Superontologie von KAON und RDFS(FA) abweicht. Eventuell möchte man später eine Mixtur aus mehreren zulassen, um beispielsweise die Mächtigkeit anderer Vokabulare nutzen zu können.

- *Andere Inferenzmaschine*
  Statt com.ontoprise.inference.Evaluator eine andere Inferenzmaschine zu benutzen, würde ebenfalls eine Anpassung der Axiom-Files nach sich ziehen. Ob sich der Aufwand lohnt ist fraglich, denn ein Dienstnehmer bekommt vom Axiomcheck und damit von der Inferenzmaschine nichts mit. Außerdem beinhaltet com.ontoprise.inference.Evaluator ein eigens programmiertes Prädikat Syntax(x), das als Builtin realisiert wurde und das Enthaltensein von x in der F-Logic-Sprache sichert.

Anhang

I. Axiome in F-Logic

```
//---------------
//HILFSFUNKTIONEN
//---------------

//Hilfsfunktion SC Alle Subkonzepte zu einem Konzept
FORALL C, D       isSC(C,D) <- statement(C,"rdfsfa:osubClassOf",D).
FORALL C, X, D    isSC(C,D) <- isSC(C,X) AND isSC(X,D).

//Hilfsfunktion SC_ Alle Subkonzepte zu einem Konzept in der Sicht
FORALL C, D       isSC_(C,D) <- isC_(C,"rdfsfa:osubClassOf",D).
FORALL C, X, D    isSC_(C,D) <- isSC_(C,X) AND isSC_(X,D).
FORALL C, D       subclass(C,D) <- isSC_(C,D).

//Hilfsfunktion SC0 Übernommene Subkonzepte zu einem Konzept
FORALL C,D,X,Y    isSC0(C,D) <- isSC(C,D) AND isC_(C,"rdfsfa:ltype",X)
                  AND NOT EXISTS U (isSC0(U,D) AND isSC0(C,U)).
FORALL C,D,X,Y    help5(C,D) <- isSC(C,D) AND isC_(C,"rdfsfa:ltype",X) AND
                  isC_(D,"rdfsfa:ltype",Y).
FORALL C,D        isSC02(C,D) <- help5(C,D) AND NOT EXISTS U (help5(U,D)
                  AND help5(C,U)).
FORALL C,D,X,Y    isSC03(C,D)  <- isSC_(C,D) AND isC(C,"rdfsfa:ltype",X)
                  AND isC_(D,"rdfsfa:ltype",Y).

//Hilfsfunktion SC1 Nächste übernommene Superklassen
FORALL X,Y,U,V    help3(X,Y) <- isSC(Y,X) AND isC_(X,"rdfsfa:ltype",U) AND
                  isC_(Y,"rdfsfa:ltype",V).
FORALL X,Y        isSC1(X,Y) <- help3(X,Y) AND NOT EXISTS Z (isSC(Z,X) AND
                  help3(Z,Y)).

//Hilfsfunktion SC2 Nächste Superklasse
FORALL C, D       isSC2(C,D) <- statement(C,"rdfsfa:osubClassOf",D).
FORALL C, X, D    isSC2(C,D) <- isSC(C,X) AND isSC(X,D).

//Hilfsfunktion SP Alle Subproperties zu einem Property
FORALL C, D       isSP(C,D) <- statement(C,"rdfsfa:osubPropertyOf",D).
FORALL C, X, D    isSP(C,D) <- isSP(C,X) AND isSP(X,D).

//Hilfsfunktion SP_ Alle Subkonzepte zu einem Konzept in der Sicht
FORALL C, D       isSP_(C,D) <- isP_(C,"rdfsfa:osubPropertyOf",D).
FORALL C, X, D    isSP_(C,D) <- isSP_(C,X) AND isSP_(X,D).

//Hilfsfunktion SP0 Übernommene Subproperties zu einem Property
FORALL C,D,X,Y    isSP0(C,D) <- isSP(C,D) AND isP_(C,"rdfsfa:ltype",X) AND
                  isP_(D,"rdfsfa:ltype",X) AND NOT EXISTS U (isSP0(U,D) AND
                  isSP0(C,U)).



//---------------------------
// Ontologie O = (n, C, P, A)
//---------------------------

//Def. Klassenhierarchie C
FORALL X    isC(X,"rdfsfa:ltype","rdfsfa:LClass") <-
            statement(X,"rdfsfa:ltype","rdfsfa:LClass").
FORALL X    isC(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualClass") <-
            statement(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualClass").
FORALL X, Y isC(X,"rdfsfa:osubClassOf",Y) <-
            statement(X,"rdfsfa:osubClassOf",Y).
```

```
//Def. Relationenhierarchie P
FORALL X    isP(X,"rdfsfa:ltype","rdfsfa:LProperty") <-
            statement(X,"rdfsfa:ltype","rdfsfa:LProperty").
FORALL X    isP(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute") <-
            statement(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute").
FORALL X    isP(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation") <-
            statement(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation").
FORALL X, Y isP(X,"rdfsfa:osubPropertyOf",Y) <-
            statement(X,"rdfsfa:osubPropertyOf",Y).

//Def. Zusicherungen A
FORALL X, Y, Z    isA(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"rdfsfa:odomain") AND EXISTS A,B (isC(X,A,B) OR
                  isP(X,A,B)).
FORALL X, Y, Z    isA(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"rdfsfa:orange") AND  EXISTS A,B (isC(X,A,B) OR
                  isP(X,A,B)).
FORALL X, Y, Z    isA(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#label") AND
                  EXISTS A,B (isC(X,A,B) OR isP(X,A,B)).
FORALL X, Y, Z    isA(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#seeAlso") AND
                  EXISTS A,B (isC(X,A,B) OR isP(X,A,B)).
FORALL X, Y, Z    isA(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#isDefinedBy")
                  AND EXISTS A,B (isC(X,A,B) OR isP(X,A,B)).
FORALL X, Y, Z    isA(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#comment") AND
                  EXISTS A,B (isC(X,A,B) OR isP(X,A,B)).
```

