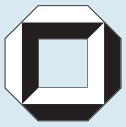
Michael Klein

Automatisierung dienstorientierten Rechnens durch semantische Dienstbeschreibungen



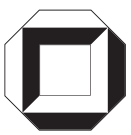
universitätsverlag karlsruhe

Michael Klein

Automatisierung dienstorientierten Rechnens durch semantische Dienstbeschreibungen

Automatisierung dienstorientierten Rechnens durch semantische Dienstbeschreibungen

von
Michael Klein



universitätsverlag karlsruhe

Dissertation, Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik, 2006

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2006
Print on Demand

ISBN 3-86644-013-8

Automatisierung dienstorientierten Rechnens durch semantische Dienstbeschreibungen

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena

von Dipl.-Inform. Michael Klein
geboren am 12.03.1976 in Traben-Trarbach

Erstgutachterin: Prof. Dr. Birgitta König-Ries

Zweitgutachter: Prof. Dr. Rudi Studer

Tag der letzten Prüfung des Rigorosums: 9. Februar 2006

Tag der öffentlichen Verteidigung: 13. Februar 2006

Geleitwort

Betrachtet man das heutige World Wide Web, so sind Weiterentwicklungen entlang von zwei Dimensionen nahe liegend:

Zum einen ist es wünschenswert, über diese Infrastruktur nicht nur Informationen verfügbar zu machen, sondern nahtlos auch den Zugriff auf Funktionalität, die von anderen Rechnern bereit gestellt wird, zu ermöglichen. *Web Services* ermöglichen die Erfüllung dieses Wunsches. Ein Webservice ist ein Stück Software, das über das Internet zugreifbar ist, das seine Funktionalität und seine Schnittstellen in Beschreibungen veröffentlicht und dessen Funktionalität mittels XML-basierter Nachrichten, die über Internet-Prtokolle ausgetauscht werden, aufrufbar macht.

Zum anderen ist es wünschenswert, Information in einer nicht nur für Menschen, sondern auch für Maschinen verständlichen Form verfügbar zu machen. Entwicklungen im Bereich des *Semantic Web* versuchen dieses Ziel zu erreichen. Grundidee hier ist es, Informationen mittels Ontologien semantisch zu annotieren.

Kombiniert man beide Dimensionen, so erhält man als Ziel ein System, in dem Funktionalität in Form von *Semantic Web Services* für Maschinen automatisch, ohne menschliches Eingreifen nutzbar wird. Hier handelt es sich um ein Thema, das in den letzten Jahren zunehmend an Bedeutung gewonnen hat.

Das vorliegende Werk greift einen – wenn nicht *den* – zentralen Aspekt aus den zahlreichen Problemen, die im Zusammenhang mit Semantic Web Services zu lösen sind, heraus: die Bereitstellung einer geeigneten Sprache zur Dienstbeschreibung. Eine solche Sprache muss eine Reihe von Anforderungen erfüllen – und diese Anforderungen sind, wie es unangenehmerweise häufig bei Anforderungen der Fall ist, teils widersprüchlich: Auf der einen Seite wird eine Sprache benötigt, die ausdrucksstark genug ist, um beliebige Dienstangebote und -anfragen präzise beschreiben zu können. Auf der anderen Seite muss es möglich sein, diese Beschreibungen effizient zu verarbeiten, insbesondere muss der automatische, semantisch korrekte Abgleich von Dienstangeboten und -anfragen möglich sein. Zu guter Letzt muss die Sprache auch noch praktisch einsetzbar sein, was insbesondere erfordert, dass der Aufwand, den ein Nutzer betreiben muss, um die Sprache verwenden zu können, sich in Grenzen hält.

Existierende Ansätze müssen hier oft unbefriedigende Kompromisse eingehen: Was nützt beispielsweise eine Sprache, mit der Dienste detailliert beschrieben werden

können, die jedoch unentscheidbar ist und es somit unmöglich macht, alle Information, die in die Dienstbeschreibung geflossen ist, auch auszuwerten, um zu einem Angebot möglichst gut passende Dienste zu finden?

Michael Klein gelingt es in seiner Arbeit, eine Balance zwischen diesen drei Anforderungen zu finden. Er schlägt eine Sprache vor, die gerade mächtig genug ist, um relevante Aspekte von Diensten zu beschreiben, die einfach genug ist, so dass alle in Beschreibungen enthalten Informationen auch tatsächlich für den Abgleich zwischen Angeboten und Anfragen genutzt werden können und die einen gangbaren Kompromiss zwischen vorab zu Leistendem und Verarbeitbarkeit eingeht.

Mit dieser Sprache ist es nun möglich – zumindest für relativ einfach aufgebaute Dienste – Anfragen und Angebote präzise zu beschreiben und automatisch gegeneinander abzugleichen. Somit wird eine *vollautomatische* Dienstnutzung Realität.

Die Sprache ist so gestaltet, dass eine geradlinige Erweiterung für komplexere Fälle problemlos möglich, ja zum jetzigen Zeitpunkt sogar schon in Arbeit ist. Sie liefert somit eine wertvolle Grundlage für hoffentlich viele weiterführende Arbeiten im Umfeld der automatischen Dienstnutzung.

Es bleibt mir, der Arbeit einen breiten Leserkreis und einen nachhaltigen Einfluss auf die weitere Forschung im Bereich der Semantic Web Services zu wünschen.

Jena, im Februar 2006 Birgitta König-Ries

Vorwort und Danksagung

Das vorliegende Buch stellt meine Dissertation dar, die während meiner Arbeit am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe, Lehrstuhl Prof. Lockemann bzw. Prof. Böhm im Rahmen des Projekts DIANE (Dienste in Ad-hoc-Netzen), einem Teilprojekt des DFG-Schwerpunktprogramms SPP1140 „Basissoftware für Selbstorganisierende Infrastrukturen für Vernetzte Mobile Systeme“ entstanden ist. Zum Gelingen der Arbeit haben eine Vielzahl von Personen beigetragen, bei denen ich mich an dieser Stelle bedanken möchte.

