Technische Universität Darmstadt

Fachbereich Informatik
Fachgebiet Knowledge Engineering
Prof. Dr.-Ing. Johannes Fürnkranz

Inferenzalgorithmen zur Auswahl ontologiebasierter Situationsbeschreibungen für ein kontextadaptives Dialogsystem

Diplomarbeit

eingereicht von

Grigori Babitski

am
26. November 2004

Gutachter: Prof. Dr.-Ing. Johannes Fürnkranz

Betreuer: Dipl.-Inf. Hans-Peter Zorn

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, November 2007 Grigori Babitski

Danksagung

Ich bedanke mich herzlich beim Professor Johannes Fürnkranz, ohne den es mir nicht möglich wäre, eine Diplomarbeit, die meine Interessen im Bereich Informatik wiederspiegelt, anzufertigen. Genauso geht mein Dank an die Firma European Media Laboratory GmbH (EML), die mir zuerst einen Arbeitsplatz für die hilfswissenschaftliche Tätigkeit anbot und danach meinem Wunsch in einem Ihrer Projekte als Diplomand zu beteiligen entgegenkam. In diesem Zusammenhang möchte ich Hans-Peter Zorn – einem Mitarbeiter in EML, der mich die ganze Zeit über betreut hatte und mit hilfreichen Ratschlägen zur Seite stand, tiefsten Dank aussprechen. Meiner Familie danke ich für die Unterstützung, Geduld und Verständnis. Ganz besonderer Dank gilt meiner Freundin Lydia Greb, die mir in allen Angelegenheiten großherzig und selbstlos beistand, mich ermutigt hat und viele Tage damit verbrachte, syntaktische Fehler in dieser Arbeit zu korrigieren. Auch meinem Freund Pavel Metelitsyn möchte ich für anregende Diskussionen, hilfreiche Ratschläge und Korrekturlesen danken.

1 EINLEITUNG 1

1.1 Motivation 1

1.2 Projekt SmartWeb 2

1.3 SitCom Modul von SmartWeb 4

1.4 Aufgabestellung 5

2 ONTOLOGIEN UND IHRE FORMALE GRUNDLAGEN 9

2.1 Ontologien 9
2.1.1 Definitionen 9
2.1.2 Logische vs. ontologische Theorien 10
2.1.3 Zentrale Elemente von Ontologien 11
2.1.4 Integration von Ontologien 11
2.1.5 Klassen von Ontologien 12

2.2 Resource Description Framework 14
2.2.1 RDF-Modell und Graphmodell 14
2.2.2 RDF-Syntax 17
2.2.2.1 XML Syntax für RDF: RDF/XML 17
2.2.2.2 N-Triple und Notation 3 Syntaxen für RDF 19
2.2.3 Formale Semantik von RDF 20
2.2.4 Abfrage von RDF-Graphen 20

2.3 Resource Description Framework Schema 22
2.3.1 Typsystem 23
2.3.2 Einschränkungen für Eigenschaften 25

2.4 Beschreibungslogiken 26
2.4.1 Syntax und Semantik 27
2.4.2 Familie von Beschreibungslogiken 29
2.4.3 A-Box und T-Box 30
2.4.4 Inferenz 31

2.5 Ontology Web Language 32
2.5.1 Syntax und Semantik 34
2.5.2 Import von Ontologien 35

2.6 Ontologisches Umfeld in SmartWeb 36
2.6.1 Grundontologie für SmartWeb 36
2.6.2 Domänenontologien in SmartWeb 38
2.6.3 Basisontologie 38

2.7 Jena Semantic Web Framework 39

3 AUFGABENSTELLUNG AM BEISPIEL EINES TYPISCHEN ANWENDUNGSSZENARIO IN SMARTWEB 45

4 GRUNDLAGEN ZUR REPRÄSENTATION UND VERARBEITUNG DES KONTEXTS 47

4.1. Kontext und Context-Awareness: Begriffsklärung 47
4.1.1 Diskussion des Kontextbegriffs in der Literatur 47
4.1.2 Auslegung des Kontextbegriffs in dieser Arbeit 48
4.1.3 Pragmatik und Kontext 48
4.1.4 Context-Aware Computing 49

4.2 Context-Aware Systems 49
4.2.1 Darbietungsablauf von Kontextinformationen 50
4.2.2 Sensorarten 51
4.2.3 Kontextmodell 51
4.2.4 Nutzbarmachung von Kontextinformationen 52
4.2.5 Historie von Kontextinformationen 53
4.2.6 Grundlegende Architektur 54

4.3 DnS-Framework 55
4.3.1 DnS-Design-Pattern 56
4.3.1.1 Idee des DnS-Design-Pattern im Überblick 56
4.3.1.2 Deskriptive Komponenten für die Modellierung von Deskriptionen und Situationen gemäß dem DnS-Design-Pattern 57
4.3.1.3 Modellierung der Deskriptionen 58
4.3.1.4 Situationen gemäß dem DnS-Design-Pattern 60
4.3.1.5 Vereinfachtes Beispiel für eine Deskription und eine Situation 61
4.3.2 Erfüllungsrelation `Satisfies` und ihre Unterrelationen 61

5 ENTWURF UND IMPLEMENTIERUNG EINER CONTEXT-AWARE INFRASTRUKTUR FÜR SMARTWEB 65

5.1 Komponenten 66
5.1.1 Sensoren 66
5.1.2 Kontextquellen 67
5.1.3 Cache 69
5.1.3.1 Methoden 70
5.1.3.2 Aufbau 75
5.1.3.3 Funktionsweise im Überblick 76
5.1.3.4 Zeitkomplexität 78
5.1.3.5 Sortierung von Suchergebnissen 78
5.1.3.6 Abschließende Bemerkung 81
5.1.4 Kontextinterface 81
5.1.4.1 Multithreading 83

5.2 Repräsentierung von Kontextinformationen 84
5.2.1 Repräsentierung als Objekte in Java 84
5.2.2 Ontologische Repräsentierung 87

5.3 Erweiterbarkeit 88

5.3.1 Sensoren 89
5.3.2 Kontextquellen 89

5.4 Inbetriebnahme 90

6 ENTWURF ONTOLOGIEBASIERTER KONTEXTREPRÄSENTATION UND KONTEXTVERARBEITUNG 93

6.1 DnS-Framework in Praxis 93

6.2 Von DnS-Framework zum eigenen Entwurf 95

6.3 Festlegung des Begriffssystems 96

6.4 Modellierungsaspekte der Ontologie von Deskriptionen 100
6.4.1 Behandlung mehrerer D-Komponenten und D-Partizipanten, und Diskussion ihrer Optionalität 100
6.4.2 Widersprüchlichkeit von Deskriptionen 103

6.5 Spezialisierung von Deskriptionen 104
6.5.1 Klassifizierung nach Hierarchien 105
6.5.2 Klassifizierung nach den Anforderungen an die spezialisierende Klasse 106
6.5.3 Diskussion über die zwei vorgeschlagenen Klassifikationen 109
6.5.4 Subsumierung von D-Komponenten 110
6.5.4.1 Definitionen der Subsumierung 110
6.5.4.2 Erläuterungen zur Subsumierung an einem Beispiel 112
6.5.5 Strukturerhaltung unter den D-Partizipanten zweier D-Komponenten 113
6.5.6 Definitionen von Spezialisierung 114
6.5.7 Diskussion über den Begriff „Spezialisierung“ und seine Definitionen 122

6.6 Komplettierung von Situationen 123

6.7 Formalisierung des Begriffs „Präzisierung einer Aussage“ 124
6.7.1 Präzisierung über eine spezialisierende Deskription 124
6.7.2 Präzisierung aus determinierten D-Partizipanten 125
6.7.3 Präzisierung aus in einer Situation vorhandenen D-Partizipanten 125

7 ALGORITHMISCHE UMSETZUNG 127

7.1 Einschränkungen des algorithmischen Entwurfs und der Implementierung. 127

7.2 Vorschlag für eine prototypische Deskriptionsontologie 128

7.3 Veranschaulichung der Problemlösung an Beispielen 130

7.4 Schema der Vorgehensweise 131

7.5 Optimierungskriterien des algorithmischen Entwurfs 132

7.6 Algorithmus zur Bestimmung relevanter Deskriptionen und der für sie benötigten Kontextquellen 133
7.6.1 Ein- und Ausgabe des Algorithmus 134

7.6.2 Darstellung des Algorithmus 135
7.6.3 Anmerkungen zu der Klassenzugehörigkeitsbestimmung 138

7.7 Komplettierungsalgorithmus 140
7.7.1 Ein- und Ausgabe des Komplettierungsalgorithmus 140
7.7.2 Darstellung des Komplettierungsalgorithmus 140

7.8 Spezialisierungsalgorithmus gemäß der Spezialisierungsdefinition 1.II 143
7.8.1 Ein- und Ausgabe des Spezialisierungsalgorithmus 143
7.8.2 Darstellung des Spezialisierungsalgorithmus 143
7.8.3 Anmerkung zum Spezialisierungsalgorithmus 146
7.8.4 Anpassung des Spezialisierungsalgorithmus an die Spezialisierungsdefinitionen 1.I und 1.III 146

7.9 Präzisierungsalgorithmus 147
7.9.1 Ein- und Ausgabe des Präzisierungsalgorithmus 147
7.9.2 Darstellung des Präzisierungsalgorithmus 147

7.10 Ansatz der Prädiktierfähigkeit von Deskriptionen 149

8 EINIGE ASPEKTE DER IMPLEMENTIERUNG 153

8.1 Trennung von Ontologien 153

8.2 Problematik beim Einsatz von Jena Framework und OWL API 154

8.3 Einige Klassen und Schnittstellen des entwickelten Programmierwerkzeugs 155

8.4 Benutzung des entwickelten Programmierwerkzeugs 161

9 EVALUIERUNG UND DISKUSSION 163

9.1 Auswertung der Context-Aware Infrastruktur für SmartWeb 163
9.1.1 Akzeptanztest 163
9.1.2 Performanz 164
9.1.2.1 Performanz von Cache 164
9.1.3 Cache vs. Datenbank 169

9.2 Auswertung der ontologiebasierten Kontextrepräsentation und -verarbeitung 170
9.2.1 Ontologische Repräsentierung von Kontextinformationen 170
9.2.2 Nutzfaktor 171
9.2.3 Zeitliche Performanz 173

9.3 Vergleich mit ähnlichen Projekten 173

10 ZUSAMMENFASSUNG UND AUSBLICK 177

ANHANG A: SYNTAX UND SEMANTIK VON $\mathcal{SHOIN}(\mathcal{D})$ 181

ANHANG B: ÄQUIVALENZ VON BESCHREIBUNGSLOGISCHEN KONSTRUKTEN ZU DEN PL1 FORMELN 183

ANHANG C: BEZEICHNUNGEN IN ABBILDUNGEN..... 185

ANHANG D: EINE KURZE ÜBERSICHT ÜBER PAKETE DES PROGRAMMIERWERKZEUGS 187

ANHANG E: KONFIGURATIONSDATEI `SETTINGS.XML` 189

LITERATURVERZEICHNIS 191

1 Einleitung

1.1 Motivation

Man stelle sich einen sprachfähigen Assistenten vor, dem man Fragen zu beliebigen Themen stellen könnte, die einem spontan einfällen. Zum Beispiel „Wo gibt es hier Italiener? Wie wird das Wetter morgen? Wie hat Deutschland gestern gespielt?“. Man stelle sich auch vor, der Assistent würde präzise antworten: „Laufen Sie geradeaus 300 Meter. Sonnig, 25°C. 1:0“. Wäre es nicht schön über einen solchen Assistenten in einem mobilen Gerät zu verfügen, das man überall mit sich tragen könnte? Im Projekt SmartWeb[1] wird daran gearbeitet, dafür Technologien zu entwickeln [45].

Kommunikation zwischen Menschen ist deshalb so effektiv, weil die Kommunikationspartner durch ihre Sinnesorgane und ihr auf Erfahrung basierendes Allgemeinwissen die Situation erkennen, in der sie sich befinden. Die Kommunikationspartner sind also oft gleicher oder ähnlicher Auffassung über die bestehende Situation und brauchen deshalb nicht darüber explizit formulierte Informationen auszutauschen. Ein klassisches Beispiel dafür ist eine sich durch einen Bahnhof eilig bewegende Person, die jemanden nur mit einem Wort „Berlin“ anspricht und darauf eine sinngemäße Antwort „Gleis 1“ bekommt. Dies zeugt also von einem kontextadaptiven Verhalten, die Menschen zu Tage bringen.

Mobile Sprachdialogsysteme wie SmartWeb sollen für den Menschen, der sie benutzt, so intuitiv bedienbar sein und seine normalen Handlungsabläufe so wenig stören wie möglich. Auch sollen sie die Situation des Benutzers erkennen, ohne von ihm detaillierte Angaben zu kontextuellen Hintergründen zu verlangen, und sich der erkannten Situation anpassen, also bei der Beantwortung von Fragen oder Hinweisgebung kontextsensitiv agieren. Dafür benötigen mobile Sprachdialogsysteme, wie auch Menschen, einerseits Sensoren, um ihre Umgebung und somit die Situation wahrzunehmen, und andererseits ein Modell der Welt, um sich in der so erkannten Situation richtig verhalten zu können.

[1] www.smartweb-project.org
[2] MDA steht für „Mobile Digitale Assistenz“ und basiert auf der PDA-Serie.

1.2 Projekt SmartWeb

SmartWeb wurde als kontextsensitives Dialogsystem konzipiert, das den mobilen Benutzer in verschiedenen Rollen – sei es ein Fußgänger unterwegs, ein Sportfan im Stadium, ein Auto- oder, was eine Weltneuheit ist, ein Motorradfahrer bei der Routenplanung – überall und jederzeit unterstützt, so dass ein ubiquitäres Zugangssystem zum semantischen Web entsteht [44]. Für SmartWeb wurde ein mobiles Endgerät der MDA[2]-Serie mit einem breitbandigen Internetzugang über UMTS, HSDPA und WLAN entwickelt. Mit Hilfe dieses Endgeräts bietet SmartWeb seinen Benutzern einen multimodalen Zugriff mit gesprochener und geschriebener Sprache, Gestik, Haptik und Video auf der Eingabeseite und Sprache, Graphik, Video und Ton auf der Ausgabeseite. Es ist also möglich, das Mobiltelefon in Alltagssprache zu fragen und dieses wird mit den Informationen aus dem Netz antworten (siehe die Abbildung 1.1[3]).

Abbildung 1.1: SmartWeb als mobiles Dialogsystem am Beispiel einer Frage zu FIFA WM 2006.

SmartWeb geht klar über traditionelle Suchmaschinen wie Google hinaus, indem es komplexe, natürlichsprachlich formulierte Anfragen zulässt, und zudem den aktuellen Kontext und die jeweilige Aufgabenstellung des Benutzers berücksichtigt [44]. Aber auch im Vergleich zu anderen Dialogsystemen weist SmartWeb eine innovative und gezielt herausgearbeitete Eigenschaft auf, *domänenübergreifend* zu sein. SmartWeb ist

[2] MDA steht für „Mobile Digitale Assistenz“ und basiert auf der PDA-Serie.
[3] Diese Abbildung wurde aus [44] übernommen.

also nicht von vorne herein begrenzt, Anfragen nur aus einem einzigen Sachgebiet beantworten zu können. Es gibt Demonstrator von SmartWeb, der Fragen zum Thema Sport beantwortet, aber auch seinen Benutzern beim Navigieren assistiert (siehe die Abbildung 1.2[4]) oder über vergangene, aktuelle oder anstehende Ereignisse informiert bzw. Anfragen beantwortet, wie zum Beispiel „ Was meldet „Die Zeit“ zum Thema Politik?“ oder „Wer hat bei den Salzburger Festspielen letztes Jahr in der Premiere von La Traviata gesungen?“. Dabei muss die Domäne, aus der die nächste Anfrage kommen wird, nicht explizit spezifiziert werden.

Abbildung 1.2: SmartWeb als Navigierungsassistent. Links – Beantwortung der Frage „*Wie wird das Wetter morgen in Berlin?*“ als textuelle Ausgabe; rechts – Anzeigen eines Webcam-Bildes auf Anfrage „*Zeige mit Bilder vom Schlossplatz*“.

Ein wichtiges Entwurfsprinzip von SmartWeb ist der Einsatz von *Ontologien* als Wissensrepräsentationsmodell in allen Verarbeitungskomponenten. Zudem wurde in SmartWeb zum Ziel gesetzt, Zugang zu semantischem Web von mobilen Geräten über multimodale Schnittstellen zu ermöglichen. Dafür wurden unter anderem neun

[4] Diese Abbildung wurde aus einem unveröffentlichten Vortrag eines Mitarbeiters in EML übernommen.

verschiedene Web Services in SmartWeb integriert (nur um einige zu nennen – Routenplanung, Wetterinformation, Veranstaltungskalender, Verkehrsinformation), die je nach Bedarf automatisch zu höherwertigen Diensten komponiert werden können [44].

SmartWeb kann parallel eine Vielzahl von Benutzern unterstützen, weil das System nach dem Client-Server-Prinzip arbeitet und der Server gleichzeitig mehrere Dialoge mit verschiedenen mobilen Nutzern abwickeln kann. Dazu wird von der SmartWeb-Dialogplattform für jeden neuen eingehenden Anruf eine neue Instanz des Dialogservers instanziiert. Die Hauptverarbeitungsleistung von SmartWeb wird nicht auf dem mobilen Endgerät, sondern auf dem über eine breitbandige Funkverbindung erreichbaren Server erbracht, auf dem eine leistungsfähige Spracherkennung mit einer semantischen Anfrageanalyse und einer Komponente zum Dialogmanagement kombiniert ist. Auf dem PDA-Client läuft eine Java-basierte Steuerungskomponente für alle multimodalen Ein- und Ausgaben sowie die graphische Bedienoberfläche von SmartWeb [44].

1.3 SitCom Modul von SmartWeb

Um kontextadaptiv zu agieren, muss ein Dialogsystem Situation erkennen, in der sich sein Benutzer befindet. Dabei ist eine Situation als Gesamtheit von Parametern der physikalischen und immateriellen Umgebung des Benutzers zu verstehen, sowie seine, daraus folgende oder damit verbundene Intentionen und gegebenenfalls Einschränkungen. In SmartWeb beschäftigt sich das Modul SitCom mit der Determinierung, Aufsammlung und Bereitstellung von *kontextuellen Informationen*, also der Umgebungsparameter, und der Mutmaßung über die des Benutzers Intentionen, die mit kontextuellen Informationen und dem Algemeinwissen (gesundem Menschenverstand) begründet ist [46].

Das Allgemeinwissen über Intentionen bzw. Verhaltensweisen, die Menschen in verschiedenen Situationen pflegen, wird innerhalb SitCom in einer *pragmatischen Ontologie* dargelegt. Handele es sich beispielsweise um die Navigation, so gehöre zu dem pragmatischen Wissen dazu, dass Fußgänger normalerweise öffentliche Verkehrsmittel einem Spaziergang vorziehen, wenn es gerade regnet, oder dass eine Person, die in ihrer momentanen Rolle als Tourist eine Stadt erkundet, bei einem schönen Wetter für sich als Route einen Spaziergang wünscht, der über möglichst viele

Sehenswürdigkeiten führt. Die auf dieses Wissen gestützte Schlüsse werden erst durch die Bereitstellung von dafür relevanten kontextuellen Informationen ermöglicht. Zum Beispiel die aktuellen Wetterverhältnisse, die Verkehrslage, das Fortbewegungsmittel des Benutzers, sein Aufenthaltsort, die Rolle, die er momentan spielt, seine Vorlieben und gegebenenfalls gesundheitliche Einschränkungen usw. Die kontextuellen Informationen können ihrerseits erst ermittelt werden, wenn die benötigten *Sensordaten* vorliegen, zum Beispiel die von einem GPS Sensor bestimmte geographische Koordinaten oder von einem sogenannten virtuellen Sensor festgestellte Angaben über den Benutzer. Außerdem unterliegen kontextuellen Informationen folgender Anforderung: Da die mutmaßliche Intentionen des Benutzers aus der pragmatischen Ontologie erschlossen werden, müssen kontextuelle Informationen ebenfalls in dieser Ontologie repräsentierbar sein.

1.4 Aufgabestellung

Die Aufgabe des SitCom ist die semantische Repräsentation einer Äußerung des Benutzers derart zu modifizieren, dass sie der Situation, in der er sich befindet, entspricht. Diese Aufgabe lässt sich in zwei Teile unterteilen.

Erstens soll eine Infrastruktur entwickelt und implementiert werden, die über folgende Funktionalitäten verfügt:

- Aktualisierung von kontextuellen Parametern eines Benutzers, sobald neue Sensordaten eintreffen,
- Bereitstellung von kontextuellen Informationen bezüglich eines Benutzers auf Anfrage,
- Speicherung sämtlicher Kontextinformationen (sowie Löschung einiger bestimmter Kontextinformationen nach Anfrage).

Für diese Zwecke muss Folgendes entwickelt und umgesetzt werden:

- Repräsentierungsform verschiedener Arten von kontextuellen Informationen. Dabei wurde anfangs eine feste Menge für Kontextinformationsarten vorgegeben (z.B. Wetterverhältnisse, Zeit, Ort). Jedoch soll die Infrastruktur in dieser Hinsicht erweiterbar sein,
- Ein flexibles Speicherungssystem mit schnellem Zugriff.

Somit soll die zu implementierende Infrastruktur Akquirierung und Management von kontextueller Informationen regeln und nicht darüber hinaus, in Bereich der praktischen Verwendung von kontextuellen Informationen, gehen. Diese Infrastruktur soll nicht nur in SmartWeb Demonstrator einsatzfähig sein, sondern auch übertragbar auf andere, ähnliche Projekte. Diesem Teil der Aufgabe ist das Kapitel 5 gewidmet.

Im zweiten, den Schwerpunkt dieser Arbeit darstellenden Teil der Aufgabestellung soll ein Programmierwerkzeug entwickelt werden, das gestützt auf das pragmatische Wissen und auf die Infrastruktur aus dem ersten Teil die Letztere erweitert, indem es Algorithmen bzw. Dienste bereitstellt, mit deren Hilfe die Situation des Benutzers erschlossen und seine Anfrage ihr angepasst wird. Dafür ist Folgendes zu verrichten:

- Entwicklung einer kleinen pragmatischen Ontologie für ein Anwendungsszenario des SmartWeb aus dem Gebiet der Navigation. Diese Ontologie soll gemäß einem *Pattern* gestaltet werden, das in dem (in SmartWeb eingesetzten) sogenannten *DnS-Framework* festgelegt ist,
- Konzipierung von oben genannten Diensten. Diese sollen eine theoretische Untersuchung zu Möglichkeiten der Verwendung einer beliebigen, nach dem genannten Pattern modellierten, pragmatischen Ontologie darstellen, also sich nicht auf eine reduzierte, für einen Demonstrator hinreichende Ontologie beschränken, sondern realitätstauglich sein,
- Entwicklung und Implementierung von oben genannten Algorithmen im für einen Demonstrator hinreichenden Umfang.

Dem zweiten Teil der Aufgabenstellung sind Kapitel 6, 7 und 8 gewidmet.

Die geschilderte Aufgabestellung kann zusammenfassend wie folgt angegeben werden: es soll im Rahmen des Moduls SitCom ein sogenanntes *Context-Aware System* für SmartWeb entwickelt werden. Die Klärung des Begriffs *Context-Awareness* sowie ein Überblick über Context-Aware Systems und die gemeinsamen Prinzipien, die ihnen unterliegen, erfolgen im Kapitel 4.

Die Implementierung beider Teile der Aufgabenstellung soll in der Programmiersprache Java erfolgen. Zur Verfügung wurden die in SmartWeb verwendeten Ontologien, sowie die im Rahmen des DnS-Framework entwickelte *DnS-Ontologie* gestellt. Ebenfalls

wurde die Nutzung des Web Service TInfo der Deutschen Telekom ermöglicht. Dieser bietet die innerhalb der Navigationsdomäne notwendigen Informationen (Wetterverhältnisse, koordinatenabhängige Adressenermittlung und Weiteres) an, indem er eine API zur Verfügung stellt, so dass der Zugriff aus Java sich sehr einfach gestalten lässt.

2 Ontologien und ihre formale Grundlagen

2.1 Ontologien

2.1.1 Definitionen

Ontologie ist ihrem Ursprung nach eine Disziplin der theoretischen Philosophie und bezeichnet dort „die Lehre vom Sein, bzw. von den grundsätzlichsten, allgemeinsten, elementarsten, fundamentalen und konstitutiven Eigenschaften, den Prinzipien, den grundsätzlichsten Wesens-, Ordnungs- und Begriffsbestimmungen des Seins“[5]. In Worten von Guarino kann die Ontologie in diesem Sinne als „particular system of categories accounting for a certain vision of the world” [41] betrachtet werden. In [41] unterscheidet jedoch Guarino zwischen *der* Ontologie als philosophische Disziplin und *einer* (von vielen möglichen) Ontologie, wie dies auf dem Gebiet der künstlichen Intelligenz bzw. Informatik überhaupt aufgefasst wird. So bezieht sich das Erstere auf die Sachverhalte, die in der Realität (in der Welt) existieren, und zwar unabhängig von den Mitteln und der Art der Betrachtung, die wir daran ausüben. Das Zweite ist ein Artefakt, das eine meist formale Art der Weltbetrachtung mit denen dafür zur Verfügung stehenden Mitteln darstellt. Ein solches Artefakt besteht aus einem Vokabular zum Beschreiben der Realität und einer Menge von expliziten, vorwiegend in einer logischen Sprache formulierten Annahmen über die intendierte Bedeutung der Wörter aus dem Vokabular. Die Letzteren sind meist entweder einstellige Prädikate – sie werden „Konzepte“ bezeichnet, oder die zweistellige Prädikate – sie nennt man „Relationen“. Weiterhin, um abgezeichnete Zweideutigkeit des Begriffs „Ontologie“ zu disambiguieren, schlägt Guarino vor, ihre philosophische Auffassung als „*Konzeptualisierung*“ zu bezeichnen. Dadurch erst wird die Bedeutung der von Gruber gegebenen Definition von Ontologie als einer „explizite formale Spezifikation einer gemeinsamen Konzeptualisierung“ [42] erkenntlich. Insbesondere ist dabei zu betonen, dass es für eine bestimmte Domäne nur eine, die Realität getreu wiedergebende Konzeptualisierung gibt, während es viele Ontologien, also diverse Vokabulare und Wortbedeutungsinterpretation, geben kann, die diese Konzeptualisierung modellieren oder erzeugen.

[5] Zitat aus einem unter http://www.philolex.de/philolex.htm online verfügbaren Philosophielexikon (Stichwortartikel „Ontologie“).

Guarinos Überlegungen führen zu einer formalen Definition der Begriffe „Konzeptualisierung“ und „Ontologie“ und zu einer formalen Diskussion, wie eine Ontologie eine Konzeptualisierung spezifiziert. Da diese Überlegungen in ihrer vollen Ausführlichkeit hier nicht dargelegt werden können, wird eine, in [30] gegebene, etwas simplifizierte und informelle, dennoch intuitiv sehr zugängliche und hinreichende Definition der Begriffe „*Ontologie*“ und „*Konzept*“ vereinbart:

> *Eine Ontologie ist die explizite, formale Spezifikation von Konzepten und deren Beziehungen zueinander. Ein Konzept ist die Vereinigung aller Elemente, die alle eine bestimmte Eigenschaft oder mehrere bestimmte Eigenschaften besitzen.*

2.1.2 Logische vs. ontologische Theorien

Da eine Ontologie formaler Natur ist und in einer logischen Sprache formuliert wird, stellt sie im Grunde eine Menge von Ausdrücken in dieser logischen Sprache dar. Jedoch sind Ontologien von bloß einer Menge von logischen Formeln strikt zu unterscheiden. Dies ist in der sogenannten *ontologischen Verpflichtung* (im Englischen: Ontological Commitment) begründet. Dies ist nach [43] „what it assumes about the nature of reality.“ Zum Beispiel liegt die Annahme der propositionalen Logik darin, dass es Fakten gibt, die in der Welt entweder falsch oder richtig sind. Prädikatenlogik erster Stufe (PL1) macht die Annahme, dass es neben den Fakten auch Objekte und Relationen gibt, und die Letzteren in der Welt entweder gelten oder nicht. Temporale Logiken nehmen zusätzlich an, dass es in der Welt Zeit existiert, von der die Gültigkeit abhängt. Hingegen sind die Ontologien in ihrer ontologischen Verpflichtung an die Konzeptualisierung der jeweiligen Domäne gebunden, die sie möglichst exakt wiedergeben sollen. Ontologien haben also, im Gegensatz zu logischen Theorien im Allgemeinen, einen speziellen Zweck, zu welchem logische Sprachen verwendet werden. Im Sinne der mathematischen Logik kann man diesen Zweck auch wie folgt formulieren: die möglichen Modelle einer ontologischen Theorie (also einer Ontologie, betrachtet als eine logische Theorie), die als Domäne eine Menge von in der Realität vorkommende Entitäten haben, sollen die Welt bzw. eine Domäne (einen Bereich der Welt) möglichst getreu wiedergeben.

2.1.3 Zentrale Elemente von Ontologien

Die zentralen Elemente einer Ontologie sind *Konzepte* (auch Begriffe oder Klassen), die eine Menge von *Individuen* (auch Instanzen) vereinen. Zum Beispiel ist diese, vorliegende Diplomarbeit ein Individuum, also eine konkrete Ausprägung des Konzepts „Diplomarbeit“. Schließlich die, meist zweistellige, *Relationen* (auch Rollen) zwischen Konzepten, oder Individuen, oder Konzepten und Individuen, oder Relationen selbst. Mit dem Begriff „*(ontologische) Entität*“ bezeichnet man ein Element einer Ontologie, ohne zu spezifizieren, welcher der genannten Arten dieses Element ist.

Eine der zentralen Relationen ist diejenige, die einem Individuum die Zugehörigkeit zu einem oder mehreren Konzepten zuweist. Eine weitere wichtige Relation ist die Unterklassenbeziehung zwischen Konzepten. Zum Beispiel ist das Konzept „Diplomarbeit“ gegebenenfalls ein Unterkonzept von „wissenschaftliche Publikation“. Diese Relation wird als transitiv interpretiert, wodurch sich eine hierarchische Struktur der Konzepte einer Ontologie, deren *Taxonomie*, ergibt. Im einfachsten Fall besteht eine Ontologie einzig und allein aus der Taxonomie. Eine Taxonomie ist im Allgemeinen eine baumähnliche Struktur, jedoch keine baumartige, denn ein Konzept sowohl mehrere Ober- als auch mehrere Unterklassen haben kann. Schließlich gehört zu den zentralen Elementen von Ontologien die Unterrelationsbeziehung zwischen Relationen. Zum Beispiel ist die Relation „hat Eigenschaft“ eine Oberrelation von „hat zeitliche Eigenschaft“, „hat räumliche Eigenschaft“ und „hat abstrakte Eigenschaft“.

2.1.4 Integration von Ontologien

Ein bedeutender Aspekt bei der Verwendung von Ontologien ist die *Integration*. Eine Ontologie modelliert sozusagen einen Ausschnitt der Welt, jedoch nie alle Sachverhalte der Welt (dies wäre eine unrealistisch komplexe Aufgabe). Die Vision von Semantic Web ist, dass je nach Bedarf des Einzelfalls Ontologien zu unterschiedlichen Domänen aufgesucht werden und, einander ergänzend, gemeinsam einen Dienst leisten oder zur Problemlösung beitragen [38]. Zum Beispiel könnte eine Ontologie, die Web Services beschreibt, mit einer Ontologie, die einer Person bei der Navigation hilft, zusammengeschlossen werden, um auf diese Weise das Vorhaben dieser Person zu erschließen und je nach Situation durch gezielte Aufrufe von bestimmten Web Services

ihr beispielsweise Informationen über die Sehenswürdigkeiten, Verkehrsverbindungen, Wetterverhältnisse usw. zu liefern.

Doch die Integration von Ontologien ist kein triviales Problem, unter anderem deswegen, weil Ontologien grundsätzlich unterschiedliche Prinzipien bzw. Weltsichten unterliegen können [38]. Zum Beispiel postulieren einige Ontologien, dass zu einer bestimmten Zeit an einem bestimmten Ort sich nur ein Objekt befinden kann und die anderen Ontologien lassen in diesem Sinne durchaus Überschneidungen zu. Einige Ontologien unterscheiden zwischen vergänglichen, in der Zeit existierenden Entitäten und den zeitlosen Entitäten, in denen die Ersten teilnehmen können; andere behandeln die Vergänglichkeit nicht. Einige Ontologien lassen Modalität zu, d.h. Sachverhalte können zwingend oder möglicherweise gültig sein, wie es bei den modalen Logiken angenommen wird.

Selbst wenn zwei Ontologien in bestimmten Prinzipen harmonieren, ist damit noch nicht gegeben, dass diese dort in einer übereinstimmenden Weise modelliert sind. Denn es sind verschiedene Wege möglich, wie man beispielsweise Ort und Zeit und Prozesse, die in der Zeit stattfinden, modelliert. Verschiedene Vokabulare, die Ontologien verwenden, stellen dabei nicht das größte Problem dar, denn Parallelen zwischen Wörtern der Vokabulare zu finden ist oft ein mühsam zu überwindendes Hindernis.

2.1.5 Klassen von Ontologien

Um die geschilderte Problematik bei der Integration zu überwinden, werden die Ontologien nach deren Abstraktionsniveau bzw. Grad der Generalisierung in folgende drei Klassen unterteilt [41] (siehe Abbildung 2.1[6]):

- *Grundontologien* (im Englischen: Foundational oder top-level Ontologies): sie beschreiben sehr allgemeine Konzepte wie Zeit, Raum, Objekt, Ereignis, Aktion usw. Diese sind unabhängig von einer konkreten Domäne, haben also eine allgemeine Gültigkeit und sind somit in mehreren (sogar allen) Domänen nutzbar.
- *Domänenontologien* und *Aufgabenontologien* (im Englischen: Task Ontologies): sie beschreiben Vokabulare von jeweils einer Domäne (z.B. Navigation,

[6] Diese Abbildung wurde von einer äquivalenten Abbildung aus [41] abgezeichnet.

Sportereignisse) bzw. einer Aufgabe oder Aktivität (z.B. Diagnosestellung) und sind daher für mehrere Anwendungen der jeweiligen Domäne nutzbar. In ihrer Beschreibung machen Domänen- und Aufgabenontologien allerdings keine Aussagen über Entitäten außerhalb der Domäne. Die Eigenschaften der repräsentierten Entitäten sind auf die Elemente eingeschränkt, die in dem repräsentierten Teilbereich der Welt relevant sind.

- *Anwendungsontologien*: sie beschreiben Konzepte, die von einer Domäne bzw. einer Aufgabe abhängen und eine Spezialisierung von beiden darstellen. Diese Konzepte korrespondieren oft mit den Rollen, die Entitäten der Domäne während der Ausübung einer bestimmten Aktivität ausüben (z.B. Entität „Komponente“, Rolle „optional“). Eine Anwendungsontologie wird primär an ihrer Nutzbarkeit und Nützlichkeit in der ihr bestimmten Anwendung gemessen. Beim Entwurf wird eine Nutzbarkeit außerhalb der bestimmten Anwendung nicht explizit berücksichtigt.

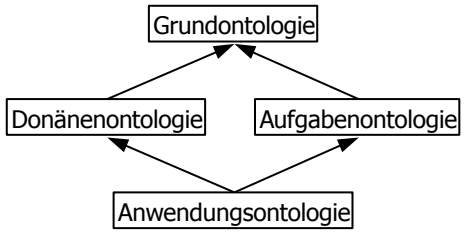


Abbildung 2.1: Drei Klassen der Generalisierung von Ontologien. Pfeile repräsentieren die Spezialisierungsrelation.

Formal gibt es keine Unterschiede zwischen Ontologien aus verschiedenen Klassen. Viel mehr handelt es sich um eine gewisse Richtlinie der ontologischen Modellierung und eine ungefähre, intendierte Klassifikation.

Es ist wünschenswert, dass es nur eine Grundontologie gibt und sie als Ausgangspunkt bei der Modellierung von Domänenontologien dient, anstatt dass jede Domänenontologie, bildlich gesprochen, „das Fahrrad neu erfindet“ und weder mit einer Grundontologie kompatibel noch mit den anderen Domänenontologien integrierbar ist. Für diese Zwecke sollte die Grundontologie auch die sogenannten *Ontology Design Patterns* definieren, mit deren Hilfe Domänenontologien die wiederkehrenden Angelegenheiten, wie die zeitliche und räumliche Lokation, in konsistenter Weise modellieren. Im diesen Sinne ist eine Grundontologie als *generisch* zu bezeichnen.

2.2 Resource Description Framework

Resource Description Framework (*RDF*) ist eine formale Sprache zur Darstellung von Metadaten.

RDF wurde vom World Wide Web Consortium (W3C)[7] zusammen mit der Web Ontology Language als Grundstein für das Semantische Web entwickelt und ist frei verfügbar. RDF baut auf der von XML zur Verfügung gestellter Möglichkeit zum Datenaustausch auf und ermöglicht das Modellieren und Repräsentieren von Metadaten, also von Aussagen über Daten (genauer über Ressourcen), jedoch nicht deren Interpretation oder gar Verarbeitung im Sinne von Inferenz.

Wie jede formale Sprache verfügt RDF über Syntax und Semantik. Darüber hinaus wurde im Rahmen von RDF eine weitere Komponente entwickelt: die Abfragesprache. Schließlich werden im Zusammenhang mit RDF Bezeichnungen „Graphmodell“ bzw. „RDF-Modell“ stark frequentiert genutzt. In folgenden Abschnitten werden diese Begriffe und ihr Bezug zueinander dargestellt und somit das für diese Arbeit notwendige Grundwissen von RDF vermittelt.

2.2.1 RDF-Modell und Graphmodell

Im Kern standardisiert RDF die Beschreibung von Aussagen der Form „`S` *hat die Eigenschaft* `P` *mit Wert* `O`“, also von Relationstupeln `P(S,O)`. Um festzulegen, wie solche Aussagen dargestellt werden, ist eine abstrakte Repräsentation definiert worden – das sogenannte *RDF-Modell*. Äquivalent dazu, jedoch für Menschen lesbarer, wurde das RDF *Graphmodell* entwickelt, welches die graphische Darstellung von Relationstupeln `P(S,O)` festlegt. Weil die beiden Modelle zwar auf unterschiedliche Weise, jedoch dasselbe Datenmodell definieren, werden sie im Weiterem gemeinsam zur Darstellung dieses Datenmodells verwendet.

In der grundlegenden Form von Aussagen – `P(S,O)` – wird `S` häufig in Analogie zu natürlichsprachlichen Grammatiken als *Subjekt*, `P` als *Prädikat* und `O` als *Objekt*

[7] http://www.w3.org

bezeichnet. Solche Aussagen werden auf folgende drei *Grundtypen des Datenmodells* abgebildet:

- *Ressourcen*: Dies ist die Superklasse aller anderen Datentypen. Eine Ressource ist eine Entität, die eindeutig durch einen Unified Resource Identifier (URI) identifiziert werden kann. Dies kann eine einzelne Webseite, eine Sammlung von Webseiten, aber auch ein Objekt sein, auf das nicht über das Web zugegriffen werden kann, wie z.B. ein Buch.
- *Properties* bzw. *Eigenschaften*: Eine Eigenschaft ist eine bestimmte Charakteristik zur Beschreibung einer Ressource, z.B. `Autor` oder `Titel`. Jede Eigenschaft hat eine spezifische Bedeutung, deren Definition jedoch nicht Teil des Grundmodells von RDF ist. Um beispielsweise zu definieren, was die Bedeutung der Eigenschaft `Autor` ist, muss man auf Mechanismen zurückgreifen, die nicht zum Basismodell von RDF gehören (z.B. auf das in 2.5 dargestellte OWL).
- *Statements* bzw. *Aussagen*: Aussagen bestehen aus der Kombination einer Ressource, einer Eigenschaft und eines Wertes für die Eigenschaft (eine Ressource oder ein Literal[8]). Diese drei Teile werden als Subjekt, Prädikat und Objekt bezeichnet, wie bereits oben gezeigt. Da Aussagen Ressourcen sind, ist es möglich Aussagen über Aussagen zu treffen.

Mit Hilfe dieser Grundtypen lässt sich ein Statement als ein Tripel aus einem Ressource oder einem so genannten leeren (auch als anonym bezeichnet) Knoten, einer Eigenschaft und einem Ressource bzw. Literal bzw. leeren Knoten angeben. Dies wird *RDF-Tripel* genannt. Ein RDF-Dokument ist eine endliche Menge von RDF-Tripeln. Ein derartig im RDF-Modell definiertes RDF-Tripel wird im Graphmodell durch einen gerichteten kanten-beschrifteten Graph, bestehend aus einem Subjekt-, einem Objektknoten und einer gerichteten eine Eigenschaft bezeichnender Kante, repräsentiert (siehe die Abbildung 2.2[9]). Ein Objektknoten wird elliptisch gezeichnet, falls er ein Ressource repräsentiert bzw. rechteckig, wenn er für ein Literal steht.

[8] Literale ist ein Wert für eine Eigenschaft, der allein durch seine lexikalische Repräsentation identifiziert wird (z.B. eine Zahl oder Zeichenkette).
[9] Diese Abbildung wurde von der entsprechenden in [49] abgezeichnet.

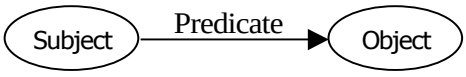


Abbildung 2.2: Graphendarstellung eines RDF-Tripels.

Bisher wurden nur binäre Relationen dargestellt. Es ist jedoch möglich mit ihrer Hilfe auch *n*-äre Relationen zu repräsentieren, indem man für die *n*-äre Relation einen eigenen anonymen Knoten erzeugt und die an der Relation beteiligten Entitäten zu diesem Knoten in Relation setzt. Dieser anonyme Knoten ist implizit existentiell quantifiziert. So kann auch eine Aussage wie „Lucy wiegt 100 Pfund" repräsentiert werden. Es ergibt sich dann folgende Graphdarstellung:

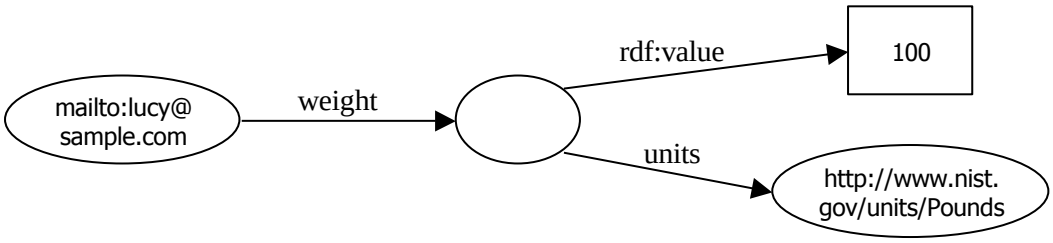


Abbildung 2.3[10]: Graphdarstellung für „Lucy wiegt 100 Pfund".

Die ursprüngliche Aussage wird genaugenommen verändert zu „Lucy hat das Gewicht X und X hat den Wert 100 und X hat die Einheit Pfund".

RDF-Modell erlaubt mit Hilfe von bisher beschriebenen Mitteln sowie eines speziellen Konstrukts Aussagen über Mengen von Ressourcen zu treffen. Dieses Konstrukt besteht aus bestimmten vordefinierten und eingebauten Relationen und Typen[11] und wird „*Container*" bezeichnet. Es gibt drei Typen von Containern: *Bag* – für ungeordnete Liste von Ressourcen und Literalen, *Sequence* – für geordnete Liste von Ressourcen und Literalen und *Alternative* – für Liste von Ressourcen und Literalen, die Alternativen für den einzigen Wert einer Eigenschaft darstellen. So lässt sich mit Hilfe von Container die Aussage „Lucy isst nur Japanisch, Brasilianisch und Französisch gerne" durch folgende, einem RDF-Triplet äquivalente Teilaussagen modellieren: „Lucy isst gerne X", „X ist vom Type Bag", „X beinhaltet Japanisch", „X beinhaltet Brasilianisch" und „X beinhaltet

[10] Diese Abbildung wurde von der entsprechenden in [49] abgezeichnet.
[11] Der Begriff „Type" wird im Abschnitt 2.3.1 erläutert.

`Französisch`“, wobei `X` ein leerer Knoten ist und „`beinhaltet`“ für eine in RDF vordefinierte Relationsart[12] steht.

Resümierend lässt es sich behaupten, dass RDF-Modell essentiell ein Semantisches Netz ist, das durch zwei Konzepte ergänzt wurde. Erstes – die bereits genannte globale Identifizierbarkeit von Ressourcen durch URIs. Zweites – die Reifikation, die hier nur als eine explizite Möglichkeit erwähnt wird, Aussagen über Aussagen zu machen, z.B. „*Statement X hat Subjekt Y*“.

2.2.2 RDF-Syntax

Bisher wurden die Modellierungsprimitiven des RDF-Modells eingeführt. Dazu wurde eine Graphsyntax (oder abstrakte Syntax, wie es in [51] bezeichnet wird) verwendet. In diesem Abschnitt werden nun zwei Ansätze zur Serialisierung dieser Graphen vorgestellt. Zu einem ist es die *XML Serialisierung* – die kanonische Syntax von RDF, die zum Austausch von Wissensdaten im Semantic Web verwendet wird, zum anderen sind es *N-Triple* und *Notation 3* Syntaxen, die kürzer und für Menschen besser lesbar sind.

Im Wesentlichen müssen nur vier Konstrukte aus der Graphsyntax in eine Serialisierung übersetzt werden: Knoten, die eine Ressource repräsentieren; Knoten, die ein Literal repräsentieren; anonyme Knoten und Kanten zwischen Knoten. Alle anderen Konstrukte können darauf zurückgeführt werden (auch Reifikation und Container). Um eine Syntax zu definieren genügt es also darzulegen, wie normale (Ressourcen- und Literal-) Knoten, anonyme Knoten und Kanten serialisiert werden.

2.2.2.1 XML Syntax für RDF: RDF/XML

Das XML Syntax für RDF erlaubt ein RDF-Dokument mit XML darzustellen. Das vollständige RDF/XML wird in [51] definiert. An dieser Stelle werden lediglich die wichtigsten Konstrukte an einem Beispiel vorgestellt. Die Aussage aus dem Beispiel in der Abbildung 2.3, „`Lucy wiegt 100 Pfund`“, wird wie folgt in RDF/XML dargestellt [49]:

[12] Genauer genommen handelt es sich dabei um sogenannte container membership properties `rdf:_n`, wobei n für eine natürliche Zahl steht, welche die Nummer eines Elements in der Liste angibt.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                                        xmlns:u="http://www.nist.gov/units/">
   <rdf:Description rdf:about="mailto:lucy@sample.com">
      <u:weight>
         <rdf:Description>
            <rdf:value>100</rdf:value>
            <u:units rdf:resource="http://www.nist.gov/units/Pounds"/>
         </rdf:Description>
      </u:weight>
   </rdf:Description>
</rdf:RDF>
```

Abbildung 2.4: Darstellung von „*Lucy wiegt 100 Pfund*“ in XML/RDF.

An diesem einfachen Beispiel kann man als Erstes ableiten, dass der Namensraum von RDF URI-Referenzen `http://www.w3.org/1999/02/22-rdf-syntax-ns`[13] heißt und in XML üblicherweise Präfix `rdf` zugewiesen bekommt; zweitens – die Grundstruktur der Abbildung von der Graphsyntax in XML:

`rdf:RDF`:
optionaler Wrapper, um RDF in andere XML Anwendungen (z.B. XHTML) einzubetten.

`rdf:Description`:
fasst Aussagen über eine Ressource zusammen. Die Ressource, über die gesprochen wird (die also Subjekt der Aussagen ist), kann durch ein `rdf:about` Attribut angegeben werden, falls es sich um eine externe Ressource handelt. Wird das rdf:about Attribut weggelassen, so wird eine neue Ressource erzeugt, die durch ein `rdf:ID` Attribut benannt werden kann.

Eigenschafts-Elemente:
(in der Abbildung 2.4: `u:weight`, `u:units`, `rdf:value`) In `rdf:Description` Elementen können beliebig viele Eigenschafts-Elemente vorkommen, die als Namen URI (bzw. den entsprechenden qualifizierten Namen) des Prädikates der Aussage tragen.
Das Objekt zu einem solchen Prädikat kann nun auf zwei Arten spezifiziert werden:

[13] Unter diesem Namensraum ist das komplette RDF Vokabular definiert und ist von [50] aus verfügbar. Modelltheoretische Semantik seiner Terme ist ebenfalls von [50] aus verfügbar.

1. Ist das Objekt ein Literal, so wird es als Inhalt des Eigenschafts-Elements verwendet.
2. Ist das Objekt eine Ressource, so wird seine URI als Wert des `rdf:resource` Attributs des Eigenschafts-Elements angegeben.

Verschachtelung von Aussagen:
Aussagen können verschachtelt werden, falls das Objekt einer Aussage mit dem Subjekt einer anderen übereinstimmt. Durch diese Technik können dann auch anonyme Knoten repräsentiert werden.

Das Subjekt einer Aussage wird also durch das umgebende `rdf:Description` Element, das Prädikat durch das jeweilige Eigenschafts-Element und das Objekt, falls es sich um eine Ressource handelt, durch den Wert des `rdf:resource` Attributs des Eigenschafts-Elements bzw., falls es sich um ein Literal handelt, durch den Inhalt des Eigenschafts-Elements angegeben.

2.2.2.2 N-Triple und Notation 3 Syntaxen für RDF

Notation 3 (N3) ist eine von Berners-Lee entwickelte Syntax, die eine kompakte und für Menschen einfacher lesbare Alternative zu RDF/XML darstellt. N-Triple ist eine Vereinfachung von Notation 3. Die wesentliche Idee, die beiden Formalismen unterliegt, ist die Beziehung zwischen zwei Knoten eines RDF-Graphen durch eine Kante desselbigen direkt als Tripel zu repräsentieren. Weil N-Triple und Notation 3 vom Prinzip her recht ähnlich sind, wird im Weiteren nur die erste Syntax vorgestellt.

Die Aussage des bereits vertrauten Beispiels „`Lucy wiegt 100 Pfund`“ wird in N-Triple wie folgt notiert [49]:

```
<mailto:lucy@sample.com> <http://www.nist.gov/units/weight> _:a.