(Ax1) *Existenz einer Wurzelklasse* Kapitel 4.2 Seite 37

```
FORALL X,U  help("Class without superclass",X) <-
            isC(X,"rdfsfa:ltype",U) AND NOT EXISTS C (isSC(X,C)).
FORALL X,Y  violation("Several root classes",X) <-
            help("Class without superclass",X) AND help("Class without
            superclass",Y) AND NOT unify(X,Y).
```

(Ax2) *Zyklenfreie Klassenhierarchie* Kapitel 4.2 Seite 37

```
FORALL D    violation("Class-hierarchy cycle violation",D) <-
            EXISTS D (isSC(D,D)).
```

(Ax3) *Zyklenfreie Relationenhierarchien* Kapitel 4.2 Seite 37

```
FORALL D    violation("Relation-hierarchy cycle violation",D) <-
            EXISTS D (isSP(D,D)).
```

(Ax4) *Korrekte Instantiierung der Subklassen* Kapitel 4.2 Seite 38

```
FORALL X,Y  violation("No definition for class",X) <-
            isC(X,"rdfsfa:osubClassOf",Y) AND NOT EXISTS Z
            isC(X,"rdfsfa:ltype",Z).
```

(Ax5) *Korrekte Instantiierung der Subrelationen* Kapitel 4.2 Seite 38

```
FORALL X,Y  violation("No definition for property",X) <-
            isC(X,"rdfsfa:osubPropertyOf",Y)
            AND NOT EXISTS Z isC(X,"rdfsfa:ltype",Z).
```