Zunächst möchte ich meiner Erstgutachterin *Prof. Dr. Birgitta König-Ries* danken. Als ihr erstes „Doktorkind“ genoss ich eine besonders intensive Betreuung. Stets stand sie für Diskussionen zu Fragestellungen der Arbeit zur Verfügung. Auch ihr ständiges Daraufpochen, die Ideen der Dissertation in Form von Papieren zu veröffentlichen, war zwar anfänglich stressig, stellte sich rückblickend jedoch als enorm vorteilhaft für das Vorankommen und die Qualität der Arbeit heraus. Ihren oft benutzten Satz „Wenn man eine Idee hinschreiben muss, wird man gezwungen, sie sich klar zu machen“, würde ich heute sofort unterschreiben. Und die mit den Veröffentlichungen verbundenen Konferenzbesuche in aller Welt waren ein angenehmer Nebeneffekt. Birgitta König-Ries betreute jedoch nicht nur meine Dissertation. Bei allen Arbeiten am Institut – war es die Durchführung von Praktika und Seminaren, die Betreuung von Studien- und Diplomarbeiten, das Schreiben von Anträgen oder die Organisation von Workshops – stand sie mir tatkräftig zur Seite.

Daneben will ich meinem Büro- und Projektkollegen *Philipp Obreiter* danken, mit dem ich die Höhen und insbesondere auch die Tiefen des nicht immer ganz einfachen Mitarbeiteralltags durchgemacht habe. Neben zahlreichen Ratschlägen, Warnungen, Tipps, Hinweisen und Vorschlägen für meine Forschungen, habe ich von ihm vor allem auch viel Nicht-Fachliches für das Leben außerhalb der Universität gelernt: im Bereich des Teekochens, im Bereich neuer Musik, Kriegsführung, chinesischer Schriftzeichen, oder Spieltheorie. Seine Mottos vom „Explizitmachen“ und „Trade-offs-Erkennen“ habe ich nach anfänglicher Skepsis übernommen.

Zu bedanken habe ich mich ebenfalls bei vier Professoren: Zunächst *Prof. Dr. Studer*, der freundlicherweise das Korreferat der Dissertation übernommen und viel Interesse an der Arbeit gezeigt hat. Dann bei *Prof. Dr. Dr. h.c. Lockemann*, der die Betreuung meiner Arbeit vor der Berufung von Prof. König-Ries übernommen hatte, bei

Prof. Dr. Alt, der den Vorsitz meiner Prüfungskommission in Jena übernommen hat, sowie bei *Prof. Dr. Böhm*, dem Nachfolger von Prof. Lockemann, an dessen Institut ich ab 2005 arbeiten durfte.

Dank gilt aber natürlich auch all meinen Studierenden, deren Ergebnisse aus Studien- und Diplomarbeiten in meine Dissertation eingeflossen sind. Ganz besonders möchte ich folgende Personen erwähnen: Erstens *Michael Müssig*, der in seiner Diplomarbeit wichtige Teile des Vergleichers konzipiert und implementiert hat und die sogar innerhalb eines gemeinsamen Zeitschriftenartikels veröffentlicht wurde. Zweitens *Mirco Stern*, der in seiner Studienarbeit einen Generator für Dienstbeschreibungen entwickelte, mit dessen Hilfe die Korrektheit der implementierten Komponenten überprüft werden konnte, und in seiner Diplomarbeit bereits weiterführende Arbeiten auf dem Gebiet der Dienstkomposition basierend auf dem entwickelten Ansatz betrachtete. Drittens *Thomas Fischer*, der in seiner Studienarbeit die Komponente zur Ausführung von Diensten erstellte und sich in seiner Diplomarbeit mit der Durchführung von Experimenten um wichtige Teile der Evaluation kümmerte. Weiterhin danke ich *Holger Schmitt*, der in seiner Diplomarbeit erste Ideen zur Ontologieverwaltung untersuchte, *Daniel Matheis*, in dessen Studienarbeit der entstandene Ansatz zur Implementierung einer mobilen Anwendung zur Gruppenkooperation eingesetzt wurde, *Thomas Herzog*, der in seiner Studienarbeit ein Werkzeug erstellte, das es einem Benutzer erlaubt, Beschreibungen der entwickelten Sprache auf graphischem Wege einzugeben, sowie *Frank Schell*, *Christoph Schaa* und *Thorsten Höllrigl*, die in ihrer gemeinsamen Studienarbeit erste Vorschläge für die formale Notation der Beschreibungssprache machten.

Auch meine Arbeitskollegen haben Dank verdient. Zuerst genannt sei hier *Jutta Mülle*, die „Managerin des Instituts“, die ich stets zu allen organisatorischen und fachlichen Dingen rund um das IPD fragen konnte und die auch immer eine hilfreiche Antwort parat hatte. Wichtig für eine erfolgreiche Dissertation ist auch ein angenehmes Arbeitsumfeld. Hierfür will ich allen meinen Kollegen danken, vor allem *Philipp Bender*, *Heiko Schepperle*, *Jens Nimis*, *Daniel Pfeifer*, *Antje Dietrich*, *Khaldoun Ateyeh* und *Björn-Oliver Hartmann*, mit denen ich mich insbesondere beim alltäglichen Kaffee in der Mensa austauschen konnte. Und Dank gilt natürlich auch den Sekretärinnen *Frau Götz*, *Frau Horcic*, *Frau Wegl* und *Frau Weisenburger*, die mir immer freundlich bei allen organisatorischen Dingen am Institut halfen.

Nicht zuletzt haben aber auch Personen außerhalb des universitären Umfelds dazu beigetragen, dass ich diese Arbeit verwirklichen konnte. Da sind zunächst meine Eltern *Margit* und *Berthold*, meine *Großeltern* sowie *Matthias* und *Linda*, die immer an die Realisierung meiner Pläne geglaubt und mich entsprechend unterstützt haben. Dank gilt aber auch allen meinen *Freunden* in Karlsruhe und im Hunsrück, die mich immer wieder von meiner Arbeit abgelenkt und so für den nötigen Ausgleich gesorgt haben.

Karlsruhe, im März 2006 Michael Klein

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I.

Ausgangspunkt

Die Grenzen meiner Sprache
bedeuten die Grenzen meiner Welt.
Ludwig Wittgenstein
(1889-1951)

1. Einleitung

1.1. Motivation

Rechner wurden entwickelt, um nützliche Funktionen für den Menschen auszuführen. Zu Beginn stellten sie in sich abgeschlossene Systeme dar, welche der Benutzer über direkt angeschlossene Eingabe- oder Peripheriegeräte mit Daten versorgte. Diese wurden dann lokal verarbeitet und anschließend ausgegeben. Der Rechner war nur in der Lage die Funktionalität auszuführen und die Daten zu verarbeiten, die zuvor explizit eingespielt wurden.