_:a <http://www.w3.org/1999/02/22-rdf -syntax-ns#value> "100".
_:a <http://www.nist.gov/units/units> <http://www.nist.gov/units/Pounds>.
```

Abbildung 2.5: Darstellung von „*Lucy wiegt 100 Pfund*“ in N-Triple.

Ein RDF-Graph wird wie folgt in N-Triple serialisiert:

- Ein mit einer URI `u` markierter Knoten wird durch `<u>` repräsentiert.
- Ein mit einem Literal `l` markierter Knoten wird durch `"l"` repräsentiert.
- Ein anonymer Knoten wird durch `_:id` repräsentiert, wobei `id` bisher noch nicht verwendeter Identifikator ist.
- Eine mit einer URI `p` markierte Kante zwischen Knoten k_1 und k_2 wird durch r_1<p>r_2 repräsentiert, wobei r_i die Repräsentation von k_i in N-Triple ist.

2.2.3 Formale Semantik von RDF

In [49] wird eine modelltheoretische Semantik von RDF vorgestellt. Ihr Ziel ist eine Abbildung (Interpretation) von Graphsyntax auf ein mögliches Modell der dargestellten Welt zu geben. Dies ermöglicht natürlich aus einem RDF-Graph zu inferieren. In [52] wird gezeigt wie RDF mittels Prädikatenlogik erster Stufe formalisiert werden kann. Dies impliziert insbesondere, dass Letzteres expressiver als RDF ist, und folglich ist RDF, als logischer Formalismus betrachtet, monoton.

Da die vorliegende Arbeit keine tiefere Einsicht in die Semantik von RDF voraussetzt, wird es in Weiterem nicht näher darauf eingegangen.

2.2.4 Abfrage von RDF-Graphen

Zum Abfragen von bzw. in RDF-Graphen gibt es verschiedene Sprachen. Ein Vergleich von diversen RDF-Abfragesprachen [49] zeigt, dass die Standardisierung hier noch nicht sehr weit fortgeschritten ist. Mit der SPARQL[14] Query Language [48], vorgeschlagen vom W3C, scheint sich jedoch ein neuer Standard auf diesem Gebiet zu etablieren.

Ein Vertreter einer RDF-Abfragesprache – welcher auch im Jena Framework (siehe Kapitel 2.7) verfügbar ist – ist die RDF Query Language (RDQL) [53], welche der Form nach stark der SQL ähnelt. Die prinzipielle Syntax ist folgende:

```
SELECT {Liste von Variablen}
```

[14] http://www.w3.org/TR/rdf-sparql-query/

```
WHERE {Vergleichsmuster zum Graphen}
AND {Filter}
USING {Namespace-Mappings}
```

Im `SELECT`-Teil wird eine Liste von Variablen angegeben, für die eine Belegung mit Ressourcen bzw. Literalwerten gesucht wird. Im `WHERE`-Teil werden beliebig viele RDF-Tripel angegeben, bei denen Subjekt, Objekt und/oder Prädikat durch eine Variable (nicht zwingend aus obiger Liste) besetzt werden kann. Im `AND`-Teil können optional durch die Angabe möglicher Variablenbelegungen diejenige Tripel herausgefiltert werden, welche die vorgegebenen Werte für diese Variablen aufweisen. Schließlich können im `USING`-Teil optional für die URIs Präfixe definiert werden – dies dient allein der verkürzten Schreibweise einer Anfrage.

Das Ergebnis einer Anfrage ist eine Tabelle mit allen möglichen Belegungen von in `SELECT` angegebenen Variablen.

Folgendes Beispiel illustriert eine Anfrage nach Ressourcen (implizit – Personen), die über ein Gewicht verfügen, das wenigstens 50 beträgt (ohne Angabe von Einheiten):

```
SELECT
 ?person ?weight

WHERE
 (?person, <u:weight>, ?x),
 (?x, <rdf:value>, ?weight)

AND
 ?pub >= 50

USING
  rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
  u for <http://www.nist.gov/units/>
```

Diese Anfrage, vorausgesetzt sie wurde an den RDF-Graph aus der Abbildung 2.3 gestellt, hat als Ergebnis eine Tabelle mit nur einem Eintrag: Bindung von `?person` an `mailto:lucy@sample.com` und `?weight` an den Wert `100`.

2.3 Resource Description Framework Schema

Ergänzend zu dem im vorigen Kapitel dargestellten Resource Description Framework wird in diesem ein Überblick über *Resource Description Framework Schema* (*RDFS***)** gegeben, indem die wichtigsten Konstrukte informell erläutert werden. RDFS ist wie RDF eine W3C-Empfehlung [50]. RDFS erweitert RDF, indem es ein größeres Vokabular[15] mit komplexeren semantischen Konzepten und Beziehungen zur Verfügung stellt. Im Gegensatz zu XML Schema, das rein syntaktische Bedingungen über die Struktur eines XML Dokumentes spezifizieren kann, stellt RDF Schema (wenn auch in begrenztem Umfang) Informationen über die Interpretation einer Aussage im RDF Datenmodell zur Verfügung, mit anderen Worten es wird eine (sehr eingeschränkte) Ontologie vordefiniert. Dabei fügt RDFS dem RDF-Modell keine neue syntaktische Konstrukte hinzu, so dass jedes RDFS-Dokument auch ein RDF-Dokument ist. In [54] wird in selber Manier wie für RDF eine modelltheoretische Semantik für RDFS festgelegt (darauf wird hier nicht eingegangen).

Während RDF ein Datenmodell definiert, in dem Ressourcen, Eigenschaften und die dazugehörigen Werte beschrieben werden können, ist das Ziel des RDFS ein Vokabular aufzustellen, welches zwei typische Arten von Aussagen über Ressourcen und Eigenschaften zur Verfügung stellt [49]:

1. **Typisierung:** Über eine Ressource soll es ermöglicht werden auszusagen, von welchem Typ sie ist. RDF bietet als einziges Konzept zur Typisierung das `type`-Element. Dieser ist aber nicht genügend aussagekräftig, um beispielsweise eine Taxonomie von Begriffen zu erzeugen. Dafür werden weitere Konzepte (Klasse, Eigenschaft) benötigt. RDFS stellt Mittel zur Verfügung, um Klassen und Unterklassen, Eigenschaften und Untereigenschaften zu definieren. Über eine Ressource sind dann Aussagen möglich, dass bzw. ob sie beispielsweise eine Eigenschaft, eine Klasse oder ihre Instanz ist.
2. **Einschränkungen von Eigenschaften:** Im Gegensatz zu einem objektorientiertem Typsystem besitzen Klassen keine Eigenschaften, sondern wird vielmehr ein Ansatz verfolgt, für die Eigenschaften Klassen anzugeben, auf die sie angewandt werden können. Diese ungewöhnliche Modellierung ist erforderlich, da es in einem globalen System nicht von vorneherein möglich ist

[15] Der Namensraum für dieses Vokabular ist `http://www.w3.org/2000/01/rdf-schema#`. Als Präfix für diesen Namensraum wird üblicherweise `rdfs` verwendet.

einzuschränken, was jemand über eine Ressource aussagen will, während es sehr wohl möglich ist, sinnvolle Argumente einer Eigenschaft zu definieren (z.B. das nur Menschen Väter haben können). RDFS definiert einen einfachen Mechanismus zur Beschreibung von Ressourcen, mit denen eine Eigenschaft kombiniert werden kann, also welche Ressourcen Subjekt und Objekt von einer Eigenschaft sein dürfen.

2.3.1 Typsystem

Das Typsystem von RDFS ist stark an ein objektorientiertes Typsystem angelehnt, allerdings stark vereinfacht. Es gibt drei zentrale Klassen zur Klassifizierung von Entitäten:

- **rdfs:Resource:** Alle Entitäten, die durch RDF Ausdrücke beschrieben werden, sind Ressourcen. Diese ist die Oberklasse aller Klassen, die RDFS definiert.
- **rdfs:Class:** Diese Unterklasse von `rdfs:Resource` repräsentiert das generische Konzept eines Typs oder eine Kategorie, vergleichbar mit dem Begriff der Klasse in objektorientierten Sprachen. Sie enthält also all jene Ressourcen, die für einen bestimmten Typ von Ressourcen stehen (z.B. Autos, Tiere, Menschen, Väter). Alle Ressourcen von Typ `rdfs:Class` sind Klassen. Unter anderem sind es auch `rdfs:Resource`, `rdf:Property` und `rdfs:Literal`.
- **rdf:Property:** Wiederum eine Unterklasse von `rdfs:Resource`, stellt `rdf:Property` die Sammlung aller Ressourcen dar, die eine Eigenschaft im Sinne des RDF Datenmodells repräsentieren (z.B. „`hat Autor`“, „`hat Vater`“, „`ist ein`“).

Neben diesen drei Klassen gibt es unter anderem noch die Klasse `rdfs:Literal`. Das RDF-Modell lässt als Wert von Objekten auch Literale zu. Diese werden als Instanzen von `rdfs:Literal` betrachtet. Da Literale keine Ressourcen sind, ist `rdfs:Literal` keine Unterklasse von `rdfs:Resource` und somit gibt es keine Basisklasse, von der alle anderen Klassen abgeleitet sind.

Die gerade informell eingeführte Beziehung „`von Typ sein`“ (im Sinne von „`X` ist von Typ `Y`“) oder, wie sie auch bezeichnet wird, „`ist ein`“ Beziehung modelliert RDF

mittels Eigenschaft `rdf:type`. Diese Eigenschaft gibt also an, dass ein Ressource `X` Instanz der Klasse `Y` ist.

Die ebenfalls informell eingeführte Beziehung „`Oberklasse von`“ (bzw. „`Unterklasse von`“) modelliert RDFS mittels Eigenschaft `rdfs:subClassOf`. Sie impliziert, dass wenn eine Klasse `X` Unterklasse von Klasse `Y` ist, sind alle Instanzen von `X` auch Instanzen von `Y`. Diese klassenhierarchische Beziehung ist transitiv. Eine Klasse kann Unterklasse mehrerer anderer Klassen sein. In äquivalenter Weise führt RDFS eine hierarchische Beziehung unter Eigenschaften ein: `rdfs:subPropertyOf`.

Mit Hilfe hier definierten Klassen und Eigenschaften können nun die Beziehungen von RDFS in RDF formalisiert werden. Es ergibt sich dann der in Abbildung 2.6[16] dargestellte Graph (wegen der Übersichtlichkeit nur für die Grundklassen):

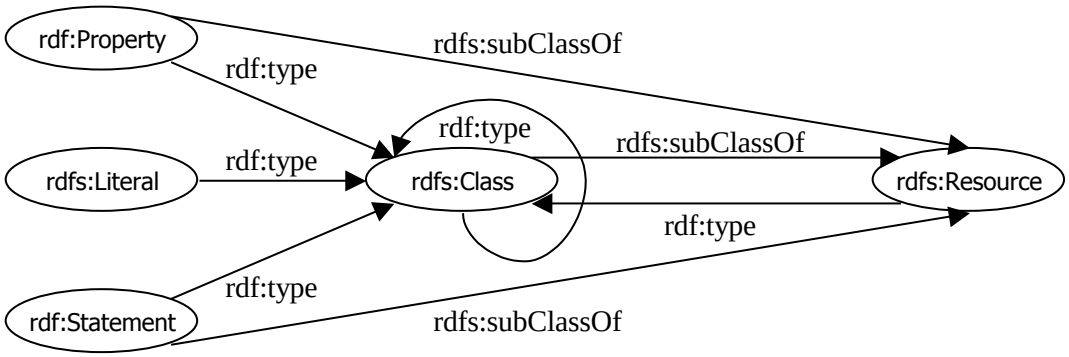


Abbildung 2.6: Klassenhierarchie und `type`-Beziehung der Modellierungsprimitiven in RDFS (in RDF).

Dadurch ist es nun leicht möglich, neue Klassen zu definieren und diese in Beziehung zu existierenden Klassen aus RDF Schema zu setzen. Dies wird am Beispiel in der Abbildung 2.7[17] demonstriert. Dort werden die Aussagen „`Bill hat den Vater Bob`“, „`Bill ist ein Mann`“ und „`Bob ist ein Mann`“ modelliert. Diesen liegt ein Schema zugrunde, in dem die Aussagen „`Männer sind eine Untermenge von Personen`“, und „`Frauen sind eine Untermenge von Personen`“ getroffen werden.

[16] Diese Abbildung wurde von der entsprechenden in [49] abgezeichnet.
[17] Diese Abbildung wurde von der entsprechenden in [49] abgezeichnet.

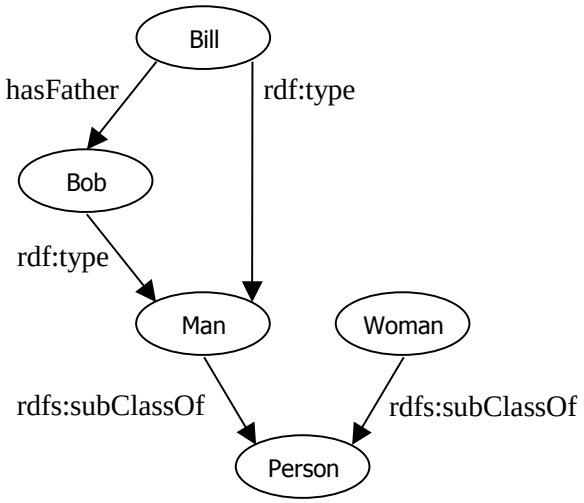


Abbildung 2.7: Graphenrepräsentation der Aussage „`Bill hat den Vater Bob`“.

2.3.2 Einschränkungen für Eigenschaften

Mittels des soeben definierten Typmodells ist es bereits möglich, festzulegen, welche Ressourcen als Eigenschaften zu interpretieren sind. Durch die im Folgenden vorgestellten Einschränkungen für Eigenschaften wird ein Mechanismus definiert, der erlaubt es festzulegen, mit welchen Klassen von Ressourcen (wiederum definiert durch das Typmodell) eine Eigenschaft kombiniert werden kann. Dafür definiert RDFS zwei auf Eigenschaften anwendbare Eigenschaften:

1. **`rdfs:domain`:** Legt den Anwendungsbereich einer Eigenschaft in Bezug auf eine Klasse fest. Damit stellt RDFS unter anderem folgende so genannte axiomatische Trippeln [48] auf:

   ```
   (rdfs:domain, rdfs:domain, rdf:Property)
   (rdfs:range, rdfs:domain, rdf:Property)
   (rdf:type, rdfs:domain, rdfs:Resource)
   (rdfs:subPropertyOf, rdfs:domain, rdf:Property)
   (rdfs:subClassOf, rdfs:domain, rdfs:Class)
   ```

   Die ersten beiden Tripel sagen beispielsweise aus, dass `rdfs:domain` bzw. `rdfs:range` auf Eigenschaften anwendbar ist, d.h. Subjekt eines Tripels, das als Prädikat `rdfs:domain` hat, ist eine Eigenschaft.

2. **`rdfs:range`:** Legt den Wertebereich einer Eigenschaft fest. Damit stellt RDFS unter anderem folgende axiomatische Trippeln auf:

   ```
   (rdfs:domain, rdfs:range, rdfs:Class)
   (rdfs:range rdfs:range rdfs:Class)
   ```

```
(rdf:type, rdfs:range, rdfs:Class)
(rdfs:subPropertyOf, rdfs:range, rdf:Property)
(rdfs:subClassOf, rdfs:range, rdfs:Class)
```

Die ersten beiden Tripel sagen beispielsweise aus, dass `rdfs:domain` bzw. `rdfs:range` Klassen als Wertebereich hat, d.h. Objekt eines Tripels, das als Prädikat `rdfs:range` hat, ist eine Klasse.

Eine Eigenschaft `e` darf keine oder eine Klasse, oder mehrere als Wertebereich aufweisen (sprich es darf von keinem bis mehreren `rdfs:range` für eine Eigenschaft angegeben werden). Im ersten Fall heißt es also, es wurde keine Aussage über den Wertebereich von `e` getroffen. Der zweiten Fall legt fest, dass `e` Instanzen angegebener Klasse als Werte haben kann. Genauer bedeutet dies, dass wenn `e`, angewandt auf eine Instanz des Anwendungsbereich von `e`, eine Instanz `i` als Wert hat, kann daraus gefolgert werden, dass `i` eine Instanz der als Wertebereich angegebener Klasse ist. Doch eine umgekehrte Schlussfolgerung, nämlich dass der Wertebereich von `e` eine Klasse `k` mit einschließt, falls `e`, angewandt auf eine Instanz von der Klasse aus dem Anwendungsbereich von `e`, als Wert eine Instanz von `k` hat, wird nicht gezogen. Schließlich legt der dritte Fall fest, dass wenn `e`, angewandt auf eine Instanz des Anwendungsbereich von `e`, als Wert eine Instanz `i` hat, dann gehört `i` zu allen als Wertebereich spezifizierten Klassen. Gleiches gilt auch für die Eigenschaften in Bezug auf `rdfs:domain` bzw. für die Instanzen, auf die diese Eigenschaften angewandt werden.

2.4 Beschreibungslogiken

Die Inhalte dieses Unterkapitels referenzieren auf Literaturquellen [25] bis [33], von denen jede an seiner Stelle einen Beitrag zu dieser Arbeit leistet. [25] ist dabei eine sehr empfehlenswerte, fundierte Darstellung von Beschreibungslogiken.

Innerhalb der logischen Sprachen existieren verschiedene Ansätze. Einer der bekanntesten darunter ist *Beschreibungslogiken*, auch *Description Logics* (*DLs*). Beschreibungslogiken stellen eine Teilsprache der Prädikatenlogik erster Stufe dar, die eine hohe Ausdrucksmächtigkeit im Bezug auf die Darstellung von strukturiertem Wissen besitzt, ohne dabei auf entscheidbare und effiziente Inferenzprozeduren zu

verzichten. [33] beschreibt den Kerngedanken von Beschreibungslogiken wie folgt: ”DLs decribe knowledge in terms of concepts and role restrictions that are used to automatically derive classification taxonomies.“

Von Beschreibungslogiken wird in Mehrzahl gesprochen, da es eine Familie von solchen gibt. Jede Beschreibungslogik weist die typischen Eigenschaften einer Logiksprache auf: eine Syntax und wohl definierte Semantik, und unterscheidet sich von einer anderen in erster Linie durch die Wahl von erlaubten syntaktischen Konstrukten, wodurch sich Unterschiede in der Ausdrucksmächtigkeit und Komplexität ergeben.

Die Semantik von in Beschreibungslogiken verwendeten Konstrukten wird üblicherweise modelltheoretisch angegeben, obwohl sie sich auch aus der Übereinstimmung der Letzteren mit prädikatenlogischen Formeln ergibt (siehe Anhang B).

2.4.1 Syntax und Semantik

In diesem Abschnitt werden syntaktische Elemente und die Konstrukten von Beschreibungslogiken dargestellt. Informell wird auch ihre Semantik (Bedeutung) erläutert; eine formale Semantik ist im Anhang A angegeben.

Die grundlegenden syntaktischen Elemente von allen Beschreibungslogiken sind folgende:

- *Individuen*: sie entsprechen Konstanten in der Prädikatenlogik erster Stufe (PL1). Jedes Individuum ist *Instanz* von mindestens einem Konzept.
- *atomare Konzepte:* sie sind äquivalent zu den unären Prädikaten der PL1 und werden als eine Menge von Individuen der Domäne interpretiert.
- *atomare Rollen*: sie sind äquivalent zu den binären Prädikaten der PL1 und werden als Menge von Paaren aus Individuen der Domäne interpretiert.

Aus den grundlegenden syntaktischen Elementen werden die sogenannten *Konzeptbeschreibungen* (auch Konzeptterme) gebildet, indem die unten aufgeführten (sowie gegebenenfalls auch andere) Konstrukte verwendet werden.

Seien *A*, *B* atomare Konzepte, *R*, *S* atomare Rollen, R^+ – transitive Hülle von *R*, und *C*, *D* Konzeptbeschreibungen. Dann stellt jeder der folgenden Ausdrucke eine Kontextbeschreibung dar [30]:

Bez.	**Syntax, Konstruktbezeichnung und informale Semantik am Beispiel**
∘	*A* (ein *atomares Konzept* wie „Mann“, „Frau“, „Mobiltelefon“),
∘	$\top$ (das *universelle Konzept*, zu dem alle Individuen gehören),
∘	$\bot$ (das *leere Konzept*, zu dem kein Individuum gehört),
∘	$\neg A$ (die *atomare Negation*, die bedeutet: alles, was nicht zum atomaren Konzept *A* gehört),
∘	$C \sqcap D$ (der *Durchschnitt* bzw. die *Konjunktion*, was bedeutet: alle Individuen, die sowohl mit der Konzeptbeschreibung *C* als auch mit dem Konzeptbeschreibung *D* beschrieben werden),
∘	$\forall R.C$ (die *Wertrestriktion*; zum Beispiel `∀besitzt.Mobiltelefon`, was bedeutet: alle Individuen, die als Eigenschaft „`besitzt`“ ausschließlich Individuen haben, die zum Konzept „`Mobiltelefon`“ gehören),
∘	$\exists R.\top$ (die *limitierte Existenzquantifikation*; zum Beispiel `∃name.⊤`, was bedeutet: alle Individuen, die über die Eigenschaft „`name`“ mindestens ein anderes Individuum besitzen, sprich, mindestens einen Namen haben).
$(\mathcal{U})$	$C \sqcup D$ (die *Vereinigung* bzw. die *Disjunktion*, die bedeutet: alle Individuen, die durch den Konzeptbeschreibung *C* oder den Konzeptbeschreibung *D* beschrieben werden),
$(\mathcal{E})$	$\exists R.C$ (die *volle existenzielle Restriktion*; zum Beispiel `∃nutzt.Mobiltelefon` bedeutet: Alle Individuen, die über die Eigenschaft „`nutzt`“ mindestens ein Individuum des Konzeptes „`Mobiltelefon`“ besitzen),
$(\mathcal{C})$	$\neg C$ (die *volle Negation*, die bedeutet: alle Individuen, die nicht durch den Konzeptterm *C* beschrieben werden),
$(\mathcal{N})$	$\geq_n R$ und $\leq_n R$ (die *Kardinalitätsrestriktionen*; zum Beispiel `≥2hatFreund`, was bedeutet: alle Individuen, die mindestens zwei Eigenschaften „`hatFreund`“, sprich zwei Freunde, besitzen),
$(\mathcal{H})$	$R \sqsubseteq S$ (die *Rollenhierarchie*, zum Beispiel: `hatTocher ⊑ hatKind`),

(O)	*a* oder {*a*} (der *Singleton* Konstrukt für ein Konzept, das nur aus dem Individuum *a* besteht),
(I)	$R \equiv S^-$ (die *inversen Rollen*, zum Beispiel: `hatTochter` ≡ `istTochterVon`$^-$),
(F)	$\top \sqsubseteq \leq 1\, R$ (die *funktionale Rolle*, zum Beispiel: $\top \sqsubseteq \leq 1$ `hatMutter`, was bedeutet: alle Individuen mit höchstens einer Eigenschaft „`hatMutter`“),
∘	$R^+ \sqsubseteq R$ (die *transitive Rolle*, zum Beispiel: `hatVorfahren`$^+$ ≡ `hatVorfahren`, was bedeutet: wenn `X` ein Vorfahren von `Y` ist und `Y` – ein Vorfahren von `Z`, ist `X` ein Vorfahren von `Z`),
(Q)	$\geq n\, R.C$ und $\leq n\, R.C$ (die *qualifizierte Kardinalitätsrestriktion*; zum Beispiel ≥ 2`hatFreund.Frau`, was bedeutet: alle Individuen, die mindestens zwei Freundinnen haben).

Mit Hilfe dieser Konstrukte kann man zum Beispiel eine Konzeptbeschreibung für: „eine männliche Person, deren alle Kinder Ärzte sind oder selbst einen Arzt als Kind haben“ wie folgt definieren:

$$\texttt{Mann} \sqcap \forall \texttt{hatKind.(Arzt} \sqcup \exists \texttt{hatKind.Arzt)}$$

2.4.2 Familie von Beschreibungslogiken

Wie bereits erwähnt, wird eine Beschreibungslogik dadurch bestimmt, welche Fragmente der PL1 sie zulässt. Die minimale Sprache wurde 1991 mit der *attributive language* $\mathcal{AL}$ eingeführt. Sie erlabt die ersten sieben der oben angegeben Konstrukte. Diese Sprache wird erweitert, indem weitere Fragmente der PL1 als Konstrukten erlaubt werden. Es ist eine Konvention, dabei den Namen der Beschreibungslogiken um die jeweiligen Stellvertreterbuchstaben (die linke Spalte der Tabelle) der Konstrukte zu erweitern (außer bei transitiven Rollen). Eine Sprache die $\mathcal{AL}$ um transitive Rollen und volle Negation $\mathcal{C}$ erweitert, wird oft mit $\mathcal{S}$ abgekürzt. Eine wichtige Beschreibungslogik heißt deshalb $\mathcal{SHIQ}$. Ihre Spezialform $\mathcal{SHOIN}(\mathcal{D})$[18] ist unter anderem die Basis für Dialekte der *Web Ontology Language* (*OWL*), die in 2.5 vorgestellt wird.

[18] $\mathcal{D}$ steht für einfache Datentypen wie Zahlen und Zeichenketten (siehe 2.5).

Die Menge von Konstrukten für Beschreibungslogiken ist redundant. Zum Beispiel ist $\mathcal{C}$ äquivalent zu $\mathcal{UE}$, denn es gilt:

$$C \sqcup D \equiv \neg (\neg C \sqcup \neg D)$$

$$\exists R.C \equiv \neg \forall R. \neg C$$

Ebenfalls gilt, dass $\mathcal{F}$ von jeder Beschreibungslogik, die $\mathcal{N}$ erlaubt, impliziert wird (die in $\mathcal{F}$ verwendete $\sqsubseteq$ Relation ist in jeder Beschreibungslogik verwendbar; sie wird in 2.4.3 erläutert).

2.4.3 A-Box und T-Box

Ein Wissensrepräsentationssystem, das auf einer Beschreibungslogik basiert, bietet die Möglichkeit, Wissensbasen zu realisieren. Eine Wissensbasis beinhaltet in der Regel Wissen über eine bestimmte Domäne oder mehrere Domänen. Dieses Wissen ist von zwei Kategorien: *intensional*, d.h. statisch, unabhängig von einer Diskursdomäne vs. *extensional*, d.h. dynamisch, veränderlich mit der Zeit. Das intensionale Wissen wird in einer *T-Box* (vom englischen „*terminology box*“) festgehalten, und zwar in Form von (atomaren) Konzepten und Rollen, welche die Terminologie, d.h. das Vokabular zum Beschreiben einer Domäne und zum benennen von Kontextbeschreibungen, ausmachen. Das extensionale Wissen wird in einer *A-Box* (vom englischen „*assertional box*“) festgehalten, und zwar in Form von Individuen in Begriffen der Terminologie. Jeder Term von A-Box wie auch T-Box wird *Axiom* genannt.

Eine T-Box besteht aus Axiomen folgender Arten (Variablenbezeichnungen wie in 2.4.1) [26]:

$$C \sqsubseteq D, \quad C \equiv D, \quad R \sqsubseteq S, \quad R \equiv S, \quad R^+ \sqsubseteq R,$$

Dabei wird mit $\sqsubseteq$ die sogenannte *Subsumierungsrelation* bezeichnet, die zu folgenden PL1-Formeln äquivalent ist:

$$C \sqsubseteq D \text{ gdw. } \forall x.C(x) \rightarrow D(x)$$

$$R \sqsubseteq S \text{ gdw. } \forall x,y.R(x,y) \rightarrow S(x,y)$$

So ist es möglich für die obige Konzeptbeschreibung „einer männlichen Person, deren alle Kinder Ärzte sind oder selbst einen Arzt als Kind haben“, ein atomares Konzept `Ärztefamilieoberhaupt` einführen:

`Ärztefamilieoberhaupt` $\sqsubseteq$ `Mann` $\sqcap$ $\forall$`hatKind.(Arzt` $\sqcup$ $\exists$`hatKind.Arzt)`

Eine A-Box besteht aus Axiomen folgender Arten, die wie angegeben äquivalent zu den PL1-Formeln sind:

$$x : C \text{ gdw. } C(x)$$

$$\langle x, y \rangle : R \text{ gdw. } R(x, y),$$

wobei x und y Namen von Individuen sind.

Für jedes atomare Konzept *A* wird der rechte Teil des Axioms $A \equiv C$ als *Definition* von *A* bezeichnet. Wenn eine fortlaufende Ersetzung von nicht atomaren Konzepten in der Definition von *A* durch die Definition dieser Konzept terminiert, dann wird dass Ergebnis dieser Ersetzung als *Erweiterung* von *A* bezeichnet. Wenn jedes atomare Konzept einer T-Box eine Erweiterung hat, wird diese T-Box als *azyklisch* bezeichnet; sonst ist sie *zyklisch*. Als Erweiterung einer T-Box wird das Ergebnis von Ersetzung der Definition jedes atomaren Konzepts durch seine Erweiterung bezeichnet. Es gilt also, dass in der Erweiterung einer T-Box keine nichtatomare Konzepte vorkommen. Wenn eine T-Box $\mathcal{T}$ azyklisch ist, lässt sie sich in folgendem Sinne eliminieren: für jedes Axiom *X*, das entweder eine T-Box-Axiom oder eine in Begriffen von $\mathcal{T}$ formulierte A-Box Axiom ist gilt: *X* ist gültig bezüglich $\mathcal{T}$ genau dann, wenn *X* ist gültig bezüglich der Erweiterung von $\mathcal{T}$.

2.4.4 Inferenz

Implizites Wissen über Konzepte und Individuen kann automatisch mit Hilfe von Inferenzprozeduren geschlossen werden. Für diese Zwecke wünscht man sich folgende Inferenzdienste (zusammenfassend):

- *Subsumierung*: ob zwischen zwei Konzepten die Subsumierungsrelation gilt,
- *Erfüllbarkeit*: ob ein Konzept erfüllt ist, d.h. ob für ihn ein nicht leeres Modell existiert,
- *Konsistenz*: ob eine, in Begriffen einer T-Box formulierte A-Box ein mit dieser T-Box gemeinsames Modell hat,
- *Instanziierung*: ob ein Individuum *c* Instanz von Konzept *C* aus einer T-Box $\mathcal{T}$ ist, d.h. ob *C*Priorität und $\mathcal{T}$ ein gemeinsames Modell haben.

[32] zeigt, dass alle diese Inferenzdienste sowohl auf Konsistenz als auch auf Instanziierung allein reduziert werden können. Außerdem lässt sich Subsumierung auf

Erfüllbarkeit reduzieren, und umgekehrt. Von der Reduktion auf Konsistenz profitieren die sogenannten *Tableau Algorithmen*. Sie versuchen für eine, in Begriffen einer T-Box $\mathcal{T}$ formulierte A-Box durch die Eliminierung von $\mathcal{T}$ und die Anwendung von speziellen Tableau Regeln ein Modell für diese A-Box zu konstruieren. Die Tableau Algorithmen sind heute die effizientesten Inferenzalgorithmen. (Sie werden jedoch hier nicht erläutert, da sie nicht zum Gegenstand dieser Arbeit gehören.)

Die Komplexität von Inferenz in einer Beschreibungslogik-basierten Wissensbasis wächst mit ihrer Ausdrucksmächtigkeit. Es kommt aber auch darauf an, ob die T-Box einer Wissensbasis zyklisch ist. So gibt [31] für einige Beschreibungslogiken folgende Worst-Case Komplexität an:

Beschreibungslogik	**Azyklische T-Box**	**Zyklische T-Box**
$\mathcal{SHIF}(\mathcal{D})$	NP	Exptime
$\mathcal{SHIN}(\mathcal{D})$	NP	Exptime
$\mathcal{SHOIN}(\mathcal{D})$	unbekannt	Nexptime

Abbildung 2.8: Komplexität einiger Beschreibungslogiken.[19]

Um ein Gespür für solche hohe Komplexität zu bekommen, sei es erwähnt, dass bei einer in der Beschreibungslogik $\mathcal{EL}$[20] formulierten T-Box aus 2740 Konzepten und 413 Rollen(leider ohne Angabe zu der Zyklenfreiheit) benötigte das Pellet Reasoner (siehe 2.7) 75 Sekunden, um eine Inferenzaufgabe zu lösen [32]. Die anderen Reasoner haben laut dieser Quelle 14 bis 50 Sekunden für dieselbe Aufgabe gebraucht.

2.5 Ontology Web Language

Ontology Web Language (*OWL*) ist eine von World Wide Web Consortium standardisierte Sprache zur Definition und Realisierung von Web Ontologien. OWL basiert auf Beschreibungslogiken und existiert in drei Ausführungen: *OWL Lite*, *OWL DL* und *OWL Full*. OWL Lite ist äquivalent zu der Beschreibungslogik $\mathcal{SHIF}(\mathcal{D})$; OWL DL – zu $\mathcal{SHOIN}(\mathcal{D})$. Es gelten also für diese Varianten von OWL bezüglich der

[19] Diese Abbildung wurde in einer überarbeiteten Form aus [31] übernommen.
[20] $\mathcal{L}$ steht hier für die Konjunktion zweier Konzepte.

Ausdruckmächtigkeit und Komplexität die obigen Ausführungen (siehe 2.4 und die Abbildung 2.9).

Das Kürzel $\mathcal{D}$ steht für konkrete Datentypen wie zum Beispiel XML-Schema Datentypen. Dazu gehören beispielsweise Zahlen (`xsd:integer`, `xsd:long`, und andere) und Zeichenketten (`xsd:string`). Dabei hat jeder Datentyp eine feste Interpretationsdomäne. So ist es möglich, eine sogenannte Datentyprolle oder, in Begriffen von RDF(S), eine Eigenschaft zu definieren, die als Wertebereich einen bestimmten Datentyp hat. Zum Beispiel kann man die Eigenschaft „Geburtsjahr“ durch folgende RDF-Tripel angeben:

```
(Geburtsjahr, rdf:type, owl:DatatypeProperty)
(Geburtsjahr, rdfs:range, xsd:integer)
```

Die OWL Full verfügt über dieselben syntaktischen Konstrukten wie OWL DL, verzichtet jedoch auf die Einschränkung, dass ein Konzept nicht ein Individuum eines anderen Konzeptes sein darf. Die Aufhebung dieser Einschränkung, sowie der Verzicht auf einige weitere, führt dazu, dass OWL Full, im Gegensatz zu den anderen OWL Varianten, die Ausdrucksmächtigkeit von RDF(S) erlangt (obwohl RDF(S) bezüglich der Interpretierbarkeit von Maschinen jeder OWL Variante weit unterlegen ist). Ein daraus resultierender Nachteil von OWL Full, wie auch bei RDF(S), ist der Verlust der Entscheidbarkeit. Dies äußert sich auch darin, dass es heutzutage kein vollständiges Inferenzsystem für OWL Full gibt.

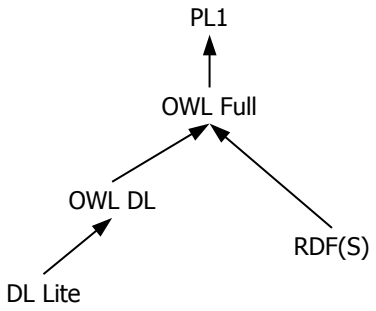


Abbildung 2.9: Ausdrucksmächtigkeit logischer Sprachen.

2.5.1 Syntax und Semantik

OWL[21] definiert ein Vokabular. Da OWL auf Beschreibungslogiken basiert, kann Semantik dieses Vokabulars durch Entsprechung den Konstrukten bzw. Termen der Beschreibungslogik angegeben werden. Die vollständige Semantik findet man in [37]. In der Abbildung 2.10 findet man die Entsprechung von wichtigsten OWL-Vokabeln.

`Class`	A	`minCardinality`	$\geq n\ R.C$
`Property`	R	`maxCardinality`	$\leq n\ R.C$
`subClassOf*`	$C \sqsubseteq D$	`disjointWith`	$C \sqsubseteq \neg D$
`equivalentClass`	$C \equiv D$	`sameAs`	$\{c_1\} \equiv \{c_2\}$
`equivalentProperty`	$R \equiv S$	`differentFrom`	$\{c_1\} \sqsubseteq \neg\{c_2\}$
`subPropertyOf*`	$R \sqsubseteq S$	`inverseOf`	$R \equiv S^-$
`intersectionOf`	$C \sqcap D$	`TransitiveProperty`	$R^+ \sqsubseteq R$
`unionOf`	$C \sqcup D$	`FunctionalProperty`	$\top \sqsubseteq \leq 1\ R$
`complementOf`	$\neg C$	`InverseFunctionalProperty`	$\top \sqsubseteq \leq 1\ R^-$
`oneOf`	$\{c_1, \ldots, c_n\}$	`range*`	$\top \sqsubseteq \forall R.C$
`someValuesFrom`	$\exists R.C$	`domain*`	$\exists R.\top \sqsubseteq C$
`allValuesFrom`	$\forall R.C$	`c type* C`	$c : C$
`hasValue`	$\exists R.\{c\}$	c_1 `R` c_2	$\langle c_1, c_2 \rangle : R$
`cardinality`	$=n\ R.C$		

Abbildung 2.10: Semantik von OWL Vokabeln durch die Angabe entsprechender beschreibungslogischer Terme. Die mit * markierte Vokabeln sind aus dem RDF(S) Vokabular übernommen worden.

Syntaktisch basiert OWL auf RDF. Jedes OWL Dokument ist auch ein RDF Dokument. Dies impliziert, dass jedes OWL Dokument syntaktisch im Grunde genommen als eine Menge von RDF-Tripeln gesehen werden kann. Für die vollständige OWL Syntax sei es auf [37] verwiesen. Hier wird nur ein Beispiel demonstriert, in dem das Konzept `Ärztefamilieoberhaupt` aus 2.4.3 definiert wird:

[21] Hier und im Weiterem sei mit „OWL“ die Variante OWL DL gemeint.

```
1   <owl:Class rdf:about="#Ärztefamilieoberhaupt „>
2      <owl:intersectionOf rdf:parseType=" collection">
3         <owl:Class rdf:about="#Mann"/>
4         <owl:Restriction>
5            <owl:onProperty rdf:resource="#hatKind"/>
6            <owl:allValuesFrom >
7               <owl:unionOf rdf:parseType=" collection">
8                  <owl:Class rdf:about="#Arzt"/>
9                  <owl:Restriction>
10                    <owl:onProperty rdf:resource="#hatKind"/>
11                    <owl:someValuesFrom rdf:resource="#Arzt"/>
12                 </owl:Restriction>
13              </owl:unionOf>
14           </owl:allValuesFrom >
15        </owl:Restriction>
16     </owl:intersectionOf>
17  </owl:intersectionOf>
```

Es wird in Zeilen 1 und 17 das atomare Konzept `Ärztefamilieoberhaupt` definiert. Die Definition selbst ist eine Konzeptbeschreibung in Zeilen 2 bis 16. Eine Konzeptbeschreibung ist in OWL eine, im Gegensatz zu den atomaren Klassen, namenslose Klasse, die aus Konzeptbeschreibungen zusammengesetzt ist, die mit Konjunktionen und Desjunktionen miteinander verbunden sind. Ist eine solcher Konzeptbeschreibungen keine atomare Klasse oder deren Negation, dann handelt es sich um eine *Restriktion* (Einschränkung), also um eines der folgenden beschreibungslogischen Konstrukten: Werterestriktion, limitierte Existenzquantifikation, volle existentielle Restriktion oder (qualifizierte) Kardinalitätsrestriktion (siehe 2.4.1). Im obigen Beispiel werden zwei Restriktionen definiert: Zeilen 4 und 15 bzw. 9 und 12. Eine Restriktion ist in OWL eine namenslose Klasse, die durch das Element `owl:Restriction` aus dem OWL Vokabular dargestellt wird. Bei einer Restriktion gibt es immer eine Eigenschaft (Rolle), auf die sich die Einschränkung bezieht. Diese wird mit `owl:onProperty` spezifiziert (z.B. Zeile 10). Die Angabe der Restriktionsart erfolgt innerhalb einer Restriktion unmittelbar nach der Spezifikation der Eigenschaft (z.B. Zeile 11).

2.5.2 Import von Ontologien

OWL ist ein Teil der Semantic Web Aktivitäten. Diese zielen darauf ab, Ressourcen im Web für automatisierte Prozesse zugänglicher zu machen, indem man Informationen

über diese Ressourcen hinzufügt. Da das Semantic Web von Haus aus ein verteiltes Netz ist, muss es in OWL möglich sein, Information aus verteilten Quellen zu beziehen. Dies wird zum Teil dadurch erreicht, dass den Ontologien erlaubt wird in Beziehung miteinander zu stehen, insbesondere durch expliziten Import von Information aus anderen Ontologien. Der Import einer weiteren Ontologie bringt also die gesamte Menge an Aussagen aus jener Ontologie in die gegenwärtige Ontologie. Außerdem macht OWL eine Open World Annahme. Das bedeutet, dass Beschreibungen von Ressourcen nicht auf eine einzige Datei oder einen einzigen Gültigkeitsbereich beschränkt sind. Eine Klasse C mag ursprünglich in der Ontologie O definiert worden sein, kann jedoch in anderen Ontologien erweitert werden. Die Folgen dieser Erweiterungen von C sind monoton. Neue Information kann vorhergehende Information nicht überschreiben. Neue Information kann widersprüchlich sein, aber Fakten und Schlussfolgerungen können nur hinzugefügt, jedoch nie entfernt werden.

2.6 Ontologisches Umfeld in SmartWeb

Intelligentes, domänenübergreifendes Dialogsystem wie SmartWeb benötigt eine Repräsentation von Wissen aus verschiedenen Domänen, zum Beispiel Navigations- und Diskurskontext des dialogführenden Benutzers, Fragen, die er möglicherweise stellen würde, Informationen die über die Web Services zugänglich sind usw. Zur Integration dieses Wissens benötigt man, wie in 2.1.5 argumentiert, eine Grundontologie.

2.6.1 Grundontologie für SmartWeb

Ausgehend von den Anforderungen von SmartWeb und einer Reihe von Kriterien bezüglich der grundsätzlichen Prinzipien, die einer Grundontologien unterliegen können (siehe 2.1.4, bzw. detaillierte Diskussion in [38],[39]), wurden mehrere etablierte Grundontologien untersucht. Dabei wurde für SmartWeb folgende Wahl getroffen. Zwei Grundontologien, Suggested Upper Merged Ontology (SUMO)[22] und Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE)[23] [40], wurden überarbeitet, um Vorteile der jeweiligen Grundontologie in einer, für SmartWeb gemeinsamen Grundonologie, SmartSUMO [38] zu vereinen.

[22] http://ontology.teknowledge.com/
[23] http://www.loa-cnr.it/Ontologies.html

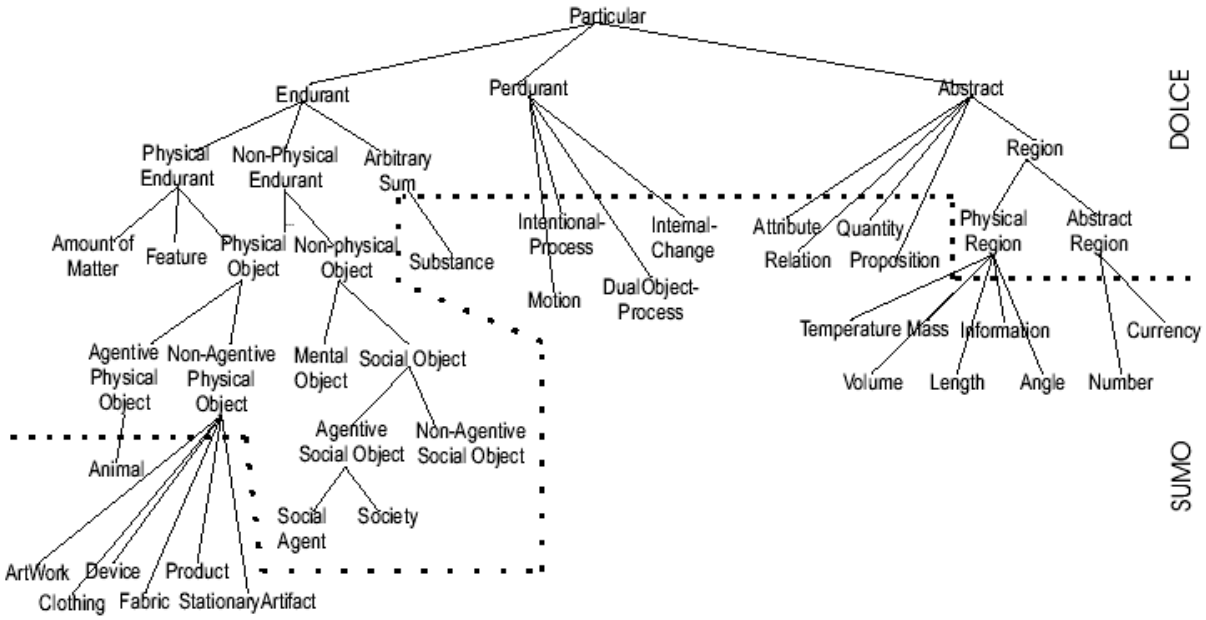


Abbildung 2.11: Die obersten Konzepte der SmartSUMO Taxonomie.

Die obersten Konzepte von SmartSUMO sind in der Abbildung 2.11[24] dargestellt. Die punktierte Linie teilt die Konzepte in die SUMO bzw. DOLCE angehörige auf, wobei es deutlich wird, dass die höchste Ebene von DOLCE über die oberen Konzepte von SUMO platziert sind. Dies, weil DOLCE sehr abstrakt ist und nur die generellsten Konzepte modelliert, und SUMO über eine sehr extensive und detaillierte Taxonomie verfügt [38]. Somit beruht SUMO auf in DOLCE verankerten fundamentalen Unterschied zwischen *Enduranten* (z.B. Objekte, Substanzen) und *Perduranten* (z.B. Ereignisse, Prozesse). Dieser kann wie folgt beschrieben werden: Enduranten existieren in der Zeit und nehmen in Perduranten teil. Zum Beispiel eine Person, die ein Endurant ist, beteiligt sich an ihrem eigenen Leben, das ein Perdurant ist.

Weiterhin übernimmt SmartSUMO die in DOLCE auf demselben Abstraktionsniveau wie die Perduranten und Enduranten eingeführten, Eigenschaften repräsentierenden Klassen (Qualities). Sie sind in der Abbildung 2.11 nicht eingezeichnet. Diese stellen die wahrnehmbare oder messbare Entitäten dar, zum Beispiel Farben, Größen, Formen, Geräusche, Gerüche, Gewichte usw. Qualities sind in temporale (zeitliche Lokation), physikalische (u.a. räumliche Lokation, Wetter- und Straßenverhältnisse, emotionale Zustände) und abstrakte unterteilt.

[24] Diese Abbildung wurde aus [38] übernommen.

Schließlich ist auf derselben Abstraktionsniveau die Klasse `Abstract` definiert, die selbst keine Eigenschaft (Qualities) ist und keine temporale oder räumliche Eigenschaften besitzt. Zum Beispiel `Region`, eine Unterklasse von `Abstract`, korrespondiert mit den Eigenschaften und wird benutzt, um ihre Messwerte zu enkodieren. So wurde die Klasse `WeatherRegion` definiert, um konkrete, von entsprechendem Web Service ermittelte Ausprägung der Eigenschaft `WeatherQuality` zu repräsentieren.

2.6.2 Domänenontologien in SmartWeb

SmartSUMO wurde zusammen mit den für SmartWeb entwickelten und eingesetzten Domänenontologien in eine gemeinsame Ontologie integriert, die SmartWeb Integrated Ontology (SWIntO)[25] [38]. Die einzelnen Domänenontologien werden in [38] ausführlich dargestellt. Es handelt sich dabei um eine Ontologie, die Sport bzw. sportliche Ereignisse, insbesondere im Hinblick auf die Fußballweltmeisterschaft 2006, repräsentiert; eine Navigationsontologie, die dem Benutzer des Dialogsystems bei der Wegfindungsaufgabe assistiert; eine Diskursontologie, die Interaktion mit dem Benutzer modelliert; eine multimediale Ontologie, die Multimediadaten wie Webcam Streams oder Bilder umschreibt; und schließlich eine Ontologie für die linguistischen Informationen, die eine ontologiebasierte Informationsextraktion aus Text sowie semistrukturierten Daten unterstützt. SWIntO stellt also die Gesamtheit dieser Ontologien dar, die im Weiteren als *Basisontologie* bezeichnet wird. (Übertragen auf einen Allgemeinfall wird als Basisontologie einer Anwendung die durch den Zusammenschluss von dort eingesetzten Grund- und Domänenontologien entstandene Ontologie bezeichnet.)

2.6.3 Basisontologie

SWIntO verwendet vergleichsweise wenige Konstrukte: es entspricht der $\mathcal{EL}\,\mathcal{I}\,\mathcal{H}\,\mathcal{N}(\mathcal{D})$ Beschreibungslogik. Dennoch bedarf es, wegen der Kardinalitätsrestriktionen ($\mathcal{N}$), eines OWL DL mächtigen Reasoner zum vollständigen Inferieren innerhalb SWIntO. SWIntO definiert 2356 Konzepte, keine Individuen und 1063 Rollen (davon 413

[25] http://www.smartweb-project.de/ontology.html

Datentyprollen). Dies führt, insbesondere in Anbetracht des Kapitels 2.4.4 dazu, dass das vollständige Inferieren in SWIntO mit Hilfe von Pellet Reasoner als sehr zeitaufwändig zu erwarten ist. In Praxis hat es sich als unmöglich erwiesen (siehe 9.2.3).

2.7 Jena Semantic Web Framework

Das *Jena Semantic Web Framework (Jena)* [48] wird maßgeblich von den HP Labs[26] im Rahmen ihrer Forschungen im Bereich Semantic Web entwickelt. Es handelt sich bei Jena um Open-Source Software, die online verfügbar ist. Das Projekt bezeichnet sich als das führende Werkzeug zur Entwicklung von Semantic Web Applikationen mit Java. In dieser Arbeit wird die zur Zeit aktuellste Version Jena2, Version 2.4 verwendet.

Jena stellt dem Entwickler eine Reihe von APIs zum Lesen, Modifizieren und Schreiben von RDF(S) und OWL Ontologien, sowie zum Inferieren in solchen zur Verfügung. Im Weiteren werden einige dieser Funktionalitäten an einfachen Beispielen erläutert.

Die interne Repräsentation eines RDF- oder OWL-Dokuments wird in Jena als (RDF) *Modell* bezeichnet. Dabei unterscheidet Jena zwischen Modell (Java-Klasse `Model`), wie dies in RDF(S) aufgefasst wird, und ontologischem Modell (von `Model` abgeleiteter Klasse `OntModell`), das das erste um Konzeption und entsprechendes Vokabular von OWL erweitert. Folgender Quellcode zeigt, wie ein ontologisches Modell, das eine unter `C:/Ontology.owl` gespeicherte Ontologie repräsentiert, erzeugt werden kann (ein RDF(S) Modell wird ähnlich erzeugt):

```
OntModel m =
  ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM_TRANS_INF);
m.read("C://Ontology.owl");
```

Dabei werden über eine Konstante der Klasse OntModelSpec – in diesem Fall `OWL_DL_TRANS_INF` – drei wichtige Optionen der Modellerzeugung festgelegt:

- Reasoner, das mit dem Modell verbunden wird; hier wird ein transitives Reasoner verwendet (s.u.). Es können auch Modelle ohne Reasoner erzeugt werden.

[26] http://hpl.hp.com/semweb/

- Sprachformalismus, in dem die Ontologie formuliert ist (hier: OWL DL).
- Speicherungsmethode des Modells: hier wird ein „In-Memory“ Modell erzeugt (im Gegensatz zum persistenten Model, das sich bspw. direkt mit einer relationalen DB verbinden ließe).

Neben `Model` und `OntModel` gehören folgende Klassen bzw. Datentypen, die Jena zur Verfügung stellt, zu den für diese Arbeit relevanten:

- `OntClass`: Repräsentiert eine ontologische Entität, die in OWL Ontologie als eine Klasse definiert ist. Da RDF(S) keine strikte Trennung zwischen Klassen und Individuen voraussetzt, werden in RDF Modellen Klassen als eine Art ontologischer Entitäten nicht unterschieden.
- `Property`, `OntProperty`: Repräsentiert eine RDF(S) bzw. OWL Eigenschaft (also eine ontologische Relation).
- `Individual`: Repräsentiert eine ontologische Entität, die in OWL Ontologie als ein Individuum definiert ist. Da RDF(S) keine strikte Trennung zwischen Klassen und Individuen voraussetzt, werden in RDF Modellen Individuen als eine Art ontologischer Entitäten nicht unterschieden.
- `Resource`, `OntResource`: Repräsentiert ontologische Ressourcen, die einen URI haben. Dies sind unter anderem auch Klassen, Individuen und Eigenschaften.
- `RDFNode`: Repräsentiert alle RDF Ressourcen, d.h. auch Literale, die keinen URI haben.

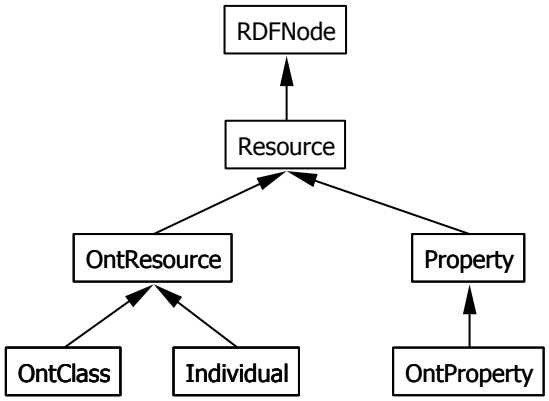


Abbildung 2.12: Hierarchische Beziehungen zwischen ausgewählten Klassen in Jena.

Die hierarchischen Beziehungen zwischen diesen Klassen sind in der Abbildung 2.12 dargestellt. Wobei Jena auch zwischen weiteren Typen von Ressourcen unterscheidet. Zum Beispiel verschiedene Arten von Restriktionen, die als Konzeptbeschreibugen, ebenfalls ontologische Klassen (bzw. Konzepte) sind. Ein Modell lässt sich über alle Ressourcen eines Typs (z.B. alle Individuen) oder über eine spezielle, durch ihren URI spezifizierte Ressource abragen. Zum Beispiel gibt

```
m.getOntClass(<uri>);
```

diejenige Klasse (also eine Instanz von `OntClass`) zurück, die im ontologischen Modell `m` den URI `<uri>` hat.

Jena stellt diverse Reasoner unterschiedlicher Ausdrucksmächtigkeit sowohl für RDF(S) als auch für OWL Ontologien zur Verfügung. Aus dieser Vielfalt sei nur das, mit der Basisontologie verknüpfte Reasoner genannt: Das ist das sogenannte transitive Reasoner, das nur eine transitive Hülle über `rdfs:subClassOf` und `rdfs:subPropertyOf` bilden kann. Somit ist die einzige Schlussfolgerung, die es anbietet (Entsprechendes auch für die Eigenschaften):

$$C \sqsubseteq C' \text{ genau dann wenn es Klassen } C_1,..C_n \text{ gibt mit } C \sqsubseteq C_1 \sqsubseteq C_2 \ldots \sqsubseteq C_n \sqsubseteq C',$$

wobei C und C' ontologische Klassen sind. Es wird angegeben, dass das transitive Reasoner bereits während des Aufbauens eines ontologischen Modells die beiden transitiven Hüllen bildet.

Neben der von Jena angebotenen Reasoner ist es möglich, *Pellet* mit einem Modell zu verknüpfen. Pellet ist ein Java basiertes Open-Source OWL DL Reasoner, das von University of Maryland's Mindswap Lab entwickelt wurde. Dieses unterscheidet sich von Reasoner in Jena dadurch, dass es komplett ist: laut der Angabe auf Internetseiten von Jena verfügen die Reasoner von Jena nur über die A-Box Inferenz, während Pellet auch T-Box Inferenz realisiert.

Die Verwendung von Pellet in Jena ist denkbar einfach: ein (ontologisches) Modell wird wie folgt erzeugt:

```
OntModel model = ModelFactory.createOntologyModel(
        PelletReasonerFactory.THE_SPEC);
```

Im Sonstigen ist die Verwendung von Pellet mit dem Gebrauch von in Jena integrierten Reasoner identisch.

Sämtliche Inferenzen (die für diese Arbeit relevant sind) und Abfragen können in einem RDF Modell mit Hilfe einer Methode veranlasst werden:

```
StmtIterator listStatements(Resource s, Property p, RDFNode o),
```

wobei `s`, `p` und `o` für Subjekt, Prädikat und Objekt (respektive) eines RDF-Tripels stehen und `StmtIterator` – ein Iterator über RDF-Statements ist. Diese Methode liefert also ein Iterator über alle RDF-Statements zurück, bei denen `s` als Subjekt, `p` als Prädikat und `o` als Objekt inferiert werden konnten. Dabei werden alle `null`-Parameter dieser Methode als freie Variablen interpretiert. Zum Beispiel liefert folgender Aufruf:

```
m.listStatements(null,RDF[27].type,(RDFNode)null);
```

alle RDF-Statements des Modells `m` zurück, deren Prädikat von dem mit `m` verknüpften Reasoner als `rdf:type` festgestellt wurde, also mit anderen Worten – alle Individuen von `m`. Dabei sollte es jedoch beachtet werden, dass die Subjekte zurückgegebener Statements keine Objekte von Typ `Individual` sind, sondern zu `Resource` gehören. Zu Individuen der entsprechenden OWL Ontologie müssen sie erst konvertiert werden, was in der Praxis leider aus ungeklärten Gründen nicht immer von Jena ermöglicht wird.

Für ontologische Modelle steht die obige Methode ebenfalls zur Verfügung. Es besteht aber auch eine Möglichkeit bestimmte Inferenzen über Methoden derjenigen Objekten auszuführen, die Entitäten eines ontologischen Modells darstellen. Zum Beispiel gibt