(Ax6) *Korrekte Instantiierung der Relationen* Kapitel 4.2 Seite 38

```
FORALL X,U  violation("Domain missing for relation",X) <-
            isP(X,"rdfsfa:ltype",U) AND NOT EXISTS A,B,C (isA(A,B,C) AND
            unify (A,X) AND unify(B,"rdfsfa:odomain")).
FORALL X,U  violation("Range missing for relation",X) <-
            isP(X,"rdfsfa:ltype",U) AND NOT EXISTS A,B,C (isA(A,B,C) AND
            unify (A,X) AND unify(B,"rdfsfa:orange")).
```

```
FORALL X,U  violation("Only 1 range allowed",X) <- isP(X,"rdfsfa:ltype",U)
            AND EXISTS A,B,C,I,J,K ( (isA(A,B,C) AND unify (A,X) AND
            unify(B,"rdfsfa:orange") AND isA(I,J,K) AND unify (I,X) AND
            unify(J,"rdfsfa:orange")) AND NOT unify(C,K)).
FORALL X,Y,U      help7("No real domain",X) <- isP(X,"rdfsfa:ltype",U)
            AND isA(X,"rdfsfa:odomain",Y) AND NOT
            isC(Y,"rdfsfa:ltype","rdfsfa:LClass").
FORALL X,Y,U      help7("No virtual domain",X) <- isP(X,"rdfsfa:ltype",U)
            AND isA(X,"rdfsfa:odomain",Y) AND NOT
            isC(Y,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualClass").
FORALL X    violation("Domain not part of ontology",X) <-
            help7("No real domain",X) AND help7("No virtual domain",X) AND
            help7("No virtual domain in view",X).
FORALL X,Y,U      help7("No real range",X) <- isP(X,"rdfsfa:ltype",U)
            AND isA(X,"rdfsfa:orange",Y) AND NOT
            isC(Y,"rdfsfa:ltype","rdfsfa:LClass").
FORALL X,Y,U      help7("No virtual range",X) <- isP(X,"rdfsfa:ltype",U)
            AND isA(X,"rdfsfa:orange",Y) AND NOT
            isC(Y,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualClass").
FORALL X,U  help7("No literal range",X) <- isP(X,"rdfsfa:ltype",U)
            AND NOT isA(X,"rdfsfa:orange","http://www.w3.org/TR/WD-rdf-
            schema#Literal").
FORALL X    help7("No virtual attribute",X) <-
            isP(X,"rdfsfa:ltype","rdfsfa:LProperty").
FORALL X    help7("No virtual attribute",X) <-
            isP(X,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation").
FORALL X    violation("Range not part of ontology",X) <- help7("No real
            range",X) ANDhelp7("No virtual range",X) AND help7("No literal
            range",X) AND help7("No virtual attribute",X).
```

(Ax7) *Domänen von Subrelationen* Kapitel 4.2 Seite 38

```
FORALL X,Y,Z      domains(X,Z) <- isP(X, "rdfsfa:osubPropertyOf", Y) AND
                  isA(Y,"rdfsfa:odomain",Z).
FORALL X,Y,Z,C    domains(X,C) <- isP(X,"rdfsfa:osubPropertyOf",Y) AND
                  isA(Y,"rdfsfa:odomain",Z) AND isSC(C,Z).
FORALL X,Y,Z      violation("Incorrect domain of subrelation",X) <-
                  isP(X,"rdfsfa:osubPropertyOf",Y) AND
                  isA(X,"rdfsfa:odomain",Z) AND NOT domains(X,Z).
```

(Ax8) *Bilder von Subrelationen* Kapitel 4.2 Seite 38

```
FORALL X,Y,C,D    violation("Incorrect range of subrelation",X) <-
                  ( isP(X,"rdfsfa:osubPropertyOf",Y) AND
                    isA(X,"rdfsfa:orange",C) AND
                    isA(Y,"rdfsfa:orange",D) AND NOT
                    isC(C,"rdfsfa:osubClassOf",D)) AND
                  ( isP(X,"rdfsfa:osubPropertyOf",Y) AND
                    isA(X,"rdfsfa:orange",C) AND
                    isA(Y,"rdfsfa:orange",D) AND NOT
                    unify(D,"http://www.w3.org/TR/WD-rdf-schema#Literal")).


FORALL X    root(X, "rdfsfa:ltype", "rdfsfa:LClass") <-
            help("Class without superclass",X).

//---------------------------------
// Subontologie SO = (n, C_, P_, A_)
//---------------------------------

//Def. Klassenhierarchie C_
FORALL X, Y, Z    isC_(X,Y,Z) <- isC_neu(X,Y,Z).

//Def. Relationenhierarchien P_
FORALL X, Y, Z    isP_(X,Y,Z) <- isP_neu(X,Y,Z).

//Def. Zusicherungen A_
FORALL X, Y, Z    isA_(X,Y,Z) <- isA_neu(X,Y,Z).
```

(Ax9) $C' \subseteq C \cup C^{neu}$ Kapitel 4.3 Seite 39

```
FORALL X,Y,Z      violation("Class is not part of superontology",X) <-
                  isC_(X,"rdfsfa:ltype","rdfsfa:LClass") AND NOT
                  isC(X,"rdfsfa:ltype","rdfsfa:LClass").
FORALL X,Y,Z      violation("Class is not part of superontology",X) <-
                  isC_(X,"rdfsfa:ltype","http://kaon.aifb.uni-
                  karlsruhe.de/kaon#virtualClass") AND NOT
                  isC_neu(X,"rdfsfa:ltype","http://kaon.aifb.uni-
                  karlsruhe.de/kaon#virtualClass") AND NOT
                  isC(X,"rdfsfa:ltype","http://kaon.aifb.uni-
                  karlsruhe.de/kaon#virtualClass").
```