Mit der Möglichkeit zur Vernetzung mehrerer Rechner änderte sich die Situation. Systeme waren fortan nicht mehr isoliert, sondern konnten Daten untereinander auszutauschen. Die Vernetzung der lokalen Netze zum Internet erlaubte schließlich eine institutionsübergreifende, weltweite Kommunikation. Ein wichtiges Anwendungsgebiet stellte der Austausch von (zunächst vorwiegend wissenschaftlichen) Dokumenten im Rahmen des *World Wide Web* dar. Vor allem durch die plattformunabhängige und einfach zu verwendende Dokumentensprache HTML waren Benutzer in der Lage, ihre Dokumente öffentlich verfügbar zu machen, um so von anderen Benutzern aufgefunden, über das Netz geladen und lokal weiterverarbeitet werden zu können. War die Menge der Dokumente zu Beginn noch so gering, dass diese alleine über Verweise untereinander auffindbar waren, stiegt ihre Anzahl in kürzester Zeit enorm an:

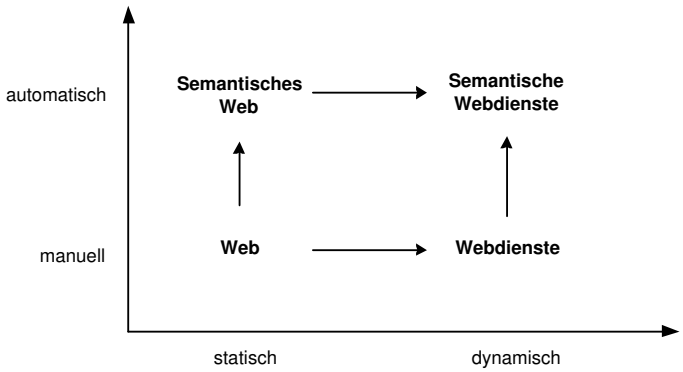


Abbildung 1.1.: Vom Web zu Semantischen Webdiensten.

Zusätzliche Suchmechanismen wurden unerlässlich. Da die angebotenen Dokumente in der Regel nicht durch zusätzliche Metadaten beschrieben waren, beschränkten sich die Suchmöglichkeiten auf Stichworte oder einfache Eigenschaften der Dateien. Die Präzision solcher Suchmaschinen stieg zwar stetig an, erreichte jedoch bis heute eine nur mäßige Präzision, was Benutzer in den meisten Fällen dazu zwingt, die gelieferte Trefferliste durchzusehen, um anschließend geeignet erscheinende Dokumente anzufordern.

Die Vernetzung von Rechnern eignete sich jedoch nicht nur zum Austausch von Dokumenten, auch die Verwendung von externer Funktionalität, die lokal nicht zur Verfügung stand, wurde möglich. Das World Wide Web entwickelte sich daher von statischen zu dynamischen Ressourcen (siehe Abbildung 1.1). Stand anfänglich noch die Nutzung von Funktionalität innerhalb der eigenen Institution im Mittelpunkt, so zielten *Webdienste* (engl. web services) auch auf eine unternehmensübergreifende Verwendung entfernter Dienste. Ein Dienst ist dabei eine hinter einer Schnittstelle verborgene öffentliche Funktionalität, welche von anderen Teilnehmern des Systems über das Netzwerk aufgefunden und aufgerufen werden kann. Dies erlaubt es einem einzelnen Rechner, ein wesentlich größeres Spektrum an Funktionalität bereitzustellen, als es isoliert möglich wäre. Im Gegensatz zu Dokumenten werden Webdienste häufig mit zusätzlichen Metadaten, der Dienstbeschreibung umschrieben, welche ein effizientes Auffinden von Diensten ermöglichen soll. Nach wie vor muss der menschliche Dienstnutzer jedoch aus einer Liste vorgeschlagener Dienste einen geeigneten auswählen, diesen anpassen und in die Applikation integrieren.

Neben dem Fortschreiten von statischen zu dynamischen Ressourcen entwickelte sich das World Wide Web unter dem Namen des *Semantischen Webs* (engl. Semantic Web) auch in einer anderen Dimension. Die Vision ist ein Netz, in dem Dokumente nicht nur in menschenverständlicher Form vorliegen, sondern auch so für Rechner verstehbar sind, dass diese intelligente Operationen darauf ausführen können. Insbesondere soll damit ein automatisches Finden und Verwenden von benötigter Information ermöglicht werden. Zur Erreichung dieses Ziels werden die Metadaten für Dokumente so erweitert, dass sie explizit Bezug auf eine zusätzliche, rechnerverarbeitbare Beschreibung der realen Welt nehmen. Hierzu steht aus dem Bereich der künstlichen Intelligenz eine Reihe von Techniken zur Wissensrepräsentation zur Verfügung, allen voran Ontologien. Rechner sollen durch Rückgriff auf dieses Wissen sowohl automatisch als auch semantisch korrekt mit Dokumenten umgehen können.

Es liegt nun nahe, beide Dimensionen miteinander zu verbinden, um so auch Funktionalität in Form von Webdiensten automatisch und semantisch korrekt nutzen zu können. Die Forschung um *Semantische Webdienste* (engl. Semantic Web Services) gewinnt daher zunehmend an Bedeutung. Die Vision ist ein offenes System von Teilnehmern, in welchem Dienste automatisch, dynamisch und abhängig von den aktuellen Gegebenheiten genutzt, d.h. gefunden, konfiguriert und aufgerufen werden

können. Semantische Webdienste ermöglichen so verteilte Applikationen und unternehmensübergreifende Geschäftsprozesse, die sich durch eine wesentlich erhöhte Robustheit, Effizienz und Kontextbeachtung auszeichnen, da ungeeignete Dienstgeber zur Laufzeit ausgetauscht werden können. Wie schon beim Übergang zum Semantischen Web ist eine rechnerverständliche Beschreibung der Dienstfunktionalität der kritische Erfolgsfaktor. Eine Beschreibung, die die vom Dienst ausgetauschten Dokumente semantisch beschreibt, reicht dazu jedoch nicht. Vielmehr muss den Besonderheiten von Diensten durch spezielle Beschreibungselemente Rechnung getragen werden. Nur so wird es Rechnern möglich sein, benötigte Funktionalität vollständig automatisch und wunschgemäß zu finden und durch externe Dienstgeber ausführen zu lassen.

Im Mittelpunkt dieser Arbeit stehen daher Ansätze, die Vision Semantischer Webdienste Realität werden zu lassen. Konkret werden eine neuartige semantische Dienstbeschreibungssprache und ein Rahmenwerk zu deren Verarbeitung vorgestellt, mit deren Hilfe eine vollständige Automatisierung der Nutzung bestimmter Webdienste in dynamischen Umgebungen erreicht werden kann.