```
c.listSubClasses(boolean direct);
```

[27] `RDF` ist eine Klasse, die in Jena das Vokabular von RDF repräsentiert.

alle Unterklassen von `c` zurück (`c` ist eine Instanz von `OntClass`), wenn `direct false` ist. Falls `direct true` ist, werden nur diejenige Unterklassen von `c` zurückgegeben, die in der Taxonomie, bildlich gesprochen, nur über eine Kante mit `c` verbunden sind. Es werden also keine Unterklassen von Unterklassen von `c` zurückgegeben.

Zum Modifizieren einer Ontologie bietet Jena die Möglichkeit an, einem Modell einzelne RDF-Tripel oder ontologische Entitäten (Klassen, Relationen und Individuen) oder andere Modelle (d.h. alle ihre RDF-Tripel) hinzuzufügen. Jena ermöglicht auch diese Operationen rückgängig zu machen bzw. beliebige RDF-Tripel aus einem Modell zu entfernen.

Schließlich kann Jena auch ein (gegebenenfalls modifiziertes) Modell in eine Datei bzw. generell in ein Stream schreiben und zwar in unterschiedlichen RDF Syntaxen.

3 Aufgabenstellung am Beispiel eines typischen Anwendungsszenario in SmartWeb

Nun, nachdem die ontologischen Grundlagen gelegt wurden, ist es möglich die Aufgabestellung zu präzisieren, und zwar anhand eines Beispiels, das ein typisches Anwendungsszenario in SmartWeb aufzeigt.

In der Abbildung 3.1 ist ein Motivationsbeispiel skizziert, aus dem die typische Aufgabenstellung in SmartWeb hervorgeht. Auf der linken Seite ist die an das SitCom Modul übergebene Eingabe grundformartig dargestellt. Dies ist ein die Benutzeraussage semantisch in RDF(S) repräsentierendes Dokument, das aus Individuen basisontologischer Klassen besteht, die untereinander mit basisontologischen Relationen verbunden sind. Es handelt sich dabei also um eine A-Box der Basisontologie. In diesem Fall thematisiert die Benutzeraussage offensichtlich eine Transportierung von `X` nach `Y` mittels eines Transportweges `Z` des verallgemeinerten Typs „Strasse“. Dabei ist es davon auszugehen, dass `Z` ein leeres Individuum ist, welches impliziert, dass der Benutzer keine bestimmte Strasse angibt, die er befahren oder begehen möchte. Die Ausgabe ist ebenfalls eine A-Box, die in diesem Fall bis auf Zeile acht mit der eingegebenen A-Box übereinstimmt. Die achte Zeile präzisiert, dass `Z` eines spezielleren Typs „Landstrasse“ sein soll, begründet dadurch, dass es ein schönes, sonniges Wetter herrscht. Dabei bleibt `Z` ein leeres Individuum, weil es keine Aufgabe des SitCom Moduls ist, eine bestimmte Route auszurechnen, sondern ihre Eigenschaften und Parameter festzulegen. Jedoch kann es durchaus auch sein, dass zusätzlich konkrete Inhalte in Form von nicht leeren Individuen und relationaler Beziehungen zwischen ihnen der Benutzeraussage hinzugefügt werden, z.B. die ermittelten Angaben zur aktuellen Wetterlage.

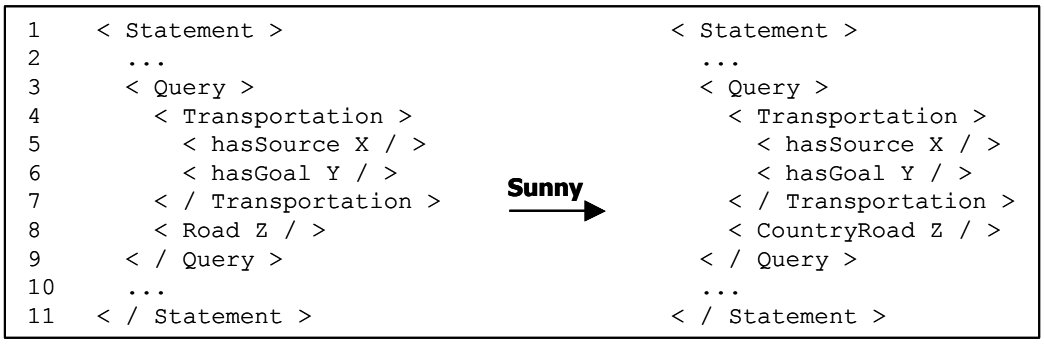


Abbildung 3.1: Durch eine beispielhafte Grundform der Eingabe (links) und Ausgabe (rechts) dargestellte typische Aufgabenstellung des Kontextmoduls SitCom.

Die Aufgabe ist es nun einen Mechanismus zu entwickeln und zu implementieren, der in der Lage ist die in der Abbildung 3.1 geschilderte Modifikation durchzuführen. Dabei ist es zu beachten, dass in der Abbildung 3.1 beschriebenes Beispiel einleitend und motivierend ist und keinesfalls den allgemeinen Fall darstellt. Die allgemeine Aufgabestellung ist zwar in 1.4 gegeben. Doch aussagekräftiger sind die in Kapiteln 6.5.6, 6.6 und 6.7 dargestellten Konzepte, die alle zusammen das allgemeine Aufgabenfeld am es besten beschreiben und charakterisieren. Das hier vorgestellte Beispiel wird dabei die folgenden Kapitel begleiten und wird dort konsequent in tiefergehenden Details herausgearbeitet.

4 Grundlagen zur Repräsentation und Verarbeitung des Kontexts

4.1. Kontext und Context-Awareness: Begriffsklärung

4.1.1 Diskussion des Kontextbegriffs in der Literatur

So intuitiv verständlich die Bedeutung des Begriffs „Kontext“ scheinen mag, bedarf es trotzdem, und vielleicht gerade deswegen, ihrer einleitenden Klärung. Bei der Fülle der auf Kontext eingehender Schlagworte, die in verschiedenen IT-Zweigen kursieren – z.B. „kontextuelle Suche“, „kontextsensitive Assistierung“ usw., wird hier der Fokus auf die Auffassung dieses Begriffs im Rahmen des Mobile Computing[28] gelegt. Auch in dieser Domäne herrscht keine Eindeutigkeit darüber, wie Kontext zu definieren sei. Einige Autoren schlagen dabei eine das Phänomen umschreibende Definition vor. Eine sehr gelungene aus dieser Reihe wird in [9] gegeben: “*any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves*”.

Doch weil Definitionen dieser Art wegen einer vielseitigen Natur des Phänomens zu allgemein geraten, versuchen andere Autoren verschiedene Ausprägungen der Kontexte zu klassifizieren. So wird beispielsweise in [6] und [7] zwischen *externem* und *internem* Kontext unterschieden. Das Externe bezeichnet diejenige Kontextinformationen, die von Hardwaresensoren gemessen werden, z.B. Zeit, Standort, Temperatur, Lichtverhältnisse, Geräusche, Bewegung usw. Hinzu zählen auch Kontextarten, die aus rohen Sensordaten – z.B. GPS-Koordinaten – Informationen wie Adresse oder Wetterverhältnisse liefern. Das Interne bezeichnet diejenige Kontextinformationen, die aus direkten Angaben des Benutzers (dem sog. Benutzerprofil) bzw. aus seinem Verhalten und seinen Interaktionen erschlossen werden: Ziele, Wünsche, Aufgaben und Intentionen des Benutzers. Die beiden Definitionsarten sind nicht streng formal gehalten und es ist auch sehr fraglich, ob es möglich ist eine sinnvolle formale Definition des Begriffs „Kontext“ anzugeben.

[28] Mit Mobile Computing werden hier mobile dialogführende bzw. dienstleistende Systeme bezeichnet.

4.1.2 Auslegung des Kontextbegriffs in dieser Arbeit

In dieser Arbeit wird eine Klassifizierung verschiedener Arten des Kontexts vorgeschlagen, die nicht beansprucht allumfassend zu sein, sondern vielmehr eine notwendige Ordnung in die Auffassung dieses Begriffs einzubringen bezweckt. Die Kontexte (oder genauer gesagt konkrete Instanzen von kontextuellen Informationen in einer bestimmten Situation), die für typische Anwendungsszenarien von SmartWeb relevant sind, werden in folgende drei Kategorien unterteilt, und zwar gemäß der Art und Weise, wie sie im hier vorgestellten Kontextmodell behandelt werden:

- Kontextuelle Informationen, die aus Sensordaten ermittelt werden, sowie das Benutzerprofil, der gleichermaßen repräsentiert wird wie die übrigen Kontextarten. Den Vertretern dieser Kategorie ist eigen, dass sie abgefragt bzw. ermittelt werden können (müssen jedoch nicht alle jederzeit verfügbar sein),
- Kontextuelle Informationen, die im typischen Anwendungsszenario in einer Benutzeraussage implizit vorhanden bzw. vermutlich gemeint sind, aber nicht explizit ausgesprochen, wie z.B. der Zusatz „zu Fuß“ in der Aussage „ich möchte zum Schloss“. Diese Art von kontextuellen Informationen wird mit Hilfe einer pragmatischen Ontologie und kontextuellen Informationen der ersten Kategorie inferiert und in die semantische Repräsentation der Benutzeraussage eingefügt,
- Kontext im Sinne des Gegenstandes einer Aussage. Im obigen Beispiel könnte man es als eine Intention der Fortbewegung zum Ziel bezeichnen. Dies, also in dem Fall der Begriff „Fortbewegung“, wird als ein Sachverhalt über eine Menge von ihn ausmachenden Komponenten modelliert, und zwar in der besagten pragmatischen Ontologie.

4.1.3 Pragmatik und Kontext

Die obige Kategorisierung erfasst und zeigt die enorme Vielseitigkeit des Phänomens Kontext: von offensichtlichen Hintergrundinformationen bis hin zu der Pragmatik der Benutzeraussage. Das Letztere, also die En- und Dekodierung der Bedeutung einer Aussage (wie es in [2] definiert wird), ist samt des pragmatischen Weltwissens ein wesentlicher Bestandteil eines intelligenten dialogführenden Systems, denn es der Domänendetektion, d.h. der Erkennung, ob das momentane Thema des Dialogs beispielsweise Navigation oder Sport ist, dient. Auch erst durch das pragmatische

Weltwissen wird es möglich über Intentionen des Benutzers zu spekulieren, also ob er beispielsweise das Ziel zu Fuß oder mit einem Taxi erreichen möchte. Eine Alternative zur Hinzuziehung der Pragmatik wäre die Beschränkung auf nur eine Domäne oder die Aufforderung des Benutzers fehlende Informationen anzugeben bzw. aus mehreren Optionen eine zu wählen. Diese Alternative ist nicht im Sinne einer intelligenten Dialogführung. Somit hängen Pragmatik und Kontext untrennbar zusammen, und zwar in solcher Art des Zusammenhangs, dass einerseits Pragmatik sich aus dem Kontext ergibt und andererseits die kontextuellen Informationen sich erst dann ermitteln lassen, wenn die Pragmatik, also die Bedeutung einer Aussage erschlossen ist. Dieser zweiseitige Zusammenhang wird detailliert in [2] untersucht[29].

4.1.4 Context-Aware Computing

Ergänzend zum Begriff „Kontext" wird nun *Context-Aware Computing* geklärt. Es herrscht Einigkeit darüber, dass dies Anwendungen – die sogenannten *Context-Aware Systems* – bezeichnet, die von kontextuellen Informationen Gebrauch machen. Doch viele Definitionen unterscheiden sich darin, welche Klassifizierung sie bezüglich der Art und dem Zweck der Verwendung dieser Informationen vorschlagen. Beispiele dafür, auf die hier nicht näher eingegangen wird, findet man in [8] und [9]. Stattdessen sei hier eine sehr anregende Definition aufgeführt, die einen bedeutenden Aspekt betont: *aktives* vs. *passives* Context Awareness [10]:

- Active context awareness: an application automatically adapts to discovered context, by changing the application's behavior.
- Passive context awareness: an application presents the new or updated context to an interested user or makes the context persistent for the user to retrieve later.

Das aktive Context Awareness ist wesentlich interessanter und stellt deutlich größere Herausforderung für Mobile Computing Systeme wie SmartWeb dar.

4.2 Context-Aware Systems

Es existiert heute eine Menge von Context-Aware Systems. Eine sehr gute Übersicht von diesen findet man in [5]. Dort sind einige Dimensionen herausgearbeitet, in denen

[29] Jedoch abweichend von der in [2] propagierten begrifflichen Trennung zwischen Kontext und Pragmatik, wird es in dieser Arbeit auf sie verzichtet: viel mehr wird Pragmatik bzw. das pragmatische Wissen als eine spezielle Art des Kontexts betrachtet.

sich die Context-Aware Systems voneinander unterscheiden. Dennoch – so [5] – liegt ihnen konzeptionell dieselbe Architektur zu Grunde. Im Weiteren dieses Kapitels werden zuerst die Dimensionen der Context-Aware Systems und dann die grundlegende Architektur dargestellt.

4.2.1 Darbietungsablauf von Kontextinformationen

Es wird von einem Klient – einer Softwarekomponente, die das Interesse an den Kontextinformationen äußert – ausgegangen. Gleich, ob ein solcher Klient Teil eines Context-Aware Systems oder eine externe Anwendung ist, gibt es (resümierend die in [18] und [19] gegebene Unterscheidung) prinzipiell drei Möglichkeiten für den Ablauf dessen, wie die Kontextinformationen aufgesammelt und diesem Klient dargeboten werden:

1. Der Klient äußert sein Interesse an bestimmten Kontextinformationen bzw. ermittelt diese selbständig. Dies kann zweierlei geschehen: erstens, durch einen direkten Zugriff an die entsprechenden Sensoren; zweitens, durch den Aufruf gewisser vermittelnder Softwarekomponenten (Middleware), die aus rohen Sensordaten, die an sich eventuell kein Interesse darstellen, relevante Informationen gewinnen (z.B. die Adresse aus GPS Koordinaten). Das Letztere ist vorteilhaft, weil die Softwarekomponenten Interfaces definieren, die unabhängig vom Datenformat eines konkreten Sensors sind. Somit sind Sensoren austauschbar: z.B. kann ein Thermometer, das die Temperatur in Celsius angibt, durch eines ersetzt werden, das dies in Fahrenheit macht, ohne, dass sich dabei der Zugriff für den Klienten ändert, d.h. ohne der Modifizierung des Quellcodes.
2. Der Klient spricht die Kontextinformationen-liefernden Quellen nicht an, sondern äußert sein Interesse an bestimmten Ereignissen und wird benachrichtigt, sobald diese eintreten. Hierbei gestaltet sich die Erweiterbarkeit von Kontextquellen besonders einfach. Der Nachteil ist, dass der Klient Kontextinformationen in zwei Phasen der Kommunikation weniger effizient erlangt, und dass eine zentrale Komponente benötigt wird, um diese Kommunikation zu verwalten.
3. Es gibt einen zentralen Kontextinformationen-ermittelnden Kontextserver, den der Klient im Fernzugriffmodus anspricht. Dies ist aus zwei Gründen vorteilhaft: erstens, im Sinne der Wiederverwendung, wenn es mehrere Klienten gibt, die

dieselben Kontextquellen einbeziehen, und zweitens, weil diese Klienten von oft zeit- und speicheraufwändiger Ermittlung und Behaltung von Kontextinformationen verschont bleiben. Letzteres ist besonders relevant, wenn den Klienten zur Verfügung stehende Ressourcen stark eingeschränkt sind (wie z.B. bei den mobilen Geräten). Die Kehrseite dieser serverbasierten Architektur ist, dass Netzprotokolle entwickelt und implementiert werden sollen.

4.2.2 Sensorarten

Nach [20] können die Sensoren in drei Gruppen unterteilt werden:

1. *Technisch-physikalische Sensoren*: Dies sind Hardwarekomponenten, welche die Parameter der physikalischen Umgebung wahrnehmen bzw. bestimmen. Dazu zählen Sensoren, die beispielsweise Zeit, Ort, Temperatur, Lichtstärke ermitteln, aber auch Sensoren, die audio-visuelle Daten sowie die Berührung wahrnehmen. Die Sensoren dieser Gattung werden in den heutigen Context-Aware Systems am meisten eingesetzt.
2. *Virtuelle Sensoren*: Dies sind Sensoren, die Daten aus Softwareanwendungen beziehen. Zum Beispiel kann man aus einem elektronischen Terminkalender Informationen zum Aufenthaltsort seines Besitzers gewinnen. Auch das Auflauern der Benutzeraktivität in Umgang mit einer benutzten Softwareanwendung gehört zu dieser Gattung der Sensorik.
3. *Logische Sensoren*: Diese Sensoren kombinieren die Daten von anderen Sensoren sowie das Wissen aus anderen Quellen (z.B. Wissensbasen) in logische Ausdrücke. Zum Beispiel kann der Aufenthaltsort einer Person aus den Informationen ermittelt werden, aus denen hervorgeht, dass diese Person sich an einem bestimmten Rechner eines lokalen Netzwerks angemeldet hat und dieser Rechner sich in einem bestimmten Raum befindet.

4.2.3 Kontextmodell

Das Kontextmodell legt das Format fest, in dem die Kontextinformationen zur maschineller Verarbeitung innerhalb des jeweiligen Context-Aware System repräsentiert werden. In [21] sind die wichtigsten Datenstrukturen bzw. Modelle und Techniken genannt, die dafür geeignet sind:

- *Attribut-Wert Paare*: Dies ist die einfachste Alternative.

- *Markup scheme (Textauszeichnungsschemas)*: Eine Struktur von hierarchisch angeordneten Kennzeichnungsmarken (Tags), denen ein Attribut-Wert Paar zugeteilt ist. Auf diese Art wurden verschiedene Schemata zur Repräsentation des Benutzerprofils definiert. Diese Schemata sind in RDF(S) encodiert.
- *Graphische Modelle*: Auch diese (z.B. UML[30]) sind zur Repräsentation von Kontextinformationen geeignet.
- *Objektorientierung*: Zur Repräsentierung verschiedenartiger Kontextquellen wird eine Interfaceinfrastruktur definiert. Es lassen sich dabei Konzepte und Techniken der objektorientierten Programmierung wie die Hierarchie, Vererbung, Wiederverwendung verwenden.
- *Logikbasierte Modelle*: Es werden (im Sinne der mathematischen Logik) Theorien entwickelt, innerhalb deren Kontextquellen und Informationen, die sie ermitteln, repräsentiert sind. Aus solchen Theorien und den aktuell vorliegenden Kontextinformationen lässt sich neues Wissen erschließen. Das erste logikbasierte Modell wurde von McCarthy und Buvac vorgeschlagen [17].
- *Ontologische Modelle*: Diese können als spezifische Ausprägungen der logikbasierten Modelle gesehen werden, und zwar derartige, die eine spezielle Sicht auf die Welt vertreten. Nämlich, dass die Welt sich durch hierarchisch angeordnete Begriffe und Beziehungen zwischen ihnen beschreiben lässt.

Die obigen Alternativen wurden in [21] nach einer bestimmten Reihe von Kriterien untersucht. Dabei wurden die ontologischen Modelle als am besten zur Repräsentation von Kontextinformationen geeignete befunden.

4.2.4 Nutzbarmachung von Kontextinformationen

Die wohl wichtigste Frage in einer Diskussion über die Context-Aware Systems ist nach der Nutzbarkeit von Kontextinformationen. In [16] werden zwei Modalitäten genannt von diesen zu profitieren: die *Aggregation* und die *Interpretation* von Kontextinformationen.

Die Aggregation bezeichnet die Zusammensetzung von Kontextinformationen aus verschiedenen Quellen zu einer übergeordneten Kontextinformation. Ein Beispiel dazu

[30] UML – Unified Modelling Language – wird in dieser Arbeit nicht vorgestellt.

wird im Projekt Gaia demonstriert [22]. Dort werden Kontextinformationen als vierstellige Prädikate repräsentiert, die folgende Grundform aufweisen:

```
Context(<ContextType>, <Subject>, <Relater>, <Object>)
```

Dabei steht `<Subject>` für eine Person, einen Ort oder ein Ding, auf das sich der Kontext bezieht; `<Object>` ist ein Attribut oder eine Variable, mit der `Subject` assoziiert ist; und `<Relater>` ist eine Relation zwischen den beiden, z.B. ein Vergleichsoperator, ein Verb oder eine Präposition. Eine Aggregation wäre beispielsweise folgende Regel:

```
Context(Social Activity, Room 2401, Is, Presentation) ←
    Context(Number of people, Room 2401, >, 4) ∧
    Context(Application, Powerpoint, is, Running)
```

Die Interpretation bezeichnet das Schlussfolgern bestimmter Aktionen aus aktuellen Kontextinformationen und gegebenenfalls unter der Einbeziehung einer (statischen) Wissensbasis. Als Beispiel ist im Projekt SOCAM [23] folgende Regel angegeben:

```
alert me(John) ←
    socam:locatedIn(John,socam:MyCar) ∧
    socam:status(SeatBelt,LOOSE)
```

Im Großen und Ganzen lassen sich die in Context-Aware Systems mit Reasoning bezeichnete Dienste auf die Aggregation und Interpretation zurückführen. Dabei fällt jedoch bei vielen Context-Aware Systems, die diese Dienste anbieten, negativ auf, dass sie (die Dienste) nur an Beispielen demonstriert und nicht wohldefiniert sind.

4.2.5 Historie von Kontextinformationen

Oft ist es erwünscht die im Laufe der Zeit aufgesammelten Kontextinformationen zu bewahren, um beispielsweise mittels Lernalgorithmen Trends aufspüren oder Prognosen erstellen zu können. Im Falle eines Kontextservers als zentrale Anlaufstelle für die Klienten (siehe 4.2.1) ist die Aufbewahrung von Kontextinformationen sogar notwendig. Für diese Zwecke werden meist in Vielzahl existierenden Datenbanken eingesetzt.

Bei einigen Context-Aware Systems, insbesondere bei den in ein mobiles Gerät integrierten, ist der zur Verfügung stehende Speicher sehr begrenzt. In diesem Fall wird es von einer Aufbewahrung von Kontextinformationen abgesehen.

4.2.6 Grundlegende Architektur

Die zuvor betrachteten Aspekte finden ihren jeweiligen Platz in der in [24] vorgeschlagenen grundlegenden 5-Schichten-Architektur von Context-Aware Systems. Diese ist in der Abbildung 4.1[31] vorgestellt.

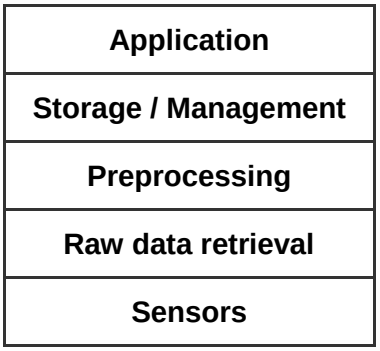


Abbildung 4.1: Die den Context-Aware Systems grundlegende 5-Schichten-Architektur.

Im Folgenden werden diese Schichten kurz kommentiert:

- *Sensors*: Diese Schicht referenziert die unterste Ebene von Sensoren bzw. die rohe Daten, die diese Sensoren liefern (z.B. die GPS Koordinaten). Es ist jedoch zu beachten, dass diese Schicht Sensoren aller drei (in 4.2.2 genannten) Arten repräsentiert.
- *Raw Data Retrieval*: Diese Schicht realisiert die Abstrahierung von der Verkörperung konkreter Sensoren und vom Datenformat, in dem sie ihre Messungen[32] liefern, indem ihre jeweilige Treiber bzw. APIs benutzt werden, um aus den nachrangigen rohen Sensordaten heraus die abstrakteren Methoden zur Verfügung zu stellen. Zum Beispiel eine Methode `getTemperature()`, die dem Benutzer, unabhängig vom Datenformat des Sensors und der von ihm

[31] Es handelt sich dabei um eine Abzeichnung der Originalabbildung aus [5].
[32] Das Wort „messen“ ist im Zusammenhang mit den Sensoren im Algemeinen etwas ungünstig, insbesondere hinsichtlich der virtuellen und logischen Sensoren, die keine Messungen durchführen. Dieses Wort wird jedoch weiterhin verwendet, um den Unterschied zwischen rohen Sensordaten und den Kontextinformationen zu unterstreichen: in Bezug auf die Letzteren wird das „ermitteln“ benutzt.

verwendeten Temperaturskala, die in `{cold,cool,normal,warm,hot}` diskretisierte Temperatur liefert.

- *Preprocessing*: Diese Schicht erledigt viele Vorverarbeitungsaufgaben, um die nächsthöhere Abstraktionsniveau zu erreichen. Zum Beispiel sind die GPS Koordinaten einer Person an sich nicht relevant; die wichtigere Information ist der Raum, in dem sich diese Person befindet. Dieser ist möglicherweise erst durch die Aggregation von Daten mehrerer Sensoren determinierbar: es könnte beispielsweise notwendig sein den Terminplaner betreffender Person zu analysieren; dabei müssen die eventuell aufkommende Unstimmigkeiten zwischen verschiedenen Sensoren behandelt werden (zum Beispiel meldet eine Videoüberwachungskamera diese Person in einem anderen Raum detektiert zu haben, als es aus GPS-Koordinaten hervorgeht). All das gehört zu den Aufgaben dieser Schicht.
- *Storage/Management*: In dieser Schicht wird der Zugriff eines Klienten auf die Kontextinformationen (siehe 4.2.1) sowie die Aufbewahrung dieser (siehe 4.2.5) geregelt.
- *Application*: Diese Schicht realisiert den Klienten, der die Kontextinformationen (im Sinne von 4.2.1) „konsumiert“ und interpretiert (siehe 4.2.4). Eventuell werden auch Aufgaben von Preprocessing in diese Schicht deligiert. Doch davon ist es insbesondere dann abzuraten, wenn der Klient in ein mobiles Gerät integriert ist, weil ein solches oft über eine vergleichsweise niedrige Rechenleistung und geringe Speicherkapazität verfügt.

Alle Context-Awere Systems implementieren jede dieser Schichten, bis auf einige Ausnahmen, in denen von Preprocessing abgesehen wird [5].

4.3 DnS-Framework

Um wie im obigen Beispiel aus der Abbildung 3.1 „Strasse“ durch „Landstrasse“ sinngemäß zu ersetzen, bedarf es, wie bereits erwähnt, pragmatisches Wissen, aus dem hervorgeht, dass es bei schönem Wetter empfehlenswerter ist. Es hat sich bewährt dieses pragmatische Wissen in Form einer Ontologie zu repräsentieren [1]. Es wurde beschlossen, in SmartWeb für diese Zwecke *Descriptions & Situations Framework* (DnS-Framework) einzusetzen, da dies gegenwärtig einziges zum ontologischen Repräsentieren des vielfältigen Phänomens „Kontext“ geeignetes Framework ist [2]. Im

Weiteren wird das DnS-Framework in dem für diese Arbeit benötigten Maß der Ausführlichkeit vorgestellt; die tiefer gehende Information findet man in [3] und [4].

4.3.1 DnS-Design-Pattern

4.3.1.1 Idee des DnS-Design-Pattern im Überblick

DnS-Framework präsentiert die *Ontology of Descriptions and Situations* (*DnS-Ontologie*), die den sogenannten *DnS-Design-Pattern* (siehe Abbildung 4.2[33]) axiomatisch umschreibt. Im DnS-Framework werden zwei grundlegende Begriffe eingeführt: *Deskription* und *Situation*. Die Idee ist dabei folgende: eine Deskription beschreibt ein nicht physikalisches Objekt[34], wie z.B. ein Wunsch, Plan, Gesetz oder – aus dem für SmartWeb relevantem Bereich – Fortbewegung; eine Situation entspricht einer konkreten Gegebenheit in der Welt, stellt also Teilstück des faktischen Wissens in Form einer Menge von Individuen und relationaler Beziehungen unter ihnen dar. So kann beispielsweise eine Situation eine A-Box sein, welche die Aussage „In Darmstadt scheint gerade die Sonne“ repräsentiert. Es soll möglich sein, über eine Situation urteilen zu können, ob sie die Verkörperung einer bestimmten Deskription ist, z.B. ob eine Situation `S` eine Fortbewegungssituation ist, wofür gewissermaßen das Matching zwischen `S` und der Fortbewegungssituation untersucht wird. Wobei tatsächlich der Prozess des Matching der ontologischen Sichtweise bzw. innerhalb der dahinter stehender Beschreibungslogik fremd ist; vielmehr wird geprüft, ob `S` ein Modell für die Fortbewegungsdeskription ist.

[33] Es handelt sich dabei um eine originale Abbildung aus [3]. Aus dieser werden nur diejenige ontologische Entitäten erläutert, die für diese Arbeit von Bedeutung sind.

[34] Genauer: eine Deskription ist eine Instanz der Klasse `Description`, die eine Unterklasse von `SocialObject` (vergleiche die Abbildung 2.11 und siehe [3] für weitere Details).

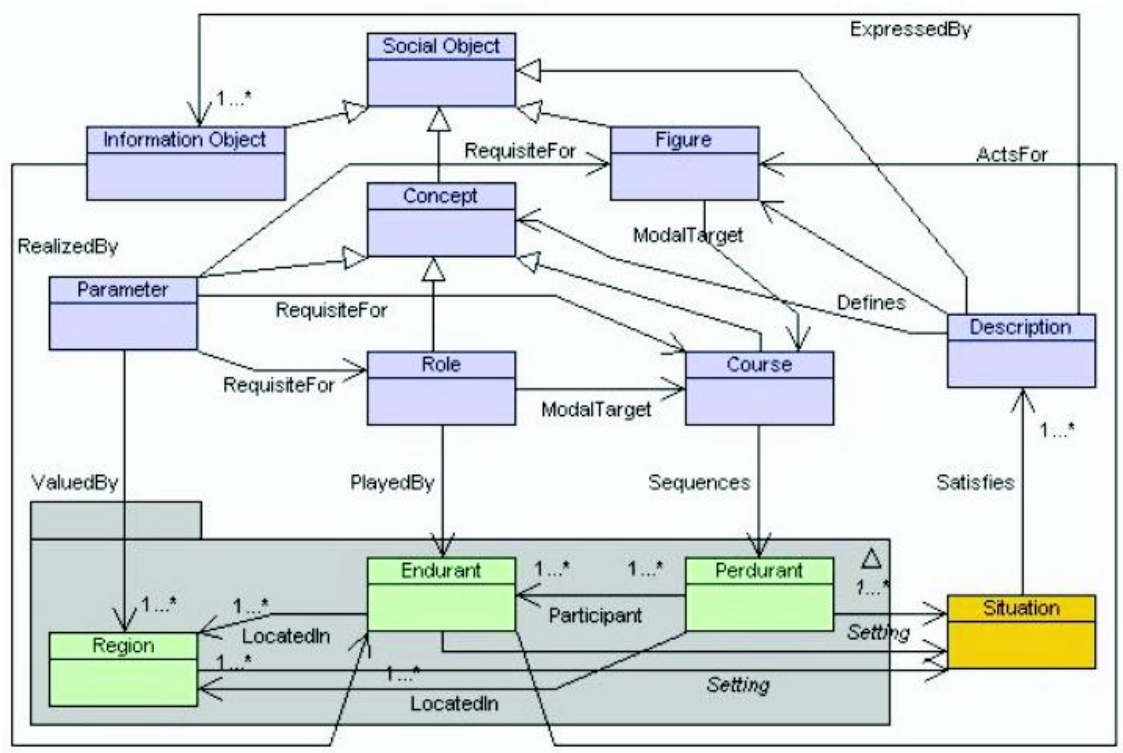


Abbildung 4.2: DnS-Design-Pattern.

4.3.1.2 Deskriptive Komponenten für die Modellierung von Deskriptionen und Situationen gemäß dem DnS-Design-Pattern

Die Vorlage dafür, wie Deskriptionen und Situationen zu modellieren seien, gibt das DnS-Design-Pattern vor. Es sieht vor, dass Deskriptionen mittels Rollen, Parametern und Abläufen modelliert werden und legt fest, welche Entitäten die jeweiligen Rollen spielen dürfen, welche Wertebereiche die Parameter haben sollen und welche konkrete Aktivitäten die Abläufe ausmachen. Dies wird in der DnS-Ontologie durch folgende deskriptive Komponenten untermauert:

- Erstens durch die Klasse `Description`, die über die Relation `Defines` mit den Klassen `Parameter`, `Role` und `Course` verbunden ist, wodurch eine bestimmte Deskription als Gesamtheit der für sie relevanten Parametern, Rollen und Abläufen beschrieben wird. Zu beachten ist, dass diese deskriptive Komponenten optional und nicht obligatorisch aufzufassen sind, d.h. es ist bei einer Deskription durchaus erlaubt, beispielsweise auf Abläufe zu verzichten, wenn sie nicht erforderlich sind. Dies wird auch bei den in dieser Arbeit verwendeten Beispielen der Fall sein, weil Abläufe insbesondere bei solchen Deskriptionen relevant sind, die mit dem Planen verbundene Begriffe

modellieren [3]. Derartige Begriffe werden jedoch in dieser Arbeit nicht gebraucht.

- Zweitens durch die Relationen `RequisiteFor` und `AttitudeTowards`, die Parameter und Rollen bzw. Parameter und Abläufe bzw. Rollen und Abläufe zueinander in Verbindung setzen, wodurch eine Deskription nicht nur als eine zusammenhangslose Auflistung von Parametern, Rollen und Abläufen modelliert werden kann, sondern eben als eine Gesamtheit dieser Entitäten, bei der bestimmte Rollen und Abläufe Einhaltung gewisser Parameter voraussetzen und einzelne Rollen auf bestimmte Weise in Abläufen teilnehmen. Diese deskriptive Komponenten sind ebenfalls optional; sie verlieren in jenen Fällen die Bedeutung, in denen es bei jeder möglichen Situation die Beziehungen unter dort vorkommenden Parametern, Rollen und Abläufen a priori bekannt sind, insbesondere also wenn alle Situationen so beschaffen sind, dass in ihnen jeweils höchstens ein Parameter, eine Rolle und ein Ablauf vorkommt. Aus dieser Überlegung heraus verzichteten die Entwickler von SmartWeb auf diese deskriptive Komponenten.
- Drittens durch die Relationen `ValuedBy`, `PlayedBy` und `Sequences`, welche die Parameter, Rollen und Abläufe mit jenen Klassen der Basisontologie verbinden, die eine `Region` bzw. ein `Endurant` bzw. ein `Perdurant` sind, wodurch ausgesagt wird, dass den Parametern in der Basisontologie bestimmte Wertebereiche zugeordnet sind, die Rollen von bestimmen basisontologischen `Enduranten` gespielt werden dürfen und die Abläufe sich in bestimmten Aktivitäten äußern, die in der Basisontologie `Perdurante` sind.

4.3.1.3 Modellierung der Deskriptionen

Um eventuelle Unbestimmtheiten bezüglich dessen zu beseitigen, wie nun genauestens eine neue Deskription gebildet wird, d.h. welche Entitäten dafür erzeugt werden müssen, zeigt die Abbildung 4.3, eine in Ahnlehnung an das DnS-Design-Pattern konstruierte Deskription `New Description`. Sie entsteht im Algemeinen dadurch, dass von Klassen `Description`, `Parameter`, `Role`, `Course`, `Region`, `Endurant` und `Perdurant` jeweils eine Unterklasse abgeleitet und eine Instanz von ihr erzeugt werden. Die Instanzen werden gemäß DnS-Design-Pattern in eine Beziehung zueinander gesetzt. So bedeutet beispielsweise `ValuedBy(New Parameter, New`

Region), dass die Klasse New Parameter, als deskriptive Komponente von New Description, die Klasse New Region zum Wertebereich erklärt (Entsprechendes gilt für Rollen und Abläufe). Die Gesamtheit von Instanzen bildet die neu modellierte Deskription. Dabei ist Folgendes zu beachten:

- Erstens, müssen die Unterklassen nicht zwangsläufig abgeleitet werden: es ist durchaus möglich die Oberklassen zu nutzen, falls die sonstige designirische Prinzipien es zulassen. Für die Aufgabenstellung des SmartWeb ist es erforderlich, Deskriptionen hierarchisch zu gestalten, somit ist die Ableitung von Unterklassen die naheliegende Lösung.
- Zweitens, können statt den verwendeten Relationen die Unterrelationen abgeleitet und benutzt werden. Damit kann beispielsweise die Art der Rolle spezifiziert werden, die in einer Deskription ein Endurant spielt.
- Drittens, zeigt die Abbildung 4.3 nicht die allgemeine Form einer Deskription. Einige Details, zum Beispiel wie man mehrere Parameter definiert oder mehrere Klassen als Wertbereich eines Parameters angibt – werden im Kapitel 6 herausgearbeitet.
- Viertens, möchte man eine weitere Deskription konstruieren, die einige deskriptive Komponenten beinhaltet, die in New Description vorkommen – es handele sich um z.B. die Klasse New Parameter – dann gilt Folgendes zu beachten: Jede deskriptive Komponente „verhält sich“ in zwei unterschiedlichen Deskriptionen identisch. Möchte man bei einer weiteren Deskription, dass beispielsweise New Parameter anderes Wertebereich hat, muss man eine andere Instanz von New Parameter erzeugen und sie in entsprechende Beziehung mit erwünschtem Wertebereich setzen (Entsprechendes gilt für Rollen und Abläufe).

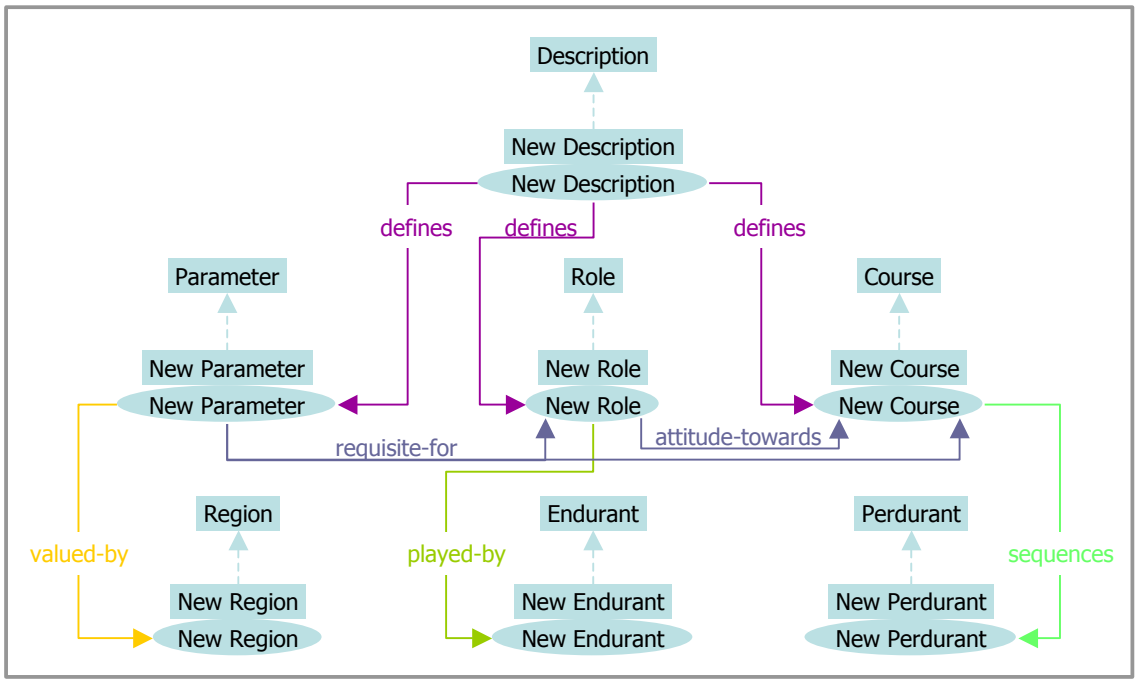


Abbildung 4.3: Modellierung einer Deskription `New Description` gemäß dem DnS-Design-Pattern. (∗)[35]

4.3.1.4 Situationen gemäß dem DnS-Design-Pattern

Eine Situation ist im Grunde genommen eine A-Box, in der eine Instanz von `Situation` vorkommt. Des weiteren beinhaltet eine Situation gewöhnlich Instanzen von `Region`, `Endurant` und `Perdurant`, die über die Relation `Setting` als zu Situation angehörend gekennzeichnet und gegebenenfalls über die grundontologische Relationen `LocatedIn` und `Participant` zueinander in Bezug gesetzt sind. (Die Letzteren entsprechen in der DnS-Ontologie den Relationen `RequisiteFor` bzw. `AttitudeTowards`.) Eine Situation genügt einer Deskription (stellt also ihre Verkörperung dar), falls ihre (der Situation) Entitäten gewisse Bedingungen erfüllen, deren Zweck es zu prüfen ist, ob eine gewisse Übereinstimmung zwischen dieser Situation und der Deskription vorliegt. Im nächsten Abschnitt werden diese Bedingungen genauer betrachtet.

[35] Hier und im Weitern werden mit (∗) diejenige Abbildungen markiert, welche die im Anhang C vereinbarte Bezeichnungen verwenden.

4.3.1.5 Vereinfachtes Beispiel für eine Deskription und eine Situation

Die Abbildung 4.4 zeigt ein Beispiel für eine Deskription, in der Begriff „Fortbewegung“ modelliert wird. Diesem Beispiel geht hervor, dass die Fortbewegung durch den Parameter `Environment` und die Rolle `Path` definiert ist, wobei als Wertebereich für `Environment` `WeatherRegion` (ein Oberbegriff für bestimmte Wetterverhältnisse wie „sonnig“, „regnerisch“ usw.) genannt ist und die Rolle des `Path` von `Road` (eine Klasse, die für allgemeinen Begriff „Strasse“ steht) gespielt werden darf. Somit wird es in diesem vereinfachten Fall versucht, eine Fortbewegung nur durch vorherrschende Wetterverhältnisse und eine teilnehmende Strasse zu erklären (daher auch der Name dieser Deskription). Des weiteren zeigt die Abbildung 4.4 eine Situation `Some Situation`, die aus Instanzen `Sunny` und `A 60` von Klassen `WeatherRegion` bzw. `Road` besteht und der Fortbewegungsdeskription aus der Abbildung offensichtlich genügt[36].

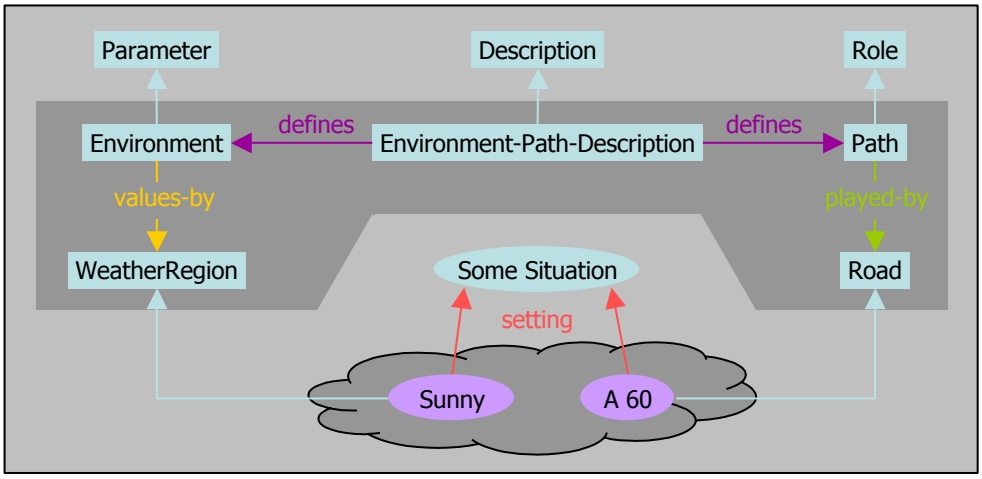


Abbildung 4.4: Vereinfachtes Beispiel für eine Deskription, die den Begriff Fortbewegung modelliert, und eine Situation, die sie erfüllt. (∗)

4.3.2 Erfüllungsrelation `Satisfies` und ihre Unterrelationen

Im Sinne des DnS-Framework genügt eine Situation `S` einer Deskription `d` genau dann, wenn das Hinzufügen von Behauptung `Satisfies(S,D)` oder `P-Sat(S,D)` oder `R-Sat(S,D)` oder `C-Sat(S,D)` in die Basis- und DnS-Ontologie beinhaltende

[36] Die Offensichtlichkeit wird sich spätestens nach dem Kapitel 6.3 ergeben. In diesem werden die bisher informell eingeführten Begriffe, formal dargelegt.

Wissensbasis zu keiner Inkonsistenz führt. Die letzten drei Relationen sind Spezialisierungen von `Satisfies`, stellen also höhere Anforderungen auf. Außerdem verkörpern sie grundlegend unterschiedliche Ideen (z.B. bezüglich einzelner zu modellierender Wissensbereiche, in denen die jeweilige Relation zweckmäßig eingesetzt werden kann), die hier detailliert nicht geschildert werden.

Die Relation `Satisfies(S,D)` besteht genau dann, wenn zumindest eine Komponente in `D` (d.h. ein Parameter, Ablauf oder eine Rolle) ein Individuum `I` selektiert[37], das ein Setting für `S` ist (für welches also `Setting(I,S)` gilt). Dies kann man auf unterschiedliche Arten spezialisieren; folgend werden vier Varianten beschrieben.

- Die erste Variante besteht in der Forderung, dass jedes Individuum, das ein Setting für `S` ist, von einer Komponente von `D` selektiert wird. Doch diese Forderung wäre zu unrealistisch, denn wie es sich gezeigt hat, werden in reellen Anwendungen, die von DnS-Framework Gebrauch machen, Situationen meist aus bereits vorhandenen A-Boxen konstruiert, die eine vorgegebene und nicht auf DnS-Framework bezogene Struktur aufweisen [3], wie es auch in SmartWeb der Fall ist.
- Die zweite Variante besteht in der Forderung, dass jede Komponente in `D` ein Individuum selektiert, das ein Setting für `S` ist. Dabei wäre es durchaus erlaubt, dass einige Settings für `S` nicht selektiert sind. Für solche könnte man zulassen, dass sie Instanzen derjenigen Klassen sind, die in der Deskriptionen modellierender Ontologie nicht definiert sind[38]. In diesem Falle stöße man innerhalb dieser bzw. der DnS-Ontologie auf das Unentscheidbarkeitsproblem: man würde weder nachweisen können, dass die gegebene Situation ein Modell für die aus beiden Ontologien bestehende Theorie ist, noch dass es kein Modell für sie ist. Ein Ausweg aus jenem Problem bestünde dann darin, die

[37] Eine Deskriptionskomponente, z.B. ein Parameter, selektiert ein Individuum, wenn es zum Wertebereich dieses Parameters gehört. Entsprechendes gilt für die Rollen und Abläufe.

[38] Hierbei ist die Annahme getroffen worden, dass die Deskriptionen in einer separaten Ontologie modelliert sind, welche die DnS-Ontologie importiert, denn sonst, falls man die Deskriptionen in der Basisontologie modellieren würde, wäre das Inferieren in diesem, meist recht umfangreichem Gebilde mit hoher Wahrscheinlichkeit ungenügend performant. Äquivalenterweise können die Deskriptionen selbstverständlich auch direkt in der DnS-Ontologie modelliert werden. Jedoch ist allenfalls zu beachten, dass als Wertebereiche von den Relationen `ValuedBy`, `PlayedBy` und `Sequences` bzw. ihrer Unterrelationen üblicherweise Klassen der Basisontologie angegeben werden (z.B. ist der Wertebereich von Parameter `Environment` aus der Abbildung 4.4 die basisontologische Klasse `WeatherRegion`). Dies bedeutet, damit Instanzen dieser Klassen innerhalb der Deskriptionen modellierender Ontologie als `Enduranten` bzw. `Perduranten` erkannt werden, muss diese Ontologie entsprechende Oberklassenhierarchien mit einschließen.

Basisontologie in die Theorie aufzunehmen, doch dies würde mit hoher Wahrscheinlichkeit zu unperformanter Inferenz führen, denn die Basisontologie oft ein sehr umfangreiches Gebilde ist.

- Die dritte Variante ist eine Kombination der beiden zuvor genannten. Folglich hätte sie den Nachteil, den die erste Variante aufweist.
- Und schließlich bildet die vierte Variante eine dahingehende Modifikation der zweiten, dass man einige Komponenten von `D` als optional ansieht und auf diese Weise damit festlegt, dass `S` bereits dann `D` genügt, wenn alle obligatorische Komponenten von `D` ein Setting von `S` selektieren. Diese Variante weist natürlich denselben Nachteil auf wie die zweite.

Die Relationen `P-Sat`, `R-Sat` und `C-Sat` unterscheiden sich unter anderem darin, welche Spezialisierungsvariante von `Satisfies` ihnen zugrunde liegt.

5 Entwurf und Implementierung einer Context-Aware Infrastruktur für SmartWeb

Die in 4.2 erwähnten Context-Aware Systems sind nicht zum Einsatz in SmartWeb geeignet, weil selbst wenn sie ein ontologisches Kontextmodell anwenden, handelt es sich dabei nicht um die in SmartWeb verwendete Basisontologie. Es ist also notwendig, dass die Kontextinformationen mit deskriptiven Mitteln von SWIntO beschrieben werden können. Außerdem entstammen diese Context-Aware Systems Forschungsprojekten, die ihre Ergebnisse zur Nutzung in anderen Projekten nicht bereitstellen. Somit wurde eine eigene Infrastruktur entwickelt und in Java implementiert. Sie ist in der Abbildung 5.1 dargestellt, bestehend aus einigen Komponenten und Funktionalitäten, die eine Komponente in einer anderen aufruft. Im Folgenden dieses Kapitels werden diese Komponenten, außer der Anwendung, erläutert. Mit der Letzteren beschäftigen sich die Kapitel 6, 7 und 8.

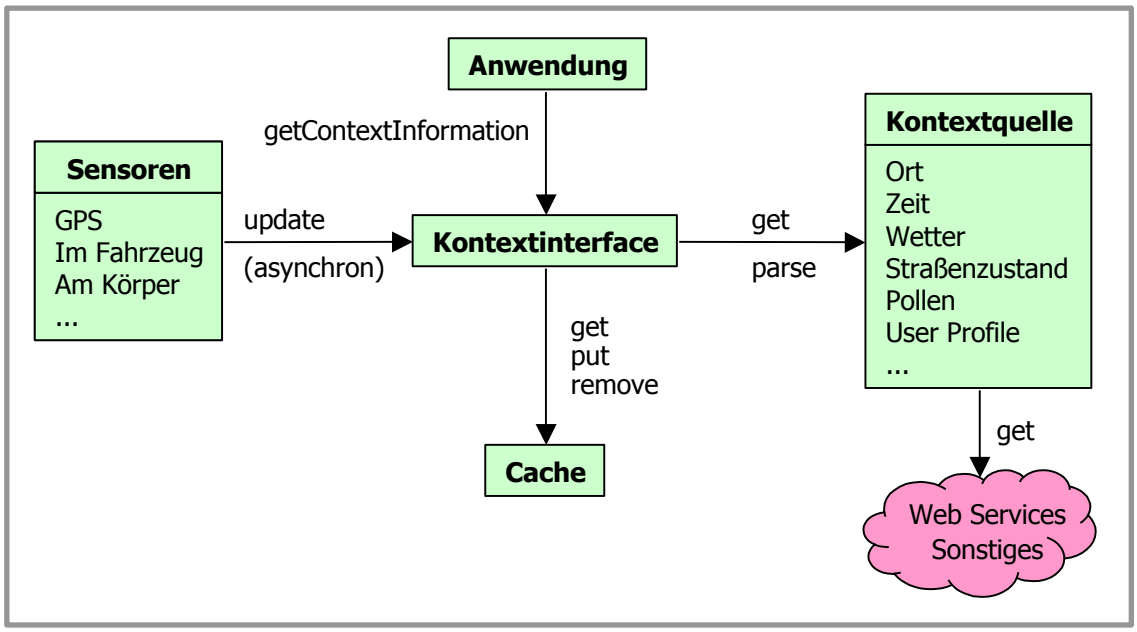


Abbildung 5.1: Die Context-Aware Infrastruktur für SmartWeb. (Nicht alle aufgelistete Sensoren bzw. Kontextquellen sind tatsächlich implementiert.)

5.1 Komponenten

5.1.1 Sensoren

Die Sensoren sind unmittelbar in der Infrastruktur, oder genauer, in der sie implementierenden Software nicht präsent. Sie können nicht abgefragt werden; stattdessen schicken sie asynchron Benachrichtigungen, sobald neue Daten gemessen wurden. Zu welchen Zeitpunkten diese Messungen stattfinden und an welcher Stelle die Sensoren softwaremäßig implementiert sind (d.h. welche Komponente in SmartWeb die Benachrichtigungen verschickt) – dies wird gemäß den Vorgaben des SmartWeb außerhalb der vorgeschlagenen Infrastruktur geregelt.

Man kann sich diverse Sensoren vorstellen, die in SmartWeb einsetzbar wären: GPS Sensor; Sensoren im Fahrzeug, die beispielsweise dessen Geschwindigkeit messen; Sensoren am Körper einer Person, die zum Beispiel den Pulsschlag messen, um daraus Anregungs- oder (körperliches) Anstrengungsgrad zu ermitteln. Momentan werden (innerhalb der Infrastruktur) nur der GPS Sensor und der das Benutzerprofil bestimmender Sensor verwendet. Der Erste sendet die geographischen Koordinaten (Längengrad, Breitengrad, Höhenlage) sowie die aktuelle Geschwindigkeit des Sensorträgers in einem festgelegten Format in XML (siehe die Abbildung 5.2).

```
<context>
  <gps xsi:noNamespaceSchemaLocation="../gps.xsd" status="A">
     <utc>121322</utc>
     <utdate>150905</utdate>
     <long hem="E">1337.719</long>          // Längengrad
     <lat hem="N">5251.596</lat>            // Breitengrad
     <alt unit="M">200</alt>                // Höhenlage
     <speed>122.3</speed>
  </gps>
</context>
```

Abbildung 5.2: Beispiel der vom GPS Sensor gesendeten Daten.

Der Sensor zur Bestimmung des Benutzerprofils (derjenigen Person, die das SmartWeb Dialogsystem gerade benutzt) liefert seine Daten ebenfalls in einem festgelegten Format in XML. Diese Daten beinhalten obligatorische Angaben, zum Beispiel Name, Geschlecht, Alter, eventuelle Behinderungen des Benutzers, und gegebenenfalls einige

optionale Angaben, zum Beispiel die Vorlieben des Benutzers bezüglich der Filmgenres. Jeder Benutzer verfügt über ein ihn eindeutig identifizierendes Profile.

5.1.2 Kontextquellen

Eine Kontextquelle wird durch (im Sinne der objektorientierten Programmierung) Oberklasse `ContextSource` repräsentiert und hat die Aufgabe Kontextinformationen zu ermitteln. Vereinfacht gesagt, entspricht jede Kontextquelle einem Typ des Kontextes; präziser ausgedrückt entspricht sie genau einer Art wie eine Kontextinformation ermittelt wird. Letzteres impliziert eine Möglichkeit, dass beispielsweise Kontextinformationen des Typs „Wetter“ auf diverse Arten ermittelt werden können, zum Beispiel über mehrere unterschiedliche Web Services. In diesem Fall soll es entsprechend viele von `ContextSource` abgeleitete Klassen geben. Im Folgenden werden die wichtigsten Methoden von `ContextSource` vorgestellt.

`HashMap<String,Boolean> need ()`

Gibt alle Sensorennamen zurück (bzw. eine Hashtabelle mit Einträgen `(Sensorname, true)`), deren Daten notwendig sind, damit diese Kontextquelle eine Kontextinformation ermitteln kann. Zum Beispiel ist das Wetter bekanntlich ortsabhängig, d.h. in diesem Fall sind die Daten vom GPS Sensor notwendig.

`Object[] parse(String xml)`

Falls `xml` eine Messwerte-beinhaltende Nachricht von einem für diese Kontextquelle notwendigen Sensor ist, wird diese Nachricht geparst; zurückgegeben werden die für diese Kontextquelle relevanten Datenfelder aus `xml` (z.B. sind es bei der Wetterkontextquelle der Längen- und Breitengrad sowie die Höhenlage). Falls `xml` für diese Kontextquelle irrelevant ist, wird `null` zurückgegeben.

`ContextSourceValueInterface update(`
`ContextInterface ci, Object[] updates)`

Ermittelt und gibt eine Kontextinformation mit den in `updates` übergebenen Parametern zurück (diese sind das Ergebnis der Methode `parse(String xml)`).

`public ContextSourceValueInterface get(ContextInterface ci)`

Ermittelt eine Kontextinformation und gibt sie zurück. Wenn dafür Sensordaten notwendig sind, werden jeweils die durch Methode `parse(String xml)` zuletzt bereitgestellten verwendet.

Wie es den obigen Methoden zu entnehmen ist, werden zwei Ideen verfolgt: erstens, die Sensoren und die Kontextinformationen begrifflich und bei der Infrastrukturmodellierung voneinander zu trennen, und zweitens die Letzteren sozusagen abhängig von den Ersten zu machen. In der Literatur (siehe [5]) werden diese Ideen nicht auf die gleiche Weise konsequent verfolgt. Der Schritt von den rohen Sensordaten zu den Kontextinformationen lässt sich in der Schichtenarchitektur aus 4.2.6 in die Schichten „Raw Data Retrieval“ und „Preprocessing“ einordnen.

Weil in SmartWeb nicht benötigt, wurde es von der Option abgesehen, Abhängigkeit einer Kontextquelle von einigen anderen zu implementieren. Zum Beispiel wäre es eine überaus anregende Herausforderung den momentanen emotionalen Zustand des Benutzers zu erraten. Dafür könnte man wahrscheinlich eine Menge von Kontextinformationen in Betracht ziehen: momentanen Gesichtsausdruck, Stimmlage und Intonation des Benutzers sowie eventuell einige Angaben aus seinem Profile, zuletzt stattgefundenen Ereignisse, und möglicherweise auch aktuellen Wetterverhältnisse. Aber auch ohne dieses komplexen Szenarios ist es sicherlich überlegenswert, die Abhängigkeit der Kontextquellen voneinander einzuführen.

SmartWeb verfügt über eine Reihe von Kontextquellen; jede Kontextquelle hat einen unikaten, sie identifizierenden Namen. Die im Rahmen dieser Arbeit implementierten Kontextquellen sind nachfolgend aufgelistet:

- `TimeSource`: Diese simple Kontextquelle ermittelt aktuelle Uhrzeit (bzw. Datum). Der Name dieser Kontextquelle ist: „`time`“[39].
- `LocationSource`: Diese Kontextquelle ermittelt die mit den aktuellen geographischen Koordinaten am besten übereinstimmende Adresse. Der Name dieser Kontextquelle ist: „`location`“.

[39] Für diese und andere Kontextquellen werden im Weiteren auch die äquivalent ins Deutsche übersetzten Namen verwendet.

- `WeatherSource`: Diese Kontextquelle ermittelt die für die spezifizierten geographischen Koordinaten aktuelle Wetterlage. Der Name dieser Kontextquelle ist: „`weather`“.
- `WeatherPollenSource`: Diese Kontextquelle ermittelt die für die spezifizierten geographischen Koordinaten aktuelle Pollenbelastung. Der Name dieser Kontextquelle ist: „`weatherPollen`“.
- `UserIDSource`: Diese Kontextquelle „ermittelt“ das Benutzerprofile. Tatsächlich parst sie nur Daten des entsprechenden Sensors, erzeugt daraus eine Instanz der Kontextinformation und hält diese auf Anfrage bereit. Der Name dieser Kontextquelle ist: „`userID`“.

5.1.3 Cache

Zum Beibehalten und schnellem Zugreifen auf die Kontextinformationen wurde ein Speicher entwickelt, der die Bezeichnung „Cache“ erhielt. Diese impliziert nur die Eigenschaft einer schnellen Arbeitsweise dieses Speichers, jedoch keinesfalls seine Ähnlichkeit zu den Hardwarecaches bzw. zu den dort eingesetzten Techniken. Ein Speicher für die Kontextinformationen ist deshalb wünschenswert, weil ihre Ermittlung sehr zeitaufwändig sein kann (etwa eine Sekunde beim Aufrufen von Web Services) und kostenpflichtig ist, wenn bestimmte Web Services genutzt werden. Außerdem lässt sich mit Hilfe eines solchen Speichers die Historie von Kontextinformationen verfolgen.

Cache ist assoziativ, d.h. er speichert aus beliebig vielen Attribute-Wert Paaren bestehende Tupel und ermöglicht die Werte bestimmter Attribute von Tupeln über die Werte beliebiger andrer Attribute bzw. bei der Angabe von Intervallen, in denen diese Werte liegen sollen, ausfindig zu machen. Folglich unterliegen die in Cache zu speichernde Attributwerte der Einschränkung vergleichbar zu sein, also das Java Interface `Comparable` zu implementieren. Neben den Intervallen für Attributwerte ist es auch möglich die Suche auf diejenige Tupeln zu beschränken, deren Attributwerte nach einem bestimmten Zeitpunkt im Cache gespeichert wurden. (Dabei gilt als Speicherungszeitpunkt das jeweils aktuelle Ergebnis der Java Methode `currentTimeMillis()`, also die Anzahl der seit dem 1.1.1970 vergangenen Millisekunden.)

Gemäß Vorgaben von SmartWeb gilt für Cache Folgendes. Dem Cache steht nur der Arbeitsspeicher zur Verfügung, d.h. erstens, die Kapazität von Cache ist entsprechend begrenzt, und zweitens, Cache existiert nur solange die ihn betreibende Anwendung ausgeführt wird. Der Überlauf, also die Überschreitung der maximalen Speicherkapazität, und die Verdrängungsstrategie wurden in Cache nicht behandelt. Dies stellt natürlich eine Menge von Nachteilen dar, wenn man Cache in anderen Anwendungen wiederverwenden möchte. Cache wurde jedoch primär zum Einsatz in SmartWeb entwickelt und wegen einer Vielzahl von sonstigen Aufgaben höherer Priorität wurde es entschieden, diese SmartWeb nicht beeinträchtigende Nachteile unbehandelt zu lassen.

5.1.3.1 Methoden

Cache ist eine zentrale Komponente: er existiert nur in einem Exemplar und verwaltet die kontextuellen Informationen bezüglich aller Benutzer. Für diese Zwecke wurden vier unten angegebene Methoden implementiert (es sind jeweils die Signatur und die lesbarere Form angegeben; der Letzteren ist jeweils zu entnehmen, dass die Vektoren `types`, `values`, `valueThresholds` und `timeThresholds` gleichlang sein sollen; ferner stimmen die Reihenfolgen der Elemente in diesen Vektoren miteinander überein):

1. *synchronized*[40] *static<CacheEntry extends Comparable> void store(*
Object[] types, CacheEntry[] values)

---

store (type$_1$,…, type$_n$, value$_1$,…,value$_n$)

2. *static synchronized Vector<Vector<Object[]>> get(*
Object[] types, Object[] values,
long[] timeThresholds, float[] valueThresholds,
Object[] obligatoryRequestTypes, Object[] optionalRequestTypes)

---

get(type$_1$,…, type$_n$,
value$_1$,…, value$_n$,
time_threshold$_1$,…, time_threshold$_n$,
value_threshold$_1$,…, value_threshold$_n$,
obligatory_request_type$_1$,…, obligatory_request_type$_m$,
optional_request_type$_1$,…, optional_request_type$_k$)

[40] Dadurch dass diese und die anderen Cachezugriffsmethoden synchronisiert sind, wird es auf einfachste (jedoch nicht ideale) Weise für die Ermöglichung des nebenläufigen Zugriffs gesorgt.

3. *static synchronized void removeRecords(*
Object[] types, Object[] values,
long[] timeThresholds, float[] valueThresholds)

---

removeRecords(type$_1$,…, type$_n$,
value$_1$,…, value$_n$,
time_threshold$_1$,…, time_threshold$_n$,
value_threshold$_1$,…, value_threshold$_n$)

4. *static synchronized void removeValues(*
Object[] types, Object[] values,
long[] timeThresholds, float[] valueThresholds,
Object[] typesToRemove)

---

removeValues(type$_1$,…, type$_n$,
value$_1$,…, value$_n$,
time_threshold$_1$,…, time_threshold$_n$,
value_threshold$_1$,…, value_threshold$_n$,
typeToRemove$_1$,..., typeToRemove$_n$)

Im Folgenden wird die Funktionalität der obigen Methoden anhand von Beispielen, dargestellt in der Abbildung 5.3, erläutert. Diese Abbildung besteht aus vier Tabellen, die jeweils einen Zustand des Cache repräsentieren. Die erste Spalte von links jeder Tabelle zeigt die Nummerierung der Einträge in Cache in der Speicherungsreihenfolge an; man nehme Einfachheit halber an die Nummerierungszahlen entsprechen dem Speicherungszeitpunkt des jeweiligen Eintrags. Der Cacheeintrag bestehe jeweils aus einer Zeile (außer der Nummerierungsspalte) und den dazugehörigen Attributnamen. Zum Beispiel ist der erste Eintrag in der ersten Tabelle: `((Benutzer,1),(Ort,4),(Wetter,2))` ; er wurde zum Zeitpunkt `1` erstellt. Es gilt also vereinfachend die Annahme, dass alle Attribute als Wertebereich die natürlichen Zahlen haben.

NR	Benut-zer	Ort	Wetter
1.	1	4	2
2.	2	1	5
3.	3	3	4

1

Nr	Benut-zer	Ort	Wetter	Zeit
1.	1	4	2	-
2.	2	1	5	-
3.	3	3	6	-
4.	1	2	1	1

2

Nr	Benut-zer	Ort	Wetter	Zeit
2.	2	1	5	-
3.	3	3	6	-
4.	1	2	1	1

3

Nr	Benut-zer	Ort	Wetter
2.	2	1	5
3.	3	3	6
4.	1	2	1

4

Abbildung 5.3: Beispiel für die Operationen in Cache.

`store:`

Man betrachte die Tabelle 1 aus der Abbildung 5.3. Angenommen, Cache hat in diesem Zustand folgende Anweisung erhalten:

```
Cache.store([Benutzer,Ort,Wetter,Zeit],[1,2,1,1]);
```

Die Methode „`store`“ speichert die übergebenen Attribute-Wert Paare in Cache; dabei wird jedes Mal ein neues Cacheeintrag erzeugt. In diesem Fall handelt es sich um den Eintrag 4, der erzeugt wird; der neue Cachezustand ist in der Tabelle 2 (Abbildung 5.3) wiedergegeben. Diese Tabelle erweckt den Anschein, dass für das neue Attribut „`Zeit`“ eine neue Spalte erzeugt wurde. Da aber die interne Struktur von Cache nicht tabellarisch ist, werden in diesem Sinne keine Spalten erzeugt, so dass die vorigen Einträge von neuem Attribut „unberührt“ bleiben (ihnen wird also nicht nachträglich das neue Attribut mit dem Wert „`null`“ hinzugefügt). Eine solche Funktionsweise scheint vorteilhaft zu sein im Vergleich zu den Datenbanken: in diesen hätte man bei jedem neuen Eintrag zuerst prüfen sollen, ob alle Spalten bereits existieren, und dann gegebenenfalls die entsprechende Tabelle um einige neue Spalten erweitern. Diese Operationen sind zeitaufwändig.

`get:`

Man gehe nun von dem der Tabelle 2 entsprechenden Cachezustand aus. Angenommen, Cache hat folgende Anweisung erhalten:

```
Cache.get([Benutzer,Ort],[null,3],[-1,2],[-1,1], [Wetter],[Zeit]);
```

Die Methode „`get`“ extrahiert aus dem Cache die durch ihre Parametern spezifizierten Attribut-Wert Paare. Übersetzt in die natürliche Sprache bedeutet die obige Anweisung:

Von jedem Eintrag in Cache, bei dem irgendein Wert für „Benutzer“ vorhanden und der Wert für „Ort“ 3 ± 1 ist und nicht früher als vor 2 Zeiteinheiten gespeichert wurde, und ein Wert für „Wetter“ existiert, gib den Letzteren und, falls vorhanden, auch den Wert für „Zeit“ zurück.

Die Vektoren `types`, `values`, `timeThresholds` und `valueThresholds` spezifizieren allesamt diejenige Einträge in Cache, aus denen die Rückgabemenge bestimmt wird. Dabei gilt, dass die Werte im Vektor `values` `null` sein dürfen. Dies entspricht der Forderung, dass in dem spezifizierten Eintrag die entsprechenden Attribute existieren sollen, wobei es keine Rolle spielt, welche Werte sie haben, bzw. zu welchem Zeitpunkt sie in Cache gespeichert wurden.

Für den Vektor `timeThresholds` gilt:

- Hat ein Element aus diesem Vektor Wert `-1`, so ist die Zeitschranke für das entsprechende Attribut unendlich groß, d.h. es spielt keine Rolle, wann die Werte dieses Attributes in Cache gespeichert wurden.
- Hat ein Element aus diesem Vektor Wert `t` größer gleich `0`, und ist die aktuelle Zeit `T`, so müssen die Werte des entsprechenden Attributes im Zeitintervall `T-t` in Cache gespeichert worden sein.
- Ist dieses Vector gleich `null`, so ist es gleichbedeutend mit dem Wert `-1` für jedes Element dieses Vektors.

Für den Vektor `valueThresholds` gilt:

- Hat ein Element aus diesem Vektor Wert `-1`, so ist die Schranke für das entsprechende Attribut unendlich groß, d.h. es spielt keine Rolle, welche Werte dieses Attribut hat.
- Hat ein Element aus diesem Vektor Wert `x` größer gleich `0`, dann muss für jedes Wert `y` des entsprechenden Attributes eines Eintrags in Cache gelten: `|z-y|≤ x`, wobei `z` das entsprechende Element aus dem Vektor `values` ist.
- Ist dieser Vector gleich `null`, so ist es gleichbedeutend mit dem Wert `0` für jedes Element dieses Vektors.

Das Ergebnis der Methode „`get`“ ist eine Matrix, deren jedes Element ein Attribut-Wert Paar ist. Jede Zeile dieser Matrix entspricht einer geordneten Menge von solchen Attribut-Wert Paaren, die alle demselben Cacheeintrag gehören. Somit repräsentiert jede Zeile der Matrix eine Antwort auf die durch die „`get`“ Methode gestellte Anfrage.

Gemäß der gegebenen Spezifikationen und unter der Annahme, der aktuelle Zeitpunkt sei `4`, lautet das Ergebnis der obigen Anfrage:

(Wetter,6)	
(Wetter,1)	**(Zeit,1)**

removeRecords:

Die Methode „`removeRecords`“ entfernt bestimmte Einträge aus dem Cache. Die zu entfernende Einträge werden genau so wie bei der Methode „`get`“ durch die Vektoren `types`, `values`, `timeThresholds` und `valueThresholds` spezifiziert. Als Beispiel nehme man erneut an, Cache sei in dem der Tabelle 2 entsprechenden Zustand, und er hätte folgende Anweisung erhalten:

```
Cache.removeRecords([Benutzer,Ort],[1,4],null,[0,1]);
```

Es wird also angeordnet, alle Einträge, bei denen „`Benutzer`“ den Wert `1` und „`Ort`“ den Wert `4±1` hat, zu entfernen. Das Ergebnis dieser Anweisung ist in der Tabelle 3 der Abbildung 5.3 dargestellt: der erste Eintrag ist gelöscht.

removeValues:

Die Methode „`removeValues`“ entfernt aus Cacheeinträgen, die wie vorhin spezifiziert werden, alle Attribute (bzw. die dazugehörigen Werte), die in `typesToRemove` spezifiziert sind. Als Beispiel gehe man nun von der Tabelle 3 aus und betrachte folgende Anweisung:

```
Cache.removeValues([Wetter],[1],null,null,[Zeit]);
```

Es soll also der Wert für „`Zeit`“ aus allen Einträgen gelöscht werden, die für „`Wetter`“ den Wert `1` haben. Das Ergebnis dieser Anweisung ist in der Tabelle 4 dargestellt. Im vierten Eintrag wurde der Wert für „`Zeit`“ gelöscht. In der tabellarischen Darstellung des Cache wurde auch die ganze Spalte des Attributes „`Zeit`“ entfernt, weil kein Eintrag in Cache einen Wert für dieses Attribut hat. Die Architektur des Cache ermöglicht eine sehr einfache und schnelle Entfernung eines Attributes: dies erfolgt automatisch, ohne jeglicher zusätzlicher Anweisungen und ohne einer Notwendigkeit in der internen Cachestruktur irgendwelche zeitintensive Umstrukturierungen auszuführen.

5.1.3.2 Aufbau

Cache ist auf Basis von Hashtabellen aufgebaut und weist intern folgende Struktur auf:

```
HashMap<Object,BinaryHashMap<String,Centry>>
```

Es wird also im Cache repräsentierenden `HashMap` einem `Object` ein gewisses `BinaryHashMap`[41] zugeordnet. Das `Object` stellt dabei den Namen eines Attributes dar und das `BinaryHashMap` enthält alle Werte für dieses Attribut, die in der Menge von Cacheeinträgen vorkommen. Ein `BinaryHashMap` ist eine Hashtabelle, bei der die Schlüssel über die ihnen entsprechenden Objekte ebenfalls abrufbar sind. Dies, sowie die Speicherung und Entfernung von Schlüssel-Objekt Paaren aus einem `BinaryHashMap`, ist ohne Verlust der asymptotischen Zeitkomplexität möglich, nur wenn einem Objekt in `BinaryHashMap` genau ein Schlüssel entspricht. In dieser Anwendung von `BinaryHashMap` konnte dies gewährleistet werden, da jedes solches wie folgt aufgebaut ist: Einem Schlüssel, der durch einen `String` repräsentierten Zeitpunkt[42] der Speicherung des Attributwertes in Cache darstellt, wird ein Attributwert, umhüllt in ein `Centry` Objekt, gegenübergestellt. Gerade diese Umhüllung gewährleistet, dass selbst wenn ein Attribut in verschiedenen Cacheeinträgen gleichen Wert hat, alle umhüllende `Centry` Objekte unterschiedlich sind.

Entsprechend der angegebenen Struktur lässt sich der Cachezustand aus der Abbildung 5.3, Tabelle 3 wie in der Abbildung 5.4 darstellen (wiederum mit der vereinfachenden Ersetzung des Speicherungszeitpunktes durch frei gewählte Zahlen). Die Zusammengehörigkeit der einzelnen Attribut-Wert Paare zu einem Cacheeintrag ist dabei durch die Übereinstimmung des Speicherungszeitpunktes gewährleistet: alle Attributwerte mit demselben `String` Schlüssel des `BinaryHashMap` gehören zu einem Cacheeintrag.

---

[41] Die Klasse `BinaryHashMap` wurde im Rahmen dieser Arbeit implementiert. Sie basiert auf den Hashtabellen und ist vergleichsweise einfach. Deshalb wird ihre Realisierung hier nicht diskutiert.

[42] Der Zeitpunkt ist ein `String`, da er intern durch ein Paar aus Milli- und Nanosekunden repräsentiert ist. Die Nanosekunden sind insbesondere bei einer unmittelbar nacheinander folgenden Speicherung der Einträge in Cache notwendig, da in diesem Fall die Auflösung von Millisekunden zu gering ist. Es kann also ohne Nanosekundengenauigkeit gegebenenfalls nicht betont werden, dass zwei Einträge nacheinander gespeichert wurden. Die Nanosekundenzeit wird ab Java 5 mit dem Aufruf `System.nanoTime()` ermittelt. Jedoch ist einzig das Ergebnis dieser Methode zum Repräsentieren der aktuellen Zeit nicht ausreichend, da es keinen Bezugpunkt zu irgendeinem Datum hat, d.h. es wiederholt sich in bestimmten Zeitabständen.

Object	BinaryHashMap	
	String	Centry
Benutzer	2	1
	3	3
	4	1
Ort	2	1
	3	3
	4	2
Wetter	2	5
	3	6
	4	1
Zeit	4	1

Abbildung 5.4: Struktur des Cache am Beispiel des Cachezustandes wie in der Abbildung 5.3, Tabelle 3.

5.1.3.3 Funktionsweise im Überblick

Die Funktionsweise der Methode „`store`“ ist trivial: es werden gegebenenfalls neue `BinaryHashMap` erzeugt; danach werden einigen `BinaryHashMap` Schlüssel-Objekt Paare hinzugefügt.

Wesentlich anspruchsvoller sind die anderen oben genannten Methoden. In diesen findet, wie bereits erwähnt, die Suche nach denjenigen Cacheeinträgen statt, die durch die Parameter `types`, `values`, `timeThresholds` und `valueThresholds` spezifiziert sind. Dies kann man sich so vorstellen, dass jedes `i`-te Quadrupel:

$$(type_i, value_i, time\_threshold_i, value\_threshold_i)$$

eine Menge M_i von Cacheeinträgen spezifiziert und man dabei eine Menge `M` sucht, die Schnitt dieser Mengen von `1` bis $|types|$ ist. `M` kann bestimmt werden, indem man für irgendein `i` M_i bestimmt und dann für jeden Cacheeintrag aus M_i prüft, ob jedes `j`-te Quadrupel (auch für `j=i`) ihn spezifiziert. Wenn dies zutrift, wird dieser Cacheintrag der Menge `M` hinzugefügt. Bei diesem Vorgehen stellen sich zwei Fragen:

1. Wie wird für irgendein `i` M_i bestimmt (ohne alle Cacheeintrage betrachten zu müssen)?
2. Für welches `i` hat M_i die kleinste Kardinalität? (Dies ist aus Gründen der Performanz wichtig, weil alle Einträge aus M_i behandelt werden müssen.)

Die erste Frage löst sich einfach für alle i, bei denen gilt:

$$valueThreshold_i = 0 \land value_i \neq null$$ [43]

Wenn es ein solches i gibt, dann sei es im Folgenden vom günstigeren Fall die Rede; sonst – von einem ungünstigen. Für jedes solches i lassen sich alle Cacheeinträge (bzw. jeden Cacheeintrag eindeutig identifizierender Speicherungszeitpunkt), bei denen Attribut $type_i$ den Wert $value_i$ hat, dank der speziellen Eigenschaft der `BinaryHashMap` durch das Abfragen von Schlüsseln, die zum Wert $value_i$ gehören, in einer simplen Operation ermitteln. Diese Operation basiert auf der „`get`“ Methode, welche die Hashtabellen anbieten[44]; die asymptotische Komplexität dieser Operation ist dieselbe wie bei dieser „`get`“ Methode.

Im ungünstigen Fall ist bei der vorliegenden Implementierung von Cache bei der Bestimmung von M_i (für jedes i) notwendig alle Eintrage in Cache zu behandeln. Weil es extrem zeitaufwändig ist, wird es dringend davon abgeraten, solche Anfragen an Cache vorzunehmen.

Wenn die erste Frage auf die günstigere Weise gelöst werden konnte, lässt sich die zweite Frage für diese i ebenfalls einfach beantworten, denn die Bestimmung der Kardinalität einer Menge erfolgt in Java durch einen schnellen und simplen Methodenaufruf. In diesem Fall wählt man also für die weitere Behandlung aus allen Mengen M_i, bei denen i die besagte Eigenschaft hat, diejenige mit der kleinsten Kardinalität aus.

Im ungünstigen Fall ist die Suche nach der M_i mit der kleinsten Kardinalität zeitaufwendiger, als die weitere Behandlung dieser Menge. In diesem Fall wird es somit von der Bestimmung eines solchen M_i abgesehen. Stattdessen werden in Cache, wie bereits bemerkt, alle Einträge behandelt.

[43] `valueThresholds = null` sei gleichwertig mit $valueThreshold_i = 0$ für alle i.
[44] Es handelt sich dabei (hier und im Weiteren) um die Methode `public Object get(Object key)` in der Klasse `java.util.HashMap`.

5.1.3.4 Zeitkomplexität

Das beschriebene Verfahren zeigt wie die Menge `M` von Cacheeinträgen, die durch die Parameter `types`, `values`, `timeThresholds` und `valueThresholds` spezifiziert sind, bestimmt werden kann. Dazu sei es angemerkt, dass die Reduzierung der Kardinalität von `M` (bzw. von jeweiligen M_i) durch die Angabe von Zeitschranken leider nicht zu einer Verringerung des zeitlichen Aufwandes führt, der zur Bestimmung von `M` notwendig ist.

Im günstigeren Fall haben die Methoden „`get`“, „`removeRecords`“ und „`removeValues`“ folgende asymptotische Zeitkomplexität (ohne Nachweis):

$$O(|M| \bullet |types| \bullet G) ,$$

wobei `types` der entsprechende Parameter jeder dieser Methoden bezeichnet und `G` – die Zeitkomplexität der von den Hashtabellen bekannten „`get`“ Methode.

Im ungünstigen Fall haben die besagten Methoden folgende asymptotische Zeitkomplexität (ohne Nachweis):

$$O(|Cache| \bullet |types| \bullet G) ,$$

wobei `|Cache|` die Anzahl von Einträgen in Cache bezeichnet. In diesem Fall degradiert die Performanz von Cache zu einer trivialen Suche in einer Tabelle.

5.1.3.5 Sortierung von Suchergebnissen

Wie bereits erwähnt, liefert eine „`get`“-Anfrage an Cache eine Matrix, deren jede Zeile ein Ergebnis dieser Anfrage darstellt. Falls dabei der Vektor `valueThresholds` in der Anfrage kein Nullvektor ist, kann es vorkommen, dass die Attributwerte der von dieser Anfrage spezifizierten Cacheeinträgen mit den entsprechenden Attributwerten im Vektor `values` der Anfrage nicht übereinstimmen. Zum Beispiel spezifiziert die „`get`“-Anfrage in der Abbildung 5.5 die Einträge `1` und `2`; die Attributwerte für „`Benutzer`“, „`Ort`“ und „`Wetter`“ in diesen Einträgen unterscheiden sich von den Werten entsprechender Attribute in der Anfrage. Man stellt sich nun die Frage, ob der Eintrag 1 oder 2 besser mit der Anfrage übereinstimmt, denn davon hängt es ab, welcher der beiden Ergebnisse – sie sind ebenfalls in der Abbildung 5.5 dargestellt – in höherem Maße der Anfrage genügt.

Nr	Benut-zer	Ort	Wetter	Zeit
1.	1	4	2	-
2.	2	1	5	-
3.	3	3	6	-
4.	1	2	1	1

```
Cache.get(
    [Benutzer,Ort,Wetter],[1,3,4],
    null,[1,2,2],[Benutzer],null)
                ↓
        (Benutzer,1)
        (Benutzer,2)
```

Abbildung 5.5: Beispiel für eine „`get`“-Anfrage an Cache und ihre zwei Ergebnisse.

In Cache ist ein Ansatz implementiert, der dieser Frage nachgeht und zwar bezüglich aller solcher Attribute, deren Werte derartig auf die Menge von ganzen Zahlen abgebildet werden können, dass diese Abbildung den Absolutbetrag dieses Attributwertes repräsentiert (zum Beispiel könnten die Werte von „`Wetter`“ in „unangenehm“, „mittelmäßig“ und „schön“ bzw. in 1, 2 und 3 diskretisiert werden). Dabei wird der Gedanke verfolgt, der Unterschied zwischen zwei Werten eines Attributes sei umso unbedeutender, je größer das Intervall auf der Menge der ganzen Zahlen ist, in dem die bisher bekannten Werte dieses Attributes liegen. Der Unterschied zweier Attributwerte wird also mit dieser Intervallbreite (auf 1) normiert. Die Intervallbreite `B` eines Attributes `A` ist eine dynamische, monoton wachsende Größe. Sie wird bei jeder Speicherung eines Wertes `W` für dieses Attribut auf folgende Weise neu bestimmt:

$$U_B := \min(W_Z, U_B);$$
$$O_B := \max(W_Z, O_B);$$
$$B := O_B - U_B;$$

dabei wird mit U_B die bisherige untere Grenze und mit O_B – die bisherige obere Grenze des Intervalls bezeichnet. Am Anfang sei $U_B = -\infty$ und $O_B = \infty$. W_Z ist die Abbildung von `W` auf die Menge der ganzen Zahlen; falls `A` nicht auf die geforderte Weise auf die ganzen Zahlen abgebildet werden kann, sei $W_Z = 0$.

Aus obiger Überlegung ergibt sich der Abstand `D` eines Cacheeintrags `C` zu der Anfrage wie folgt:

$$D = \sum_{i=1}^{|values|} \frac{|W_i - A_i|}{B_i}$$

wobei:

$$A_i = \begin{cases} values_{i_z} & , \text{ falls } values_i \neq null \\ W_i & , \text{ sonst} \end{cases}$$

Die Bezeichnungen haben dabei folgende Bedeutung:

`values`$_{iz}$: Abbildung des `i`-ten Attributwertes aus dem Vektors `values` der Anfrage auf die Menge der ganzen Zahlen.

W_i: Wert, den das `i`-te Attribut aus dem Vektor `types` der Anfrage im Cacheeintrag `C` hat.

B_i: aktuelle Intervallbreite des `i`-ten Attributes aus dem Vektor `types` der Anfrage.

Für das Beispiel aus der Abbildung 5.5 ergibt sich aus den Attributen „`Benutzer`“, „`Ort`“ und „`Wetter`“ folgender Abstand des ersten Cacheeintrags zu der Anfrage:

$$0+\frac{|4-3|}{3}+\frac{|2-4|}{5}=\frac{11}{15}$$

Der Abstand des zweiten Cacheeintrags zu der Anfrage ist:

$$\frac{|2-1|}{2}+\frac{|1-3|}{3}+\frac{|5-4|}{5}=\frac{41}{30}$$

Folglich genügt das erste Ergebnis in höherem Maß der Anfrage.

Die Bestimmung des besten Ergebnisses aus der Rückgabemenge `R` hat vergleichsweise geringe Zeitkomplexität:

$$O(|\mathtt{R}|\bullet|\mathtt{types}|)$$

Sie ist geringer als die in 5.1.3.4 angegebene Zeitkomplexität der Methode „`get`“.

Cache positioniert das beste Ergebnis an die erste Stelle der Rückgabemenge der „`get`“-Anfrage. Prinzipiell ist es möglich, diese Menge auf dieselbe Weise vollständig zu sortieren, doch der damit verbundene zeitliche Aufwand wäre höher als der zur Bestimmung der Menge von Ergebnissen benötigte[45]. So wurde dies nicht implementiert.

In Wirklichkeit, also abweichend von dem obigen Beispiel, hat ein durch zwei oder drei Koordinaten angegebener Ort keinen Absolutbetrag. Da aber der Abstand zweier Orte voneinander ohne des Absolutbetrags bestimmt wird, könnte dieselbe Vorgehensweise wie oben auf solche mehrdimensionale Attribut dann übertragen werden, wenn die Breite desjenigen Intervalls bestimmt werden könnte, das eine beliebige Menge von

[45] Unter der Annahme, dass die Zeitkomplexität von Sortieralgorithmen bestenfalls $O(\mathtt{n}\bullet\mathtt{log(n)})$ ist.

Werten dieses Attributs aufspannt. Man kann zwar als Interwallbreite den größten Abstand zweier Werte aus der gegebenen Menge festlegen, doch in dieser Arbeit wird die Frage offen gelassen, ob die derartig definierte Intervallbreite effizient bestimmt werden kann. Alternativ dazu können mehrdimensionale Attribute durch mehrere eindimensionale ersetzt werden. Der Nachteil dabei jedoch ist, dass die Werte eines mehrdimensionalen Attributs, die innerhalb eines bestimmten Abstandes zu einem gegebenen Wert liegen, bei der Ersetzung dieses Attributes durch mehrere eindimensionale Atrribute nicht bestimmt werden können.

Schließlich sei der Vollständigkeit halber bemerkt, dass um die Abbildung eines Attributwertes `X` auf die ganzen Zahlen zu erhalten, verlangt Cache, sie durch den Aufruf `X.compareTo(null)` zu ermitteln. Falls `X` nicht auf die geforderte Weise auf die ganzen Zahlen abgebildet werden kann, soll dieser Aufruf die größte `Integer` Zahl liefern.

5.1.3.6 Abschließende Bemerkung

Der Einblick in Cache wird an dieser Stelle durch folgende abschließende Ergänzung vervollständigt. Cache betrachtet Attributwerte `X` und `Y` genau dann als gleich, wenn gilt:

```
X.toString().equals(Y.toString()) == true
```

5.1.4 Kontextinterface

Kontextinterface ist die zentrale Komponente der entwickelten Infrastruktur; sie ist in der Klasse `ContextInterface` implementiert. Kontextinterface repräsentiert einen Benutzer des Dialogsystems (also eine Person) während einer seiner Session. Über das Kontextinterface erfolgt die Abfrage von auf diesen Benutzer bezogenen Kontextinformationen und die Aktualisierung von Daten derjenigen Sensoren, die sich auf diesen Benutzer bzw. seine Umgebung beziehen. Dazu stellt das Kontextinterface folgende Methoden zur Verfügung.

void update(String xml)

Empfängt die in `xml` übergebenen Sensordaten und aktualisiert die betroffenen Kontextinformationen (gegebenenfalls speichert sie auch in Cache). Es wird zunächst allen Kontextquellen angeboten `xml` zu parsen (siehe Methode `parse(String xml)` in 5.1.2). Dann werden von allen solchen Kontextquellen kontextuelle Informationen ermittelt, die `xml` als eine für sie adressierte Nachricht detetktieren. Schließlich werden die ermittelten Kontextinformationen als ein Eintrag in Cache gespeichert.

ContextSourceValueInterface[] getContextInformation(
String[] types, ContextSourceValueInterface[] values,
String type)

In `types` und `type` sind die Namen von Kontextquellen angegeben (siehe 5.1.2); `values` enthält Kontextinformationen der Kontextquellen aus `types` (in übereinstimmender Reihenfolge). Wenn `types = null` und `values = null`, wird die zuletzt ermittelte Kontextinformation der Quelle `type` zurückgegeben, vorausgesetzt sie ist bezüglich der für `type` geltenden Zeitschranke nicht veraltet. Soll sie veraltet sein, wird sie neu ermittelt.

Wenn `types` ungleich `null`, wird folgende „`get`"-Anfrage[46] an Cache ausgeführt:

```
Cache.get([„UserID”|types],[<userIDValue>|values],
            timeThresholds,valueThresholds,type)
```

Das Ergebnis dieser Anfrage wird zurückgegeben. Es werden also Kontextinformationen von `type` gesucht, die im Zusammenhang mit den in `values` angegebenen Kontextinformationen der Quellen aus `types` vorkommen. (Die Zeit- und Größenschranke für die jeweilige Kontextquelle wird bei der Initialisierung dieser Instanz von `ContextInterface` spezifiziert, siehe 5.3.2.)

[46] Die Schreibweise `[X,Y|Z]` bedeutet: dem Vektor `Z` werden Element `X` und `Y` hinzugefügt. `<userIDValue>` steht für die Variable der Klasse `UserIDSource`, die das Benutzerprofil desjenigen Benutzers repräsentiert, der mit der betrachteten Instanz von `ContextInterface` assoziiert ist.

ContextSourceValueInterface[] getAndSave(
String[] types, ContextSourceValueInterface[] values, String type)

Diese Methode macht das gleiche wie die vorige Methode, bis auf Folgendes: Wenn das Ergebnis der „`get`“-Anfrage `null` ist, wird eine Kontextinformation `newValue` über die Quelle `type` ermittelt und zurückgegeben. Außerdem wird dem Cache folgender Eintrag hinzugefügt:

```
Cache.store([„UserID”,type|types],
            [<userIDValue>,newValue|values])
```

Es wird also behauptet, dass die neu ermittelte Kontextinformation im Zusammenhang mit den Kontextinformationen in `values` betrachtet werden kann.

ContextSourceValueInterface[][] getContextInformation(
String[][] types, ContextSourceValueInterface[][] values, String[] type)

Diese Methode macht das gleiche wie die obige `getContextInformation` Methode, jedoch für die zu ermittelnden Kontextinformationen von mehreren Quellen zugleich (dabei werden für jede dieser Quellen eigene Vektoren `values` und `types` spezifiziert).

ContextSourceValueInterface[][] getAndSave(
String[][] types, ContextSourceValueInterface[][] values, String[] type)

Diese Methode macht das gleiche wie die obige `getAndSave` Methode, jedoch für die zu ermittelnde Kontextinformationen von mehreren Quellen zugleich (dabei werden für jede dieser Quellen eigene Vektoren `values` und `types` spezifiziert).

5.1.4.1 Multithreading

Weil die Ermittlung von Kontextinformationen, besonders wenn dabei die Web Services hinzugezogen werden, sehr zeitaufwändig sein kann, lohnt es sich, zur Reduzierung dieses Aufwandes, von Multithreading Gebrauch zu machen. Im Kontextinterface geschieht es auf folgende Weise:

- Sollen Kontextinformationen unterschiedlicher Quellen ermittelt werden, erfolgt dies nebenläufig, in einem eigenen Thread für jede dieser Quellen. Dies verkürzt

die Antwortzeit deutlich, denn jeder Zugriff auf ein Web Service zu einer nicht vernachlässigbaren Wartedauer auf die Antwort dieses Web Services führt. Durch die nebenläufigen Zugriffe werden diese Wartezeiten parallelisiert.

- Die von der Methode `update(String xml)` ausgelösten Ermittlungen von kontextuellen Informationen werden in eigenem Thread, sozusagen im Hintergrund, durchgeführt. Dadurch wird die Ausführung dieser Methode abgeschlossen, bevor die kontextuellen Informationen ermittelt sind. Dies ist auch vollkommen sinngemäß, denn `update(String xml)` keine Rückgabe liefert. Der Nutzen dieser Vorgehensweise sei am folgenden einfachen Beispiel demonstriert: der Code

```
update(GPS);
timeValue = get(null,null,"Zeit");
```

  löst zunächst Ermittlung einer Kontextinformation der Quelle „`Ort`“ (sowie einiger anderer Kontextquellen) aus, weil sie von GPS Koordinaten abhängig ist. Dann wird nach einer Kontextinformation der Quelle „`Zeit`“ gefragt. Dank der hintergründigen Ausführung von Update wird die Kontextinformation der Quelle „`Zeit`“ sehr viel früher zurückgegeben, als sich „`Ort`“ mittels des Zugriffes auf das entsprechende Web Service ermitteln lässt.

5.2 Repräsentierung von Kontextinformationen

Bisher wurde es von Kontextinformationen gesprochen, ohne zu präzisieren, was genau darunter zu verstehen ist. Gemäß der in dieser Arbeit verfolgten Idee sollen Kontextinformationen zweierlei repräsentiert werden können: nicht ontologiebasiert als Objekte in Java und als A-Boxen der Basisontologie.

5.2.1 Repräsentierung als Objekte in Java

Eine Kontextinformation ist stets eine Instanz einer von `ContextSourceValueInterface` abgeleiteten Klasse. Für jede Kontextquelle wurde eine solche abgeleitete Klasse implementiert (siehe die Abbildung 5.6). Prinzipiell ist es auch möglich, dass zwei Kontextquellen Kontextinformationen derselben Klasse ermitteln.

Kontextquelle	Klasse von Kontextinformationen
TimeSource	TimeSourceValue
LocationSource	LocationSourceValue
WeatherSource	WeatherSourceValue
WeatherPollenSource	WeatherPollenSourceValue
UserIDSource	UserIDSourceValue

Abbildung 5.6: Kontextquellen und die ihnen jeweils entsprechende Kontextinformation-implementierende Klasse.

Folgende zwei Methoden von `ContextSourceValueInterface` soll jede Kontextinformation-repräsentierende Klasse (unter Erhaltung der unten genannten Funktionalität) überschreiben:

`int compareTo(Object toCompare)`

Vergleicht diese Kontextinformation mit einer anderen Kontextinformation derselben Klasse. Wenn `toCompare = null`, gibt Absolutbetrag dieser Kontextinformation zurück; falls kein solcher existiert, gibt `Integer.MAX_VALUE` zurück (siehe 5.1.3.5).

`String toString()`

Gibt diese Kontextinformation zurück. Dabei ist die Anmerkung in 5.1.2.6 unbedingt zu beachten.

Im Folgenden wird ein kurzer Überblick über die in der Abbildung 5.6 genannten Klassen gegeben.

`TimeSourceValue:`

Die Zeit wird durch das Ergebnis der Java Methode `System.currentTimeMillis()` (die Anzahl der seit dem 1.1.1970 vergangenen Millisekunden) und durch eine Instanz der Klasse `java.Util.Calendar` repräsentiert. Die „`compareTo`“-Methode liefert die Differenz der Werte für Millisekunden zweier Instanzen von `TimeSourceValue`. Als Absolutbetrag wird der Millisekundenwert Modulo größte `Integer` Zahl zurückgegeben.

LocationSourceValue:

Eine Instanz dieser Klasse enthält die GPS Koordinaten des repräsentierten Ortes und die dazu am besten passenden Postadressen[47], also Hausnummer, Straßenname, Stadtname usw. Die Postadresse wird von Web Service Tinfo ermittelt. Als Vergleichwert zweier Instanzen dieser Klasse dient der euklidische Abstand; der Absolutbetrag wird, wie in 5.1.3.5 angemerkt, nicht angegeben.

WeatherSourceValue:

Der aktuelle Wetterstand kann mit Hilfe von Web Service Tinfo in Textform (als kurzer Text aus zwei-drei Stichworten oder als längere Beschreibung in natürlicher Sprache) bzw. als niedrigste und höchste Lufttemperatur[48] in Celsius ermittelt werden. Als Vergleichswert zweier Instanzen dieser Klasse dient die Differenz der durchschnittlichen Lufttemperaturwerten dieser Instanzen. Da der Wetterstand in Hinblick auf die Lufttemperatur als ein Intervall repräsentiert ist, wird als Absolutbetrag für den Wetterstand die Breite dieses Intervalls zurückgegeben.

WeatherPollenSourceValue:

Die Pollenbelastung kann für diverse Pollenarten mit Hilfe von Web Service Tinfo als nicht vorhandene, schwache, mäßige oder starke ermittelt werden. Zwei Instanzen dieser Klasse können verglichen werden, indem man diesen Belastungsstärken Zahlen zwischen `1` und `4` zuweist und dann die Summe von Differenzen der Pollenbelastungen von je zwei übereinstimmenden Pollenarten, die in diesen Instanzen vorkommen, bildet. Der Absolutbetrag wird durch die Summierung von Pollenbelastungen aller Pollenarten, die in einer Instanz von `WeatherPollenSourceValue` vorkommen, gebildet.

UserIDSourceValue:

Das Benutzerprofil besteht aus diversen Angaben zum Benutzer (siehe 5.1.1). Das Vergleichsergebnis von zwei Benutzerprofilen ist entweder `0` bei der Übereinstimmung oder die größte `Integer` Zahl sonst. Einen Absolutbetrag gibt es nicht.

[47] Es ist möglich, dass zu GPS Koordinaten mehrere Adressen gefunden werden.
[48] Die Lufttemperatur an einem Ort wird offenbar durch das Temperaturenintervall innerhalb einer diesen Ort umgebender Region approximiert.

5.2.2 Ontologische Repräsentierung

Um eine Kontextinformation, also eine Instanz einer der in 5.2.1 genannten Klasse als eine basisontologische A-Box repräsentieren zu können, wird es zunächst von Kontextquellen, die Kontextinformationen dieser Klassen ermitteln, gefordert, dass sie per explizite Angabe mit einer oder mehreren basisontologischen Klassen in Verbindung gebracht werden (dies wird insbesondere in Kapiteln 6 (vor allem 6.3), 7 und 9 relevant) . Dies geschieht, indem in einer Konfigurationsdatei entsprechende Angaben gemacht werden (siehe 5.3.2). Zum Beispiel wird Kontextquelle „`Wetter`“ in der Konfigurationsdatei mit den basisontologischen Klassen `WeatherRegion` und `OutsideTemperatureRegion` assoziiert[49]. `WeatherRegion` ist die Oberklasse für Klassen, die einen Wetterzustand darstellen, und `OutsideTemperatureRegion` – Oberklasse der Lufttemperaturintervalle repräsentierenden Klassen (siehe die Abbildung 5.7). Somit wird es festgelegt, dass Kontextquelle „`Wetter`“ Instanzen von `WeatherRegion` und `OutsideTemperatureRegion` ermittelt oder genauer gesagt, ein Java Objekt, dass Kontextinformation von „`Wetter`“ ist, sich als eine A-Box darstellen lässt, in der Instanzen dieser Klassen vorkommen.

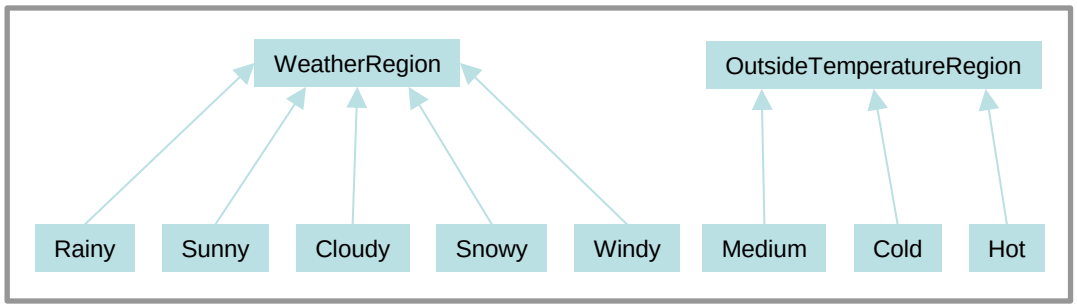


Abbildung 5.7: Klasse `WeatherRegion` der Basisontologie und ihre Unterklassen. (*)

Kontextinformationen als Instanzen der von `ContextSourceValueInterface` abgeleiteten Klassen werden in A-Boxen mit Hilfe folgender zwei Methoden konvertiert:

[49] Vollständigkeit halber sei angemerkt, dass Kontexquelle „`Ort`“ mit basisontologischen Klassen `Address` und `StationaryArtifact` assoziiert ist; „`Zeit`“ – mit `TimePoint` und `TimeInterval`; „`Benutzer`“ – mit `UserProfile`. Kontextquelle „`Pollen`“ bzw. die von ihr ermittelten Kontextinformationen haben keine ontologische Repräsentierung.

`List<String> asOntClassURIs()`

Liefert URI derjenigen basisontologischen Klassen zurück, als deren Instanz diese Kontextinformation darstellbar ist. Wenn es sich beispielsweise um den Wetterzustand handelt und in dieser Kontextinformation die ermittelten regnerischen Wetterverhältnisse festgehalten sind, ist sie als Instanz von `Rainy` darstellbar.

`OntModel getModel(OntClass c, String individualURI)`

Versucht diese Kontextinformation (mit Hilfe der Methode `asOntClassURIs()`) als eine Instanz, die URI `individualURI` hat und der basisontologischen Klasse `c` angehört, darzustellen. Wenn es gelingt, wird eine A-Box zurückgegeben, die diese Instanz enthält.

Es gilt also, dass jede Klasse, die Kontextinformationen ontologisch repräsentieren können sollte, diese zwei Methoden überschreiben muss. Die zweite Methode basiert auf der ersten und ist im Übrigen mit Hilfe von Jena mit wenigen Befehlen schnell und simpel realisierbar. Die Implementierung der ersten Methode erfolgt, um ein Beispiel zu geben, bei der Klasse `WeatherSourceValue` wie folgt.

Die ontologischen Klassen `Medium`, `Cold` und `Hot` beziehen sich auf die Lufttemperatur. So wird für die Zuordnung eines Wertes für die Lufttemperatur zu einer dieser drei Klassen folgende Regel verwendet:

$$\texttt{Cold} < 10^\circ \leq \texttt{Medium} \leq 25^\circ < \texttt{Hot}$$

Um die Entsprechung einer Instanz von `WeatherSourceValue` einer oder mehreren ontologischen Klassen aus `{Rainy,Sunny,Cloudy,Snowy,Windy}` festzustellen, muss die textuelle Beschreibung des Wetterzustandes analysiert werden: es wird nach Stichworten wie „Regen“, „regnerisch“, „Sonne“, „sonnig“ usw. gesucht. Es wäre an dieser Stelle wünschenswert, dass das Tinfo Web Service Beschreibungen in einem vordefinierten Format angeben würde.

5.3 Erweiterbarkeit

„Erweiterbarkeit“ wird hier als eine Möglichkeit verstanden, einem Context-Aware System Kontextquellen bzw. Sensoren hinzuzufügen. Dies ist im Generellen wichtig,

wenn ein abstraktes Framework entwickelt werden soll, das in einer konkreten Anwendung eingesetzt werden könnte. Aber auch für SmartWeb spielt die Erweiterbarkeit aus folgenden zwei Gründen eine bedeutende Rolle.

Erstens, gibt es seitens SmartWeb keine präzise Angaben, welche Kontextquellen unbedingt zu implementieren sind. Vielmehr wurde dies in der Entwicklungsphase herausgearbeitet und, um zusätzliche Funktionalität zu erlangen, wiederholt ergänzt. So wurden die fünf vorgestellten Kontextquellen im Rahmen dieser Arbeit in einer relativ frühen Phase implementiert; im Laufe der Zeit wurden dann von anderen Mitarbeitern im SmartWeb Projekt weitere Kontextquellen implementiert, die beispielsweise Zustand der befahrenen Strasse, Geschwindigkeit des Fahrzeugs und Luftdruck seiner Reifen sowie andere Informationen liefern.

Zweitens, finden die in SmartWeb erzielten Ergebnisse, oder genauer gesagt, die in diesem Kapitel vorgestellte Infrastruktur für das Context-Aware System für SmartWeb, im Informationsportal „Heidelberg mobil“[50] ihre Verwendung. Dieses bietet Fußgängern mobilen Zugriff auf diverse Informationen zur Stadt Heidelberg an, und zwar kontextabhängig, also angepasst an die Situation, in der sich die Fußgänger gerade befinden.

5.3.1 Sensoren

Da Sensoren nicht als selbständige Komponenten der Infrastruktur implementiert sind (siehe 5.1.1), gestaltet sich die Erweiterung des Systems um weitere Sensoren sehr einfach: jeder, der entsprechende XML-Nachrichten an Kontextinterface schicken vermag, präsentiert sich ihm gegenüber als ein Sensor, ohne sich auf irgendeine Weise identifizieren oder über weitere Funktionalität verfügen zu müssen.

5.3.2 Kontextquellen

Um eine neue Kontextquelle zu implementieren wird eine von `ContextSource` abgeleitete Klasse deklariert; sie soll die in 5.1.2 genannten Methoden überschreiben. Gegebenenfalls kann auch eine Klasse implementiert werden, welche von der neuen Kontextquelle ermittelnde Kontextinformationen repräsentiert. Diese Klasse soll von

[50] www.heidelberg-mobil.de

ContextSourceValueInterface abgeleitet werden und die in 5.2.1 (und gegebenenfalls in 5.2.2) genannten Methoden überschreiben. Schließlich muss die neue Kontextquelle in der Konfigurationsdatei eingetragen werden. Dies geschieht auf dieselbe Weise wie es folgend am Beispiel der Kontextquelle „Wetter“ gezeigt ist:

```
<source>
    <type>weather</type>
    <alias>
        http://smartweb.semanticweb.org/ontology/navigation#OutsideTemperature
        Region
    </alias>
    <alias>
        http://smartweb.semanticweb.org/ontology/navigation#WeatherRegion
    </alias>
    <class>
        de.emld.context.sources.WeatherSource
    </class>
    <thresholds>
        <timeThreshold>1000000</timeThreshold>
        <valueThreshold>1</valueThreshold>
    </thresholds>
</source>
```

Zwingend erforderlich ist das Tag class, in dem Klassenpfad der Kontextquelle angegeben ist, sowie das Tag type, der einen Namen der Kontextquelle vereinbart. Soll ein Bezug zu Basisontologie hergestellt werden, gibt man in Tags alias die Namen von basisontologischen Klassen an. Fehlender Tag für die Zeitschranke wird mit dem Zeitschrankenwert −1 gleichgesetzt, also mit keinem Gültigkeitsverlust einer Kontextinformation dieser Kontextquelle nach einer gewissen Zeit; das Fehlen der Betragsschranke impliziert die exakte Übereinstimung, also den Wert 0 (siehe 5.1.3.1).

5.4 Inbetriebnahme

Um mit einem Benutzer des SmartWeb Dialogsystems eine Instanz von ContextInterface zu verbinden, über die Kontextinformationen bezüglich dieses Benutzers ermittelt werden können, verfügt die Klasse ContextInterface über folgende zwei Konstruktoren:

`ContextInterface(boolean init, InputStream config)`

Erzeugt eine Instanz von `ContextInterface`. Wenn `init = true`, wird Kontextinterface initialisiert, d.h. die als ein Stream repräsentierte Konfigurationsdatei wird ausgelesen, wodurch Kontextquellen (bzw. Instanzen entsprechender Klassen) erzeugt werden. Als Benutzerprofil wird eine leere Instanz von `UserIDSourceValue` erzeugt.

`ContextInterface(boolean init, String configFileName)`

Hat dieselbe Funktionsweise wie der vorige Konstruktor, nur wird die Konfiguration aus Konfigurationsdatei mit dem Namen `configFileName` ausgelesen.

Zur Erzeugung und Verwendung von Instanzen der Klasse `ContextInterface` ist folgendes zu bemerken:

- Das nach der Erzeugung leere Benutzerprofil wird aktualisiert, indem die Kontextinterface betreibende Anwendung (über die „`update`"-Methode von `ContextInterface`, siehe 5.1.4) eine das Benutzerprofil beinhaltende Nachricht schickt. Alternativ, vor allem für Testzwecke, kann einer speziellen Variable von `UserIDSourceValue` eine ganze Zahl zugewiesen werden, anhand derer dieses Benutzerprofil identifizierbar wird. Diese Variable heißt `testValueOnly`[51].
- Kontextquellen „`Ort`", „`Wetter`" und „`Pollen`" benötigen GPS Koordinaten. Bevor diese per Update an das Kontextinterface übermittelt werden, können diese Kontextquellen keine Kontextinformationen ermitteln.
- Dieselben Kontextquellen greifen auf das Web Service Tinfo zu. Dafür werden Zugriffsdaten benötigt, die in der Konfigurationsdatei wie folgt anzugeben sind:

```
<web_services_access>
    <tinfo>
        <login>...</login>
        <password>...</password>
    </tinfo>
</web_services_access>
```

[51] Der eigentliche Typ von `testValueOnly` ist `String`. Diese Variable darf jedoch als Wert nur die als `String` repräsentierten ganzen Zahlen haben.

6 Entwurf ontologiebasierter Kontextrepräsentation und Kontextverarbeitung

In diesem Kapitel wird der im Rahmen dieser Arbeit entwickelte Ansatz zur ontologiebasierten Kontextrepräsentation vorgestellt. Dies beinhaltet auch die erst durch diese Repräsentation ermöglichte Verarbeitung von Kontextinformationen, also gewisse Dienste, die zum Beispiel Kontextinformationen aus der ontologischen Repräsentation inferieren.

Das Beispiel, das im Kapitel 3 vorgestellt wurde, begleitet das sechste Kapitel durchgehend: es wird an vielen Stellen aufgegriffen, erweitert und tiefer ins Detail gehend präsentiert. In allen Einzelheiten wird es erst in 7.2 und 7.3 herausgearbeitet. Für einen beispielorientierten Leser ist es deshalb zu empfehlen, diese Unterkapitel vor oder während des Lesens von dem Unterkapitel 6.2 wahrzunehmen.

6.1 DnS-Framework in Praxis

Der bisherigen Betrachtung des DnS-Frameworks unterlagen ausschließlich die oben genannten Publikationen. Nun wird seine Tauglichkeit aus Sicht eines potentiellen Anwenders untersucht.

Hierbei ist zunächst einmal zu erwähnen, dass in den besagten Publikationen Ideen, die das DnS-Framework einbringt, formal mittels der Prädikatenlogik festgehalten werden. Jedoch werden sie dort auf diese Weise nicht vollständig spezifiziert, sondern eher zwecks der Anschaulichkeit niedergelegt. Für fehlende Details wird auf das OWL-Code verwiesen, in dem die DnS-Ontologie formal dargelegt ist. Doch weil die Übertragung aus einem Formalismus in das andere die syntaktischen Strukturen stark verändert, ist es sehr schwierig und zeitaufwendig sich an denjenigen Stellen einen genaueren Überblick zu verschaffen, an denen die prädikatenlogische Darbietung wegen ihres überblickenden Charakters unvollständig ist.

Weiterhin stellt man bei genauer Betrachtung des OWL-Codes fest, dass man bei reiner textueller Suche die Präsenz von Relationen `P-Sat`, `R-Sat`, `C-Sat` nur an denjenigen Stellen findet, an denen sie oder ihre jeweilige Inversen definiert sind. Dabei wird über sie nur ausgesagt, dass sie Unterrelationen von `Satisfies` sind und als `Domain`

`Situation` bzw. als `Range Description` haben. Entsprechende Aussagen findet man über die jeweiligen Inversen vor. Es ist offensichtlich, dass derartige Angaben, die diese Relationen nicht einmal (außer ihrer Benennung) voneinander unterscheiden, mangelhaft sind, um sie gemäß ihrer jeweiligen Definition in der Prädikatenlogik zu spezifizieren.

Es bleibt zwar alternativ die `Satisfies` Relation, über deren Auftreten in OWL-Code es aus oben genanntem Grund schwierig zu beurteilen ist, ob es die prädikatenlogischen Beschreibung erfüllt. Jedoch selbst wenn es der Fall wäre, ist die Anforderung, die `Satisfies` an eine Situation stellt, damit sie einer Deskription genügt, einfach zu gering um für das in dieser Arbeit behandeltes Anwendungsszenario ausreichend zu sein – dies wird im Laufe dieses Kapitels deutlich. Außerdem ist es leider bei einem Testversuch misslungen die beschriebene Verwendung der DnS-Ontologie und insbesondere der `Satisfies` Relation an einem einfachen Beispiel zu reproduzieren. Es wurde nämlich eine Deskription, die der in der Abbildung 4.4 sehr ähnlich ist, sowie eine Situation ohne Settings modelliert. Über die Situation wurde eine Behauptung aufgestellt, dass sie der Deskription genügt. Trotz der Erwartung konnte dabei keine Inkonsistenz nachgewiesen werden.

Möglicherweise lag dieser Fehlschlag an einer unkonformen Modellierung der Deskription bzw. Situation, denn die Entwickler von DnS-Framework selbst bei der Gestaltung des DnS-Design-Patterns bedauerlicherweise nicht konsequent bleiben. So hat es je nach Publikation (z.B. [3] vs. [4]) abweichende Struktur, der teils divergierendes System von Begriffen unterliegt. Dies deutet offensichtlich darauf hin, dass die Entwicklung von DnS-Framework ihren Abschlusspunkt noch nicht erreicht hat.

Somit haben sich genügend starke und zahlreiche Argumente gesammelt, die gegen einen eventuell überlegenswerten Versuch sprachen, die gerade beschriebenen Mängel durch einige Korrekturen und Veränderungen in der OWL Spezifikation der DnS-Ontologie zu beheben. Diese Argumente haben sich durch die Befürchtung, auf weitere, noch unentdeckte Mängel zu stoßen, bekräftigt und konnten leider nicht durch einige in der Literatur [11, 12, 13, 14, 15] beschriebene Fälle überwogen werden, in denen das DnS-Framework zum Einsatz kam, denn dortigen Beispielen war nicht ersichtlich, ob

und wie sie tatsächlich implementiert wurden bzw. ob sie (was das DnS-Framework angeht) bloß als Entwürfe auf dem Papier zutage treten.

6.2 Von DnS-Framework zum eigenen Entwurf

Die in 6.1 geschilderten Gründe führen zum Entschluss, das DnS-Framework, entgegen der ursprünglichen Absicht, nur teilweise zu integrieren. So wurde die Idee übernommen, das pragmatische Wissen mittels deskriptiver Komponenten des DnS-Design-Patterns, also in Form von Deskriptionen zu modellieren. Doch entgegen dem ursprünglichen Gedanken aus einer basisontologischer A-Box, das konkrete Gegebenheiten in der Welt (bzw. in SmartWeb die Benutzeraussagen) repräsentiert, Situationen zu kreieren und dann die vorgeschlagenen Erfüllungsrelationen darauf anzuwenden, wird in nächsten Abschnitten dieses Kapitels eigener Ansatz vorgestellt, der unter anderem Deskriptionen zu bestimmen ermöglicht, denen eine A-Box genügt. Somit ist es sehr fraglich, ob es bei der resultierender Symbiose aus DnS-Framework und dem eigenen Ansatz behaupten ließe, diese Arbeit verwende das DnS-Framework. Allerdings bedeutet das eine gewisse Umstellung für das SmartWeb Projekt bzw. dessen betroffene Teilgebiete, die sich mit dem Kontext beschäftigen und für die der Einsatz des DnS-Frameworks vorgesehen war. Diese Umstellung ist auch mit einer Zeitverzögerung verbunden. Dies sind mitunter die Gründe, aus denen die in diesem und in Kapiteln 7, 8 geschilderte und implementierte Ideen in SmartWeb keine Verwendung gefunden haben.

Nach dem Verzicht auf die Erfüllungsrelationen der DnS-Ontologie, muss selbstverständlich an ihre Stelle ein entsprechender Ersatz treten. Dafür kommt erst einmal ein logikbasierter Ansatz in Frage, d.h. statt der DnS-Ontologie eine eigene zu kreieren. Doch in Hinblick auf die Komplexität, Vielseitigkeit und das enorme wissenschaftliche Rückgrat, das in die DnS-Ontologie eingebracht wurde, wäre die eigene, die im Rahmen dieser Arbeit hätte entstehen können, wohl eher bescheiden und auf den Fall zugeschnitten geworden. Außerdem ist es fraglich, inwiefern die im Kapitel 3 vorgestellte Aufgabenstellung mit Hilfe eines beschreibungslogikbasierten Ansatzes gelöst werden kann. Alsbald jedoch von diesem Ansatz abstrahiert wird, hebt sich die Sichtweise auf, die Deskriptionen samt sie definierendem Begriffssystem als eine, im Sinne der mathematischen Logik, Theorie zu betrachten und die Situationen – als eventuelle Modelle dieser Theorie. Stattdessen wird auf Graphenstruktur einer

jeweiligen Deskriptionen zurückgegriffen und versucht, zwischen ihr und der gegebenen Situation eine Übereinstimmung (ein Matching) festzustellen. Ob die einzelnen Entitäten der Situation mit entsprechenden Elementen der Deskription übereinstimmen – also die Frage nach hierarchischen Beziehungen zwischen Rollen bzw. Klassen sowie die Klassenzugehörigkeit eines Individuums – wird nach wie vor aus entsprechenden Ontologien inferiert.

6.3 Festlegung des Begriffssystems

Nun, nachdem motivierende Argumente für die alternative Handhabung von Deskriptionen und Situationen eingebracht wurden und bevor die algorithmische Lösung des Problems vorgestellt werden kann, bedarf es mehrerer Zwischenschritte. Zuerst scheint die Festlegung eines Begriffsystems notwendig. Jedoch kann dieser Abschnitt vorläufig übersprungen und erst bei erstmaligem Auftreten eines neuen Begriffs nachgeschlagen werden, denn der Inhalt einiger Begriffe an dieser Stelle möglicherweise nicht im vollen Maße nachvollziehbar wird.

Mit *Deskription* bezeichne man entweder die Klasse `Description` bzw. eine von ihr abgeleitete Subklasse oder eine Instanz einer von diesen Klassen.[52]

Mit *Situation* bezeichne man, im Gegensatz zu DnS-Framework, eine beliebige Menge ontologischer Entitäten. Üblicherweise ist es eine A-Box, welche die Benutzeraussage repräsentiert.

Eine zwischen zwei Entitäten einer Ontologie herrschende Klassenzugehörigkeits- oder Ober- bzw. Unterklassenbeziehung oder Ober- bzw. Unterrelationsbeziehung sei genau dann *explizit*, wenn sie aus der Ontologie ohne zu inferieren folgt, also ein Element der A-Box dieser Ontologie ist. Die Motivation für die Einführung dieses Begriffs ist am Beispiel in der Abbildung 6.1 gezeigt. In manchen Situationen ist es nämlich wichtig von allen Instanzen von `Road` nur diejenige in Betracht zu ziehen, über die keine spezifischere Einordnung in eine bestimmte Gattung von Strassen bekannt ist. So sind

[52] Hier und im Weiteren sind einige Begriffe als Klassen oder Instanzen definiert. Es wird davon ausgegangen, dass es bei jedem Vorkommnis eines solchen Begriffs aus dem Kontext heraus nachvollziehbar sein wird, ob es sich jeweils um eine Klasse oder eine Instanz handelt. In sonstigen Fällen wird Genaueres angegeben.

beide Individuen in der Abbildung 6.1 Instanzen von `Road`, jedoch nur `Some Road` ist es explizit.

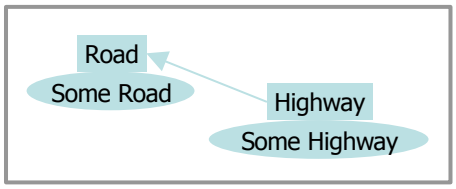


Abbildung 6.1: Explizite Klassenzugehörigkeit. (*)

Mit *Partizipierungsrelation* (*PR*) bezeichne man jede Unterrelation von `ValuedBy`, `PlayedBy` oder `Sequences`, sowie auch diese drei Relationen. Mit PR^{-1} bezeichne man jede Relation, die invers zu einer Partizipierungsrelation ist.

Mit *Deskriptionskomponente* (*D-Komponente*) einer Deskription bezeichne man entweder jedes Individuum, das (in Begriffen von RDF(S)) Objekt einer Aussage ist, die als Subjekt diese Deskription und als Prädikat `Defines` hat; oder eine explizite Klasse eines solchen Individuums. Beispielsweise hat die in der Abbildung 4.3 dargestellte Deskription folgende drei D-Kopmonenten: `New Parameter`, `New Role` und `New Course`.

Mit *Deskriptionsrelation* (*DR*) bezeichne man jede Unterrelation von `RequisiteFor`, oder `AttitudeTowards`, sowie auch diese beide Relationen. Mit DR^{-1} bezeichne man jede Relation, die invers zu einer Deskriptionsrelation ist.

Mit *Situationsrelation* (*SR*) bezeichne man jede Unterrelation von `LocationOf` (Inverse zu `LocatedIn`), oder `ParticipantIn` (Inverse zu `Participant`), sowie auch diese beide Relationen. Mit SR^{-1} bezeichne man jede Relation, die invers zu einer Situationssrelation ist.

Mit *PR-Target* einer D-Komponente bezeichne man entweder jedes Individuum, das Objekt einer Aussage ist, die als Prädikat eine Partizipierungssrelation hat; oder eine explizite Klasse eines solchen Individuums. Beispielsweise hat die D-Komponente `New Parameter` aus der Abbildung 4.3 nur ein PR-Target: `New Region`. Eine ontologische Entität, die PR-Target einer D-Komponente, oder Unterklasse eines solchen PR-Targets

oder eine Instanz solcher Unterklasse ist, *parametrisiert* sie. Eine Menge ontologischer Entitäten `M` (z.B. eine A-Box) parametrisiert eine D-Komponente, wenn in `M` ein Element existiert, das dies tut.

Mit *Deskriptionspartizipant* (*D-Partizipant*) einer Deskription bezeichne man jedes PR-Target jeder D-Komponente dieser Deskription.

Für folgende Definition sei es angenommen, dass jede Kontextquelle auf irgendeine, an dieser Stelle nicht spezifizierte Weise mit mindestens einer Klasse der Basisontologie assoziiert ist. Dies impliziert die Tatsache, dass die jeweils aktuellen Instanzen einiger basisontologischer Klassen von einigen Kontextquellen ermittelt werden können. Zum Beispiel kann eine aktuelle Instanz der basisontologischen Klasse `time-point`, also der gegenwärtige Zeitpunkt, mittels Systemuhr bestimmt werden. Dann, gefolgt diesem Gedanken, sei eine ontologische Klasse `K` *von einer Kontextquelle* `Q` *determinierbar* genau dann, wenn `Q` mit `K` oder einer Oberklasse von `K` (per expliziter Eingabe) assoziiert ist. `K` sei *determinierbar* genau dann, wenn `K` von irgendeiner Kontextquelle determinierbar ist. Ein Individuum sei genau dann determinierbar, wenn es zu einer determinierbaren Klasse gehört.

Die nächste Überlegung bezweckt die Idee zu formalisieren, dass eine A-Box `A` nur dann mit einer Deskription `D` (im Sinne von Matching) übereinstimmt, wenn diejenige Individuen in `A`, die D-Komponenten von `D` parametrisieren, über die Situationsrelationen derartig zueinander in Bezug stehen, wie es von `D` über die Deskriptionsrelationen vorgegeben ist. Diese Idee wird nun durch einen Graphenisomorphismus untermauert. Zuerst werden zwei RDF-Graphe wie folgt konstruiert:

- Der erste Graph ergibt sich aus denjenigen D-Komponenten von `D`, die von `A` parametrisiert sind, und aus Deskriptionsrelationen zwischen ihnen. Dabei wird für jede D-Komponente[53] nur die Zugehörigkeit zu Klassen aus `{Parameter, Role, Course}` in den Graph übernommen. Genauso statt der eigentlichen Deskriptionsrelationen zwischen den besagten D-Komponenten werden ihre

[53] Hier sind mit D-Komponenten Individuen gemeint.

jeweilige Oberrelationen[54] aus {RequisiteFor, AttitudeTowards} in den Graph übernommen.

- Der zweite Graph ergibt sich aus Individuen aus A, die eine D-Komponente von D parametrisieren, und aus den Situationsrelationen zwischen diesen Individuen. Dabei wird für jedes Individuum nur die Klassenzugehörigkeit zu Klassen aus {Region, Endurant, Perdurant} in den Graph übernommen. Entsprechend statt der eigentlichen Situationsrelationen zwischen den besagten Individuen werden ihre jeweilige Oberrelationen[55] aus {LocationOf, ParticipantIn} in den Graph übernommen.

Nun sei A bezüglich D genau dann *DR-Konsistent*, wenn der zweite Graph einen zu dem ersten Graph isomorphen Teilgraph besitzt.

Die folgenden zwei Begriffe unterscheiden zwei Arten von Matching zwischen einer A-Box und einer Deskription.

Eine A-Box A *erfüllt* eine Deskription D genau dann, wenn A alle D-Komponenten von D parametrisiert und A DR-Konsistent bezüglich D ist.

Eine A-Box A *selektiert* eine Deskription D genau, dann wenn folgende Bedingungen erfüllt sind:

1. A parametrisiert mindestens eine D-Komponente von D
2. Für jede von A nicht parametrisierte D-Komponente von D gilt: sie hat mindestens ein determinierbares PR-Target.
3. A ist DR-Konsistent bezüglich D.

Ein Beispiel für eine erfüllte Deskription ist in der Abbildung 4.4 gegeben: in der Tat erfüllt die dort abgebildete A-Box die Environment-Path-Description. Hätte jedoch in der A-Box Sunny gefehlt, würde sie diese Deskription nicht erfüllen, jedoch selektieren, vorausgesetzt, WeatherRegion sei determinierbar.

Diese zwei Definitionen trennen Deskriptionen (respektive einer A-Box) nicht in zwei disjunkte Mengen: selektierte vs. erfüllte Deskriptionen. Vielmehr implizieren die

[54] Hier ist zu beachten, dass eine Relation eine Oberrelation für sich selbst ist.
[55] Hier ist zu beachten, dass eine Relation eine Oberrelation für sich selbst ist.

Letzteren die Ersten. Die erwünschte Art von Matching soll sich optional angeben lassen. Genauer ausgedrückt, es soll möglich sein, die selektierten Deskriptionen als gematchte gelten zu lassen. Daher seien (respektiv einer A-Box) *relevante* Deskriptionen genau diejenige, die erfüllt sind und, falls entsprechende Option gewählt ist, auch diejenige, die selektiert sind.

6.4 Modellierungsaspekte der Ontologie von Deskriptionen

In diesem Unterkapitel werden einige Modellierungsaspekte herausgearbeitet, auf die man bei tiefer gehenden Beschäftigung mit der Modellierung von Deskription nahezu zwangsläufig stoßt. Es wird sich zunächst um die Möglichkeit, optionale D-Komponenten bzw. D-Partizipanten für eine Deskription anzugeben, handeln und dann um die Idee der Widersprüchlichkeit zweier Deskriptionen zueinander.

6.4.1 Behandlung mehrerer D-Komponenten und D-Partizipanten, und Diskussion ihrer Optionalität

Laut dem DnS-Design-Pattern darf sowohl eine Deskription beliebig viele D-Komponente als auch eine D-Komponente beliebig viele PR-Targets einschließen. Doch demselbigen ist es nicht ersichtlich, wie diese beide Optionen gehandhabt werden. In dieser Arbeit wurde bisher die Behandlung beider Optionen indirekt bei der Definition von erfüllter bzw. selektierter Deskriptionen spezifiziert. An dieser Stelle wird es nun explizit an einem Beispiel veranschaulicht.

Für eine D-Komponente gilt, dass sie von einer A-Box genau dann parametrisiert ist, wenn zumindest eine ihrer PR-Targets in der A-Box vorkommt. Dies bedeutet, dass man auf eine einfache und intuitive Weise mehrere Alternativen eines D-Partizipanten für eine D-Komponente angeben kann, und zwar nach dem Motto „alles kann, nichts muss“. So sieht man in der Abbildung 6.2 eine Deskription, die einen bei gutem Wetter stattfindenden Spaziergang oder eine Spazierradfahrt in bzw. zu der Stadtmitte, Altstadt oder dem Park modelliert (die Angabe von `owl:Thing` als Wertebereich für `NiceLocationEnvironment` sei zunächst einmal ungeachtet). Somit drückt diese Deskription aus, dass in einer Situation mindestens eine Standortangabe vorhanden (oder determinierbar) sein muss, damit sie (die Deskription) aktiviert wird. Der Standort muss also nicht gleichzeitig Stadtmitte, Altstadt und Park sein, aber er kann es. Eine

derartige Interpretation mehrerer PR-Targets einer D-Komponente sei als *disjunktiv* bezeichnet. Möchte man jedoch die *konjunktive* Sichtweise ins Spiel bringen, d.h. einige PR-Targets einer D-Komponente als notwendig voraussetzen, so bedarf es mehrerer Individuen dieser D-Komponente, die auf dieselbe Art parametrisiert werden, wie es in der Abbildung 6.3 gezeigt ist. Diese stellt ein Beispiel dar, in dem als Standort ein Park in der Stadtmitte oder Altstadt gefordert wird. Dies bedeutet also, dass Park konjunktiv mit der Disjunktion der Stadtmitte und Altstadt verbunden ist.

An dieser Stelle sei eine Bemerkung erlaubt: eine disjunktive Interpretation mehrerer PR-Targets einer D-Komponente ist äquivalent zu mehrfacher expliziter Klassenzugehörigkeit eines D-Partizipanten. In der Tat, man stelle sich vor, `NiceLocationEnvironment` hätte als PR-Target ein Individuum, welches in expliziter Weise eine Instanz von sowohl `OldTown` als auch von `City-Center` und von `Park` wäre. Dies würde gemäß dem festgelegten Begriffssystem bedeuten, dass alle drei Klassen auch PR-Targets von `NiceLocationEnvironment` wären. Folglich wäre diese D-Komponente durch jedes Individuum der A-Box parametrisiert, das Instanz einer dieser drei Klassen ist. Diese redundante Äquivalenz wird zwecks Vereinfachung der Modellierung und Verarbeitung von Deskriptionen aufgehoben, indem die mehrfache explizite Klassenzugehörigkeit eines D-Partizipanten verboten wird.

Eine Deskription kann mehrere D-Komponente definieren, und zwar unabhängig davon, ob es Parameter, Rollen oder Abläufe sind. Damit eine Deskription erfüllt ist, müssen alle ihre D-Komponenten parametrisiert sein, d.h. sie unterliegen der konjunktiven Interpretation. So wird beispielsweise in der Abbildung 6.2 ein schönes Wetter durch die Himmelbedeckung und Lufttemperatur bestimmt. Das disjunktive Verhalten an dieser Stelle würde bedeuten, einige D-Komponenten als optional festzulegen. Ein solches Verhalten wird durch einen Trick erreicht, der in der Abbildung 6.2 bei `NiceLocationEnvironment` angewandt ist. Er besteht darin, dass als PR-Target einer optional intendierter D-Komponente ein Individuum angegeben wird, das explizit eine Instanz der Wurzelklasse `owl:Thing` ist. Dies hat zur Folge, dass jede nicht leere A-Box diese D-Komponente parametrisiert, wodurch keine Deskription für eine A-Box als irrelevant befunden wird, nur weil diese D-Komponente dort „fehlt“. So wird auch `NiceLocationEnvironment` stets parametrisiert. Ob es sich dabei um eine sozusagen echte Parametrisierung handelt, d.h., ob in der A-Box tatsächlich eine Instanz von

`OldTown` oder `City-Center` oder `Park` vorkommt, lässt sich bei Bedarf einfach feststellen: eine Entität parametrisiert eine D-Komponente echt genau dann, wenn sie expliziterweise keine Instanz von `owl:Thing` ist.

Scheinbar produziert der gerade beschriebene Trick eine Inkonsistenz, weil als D-Partizipant `owl:Thing` auftritt, wobei das Wertebereich der Partizipierungsrelationen diese Klasse nicht beinhaltet. Doch es ist nicht der Fall, da gemäß der Semantik von OWL ein als Objekt einer Relation auftretendes Individuum als zum Wertebereich dieser Relation angehörig inferiert wird.

Ein Nachteil bei dem oben genannten Trick könnte eventuell darin liegen, dass über derartige Vorgehensweise in dieser Arbeit nicht mit Sicherheit ausgesagt werden kann, ob sie mit dem DnS-Framework konform ist. Es ist also schwer abschätzbar, was passiert, wenn die geschilderte Vorgehensweise in DnS-Framework angewandt wird. Deshalb ist es möglich alternativ vorzugehen und optionale D-Komponente durch zwei Deskriptionen zu modellieren, die bis auf die Anwesenheit dieser D-Komponente identisch sind. Doch würde diese Lösung die Ontologie von Deskriptionen in einem unvertretbar hohem Maße vergrößern und eignet sich somit nur für Einzelfälle.

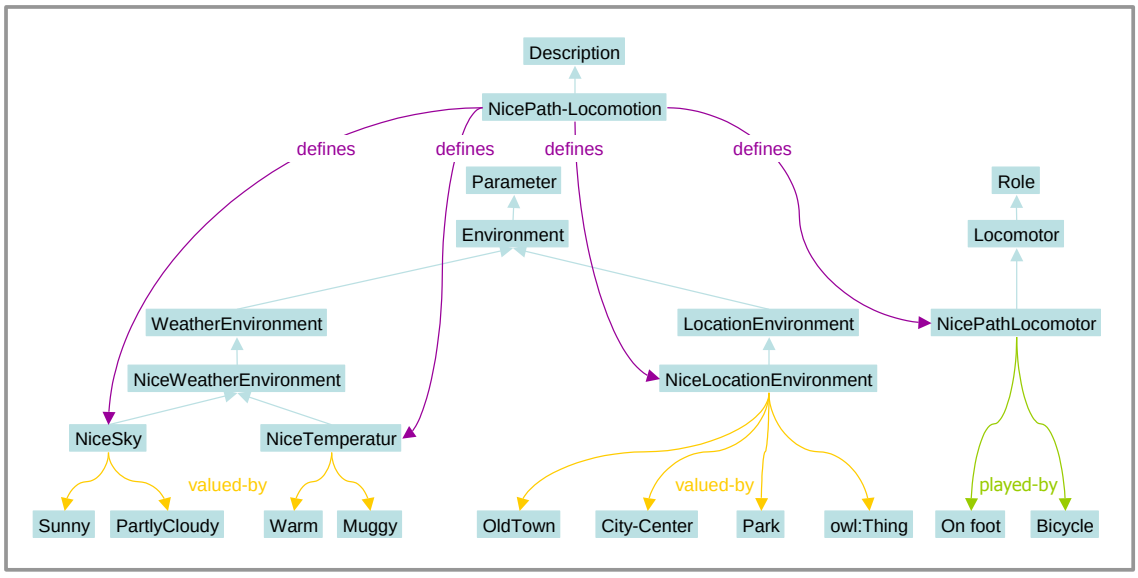


Abbildung 6.2: Die `NicePath-Locomotion` Deskription. D-Partizipant `owl:Thing` bedeutet hier, dass es eine Instanz von `NiceLocationEnvironment` gibt, die über `valued-by` mit einer Instanz von `owl:Thing` verbunden ist. Dies bedeutet nicht, dass `valued-by` als Wertebereich `owl:Thing` hat. (*)

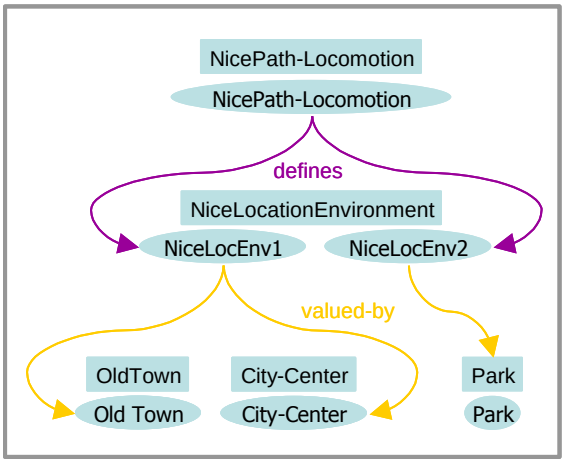


Abbildung 6.3: Konjunktive vs. disjunktive Interpretation der PR-Targets von `NicePath-Locomotion`. (*)

6.4.2 Widersprüchlichkeit von Deskriptionen

Es kann vorkommen, dass durch zwei Deskriptionen modellierte Begriffe einander ausschließen. Zum Beispiel, eine Fortbewegung bei schönem, sonnigem Wetter ist etwas ganz Anderes als wenn es regnerisch ist, insbesondere, wenn es einen Fußgänger betrifft. Es kann jedoch, zumindest theoretisch, passieren, dass eine Kontextquelle irrtümlich gute Wetterverhältnisse meldet, während eine andere, die sich möglicherweise auf einen verlässlicheren Wetterdienst stützt, korrekterweise Regen angibt. Dadurch bedingt, würden sich bei dieser Situation eventuell zwei Deskriptionen als relevant erweisen, die einander hätten ausschließen sollen. Somit resultiert sich die Notwendigkeit, je zwei Deskriptionen als einander widersprüchlich markieren zu können. Dies erfolgt, indem die betroffene Deskriptionen durch das OWL Prädikat `disjointWith` als solche markiert werden. Nachteilig wirkt dabei die Tatsache, dass dieses Prädikat ein Element von OWL DL und nicht von OWL Lite ist. So, wenn das OWL DL Reasoning in einer Anwendung unmöglich ist, bietet sich an, entweder auf Widersprüchlichkeit völlig zu verzichten oder sie auf einem nicht ontologischem Wege zu behandeln.

Im Allgemeinen kann eine Situation eine Menge von Deskriptionen erfüllen. Es gibt zwei Möglichkeiten festzustellen, ob eine Menge zwei disjunkte Deskriptionen

beinhaltet. Die naheliegendste ist, alle Deskriptionen paarweise zu überprüfen. Eine möglicherweise geschicktere Option besteht darin, der Ontologie von Deskriptionen temporär ein neues Individuum hinzuzufügen, das Instanz aller betroffener Deskriptionen ist. Falls der mit der Deskriptionsontologie verknüpfte Reasoner dabei keine Inkonsistenz meldet, ist die Menge von Deskriptionen widerspruchsfrei. Die letztere Variante ist nur dann zufriedenstellend, wenn es akzeptierbar ist, dass bei der Aufdeckung einer Widersprüchlichkeit die Verarbeitung sofort abbricht und die eingegebene A-Box unverändert ausgegeben wird. Es ist jedoch eine intelligentere Widersprüchlichkeitsbehandlung denkbar. So können z.B. zwei relevante Deskriptionen, die zueinander disjunkt sind, aus weiterer Verarbeitung ausgeschlossen werden. Oder es können Kriterien aufgestellt werden, nach denen eine von beiden disjunkten Deskriptionen sich bei den gerade vorliegenden Umständen vertrauenswürdiger erweist. Dabei kommt vor allem die Zuverlässigkeit von Kontextquellen bzw. einzelner Informationen, die sie ermitteln, in Frage. Es kann aber auch sinnvoll sein, die Qualität der Matching der Situationen mit den jeweiligen Deskriptionen als mögliches Kriterium zu untersuchen, insbesondere in Hinblick darauf, ob, welche und wie viele optionale D-Komponenten und D-Partizipante einer Deskription in der A-Box vorkommen bzw. wie viele und welche nicht vorkommende optionale D-Partizipante determinierbar sind.

Abschließend sei noch gesagt, dass dieser Abschnitt nur dem Zwecke dient, auf die hier diskutierten Angelegenheiten hinzuweisen. Ob und wie diese verwendet werden – dies hängt gänzlich von jeder konkreter Anwendung ab, unter anderen von ihrer Zielsetzung und von der Beschaffung dort eingesetzter Ontologien.

6.5 Spezialisierung von Deskriptionen

Um die breite Palette von Diensten zu überdecken, um die sich die Deskriptionsmodellierung bemüht, reicht es nicht die für eine Situation passende Deskriptionen zu finden. Man möchte die in einer Situation präsente Angaben präzisieren, wie es das Beispiel aus der Abbildung 3.1 zeigt, in dem der allgemeine Straßentyp durch einen Unterbegriff, die Autobahn, ersetzt wurde. Für diese Zwecke geht man von in der Deskriptionsontologie hierarchisch angeordneten Deskriptionen aus, wie es unten und in 7.2 demonstriert wird. An dieser Stelle ist es zunächst wichtig,

dass eine Hierarchie von Deskriptionen es ermöglicht, die für eine Situation relevante Deskription zu spezialisieren, um dadurch präzisere Informationen zu gewinnen. Im Folgenden werden nun tiefgründige Überlegungen geschildert, wie und womit eine Deskription spezialisiert werden kann. Zuerst werden zwei Klassifizierungen für solche Fälle vorgestellt, bei denen man von einem Objekt behaupten kann, es würde eine vorliegende Deskription spezialisieren. Dann werden einige wichtige Begriffe eingeführt. Danach, basierend auf den Klassifikationen, werden die Definitionen für die Spezialisierung von Deskriptionen folgen.

6.5.1 Klassifizierung nach Hierarchien

Wie bereits gesehen „verbergen sich“ in einer Ontologie der Deskriptionen die eigentlichen kontextuellen Informationen, die als implizit in einer Benutzeraussage vorhanden gelten, in D-Partizipanten der jeweils relevanten Deskriptionen. D-Komponenten können als Struktur von Deskriptionen prägende Elemente betrachtet werden, welche diejenige D-Partizipanten zusammenhalten, die die jeweiligen Deskriptionen ausmachen sollen. Die den Gegenstand einer Aussage encodierende Kontextinformation sind die Deskriptionen selbst, also die von `Description` abgeleitete Klassen und ihre Instanzen (so z.B. wenn für eine Aussage die Deskription `Locomotion` als relevant befunden wurde, bildet offensichtlich die Fortbewegung den Gegenstand dieser Aussage). Die hierarchische Anordnung von Deskriptionen bedeutet, dass es sowohl eine Hierarchie von diesen, von `Description` abgeleiteten Klassen, als auch die von D-Komponenten und D-Partizipanten (bzw. Unterklassen von D-Partizipanten) gibt. Weil es im Allgemeinen keine Begründung gibt, diese drei Hierarchien als isomorph vorauszusetzen, sind sie alle für die Klärung des Begriffs Spezialisierung von Deskriptionen unerlässlich. Dies bedeutet, dass wenn man Forderungen an ein Objekt aufstellen möchte, das eine Deskription spezialisieren soll, werden sie sich im Allgemeinfall auf Beziehung der beiden sowohl hinsichtlich der Hierarchie von D-Partizipanten als auch der Hierarchie von D-Komponenten und der Hierarchie von Deskriptionen (bzw. von `Description` abgeleiteten Klassen) eingehen. Im Einzelfall kann es (oder sogar muss) auf eine oder zwei von drei Hierarchien wegen der Beschaffenheit verwendeter Ontologien – vor allem wenn die Hierarchien isomorph

zueinander sind – oder aus Gründen der Performanz verzichtet werden[56]. Somit ergibt sich eine Klassifizierung der Spezialisierung gemäß der in Betracht gezogenen Hierarchien; in 6.5.6 werden die möglichen Fälle dieser Klassifizierung mit 1, 2 usw. nummeriert.

6.5.2 Klassifizierung nach den Anforderungen an die spezialisierende Klasse

Stellt sich die Aufgabe, implizite Kontextinformationen ausfindig zu machen, so liegt der Beschluss nahe, Informationen, die von einer Situation selektierte Deskriptionen spezialisieren, in der Hierarchie von D-Partizipanten bzw. Unterklassen von solchen zu suchen; dabei wird es sich im Folgenden um die eine Deskription *spezialisierenden Klassen* handeln.

Es wird nun eine Klassifizierung solcher Fälle vorgeschlagen, in denen man von einer basisontologischen Klasse `C` behaupten könne, dass sie eine bezüglich der vorgegebenen Situation `S` relevante Deskription `D` spezialisiert. Diese Klassifizierung nimmt eine Unterteilung in drei Gruppen vor, die mit I, II, und III bezeichnet werden. Die erste Gruppe betrachtet diejenige Fälle, in denen von `C` erwartet wird, dass es eine Unterklasse eines D-Partizipanten von `D` ist. In der zweiten Gruppe soll `C` zusätzlich expliziterweise D-Partizipant einer weiteren Deskription `D'` sein. In der dritten Gruppe ist die letzte Bedingung insofern gelockert, dass `C` eine Unterklasse eines D-Partizipanten aus `D'` sein darf. Diese Unterteilung entspringt folgenden Überlegungen.

Erstens, es soll eine Möglichkeit geben, auf einfachste Weise über eine Klasse zu spezialisieren, die nicht in der Deskriptionsmodellierung beteiligt sein muss. Angenommen, man hat in `S` eine leere Instanz von `WeatherRegion` als ein D-Partizipant von `D` entdeckt – so ist es durchaus berechtigt, `D` durch eine Unterklasse von `WeatherRegion` spezialisieren zu wollen, denn die Unterklassen von `WeatherRegion` jeweils einen möglichen Wetterstand repräsentieren (z.B. „sonnig“, “regnerisch“ usw.). Der Zweck dabei wäre das Hinzufügen einer Instanz der spezialisierenden Unterklasse zu `S`, also quasi die Anreicherung von `S` mit den aktuellen Wetterangaben. Vorteilhaft bei einem solchen Vorgehen ist, dass wenn man die

[56] Dabei kann es keine pauschale Aussage geben, in welchen Fällen welche Hierarchien relevant sind. Dies ist anwendungsspezifisch und ist dem Entwickler einer konkreten Anwendung überlassen.

Taxonomie der Basisontologie wie in der Abbildung 6.4 approximiert betrachtet, müssen die „unteren“ Klassen nicht in die Deskriptionen eingebunden sein, um bei der Spezialisierung eine Rolle zu spielen. Die Deskriptionen müssen also nur für die „oberen“ Klassen modelliert werden. (Dabei liegt es im Ermessen einer konkreten Anwendung, wo die Grenze zwischen Unten und Oben gezogen wird). Der damit verbundene Nachteil jedoch ist, dass man über die Deskriptionen erfolgende Präzisierung einbüßt: so z.B. einstmals die Deskription, in der `WeatherRegion` und `Road` partizipieren, als relevant erkannt und `WeatherRegion` (über entsprechende Kontextquelle) mit `Rainy` spezialisiert, wird man trotzdem nicht das `Road` mit `Highway` präzisieren können, denn Letzteres in der Modellierung des pragmatischen Wissens unbeteiligt ist.

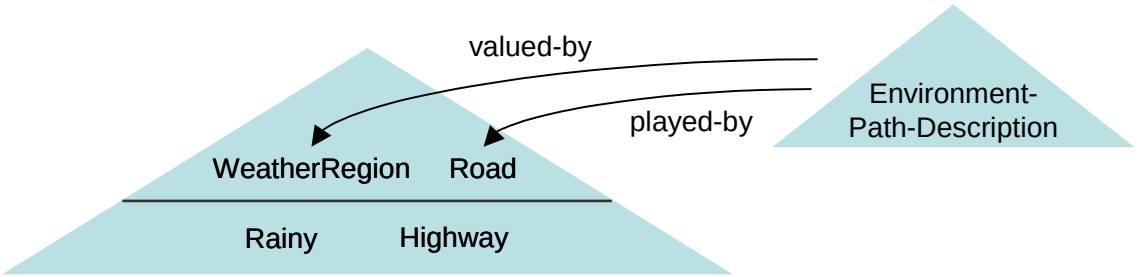


Abbildung 6.4: Links – dreieckförmig approximierte Taxonomie der Basisontologie, rechts – der von `Description` abgeleiteten Klassen. Die Erste ist im Unterteil in der Deskriptionsmodellierung unbeteiligt.

Aus der vorigen Überlegung ergibt sich die zweite: die spezialisierende Klasse `C` soll ein expliziter[57] D-Partizipant einer (anderer als `D`) Deskription `D'` sein. Der wesentliche Unterschied zu dem vorigen Fall ist in der Abbildung 6.5 skizziert. Dort sind die unteren Klassen der Basisontologie in `Rainy-FastPath-Description`, einer Unterdeskription von `Environment-Path-Description`, beteiligt, wodurch `Rainy` in einem derartigen Zusammenhang mit `Highway` auftritt, dass durch ihn `Road` präzisiert werden kann (selbst wenn es an dieser Stelle noch unklar ist, wie dies in allen Einzelheiten geschieht). Der Nachteil liegt jedoch in einem erhöhten Entwicklungsaufwand bei der Ontologie der Deskriptionen: es muss sorgfältig durchdacht werden, welche basisontologische Klassen in die Deskriptionen

[57] „Explizit“ ist hier so zu verstehen, dass die Klasse `C` eine Instanz haben soll, die ein D-Partizipant ist.

eingebunden werden sollen bzw. wie tief die Letzteren reichen sollen, so dass sowohl der Aufwand vertretbar als auch der Zweck erfüllt ist.

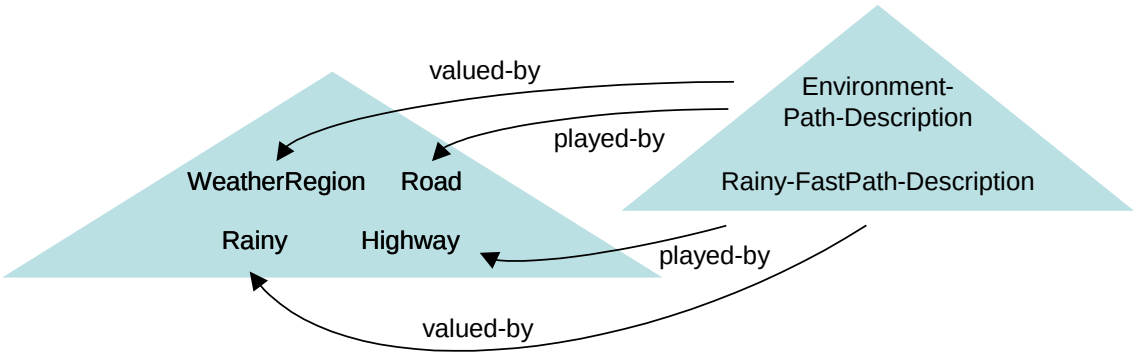


Abbildung 6.5: Links – dreieckförmig approximierte Taxonomie der Basisontologie, rechts – der von `Description` abgeleiteten Klassen.

Abgehalten von einem eventuell unzumutbaren Aufwand, wird in der nächsten Überlegung eine gelockerte Bedingung formuliert: die spezialisierende Klasse `C` soll entweder ein expliziter D-Partizipant einer (anderer als `D`) Deskription `D′` oder Unterklasse eines solchen D-Partizipanten sein. Diese Lockerung behebt zwar die vorigen Nachteile, jedoch liegt ein prinzipieller Unterschied zwischen der Forderung, `C` sei ein expliziter D-Partizipant von `D′` und der Lockerung, `C` darf Unterklasse eines solchen sein. Dieser Unterschied lässt sich am besten am Beispiel in der Abbildung 6.6 erläutern, die aus der Abbildung 6.5 durch das Hinzufügen von `Aquaplaning-Prevention-Description` Deskription und ihrer D-Partizipanten `Rainstorm` und `Railway` entstanden ist. Die neue Deskription besagt, man solle bei Regenschauer vorsichtshalber mit der Bahn fahren.

Nun, es sei wieder `Environment-Path-Description` die zu spezialisierende Deskription `D` und man nehme an, es wurde `Rainstorm` als `C`, also als diejenige Klasse, die `D` spezialisieren kann, determiniert. Ohne der gelockerten Bedingung wird im vorliegenden Fall, auf dieselbe Weise wie bereits oben gezeigt, `Road` aus der Situation `S` durch `Railway` präzisiert. Wird die gelockerte Bedingung angenommen, so wird als `D′`, also als eine Deskription, die mit `C` in Bezug steht, auch `Rainy-FastPath-Description` Deskription akzeptiert, denn `Rainstorm` ist eine Unterklasse ihres D-Partizipanten `Rainy`. Aber dann wird das `Road` in der Situation auch mit `Highway` präzisiert – und genau dies soll ja vermieden werden.

Jedoch dies tritt ein, und zwar deswegen, weil die Taxonomien der Basisontologie und der Deskriptionsontologie nicht isomorph sind: der D-Partizipant `Rainstorm` ist eine Unterklasse vom D-Partizipanten `Rainy` und erbt sozusagen seine Deskription, die aber keine Oberdeskription von `Rainstorm` selbst ist. Nun kurz und knapp auf den Punkt gebracht: der Unterschied zwischen der gelockerten und ungelockerten, strengen Bedingung besteht in der Zu- bzw. Nichtzulassung derartiger Vererbungen, was insbesondere bei den nicht isomorphen Taxonomien von Bedeutung ist.

Zu diesem Beispiel abschließend sei noch angemerkt, man könne die unerwünschte Präzisierung zwar dadurch vermeiden, dass `Rainy-FastPath-Description` und `Aquaplaning-Prevention-Description` als disjunkt zueinander markiert werden. Doch es ist in diesem Fall a priori nicht klar wie die Widersprüchlichkeit zweier Deskriptionen gelöst werden kann (vergleiche 6.4.2).

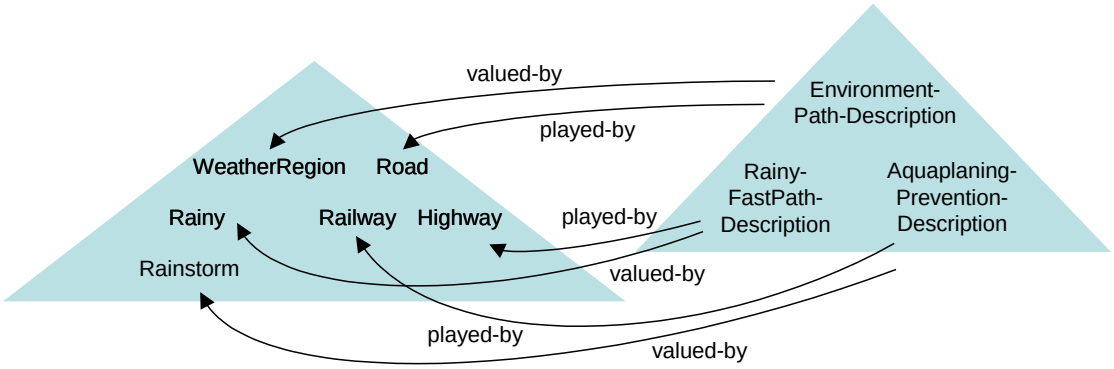


Abbildung 6.6: Links – dreieckförmig approximierte Taxonomie der Basisontologie, rechts – der von `Description` abgeleiteten Klassen. Die zwei Taxonomien sind nicht isomorph.

6.5.3 Diskussion über die zwei vorgeschlagenen Klassifikationen

Die Entscheidung darüber, auf welchem der drei Wege man die spezialisierenden Klassen sucht, hängt natürlich von der Beschaffenheit der Basis- und Deskriptionsontologie, sowie von der Aufgabenstellung und evtl. von dem Aufwand ab, der im Rahmen einer Anwendung vertretbar ist. Hinzu zählt neben dem Entwicklungsaufwand für die Deskriptionsontologie auch der Verarbeitungsaufwand, im Sinne der zeitlichen Performanz jeweiliger Spezialisierungsgruppe I, II, oder III. Dabei lassen sich die Performanzen jeweiliger Gruppen nicht immer pauschal vergleichen. So gewinnt die zweite Gruppe im Vergleich zu der dritten dadurch, dass in

der zweiten Gruppe die Basisontologie unbeachtet bleibt: die Suche findet nur unter den D-Partizipanten, also innerhalb der Deskriptionsontologie statt. Hingegen wird bei der ersten Gruppe nur in der Basisontologie gesucht, denn die spezialisierende Klasse nicht in die Deskriptionen eingebunden werden muss. Ob dies allerdings ein Gewinn im Vergleich zu den Gruppen II und III ausmacht – ist unter anderem von den Größen und der Beschaffenheit beider Ontologien abhängig. Somit behalten alle drei Gruppen die Existenzberechtigung und bleiben an dieser Stelle nur vorgeschlagen und zur Auswahl gestellt.

Betrachtet man die beiden vorgeschlagenen Klassifizierungen noch ein Mal zusammenfassend, so kann man sie sich veranschaulichend wie folgt vorstellen. Die Klassifikation nach Hierarchien untersucht das Verhältnis von D' zu D, also das Verhältnis einer Deskription, in der die spezialisierende Klasse C partizipiert, zu der Deskription, die zu spezialisieren gilt. Und die zweite Klassifikation stellt die Frage, ob überhaupt eine solche Deskription D' für die Spezialisierung notwendig ist, und wenn es der Fall sein sollte, welches Verhältnis sollte dann die spezialisierende Klasse C zu D' haben – in ihr unmittelbar (explizit) partizipieren oder eventuell auch eine Unterklasse eines D-Partizipanten von D' sein.

6.5.4 Subsumierung von D-Komponenten

In 6.5.6 wird es notwendig sein, D-Komponenten in Hinblick auf die Deskriptionsrelationen, in die sie eingebunden sind, zu vergleichen. Deswegen wird an dieser Stelle Begriff DR*-Subsumierung eingeführt, der eigentlich eine spezielle Subsumierungsrelation auf der Menge aller solchen Individuen darstellt, die Instanzen von D-Komponenten sind. Dabei wird hier keine feste und verbindliche Definition formuliert, sondern vielmehr bestimmte Gruppen von Bedingungen, von denen für jede konkrete Anwendung die geeigneten gewählt werden können.

6.5.4.1 Definitionen der Subsumierung

Ein Individuum i_2 *DR-subsumiert* ein Individuum i_1 genau dann, wenn von folgenden zwei Bedingungsgruppen (G1 und G2) die für eine konkrete Anwendung geforderten – also keine, eine von beiden oder beide – erfüllt sind:

G1 Für jede solche Deskriptionsrelation dr_1 und Individuum x, dass:

1) $dr_1(i_1,x)$,
2) x eine nicht optionale D-Komponente ist,

eine Deskriptionsrelation dr_2 und ein Individuum y existieren, für die gilt[58]:

a) $dr_2(i_2,y)$,
b) dr_1 ist eine Unterrelation von dr_2 oder sie sind identisch,
c) eine explizite Klasse von x ist Unterklasse einer expliziten Klasse von y oder sie sind identisch,
d) y ist eine nicht optionale D-Komponente.

G2 Für jede solche Deskriptionsrelation dr_2 und Individuum y, dass:

1) $dr_2(i_2,y)$,
2) y ist eine nicht optionale D-Komponente,

eine Deskriptionsrelation dr_1 und ein Individuum x existieren, für die gilt:

a) $dr_1(i_1,x)$,
b) dr_1 ist eine Unterrelation von dr_2 oder sie sind identisch,
c) eine explizite Klasse von x ist Unterklasse einer expliziten Klasse von y oder sie sind identisch,
d) x ist eine nicht optionale D-Komponente.

Der Begriff *DR^{-1}-Subsumierung* wird genau wie die DR-Subsumierung definiert, jedoch mit dem einzigen Unterschied, dass die Argumente der Deskriptionsrelationen vertauscht werden, also z.B. $dr_1(x,i_1)$ statt $dr_1(i_1,x)$. Zwischen Individuen i_1 und i_2 ist *DR*-Subsumierung* genau dann gegeben, wenn zwischen ihnen sowohl DR-Subsumierung als auch DR^{-1}-Subsumierung gegeben sind.

Aus der DR*-Subsumierung unter Individuen, die wohl bemerkt D-Komponenten sind, lässt sich DR*-Subsumierung unter Deskriptionen formulieren. Eine Deskription D *DR-subsumiert* eine andere Deskription D' genau dann, wenn jede D-Komponente von D' von einer D-Komponente von D DR-subsumiert wird und wenn D eine Unterklasse von D' ist. Entsprechend werden *DR^{-1}-Subsumierung* und *DR*-Subsumierung* unter Deskriptionen definiert.

[58] Hier und im Weiteren verstehen sich die Bedingungen mit unterstrichener Nummerierung, also a), b) usw. als optional.

6.5.4.2 Erläuterungen zur Subsumierung an einem Beispiel

Die Bedeutung von DR-Subsumierung lässt sich am einfachsten an einem Beispiel erläutern. Die Abbildung 6.7 zeigt ein solches Beispiel, in dem zwei D-Komponenten – DK i_1 und DK i_2 – abgebildet sind, sowie Deskriptionsrelationen, in denen sie beteiligt sind und die anderen D-Komponenten, die sich an diesen Deskriptionsrelationen beteiligen und zum Teil als optional gekennzeichnet sind.

Damit DK i_2 DK i_1 DR-subsumiert, fordert G1, dass für jede Deskriptionsrelation, die, bildlich gesprochen, aus DK i_1 ausgeht und zu einer nicht optionalen D-Komponente führt (die einzige solche im Beispiel – DR7), eine aus DK i_2 ausgehende Deskriptionsrelation existiert, die die erste subsumiert und egebenenfalls zu einer nicht optionalen D-Komponente führt, die die obere subsumiert. Das ist in diesem Fall gegeben, denn DR7 von DR4 subsumiert ist. Sogar die optionale Bedingungen c) und d) sind erfüllt, denn DK4 nicht optional ist und DK7 subsumiert.

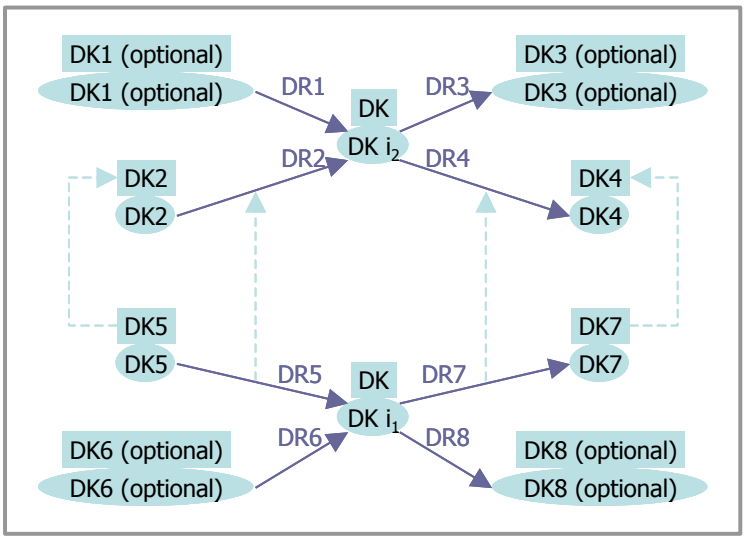


Abbildung 6.7: DK i_2 DR*-subsumiert DK i_1; es gelten jeweils sowohl die Bedingungen aus G1 als auch aus G2. DK steht abkürzend für D-Komponente und DR – für Deskriptionsrelation. (*)

Es wird also in G1 gefordert, dass für jede aus DR-subsumierter D-Komponente ausgehende Deskriptionsrelation eine „passende“ aus DR-subsumierender D-Komponente ausgehende Deskriptionsrelation existiert. Die Gruppe G2 stellt ähnliche Bedingungen auf, fordert jedoch im Gegenteil zu G1, dass für jede aus DR-

subsumierender D-Komponente ausgehende Deskriptionsrelation eine „passende“ aus DR-subsumierter D-Komponente ausgehende Deskriptionsrelation existiert. DK i_1 und DK i_2 im Beispiel aus der Abbildung 6.7 genügen auch dieser Anforderung. Wenn jedoch beispielsweise DK3 nicht optional wäre, dann wären die Bedingungen aus G2 unerfüllt, weil aus DK i_1 keine Deskriptionsrelation ausgeht, die eine Oberrelation von DR3 ist; G1 bliebe dennoch erfüllt.

Ob in einer konkreten Anwendung G1 oder G2 vorausgesetzt werden soll bzw. ob die jeweiligen optionalen Bedingungen gefordert werden, hängt mitunter davon ab, ob und inwiefern die Deskriptionen, zu denen eine subsumierte D-Komponente gehört, von solchen Deskriptionen, zu denen eine subsumierende D-Komponente gehört, die D-Komponenten bzw. das Geflecht aus ihnen und sie verbindenden Deskriptionsrelationen vererben soll (die Vererbung wurde in 6.5.2 diskutiert).

Die DR^{-1}-Subsumierung stellt dieselbe Forderungen auf wie die DR-Subsumierung auf, nur statt aus der subsumierten bzw. subsumierenden D-Komponente ausgehender Deskriptionsrelationen bezieht sich DR^{-1}-Subsumierung auf die eingehende Deskriptionsrelationen. Aus der Abbildung 6.7 ist es leicht ersichtlich, dass im dortigen Beispiel auch diese Bedingungen erfüllt sind und es somit gilt: DK i_2 DR^{-1}-subsumiert DK i_2.

6.5.5 Strukturerhaltung unter den D-Partizipanten zweier D-Komponenten

Ähnlich wie in 6.5.4 die Struktur unter D-Komponenten betrachtet und verglichen wurde, beschäftigt sich dieser Abschnitt mit den D-Partizipanten. Hier gestaltet sich ein solches Vergleich deutlich einfacher, denn die Situationsrelationen zwischen den Individuen, die D-Partizipanten sind, tatsächlich nur in den Situationen vorkommen und nicht innerhalb der Deskriptionsontologie. Somit steht es hier im Mittelpunkt, wie sich die D-Partizipanten, die eine D-Komponente parametrisieren, zu den D-Partizipanten, die eine andere D-Komponente parametrisieren, im Sinne der hierarchischen Beziehung verhalten. Diesbezüglich gelte folgende Definition: eine D-Komponente DK1 *impliziert* eine andere D-Komponente DK2 genau dann, wenn es (in der Deskriptionsontologie) ein D-Partizipant existiert, der DK1 parametrisiert und eine echte Unterklasse eines D-

Partizipanten ist, der DK2 parametrisiert. Man beachte, dass es nicht gefordert wird, jeder DK1 parametrisierender D-Partizipant solle Unterklasse eines DK2 parametrisierenden D-Partizipanten sein. Dies, weil alle D-Partizipanten per Definition als optional gelten (vergleiche 6.3 und 6.4.1). Somit kann es auf jeden einzelnen D-Partizipanten verzichtet werden, d.h. es muss für die implizierende D-Komponente mindestens einer vorhanden sein, der die geforderte Bedingung erfüllt.

Die Implikation zweier D-Komponenten lässt sich auf Deskriptionen übertragen wie folgt: Eine Deskription D *impliziert* eine andere Deskription D′ genau dann, wenn jede D-Komponente von D eine D-Komponente von D′ impliziert.

6.5.6 Definitionen von Spezialisierung

In diesem Abschnitt werden nun verschiedene Definitionen für die Spezialisierung von Deskriptionen formuliert, und zwar gemäß der in 6.5.1 und 6.5.2 vorgeschlagenen Klassifizierungen. Trotz der weit reichenden Überlappung vieler dieser Definitionen, wird es, zur besseren Übersichtlichkeit, nicht auf die genaueste Formulierung jeder einzelnen Definition sowie sie begleitender Skizze[59] verzichtet. Die Definitionen werden mit den Zahlen 1, 2, usw. – entsprechend der ersten Klassifikation – und mit I, II, oder III entsprechend der zweiten Klassifikation nummeriert. Somit hat jede Spezialisierungsdefinition eine Kennzeichnung, z.B. 1.I. Dementsprechend haben die einzelnen Bedingungen eine Kennzeichnung wie z.B. 1.I.a) bzw. a) für die allgemeine notwendige Bedingungen (s.u.) und a) für die allgemeine optionale Bedingungen (s.u.).

Für jede Definition gilt ein allgemeiner, notwendiger Teil:

> Eine Klasse C spezialisiert eine Deskription D respektive einer vorgegebenen Situation S, wenn Folgendes gilt:
>
> a) D ist erfüllt bezüglich S,
>
> b) C ist determinierbar.

[59] In den Skizzen wird mit DK D-Komponente abkürzend bezeichnet, mit DP – D-Partizipant und PR steht für Partizipierungsrelation.

Für jede Definition[60] können nach Belieben folgende allgemeine optionale Bedingungen gefordert werden:

a) C ist kein D-Partizipant von D,

b) D′ – eine Deskription, bei der C ein D-Partizipant oder Unterklasse eines solchen ist – ist bezüglich S nicht erfüllt,

c) D′ impliziert D,

d) D DR*-subsumiert D′.

Im Folgenden werden je nach der obigen Klassifikationen spezifische Bedingungen der jeweiligen Spezialisierungsdefinition aufgestellt.

1. Spezialisierung nach der Hierarchie von D-Partizipanten (bzw. Unterklassen von solchen)

Bedingungen für die Spezialisierung 1.I:

a) C ist echte Unterklasse eines D-Partizipanten Z von D,

b) Es existiert eine Kontextquelle, von der C und Z determinierbar sind.

Bedingungen für die Spezialisierung 1.II:

a) C ist echte Unterklasse eines D-Partizipanten Z von D,

b) C ist D-Partizipant einer Deskription D′,

c) D′ ist ungleich D,

d) Es existiert eine Kontextquelle, von der C und Z determinierbar sind.

Bedingungen für die Spezialisierung 1.III:

a) C ist echte Unterklasse eines D-Partizipanten Z von D,

b) C ist D-Partizipant einer Deskription D′, oder Unterklasse eines solchen,

c) D′ ist ungleich D,

d) Es existiert eine Kontextquelle, von der C und Z determinierbar sind.

[60] Bei der Gruppe I kann nur die Bedingung a) verlangt werden.

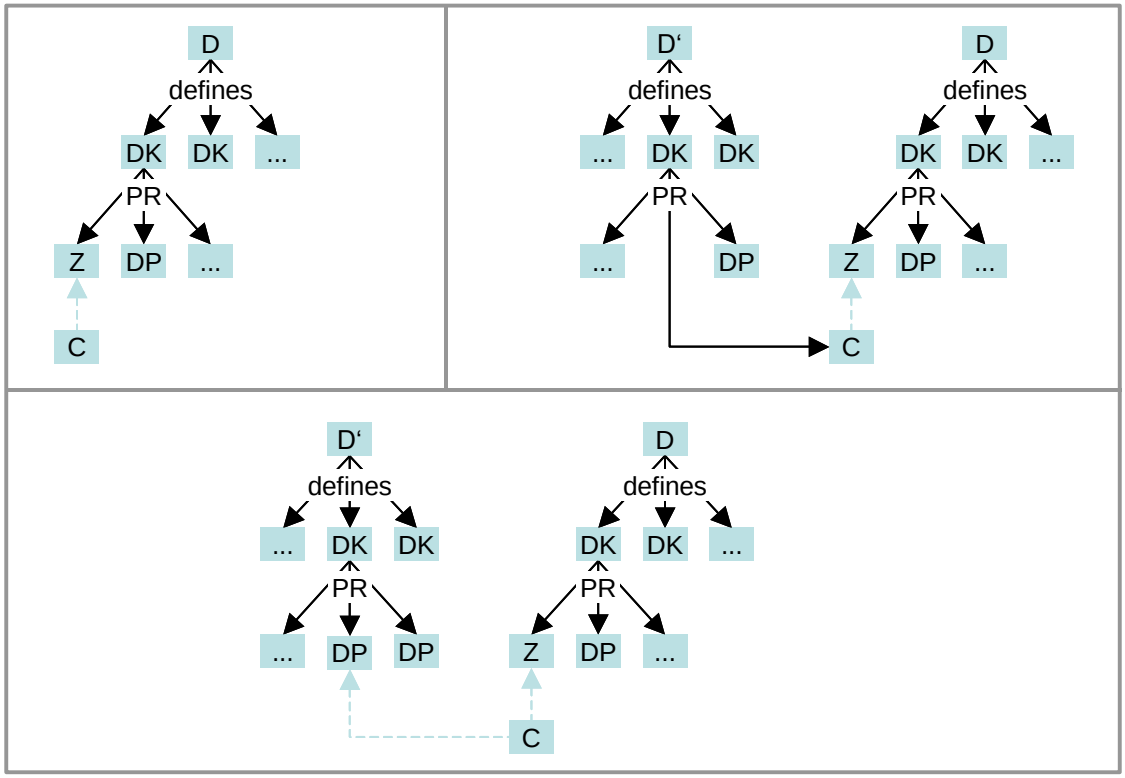


Abbildung 6.8: Oben links – Spezialisierung 1.I; oben rechts – Spezialisierung 1.II; unten – Spezialisierung 1.III. (*)

Man beachte, dass wenn es bei der Spezialisierung auf die Hierarchie von D-Partizipanten zurückgegriffen wird, lässt es sich nicht nur feststellen, dass `C D` spezialisiert, sondern auch was genau `C` in `D` spezialisiert – nämlich den D-Partizipanten `Z` von `D`.

2. Spezialisierung nach der Hierarchie von Deskriptionen

Bedingungen für die Spezialisierung 2.II:

a) `C` ist D-Partizipant einer Deskription `D'`,

b) `D'` ist eine echte Unterklasse von `D`.

Bedingungen für die Spezialisierung 2.III:

a) `C` ist D-Partizipant einer Deskription `D'`, oder Unterklasse eines solchen,

b) `D'` ist eine echte Unterklasse von `D`.

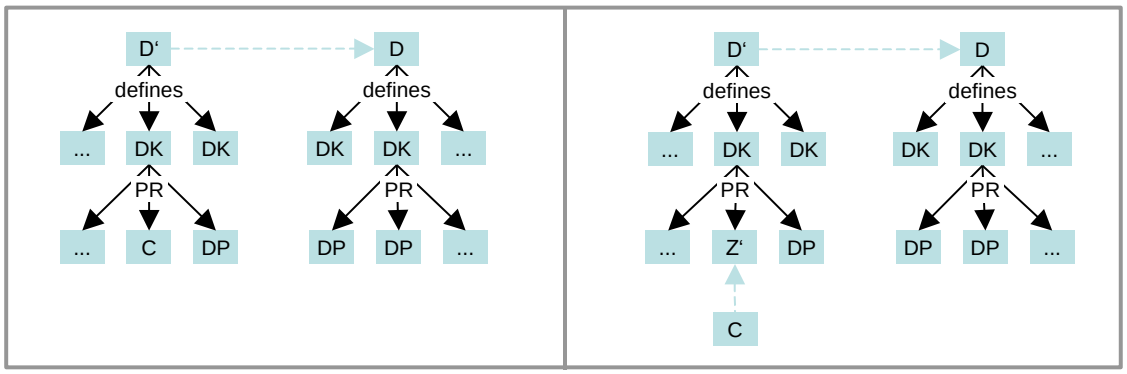


Abbildung 6.9: Links – Spezialisierung 2.II; rechts – Spezialisierung 1.III. (*)

Mit dieser, sowie auch mit folgenden Klassen der Klassifikation nach Hierarchien, ist die Gruppe I nicht kombinierbar, denn die Hierarchie von Deskriptionen bzw. von D-Komponenten einbezogen wird, und dies setzt einen Bezug von C zu einer Deskription D′ voraus.

Man beachte, dass im Gegensatz zur Spezialisierung nach der Hierarchie von D-Partizipanten, bei der es offensichtlich war, welchen D-Partizipanten C in D spezialisiert, man hier, nur auf die Hierarchie der Deskriptionen zurückgreift, aus der a priori nicht feststellbar ist, was genau C in D spezialisiert.

3. Spezialisierung nach der Hierarchie von D-Komponenten

Bedingungen für die Spezialisierung 3.II:

a) C ist D-Partizipant einer Deskription D′,
b) Eine D-Komponente K′ von D′, die von C parametrisiert wird, ist Unterklasse einer D-Komponente K von D,
c) K DR-subsumiert K′,
d) D′ ist ungleich D.

Bedingungen für die Spezialisierung 3.III:

a) C ist D-Partizipant einer Deskription D′, oder Unterklasse eines solchen D-Partizipanten Z′,
b) Eine D-Komponente K′ von D′, die von Z′ parametrisiert wird, ist Unterklasse einer D-Komponente K von D,
c) K DR-subsumiert K′,
d) D′ ist ungleich D.

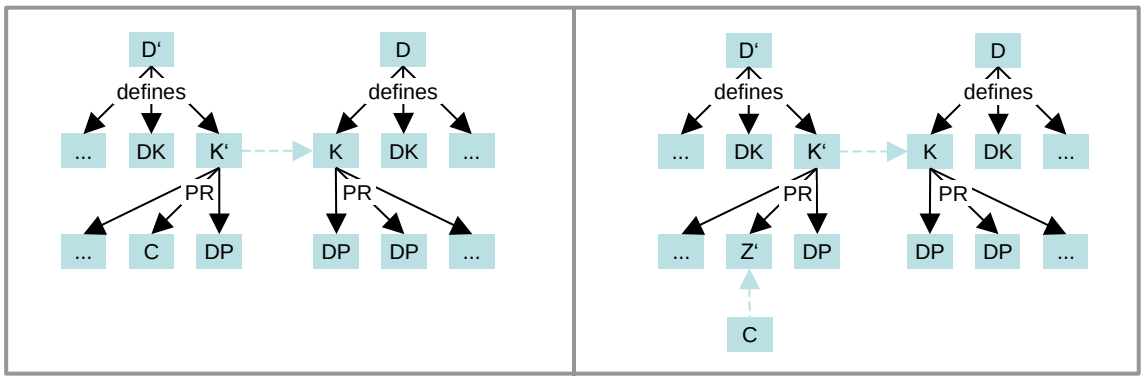


Abbildung 6.10: Links – Spezialisierung 3.II; rechts – Spezialisierung 3.III. (*)

Man beachte, dass wenn bei der Spezialisierung auf die Hierarchie von D-Komponenten zurückgegriffen wird, lässt es sich nur feststellen, welche D-Komponente von `D` von `C` spezialisiert wird, jedoch bleibt es ohne Weiteres verborgen, welchen diese D-Komponente parametrisierenden D-Partizipanten `C` spezialisiert, denn es im Allgemeinen mehrere solche gibt.

4. Spezialisierung nach den Hierarchien von Deskriptionen und D-Partizipanten (bzw. Unterklassen von solchen)

Bedingungen für die Spezialisierung 4.II:

a) `C` ist echte Unterklasse eines D-Partizipanten `Z` von `D`,

b) `C` ist D-Partizipant einer Deskription `D'`,

c) `D'` ist eine echte Unterklasse von `D`,

d) Es existiert eine Kontextquelle, von der `C` und `Z` determinierbar sind.

Bedingungen für die Spezialisierung 4.III:

a) `C` ist echte Unterklasse eines D-Partizipanten `Z` von `D`,

b) `C` ist D-Partizipant einer Deskription `D'`, oder Unterklasse eines solchen,

c) `D'` ist eine echte Unterklasse von `D`,

d) Es existiert eine Kontextquelle, von der `C` und `Z` determinierbar sind.

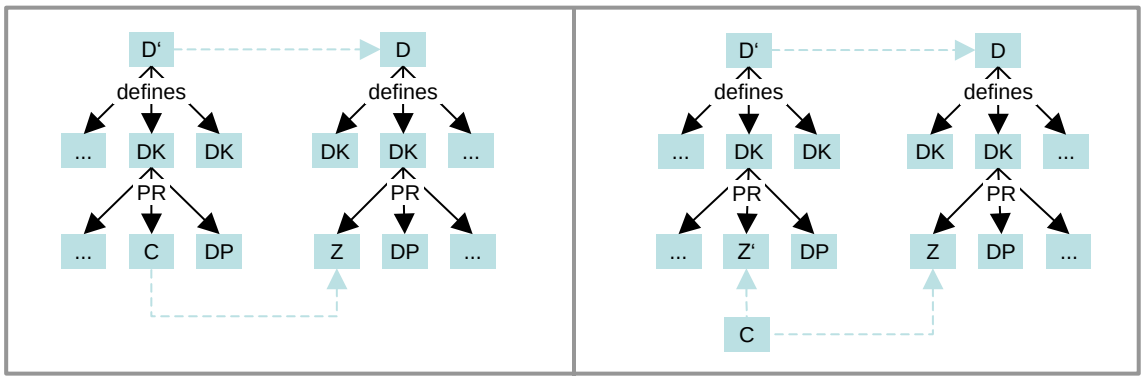


Abbildung 6.11: Links – Spezialisierung 4.II; rechts – Spezialisierung 4.III. (*)

5. Spezialisierung nach den Hierarchien von D-Komponenten und D-Partizipanten (bzw. Unterklassen von solchen)

Bedingungen für die Spezialisierung 5.II:

a) `C` ist D-Partizipant einer Deskription `D'`,

b) Eine D-Komponente `K'` von `D'`, die von `C` parametrisiert wird, ist Unterklasse einer D-Komponente `K` von `D`,

c) `K` DR-subsumiert `K'`,

d) `C` ist echte Unterklasse eines D-Partizipanten `Z` von `D`,

e) `D'` ist ungleich `D`,

f) Es existiert eine Kontextquelle, von der `C` und `Z` determinierbar sind.

Bedingungen für die Spezialisierung 5.III:

a) `C` ist D-Partizipant einer Deskription `D'`, oder Unterklasse eines solchen D-Partizipanten `Z'`,

b) Eine D-Komponente `K'` von `D'`, die `Z'` parametrisiert, ist Unterklasse einer D-Komponente `K` von `D`,

c) `K` DR-subsumiert `K'`,

d) `C` ist echte Unterklasse eines D-Partizipanten `Z` von `D`,

e) `D'` ist ungleich `D`,

f) Es existiert eine Kontextquelle, von der `C` und `Z` determinierbar sind.

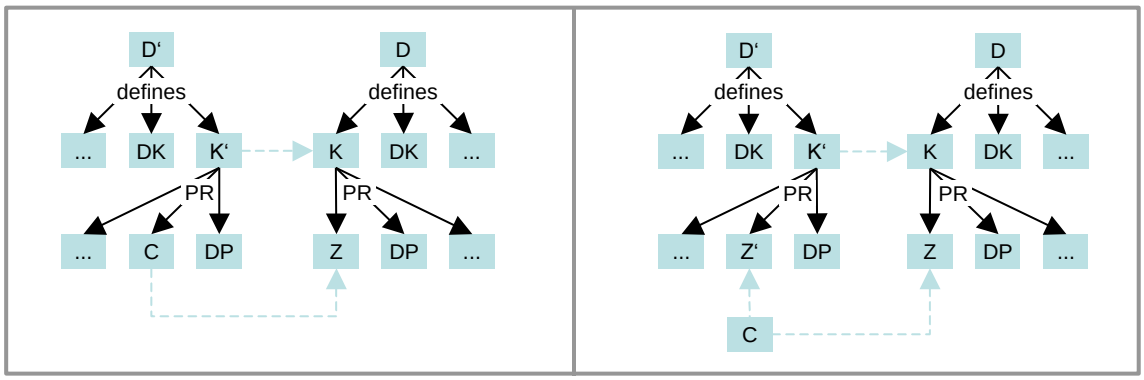


Abbildung 6.12: Links – Spezialisierung 5.II; rechts – Spezialisierung 5.III. (∗)

6. Spezialisierung nach den Hierarchien von Deskriptionen und D-Komponenten

Bedingungen für die Spezialisierung 6.II:

a) `C` ist D-Partizipant einer Deskription `D′`,

b) Eine D-Komponente `K′` von `D′`, die von `C` parametrisiert wird, ist Unterklasse einer D-Komponente `K` von `D`,

c) `K` DR-subsumiert `K′`,

d) `D′` ist eine echte Unterklasse von `D`.

Bedingungen für die Spezialisierung 6.III:

a) `C` ist D-Partizipant einer Deskription `D′`, oder Unterklasse eines solchen D-Partizipanten `Z′`,

b) Eine D-Komponente `K′` von `D′`, die von `Z′` parametrisiert wird, ist Unterklasse einer D-Komponente `K` von `D`,

c) `K` DR-subsumiert `K′`,

d) `D′` ist eine echte Unterklasse von `D`.

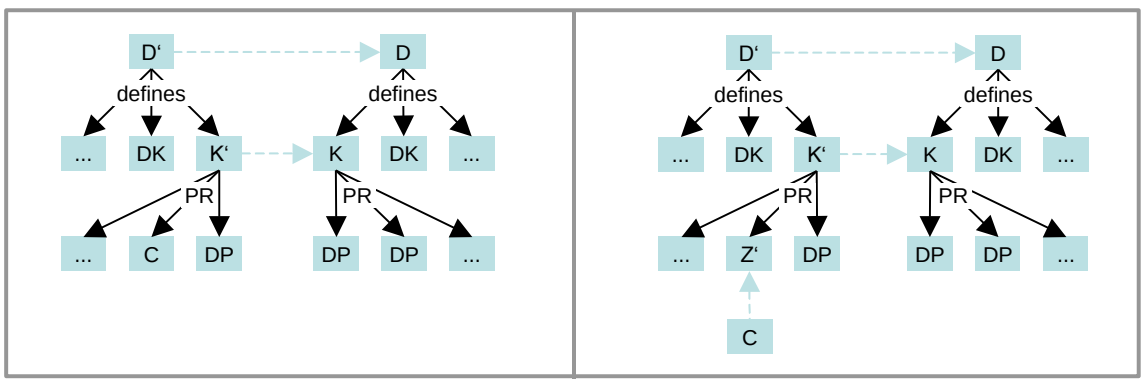


Abbildung 6.13: Links – Spezialisierung 6.II; rechts – Spezialisierung 6.III. (∗)

7. Spezialisierung nach den Hierarchien von Deskriptionen, D-Komponenten und D-Partizipanten (bzw. Unterklassen von solchen)

Bedingungen für die Spezialisierung 7.II:

a) `C` ist D-Partizipant einer Deskription `D'`,

b) Eine D-Komponente `K'` von `D'`, die von `C` parametrisiert wird, ist Unterklasse einer D-Komponente `K` von `D`,

c) `K` DR-subsumiert `K'`,

d) `C` ist echte Unterklasse eines D-Partizipanten `Z` von `D`,

e) `D'` ist eine echte Unterklasse von `D`,

f) Es existiert eine Kontextquelle, von der `C` und `Z` determinierbar sind.

Bedingungen für die Spezialisierung 7.III:

a) `C` ist D-Partizipant einer Deskription `D'`, oder Unterklasse eines solchen D-Partizipanten `Z'`,

b) Eine D-Komponente `K'` von `D'`, die `Z'` parametrisiert, ist Unterklasse einer D-Komponente `K` von `D`,

c) `K` DR-subsumiert `K'`,

d) `C` ist echte Unterklasse eines D-Partizipanten `Z` von `D`,

e) `D'` ist eine echte Unterklasse von `D`,

f) Es existiert eine Kontextquelle, von der `C` und `Z` determinierbar sind.

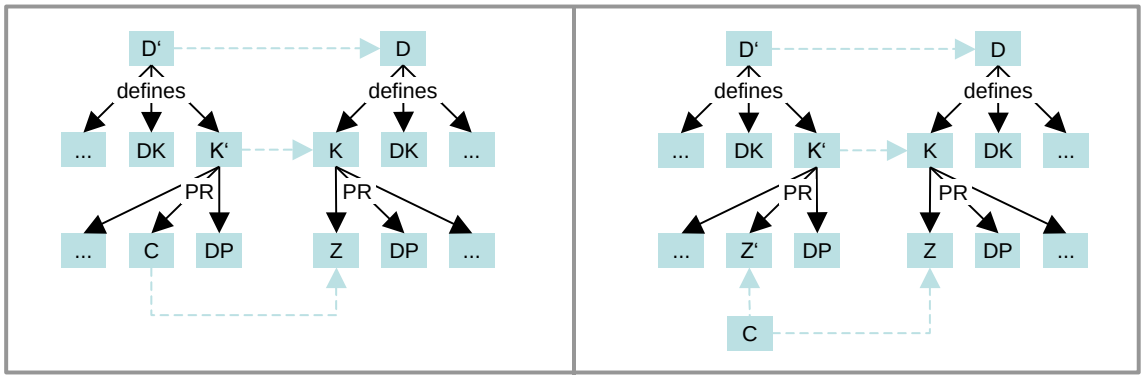


Abbildung 6.14: Links – Spezialisierung 7.II; rechts – Spezialisierung 7.III. (∗)

6.5.7 Diskussion über den Begriff „Spezialisierung“ und seine Definitionen

Die hierarchiebasierende Klassifikation von Spezialisierungsdefinitionen weist die in der Abbildung 6.15 (links) abgebildete Halbordnung auf. Diese bezieht sich auf die Implikationsrelation zwischen einzelnen Klassen der Klassifikation. Zum Beispiel implizieren die Definitionen der Klasse 4 die Definitionen aus Klassen 1 und 2. Außerdem zeigt die Abbildung 6.15 (rechts), dass jede Definition der Klasse 2 bis 7 durch die optionale Bedingung c), also quasi eine Forderung nach Strukturgleichheit von D-Partizipanten der spezialisierten und der spezialisierenden (ontologische) Klasse beinhaltenden Deskription, gestärkt werden kann. Genauso kann jede Definition der Klasse 2 bis 7 durch die optionale Bedingung d) gestärkt werden, die die Strukturerhaltung unter D-Komponenten beider Deskriptionen fordert. Schließlich kann man beides verlangen, und, wenn man dies bei den Spezialisierungsdefinitionen aus 7 tut, erhält man die strengsten Anforderungen überhaupt. Die Abbildung 6.15 schildert die wichtigsten Beziehungen zwischen verschiedenen Definitionen der Spezialisierung. Sie ist also die Essenz aus dem der Spezialisierung gewidmeten Unterkapitel.

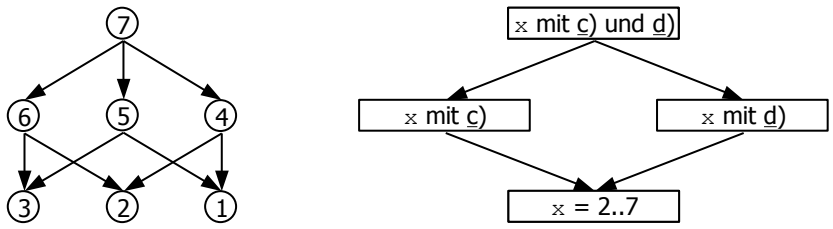


Abbildung 6.15: Implikationsbeziehungen unter verschiedenen Definitionen der Spezialisierung.

Ob die strengsten bzw. überhaupt welche Anforderungen an die Spezialisierung in einer konkreten Anwendung notwendig oder vertretbar sind, hängt, wie mehrmals erwähnt, von der Anwendung ab und kann an dieser Stelle natürlich nicht entschieden werden. Vielmehr bezweckt dieser Abschnitt sowie das ganze umfangreiche Unterkapitel zur Spezialisierung diesen Begriff als ein entscheidendes Hilfsmittel einzuführen, ihn zu klären, umfassend zu klassifizieren und zu definieren. Und es sei nochmals darauf hingewiesen, dass die Spezialisierung deswegen von enormer Wichtigkeit ist, weil sie zur Realisierung der Präzisierung von Aussagen, also der Hauptangelegenheit dieser Arbeit, maßgeblich beiträgt.

6.6 Komplettierung von Situationen

Mit der Spezialisierung wurde ein Vorgang beschrieben, bei dem man unter anderem über eine Situation und eine ihr bezüglich relevante Deskription verfügte und die in der Deskriptionsontologie genannten D-Partizipanten dieser Deskription betrachtete. Dabei wurde stillschweigend davon ausgegangen, dass in der Situation selbst diese D-Partizipanten ebenfalls vorkommen. Das Beispielszenario für diesen Vorgang war eine Situation, in der ein (leeres) nichts bestimmtes über die Wetterverhältnisse aussagendes Individuum von `WeatherRegion` und ein Individuum von allgemeinem Straßentyp `Road` vorkamen, und die für diese Situation relevante Deskription mit konkreten Angaben zum aktuellen Wetter spezialisiert wurde, woraus folgernd `Road` mit z.B. `Highway` präzisiert wurde (siehe Kapitel 3).

Doch wenn man es zulässt, dass die relevanten Deskriptionen von Situationen selektiert (statt erfüllt) sein dürfen, ist es gemäß der Definition nicht zwingend erforderlich, dass alle D-Partizipanten in der Situation tatsächlich vorkommen. Genauer ausgedrückt, die determinierbaren D-Partizipanten dürfen fehlen. In diesem Fall hat man es mit einem anderen Beispielsszenario zu tun. Dabei kommt es in der Situation beispielsweise nur ein Individuum von `Road` vor, woraus man über die entsprechende relevante Deskription feststellt, dass in gegebener Situation die Wetterverhältnisse von Bedeutung sind, diese folglich ermittelt und Schlüsse daraus zieht. Die Komplettierung bezeichnet dabei den Vorgang, in dem die Situation mit derartig ermittelten Informationen bereichert wird. Falls dies für jeden determinierbaren und in der Situation fehlenden D-Partizipanten gelingt, wird die komplettierte Situation gemäß der Definition die relevante Deskription erfüllen. Es sei also erlaubt in diesem Falle die Deskription als komplettiert zu bezeichnen bzw. den Vorgang auch als Komplettierung der (relevanten) Deskription zu nennen.

Während der Komplettierung wird also der Situation für jeden ihr fehlenden und determinierbaren D-Partizipanten ein Individuum hinzugefügt, das für diesen D-Partizipanten determiniert (d.h. über Kontextquellen ermittelt) wurde. Demzufolge ist die explizite Klasse dieses Individuums, sie sei *komplettierende Klasse* genannt, entweder die vom D-Partizipanten oder eine Unterklasse des D-Partizipanten. Ein Beispiel für den letzteren Fall wäre die ermittelte aktuelle Wetterlage `Rainy` (bzw.

genauer: ein Individuum dieser Klasse, in dem die entsprechenden Angaben zum Wetter festgehalten sind), mit der die Situation komplettiert wird, weil in ihr D-Partizipant `WeatherRegion` gefehlt hat. Dies ähnelt im Endergebnis sehr der Bestimmung einer spezialisierenden Klasse[61]. Der Unterschied liegt darin, dass eine spezialisierende Klasse unter den D-Partizipanten bzw. Unterklassen von solchen erst gefunden werden muss und dann determiniert werden kann, während eine komplettierende Klasse direkt determiniert wird. Somit ist die Komplettierung mit weniger Aufwand als die Spezialisierung verbunden.

6.7 Formalisierung des Begriffs „Präzisierung einer Aussage“

Die Idee der Präzisierung einer Aussage wurde bereits in 6.5.2 erläutert und wird in 7.3 am Beispiel demonstriert. Sie besteht darin, dass einige Individuen in einer Situation als Instanzen von Unterklassen (bzgl. der Klassen zu denen sie eigentlich explizit gehören) inferiert werden. So wurde in vorigen Beispielen (siehe u.a. Kapitel 3) eine Instanz von allgemeinem Straßentyp `Road` als eine Instanz von speziellerem Straßentyp `Highway` präzisiert. Nun soll diese Idee formal festgehalten werden.

6.7.1 Präzisierung über eine spezialisierende Deskription

In diesem Abschnitt wird eine Option geschildert, die Präzisierung aus solchen Deskriptionen zu gewinnen, in denen spezialisierende Klassen partizipieren. Die Präzisierung wird hier also als eine Konsequenz der Spezialisierung eingeführt. Dazu greife man zu den Bezeichnungen aus 6.5.2 und 6.5.6 zu. Man hat also eine Situation `S` – sie repräsentiert eine Aussage (des Benutzers), – eine bezüglich dieser Situation relevante Deskription `D`, eine sie (nach II oder III) spezialisierende Klasse `C` und eine Deskription `D'`, dessen D-Partizipant (oder Unterklasse eines solchen) `C` ist.

Jeder D-Partizipant `DP1` von `D'` (außer `C`) präzisiert jedes Individuum `I` aus `S`, für das gilt:

1) `I` ist ein D-Partizipant von `D` (man bezeichne ihn `DP2`),
2) `DP1` ist eine Unterklasse von `DP2`.

[61] Vergleiche Abbildung 6.4 (beschrieben in 6.5.2); Spezialisierungsdefinition 1.I und Abbildung 6.8.

Präzisiert wird, indem I explizit als eine Instanz von DP1 erklärt wird.

Man beachte, dass wenn bei der Spezialisierung die optionale Bedingung c) gefordert wird, sich die jeweiligen präzisierenden und präzisierten D-Partizipanten (also DP1 und DP2) bereits während der Überprüfung dieser Bedingung feststellen lassen, denn c) ja gerade die hierarchische Struktur zwischen D-Partizipanten von D und D′ zu erschließen bezweckt.

6.7.2 Präzisierung aus determinierten D-Partizipanten

Wurde eine bezüglich einer Situation S relevante Deskription D der Komplettierung unterzogen, so ist es möglich, dass eine komplettierende Klasse C, also diejenige Klasse, die für einen D-Partizipanten DP von D determiniert wurde, weil in S kein Individuum von DP vorkam, eine Unterklasse von DP ist. In diesem Fall ist eine Präzisierung von S ohne vorheriger Spezialisierung dann möglich, wenn C ein D-Partizipant einer anderen (als D) Deskription D′ ist. Mit den gerade verwendeten Bezeichnungen gleicht die Definition für die Präzisierung in diesem Fall exakt der in 6.7.1 formulierten Definition. Zu beachten ist, dass die vorherige Spezialisierung bei D′, also bei derjenigen Deskription aus der heraus präzisiert wird, bestimmte Eigenschaften bezüglich ihres Verhältnisses zu D voraussetzt, die natürlich entfallen, wenn es direkt nach der Komplettierung präzisiert wird.

6.7.3 Präzisierung aus in einer Situation vorhandenen D-Partizipanten

Es kann vorkommen, dass für eine Situation S eine Deskription D als relevant befunden wurde und ein D-Partizipant DP von D, deren dessen Instanz in S vorkommt, auch ein D-Partizipant einer anderen Deskription D′ ist. Wenn man in diesem Fall mit C jede Klasse bezeichnet, die PR-Target von D′ ist und deren Instanz DP ist, dann ist, äquivalent zu 6.7.2, die Präzisierung möglich und wird genau so definiert, wie in 6.7.1.

Um Beispiel für eine derartige Präzisierung zu geben betrachte man die Abbildung 6.5 und nehme an, in einer Situation kämen Instanzen von Rainy und Road vor. Diese erfüllen nur die Deskription Environment-Path-Description. Diese Lage ist jedoch sehr ähnlich zu derjenigen, bei der die Situation statt einer Instanz von Rainy

eine von `WeatherRegion` enthalten würde und `Rainy` wäre dann in Folge der Komplettierung ermittelt. Deshalb möchte man in diesem Fall ebenfalls präzisieren, und zwar ohne zu unterscheiden, wie die Instanz von `Rainy` in der Situation entstanden ist.

Diese Art der Präzisierung soll optional zugelassen werden, denn es folgenden grundlegenden Unterschied zu vorigen beiden Präzisierungen aufweist. Es wird nicht aus einer ermittelten Kontextinformation heraus, sondern aus einem in der Situation bereits vorhandenen D-Partizipanten präzisiert. Dabei erweitert sich der von der Präzisierung propagierte kausaler Zusammenhang „ein determinierter D-Partizipant einer Deskription impliziert alle andere D-Partizipanten von ihr“ in folgenden: „jeder D-Partizipant einer Deskription impliziert alle andere D-Partizipanten von ihr“. Eine solche Kausalisierungsausdehnung kann jedoch eventuell zu stark sein und der Wirklichkeit oder der Intention nicht entsprechen. Man könnte ihr Grenzen setzen, und zwar: „ein D-Partizipant DP_1 einer Deskription `D` impliziert einen D-Partizipanten DP_2 von ihr genau dann, wenn es D-Komponenten DK_1 und DK_2 von `D` gibt, zwischen denen eine Deskriptionsrelation vorliegt und die von DP_1 bzw. DP_2 respektive parametrisiert werden“. Doch, um die Gefahr zu entgehen, dass diese Einengung nicht ausreichend ist, und um die zwingende Verwendung von Deskriptionsrelationen zu vermeiden, soll diese Art der Präzisierung als optional angeboten werden.

7 Algorithmische Umsetzung

Die in 6.2 gelegte Grundlagen erlauben es nun eine algorithmische Lösung für das in der Aufgabenstellung formuliertes Problem herauszuarbeiten. Es müssen also die in 6.2 erarbeitete Vorgänge wie die Suche nach (für eine Situation) relevanten Deskriptionen, ihre Komplettierung, Spezialisierung sowie die Präzisierung der Situation algorithmisch umgesetzt und implementiert werden (wobei die Implementierung gemäß der im diesen Kapitel angegebenen Algorithmen erfolgt und im Kapitel 8 in einigen Aspekten vorgestellt wird).

7.1 Einschränkungen des algorithmischen Entwurfs und der Implementierung

Nicht alle Ideen, Entwürfe und Konzepte aus 6.2 wurden tatsächlich in die Praxis umgesetzt. Die Gründe dafür sind Folgende. Erstens, es wäre im sonstigen Falle jeglicher vernünftiger zeitlicher Rahmen für die Anfertigung dieser Arbeit gesprengt. Zweitens wurde in dieser Arbeit zum Ziel gesetzt, die formalen Grundlagen für ein flexibles und robustes Framework zur Erschließung kontextueller Informationen unter Einbeziehung des pragmatischen Wissens herauszuarbeiten, also sozusagen eine komplette und schlüssige Theorie zu entwerfen. Selbstverständlich gehört die Erstellung einer funktionsfähigen Implementierung gleichermaßen zu den erklärten Zielen dazu. Doch es sollte ausreichend sein, eine prototypische Lösung zu entwickeln, die dem Anliegen von SmartWeb gerecht ist und generell die Verwendungstauglichkeit des Papierentwurfs in einer reellen Anwendung beweist, wenngleich diese Lösung ausbaufähig sein mag.

Im Einzelnen ist auf sämtliche Spezialisierungen verzichtet worden, die in 6.5.6 mit 2 bis 7 nummeriert sind. Dies beeinträchtigt das Gesamtergebnis nicht, denn in Rahmen dieser Arbeit hatte man die Freiheit, die Deskriptionsontologie in Hinblick auf die vorhandene Basisontologie nach eigenem Ermessen zu gestalten, ungebunden an jegliche Vorgaben.

Auch auf die Deskriptions- und Situationsrelationen wurde verzichtet. Die Deskriptionsrelationen innerhalb einer Deskription legen fest, in welcher Beziehung zueinander einzelne Individuen einer Situation stehen sollen, damit sie diese

Deskription sozusagen auslösen. Offensichtlich erschien dies den Entwicklern von SmartWeb für ihre Zwecke unbedeutend, denn sie haben in einigen beispielhaft entworfenen Deskriptionen auf diese Verwendung von Deskriptionsrelationen verzichtet. Aus diesem Grund, aber auch weil die in 6.2 vorgeschlagene Behandlung von Deskriptionsrelationen ein kompliziertes Unternehmen ist, das vernünftige Grenzen dieser Arbeit überschreiten würde, fiel der Entschluss, sie bei der Implementierung außer Acht zu lassen.

7.2 Vorschlag für eine prototypische Deskriptionsontologie

Eine, wohl bemerkt sehr kleine und bescheidene Deskriptionsontologie ist in der Abbildung 7.1 dargestellt. Sie entspringt den diese Arbeit durchweg begleitenden Beispielen und dient erstens dem Testen der implementierten Routinen und zweitens der Veranschaulichung beschriebener Vorgänge (Spezialisierung, Präzisierung u.a.). Im Folgenden werden die Entitäten dieser Ontologie erläutert.

Die Abbildung 7.1 zeigt also eine kleine Deskriptionsontologie, in der drei Deskriptionen modelliert sind. Eine ist die aus der Abbildung 4.4 bereits bekannte `Environment-Path-Description`, in der eine allgemeine Fortbewegung als eine durch die vorherrschenden Wetterverhältnisse und die teilnehmende Strasse bestimmte Begebenheit modelliert wird. Außerdem ist in der Abbildung 7.1 die Rolle `Locomotor` (Fortbewegungsmittel) angedeutet, aber, weil sie für dieses Beispiel nicht von Bedeutung ist und aus Übersichtlichkeitsgründen ist sie in den Deskriptionen sozusagen inaktiv. Zwei weitere Deskription – `Rainy-FastPath-Description` und `Sunny-NicePath-Description` – sind Unterdeskriptionen der Ersten. Der Begriff Unterdeskription ist nicht trivial. Seine Bedeutung ist eng mit der im Abschnitt 6.5.6 diskutierter Spezialisierung von Deskriptionen verbunden. Jedoch wird in dieser Arbeit von seiner expliziten Definition abgesehen. Aus Intuition heraus sei vereinfachend an dieser Stelle mit Unterdeskription eine Subsumierungsrelation zwischen zwei Deskriptionen gemeint. So modelliert `Rainy-FastPath-Description` eine speziellere Begebenheit als `Environment-Path-Description`, nämlich eine Fortbewegung auf schnellerem Wege (eine Autobahn) bei schlechtem (regnerischem) Wetter. Als Gegenteil dazu modelliert `Sunny-NicePath-Description` eine

Fortbewegung auf einem schönerem Wege (eine Landstrasse) bei gutem (sonnigem) Wetter.

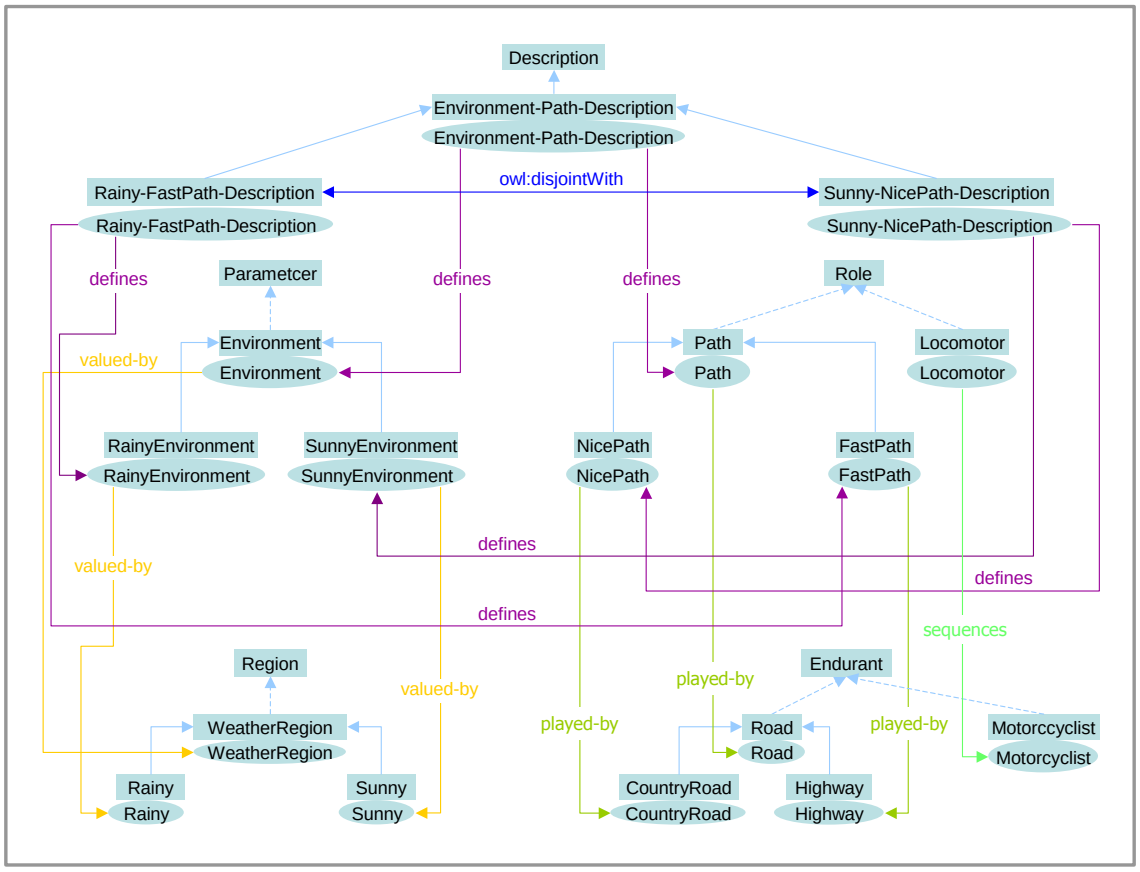


Abbildung 7.1: Eine Ontologie aus drei Deskriptionen: `Environment-Path-Description`, `Rainy-FastPath-Description` und `Sunny-NicePath-Description`. (*)

Mit einer wichtigen Anmerkung wird nun die vorgeschlagene Deskriptionsontologie kommentiert werden. Die dort modellierten Deskriptionen, so wie sie sind, implizieren eine motorisierte Fortbewegung: man würde ja nie einem Fußgänger oder Radfahrer empfehlen, auf einer Autobahn zu fahren. Dies einerseits, weil Deskriptionen in diesem Beispiel die Rolle `Locomotor` nicht berücksichtigen und andererseits, weil es keine Kontextquelle (bzw. Sensor) zum Erkunden des Fortbewegungsmittels des Benutzers vorgesehen ist. Eine solche „fest codierte“ Implikation ist selbstverständlich nicht im Sinne des Projekts SmartWeb, in dem, wie im Abschnitt 4.1.3 bereits erwähnt, zum Ziel gesetzt wurde, vermöge des pragmatischen Wissens domänenübergreifend und domänendetektierend zu sein. Nun das Beispiel aus der Abbildung 7.1 erfüllt seinen, rein demonstrativen Zweck. Sollten jedoch auch andere Arten der Fortbewegung berücksichtigt werden, dann gilt in jedem Fall die Rolle `Locomotor` „anzuschließen“.

Falls es keine Kontextquelle zur Ermittlung des Fortbewegungsmittel zur Verfügung steht, könnte sie beispielsweise in Form einer einmalig vorausgehender bzw. bei Bedarf gestellter Anfrage des Systems an den Benutzer modelliert werden. (Eine nicht wirklich zufriedenstellende Alternative wäre, die Spezifikation eines Fortbewegungsmittels in den dies verlangenden Aussagen dem Benutzer selbst zu überlassen.)

Eine bescheidene Deskriptionsontologie von insgesamt drei Deskriptionen kann natürlich den Ansprüchen einer marktfertigen Anwendung nicht gerecht werden, deren Wissensbasis eine oder sogar mehrere Domänen abdeckt und eine wirklich verwendbare Funktionalität anbietet. Vermutlich wäre dafür eine zwei- oder möglicherweise auch dreistellige Anzahl von Deskriptionen notwendig. Dies stellt einen enormen Aufwand dar, der im Rahmen dieser Arbeit leider nicht betrieben werden kann. Zudem bedarf die Modellierung einer Ontologie fundiertes computerlinguistisches Fachwissen.

7.3 Veranschaulichung der Problemlösung an Beispielen

Zur Veranschaulichung, wie mit Hilfe der modellierten Deskriptionsontologie die im Kapitel 3 formulierte Problemstellung gemeistert wird, betrachte man nun die als eine A-Box repräsentierte Benutzeraussage aus der Abbildung 4.4.

Beispiel 1

Die besagte A-Box beinhaltet unter anderem eine (leere) Instanz von `Road`. `Road` parametrisiert die Rolle `Path`, die eine Deskriptionskomponente von `Environment-Path-Description` ist. Da erstens nicht alle Komponenten dieser Deskription parametrisiert sind, zweitens die einzige unparametrisierte Komponente (`Environment`) mit `WeatherRegion` ein determinierbares PR-Target aufweist[62] und drittens, dadurch, dass es keine Verbindungsrelationen modelliert sind, die mit ihnen verbundene Bedingungen entfallen, erhält `Environment-Path-Description` den Status „selektiert“ und wird dadurch bei der Annahme relevant, dass selektierte Deskriptionen auch in Betracht gezogen werden, dass man sich also mit solchen auch begnügt. Somit ist an dieser Stelle bereits festgestellt worden, dass die A-Box zu der `Environment-Path-Description` passen würde, wenn ihr die Angaben zu

[62] Dass `WeatherRegion` determinierbar ist, geht natürlich nicht aus der Abbildung 7.1 hervor, sondern aus einer vorausgesetzter festgelegter Assoziierung von Kontextquellen mit Klassen der Basisontologie.

Wetterverhältnissen nicht fehlen würden. Demzufolge wird es versucht diese Angaben zu ermitteln. Nun sei es angenommen, die für das Wetter „zuständige“ Kontextquelle hat Regen gemeldet, d.h. es findet an dieser Stelle die Komplettierung der Situation durch `Rainy` statt. `Rainy` parametrisiert `Rainy-Environment`, das eine Deskriptionskomponente von `Rainy-FastPath-Description` ist, die als weitere Komponente `FastPath` hat. Letztendlich, weil `Rainy-FastPath-Description` eine Unterdeskription von `Environment-Path-Description` ist und `Path FastPath` subsumiert, liegt hiermit eine Präzisierung vor. Folglich darf die Instanz von `Road` in der eingegebener A-Box als eine Instanz von das `FastPath` parametrisierende `Highway` präzisiert werden. In diesem Beispiel findet also eine Präzisierung nach 6.7.1 statt.

Beispiel 2

Im Gegensatz zu dem obigen Beispiel betrachte man nun den Fall, dass selektierte, aber nicht erfüllte Deskriptionen ungeachtet bleiben und nehme an, die A-Box beinhalte eine (leere) Instanz von `WeatherRegion`. Dann verfährt man wie im vorigen Beispiel bis das Zwischenergebnis erreicht ist, dass die Deskription `Environment-Path-Description` den Status „erfüllt“ erhält und sich somit als relevant erweist. Der nächste sinnvolle Schritt an dieser Stelle ist es zu versuchen, diese Deskription zu spezialisieren. Als eine spezialisierende Klasse kommt in diesem Fall entweder `Rainy` oder `Sunny` in Betracht, denn sie Unterklassen des D-Partizipanten `WeatherRegion` von der relevanten Deskription sind und alle übrige Bedingungen für die Spezialisierung (nach jeder Definition) erfüllt sind. Wenn man nun annimmt, dass über die entsprechende Kontextquelle `Rainy` ermittelt wurde, nimmt dieses Beispiel weiterhin denselben Verlauf wie das Beispiel 1 und endet mit der Präzisierung mit `Highway`. In diesem Beispiel erfolgt also die Präzisierung nach 6.7.2.

7.4 Schema der Vorgehensweise

Den obigen Beispielen ist ein gewisses Schema abzuleiten, die zur Lösung des Problems führt. Dieses Schema ist in der Abbildung 7.2 dargestellt. Man sucht also für die eingegebene A-Box zuerst nach einer (evtl. mehreren) relevanten Deskription. Danach, falls diese selektiert aber nicht erfüllt ist, werden die benötigten

Kontextinformationen ermittelt, wodurch die Komplettierung der relevanten Deskription erfolgt. Danach, oder falls die relevante Deskription erfüllt ist, und es somit nichts zu komplettieren gibt, kann man versuchen ihre Spezialisierung zu finden und dann die benötigten Kontextinformationen zu ermitteln (im Beispiel 2 aus 7.3 traf die letztere Option zu). Schließlich kann ein Versuch folgen die Benutzeraussage zu präzisieren.

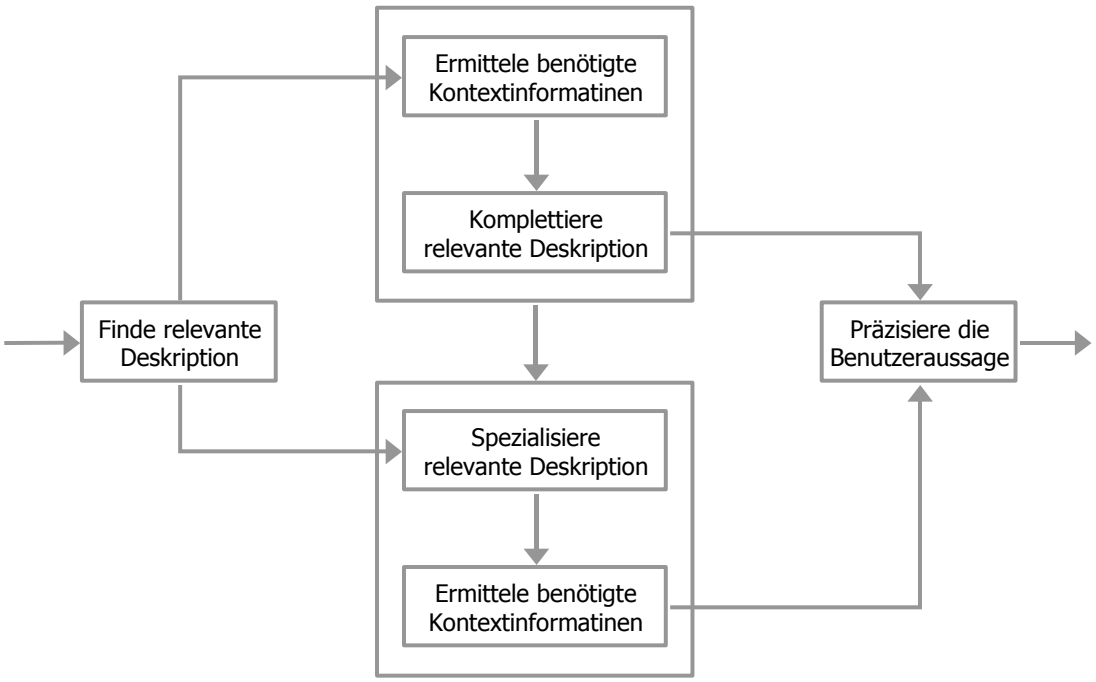


Abbildung 7.2: Verallgemeinerte Vorgehensweise bei der Problemlösung, schematisch dargestellt.

7.5 Optimierungskriterien des algorithmischen Entwurfs

Beim Entwickeln der Algorithmen, die in den nächsten Abschnitten dargestellt werden, wurde es zum Ziel gesetzt, nicht nur die erwünschte Funktionalität umzusetzen, sondern auch die Performanz der Algorithmen im Auge zu behalten. Es wurde vor allem versucht, die Zeitkomplexität möglichst gering zu halten, und zwar in erster Linie bezüglich bestimmter Operationen, die in diesem Sinne erfahrungsgemäß als kritisch angesehen werden. Es handelt sich dabei um folgende Operationen:

- Erstens stellt die Abfrage von Kontextquellen im Allgemeinen einen unbekannten Zeitfaktor dar. Bei den momentan verfügbaren Kontextquellen, die auf Web Services zugreifen, liegt die Dauer der Verbindungsaufbau im Bereich von etwa einer Sekunde. Für einige Echtzeitanwendungen wäre es deswegen

unzumutbar, öfters auf solche Kontextquellen zuzugreifen. Außerdem sind sie oft kostenpflichtig.

- Zweitens kann die Inferenz in großen Ontologien sehr zeitaufwändig sein. Die Erfahrung hat gezeigt, dass die Dauer einer Anfrage an die Ontologie von vernachlässigbar gering bis unmessbar[63] lang sein kann. Dies hängt von der Größe und Expressivität der Ontologie, aber auch von der Expressivität des verwendeten Reasoner und der Inferenzdienste, die eine Anfrage benötigt, ab. Außerdem fällt der enorme Speicherverbrauch negativ auf, den einige Anfragen auslösen.

Um die Performanz bezüglich dieser als kritisch eingestuften Operationen zu erhöhen, wurde bei den unten aufgeführten Algorithmen von der folgenden Tatsache profitiert. Die Abfrage von Kontextquellen kann enorm beschleunigt werden, wenn mehrere Kontextquellen, die auf dasselbe Web Service zugreifen, dies parallel statt sequentiell tun. Deswegen sind die Algorithmen so gestaltet, dass alle abzufragende Kontextquellen zunächst ermittelt und erst dann abgefragt werden.

Um die Performanz bezüglich des Umgangs mit den Ontologien algorithmisch zu steigern, werden Datenstrukturen verwendet, in denen die inferierte Sachverhalte – zum Beispiel, dass eine Deskription `X` eine D-Komponente `Y` hat, die von `Z` parametrisiert werden kann – gespeichert werden. Die Implementierung solcher Datenstrukturen wird in dieser Arbeit nicht thematisiert. Ihre Zeit- und Speicherkomplexität ist mit hoher Wahrscheinlichkeit verbesserungsfähig, jedoch verglichen mit den kritischen Operationen vernachlässigbar, was experimentell bestätigt wurde.

7.6 Algorithmus zur Bestimmung relevanter Deskriptionen und der für sie benötigten Kontextquellen

Folgender Algorithmus bestimmt die für eine Situation relevanten Deskriptionen und, falls selektierte Deskriptionen zugelassen sind, Kontextquellen, die abgefragt werden müssen, um alle determinierbare D-Partizipanten der relevanten Deskriptionen zu ermitteln, die in der Situation fehlen[64].

[63] Die Wartezeit war nach einigen Minuten im zweistelligen Bereich abgebrochen.
[64] Damit wird die Optionalität von D-Partizipanten einer D-Komponente nicht berücksichtigt. Das bedeutet z.B., dass wenn eine D-Komponente von zwei determinierbaren D-Partizipanten

7.6.1 Ein- und Ausgabe des Algorithmus

Die Eingabe für den Algorithmus ist eine Benutzeraussage, die als eine A-Box der Basisontologie repräsentiert ist; sie wird im Folgenden „Situation“ genannt.

Die Ausgabe des Algorithmus ist eine Funktion `m` und zwei Mengen `RK` und `DS`, und zwar:

- `m` – eine Funktion, die einer Deskription bzw. einem Tupel aus einer Deskription und seiner D-Komponente bzw. einer Tripel aus einer Deskription, seiner D-Komponente und einem D-Partizipant eine Markierung zuweist. So zum Beispiel `m(D,I,N)=„vorhanden“` bedeutet, dass die Situation die D-Komponente `I` der Deskription `D` parametrisiert, weil dort (in der Situation) eine Instanz der Klasse des D-Partizipanten `U` vorkommt. Wenn ein Strich einen Argument ersetzt, dann repräsentiert dies bei einer Zuweisung den Allquantor. Zum Beispiel bedeutet `m(_,V):=„parametrisiert“`, dass für jede Deskription, die D-Komponente `V` definiert, diese D-Komponente die Markierung „parametrisiert“ erhält. Bei sonstigem repräsentiert ein Strich den Existenzquantor. Zum Beispiel ist die Bedingung `wenn m(_,V)=„parametrisiert“` genau dann erfüllt, wenn es eine Deskription D gibt, für die `m(D,V)=„parametrisiert“` gilt. (Ähnliches gilt auch für die unten angegebene Mengen.)

- `RK` – eine Menge von Quadrupeln:

  (Kontextquelle, Deskription, D-Komponente, D-Partizipant)

- Zum Beispiel repräsentiert das Quadrupel `(Q,D,I,N)` folgenden Sachverhalt: die Kontextquelle `Q` determiniert den D-Partizipanten `N`, der die D-Komponente `I` der Deskription `D` parametrisiert.

---

parametrisierbar ist und einer davon in der Situation vorhanden ist, wird es trotzdem versucht, den zweiten D-Partizipanten zu determinieren, obwohl die D-Komponente auch ohne diesen als parametrisiert gilt. Derartig wird es deswegen verfahren, weil die optionale Information sich in diesem Fall als nützlich erweisen kann: sie kann eventuell eine Spezialisierung und Präzisierung ermöglichen.

- `DS` – eine Menge von Quintupeln:

  (Deskription, D-Partizipant, D-Partizipant, Klasse, Markierung)

  Zum Beispiel repräsentiert das Quadrupel `(D,N,K,C,"vorhanden")` folgenden Sachverhalt: dem D-Partizipanten `N` der Deskription `D` entspricht in der Situation das Individuum `K` der Klasse `C`, das dort von Beginn an vorhanden ist (im Gegensatz zu einer nachträglichen Determinierung mittels einer Kontextquelle).

7.6.2 Darstellung des Algorithmus

Vereinfacht dargestellt, traversiert der Algorithmus – er ist in der Abbildung 7.4 aufgeführt – mehrfach den in der Abbildung 7.3 exemplarisch gezeichneten Graph, markiert dabei in internen Datenstrukturen einzelne (Tupeln von) Entitäten der Deskriptionsontologie und stellt bestimmte Bezüge zwischen Kontextquellen, Individuen der Situation und den D-Partizipanten.

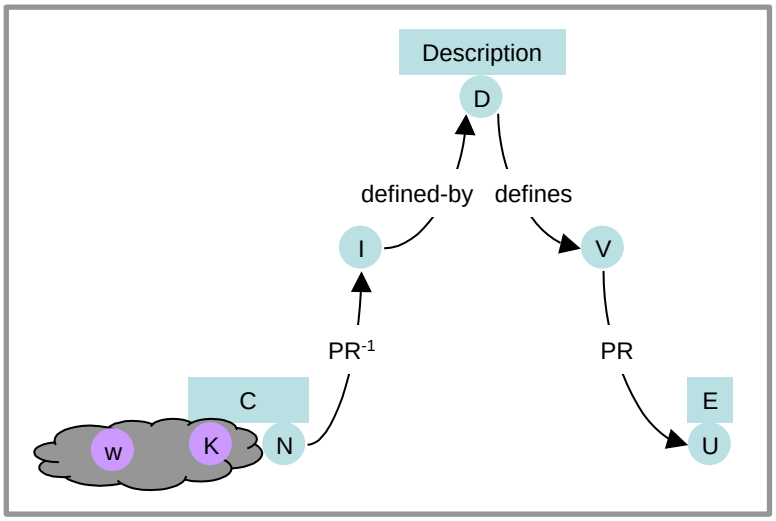


Abbildung 7.3: Exemplarisch dargestellter Graph, der während des Algorithmusablaufs mehrfach traversiert wird. Die Erklärung von Bezeichnungen ist im Text gegeben. (*)

Man fängt mit einem beliebigen Individuum `K` der Situation an, in der Hoffnung, es würde eine D-Komponente einer Deskription parametrisieren. Um dies zu prüfen, betrachtet man jede (basisontologische) Klasse von `K` und sucht in der

Deskriptionsontologie Individuen dieser Klassen. Werden welche gefunden – z.B. N – dann hat man damit festgestellt, dass K in der Situation den D-Partizipanten N repräsentiert. Danach bedarf es keiner großer Kunst über inverse Partizipierungsrelationen (PR^{-1}) bzw. die defined-by Relation diejenige Deskriptionen zu finden, in denen N partizipiert, z.B. D. Ähnlich werden danach alle D-Komponenten und D-Partizipanten von D aufgesucht; in diesem Beispiel sind es exemplarisch V und U. Nun stellt sich die Frage, ob es in der Situation Individuen vorhanden sind, die den D-Partizipanten U repräsentieren, d.h. solche Instanzen, die zur expliziten Klasse von U gehören. Ist es für ein Individuum W der Fall (Zeile 16), wird das zuvor in der Zeile 13 mit „nicht parametrisierbar" markierte Individuum V, nun mit „parametrisiert" markiert (Zeile 25); U erhält den Status „vorhanden" (Zeile 26); auch in DS wird der entdeckte Sachverhalt gespeichert (Zeile 27). Sonst, wenn selektierte Deskriptionen zugelassen sind, werden alle Kontextquellen, die U determinieren, als möglicherweise relevant eingestuft (d.h. zu der Menge MRK hinzugefügt, Zeile 20). Gibt es solche Kontextquellen, wird V mit „parametrisierbar" markiert, falls es noch nicht parametrisiert ist (Zeilen 21, 22); entsprechend erhält U den Status „determinierbar" (Zeile 23).

Sind in der Deskriptionsontologie alle D-Partizipanten von V behandelt worden und der anfängliche nicht parametrisierbare Status von V (Zeile 13) dabei unverändert blieb, dann ist die aktuelle Deskription D irrelevant[65] (Zeile 32). Es werden also diejenige Kontextquellen ausgeschlossen, die für D als bedeutend befunden wurden (Zeile 33), und man startet das Verfahren mit dem nächsten Individuum der Situation (Zeile 34).

Wurde der anfängliche Status von V nach Betrachtung aller seinen D-Partizipanten in „parametrisierbar" umgewandelt (Zeile 35), wird die anfänglich als erfüllt (Zeile 11) vermutete aktuelle Deskription D zu „selektiert" umgeändert (Zeile 36) bzw., wenn keine selektierte Deskriptionen zugelassen sind, zu „irrelevant" degradiert (Zeilen 38). Im letzteren Fall werden wiederum diejenige Kontextquellen ausgeschlossen, die für D als relevant eingestuft wurden (Zeile 39), und es wird anschließend mit dem nächsten Individuum der Situation gestartet (Zeile 40).

[65] Es wird vorerst die Stelle des Algorithmus, an der die sogenannten prädiktierfähigen Deskriptionen in Betracht gezogen werden (Zeilen 27, 28), außer Acht gelassen. Dasselbe gilt für die Zeile 35.

Schließlich, nach dem die aktuelle Deskription D auf vorgestellte Weise abgearbeitet wurde, ist die Menge der für sie möglicherweise relevanten Kontextquellen genau dann leer, wenn D erfüllt oder irrelevant ist; sollte D selektiert sein, beinhaltet diese Menge die für D relevanten Kontextquellen, die an dieser Stelle der Menge von mit Sicherheit relevanten Kontextquellen (RK) hinzugefügt (Zeile 41) werden.

INPUT: S – *Eine Situation als eine A-Box der Basisontologie*
OUTPUT: m – *eine Funktion, die (Tupeln von) Entitäten der Deskriptionsontologie mit einer Markierung verknüpft*
OUTPUT: RK **–** *Quadrupelmenge: (Kontextquelle, Deskription, D-Komponente, D-Partizipant)*
OUTPUT: DS – *Quintupelmenge: (Deskription, D-Partizipant, D-Partizipant, Klasse, Markierung)*