(Ax10) $P' \subseteq P \cup P^{neu}$ Kapitel 4.3 Seite 39

```
FORALL X, Y, Z    violation("Relation is not part of superontology",X) <-
                  isP_(X,"rdfsfa:ltype","rdfsfa:LProperty") AND NOT
                  isP(X,"rdfsfa:ltype","rdfsfa:LProperty").
```

(Ax11) $A' \subseteq A \cup A^{neu}$ Kapitel 4.3 Seite 40

```
//implizit gegeben
```

(Ax12) *Übernahme Wurzelklasse* Kapitel 4.3 Seite 40

```
FORALL X,Y  help2("Class without superclass",X) <-
            isC_(X,"rdfsfa:ltype",Y) AND NOT EXISTS C (isSC_(X,C)).
FORALL X,Y  violation("Several root classes in subontology",X) <-
            help2("Class without superclass",X) AND help2("Class without
            superclass",Y) AND NOT unify(X,Y).
```

(Ax13) *Abgeschlossenheit der Zusicherungen* Kapitel 4.3 Seite 40

```
FORALL X, Y, Z    isA_(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#comment") AND
                  EXISTS A,B (isC_(X,A,B) OR isP_(X,A,B)).
FORALL X, Y, Z    isA_(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#label") AND
                  EXISTS A,B (isC_(X,A,B) OR isP_(X,A,B)).
FORALL X, Y, Z    isA_(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#seeAlso") AND
                  EXISTS A,B (isC_(X,A,B) OR isP_(X,A,B)).
FORALL X, Y, Z    isA_(X,Y,Z) <- statement(X,Y,Z) AND
                  unify(Y,"http://www.w3.org/TR/WD-rdf-schema#isDefinedBy")
                  AND EXISTS A,B (isC_(X,A,B) OR isP_(X,A,B)).
```

(Ax14) *Klassenhierarchie* Kapitel 4.3 Seite 40

```
//siehe (Ax24)
```

(Ax15) *Domänen* Kapitel 4.3 Seite 40

```
FORALL P,C        isA_(P,"rdfsfa:odomain",C) <-
                  isP_(P,"rdfsfa:ltype","rdfsfa:LProperty") AND
                  isA(P,"rdfsfa:odomain",C) AND
                  isC_(C,"rdfsfa:ltype","rdfsfa:LClass").
FORALL P,C,X      isA_(P,"rdfsfa:odomain",X) <-
                  isP_(P,"rdfsfa:ltype","rdfsfa:LProperty") AND
                  isA(P,"rdfsfa:odomain",C) AND isSC0(X,C) AND NOT
                  isC_(C,"rdfsfa:ltype","rdfsfa:LClass").
FORALL P          violation("Domain is missing or not inferable",P) <-
                  isP_(P,"rdfsfa:ltype","rdfsfa:LProperty")
                  AND NOT EXISTS D isA_(P,"rdfsfa:odomain",D).
```

(Ax16) *Ranges* Kapitel 4.3 Seite 41

```
FORALL P,C,U,V    isA_(P,"rdfsfa:orange",C) <- isP_(P,"rdfsfa:ltype",V) AND
                  isA(P,"rdfsfa:orange",C) AND isC_(C,"rdfsfa:ltype",U).
FORALL P          isA_(P,"rdfsfa:orange","http://www.w3.org/TR/WD-rdf-
                  schema#Literal") <-
```

```
                    isP_(P,"rdfsfa:ltype","rdfsfa:LProperty") AND
                    isA(P,"rdfsfa:orange","http://www.w3.org/TR/WD-rdf-
                    schema#Literal").
FORALL P,X,C,D      violation("Range not inferable, more than one
                    subclass",P) <- isP_(P,"rdfsfa:ltype","rdfsfa:LProperty")
                    AND isA(P,"rdfsfa:orange",X) AND NOT
                    isC_(X,"rdfsfa:ltype","rdfsfa:LClass") AND isSC0(C,X) AND
                    isSC0(D,X) AND NOT unify(C,D).
FORALL P,C,X        isA_(P,"rdfsfa:orange",C) <-
                    isP_(P,"rdfsfa:ltype","rdfsfa:LProperty") AND
                    isA(P,"rdfsfa:orange",X) AND NOT

                    isC_(X,"rdfsfa:ltype","rdfsfa:LClass") AND NOT
                    violation("Range not inferable, more than one
                    subclass",P) AND isSC0(C,X).
FORALL P            violation("Range missing or not inferable",P) <-
                    isP_(P,"rdfsfa:ltype","rdfsfa:LProperty") AND
                    NOT EXISTS  D isA_(P,"rdfsfa:orange",D).
```