1.2. Szenarios

Die Möglichkeiten automatisierter, semantischer Dienstnutzung werden im Folgenden durch zwei Szenarios verdeutlicht. Aus diesen Szenarios werden dann die Ziele und Anforderungen dieser Arbeit abgeleitet.

1.2.1. Szenario 1: Internetbasierte Geschäftsprozesse

Viele Leistungen von Unternehmen sind heute in Geschäftsprozessen (engl. business processes) organisiert. Diese untergliedern sich in eine Reihe von Prozessschritten, die in miteinander miteinander verkettet sind. In jedem dieser Schritte ist eine bestimmte Operation durchzuführen und gegebenenfalls eine Entscheidung über den weiteren Verlauf des Gesamtprozesses zu treffen (siehe Abbildung 1.2). Eine Operation wird in der Regel durchgeführt, indem ein Aktor angestoßen wird, welcher eine Software, ein Gerät aber auch ein menschlicher Mitarbeiter sein kann.

Betrachten wir als Beispiel die Beauftragung eines Telefonanschlusses bei einer Telefongesellschaft. Mit dem Eingang des Auftrags im Unternehmen durchläuft der Vorgang einen definierten Prozess (der jedoch durch bestimmte Bedingungen vorzeitig beendet werden kann):

1. Überprüfung, ob der gewünschte Anschluss am angegebenen Ort verfügbar ist.

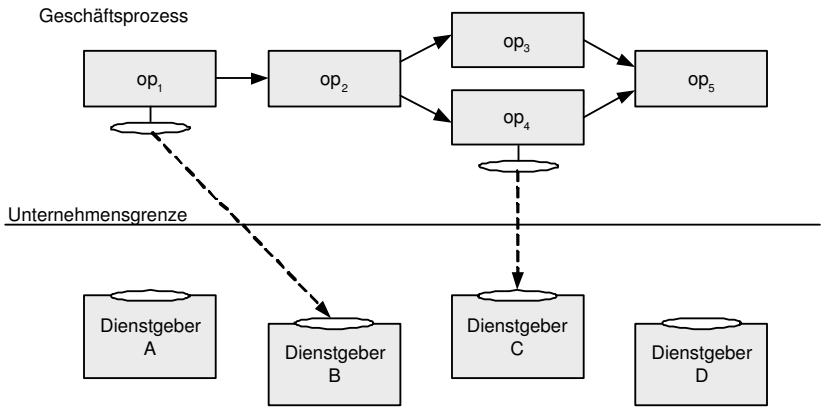


Abbildung 1.2.: Dienstorientierte Einbindung von externer Funktionalität in Geschäftsprozesse durch dynamische Dienstbindung.

2. Reservierung einer oder mehrerer freier Telefonnummern.
3. Verbindung der „letzten Meile“ vom Teilnehmer zum Firmennetz (evtl. über Drittanbieter).
4. Bereitstellung des Anschlusses in der Wohnung des Teilnehmers durch einen Techniker.
5. Eintrag des Teilnehmers in das Telefonbuch, falls gewünscht.
6. Beschaffung und Versand der vom Teilnehmer bestellten Endgeräte.
7. Neueintrag des Teilnehmers in die Buchhaltung.
8. Einzug der Anschlussgebühren vom Konto des Auftraggebers.

Dennoch soll oder kann nicht jede benötigte Funktionalität selbst erbracht werden. Gründe dafür können fehlende Ressourcen oder Kompetenzen sowie zu hohe Kosten sein. Es bietet sich daher an, externe Aktoren in Form von Dienstgebern in den Prozess einzubinden. Eine solcher *dienstorientierter Aufruf* zwischen einem Prozessschritt als Dienstnehmer und einem externen Aktor als Dienstgeber hat den wesentlichen Vorteil, dass der Aktor die Details seiner Leistungserbringung hinter einer Schnittstelle verbergen kann. Hierdurch wird es möglich, dass technisch unterschiedliche Systeme zusammenarbeiten können. Wichtig ist zunächst nur, dass ihre Funktionalität über eine entfernt aufrufbare Schnittstelle angestoßen werden kann und anhand einer Beschreibung ersichtlich ist, welche Funktionalität dadurch erbracht wird. Beispiele für solche externe Dienstgeber könnten sein:

- Ein Dienst, der die Verfügbarkeit für ISDN- und DSL-Anschlüsse für eine gegebene Wohnung überprüfen kann.

- Ein Techniker, der im Raum Karlsruhe Telefonanschlüsse freischalten kann, die Eigentum der Deutschen Telekom sind, und der sich gerade im Stadtteil x aufhält.
- Ein Dienst, um eine Leitung vom Knotenpunkt k zu einem gegebenen Anschluss in den Straßen x, y oder z zu einem bestimmten Preis nutzen zu können.
- Ein Dienst, der eine geschäftliche Telefonnummer in die Gelben Seiten einträgt.
- Ein Dienst, der Waren mit einem Gewicht von bis zu 1 kg zu einem Preis von x Euro mit einer Versanddauer von 7 Tagen verschicken kann.
- Ein Dienst, bei dem Telefone des Herstellers Siemens gekauft werden können.

Schon heute werden im Rahmen der *Enterprise Application Integration* (EAI) mittels der vorhandenen Technologien um Webdienste externe Dienstgeber eingebunden. Charakteristisch sind jedoch statische Verbindungen, d.h. ein Prozessschritt wird zur Entwicklungszeit fest an einen Dienstgeber gebunden. Hierdurch entsteht eine Reihe von Problemen zur Laufzeit: Es kann zu mangelnder Funktionalität kommen, wenn bestimmte Dienstanbieter nicht eingebunden wurden. So könnte ein Dienstgeber einen ISDN-Anschluss beispielsweise nur für bestimmte Regionen anbieten. Zudem können effizientere Dienstanbieter übersehen werden, etwa ein schnellerer Paketversender oder ein preislich günstigerer Bankeinzug. Insbesondere leidet auch die Robustheit des Gesamtprozesses, wenn einzelne, fix eingebundene Dienstgeber ausfallen. Zuletzt kann auch der aktuelle Kontext nur unzureichend betrachtet werden, etwa die Auswahl eines geeigneten Technikers, der sich möglichst in der Nähe des Anschlussortes befinden sollte.