```
1    m := ∅
2    RK := ∅
3    MRK := ∅
4    DS := ∅
5    Für jedes Individuum K in S
6     Für jede (basisontologische) evtl. explizite Klasse C von K
7      Für jedes Individuum N der Definitionsontologie, das explizit zu C gehört
8        Für jedes Individuum I mit  PR^-1(N,I)
9        Für jedes Individuum D mit  defined-by(I,D)
10         Wenn  m(D) = null
11          m(D) := „erfüllt"
12          Für jedes Individuum V mit  m(_,V) = null ∧ defines(D,V)
13           m(D,V) := „nicht parametrisierbar"
14            Für jedes Individuum U mit PR(V,U)
15             m(D,V,U) := „nicht vorhanden"
16             Wenn in S kein Individuum W vorkommt, das zu der expliziten
               Klasse E von U gehört
17               Wenn selektierte Deskriptionen zugelassen sind
18                Für jede Kontextquelle Q
19                 Wenn  E von Q determinierbar ist
20                  MRK := MRK ∪ (Q,D,V,U)
21                   Wenn  m(_,V) ≠ „parametrisiert"
22                    m(D,V) := „parametrisierbar"
23                   m(D,V,U) := „determinierbar"
24               Sonst
25                m(D,V) := „parametrisiert"
26                m(D,V,U) := „vorhanden"
27                DS := DS ∪ (D,U,W,E,„vorhanden")
28             Wenn  m(_,V) = „nicht parametrisierbar"
29              Wenn prädiktierfähige Deskriptionen zugelassen sind
30               m(D) := „prädiktierfähig"
31              Sonst
32               m(D) := „irrelevant"
```