(Ax17) *Relationenwald* Kapitel 4.3 Seite 42

```
FORALL P,Q  isP_(P,"rdfsfa:osubPropertyOf",Q) <- isSP0(P,Q).
```

(Ax18) *Abgeschlossenheit Filter* Kapitel 4.3.1 Seite 42

```
FORALL F    violation("Domain missing, Filter",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Attributefilter") AND NOT EXISTS D
            isA_(F,"rdfsfa:odomain",D).
FORALL F    violation("Domain missing, Filter",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Associationfilter")
            AND NOT EXISTS D isA_(F,"rdfsfa:odomain",D).
FORALL F,D  violation("Domain must be part of subontology, Filter",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Attributefilter") AND
            isA_(F,"rdfsfa:odomain",D) AND NOT EXISTS U
            isP_(D,"rdfsfa:ltype",U).
FORALL F,D  violation("Domain must be part of subontology, Filter",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Associationfilter") AND
            isA_(F,"rdfsfa:odomain",D) AND NOT EXISTS U
            isP_(D,"rdfsfa:ltype",U).
FORALL F,C  violation("Range must be part of subontology, Filter",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Associationfilter")
            AND isA_(F,"rdfsfa:orange",C) AND NOT EXISTS U
            isC_(C,"rdfsfa:ltype",U).
FORALL F,D,Pviolation("Range of associationfilter must be equal to domain
            or range of association",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Associationfilter")
            AND isA_(F,"rdfsfa:orange",D) AND isA_(F,"rdfsfa:odomain",P)
            AND NOT isA_(P,"rdfsfa:orange",D) AND NOT EXISTS C
            (isA_(P,"rdfsfa:odomain",C) AND unify(C,D)).
FORALL F,D,Pviolation("Range of associationfilter must be equal to domain
            or range of association",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Associationfilter")
            AND isA_(F,"rdfsfa:orange",D) AND isA_(F,"rdfsfa:odomain",P)
            AND NOT isA_(P,"rdfsfa:odomain",D) AND NOT EXISTS C
            (isA_(P,"rdfsfa:orange",C) AND unify(C,D)).
FORALL F,R  violation("Syntax error in filterexpression",F) <-
            isA_(F,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#Attributefilter")
            AND isA_(F,"rdfsfa:orange",R) AND NOT Syntax(R).
```

(Ax19) *Attributfilter Implementierung* Kapitel 4.3.1 Seite 42

```
//Implementierung -> siehe KAON-API
```

(Ax20) *Assoziationsfilter Implementierung* Kapitel 4.3.1 Seite 42

```
//Implementierung -> siehe KAON-API
```

(Ax21) *Virtuelle Attribute* Kapitel 4.3.2 Seite 43

```
FORALL VA   violation("Domain missing for virtual attribute",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute") AND NOT EXISTS C
            isA_(VA,"rdfsfa:odomain",C).
FORALL VA   violation("Range of virtual attribute must be
            rdfs:Literal",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute")
            AND NOT isA_(VA,"rdfsfa:orange","http://www.w3.org/TR/WD-rdf-
            schema#Literal").
FORALL VA,C violation("Domain must be part of subontology for virtual
            attribute",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute")
            AND isA_(VA,"rdfsfa:odomain",C) AND NOT EXISTS U
            isC_(C,"rdfsfa:ltype",U).
FORALL VA   violation("Extension (containing query) missing for virtual
            attribute",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute")
            AND NOT EXISTS C isA_(VA,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",C).
FORALL V, R, S   violation("Not more than one extension-statement allowed
            for virtual resource",V) <-
            isA_(V,"http://kaon.aifb.uni-karlsruhe.de/kaon#Extension",R)
            AND isA_(V,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",S) AND NOT unify(R,S).
FORALL VA,Q violation("Syntax Error in query-expression",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAttribute")
            AND isA_(VA,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",Q) AND NOT Syntax(Q).
```