Vorteile bei der dienstorientierten Integration von externer Funktionalität ergeben sich daher insbesondere dann, wenn eine fallweise Zuordnung von Dienstnehmern und Dienstgebern zur Laufzeit erfolgt. In diesem Fall kann zum Zeitpunkt der konkreten Ausführung des Prozessschrittes ein in diesem Moment und für diesen speziellen Zweck möglichst gut geeigneter Dienstgeber ausgewählt und entsprechend konfiguriert werden. Der Aufruf erfolgt über die Schnittstelle des Dienstes, wodurch die reale Diensterbringung etwa durch eine Software, einen weiteren Prozess oder einen menschlichen Aktor ausgelöst wird.

Eine solche fallweise Auswahl von Dienstgebern ist jedoch manuell nicht mehr sinnvoll machbar. Insbesondere in großen Geschäftsprozessen werden die einzelnen Prozessschritte sehr häufig und in wechselnder Konfiguration durchlaufen. Die vollständige Automatisierung der Dienstnutzung (bestehend aus Vergleich, Auswahl und Aufruf eines geeigneten) Dienstgebers ist daher zwingend, um eine dynamische Dienstbindung in der Praxis möglich zu machen.

1.2.2. Szenario 2: Mobile Anwendungen

Ein zweites Szenario zeigt die Wichtigkeit dynamischer Dienstnutzung in mobilen Umgebungen. Betrachtet werden soll eine Anwendungen, die auf kleinen mobilen Geräten wie Handys oder Personal Digital Assistants (PDAs) läuft, welche über drahtlose Kommunikationsmöglichkeiten verfügen. Ziel ist eine mobile Office-Anwendung, die über den Funktionsumfang einer klassischen Anwendung für Desktop-PCs verfügt. Beispielsweise könnte das `Datei`-Menü der mobilen Präsentationssoftware Funktionen bieten, um die geöffnete Datei auszudrucken, auf korrekte Rechtschreibung zu überprüfen, an andere Personen zu versenden, zu verschlüsseln, zu speichern, auf eine Leinwand zu projizieren usw. Insgesamt sollte die Anwendung für den Benutzer also so zu bedienen sein,

- dass die *beschränkte Rechenleistung* kein Hindernis für aufwändige Operationen wie Verschlüsselung oder Rechtschreibprüfung darstellt.
- dass die *beschränkte Speicherkapazität* kein Hindernis für die Ablage großer Daten darstellt.
- dass *fehlende Peripheriegeräte* kein Hindernis für Operationen wie die Erstellung eines Ausdrucks und die Projektion darstellen.
- dass eine *fehlende direkte Internetverbindung* kein Hindernis für Operationen wie das Versenden von Dateien darstellt.

Die Vision ist also ein mobiles Gerät, dessen Leistungsfähigkeit sich nicht von der eines stationären Rechners unterscheidet. Neben informationstechnischen Funktionen könnten zudem nützliche Operationen auf realweltlichen Objekten angeboten werden, z.B. in einem Menü `Film` Funktionen zum Bezug von weiteren Informationen zu diesem Film, zur Reservierung einer Kinokarte für diesen Film, zum Kauf einer DVD mit diesem Film usw.

Wie auch im Falle internetbasierter Geschäftsprozesse bietet sich für mobile Anwendungen die Einbindung erreichbarer, externer Funktionalität in Form von Dienstgebern an. Gerade hier ist eine dynamische Bindung von Dienstgebern unerlässlich, da aufgrund der Bewegung des mobilen Geräts Dienstgeber leicht unerreichbar werden können und eine kontextabhängige Verwendung von Funktionalität sehr wichtig ist. Insbesondere die Einbeziehung des aktuellen Aufenthaltsorts sowie der aktuellen Situation des Benutzers verbieten die statische Anbindung von Dienstanbietern, sondern erfordern eine fallbezogene Auswahl geeigneter Funktionalität zur Laufzeit. Sinnvoll möglich wird dies jedoch nur, wenn der Benutzer nicht in den technischen Prozess der Dienstnutzung involviert wird, um etwa geeignete Dienstanbieter zu vergleichen

oder die zum Aufruf benötigten Nachrichten zu bestimmen. Gerade bei der Verwendung mobiler Geräte mit beschränkten Ein- und Ausgabemöglichkeiten könnte der hierdurch entstehende Aufwand die Nützlichkeit des Vorgehens zunichte machen. Im Falle mobiler Anwendungen muss also ebenso wie bei internetbasierten Geschäftsprozessen eine vollständig automatisierte Dienstnutzung angestrebt werden.

1.3. Ziel der Arbeit

Die beiden Szenarios zeigen, dass die dienstorientierte Integration externer Funktionalität insbesondere dann Vorteile bringt, wenn Dienstgeber fallbezogen und zur Laufzeit ausgewählt und gebunden werden und so eine *lose Kopplung* zwischen den Teilnehmern erreicht wird. Dies ermöglicht kontextbeachtende und robuste Anwendungen, die insbesondere in dynamischen Umgebungen wie den vorgestellten internetbasierten Geschäftsprozessen oder mobilen Anwendungen wichtig sind. Eine dynamische Bindung von Dienstgebern ist jedoch nur dann sinnvoll möglich, wenn der gesamte Prozess der Dienstnutzung bestehend aus Dienstsuche, -auswahl, -konfigurierung und -aufruf *vollständig automatisch* abläuft. Eine manuelle Dienstbindung ist im Falle repetitiver Dienstaufrufe mit wechselnden Anforderungen weder effizient noch skalierbar. Benötigt wird also ein System, welches eine automatisierte Dienstnutzung ermöglicht.

Grundlage eines solchen Systems ist eine *Dienstbeschreibungssprache* und geeignete Operatoren darauf, mit welcher der Inhalt angebotener und benötigter Dienste so genau notiert werden kann, dass es maschinell möglich wird, allein auf Grundlage dieser Beschreibungen semantisch passende Dienste auszuwählen und anzustoßen. Im Detail muss ein Rechner ist der Lage sein, folgende Fragen mithilfe der Dienstbeschreibung selbstständig zu beantworten:

- *Vergleich/Auswahl:* Gegeben ist eine Anfrage nach Funktionalität in Form eines Dienstes. Welcher Dienstanbieter leistet das Gewünschte?
- *Konfiguration/Aufruf:* Wie muss mit der Schnittstelle des ausgewählten Dienstgebers umgegangen werden?