```
33                  MRK := ∅
34                  Starte mit dem nächsten Individuum in S
35              Wenn m(_,V) = „parametrisierbar“
36                m(D) := „selektiert“
37                Wenn selektierte Deskriptionen nicht zugelassen sind und
                  m(D) ≠ „prädiktierfähig“
38                  m(D) := „irrelevant“
39                  MRK := ∅
40                  Starte mit dem nächsten Individuum in S
41        RK := RK ∪ MRK
42        MRK := ∅
43  Return m, RK, DS
```

Abbildung 7.4: Algorithmus zur Bestimmung relevanter Deskriptionen und der für sie benötigten Kontexte.

Wenn der Algorithmus ausgeführt ist, dann gilt Folgendes. Jede Deskription, auf die eingegangen wurde, erhält den Status „irrelevant“, „relevant“, „selektiert“, „erfüllt“ oder „prädiktierfähig“ (Letzteres wird in 7.10 erläutert). Jede D-Komponente einer solchen Deskription erhält den Status „parametrisiert“, „parametrisierbar“ oder „nicht parametrisierbar“; jeder D-Partizipant erhält den Status „vorhanden“, „nicht vorhanden“ oder „determinierbar“.

7.6.3 Anmerkungen zu der Klassenzugehörigkeitsbestimmung

Abschließend bedarf es einiger Erläuterungen, weshalb im Algorithmus an manchen Stellen die explizite Klassenzugehörigkeit verlangt und an den anderen – die im Sinne der Beschreibungslogik konventionelle[66] Auffassung dieses Begriffs verwendet wird.

In der Zeile 6 wird es nach jeder Klasse `C` gesucht, der wahlweise entweder explizit oder konventionell das Individuum `K` aus der Situation gehört. Dies ist damit begründet, dass im letzteren Fall (bei der konventionellen Klassenzugehörigkeit) Redundanzen entstehen können. Man betrachte die Abbildung 7.1 und nehme beispielsweise an, die Situation enthält Instanzen von `Rainy` und `Highway`. Damit wird nicht nur `Rainy-Fast-Description`, sondern auch die `Environment-Path-Description` erfüllt, denn die besagten Instanzen aus konventioneller Sicht ebenfalls Individuen von `WeatherRegion` bzw. `Road` sind. Solches Verhalten ist nicht fehlerhaft oder

[66] Mit „konventionell“ ist hier Folgendes gemeint: ein Individuum `K` gehört zu einer Klasse `C`, wenn `T |– C(K)`.

unerwünscht, sondern eben redundant, da es zusätzlichen Aufwand produziert, entweder `Environment-Path-Description` nachträglich als irrelevant auszusortieren, oder sie, genau wie die sonstigen relevanten Deskriptionen, den folgendenden Prozeduren (Komplettierung, Spezialisierung und Präzisierung) zu unterziehen.

Wenn in diesem Fall nur die expliziten Klassen betrachtet werden, dann wäre zwar die eventuelle Redundanz behoben, jedoch hätte es folgenden Nachteil. Man nehme an, in der Situation kommen Individuen vor, die explizit den Unterklassen von `Rainy` bzw. `Highway` gehören. Damit gehören sie aber explizit `Rainy` bzw. `Highway` nicht an und selektieren somit `Rainy-Fast-Description` nicht. Dies stellt möglicherweise unerwünschtes Verhalten dar.[67] Deshalb ist die Art der Klassenzugehörigkeit zur Wahl gestellt.

In der Zeile 7 wird jeder D-Partizipant `N` aus der Deskriptionsontologie gesucht, der explizit zur Klasse `C` gehört. Im Falle der konventionellen Klassenzugehörigkeit würde man hier auf folgendes Problem stoßen. Man betrachte wiederum die Abbildung 7.1 und nehme an, die Situation enthält explizite Instanzen von `WeatherReagion` und `Road`. An einer Stelle im Laufe des Algorithmus wäre dann `C`, als explizite Klasse eines Individuums aus der Situation, gleich `WeatherRegion`. Aus konventioneller Sicht käme dann für `N` eine (deskriptionsontologische) Instanz von `Rainy` als zugehörig zu `WeatherRegion` (bzw. im Sinne der Abbildung 7.3 zu `C`) in Betracht. Ähnliches würde mit der (deskriptionsontologischer) Instanz von `Road` geschehen, d.h. die `Rainy-Fast-Description` Deskription würde dann als relevant resultieren, obwohl es für die angenommene Situation offensichtlich irrtümlich wäre. Deswegen kommt an dieser Stelle nur die explizite Klassenzugehörigkeit in Betracht.

Schließlich wurde die Klassenzugehörigkeit in der Zeile 16 aus ähnlichen Überlegungen wie vorhin festgelegt. Um eine Wiederholung zu vermeiden, werden diese hier nicht geschildert.

[67] Ob es tatsächlich unerwünscht ist, hängt von einer konkreten Anwendung und der Beschaffenheit der Ontologien ab.

7.7 Komplettierungsalgorithmus

Der Komplettierungsalgorithmus baut auf den Ergebnissen des Algorithmus aus 7.6 auf. Für jede relevante Deskription werden Kontextinformationen der zuvor als bedeutsam befundenen Kontextquellen ermittelt, zwischengespeichert und zu der Situation hinzugefügt. Je nach dem, ob es für eine selektierte Deskription gelingt, verliert sie entweder ihren Status oder erhält den Status „selektiert".

Es werden also alle relevanten, d.h. eventuell auch die bereits erfüllten Deskriptionen betrachtet. Für die Letzteren werden die optionalen D-Partizipanten, die in der Situation fehlen, ermittelt. Ihren Status verliert aber eine erfüllte Deskription nicht.

7.7.1 Ein- und Ausgabe des Komplettierungsalgorithmus

Als Eingabe erwartet der Komplettierungsalgorithmus die Situation (als eine A-Box repräsentierte Benutzeraussage) und die zuvor bestimmten Funktion `m` und Mengen `RK` und `DS`.

Ausgegeben werden die aktualisierten Eingaben und die neue Menge `IQ`. Sie beinhaltet alle Tupeln aus einer Kontextquelle und der von ihr ermittelten Kontextinformation.

7.7.2 Darstellung des Komplettierungsalgorithmus

Bevor der Algorithmus vorgestellt wird, empfiehlt es sich das Beispiel aus der Abbildung 7.5 zu betrachten, welches hilfreich sein wird den Algorithmus nachzuvollziehen. In der Abbildung 7.5 ist ein Beispiel für den Sachverhalt veranschaulicht, bei dem die Kontextquelle `Q` den D-Partizipanten `Z` determiniert, weil `Z` Instanz einer Unterklasse von der mit `Q` assoziierten Klasse `WeatherRegion` ist. Es gilt, dass jede Unterklasse von `WeatherReagion` von `Q` determinierbar ist. Jedoch ist es möglich, dass eine konkrete Kontextinformation den D-Partizipanten `Z` nicht ermittelt hat – nämlich genau dann, wenn diese Kontextinformation sich nicht als eine Instanz[68] von `Rainy` (auch Unterklasse von `Rainy`) repräsentieren lässt. Es muss also beachtet

[68] Genauer ausgedrückt – als eine A-Box, in der eine Instanz von `Rainy` vorkommt.

werden, ob eine Kontextinformation einer Z determinierender Quelle Z tatsächlich hat ermitteln können.

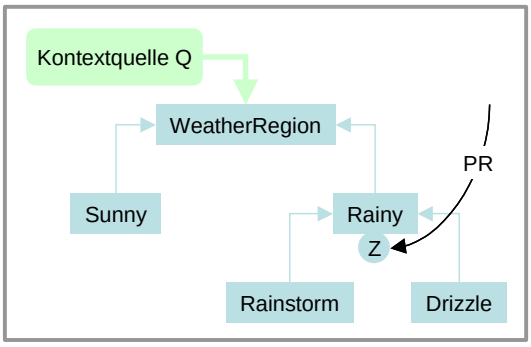


Abbildung 7.5: Die Veranschaulichung des Unterschiedes zwischen Determinierbarkeit und Ermittlung einer ontologischen Klasse (bzw. einer ihrer Instanz), dargestellt an einem Beispiel. (*)

Nun zum Komplettierungsalgorithmus: er fängt damit an, dass für eine relevante Deskription D jede Kontextquelle Q betrachtet wird, die für sie aus vorigem Algorithmus für notwendig befunden wurde (Abbildung 7.6, Zeile 3). Falls nicht bereits geschehen, wird sie abgefragt (Zeilen 4, 5). Die von ihr ermittelte Kontextinformation wird in IQ gespeichert (Zeile 6). Konnte diese Kontextinformation aus irgendeinem Grund nicht ermittelt werden, so erhält jeder von Q determinierbare D-Partizipant von D den Status „nicht determinierbar“ (Zeile 17). Andernfalls erhält jeder solcher D-Partizipant den Status „determiniert“; jede D-Komponente, die von ihm parametrisierbar ist, wird mit „parametrisiert“ markiert (Zeilen 9-11) und es geschieht Folgendes. Man betrachtet die explizite Klasse des D-Partizipanten Z und ihre Unterklassen. Wenn eine dieser Klassen von der Kontextinformation nicht nur determiniert, sondern auch ermittelt wurde, wird die Kontextinformation, repräsentiert als eine Instanz dieser Klasse, der Situation hinzugefügt (Zeile 15).

Nach dem Versuch alle D-Partizipanten von D zu ermitteln, wird es überprüft, ob alle D-Komponenten der Deskription D parametrisiert sind (Zeilen 18-25). Ist es für eine D-Komponente nicht der Fall (Zeile 19), so wird sie mit „nicht parametrisierbar“ markiert (Zeile 20) und D erhält den Status „irrelevant“ (Zeile 24). (Die Markierung von D mit „prädiktierfähig“ wird in 7.10 erläutert.) Schließlich, wenn nach diesen Betrachtungen D die Markierung „selektiert“ nicht verloren hat (Zeile 26), dann geschah es nur

deswegen, weil alle D-Komponenten von D parametrisiert werden konnten. Folglich erhält D den Status „erfüllt“ (Zeile 27).

INPUT: S – *Eine Situation als eine A-Box der Basisontologie*
INPUT: m – *eine Funktion, die (Tupeln von) Entitäten der Deskriptionsontologie mit einer Markierung verknüpft*
INPUT: RK – *Quadrupelmenge: (Kontextquelle, Deskription, D-Komponente, D-Partizipant)*
INPUT: DS – *Quintupelmenge: (Deskription, D-Partizipant, D-Partizipant, Klasse, Markierung)*
OUTPUT: S *(aktualisiert)*
OUTPUT: m *(aktualisiert)*
OUTPUT: DS *(aktualisiert)*
OUTPUT: IQ – *Tupelmenge (Kontextquelle, Kontextinformation)*

```
1   IQ := ∅
2   Für jede Deskription D mit m(D) = „relevant“
3     Für jede Kontextquelle Q, mit (Q,D,_,_) ∈ RK
4       Wenn (Q,_) ∉ IQ
5       Sei F die von Q ermittelte Kontextinformation
6       IQ := IQ ∪ (Q,F)
7       Für jedes Z mit (Q,D,_,Z) ∈ RK
8         Wenn F ≠ null
9           m(D,_,Z) := „determiniert“
10          Für jedes V mit m(D,V,Z) ≠ null
11            m(D,V) := „parametrisiert“
12          Für jede Klasse E, die mit F ermittelt wurde
13            Wenn E (unechte) Unterklasse der Klasse von Z ist
14              Sei I die Repräsentation von F als eine Instanz von E
15              S := S ∪ I
16              DS := DS ∪ (D,Z,I,E,“determiniert”)
17        Sonst m(D,_,Z) := „nicht determinierbar“
18    Für jedes V mit m(D,V) ≠ null
19      Wenn m(D,V) ≠ „parametrisiert“
20        m(D,V) := „nicht parametrisierbar“
21        Wenn prädiktierfähige Deskriptionen zugelassen sind
22          m(D) := „prädiktierfähig“
23        Sonst
24          m(D) := „irrelevant“
25          Starte mit der nächsten relevanten Deskription
26    Wenn m(D) = „selektiert“
27      m(D) = „erfüllt“
28  Return: S, m, DS, IQ
```

Abbildung 7.6: Der Komplettierungsalgorithmus.

7.8 Spezialisierungsalgorithmus gemäß der Spezialisierungsdefinition 1.II

In diesem Abschnitt wird ein Spezialisierungsalgorithmus gemäß der Spezialisierungsdefinition 1.II (siehe 6.5.6) vorgestellt. Seine Aufgabe ist, für jede erfüllte Deskription die sie spezialisierenden Klassen und die Deskriptionen, in denen diese Klassen partizipieren, zu bestimmen und eventuell die Situation mit einem ermittelten Individuum der spezialisierenden Klasse (bzw. mit einer A-Box, die ein solches Individuum enthält) zu ergänzen. Die Voraussetzung für dieses Algorithmus ist, dass die relevanten Deskriptionen bereits bestimmt wurden, als der Algorithmus aus 7.6 ausgeführt worden ist. Will man von der Komplettierung Gebrauch machen, dann soll dies vor der Spezialisierung erfolgen.

7.8.1 Ein- und Ausgabe des Spezialisierungsalgorithmus

Als Eingabe erwartet der Spezialisierungsalgorithmus eine Situation (als eine A-Box repräsentierte Benutzeraussage) und die zuvor bestimmte Funktion `m` und Mengen `DS` und `IQ`.

Ausgegeben werden die aktualisierten Situation und Menge `IQ`, sowie die neue Menge `ID` von Quintupeln. In einer Quintupel `(D,Z,C,D',M)` wird dem D-Partizipanten `Z` von der Deskription `D` eine spezialisierende Klasse `C`, eine Deskription `D'` und eine Menge `M` von Kontextquellen zugeordnet. Die Kontextquellen aus `M` determinieren `C` und gegebenenfalls auch `Z`.

7.8.2 Darstellung des Spezialisierungsalgorithmus

Der Spezialisierungsalgorithmus lässt sich in drei Teile gliedern. Im ersten Teil (Abbildung 7.8, Zeilen 1-17) werden solche Sachverhalte aufgesucht, die die Spezialisierungsdefinition 1.II verlangt. Diese Sachverhalte sind exemplarisch in der Abbildung 7.7 dargestellt[69]. Es werden für jeden determinierten oder von vornherein vorhandenen D-Partizipanten `Z` jeder erfüllten Deskription `D` alle Individuen betrachtet, die in der Situation diesem D-Partizipanten `Z` entsprechen (d.h. dieselbe D-Komponente

[69] Die Abbildung 6.8 (oben rechts), die ebenfalls Spezialisierung nach 1.II skizziert, entspricht dieser Abbildung, blendet jedoch einzelne Details aus.

parametrisieren) (Zeilen 3-5). Für ein solches Individuum ist jede seine explizite Klasse K insofern interessant (Zeilen 5-6), weil die Menge ihrer Unterklassen eventuell solche beinhaltet, die D spezialisieren. Somit wird jede Unterklasse C von K betrachtet (Zeile 7) und auf die Erfüllung von Bedingungen aus 1.II (inklusive allgemeine notwendige und optionale Bedingungen außer c) und d)) untersucht. So fordert man in der Zeile 8, dass C in D nicht partizipiert; in den Zeilen 9 bis 12 stellt man sicher, dass es Kontextquellen geben, die Z und C determinieren; und in den Zeilen 14 bis 16 werden alle Deskriptionen D′ aufgesucht, in denen C partizipiert, und die eventuell ungleich D und bezüglich der Situation nicht erfüllt sind. Waren diese Schritte erfolgreich, so wird der festgestellte Sachverhalt in der Menge ID gespeichert (Zeile 17).

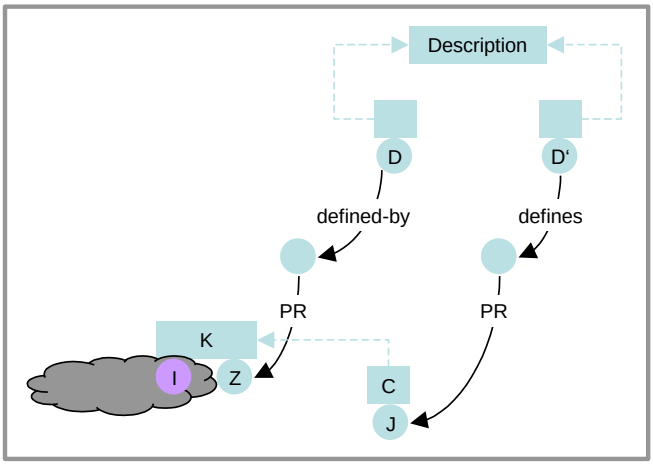


Abbildung 7.7: Der von der Spezialisierungsdefinition 1.II geforderte Sachverhalt (exemplarisch). Dieser Sachverhalt wird im ersten Teil des Spezialisierungsalgorithmus aufgesucht und festgehalten. (*)

Ebenso werden alle Kontextquellen gespeichert, die C und gegebenenfalls auch Z determinieren (Zeilen 9 und 11). Im zweiten Teil des Spezialisierungsalgorithmus (Zeilen 18-21) wird jede solche Kontextquelle abgefragt, falls es zuvor (während der Komplettierung) nicht bereits erfolgt ist. Die Kontextinformationen werden in IQ gespeichert.

Im dritten Teil des Algorithmus (Zeile 22-31) werden wiederum Sachverhalte wie in der Abbildung 7.7 betrachtet. Für einen solchen Sachverhalt, bestehend aus der Deskription D, ihrem D-Partizipanten Z und der spezialisierender Klasse C, werden die vorhin ermittelten Kontextinformationen derjenigen Kontextquellen untersucht, die C

determinieren (ab den Zeilen 25, 26). Aus der Überlegung heraus, die in der Abbildung 7.5 dargestellt ist, wird es geprüft, ob eine solche Kontextinformation F die Klasse C tatsächlich ermittelt hat (Zeilen 27, 28). Nun werden alle Instanzen I betrachtet (Zeile 29), die in der Situation den D-Partizipanten Z repräsentieren. Es wurde zwar zuvor festgestellt, dass eine ihrer expliziten Klassen mit der Klasse K von Z übereinstimmt, doch es kann theoretisch vorkommen, dass I explizit zu weiteren Klassen gehört, die Unterklassen von C sind. In diesem Fall würde ja F, repräsentiert als die Instanz I von C, I nicht spezialisieren, sondern möglicherweise sogar bezüglich rein expliziter Klassenzugehörigkeit generalisieren. Nur wenn dies nicht vorliegt, wird der Situation F, repräsentiert als die Instanz I von C, hinzugefügt (Zeilen 30, 31).

INPUT: S – *Eine Situation als eine A-Box der Basisontologie*
INPUT: m – *eine Funktion, die (Tupeln von) Entitäten der Deskriptionsontologie mit einer Markierung verknüpft*
INPUT : DS **–** *Quintupelmenge : (Deskription, D-Partizipant, D-Partizipant, Klasse, Markierung)*
INPUT: IQ – *Tupelmenge (Kontextquelle, Kontextinformation)*
OUTPUT: S *(aktualisiert)*
OUTPUT: IQ *(aktualisiert)*
OUTPUT: ID – *Quintupelmenge: (Deskription, D-Partizipant, spezialisierende Klasse, Deskription, Menge von Kontextquellen)*

```
1   M := ∅
2   ID := ∅
3   Für jede Deskription D mit m(D) = „erfüllt"
4    Für jedes Z mit m(D,_,Z) = „vorhanden" ∨ m(D,_,Z) =
     „determiniert"
5      H := { E|(D,Z,I,E,_) ∈ DS }
6      Für jede Klasse K mit K ∈ H
7        Für jede determinierbare, basisontologische, echte Unterklasse C von K
8          Wenn ¬( Bedingung a) verlangt ∧ es gibt eine explizite Instanz X
           von C mit m(D,_,X) ≠ null )
9            Sei M eine Menge aller Kontextquellen, die C determinieren
10           Wenn 1.II.d) verlangt ist
11             M := M ∩ ( Menge aller Kontextquellen, die Z determinieren )
12           Wenn M ≠ ∅
13             Für jede explizite deskriptionsontologische Instanz J von C
14               Für jedes D' mit ∃Y.( PR⁻¹(J,Y) ∧ defined-by(Y,D') )
15                 Wenn ¬( Bedingung b) verlangt ∧ m(D') = „erfüllt" )
16                   Wenn ¬( Bedingung 1.II.c) verlangt ∧ D = D' )
17                     ID := ID ∪ (D,Z,C,D',M)
18  Für jedes Q ∈ M
19   Wenn es kein X gibt mit (Q,X) ∈ IQ
```

```
20      Sei F die von Q ermittelte Kontextinformation
21      IQ := IQ ∪ (Q,F)
22    Für jede Deskription D mit m(D) = „erfüllt“
23      Für jedes Z mit
        m(D,_,Z) = „vorhanden“ ∨ m(D,_,Z) = „determiniert“
24        Für jedes C mit (D,Z,C,_,_) ∈ ID
25          Für jedes N mit (D,Z,C,_,N) ∈ ID
26            Für jede Kontextquelle Q ∈ N
27              Wenn es eine Kontextinformation F gibt mit (Q,F) ∈ IQ
28                Wenn F als eine Instanz der Klasse C repräsentierbar
29                  Für jedes I mit (D,Z,I,_,_) ∈ DS
30                    Wenn ¬∃E.( (D,Z,I,E,_) ∈ DS ∧ E ⊂ C )
31                      Füge S die Repräsentation von F als Instanz I der Klasse C
                        hinzu
32    Return: S, DS, IQ, ID
```

Abbildung 7.8: Der Spezialisierungsalgorithmus gemäß der Spezialisierungsdefinition 1.II.

7.8.3 Anmerkung zum Spezialisierungsalgorithmus

Eine genaure Betrachtung des Spezialisierungsalgorithmus verrät, dass es möglich wäre, seine drei Teile zu vereinigen, d.h. jedes Mal, wenn der Sachverhalt wie in der Abbildung 7.7 entdeckt wird, könnte man die jeweilige spezialisierende Klasse C determinierenden Kontextquellen abfragen und, wenn C dadurch tatsächlich ermittelt werden konnte, die Spezialisierung der Situation durchführen. Man würde auf diese Weise doppeltes Abarbeiten des jeweils zu untersuchenden Sachverhaltes vermeiden und somit die asymptotische Komplexität in dem konstanten Faktor halbieren. Doch der Nachteil bei solchem Vorgehen ist, dass die Kontextquellen sequentiell statt parallel abgefragt werden, was den Überlegungen aus 7.5 widerspricht.

7.8.4 Anpassung des Spezialisierungsalgorithmus an die Spezialisierungs- definitionen 1.I und 1.III

Der dargestellte Spezialisierungsalgorithmus kann durch zwei einfache Änderungen an die Spezialisierungsdefinition 1.I angepasst werden. Erstens, man streicht die Zeilen 13 bis 16, denn für die Spezialisierung nach 1.I die spezialisierende Klasse nicht in einer Deskription partizipieren muss. Dies hat die zweite Änderung zur Folge, bei der die Zeile 17 passend in

$$ID := ID \cup (D,Z,C,null,M)$$

umgeschrieben wird (bzw. kann ID als eine Menge von Quadrupeln (D,Z,C,M) aufgefasst werden).

Um einen Spezialisierungsalgorithmus gemäß 1.III zu erhalten, müssen die Zeilen 13 und 14 durch

Für jede J, D′ *mit* $\exists C,Y.(PR^{-1}(J,Y) \wedge \text{defined-by}(Y,D') \wedge C \subset C' \wedge C'(J) \wedge \forall C''.(C''(J) \rightarrow (C'' = C' \vee C'' \not\subset C')))$

ersetzt werden. Damit wird gemäß 1.III.b) verlangt, dass C (unechte) Unterklasse einer solchen Klasse C′ ist, deren explizite Instanz eine D-Komponente parametrisiert.

7.9 Präzisierungsalgorithmus

Die Präzisierung versucht Individuen der Situation, die allein aus ermittelten Kontextinformationen nicht spezialisierbar sind, als Instanzen von Unterklassen ihrer eigentlichen Klassen zu inferieren. Die Präzisierung setzt voraus, dass die Komplettierung oder Spezialisierung (oder beides) bereits ausgeführt wurde, es sei denn die Präzisierung aus der in einer Situation vorhandenen D-Partizipanten (siehe 6.7.3) erlaubt ist.

7.9.1 Ein- und Ausgabe des Präzisierungsalgorithmus

Als Eingabe erwartet der Präzisierungsalgorithmus die Situation (als eine A-Box repräsentierte Benutzeraussage) und die zuvor bestimmten Funktion m und Mengen DS und ID. Ausgegeben werden die aktualisierten Situation und Menge ID.

7.9.2 Darstellung des Präzisierungsalgorithmus

Für eine erfüllte Deskription D und eine sie spezialisierende Klasse C beginnt die Präzisierung der Situation mit der Betrachtung aller solcher Deskriptionen D′, in denen

C partizipiert[70]. Diese Deskriptionen fehlen jedoch, wenn die Spezialisierung nicht ausgeführt wurde. In diesem Fall können sie gemäß 6.7.2 „künstlich herbeigeführt" werden, vorausgesetzt, dass die vorangegangene Komplettierung das Hinzufügen neuer Individuen in die Situation eingebracht hat (oder die Präzisierung nach 6.7.3 erlaubt ist). Diese Deskriptionen ergeben sich aus den expliziten Klassen der komplettierten Individuen, indem man diese Klassen als die spezialisierenden betrachtet, und die expliziten Instanzen dieser Klassen in der Deskriptionsontologie als D-Partizipanten dieser Deskriptionen auffasst. Derartig wird es bei allen komplettierten Individuen zum Beginn des Präzisierungsalgorithmus verfahren (Abbildung 7.9, Zeilen 3-6). (Bei Präzisierung nach 6.7.3 ergeben sie spezialisierende Deskriptionen wie dort angegeben.)

Nun, nach dem dieser aufbereitende Schritt gemacht wurde, wird jede solche Deskriptionen D′ mit den besagten Eigenschaften betrachtet (Zeile 7), und zwar es werden alle Klassen in der Deskriptionsontologie aufgesucht, die in D′ partizipieren (Zeile 8) – sie sind die Kandidaten für die Präzisierung. Für eine solche Klasse K (Zeile 9) wird jedes Individuum I der Situation betrachtet, das diese in der aktuellen erfüllten Deskription D partizipiert (Zeilen 10, 11). Wenn keine explizite Klasse von I eine Unterklasse von K ist, dann liegt in diesem Fall eine Präzisierung vor: I erhält in der Situation die Klassenzugehörigkeit zu K.

[70] Bzw. aller solcher Deskriptionen D′, die während der Spezialisierung in einen Zusammenhang mit C gebracht wurden, bei dem C Unterklasse eines D-Partizipanten von D′ ist.

INPUT: S – *Eine Situation als eine A-Box der Basisontologie*
INPUT: m – *eine Funktion, die (Tupeln von) Entitäten der Deskriptionsontologie mit einer Markierung verknüpft*
INPUT : DS **–** *Quintupelmenge : (Deskription, D-Partizipant, D-Partizipant, Klasse, Markierung)*
INPUT: ID – *Quintupelmenge: (Deskription, D-Partizipant, spezialisierende Klasse, Deskription, Menge von Kontextquellen)*
OUTPUT: S *(aktualisiert)*
OUTPUT: ID *(aktualisiert)*

```
1   Für jede Deskription D mit m(D) = „erfüllt“
2     Für jedes Z mit
      m(D,_,Z) = „vorhanden“ ∨ m(D,_,Z) = „determiniert“
3       Für jede E mit
        (D,Z,_,E,“determiniert“) ∈ DS ∨
        Präzisierung nach 6.7.3 erlaubt ∧ (D,Z,_,E,“vorhanden“) ∈ DS
4         Für jede explizite Instanz I von E in der Deskriptionsontologie
5           Für jedes D’ mit
            ∃X.( PR-1(I,X) ∧ defined-by(X,D’) ∧ D’≠ D )
6             ID := ID ∪ (D,Z,null,D’,null)
7       Für jedes D’ mit (D,Z,C,D’,_) ∈ ID
8         M := { R | ∃X,Y.(defines(D’,Y) ∧ PR(Y,X)) ∧ R(X)[71] }
9         Für jedes K ∈ M
10          Für jedes P mit
            m(D,_,P) = „vorhanden“ ∨ m(D,_,P) = „determiniert“
11            Für jedes I mit (D,P,I,_,_) ∈ DS
12              Für jedes E mit (D,P,I,E,_) ∈ DS
13                Wenn E ⊂ K
14                  GOTO 11
15              S := S ∪ K(I)
16  Return: S, ID
```

Abbildung 7.9: Der Präzisierungsalgorithmus.

7.10 Ansatz der Prädiktierfähigkeit von Deskriptionen

Im Algorithmus zur Bestimmung relevanter Deskriptionen bzw. dem Komplettierungsalgorithmus wurde eine gewisse „Prädiktierfähigkeit“ von Deskriptionen bezüglich einer Situation erwähnt. Diese Algorithmen sind so konzipiert, dass optional auch diejenige Deskriptionen gesucht werden können, die von der Situation nur zum Teil parametrisiert sind und in denen der sozusagen fehlende Teil nicht vollständig determinierbar ist (bzw. nicht determiniert werden konnte). Die

[71] Hier die explizite Zugehörigkeit der Instanz X zu einer deskriptionsontologischen Klasse R gemeint.

dahinter stehende Idee ist, solche Deskriptionen insofern als prädiktierfähig bezeichnen zu können, dass sie folgender Art Schlüsse ermöglichen: „Wenn es in der Situation Angabe `X` nicht fehlen würde, dann würde es in der durch diese Situation repräsentierten Aussage (des Benutzers) um `Y` handeln“. Zum Beispiel wäre dies von Nutzen, wenn man die Abbildung 7.1 betrachtet und sich dabei vorstellt, die Rolle `Locomotor` an die Deskription `Environment-Path-Description` „angeschlossen“ wäre. Nun angenommen, das Fortbewegungsmittel sei nicht determinierbar und eine Situation, in der eine Instanz von `Road` vorkommt, spezifiziere es nicht, dann würde diese Situation keine der drei Deskriptionen aus der Abbildung 7.1 parametrisieren. Dies bedeutete, diese drei Deskriptionen blieben für die Situation irrelevant, wenn man die prädiktierfähige Deskriptionen außer Betracht ließe. Im sonstigen Fall wäre die `Environment-Path-Description` als für die Situation prädiktierfähig befunden worden und es würde sich folgende Prädiktion ergeben: „Wenn bei der Situation `Motorcyclist` eine Rolle spielen würde, würde es sich um eine Fortbewegungssituation handeln“ bzw. würde die Prädiktion vielmehr in der vermuteten Notwenigkeit bestehen, den Benutzer über das Fortbewegungsmittel zu befragen. Würde das Prädiktierte eintreten, d.h. der Benutzer gibt tatsächlich an er sei ein Motorradfahrer, könnte man weiterhin wie gehabt verfahren: Wetterverhältnisse ermitteln und daraus den Straßentyp präzisieren.

Nun, derartiges Vorgehen, also die Nachfrage über die fehlenden Angaben, wurde bereits in 7.2 angedeutet. Algorithmisch ist dies ansatzweise umgesetzt worden, d.h. es wurde herausgearbeitet, wie die prädiktierfähige Deskriptionen zu bestimmen sind, jedoch nicht, wie daraus profitiert werden könnte, denn konzeptuell betrachtet zuwiderläuft die Prädiktierfähigeit dem Anliegen von SmartWeb, proaktiv zu sein.[72]

Doch scheint die Prädiktierfähigkeit von enormer Bedeutung zu sein, da vermutlich die Proaktivität nicht in allen Fällen gewährleistet werden kann. Tatsächlich erscheint die Forderung, jegliche in einer Situation notwendige Angaben seien entweder vom Benutzer gemacht, oder über die Kontextquellen determinierbar, als etwas zu weit gegriffen, insbesondere, wenn man von den typischen Szenarien in SamrtWeb abstrahiert und zu verallgemeinerten Problemstellungen übergeht, denen das Bemühen

[72] Natürlich ist auch SmartWeb nur eingeschränkt proaktiv, denn es ist unvermeidlich, gewisse Angaben vom Benutzer zu erwarten, jedoch bezogen auf SitCom liegt die Absicht gerade darin proaktiv zu sein, also die kontextuellen (Hintergrund-)Information ohne weiterer Nachfrage zu erschließen.

der vorliegenden Arbeit in erster Linie gewidmet ist und bei denen im Allgemeinen unwahrscheinlich abzuschätzen ist, welche Erwartungen an eine Situation sinnvoller Maßen vorausgesetzt werden können. Als Beispiel für die Verwendung der Prädiktierfähigkeit könnte man sich die Unterstützung eines Benutzers vorstellen, während er in einer Software laienhaft durch eventuell unkonsequentes Ausprobieren eine erwünschte Funktionalität zu erschließen versucht. Denn eben weil seine unkonsequente Handlungen seiner Absicht nicht im vollen Maße entsprechen, muss man davon ausgehen, dass die Deskription, die diese Absicht modelliert, nicht selektiert, sondern nur als prädiktierfähig befunden wird.

8 Einige Aspekte der Implementierung

In diesem Unterkapitel werden ausgewählte Aspekte der Implementierung von im Kapitel 7 dargestellten Algorithmen und anderen Routinen beleuchtet. Auch der Aufbau und die Verwendung des entwickelten Programmierwerkzeugs werden dokumentationsmäßig vorgestellt.

8.1 Trennung von Ontologien

In der Implementierung ist es vorgesehen, folgende drei Ontologien getrennt voneinander zu behandelt: die Basisontologie, die Deskriptionsontologie und die DnS-Ontologie. Das bedeutet nicht nur, dass es drei verschiedene Dateien gibt, welche die jeweilige Ontologie beinhalten, sondern auch dass sie intern als drei verschiedene Modelle existieren. Außerdem wird die Benutzeranfrage ebenfalls als ein Modell repräsentiert, ohne, dass es in eine der Ontologien integriert wird.

Diese Trennung ist, jedenfalls bei der vorgegebener Basisontologie, vorteilhaft, aber auch notwendig, denn das Inferieren in dieser für ein heute durchschnittliches Rechner eine Überbeanspruchung darstellen kann. So gelang es, nur das einfachste, transitive Reasoner in Verbindung mit der Basisontologie zu nutzen. Bei der Verwendung eines mächtigeren Reasoner dauern einige Inferenzen unakzeptabel lang oder die Anwendung terminiert nicht. Die Trennung der Ontologien ermöglicht somit das mächtigere Inferieren innerhalb der vergleichsweise kleinen Deskriptionsontologie bei gleichzeitiger Benutzung des transitiven Reasoners in Verbindung mit der Basisontologie. Übrigens, ist das Letztere für die Implementierung gerade noch ausreichend, denn es zumindest die transitive Hülle bezüglich `rdfs:subClassOf` und `rdfs:subPropertyOf` realisiert, wodurch die Hierarchie von expliziten Unterklassen und Unterrelationen zugänglich wird.

Abschließend sei bemerkt, dass die Trennung in der Implementierung nicht erzwungen wird: liegen drei Ontologien in einer Datei vor, dann kann diese Datei für jede Ontologie referenziert werden.

8.2 Problematik beim Einsatz von Jena Framework und OWL API

Der Umgang mit den Ontologien in Java wurde an zwei Werkzeugen erprobt: das in 2.7 vorgestellte Jena Framework und das seinem Zweck Jena-ähnliches *OWL API*[73]. Trotz der etwas höheren Performanz des Letzteren, wurde OWL API wegen sehr dürftiger Dokumentation verworfen, denn die Anwendungsweise in vielen Fällen intuitiv nicht erschließbar war. Jena Framework ist in dieser Hinsicht wesentlich empfehlenswerter, bietet jedoch bei vielen Methoden auch keine vollständige Spezifikation, so dass diese sich erst aus einem mühseligen Ausprobieren ergibt.

Nur um ein Beispiel dieser Unterspezifizierung zu nennen, werden zwei ontologische Klassen, die gleichen Namen haben, jedoch in zwei verschiedenen Ontologien definiert sind, von der `equals()` Methode als gleich bewertet. Werden jedoch die Unterklassen einer der beiden Klassen gefragt, liefert eine entsprechende Methode in Jena Framework nur diejenigen, die in derselben Ontologie definiert sind. Somit ergibt sich eine seltsame Tatsache, dass zwei gleiche Klassen unterschiedliche Unterklassen besitzen. Solange diese Tatsache unentdeckt blieb, gab es an bestimmten Stellen der Implementierung Missverhalten. Dieses Beispiel ist deshalb von enormer Bedeutung, weil es in der Implementierung die Trennung von teilweise überlappenden Ontologien eingeführt wird. So sind beispielsweise die D-Partizipanten, als Klassen, in der Basis- und der Deskriptionsontologie definiert.

Ein weiteres nennenswertes Hindernis bei der Verwendung von Jean Framework liegt darin, dass die Konstruierung eines OWL Modells aus einer RDF(S) Ontologie offensichtlich nicht immer korrekt verläuft, denn in einigen Fällen lassen sich ontologische Entitäten, die in der Letzteren vorhanden sind, in dem Ersten nicht wiederfinden (z.B. die Klassen über ihre Namen). Auch einige Entitäten lassen sich nicht als solche darstellen: zum Beispiel lässt sich im OWL Modell eine Entität X, über die in der RDF(S) Ontologie das Tripel `(Y, rdf:type, X)` besteht, in manchen Fällen aus unbekannten Gründen nicht als eine ontologische Klasse darstellen. Dieses Problem, wie auch das vorige, ist mit einigen Tricks lösbar. Doch es erscheint wichtig

[73] http://wonderweb.man.ac.uk/owl/

auf diese Problematik hinzuweisen, weil diese, aber vermutlich nicht nur diese Arbeit von ihr stark betroffen ist bzw. gebremst worden war.

8.3 Einige Klassen und Schnittstellen des entwickelten Programmierwerkzeugs

Im Folgenden werden einige Interfaces und Klassen[74] vorgestellt, die aus Sicht der Benutzung dieses Programmierwerkzeugs wichtige Schnittstellen darstellen. Es werden also nur diejenige Methoden der jeweiligen Klasse präsentiert, die für den eventuellen Benutzer von Bedeutung sind. Zu jeder Methode wird nur ein kurzes Kommentar gegeben, denn eine ausführliche Diskussion im Verlaufe dieses und der vorigen Kapiteln bereits erfolgte.

1. DataFactory

Diese Klasse dient der Erzeugung von Instanzen anderer Klassen aus dem Paket `de.emld.context.ontological_processing`.

2. CommonData

Dieses Interface erzeugt und beinhaltet die gemeinsamen[75] Datenfelder, z.B. die Modelle von Ontologien und von der Benutzeraussage, Konfigurationsparameter usw. Dabei wird an das Modell der Basisontologie das transitive Reasoner aus Jena Framework gebunden und an die Deskriptionsontologie – das von Pellet implementierte OWL DL Reasoner.

3. UserQuery

Dieses Interface repräsentiert die Benutzeraussage. Es beinhaltet ihre die A-Box und stellt einige Methoden zur Verfügung, mit deren Hilfe die Entitäten der A-Box sowie hierarchische und andere Beziehungen zwischen ihnen abgefragt werden können. Diese Beziehungen lassen sich erst aus der Basisontologie inferieren. Weil mit der vorgegebener Basisontologie, wie bereits erwähnt, nur das transitive Reasoner verwendet werden kann, gibt es zwei Implementierungen von `UserQuery`: eine, die an das transitive Reasoner der Basisontologie angepasst ist und eine andere – für den Fall,

[74] Alle in diesem Abschnitt vorgestellten Interfaces und Klassen befinden sich im Paket `de.emld.context.ontological_processing`.

[75] „Gemeinsam“ im Sinne der gemeinschaftlichen Nutzung unter den Instanzen von Klassen aus dem Paket `de.emld.context.ontological_processing`.

dass in der Basisontologie ein mächtigeres Reasoner eingesetzt werden kann. Die Anpassung der Ersten besteht darin, dass aus der expliziten Zugehörigkeit eines Individuum aus der A-Box der Benutzeraussage zu einer Klasse `K` der Basisontologie die Zugehörigkeit dieses Individuums zu einer Klasse `K'` genau dann inferiert werden kann, wenn das transitive Reasoner der Basisontologie `K'` als die Oberklasse für `K` inferieiren kann[76]. Eine Methode von `UserQuery`, die an dieser Stelle erwähnt werden sollte, ist:

`Model getUserQuery()`

Gibt das Modell zurück, das die A-Box enthält, die eine Benutzeraussage repräsentiert.

4. Core

Diese Klasse implementiert die Algorithmen zur Bestimmung von relevanten Deskriptionen, zu ihrer Komplettierung, Spezialisierung und Präzisierung. Einige Methoden dieser Klasse sind:

`void findRelevantDescriptions()`

Findet die relevanten Deskriptionen und die notwendigen Kontextquellen (gemäß dem Algorithmus aus der Abbildung 7.4).

`void completeRelevantDescriptions()`

Komplettiert die relevanten Deskriptionen (gemäß dem Komplettierungsalgorithmus).

`void specializeRelevantDescriptions_1_II()`

Spezialisiert die relevanten Deskriptionen (gemäß dem Spezialisierungsalgorithmus).

`void preciseRelevantDescriptions()`

Präzisiert die relevanten Deskriptionen (gemäß dem Präzisierungsalgorithmus).

`List<RelevantDescription> getRelevantDescriptions()`

Gibt die Liste der relevanten Deskriptionen zurück. Diese Liste aktualisiert sich nach der Durchführung jedes der obigen Algorithmen.

[76] D.h. es gilt folgende Regel für die ontologischen Entitäten `x`, `y`, `z`: `(x rdf:type y)` ← `(x rdf:type z)` ∧ `(z rdfs:subClassOf y)`. Ohne dieser Regel hätte `x` von einem transitiven Reasoner nur dann als zugehörig zu `y` inferiert werden, wenn es in expliziter Weise zutreffen würde.

public void writeAllResults()

Fügt alle von den obigen Algorithmen erzielte Ergebnisse in die A-Box der Benutzeraussage hinzu. Das Modell dieser A-Box in der entsprechenden Instanz von UserQuery wird also erst infolge der Ausführung dieser Methode aktualisiert.

5. RelevantDescription

Diese Klasse repräsentiert eine bezüglich einer Benutzeraussage relevante Deskription, enthält sämtliche Informationen zu den D-Komponenten und D-Partizipanten dieser Deskription, sowie zu Komplettierungen, Spezialisierungen, und Präzisierungen, die die D-Partizipanten dieser Deskription betreffen. Einige Methoden dieser Klasse sind:

Individual getDescription()

Gibt das Individuum der Deskriptionsontologie zurück, das diese Deskription repräsentiert.

String getLabel()

Gibt die Markierung (den Status) dieser Deskription (bezüglich der Benutzeraussage) zurück.

List<Dcomponent> getDComponents()

Gibt die Liste von D-Komponenten dieser Deskription zurück.

String getDComponentLabel(Dcomponent dc)

Gibt die Markierung der D-Komponente `dc` (bezüglich der Benutzeraussage) zurück.

List<Dparticipant> getDParticipants()

Gibt die Liste von D-Partizipanten dieser Deskription zurück.

HashMap<Individual,List<String>> getCompletions()

Gibt die Komplettierungen für diese Deskription als Objekte in Java zurück. Einem komplettierten Individuum wird die Liste von URIs aller seiner expliziten Klassen zugeordnet.

Model getCompletionModel()

Gibt das Modell zurück, das alle für diese Deskription komplettierten Individuen enthält.

HashMap<Individual,List<OntClass>> getSpecializings()

Gibt die Spezialisierungen für diese Deskription als Objekte in Java zurück. Einem Individuum aus der A-Box der Benutzeraussage wird eine Liste aller seiner expliziten Klassen zugeordnet, seine Zugehörigkeit zu denen sich durch die Spezialisierung ergeben hat.

Model getSpecializingModel()

Gibt das Modell zurück, das alle in dieser Deskription spezialisierten Individuen und ihre diejenige explizite Klassen enthält, Zugehörigkeit zu denen sich durch die Spezialisierung ergeben hat.

Vector<Substitution> getSubstitutions()

Gibt alle Substitutionen zurück, die sich infolge der Durchführung der Präzisierung ergeben haben.

6. Dcomponent

Diese Klasse repräsentiert eine D-Komponente sowie beinhaltet alle ihre PR-Targets. Einige Methoden dieser Klasse sind:

Individual getDComponentIndividual()

Gibt das deskriptionsontologische Individuum dieser D-Komponente zurück.

OntClass getDComponentClass()

Gibt die deskriptionsontologische Klasse dieser D-Komponente zurück.

HashMap<Individual,Vector<OntClass>> getPRTargetIndividuals()

Gibt die deskriptionsontologischen PR-Targets dieser D-Komponente zurück. Jedem PR-Target Individuum ist die Liste derjenigen PR-Target Klassen zugeordnet, zu denen es gehört.

HashMap<OntClass,Vector<Individual>> getPRTargetClasses()

Gibt die deskriptionsontologischen PR-Targets dieser D-Komponente zurück. Jeder PR-Target Klasse ist die Liste von dazugehörigen PR-Target Individuen zugeordnet.

7. Dparticipant

Diese Klasse repräsentiert einen D-Partizipanten einer relevanten Deskription, und zwar sowohl die deskriptionsontologische Entitäten, als auch ihnen entsprechende Individuen aus der A-Box der Benutzeraussage, die komplettierten Individuen, sowie Spezialisierungen und Präzisirungen. Außerdem enthält ein Objekt diese Klasse

Referenz zu der D-Komponente, die von in diesem Objekt repräsentierten D-Partizipanten parametrisiert wird. Falls ein D-Partizipant mehrere D-Komponenten parametrisiert, wird er durch entsprechend so viel Objekte dieser Klasse repräsentiert. Einige Methoden dieser Klasse sind:

`Individual getIndividual()`

Gibt das deskriptionsontologische Individuum dieses D-Partizipanten zurück.

`OntClass getOntClass()`

Gibt die deskriptionsontologische Klasse dieses D-Partizipanten zurück.

`String getStatus()`

Gibt die Markierung (den Status) dieses D-Partizipanten zurück.

`Dcomponent getDComponent()`

Gibt diejenige D-Komponente zurück, die von diesem D-Partizipanten parametrisiert wird.

`List<Individual> getOriginalIndividuals()`

Gibt die Liste von Individuen aus der A-Box der Benutzeraussage zurück, die diesem D-Partizipanten entsprechen (d.h. dieselbe D-Komponente parametrisieren).

`List<Individual> getCompletionIndividuals()`

Gibt die Liste von komplettierten Individuen zurück, die diesem D-Partizipanten entsprechen (d.h. dieselbe D-Komponente parametrisieren). Die Reihenfolge der Individuen in dieser Liste entspricht der Reihenfolge der Klassen aus der Liste, die `getCompletionClasses()` Methode zurückgibt. Falls für ein Individuum mehrere Klassen ermittelt wurden, zu denen es gehört, kommt dieses Individuum entsprechend viele Male in der Liste vor.

`List<OntClass> getCompletionClasses()`

Gibt die Liste von Klassen von komplettierten Individuen zurück, die diesem D-Partizipanten entsprechen (d.h. dieselbe D-Komponente parametrisieren). Die Reihenfolge der Klassen in dieser Liste entspricht der Reihenfolge der Individuen aus der Liste, die `getCompletionIndividuals()` Methode zurückgibt. Falls zu einer Klasse mehrere komplettierte Individuen gehören, kommt diese Klasse entsprechend viele Male in der Liste vor.

HashMap<Individual,List<OntClass>> getSpecializings()

Gibt alle Spezialisierungen, die für diesen D-Partizipanten vorgenommen wurden, als Objekte in Java zurück. Einem spezialisierten Individuum der A-Box der Benutzeraussage wird die Liste von Klassen zugeordnet, seine explizite Zugehörigkeit zu denen sich durch die Spezialisierung ergeben hat.

OntModel getSpecializingsAsModel()

Gibt das ontologische Modell zurück, in dem die Spezialisierungen festgehalten sind, die für diesen D-Partizipanten vorgenommen wurden.

Vector<Substitution> getSubstitutions()

Gibt die Liste aller Substitutionen zurück, die die Präzisierung für diesen D-Partizipanten ergeben hat.

8. Substitution

Diese Klasse repräsentiert eine aus der Präzisierung sich resultierende Substitution. Eine Substitution besteht aus einem Individuum aus der A-Box der Benutzeraussage, einer Klasse, zu der dieses Individuum gehört und einer Klasse, zu der es nach der Durchführung der Substitution gehören wird. Die Zugehörigkeit des Individuums zu der ersten Klasse wird infolge der Substitutionsdurchführung gelöscht[77]. Die Methoden dieser Klasse sind:

Individual getReferedIndividual()

Gibt das Individuum aus der A-Box der Benutzeranfrage zurück, dessen Klassenzugehörigkeit präzisiert wurde.

List<OntClass> getSubstituted()

Gibt alle Klassen des Individuums aus dieser Substitution zurück, die, als explizite Klassen dieses Individuums, durch die präzisierte Klasse ersetzt werden.

OntClass getSubstituting()

Gibt diejenige Klasse zurück, mit der das Individuum aus dieser Substitution präzisiert wird.

[77] Dies ist in dem Präzisierungsalgorithmus nicht spezifiziert.

8.4 Benutzung des entwickelten Programmierwerkzeugs

Nachfolgend ist in einigen Zeilen des Quellcodes ein Beispiel gegeben die notwendigen Instanzen erzeugt und die Routinen zur Ausführung gebracht werden.