(Ax22) *Virtuelle Assoziationen* Kapitel 4.3.2 Seite 43

```
FORALL VA   violation("Domain missing for virtual association",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation") AND NOT EXISTS C
            isA_(VA,"rdfsfa:odomain",C).
FORALL VA   violation("Range missing for virtual association",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation")
            AND NOT EXISTS C isA_(VA,"rdfsfa:orange",C).
FORALL VA   violation("Extension (containing query) missing for virtual
            association",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation")
            AND NOT EXISTS C isA_(VA,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",C).
FORALL VA,C violation("Domain must be part of subontology for virtual
            association",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation")
            AND isA_(VA,"rdfsfa:odomain",C) AND NOT EXISTS U
            isC_(C,"rdfsfa:ltype",U).
FORALL VA,C violation("Range must be part of subontology for virtual
            association",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation")
            AND isA_(VA,"rdfsfa:orange",C) AND NOT EXISTS U
            isC_(C,"rdfsfa:ltype",U).
FORALL VA,Q violation("Syntax Error in query-expression",VA) <-
            isP_(VA,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualAssociation")
            AND isA_(VA,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",Q) AND NOT Syntax(Q).
```

(Ax23) *Korrekte Instantiierung von virtuellen Klassen* Kapitel 4.3.3 Seite 44

```
FORALL VC   violation("Extension (containing query) missing for virtual
            class",VC) <-
            isC_(VC,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualClass")
            AND NOT EXISTS C isA_(VC,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",C).
FORALL VC,Q violation("Syntax Error in query-expression",VC) <-
            isC_(VC,"rdfsfa:ltype","http://kaon.aifb.uni-
            karlsruhe.de/kaon#virtualClass")
            AND isA_(VC,"http://kaon.aifb.uni-
            karlsruhe.de/kaon#Extension",Q) AND NOT Syntax(Q).
```

(Ax24) *Klassenhierarchie* Kapitel 4.3.3 Seite 44

```
FORALL VC, C, D   isC_(VC,"rdfsfa:osubClassOf",D) <-
                  isC_(C,"rdfsfa:osubClassOf",VC) AND isSC1(D,C)
                  AND isC_neu(VC,"rdfsfa:ltype","http://kaon.aifb.uni-
                  karlsruhe.de/kaon#virtualClass").
FORALL X,Y        isC_(X,"rdfsfa:osubClassOf",Y) <-
                  isSC(X,Y) AND isSC02(X,Y).
```

(Ax25) *Attributvererbung* Kapitel 4.3.3 Seite 45

```
FORALL P,VC,C     help4(VC,C,P) <- isC_(C,"rdfsfa:osubClassOf",VC)
                  AND isA_(P,"rdfsfa:odomain",C).
FORALL VC,C,P     isA_(P,"rdfsfa:odomain",VC) <-
                  help4(VC,C,P) AND NOT EXISTS D,Q (help4(VC,D,Q)
                  AND NOT unify(Q,P)).
```

II. Abbildungsverzeichnis

III. Literatur

[AB91] Serge Abiteboul, Anthony Bonner. *Objects and Views*. SIGMOD Conference 1991:238-247

[AHV95] Abiteboul, Hull, Vianu. *Foundations of Databases*, Addison-Wesley Publishing Company, 1995.

[Ber92] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In *Advances in DB-Technology, Proc. Intl. Conf. on Extending Database Technology (EDBT*), number 580 in Lecture Notes in Computer Science, pages 136–151, Vienna, Austria, March 1992. Springer.

[BFN95] R. Busse, P. Frankhauser, and E. J. Neuhold. Federated schemata in ODMG. In *2nd Int. East/West DB Worksho*p, 1995.

[CAKBC] Tony Clark, Andy Adams, Stuart Kent, Steve Brodsky, Steve Cook. *A Feasability Study in Rearchitecting UML as a Family of Languages using a Precise OO Metamodeling Approach*

[CHHGS] D. Connolly, F. van Hamelen, I. Horrocks, D. McGuiness, L. Stein. DAML+OIL Reference Description. World Wide Web Consortium. http://www.w3.org/TR/daml+oil-reference. March 2001

[CK01] Wolfram Conen and Reinhold Klapsing. Logical interpretations of RDFS - a compatibility guide. http://nestroy.wi-inf.uni-essen.de/rdf/newinterpretation/, December 2001.