Die Beschreibungssprache legt somit den Grad der Automatisierung fest. Ihre Semantik definiert unmittelbar die Funktionsweise der am System beteiligten Komponenten, insbesondere die des Vergleichers oder die der Ausführungskomponente. Ziel der Arbeit ist daher die Entwicklung einer Dienstbeschreibungssprache, mit deren Hilfe die Dienstnutzung in dynamischen Umgebungen vollständig automatisiert werden kann, ohne dabei an semantischer Präzision zu verlieren.

ZIEL DER ARBEIT:
Schaffung einer Dienstbeschreibungssprache als Grundlage für die Automatisierung der semantisch korrekten Dienstnutzung in dynamischen Umgebungen.

Um die Szenarios aus Abschnitt 1.2 verwirklichen zu können, muss die zu entwickelnde Sprache als Grundlage für ein System dienen können, welches folgende Eigenschaften besitzt:

1. *Universalität*, d.h. das System ist nicht auf Dienste bestimmter Domänen beschränkt, sondern ist prinzipiell in allen Anwendungsgebieten verwendbar. Gerade bei mobilen Anwendungen wird deutlich, dass hier ein und das selbe Geräte im Laufe der Zeit in unterschiedlichsten Domänen agieren soll, was durch die Beschreibungssprache nicht prinzipiell beschränkt werden darf.

2. *Vollständige Automatisierung* der Dienstnutzung im System, d.h. nach Bereitstellung der Dienstangebots- oder -anfragebeschreibung braucht kein menschlicher Benutzer mehr in den Prozess einzugreifen, um Dienstvergleich, -auswahl, -konfiguration oder -ausführung manuell zu steuern. Dies ist wichtig, um die Skalierbarkeit des Systems zu gewährleisten.

3. *Semantische Korrektheit* der Dienstnutzung im System, d.h. die Entscheidungen des Systems sind im Sinne der Benutzer und werden von diesen akzeptiert. Dies ist insbesondere deshalb wichtig, da kein menschlicher Benutzer in den Nutzungsprozess involviert ist.

4. *Vertretbarer Einigungsaufwand* vor Nutzung des Systems, d.h. vorbereitende Tätigkeiten insbesondere durch Festlegung eines gemeinsamen Vokabulars sollen angemessen aufwändig und machbar sein. Dies ist wichtig, da im Falle von Geschäftsprozessen relativ teure unternehmensübergreifende Einigungen nötig sind, im Falle mobiler Anwendungen meist eine vorangehende Einigung in nichtmobiler Umgebung.

5. Verwendbarkeit in *dynamischen Umgebungen*, d.h. eine Dienstnutzung ist auch dann möglich, wenn sich die Menge der angebotenen und benötigten Dienste zeitlich verändert und somit bei der Erstellung einer Dienstbeschreibung als unbekannt vorausgesetzt werden muss. Sowohl bei internetbasierten Geschäftsprozessen als auch bei mobilen Anwendungen muss von solchen Umgebungen ausgegangen werden.

1.4. Herausforderungen

Die Erfüllung der Anforderungen wird dadurch erschwert, dass diese gegensätzlich zueinander sind. Triviale Lösungen wie eindeutige Dienstkennzeichner, syntaktische

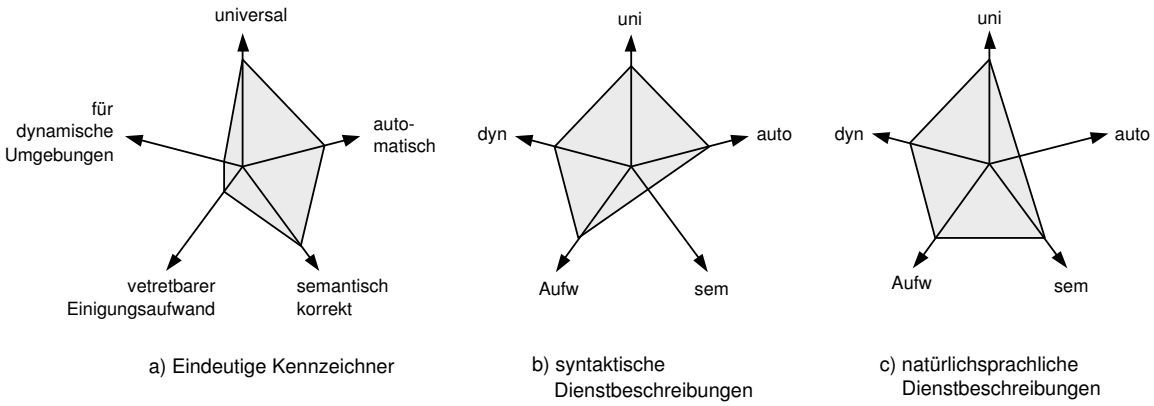


Abbildung 1.3.: Übersicht über die Erfüllung der Anforderungen der trivialen Ansätze.

Beschreibungen oder natürlichsprachliche Beschreibungen erfüllen daher nur einzelne Anforderungen. Sie sind jedoch nicht in der Lage, *alle Anforderungen gemeinsam* zu erfüllen:

- Ein vollständig automatisch und semantisch korrekt arbeitendes System könnte erreicht werden, indem alle möglichen Dienste mit *eindeutigen Kennzeichnern* versehen werden. Eine Anfrage stellt dann eine Auflistung geeigneter Kennzeichner dar; der Vergleicher überprüft lediglich, ob ein Dienstgeber einen dieser Kennzeichner im Angebot hat. Der Einigungsaufwand ist hier jedoch enorm, wenn nicht sogar unmöglich. Die Anforderung nach der Einsatzmöglichkeit in dynamischen Umgebungen erfordert zudem, dass auch Kennzeichner für zukünftige, noch unbekannte Dienste vergeben werden müssten (siehe Abbildung 1.3a).
- Ein vollständig automatisches System, das mit geringem Einigungsaufwand auskommt, kann durch *syntaktische Dienstbeschreibungen* erreicht werden. Der Vergleicher schätzt beispielsweise über Wortähnlichkeiten ab, ob angebotene und benötigte Dienste zueinander passen. Ohne Rückfragen an einen menschlichen Benutzer stellen zu dürfen, begeht das System hierbei jedoch zwangsläufig semantische Fehler (siehe Abbildung 1.3b).
- Ein semantisch korrektes System, dass mit geringem Einigungsaufwand auskommt, kann durch *natürlichsprachliche Dienstbeschreibungen* erreicht werden. Die Korrektheit kann hierbei jedoch nur erreicht werden, wenn der Vergleich manuell von einem menschlichen Experten durchgeführt wird (siehe Abbildung 1.3c).