```
CommonData cd =
  DataFactory.createCommonData(„settings.xml“,true,ci);
UserQuery uq = DataFactory.createUserQuery(cd,uqm);
Core core = new Core(cd);
core.findRelevantDescriptions();
core.completeRelevantDescriptions();
core.specializeRelevantDescriptions_1II();
core.preciseRelevantDescriptions();
List<RelevantDescription> rd = core.getRelevantDescriptions();
core.writeAllResults();
```

Es werden zuerst die gemeinsamen Datenfelder erzeugt (Zeile 1). Unter anderem gibt dabei der boolesche Parameter `true` an, dass die Ontologien gelesen und ihre Modelle erzeugt werden sollen. Der Parameter `ci` ist eine Instanz der Klasse `ContextInterface`. Danach wird Instanz der Klasse `UserQuery` aus einem Modell der A-Box der Benutzeraussage (`uqm`) erzeugt (Zeile 3). Schließlich werden die Routinen ausgeführt und die Ergebnisse in diesem Modell hinzugefügt.

9 Evaluierung und Diskussion

Bei der Evaluierung bzw. Diskussion der erzielten Ergebnisse wird es wie folgt vorgegangen: zuerst wird im Kapitel 5 dargestellte Infrastruktur, insbesondere der Cache betrachtet; dann, separat von dieser Infrastruktur, über die ontologiebasierte Behandlung des Kontextes, also über die Kapitel 6 bis 8, diskutiert und schließlich ein Überblick über die Gesamtheit dieser Ergebnisse im Vergleich zu ähnlichen Projekten gegeben.

9.1 Auswertung der Context-Aware Infrastruktur für SmartWeb

Bei der Auswertung der Context-Aware Infrastruktur aus dem Kapitel 5 sind vor allem zwei Faktoren relevant: Validierung der Erfüllung von Spezifikationen und, vor allem zeitliche, Performanz. Abschließend wird Cache der Verwendung von Datenbanken gegenübergestellt.

9.1.1 Akzeptanztest

Die Validierung erfolgte hauptsächlich in Form eines Akzeptanztestes durch den Einsatz des Kontextmoduls in SmartWeb (d.h. innerhalb der in SmartWeb entwickelten Testumgebung). Dieser bestätigte vielfach, dass die implementierte Infrastruktur korrekt funktioniert. Es wurde auch die Erweiterbarkeit von Kontextquellen erfolgreich getestet, wobei es, im Sinne der Benutzerfreundlichkeit, als besonders positiv zu verzeichnen ist, dass die zusätzlichen Kontextquellen und die korrespondierenden (Klassen für) Kontextinformationen von einem anderen Mitarbeiter im SmartWeb Projekt implementiert und integriert wurden. Die korrekte Funktionsweise von Cache wurde in grundlegenden Funktionen („`store`“- und „`get`“-Anfragen mit Schranken) ebenfalls im Rahmen des SmartWeb bestätigt, jedoch auch im Rahmen dieser Arbeit in hoher Vielfalt von Spezialfällen bzw. Anfragen unterschiedlicher Komplexität, ohne diese hier vorzuführen.

9.1.2 Performanz

Die zeitliche Performanz versteht sich hauptsächlich als Antwortzeit von „`get`“ Methoden der Klasse `ContextInterface` (siehe 5.1.4), denn die „`update`“-Methode selbiger Klasse, dank der Nebenläufigkeit innerhalb von `ContextInterface`, im Hintergrund ausgeführt wird, und somit eine vernachlässigbare Antwortzeit hat. Bei der Testierung innerhalb von SmartWeb betrug die Antwortzeit von „`get`“ Methoden, abzüglich der als invariant geltenden Ansprechdauer von Web Services, im Durchschnitt 30-40 Millisekunden (ms) und lag allenfalls unter 100 ms. Weil SmartWeb diesbezüglich keine zwingend einzuhaltende Sollwerte vorgibt, ist eine definitive Bewertung des gezeigten Zeitverhaltens problematisch. Es gibt jedoch empirische Richtwerte, dass Benutzer von SmartWeb ähnlichen Dialogsystemen die Toleranzgrenze bei Antwortzeiten bei etwa einer bis zwei Sekunden haben (in manchen, hier nicht diskutierten Fällen liegt, sie wesentlich höher). Somit ist die Antwortzeit von unter 100 ms beim Kontextmodul beachtliches Ergebnis, und zwar insbesondere unter der Berücksichtigung der Performanz anderer Module des SmartWeb, die in Gesamtheit mit dem Kontextmodul die Antwortdauer einer Anfrage an SmartWeb ausmachen.

9.1.2.1 Performanz von Cache

Die Performanz von Cache wurde bezüglich „`store`“- und „`get`“-Anfragen getestet.

Zuerst wurde eine sechsspaltige Tabelle (mit Spaltennamen von „`a`“ bis „`f`“), die auszugsweise in der Abbildung 9.1 links dargestellt ist, in Cache gespeichert. Diese Tabelle besteht aus Sätzen von aufeinander folgenden 20 Einträgen (angefangen bei dem ersten Cacheeintrag, also die erste Zeile in der abgebildeten Tabelle). Dabei ist in jedem Eintrag des `i`-ten Satzes genau drei Spalten (sprich Zellen) mit der Zahl `i` besetzt[78], und zwar derartig, dass innerhalb eines Satzes jede Kombination aus drei Spalten in genau einem Cacheeintrag vorkommt. Eine solche Tabelle wurde für eine Anzahl n von Sätzen von 50 bis 1600 mittels der „`store`“-Anfrage erzeugt.

[78] Genauer ausgedrückt gehören die in Cache gespeicherten Objekte einer Wrapper-Klasse für `Integer`, d.h. einer Klasse, die ein Datenfeld vom Typ `Integer` hat. Dies, weil `Integer` die Anforderung an die `toCompare` Methode der in Cache zu speichernden Objekte nicht erfüllt (siehe 5.1.3.5).

Danach wurden unterschiedliche Gruppen von „`get`“-Anfragen an Cache gestellt. Zur Erinnerung über die Signatur von „`get`“-Anfragen an Cache sei an dieser Stelle ein kleines Beispiel gegeben: `get([a,b],[2,3],null,[0,1],[c],null)` bedeutet „Gib aus jedem Cacheeintrag, bei dem Wert der Spalte „`a`“ 2 ± 0 und der Spalte „`b`“: (3 ± 1) beträgt, den Wert der Spalte „`c`“ zurück“. Die Gruppen von „`get`“-Anfragen findet man in der Abbildung 9.1 rechts oben: sie sind mit Buchstaben von **`A`** bis **`D`** markiert und bestehen aus jeweils fünf Anfragen, von denen die zweite, vierte und fünfte (erwartungsgemäß) null als Ergebnis liefern. In jeder Anfrage werden Werte von genau einer Spalte gesucht. Dabei wird in der ersten und zweiten Anfrage jeder Gruppe nur eine Schlüsselspalte (d.h. die Spalte, über die gesucht wird) angegeben; in den dritten und vierten Anfragen – zwei Schlüsselspalten; in den fünften Anfragen – drei Schlüsselspalten. Alle Anfragen spezifizieren keine Zeitschranke. Die Anfragen der Gruppe **`A`** geben auch keine bzw. `0` als eine Wertschranke vor. In der dritten Anfrage der Gruppe **`B`** wird eine Wertschranke für eine Schlüsselspalte vorgegeben, so dass die andere Schlüsselspalte, gemäß dem Hinweis aus 5.1.3.3 bzw. 5.1.3.4, ohne einer Wertschranke auftritt. Im Gegensatz dazu erhalten alle Schlüsselspalten der ersten Anfrage aus **`C`** bzw. der dritten Anfrage aus **`D`** eine Wertschranke. Jede Gruppe von Anfragen wird `n` Mal durchgeführt, entsprechend der Anzahl von Sätzen von Cacheeinträgen.

a	b	c	d	e	f
i	i	i			
i	i		i		
i	i			i	
i	i				i
i		i	i		
i		i		i	
i		i			i
i			i	i	
i			i		i
i				i	i
	i	i	i		
	i	i		i	
	i	i			i
	i		i	i	
	i		i		i
	i			i	i
		i	i	i	
		i	i		i
		i		i	i
			i	i	i

A	Für jedes i von 1 bis n 1: get([a],[i],null,[0],[c],null); 2: get([a],[-i],null,[0],[c],null); 3: get([b,d],[i,i],null,[0,0],[c],null); 4: get([b,d],[-i,i],null,[0,0],[c],null); 5: get([a,c,e],[i,i,i],null,[0,0,0],[d],null);
B	Für jedes i von 1 bis n wie in **A** außer 3: get([b,d],[i,i],null,[0,1],c,null);
C	Für jedes i von 1 bis n wie in **A** außer 1: get([a],[i],null,[1],c,null);
D	Für jedes i von 1 bis n wie in **A** außer 3: get([b,d],[i,i],null,[1,1],c,null);

	n:	50	100	200	400	800	1600
	store naiv:	5	16	31	47	73	214
	store cache:	78	146	224	494	865	1573
A	get naiv:	26	105	438	2005	6526	27521
A	get cache:	21	47	47	63	125	198
B	get cache:	15	11	16	36	73	177
C	get cache:	26	104	370	1864	8193	34776
D	get cache:	21	87	364	1995	7812	32328

Abbildung 9.1: Zeitliche Performanz von Cache: links – Auszug aus der im Cache gespeicherte Tabelle; rechts oben – die „`get`“-Anfragen; rechts unten – Antwortzeit auf diese Anfragen in ms.

Cache wurde algorithmisch mit einem relativ hohen Aufwand entworfen. Um zu bestätigen, dass dieser Aufwand sich bewährt hat, wurde für die Tabelle aus der Abbildung 9.1 eine simple (im Weiteren als „naiv“ bezeichnete) Datenstruktur entwickelt: ein aus sechs Elementen langen Arrays bestehendes Vektor. Dieser Datenstruktur können „`get`“-Anfragen gestellt werden, die denen aus Gruppe **A** äquivalent sind, also keine Schranken spezifizieren.

Alle Ergebnisse findet man in der Abbildung 9.1 rechts unten, sowie in den Abbildungen 9.2 und 9.3. Bei der Speicherung (Abbildung 9.2) hat es sich ergeben, dass Cache der naiven Implementierung, bei einer anscheinend gleicher asymptotischer Komplexität, im konstanten Faktor unterliegt. Dieses Ergebnis war zu erwarten, da es plausibel erscheint, dass die Speicherung in einer komplexeren, auf Abfragen optimierten Datenstruktur, entsprechend zeitaufwändiger ist. Die Ergebnisse von Cache scheinen also in dieser Hinsicht durchaus passabel zu sein; zudem besteht mit hoher Wahrscheinlichkeit ein gewisses Optimierungspotenzial, indem man das Konzept der

Nebenläufigkeit hineinbringt und die Speicherung als einen im Hintergrund ablaufenden Prozess gestaltet, der auf die Datenstruktur von Cache nur an wenigen Stellen zugreift.

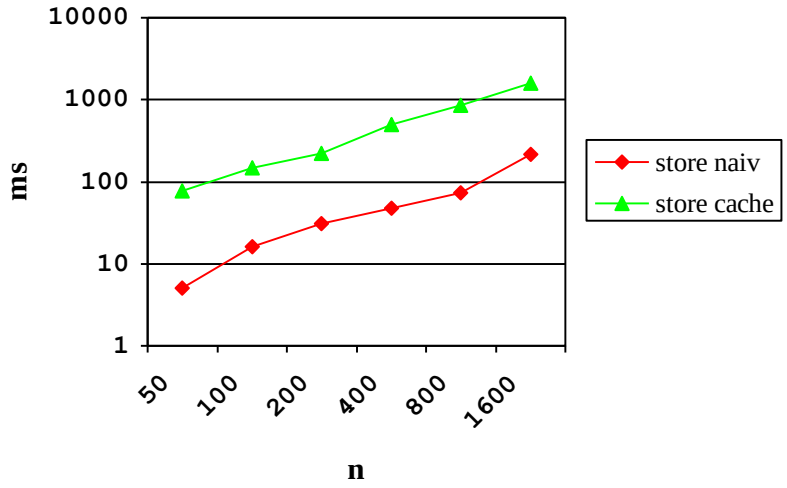


Abbildung 9.2: Antwortzeit der „`store`“ Methode in ms. (Zu beachten: logarithmische Achsenskalierung).

Das Abfragen von Cache (siehe Abbildung 9.3) hat sein diesbezüglich hervorragendes Zeitverhalten gezeigt, jedoch außer derjenigen Fälle, bei denen für jede Schlüsselspalte einer Anfrage eine Wertschranke spezifiziert wurde (also Gruppen **`C`** und **`D`**). In diesen Fällen müssen, wie bereits in 5.1.3.3 und 5.1.3.4 erläutert, fast alle Cacheeintrage überprüft werden, was die Suche linearisiert und im Zeitverhalten der naiven Datenstruktur angleicht. Für SmartWeb ist dieses Manko irrelevant, denn dort Anfragen über Kontextinformationen zu einem konkreten Benutzer gestellt werden, d.h. es wird in jeder Anfrage die Benutzer-Schlüsselspalte spezifiziert, und sie gibt keine Wertschranke vor.

Es gilt also, dass „`get`“-Anfragen wie in Gruppen **`A`** und **`B`** eine deutlich geringere asymptotische Zeitkomplexität aufweisen als die naive Datenstruktur (man beachte die logarithmische Achsenskalierung). Die Ursache für den Unterschied zwischen Gruppen **`A`** und **`B`** konnte nicht geklärt werden.

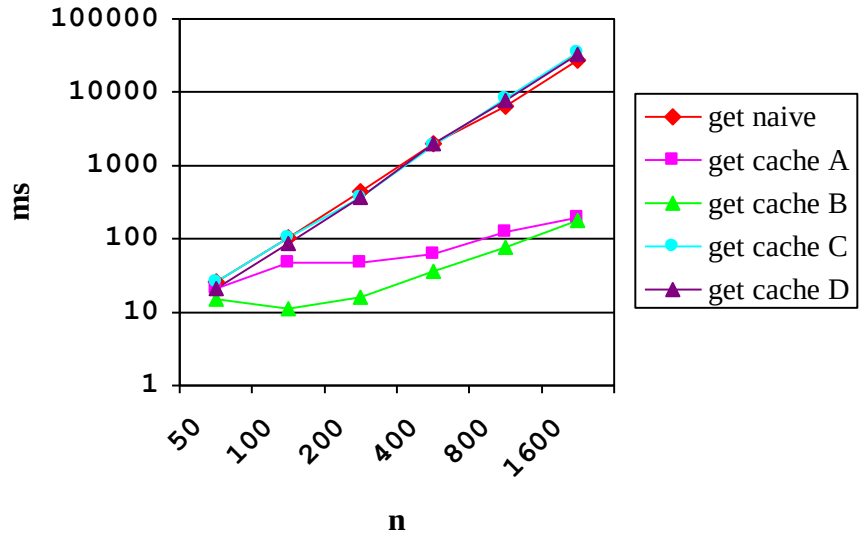
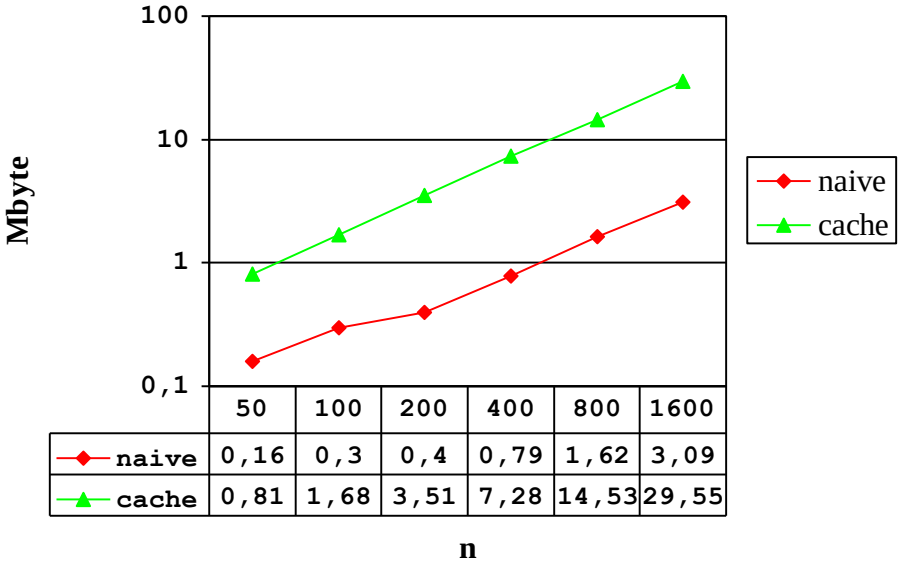


Abbildung 9.3: Antwortzeit der „`get`“-Anfrage in ms. (Zu beachten: logarithmische Achsenskalierung).

Beim Speicherverbrauch unterliegt zwar Cache der naiven Datenstruktur, aber wiederum nur im konstanten Faktor, bei anscheinend gleicher asymptotische Komplexität (siehe die Abbildung 9.4). Dies wiederspiegelt das erwartete Ergebnis, da Cache auf Hashtabellen basiert und diese bekanntlich einen höheren Speicherverbrauch haben als zum Beispiel Vektoren.



	50	100	200	400	800	1600
naive	0,16	0,3	0,4	0,79	1,62	3,09
cache	0,81	1,68	3,51	7,28	14,53	29,55

Abbildung 9.4: Speicherverbrauch in Megabyte (Zu beachten: logarithmische Achsenskalierung.)

9.1.3 Cache vs. Datenbank

Sein akzeptabler Speicherverbrauch qualifiziert Cache für eine Verwendung bei mittelgroßen zu speichernden Datenbeständen. Es ist zu vermuten, dass bei einer Anzahl von Einträgen in Cache, die unterhalb von etwa 10000 liegt, der Speicherbedarf von Cache geringer als bei einer Datenbank ist, denn selbst eine leere Datenbank samt der für sie benötigten Treiber, zum Beispiel im Falle von Java Derby Datenbank[79], etwa 10 Megabyte beansprucht. Dies ist dann insbesondere wichtig, wenn eine effiziente Datenspeicherungsstruktur auf mobilen oder sonstigen Geräten angebracht werden soll, die über eine sehr eingeschränkte Speicherkapazität verfügen. Zum Beispiel, wenn aus Gründen des Datenschutzes einige kontextuelle Informationen über die Benutzer von SmartWeb nur lokal auf dem mobilen Endgerät und nicht zentral auf dem Server gespeichert werden dürfen.

Ein gewichtiger Vorteil von Cache gegenüber den meisten Datenbanken liegt in vorgegebenen, stark einschränkenden Datentypen, die von Letzteren unterstützt sind. Im Großen und Ganzen kann man nur Zahlen und Zeichenketten speichern, sowie bestimmte Kombinationen von solchen. So ist es im Allgemeinen nicht möglich, Kontextinformationen komplexer Struktur zu speichern, es sei denn, sie werden in eine Zeichenkette umgewandelt. Dies hat zur Folge, dass innerhalb einer Datenbank die Auffassung über den Vergleich zweier Kontextinformationen, wie es über die Methode `compareTo()` implementiert werden kann, unmöglich ist. Kurz und knapp: eine Datenbank vermag beispielsweise „sonnig, 25°C“ quantitativ mit „bewölkt, 20°C“ nicht zu vergleichen, weil es ihr nicht übermittelt werden kann, wie dies zu vergleichen ist. Als Folge kann es bei Anfragen keine Wertschranke angegeben werden. Cache hingegen ist in dieser Hinsicht keinerlei beschränkt.

[79] http://developers.sun.com/javadb

9.2 Auswertung der ontologiebasierten Kontextrepräsentation und –verarbeitung

Bei der Auswertung in Kapiteln 6 bis 8 beschriebenen Sachverhalte ist es hauptsächlich drei Fragen nachzugehen:

- Inwiefern ist die dargestellte Repräsentationsform von Kontextinformationen, also das Kontextmodell (siehe 4.2.3), angemessen und der Aufgabe gewachsen, Kontextinformationen profitabel darzustellen?
- Wie aussagekräftig sind die in Kapiteln 6 und 7 vorgeschlagenen Dienste (Bestimmung relevanter Deskriptionen, Komplettierung, Spezialisierung, Präzisierung und bedingt Prädiktierfähigkeit), die in dieser Arbeit das Context-Aware Computing ausmachen? Dies ist also eine Frage nach der Bestimmung des Nutzfaktors der besagten Diensten.
- Wie ist die Performanz des im Kapitel 8 resultierenden Programmierwerkzeugs?

9.2.1 Ontologische Repräsentierung von Kontextinformationen

In dieser Arbeit wurde die Repräsentierung von kontextuellen Informationen als basisontologische, aber gleichzeitig auch deskriptionsontologische Entitäten realisiert. Das ontologische Kontextmodell wurde im Allgemeinen, wie in 4.2.3 bereits erwähnt, nach einer Reihe von Kriterien für das geeignetste befunden. [47] nennt unter anderem[80] folgende drei Merkmale, die eine gute Kontextontologie erfüllen muss:

- *Einfachheit*: die Beschaffenheit einer Kontextontologie bzw. die intendierte Bedeutung ihrer Entitäten soll (für potentielle Verwender dieser Ontologie) so einfach wie möglich anzueignen sein,
- *Generierbarkeit*: es soll statt einer festen vorgegebenen Menge von repräsentierbaren Kontextarten eine Möglichkeit geben, in Begriffen der Kontextontologie neue Kontextarten hinzuzufügen,
- *Expressivität*: eine Kontextontologie soll ermöglichen, Sachverhalte über eine Art des Kontext in hohem Detailliertheitsgrad zu beschreiben.

Man kann nun diskutieren, inwiefern diese Merkmale bei der in dieser Arbeit verwendeten Repräsentierungsform von Kontextinformationen präsent sind. In

[80] Das weitere in [47] genannte Merkmal fällt mit der Aggregation zusammen. Diese wird in 9.2.2 behandelt.

SmartWeb gibt es keine Kontextontologie; stattdessen kann man die Gesamtheit aus der Basis- und Deskriptionsontologie als solche bezeichnen. Die Basisontologie von SmartWeb (SWIntO) ist eine sehr umfangreiche und domänenübergreifende Ontologie, die Sachverhalte der Welt tief ins Detail gehend modelliert. Weiterhin ist es aufgrund einer generischen Grundontologie (SmartSUMO) möglich weitere Domänenontologien hinzuzufügen. Auch die nach dem DnS-Design-Pattern aufgebaute Deskriptionsontologie ist in hohem Maße generisch, so dass das Anfügen neuer Kontextarten sich gewiss realisieren lässt. Dies spricht für eine hohe Expressivität und Generierbarkeit der hier verwendeten Repräsentierungsform von Kontextinformationen.

Andererseits ist die Umfangreichlichkeit der Basisontologie gewissermaßen auch ihr Fluch: sich einen Überblick über ihre Beschaffenheit und intendierte Bedeutung ihrer Entitäten zu verschaffen ist eine sehr zeitintensive Angelegenheit, die zudem auch sehr komplex wäre, hätte der potentielle Benutzer (von der in dieser Arbeit entwickelten Werkzeuge) zusätzlich noch mit der ganzen DnS-Ontologie konfrontieren müssen. Da jedoch nur der DnS-Design-Pattern übernommen wurde, und nicht die übrigen, durch komplexe Konzeptbeschreibungen definierten und hier nicht dargestellten Konzepte und Relationen, bleibt dies dem potenziellen Benutzer erspart. Dennoch ist hier das Merkmal „Einfachheit“ eher nicht gegeben. Generell scheinen Einfachheit und Expressivität in Konflikt miteinander zu stehen.

9.2.2 Nutzfaktor

Es gibt auf dem Gebiet des Context-Aware Comuting keine Einigkeit über Verwendungszwecke von kontextuellen Informationen. Jedoch wie in 4.2.4 bereits erwähnt, fasst [5] zusammen, dass viele Context-Aware Systems die Aggregation (das Zusammenschließen zu höherwertigen Kontextinformationen aus den niederwertigen) und die Interpretation (Definition von Aktionen, die durch bestimmte Kontextinformationen qualifiziert werden) unterstützen. Beides lässt sich prinzipiell in Form von „*wenn ..., dann...*“ Regeln darstellen.

Sowohl die Aggregation als auch die Interpretation sind durch die in dieser Arbeit vorgeschlagenen Algorithmen abgedeckt, wobei die Aggregation im Sinne des Komponierens von Sensordaten zu höherwertigen kontextuellen Informationen in der vorliegenden Context-Aware Infrastruktur im Konzept der Abhängigkeit kontextueller

Informationen von Sensordaten verankert ist. Im übrigen besteht in dem hier vorgestellten Ansatz die Aggregation darin, dass die nicht determinierbaren D-Partizipanten einer Deskriptionen durch die Präzisierung inferiert werden, falls diese unter anderem aufgrund der ermittelten Kontextinformationen von der aktuellen Situation erfüllt wird. Dies entspricht folgender Regel: „wenn in einer Situation gewisse Sachverhalte gegeben sind und bestimmte Kontextinformationen ermittelt wurden, dann gelten auch bestimmte andere kontextuelle Informationen." Der „dann"-Teil dieser Regel kann als „... dann sind bestimmte Aktionen auszuführen" aufgefasst werden, wenn man bei den Deskriptionen `Course`, also die Aktivitäten und Abläufe, entsprechend modelliert (siehe die Abbildung 4.2 und das Kapitel 4.3.1.2). In diesem Fall werden sie, als nicht determinierbare D-Partizipanten, ebenfalls inferiert. Somit ist die Interpretation ebenfalls abgedeckt.

Als positiv bei der Nutzbarkeitsanalyse ist es zu verzeichnen, dass bei der Konzipierung von Verwendungsmöglichkeiten von im Kapitel 7 definierten Diensten auch Folgendes berücksichtigt wurde: Schlussfolgerungen nach der Regel „wenn in einer Situation gewisse Sachverhalte gegeben sind, dann gelten bestimmte andere kontextuelle Informationen" werden von der Präzisierung ebenfalls unterstützt (siehe 6.7.3.). Man kann also wie in 6.7.3 beschrieben, generell Sachverhalte präzisieren, ohne dabei auf dabei auf kontextuelle Informationen zu stützen. Die besagte Dienste, sowie die Deskriptionsontologie als Repräsentierungsformalismus, können also nicht nur innerhalb eines Context-Aware System eingesetzt werden, sondern beispielsweise auch in ontologiebasierten Planungssystemen (insbesondere auch deswegen, weil DnS-Framework auch zur Lösung von Planungsproblemen konzipiert wurde [3]).

Die Besonderheit bzw. das kennzeichnende Merkmal des in dieser Arbeit entstandenen Context-Aware System ist, dass es zum Einsatz in einem Dialogsystem konzipiert wurde, d.h. die zentrale Frage, wenn es sich um die Erkennung aktueller Situation handelt, lautet: „*Was meint ein Benutzer wenn er X sagt?*" Die Beantwortung dieser Frage ist einerseits das Ergebnis eines passenden, aussagestarken Repräsentierungsformalismus von Kontextinformationen und andererseits – das Ergebnis ihrer Verwendung, also der im Kapitel 6 und 7 formalisierten Bestimmung relevanter Deskriptionen, ihre Komplettierung und Spezialisierung.

Schließlich sei hinsichtlich der Verwendungsmöglichkeiten von Kontextinformationen auf die in 7.10 ansatzweise eingeführte Prädiktierfähigkeit hingewiesen, die eventuell ganz andere, noch unerforschte Möglichkeiten bietet, die im Bereich des unscharfen (fuzzy) oder induktiven oder modalen (d.h. modale Operatoren wie „glaube“ zulassenden) Schließens liegen.

9.2.3 Zeitliche Performanz

Die Ausführungszeit von allen vier Diensten zusammen beläuft sich etwa auf 3 Sekunden (abzüglich der Antwortzeit von Web Services und des Auslesens von Ontologien, und mit einem transitiven Reasoner für die Basisontologie). Dieses verbesserungsfähige Ergebnis ist hauptsächlich auf die Größe der Basisontologie zurückzuführen. Deshalb ist es überlegenswert, ein Konzept herauszuarbeiten, wie und ob sie aufgespaltet werden könnte. Zum Beispiel in mehrere Teile, die jeweils aus der Grundontologie und einer Domänenontologie bestehen. Man könnte also nur die jeweils benötigten Teile ansprechen und dadurch die Ausführungszeit reduzieren.

Aber es sollte auch nicht außer Acht gelassen werden, dass die vier besagten Dienste implementierendes Programmierwerkzeug nicht speziell auf Zeit optimiert wurde. Möglicherweise trägt dies auch zu der vergleichsweise hohen Ausführungszeit bei.

9.3 Vergleich mit ähnlichen Projekten

Es ist schwierig andere Context-Aware Systems mit dem hier entworfenen zu vergleichen, denn sie nicht für eine Anwendung aus dem Gebiet der Linguistik entwickelt wurden und sich somit vielmehr der Frage „*Welche Aktion soll ausgeführt werden wenn bestimmte Kontextinformationen eintreten?*“ widmen. Die Welt, in der diese Context-Aware Systems agieren, ist sehr eingeschränkt: es handelt sich meist um nur eine abgedeckte Domäne, z.B. das Autofahren oder Lokalisierung von Personen und Tätigkeiten, die sie gerade ausführen; Begriffe und Ereignisse, die in dieser Welt Platz finden, sind determiniert. Dementsprechend wird bei solchen Context-Aware Systems das ontologische Kontextmodell oft nicht benötigt; in einigen Fällen wird die simple Attribut-Wert Darstellung als ausreichend befunden.

Doch es gibt einige Context-Aware Systems, die ebenfalls Kontextmodell nutzen, zum Beispiel: Gaia [22] (siehe auch 4.2.4), SOCAM [23] (siehe auch 4.2.4) und CoBra [18]. Vor allem auf diese drei Frameworks beziehen sich die ersten zwei im Folgenden genannten Vorteile des in dieser Arbeit verfolgten Ansatzes.

Fundierte, gemeinsame Semantik von Kontextinformationen: Bei der in anderen Projekten entwickelter Kontextontologie handelt es sich um eine Anwendungsontologie; eine Domänen- oder Grundontologie wird dort nicht verwendet. Dies hat *erstens* zur Folge, dass nur ein bruchstückhaftes Wissen über Kontextinformationen und die mit ihnen verbundene Sachverhalte repräsentiert ist. Es gleicht also zum Beispiel einer Unterhaltung über Abwägung einer Auto- vs. Bus- und Radfahrt, ohne dabei eine Vorstellung davon zu haben, dass Auto, im Gegensatz zu Fahrrad, ein motorisiertes Fortbewegungsmittel ist, und dass Bus, im Gegensatz zu den beiden anderen, ein öffentliches Verkehrsmittel. *Zweitens* ist mit der „gemeinsamen Semantik“ gemeint, dass bei dem hier verwendeten Kontextmodell alle Komponenten der Anwendung, also von SmartWeb, dieselbe Terminologie benutzen. Wird an einer Stelle eine Kontextinformation ermittelt oder inferiert, so kann sie direkt der Benutzaussage hinzugefügt werden, und alle andere Komponenten von SmartWeb „erkennen“ Semantik des ergänzten Wissens, weil es in gemeinsamen Begriffen der Basisontologie ausgedrückt ist. (Dazu sei es angemerkt, dass in Kontextmodellen, die weder ontologisch noch logikbasiert sind, überhaupt keine maschinell erfassbare Semantik von Kontextinformationen gegeben ist.) *Beides* führt zu einer Expressivitätsreduzierung; für SmartWeb wäre dies gravierend.

Wohldefinierte Dienste zur praktischen Verwendung von Kontextinformationen. In dieser Arbeit wurde formal und (teilweise) algorithmisch wohldefiniert, wie von Kontextinformationen profitiert werden kann. Bei anderen Context-Aware Systems ist dies nicht gegeben (Veröffentlichungen zu diesen Frameworks lassen einen anderen Schluss nicht zu). Stattdessen wird es an Beispielen gezeigt, dass Regeln bezüglich der Kontextontologie aufgestellt werden können, aus denen inferiert wird. Eine Aussage über die Expressivität derartiger Vorgehensweise bleiben andere Context-Aware Systems schuldig. Außerdem lassen sich diese Regeln nicht generalisieren, wie es in hier vorgestelltem Ansatz durch die Verwendung eines Design-Patterns im Aufbau der Deskriptionsontologie ermöglicht wurde. Hier wurde also ein generelles, alle Arten von

Kontextinformationen betreffendes Verfahren definiert wie Inferenz- und andere Dienste zu bewerkstelligen sind.

Cache (vs. Datenbank) zur Speicherung von Kontextinformationen. Vorteile, die das hier implementierte Speicherungssystem bietet, wurden ausführlich in 9.1.3 diskutiert.

Fairerweise sollen auch Nachteile vorliegender Arbeit bezüglich der anderen Context-Aware Systems genannt werden.

Datenschutz: Weil kontextuelle Informationen – zum Beispiel diejenige, die Aufenthaltsort einer Person darstellen – privat sind, wurde in vielen Context-Aware Systems ein Datenschutzkonzept entwickelt, bei dem beispielsweise Benutzer den Zugriff aus sie betreffende Kontextinformationen selbst kontrollieren.

Zuverlässigkeit von Sensordaten: Jeder Sensor sowie gegebenenfalls jede Messung eines Sensors hat ein bestimmtes Grad der Zuverlässigkeit. In Konfliktsituationen, bei denen diverse Sensoren mit unterschiedlichem Ergebnis dieselbe Größe messen, ist die Zuverlässigkeit entscheidend. In der hier vorgestellten Context-Awre Infrastruktur wurde Zuverlässigkeit nicht behandelt.

Benachrichtigen über Kontextinformationen: In einigen Context-Aware Systems ist es möglich, einen Informationsdienst sozusagen zu abonnieren. Dies bedeutet, eine Softwarekomponente meldet ihr Interesse an Kontextinformationen einer bestimmten Art bzw. eines bestimmten Inhaltes und erhält eine Nachricht, sobald dies eintritt. Dadurch wird das unnötige und gegebenenfalls zeitaufwändiges Abfragen erspart. Die in dieser Arbeit vorgestellte Context-Aware Infrastruktur unterstützt dies nicht.

10 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Context-Aware Infrastruktur vorgestellt, die im Rahmen des Projekts SmartWeb kontextuelle Informationen ermittelt, zu Verfügung stellt und speichert. Letzteres geschieht in einem extra für diese Zwecke entwickelten schnellen und flexiblen Speicher, in Cache, der in dieser Anwendung erfolgreich mit Datenbanken konkurriert. Die Evaluierung von Cache hat gezeigt, dass beim Eingehen gewisser Kompromisse ein enormer Gewinn in der Zugriffsschnelligkeit erreicht werden kann. Die Context-Aware Infrastruktur zeichnet sich erstens durch Erweiterungsmöglichkeiten sowohl hinsichtlich der verwendeten Sensoren als auch in Bezug auf Kontextquellen und der von ihnen ermittelbaren Kontextinformationen aus, und zweitens durch die konzipierte und implementierte Nebenläufigkeit, die zur Verbesserung der zeitlichen Performanz geführt hat. Schließlich unterstützt die entworfene Infrastruktur die ontologische Repräsentierung von Kontextinformationen, und zwar in Begriffen einer sehr expressiven, fundierten und generischen Basisontologie.

Schwerpunktmäßig konzentrierte sich diese Arbeit auf einem theoretischen Entwurf (sprich begründete, formale Festlegung), algorithmischer Umsetzung und praktischer Implementierung verschiedener Diensten, darunter die Inferenz (in Form von Präzisierung), die mit Hilfe von Kontextinformationen realisierbar sind. Die wohl wichtigste Frage dieser Arbeit war „wie kann man von Kontextinformationen profitieren?“. Dabei wurden diesbezüglich gewisse Erwartungen seitens SmartWeb gesetzt, die jedoch nicht als Maßstab gewählt wurden. Stattdessen wurde versucht, eine generelle Studie zu erstellen, die weitaus höheren Ansprüchen genügt. In der Theorie ist dies in hohem Maß gelungen, jedoch war eine vollständige Umsetzung und Implementierung aller, im theoretischen Teil formal beschriebenen Ideen, wegen des enormen Aufwands im Rahmen dieser Arbeit leider nicht möglich (siehe 7.1). Trotzdem ist ein effektives Werkzeug gelungen, das die Ansprüche von SmartWeb in vollem Maße erfüllt, was für ein typisches Anwendungsszenario des SmartWeb aus dem Bereich der Navigation mit Hilfe einer kleinen, im Rahmen dieser Arbeit prototypisch entwickelten, pragmatischen Deskriptionsontologie bestätigt wurde. Letztere ist, entsprechend der Vorgabe, gemäß den generischen DnS-Design-Pattern aufgebaut. Im Vordergrund stand dabei die Idee, dass aus den observierten Sachverhalten, zum

Beispiel Wetterverhältnisse oder Aufenthaltsort, die Situation, in der sich eine Person befindet, erst erschlossen werden kann, wenn gewisses pragmatisches Allgemeinwissen bekannt ist und in geeigneter Form vorliegt. Inspiriert wurde diese Idee durch Beispiele der menschlichen Kommunikation, bei der die explizite Angabe aller Hintergründe nicht notwendig ist, um sich zu verständigen bzw. die Situation seines Visavis nachzuvollziehen.

Somit ist die Aufgabestellung erfüllt. Aber diese Arbeit ist auch deswegen als erfolgreich einzuschätzen, da es gelungen ist, zwei „Welten“ zu verbinden: das Context-Aware Computing und die ontologiebasierte Wissensrepräsentation.

Es gibt jedoch ein hohes Potenzial an Weiterentwicklung. Welche Wege sie schlägt, hängt von der Zielsetzung ab. Soll der in dieser Arbeit dargestellte Ansatz im Sinne der Vermarktungsreife vollendet werden, ist an folgenden Punkten weiterzuarbeiten. Es müsse eine der Realität genügende Deskriptionsontologie entwickelt werden, die mehr Sachverhalte festhält, statt sich nur an ein Beispiel zu orientieren. Zur Erstellung einer solchen Ontologie könne man ein graphisches Assistierungstool implementieren, der zudem ihre Konsistenz garantieren solle. Weiterhin könne man die Implementierung besagter Dienste derart vervollständigen, dass sie den im theoretischen Teil formulierten Ideen vollständig entspreche. Dies ließe komplexere Deskriptionsontologien zu, die vermutlich unentbehrlich sind, um die Realität getreu zu wiederspiegeln. Ferner ist an der Performanz dieser Dienste bzw. viel mehr an der Problematik zu arbeiten, die mit der Umfangreichlichkeit der Basisontologie verbunden ist. Schließlich könne Cache derart ausgebaut werden, dass es zur Speicherung großer Datenbestände eventuell Festplatte verwendet und zudem eine für als „Cache“ zu bezeichnende Speicher übliche Verdrängungsstrategie implementiert.

Aus wissenschaftlicher Sicht wäre eine Untersuchung interessant, inwiefern die besagten vorgeschlagenen Dienste die volle Spanne dessen überblicken, wozu Kontextinformationen verwendet werden können. Falls es in dieser Hinsicht Lücken geblieben sind, so dass ein Bedarf an Funktionalitäten bestehet, die diese Dienste nicht abdecken, könnte man weitere Dienste formulieren. Anfangen könnte man mit der Ausbau der Idee prädiktierfähiger Deskriptionen. Weiterhin stellt sich bei der Notwendigkeit der Erstellung umfangreicherer Deskriptionsontologien eine sehr

anspruchsvolle Frage, ob dies automatisch oder semiautomatisch realisierbar wäre, indem Deskriptionen beispielsweise aus Text oder aus Observierung des menschlichen Verhaltens erlernt würden. Schließlich ist eine Untersuchung lohnenswert, ob die in dieser Arbeit geschilderten theoretischen Auslegungen, in eine ontologische Repräsentierung, beispielsweise eine in OWL definierte Ontologie, überführbar sind. Damit würde man von der Implementierung in einer konkreten Programmiersprache, hier – in Java, abstrahieren.

Anhang A: Syntax und Semantik von $\mathcal{SHOIN}(\mathcal{D})$

In der Abbildung A.1[81] ist die Syntax und formale Semantik der Beschreibungslogik $\mathcal{SHOIN}(\mathcal{D})$ angegeben. Dabei werden mit *C*, und *D* Konzeptbeschreibungen bezeichnet, mit *R* – die Rollen, mit R^- – die zu *R* inverse Rolle, mit R_+ – die transitive Hülle von *R*. # bezeichnet die Kardinalität, *I* – die Interpretationsfunktion, Δ^I – Interpretationsdomäne, die disjunkt zu Δ_D, der Interpretationsdomäne von Datentypen, ist.

Construct name	Syntax	Semantics
atomic abstract role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
concrete role	U	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
transitive role	$R \in \mathbf{R}_+$	$R^{\mathcal{I}} = R^{\mathcal{I}+}$
inverse abstract role	R^-	$\{\langle x, y\rangle \mid \langle y, x\rangle \in R^{\mathcal{I}}\}$
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
datatype	d	$d_{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
concrete value	v	$v^{\mathcal{I}} = v^{\mathbf{D}}$
negation	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
exists restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
concrete exists restriction	$\exists R.d$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in d^{\mathbf{D}}\}$
concrete value restriction	$\forall R.d$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in d^{\mathbf{D}}\}$
atleast restriction	$\geq nR$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \geq n\}$
atmost restriction	$\leq nR$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \leq n\}$
nominal	o	$\sharp\{o^{\mathcal{I}}\} = 1$
subsumption	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$

Abbildung A.1: Syntax und Semantik von $\mathcal{SHOIN}(\mathcal{D})$.

[81] Diese Abbildung wurde in überarbeiteter Form aus [28] übernommen. Kommentare und Einzelheiten dazu findet man in [29].

Anhang B: Äquivalenz von beschreibungslogischen Konstrukten zu den PL1 Formeln

In der Abbildung B.1[82] sind für die Konstrukte der Beschreibungslogik die jeweils äquivalente PL1 Formel angegeben. Dabei werden mit *C*, C_1, ... C_n und *D* Konzeptbeschreibungen bezeichnet, mit *P* und *Q* – die Rollen, mit P^- – die zu *P* inverse Rolle, mit P^+ – die transitive Hülle von *P* und mit a_1, ..., a_n – Individuen.

DL	FOL
$a : C$	$C(a)$
$\langle a, b\rangle : P$	$P(a, b)$
$C \sqsubseteq D$	$\forall x.C(x) \rightarrow D(x)$
$P^+ \sqsubseteq P$	$\forall x, y, z.(P(x, y) \wedge P(y, z)) \rightarrow P(x, z)$
$\top \sqsubseteq \leqslant 1\, P$	$\forall x, y, z.(P(x, y) \wedge P(x, z)) \rightarrow y = z$
$P \equiv Q^-$	$\forall x, y.P(x, y) \iff Q(y, x)$
$C_1 \sqcap \ldots \sqcap C_n$	$C_1(x) \wedge \ldots \wedge C_n(x)$
$C_1 \sqcup \ldots \sqcap C_n$	$C_1(x) \vee \ldots \vee C_n(x)$
$\neg C$	$\neg C(x)$
$\{a_1, \ldots, a_n\}$	$x = a_1 \vee \ldots \vee x = a_n$
$\exists P.C$	$\exists y.(P(x, y) \wedge C(y))$
$\forall P.C$	$\forall y..(P(x, y) \rightarrow C(y))$
$\geqslant n\, P.C$	$\exists y_1, \ldots, y_n. \bigwedge_{1 \leqslant i \leqslant n}(P(x, y_i) \wedge C(y_i)) \wedge \bigwedge_{1 \leqslant i < n, i < j \leqslant n} y_i \neq y_j$
$\leqslant (n-1)\, P.C$	$\forall y_1, \ldots, y_n.(\bigwedge_{1 \leqslant i \leqslant n}(P(x, y_i) \wedge C(y_i))) \rightarrow (\bigvee_{1 \leqslant i < n, i < j \leqslant n} y_i = y_j)$

Abbildung B.1: Äquivalenz von beschreibungslogischen Konstrukten zu den PL1 Formeln.

[82] Diese Abbildung wurde in überarbeiteter Form aus [27] übernommen.

Anhang C: Bezeichnungen in Abbildungen

Im Folgenden sind die Bezeichnungen erklärt, die in mehreren Abbildungen dieser Arbeit verwendet wurden. Diese Abbildungen sind in ihrer Beschriftung mit (*) markiert.

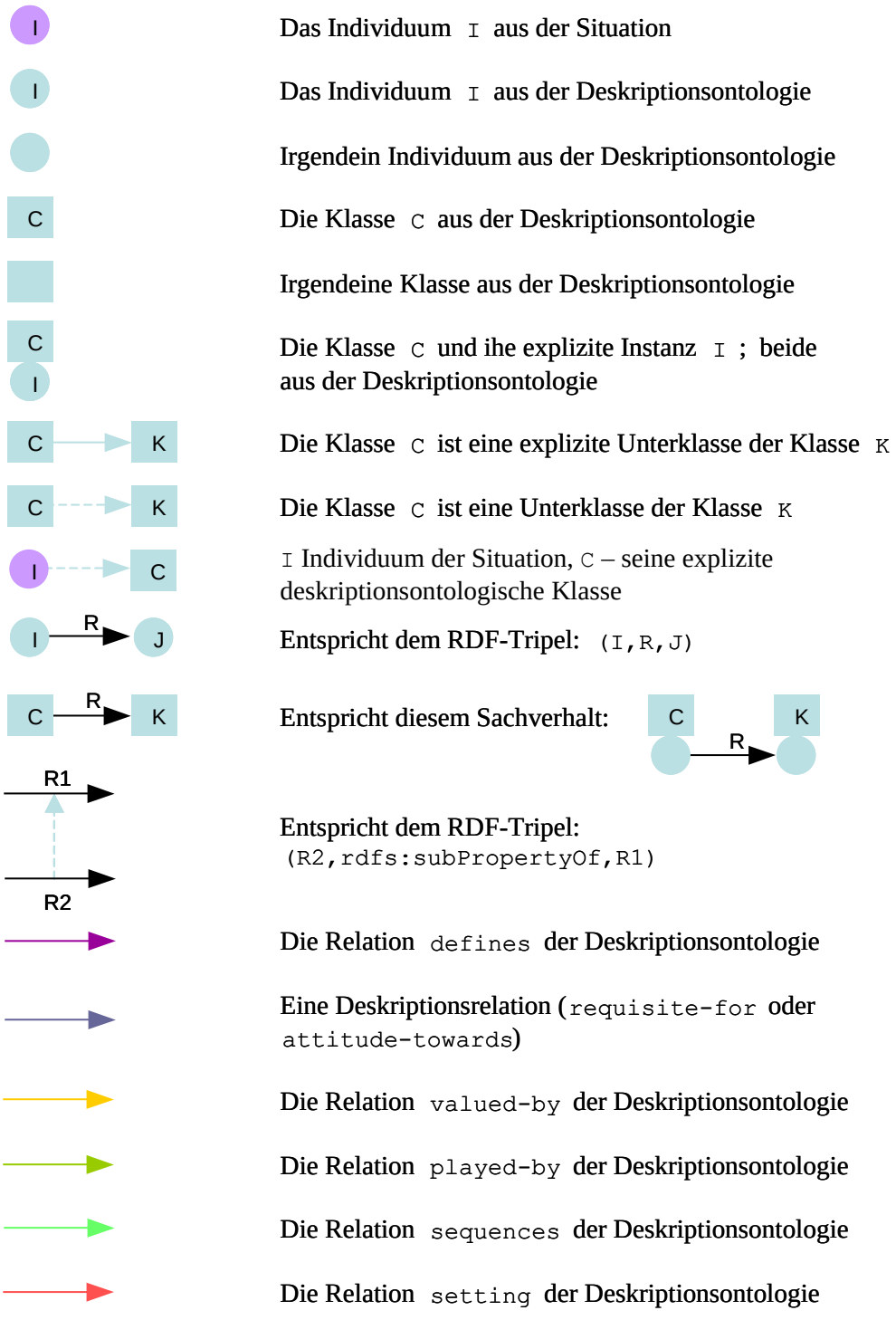


Abbildung C.1: Bezeichnungen für die mit (*) markierten Abbildungen.

Anhang D: Eine kurze Übersicht über Pakete des Programmierwerkzeugs

Das in dieser Arbeit entstandene Programmierwerkzeug beinhaltet drei Pakete. Im Folgenden wird eine Übersicht über ihre Inhalte gegeben:

- `de.emld.context.cache`: Beinhaltet unter anderem Klassen `ContextInterface` (siehe 5.1.4), `Cache` (siehe 5.1.3), `BinaryHashMap` (siehe 5.1.3.3) `NaiveCache` („naive“ Implementierung eines Datenspeichers mit geringerer Funktionalität als Cache, die, wie in 9.1.2.1 beschrieben, der Evaluierung von Cache dient).
- `de.emld.context.sources`: Beinhalten unter anderem Klassen, die Kontextquellen sowie Klassen die ihnen jeweils entsprechende Kontextinformationen implementieren (siehe 5.1.2 und 5.2). In diesem Paket sind nicht nur die in dieser Arbeit beschriebenen Kontextquellen und –informationen vorhanden, sondern auch andere, die von einem anderen SmartWeb Mitarbeiter implementiert wurden.
- `de.emld.context.ontological_processing`: Enthält die im Kapitel 8 genannte Klassen sowie einige andere, im Zusammenhang damit implementierte Hilfsklassen.

Anhang E: Konfigurationsdatei `settings.xml`

Folgend ist ein Auszug aus der Konfigurationsdatei `settings.xml` aufgeführt, genauer: derjenige Teil von ihr, in dem wichtige Einstellungen vorgenommen werden können. Diese Datei befindet sich im Paket `de.emld.context.ontological_processing`

Innerhalb von Tag `<ontology_destinations>` werden Dateinamen der Grund-, DnS- und Deskriptionsontologie übermittelt.

Innerhalb von Tag `<options>` werden können einige Option aktiviert (Angabe „`yes`“) und deaktiviert (Angabe „`no`“) werden, die sich auf das Programmierwerkzeug aus dem Kapitel 8 beziehen. So kann die im Ansatz implementierte Prädiktierfähigkeit gemäß dem Algorithmus 7.6.2 wahlweise erlaubt werden (`<predict_mode>`). Selektierte Deskriptionen können gegebenenfalls als relevant akzeptiert werden (`<accept_selected_descriptions>`). Nach Bedarf kann die jeweilige Situation mit spezialisierten Individuen (bzw. sie beinhaltenden A-Boxen) ergänzt werden (`<build_specifying_model>`). Tag `<complete_reasoning_in_descriptions>` gilt dem Kapitel 7.6.3 bzw. der dort gemachten Anmerkung bezüglich der Zeile 6 des Algorithmus aus 7.6.2. In `<remove_substituted_classes>` wird festgelegt, ob die präzisierten Klassen von Individuen der Situation durch die vom Präzisierungsalgorithmus inferierten präzisierenden Klassen ersetzt werden sollen, bzw. genauer: ob die explizite Zugehörigkeit besagter Individuen zu den Ersten gelöscht werden soll. Schließlich regelt `<precise_from_situation>`, ob die Präzisierung aus in der Situation vorhandenen Individuen erwünscht ist (siehe 6.7.3).

Im Tag `<options_for_specializing>` werden allgemeine optionale (`<case_specific_options>`) und optionale Bedingungen aus 1.II (`<ground_options>`) für die Spezialisierung festgesetzt (siehe 6.5.6). Die in 6.5.6 mit a) bezeichnete allgemeine optionale Bedingung entspricht `<specializing_class_not_in_specialized description>`; Bedingung b) – `<specializingdescriptionnotsatisfied>`; Bedingung 1.IIc) – `<specializingandspecializeddescriptionsarenotthesame>`; Bedingung 1.IId) – `<specializingandspecializedclassallocatablebysame _source>`.

```
<settings>
  <ontology_destinations>
    <ground_ontology>
      src\\de\\emld\\context\\ontological_processing\\swinto.owl
    </ground_ontology>
    <dns_ontology>
      src\\de\\emld\\context\\ontological_processing\\ExtDnS_397.owl
    </dns_ontology>
    <descriptions_ontology>
      src\\de\\emld\\context\\ontological_processing\\Pronto_me.owl
    </descriptions_ontology>
  </ontology_destinations>
  <options>
    <predict_mode>
      no
    </predict_mode>
    <accept_selected_descriptions>
      yes
    </accept_selected_descriptions>
    <build_specifying_model>
      yes
    </build_specifying_model>
    <complete_reasoning_in_descriptions>
      yes
    </complete_reasoning_in_descriptions>
    <remove_substituted_classes>
       yes
    </remove_substituted_classes>
    <precise_from_situation>
      yes
    </precise_from_situation>
  </options>
  <options_for_specializing>
    <ground_options>
      <specializing_class_not_in_specialized_description>
        no
      </specializing_class_not_in_specialized_description>
      <specializing_description_not_satisfied>
        no
      </specializing_description_not_satisfied>
    </ground_options>
    <case_specific_options>
      <specializing_and_specialized_class_allocatable_by_same_source>
        no
      </specializing_and_specialized_class_allocatable_by_same_source>
      <specializing_and_specialized_descriptions_are_not_the_same>
        no
      </specializing_and_specialized_descriptions_are_not_the_same>
    </case_specific_options>
  </options_for_specializing>
</settings>
```

Literaturverzeichnis

[1] Berenike Loos, Robert Porzel: Towards Ontology-based Pragmatic Analysis. In: Claire Gardent and Bertrand Gaiffe, editors, *Proceedings of 9th Workshop on the Semantics and Pragmatics of Dialogue (Dialor05)*, Nancy, France, June 9-11, 2005, S. 163-166

[2] Robert Porzel, Hans-Peter Zorn, Berenike Loos, Rainer Malaka (EML): Towards a Separation of Pragmatic Knowledge and Contextual Information. In: *Proceedings of the ECAI-06 Workshop on Contexts and Ontologies*, Riva del Garda, Italy, August 2006

[3] Gangemi A., Borgo S., Catenacci C., Lehmann J.: Task Taxonomies for Knowledge Content. *Deliverable of the EU FP6 project "Metokis" (D07)*

[4] A Gangemi, P Mika: Understanding the Semantic Web through Descriptions and Situations. In: R Meersman et al. (eds.), *Proceedings of ODBASE03 Conference*, 2003

[5] Baldauf M., Dustdar S.: A Survey on Context-aware systems. In: Vienna University of Technology, *Technical report*, TUV-1841-2004-24, 2004

[6] Gustavsen, R.M.: Condor – an application framework for mobility-based context-aware applications. In: *Proceedings of the Workshop on Concepts and Models for Ubiquitous Computing*, Goeteborg, Sweden, 2002

[7] Prekop, P., Burnett, M.: Activities, context and ubiquitous computing. In: *Special Issue on Ubiquitous Computing Computer Communications*, Vol. 26, No. 11, 2003, S. 1168–1176

[8] Bill Schilit, Norman Adams, Roy Want: Context-aware computing applications. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, December 1994. IEEE Computer Society Press. S. 85-90

[9] Anind K. Dey, Gregory D. Abowd: Towards a Better Understanding of context and context-awareness. In: Georgia Institute of Technology, College of Computing *Technical Report*, GIT-GVU-99-22, June 1999

[10] Guanling Chen, David Kotz: A Survey of Context-Aware Mobile Computing Research. In: Dartmouth College, Dept. of Computer Science, *Technical Report* TR2000-381, November, 2000

[11] Peter Mika et al: Foundations for OWL-S: Aligning OWL-S to DOLCE. In *Semantic*
Web Services: Papers from the 2004 AAAI Spring Symposium, AAAI Press, 2004, S. 52–59

[12] Oberle, D., Mika, P., Gangemi, A., Sabou, M.: Foundations for service ontologies: Aligning OWL-S to DOLCE. In: Staab S and Patel-Schneider P (eds.), *Proceedings of the World Wide Web Conference (WWW2004)*, Semantic Web Track, 2004

[13] Gangemi, A., Catenacci, C., Battaglia, M.: Inflammation Ontology Design Pattern: an Exercise in Building a Core Biomedical Ontology with Descriptions and Situations. In: D.M. Pisanelli (ed.) Ontologies in Medicine, IOS Press, Amsterdam, 2004

[14] Pisanelli, D.M., Gangemi, A., Steve, G.: An ontology of descriptions and situations for Lyee's hypothetical world. In: *Proceedings of Somet Workshop*, Stockholm, September 2003, S. 24-26

[15] A. Gugliotta, L. Cabral, J. Domingue, V. Roberto, M. Rowlatt, R. Davies: A Semantic Web Service-based Architecture for the Interoperability of E-government Services. In: *Proceeding of Web Information Systems Modeling Workshop (WISM 2005) in conjunction with The 5th International Conference on Web Engineering (ICWE 2005)* Sydney, Australia, 25-29 July, 2005

[16] A. K. Dey, G. D. Abowd: The Context Toolkit: Aiding the Development of Context-Aware Applications. In: *Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, 2000

[17] McCarthy, J. and Buvac, S. Formalizing context (expanded notes). In: Buvac, S. and Iwahska, L. (Eds.): *Working Papers of the AAAI Fall Symposium on Context in Knowledge Representation and Natural Language*, California. American Association for Artificial Intelligence, Menlo Park, 1997, S. 99–135

[18] Chen, H., Finin, T., Joshi, A.: An ontology for context-aware pervasive computing environments. In: *TheKnowledge Engineering Review*, Cambridge University Press, 2003, Vol. 18, S..197–207

[19] Winograd, T.: Architectures for context. In: *Human-Computer Interaction (HCI) Journal*, 2001, Vol. 16, No. 2, S. 401–419

[20] Indulska, J., Sutton, P.: Location management in pervasive systems. In: *CRPITS'03: Proceedings of the Australasian Information Security Workshop*, 2003, S. 143–151

[21] Strang, T., Linnhoff-Popien, C.: A Context Modeling Survey. In: *First International Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*, 2004

[22] Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A middleware infrastructure for active spaces. In: *IEEE Pervasive Computing*, 2002, Vol. 1, No. 4, S.74–83

[23] Gu, T., Pung, H.K. and Zhang, D.Q.: A middleware for building context-aware mobile services. In: *Proceedings of IEEE Vehicular Technology Conference (VTC 2004)*, Milan, Italy, 2004

[24] Ailisto, H., Alahuhta, P., Haataja, V., Kylloenen, V., Lindholm, M.: Structuring context aware applications: Five-layer model and example case. In: *Proceedings of the Workshop on Concepts and Models for Ubiquitous Computing*, Goteborg, Sweden, 2002

[25] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider, editors: The Description Logic Handbook – Theory, Implementation and Applications. Cambridge University Press, 2003

[26] Ian Horrocks, Ulrike Sattler: Description logics: Basics, inference algorithms, and applications. *Slides available at* http://www.cs.man.ac.uk/_horrocks/Slides/IJCARtutorial/, Tutorial given at the International Joint Conference on Automated Reasoning (IJCAR), Siena, 2001

[27] B. Grossof, I. Horrocks, R. Volz, S. Decker: Description Logic Programs: Combining Logic Programs with Description Logic. In: *Proceedings of WWW 2003*, Budapest, Hungary, May 2003

[28] Thi Dieu Thu Nguyen, Nhan Le Thanh: Modeling ORM Schemas in Description Logics. In: *Complex Systems Concurrent Engineering*, Springer London Verlag, 2007

[29] T D Thu Nguyen, N Le-Thanh. Identification constraints in *SHOIN*(**D**). In: *Proc. of the 1th Int. Conf. on Research Challeges in Information Science (RCIS 2007)*, 2007

[30] Michael Krause: Kontextbereitstellung in offenen, ubiquitären Systemen. Dissertation, LMU München: Faculty of Mathematics, Computer Science and Statistics, 2006

[31] Robert Tolksdorf, Elena Paslaru Bontas : Tutorial: Grundlagen des Semantic Web. *Slides available at* http://www.ag-nbi.de/lehre/0506/P_SW/

[32] Franz Baader: Everything you always wanted to know about description logics, but were afraid to ask your ontology engineer. Tutorial given at the 17th European Conference on Artificial Intelligence (ECAI 2006), Riva del Garda, Italy, 2006

[33] Robert Stevens, Carole A. Goble, Sean Bechhofer: Ontology-based knowledge representation for bioinformatics. In: *Briefings in Bioinformatics*, Vol. 1, No. 4, 2000. S. 398–414

[34] OWL Web Ontology Language: Overview. W3C Recommendation, TheWeb Ontology Working Group – The W3 Consortium, 2004. *Available at* http://www.w3.org/TR/owl-features/

[35] OWL Web Ontology Language: Guide. W3C Recommendation, TheWeb Ontology Working Group – The W3 Consortium, 2004. *Available at* http://www.w3.org/TR/owl-guide/

[36] OWL Web Ontology Language: Reference. W3C Recommendation, TheWeb Ontology Working Group – The W3 Consortium, 2004. *Available at* http://www.w3.org/TR/owl-ref/

[37] OWL Web Ontology Language: Semantics and Abstract Syntax. W3C Recommendation, TheWeb Ontology Working Group – The W3 Consortium, 2004. *Available at* http://www.w3.org/TR/owl-semantics/

[38] D. Oberle, A. Ankolekar, P. Hitzler, P. Cimiano, M. Sintek, M. Kiesel, B. Mougouie, S. Vembu, S. Baumann, M. Romanelli, P. Buitelaar, R. Engel, D. Sonntag, N. Reithinger, B. Loos, R. Porzel, H.-P. Zorn, V. Micelli, C. Schmidt, M. Weiten, F. Burkhardt, J. Zhou: DOLCE ergo SUMO: On Foundational and Domain Models in SWIntO (SmartWeb Integrated Ontology). In: *Journal of Web Semantics*, July 2006

[39] Philipp Cimiano, Andreas Eberhart, Pascal Hitzler, Daniel Oberle, Steffen Staab, Rudi Studer. The SmartWeb Foundational Ontology. *SmartWeb Project Report*, September 2004

[40] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari: Ontology Library (final). *WonderWeb Deliverable D18*, Dezember 2003. http://wonderweb.semanticweb.org

[41] Guarino, Nicola: Formal Ontology and Information Systems. In: *Proceedings of the first international conference on Formal ontology in Information Systems (FOIS)*, Trient, Italien, 1998, Seiten 3–15. IOS Press, Amsterdam

[42] Gruber, Thomas R.: A translation approac to portable ontology specifications. In: *Knowledge Acquisition, 5(2)*, Juni 1993. Seiten 199–220

[43] Stuart Russell, Peter Norwig: Artificial intelligence: a modern approach. Second Edition. Prentice Hall Verlag, 2003

[44] Wolfgang Wahlster: SmartWeb - Ein multimodales Dialogsystem für das semantische Web. In: Reuse, B., Vollmar, R.: *40 Jahre Informatikforschung in Deutschland*, Heidelberg, Berlin, Springer Verlag, 2007

[45] All you need is your mobile device: An information portal combines a map and a tourist guide with up-to-the-minute information on the city. *European Media Laboratory GmbH, Report*, 2006

[46] Berenike Loos, Hans-Peter Zorn: Combining Information Extraction and Knowledge Acquisition for Spoken Dialog Systems. 2007

[47] Korpipää, P., Mäntyjärvi, J.: "An ontology for mobile device sensor-based context awareness". In: *Proceedings of CONTEXT, 2003*, Vol. 2680 of *Lecture Notes in Computer Science*, 2003, S. 451–458

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

[49] T. Furche: RDF: Resource Description Framework. Tutorial, 2001, *available at* http://www.furche.net/projects/2001/rdf-tutorial/

[50] Manola, F., E. Miller, B. McBride: RDF Primer. W3C Recommendation, Februar 2004, *available at* http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

[51] Beckett, D. und B. McBride: RDF/XML Syntax Specification (Revised). W3C Recommendation, Februar 2004, *available at* http://www.w3.org/TR/rdf-syntax-grammar/

[52] Jos de Bruijn, Sergio Tjessaris, Enrico Franconi: Logical reconstruction of RDF and ontology languages. In: *Proc. of PPSWR-05*, 2005

[53] Andy Seabone: Jena Tutorial: A Programmer’s Introduction to RDQL. April 2002. *Availaible at* http://jena.sourceforge.net/tutorial/RDQL

[54] Patrick Hayes, Brian McBride: RDF Semantics. W3C Recommendation Februar 2004. *Availaible at* http://www.w3.org/TR/2004/REC-rdf-mt-20040210/