[DBSA] S. Decker, D. Brickley, J. Saarela, J. Angele. A Query and Inference Service for RDF. In: *QL´98 – The Query Languages Qorkshop*. http://www.w3.org/TandS/QL/QL98/pp/queryservice.html

[Dob97] M. Dobrovnik. *Externe Schemata in objekt-orientierten Datenbankmanagementsysteme; Logische Datenunabhängigkeit durch Änderungen über Sichten*, volume 25 of *DISDBI*S. Infix Verlag, 1997.

[GGMS] Dieter Gluche, Torsten Grust, Christof Mainberger, Marc H. Scholl. Incremental Updates for Materialized OQL Views. DOOD 1997: 52-66

[GMS93] Ashish Gupta, Inderpal Singh Mumick, V. S. Subrahmanian: Maintaining Views Incrementally. SIGMOD Conference 1993: 157-166

[Gru93] T. R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. Technical Report KSL-93-04, Stanford Knowledge Systems Laboratory (KSL), Stanford University, 1993.

[Hay01] Pat Hayes. RDF Model Theory. http://www.w3.org/TR/2001/WD-rdf-mt-20010925/, September 2001.

[HMSV01] S. Handschuh, A. Mädche, L. Stojanovic, R. Volz. *KAON – The Karlsruhe Ontology and Semantic Web Infrastructure*. Forschungszentrum Informatik, Institut AIFB, Universität Karlsruhe.

[HTML] *HTML 4.01 Specification*
http://www.w3.org/TR/html4/

[Jeu92] M. Jeusfeld. *Änderungskontrolle in deduktiven Objektbanken.* Infix-Verlag, St. Augustin, 1992.

[Jyth] Jython Language. http://www.jython.org/docs/index.html

[KK95] W. Kim and W. Kelley. On view support in object-oriented database systems. In W. Kim, editor, *ModernDatabase Systems, The Object Model, Interoperability And Beyon*d. Addison-Wesley Publishing Company, 1995.

[Kuno] Harumi A. Kuno. View Management Issues in Object-Oriented Databases. Extended Abstract of Thesis, Dept. of Elect. Engineering and Computer Science, Software Systems Research Laboratory, University of Michigan.

[Mau] Christian Maurer. *Die Unlösbarkeit des Halteproblems und sein Bezug zur Russellschen Antinomie*, http://www.inf.fu-berlin.de/~maurer/halt/

[PH01] Jeff Z. Pan, Ian Horrocks. Metamodeling Architecture of Web Ontology Languages. *In Proceedings of the First Semantic Web Working Symposium (SWWS´01)*, Seite 131-149, 2001.

[RDF] *Resource Description Framework (RDF) Model and Syntax Specification*
http://www.w3.org/TR/REC-rdf-syntax/

[RDFS00] *Resource Description Framework (RDF) Schema Specification 1.0*
W3C Candidate Recommendation 27 March 2000
http://www.w3.org/TR/2000/CR-rdf-schema-20000327/

[SAC] Cassio Souza dos Santos, Serge Abiteboul, Claude Delobel: Virtual Schemas and Bases. *In International Conference on Extending Data Base Technology*, Cambridge, March 1994: 81-94.

[SLT] Marc H. Scholl, Christian Laasch, Markus Tresch. Updatable Views in Object-Oriented Databases. DOOD 1991: 189-207

[Tec94] O2 Technology. The O2 User's Manual, version 4.5, November 1994.

[Tol00] Karsten Tolle. *Validating RDF Parser (VRP) – Analyzing and Parsing RDF*, Heraklion, April 2000.

[WHM01] W. Nejdl, H. Dhraief, and M. Wolpers. O-Telos-RDF: A resource description format with enhanced meta-modeling functionalities based on o-telos. In *Workshop on Knowledge Markup and Semantic Annotation at the First International Conference on Knowledge Capture (K-CAP 2001)*, Kanada, Oktober 2001.

[W01] Boris Wolf. Peer-to-peer Networking for Distributed Learning Repositories. Universität Hannover, Institut KBS. Dezember 2001.

[WR00] Wolfram Conen and Reinhold Klapsing. A Logical Interpretation of RDF. In *Electronic Transactions on Artificial Intelligence (ETAI), ISSN:1401-984*1, Volume 5, 2000.

[XML] *Extensible Markup Language (XML) 1.0*
http://www.mintert.com/xml/trans/REC-xml-19980210-de.html