Die einzelnen Anforderungen können daher nicht einzeln betrachtet werden, sondern müssen im System gemeinsam erfüllt werden. Eine Lösung wird nur dann erfolgreich

sein, wenn sie einen speziell auf Dienstbeschreibungen zugeschnittenen Mittelweg beschreitet, d.h. eine ausgewogene Herangehensweise und Kompromisse an den Tag legt:

- *Ausdrucksmächtigkeit vs. Verarbeitbarkeit.* Einerseits muss die Sprache ausdrucksstark genug sein, um real existierende Dienste vollständig und eindeutig erfassen zu können, andererseits muss die Ausdruckskraft so beschränkt werden, dass es möglich ist, Beschreibungen automatisch und effizient zu vergleichen.

- *Flexibilität vs. Strukturiertheit.* Einerseits muss die Sprache flexibel genug sein, um die verschiedensten Dienste aller Domänen beschreiben zu können, andererseits muss die Sprache genügend Struktur vorgeben, damit sie auch im Fall unabhängig erstellter Beschreibungen vergleichbar bleibt.

1.5. Abgrenzung

Der Begriff des Dienstes ist sehr breit. Prinzipiell stellt jede hinter einer Schnittstelle verborgene öffentliche Funktionalität, die über das Netz aufgefunden, konfiguriert und verwendet werden kann, einen Dienst dar. Die Automatisierung der Dienstnutzung stellt daher ein gewaltiges Aufgabengebiet dar. Für diese Arbeit werden daher folgende Einschränkungen an die betrachteten Dienste und die daraus resultierende Dienstnutzung gemacht:

- Dienste haben eine Reihe nicht-funktionaler Aspekte, die orthogonal zu ihrer Funktionalität sind, wie etwa die Verfügbarkeit, die Ausfallsicherheit, ihre Ausführungsgeschwindigkeit etc. Der Hauptfokus der Arbeit liegt jedoch auf den *funktionalen Aspekten* eines Dienstes. Die zu entwickelnde Sprache soll jedoch so erweiterbar sein, dass nicht-funktionale Aspekte erfasst werden können.

- Die korrekte Verwendung eines Dienstes kann eine komplexe Interaktion zwischen Dienstnehmer und Dienstgeber erfordern, bei der über einen längeren Zeitraum synchron und asynchron Nachrichten ausgetauscht werden müssen. In der Arbeit werden hingegen nur solche Dienste betrachtet, die sich durch eine *einfache Choreographie* auszeichnen: Nach einer eventuellen Informationsbeschaffungsphase wird der Dienstgeber durch Empfang einer Nachricht konfiguriert und angestoßen, worüber er möglicherweise nach Beendigung des Dienstes mit einer Antwortnachricht informiert. Als erste Näherung ist diese Einschränkung unkritisch, da sich eine Vielzahl von Diensten auf dieses Muster abbilden lassen.

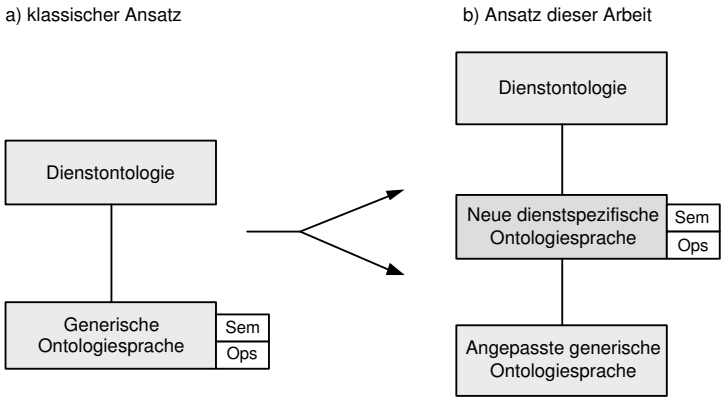


Abbildung 1.4.: Ansatz der Arbeit: Erweiterung der zugrunde liegenden Ontologiesprache um spezielle Konstrukte zur Erfassung der Besonderheiten von Diensten. Die relevante Semantik und die verwendeten Verarbeitungsoperatoren sind nicht mehr nur generisch, sondern finden sich auf Ebene der dienstspezifischen Elemente.

- Kann ein benötigter Dienst nicht alleine durch einen einzelnen Dienstgeber erbracht werden, so ist es möglich, mehrere Dienstgeber zur Laufzeit zu kombinieren. Diese *Dienstkomposition* wird in dieser Arbeit nur am Rande betrachtet, indem Abschnitt 11.1 mögliche weiterführende Arbeiten vorstellt. Der Ansatz sollte jedoch für eine solche Erweiterung geeignet sein.

- Der innere Aufbau eines Dienstes (d.h. seine Vorgehensweise zur Erbringung seiner Funktionalität) wird in dieser Arbeit nicht betrachtet. Entsprechend des Paradigmas dienstorientierter Architekturen werden Dienste als Blackbox gesehen, deren Funktionalität allein aus der Dienstbeschreibung erfassbar sein muss, ohne die konkrete algorithmische Vorgehensweise verstehen zu müssen.

1.6. Eigener Ansatz

Der Ansatz der Arbeit besteht darin, eine neue Dienstbeschreibungssprache zu entwickeln, die speziell auf die Besonderheiten der Beschreibung von Diensten ausgerichtet ist und so den Kompromiss zwischen den konkurrierenden Anforderungen besser umsetzen kann. Die Vorgehensweise dieser Arbeit unterscheidet sich daher von existierenden Ansätzen zur Dienstbeschreibung, die von einer gegebenen Ontologiebeschreibungssprache ausgehen und untersuchen, wie diese zur Beschreibung von Diensten eingesetzt werden kann. In der vorliegenden Arbeit hingegen wird anforderungsgetrieben vorgegangen und es werden Fragestellungen auf drei Ebenen beantwortet (siehe Abbildung 1.4):

- *Generische Ontologiesprache.* Welche Anforderungen müssen an eine Ontologie zur Beschreibung einer Domäne erfüllt sein, damit eine semantisch korrekte und effiziente Vermittlung von Diensten erfolgen kann, die in dieser Domäne operieren? Welche zusätzlichen Beschreibungselemente sind nötig, um die nötigen Informationen ausdrücken zu können? Wie kann sie helfen, zugleich einen moderaten Einigungsaufwand und eine Strukturierung von Dienstbeschreibungen zu erreichen?

- *Dienstspezifische Ontologiesprache.* Welche Besonderheiten hat ein Dienst im Vergleich zu anderen Entitäten der realen Welt? Welche sprachlichen Mittel sollten daher exklusiv zur Erfassung seiner Funktionalität zur Verfügung stehen? Welche Grundoperationen auf diesen neuen Elementen könnten einen effizienten und sich mit der Intuition deckenden Vergleich von Dienstbeschreibungen ermöglichen?

- *Dienstontologie.* Wie müssen Dienstbeschreibungen aus den generischen und dienstspezifischen Elementen aufgebaut werden, sodass flexible, aber zugleich strukturierte Beschreibungen entstehen?

Vorgeschlagen wird also eine neue Dienstbeschreibungs*sprache*, indem zwischen Ontologie und generischer Ontologiesprache neue sprachliche Beschreibungselemente eingeführt und diese mit einer formalen Semantik versehen werden. Die Semantik dieser Elemente ist dabei *dienstspezifisch*, d.h. sie bezieht sich explizit auf Akteure, Zeitpunkte und Vorgänge, die während des Prozesses der Dienstnutzung beteiligt sind. Hierdurch erhalten konkrete Dienstbeschreibungen eine eindeutige Aussage, was einen automatischen, semantisch korrekten und sich mit der Intuition deckenden Vergleich ermöglicht.

Konkret lassen sich diese neuen Sprachelemente der dienstspezifischen Ontologiesprache in vier Gruppen einteilen, die sich aus grundlegenden Charakteristika von Diensten herleiten:

- Dienste sind nicht nur Teil der Welt, sondern wirken auch verändernd in ihr. Daher werden *operationale Elemente* benötigt.

- Dienste sind häufig in der Lage, verschiedene ähnliche Effekte zu erwirken. In ihrer Beschreibung sollen diese zusammengefasst werden. Daher werden *aggregierende Elemente* benötigt.

- Dienste können vor ihrer Ausführung konfiguriert werden, meist indem sich Anfrager und Anbieter auf Entitäten einigen, auf denen die Dienstoperationen durchgeführt werden sollen. Daher werden *selektierende Elemente* benötigt.

- Dienste können Effekte erwirken, die von menschlichen Benutzern unterschiedlich präferiert werden. Um dies ausdrücken zu können, werden *bewertende Elemente* benötigt.

Der Ansatz umfasst auch neue Verarbeitungsoperatoren, die speziell auf die Unterstützung der wesentlichen Funktionen eines dienstorientierten Systems (wie den Vergleich, die Auswahl, die Konfiguration etc.) ausgelegt sind. Diese sind auf der Ebene der dienstspezifischen Ontologiesprache angesiedelt und gehen über die typischen generischen Schlussfolgerungsoperationen hinaus. Aufgrund der speziellen Ausrichtung können diese vorgeschlagenen Operatoren effizient implementiert werden und zur Berechnung intuitiver Vergleichsergebnisse verwendet werden.

1.7. Gliederung der Arbeit

Die Arbeit ist wie folgt aufgebaut: Im ersten Teil der Arbeit werden zunächst in Kapitel 2 die zum Verständnis nötigen Grundlagen zu Webdiensten und zum Semantischen Web eingeführt. Kapitel 3 analysiert dann den Stand der Forschung, indem Ansätze der Literatur mit ähnlichen Zielen vorgestellt und untersucht werden. Der zweite Teil präsentiert den eigenen Ansatz. Einen Überblick über die Konzepte gibt zunächst Kapitel 4. Die Details der Durchführung werden entlang der drei Ebenen vorgestellt: Kapitel 5 zeigt die Arbeiten auf Seiten der generischen Ontologiesprache, Kapitel 6 führt die neuen Sprachelemente der dienstspezifischen Ontologiesprache ein, während Kapitel 7 die Arbeiten auf Ontologieebene vorstellt. Die formale Semantik aller Sprachelemente findet sich in Kapitel 8. Kapitel 9 schließt den Teil mit der Präsentation des Vergleichers für die entwickelte Dienstbeschreibungssprache ab. Im dritten Teil wird die Arbeit untersucht und Ideen zur Weiterentwicklung vorgestellt. Kapitel 10 führt zunächst eine Evaluation der Sprache durch, Kapitel 11 zeigt, welche weiteren Arbeiten ausgehend von der entwickelten Sprache möglich sind. Abschließend fasst Kapitel 12 den Ansatz zusammen und gibt einen Ausblick auf mögliche Erweiterungen und Verbesserungen.

In der Arbeit werden folgende Schriften verwendet:

- *Kursive Schrift* zur Einführung neuer Begriffe bzw. zur Hervorhebung gliedernder Begriffe in Aufzählungen.
- **Fettschrift** zur Hervorhebung wichtiger Sachverhalte oder zur besseren Gliederung von Textteilen.
- Serifenlose Schrift zur Beschreibung von Klassen-, Attribut- und Instanznamen der Ontologien sowie zur Kennzeichnung von Variablenkategorien.

- `Schreibmaschinenschrift` zur Beschreibung von Quellcode, Ontologienamen, Dateinamen und primitiven Typen.
- Kapitälchen zur Kennzeichnung von Personennamen.

2. Grundlagen

Dieses Kapitel gibt eine Einführung in die Themen, die für das Verständnis der Arbeit benötigt werden. Wie in der Einleitung bereits angeklungen, wird dazu auf das Themengebiet der Webdienste (Abschnitt 2.1) sowie auf das Semantische Web (Abschnitt 2.2) eingegangen. Die grundlegenden Begriffe zu Semantischen Webdiensten finden sich zu Beginn der Untersuchung des Stands der Forschung im nächsten Kapitel.

2.1. Webdienste

Webdienste (engl. Web Services) stellen eine Anwendung des Internets dar. Ziel hierbei ist die entfernte Nutzung von Funktionalität in Form von Diensten über das Netz. Das Internet erscheint hier also nicht wie im World Wide Web als verteilte Sammlung von Dokumenten, sondern als verteilte Sammlung nutzbarer Dienste. Als Webdienst gilt jede Funktionalität, die über das Internet veröffentlicht, gefunden und ausgeführt werden kann. Basistechnologie für Webdienste im Internet ist die *Extensible Markup Language* (XML), die als Verallgemeinerung von HTML angesehen werden kann.[1] Sie dient einerseits als sprachliche Grundlage, um die Funktionalität eines Webdienstes zu beschreiben, andererseits durch die in XML Schema definierten atomaren Datentypen als gemeinsame Basis zur Repräsentation der Nachrichten, die zwischen Dienstnehmer und Dienstgeber ausgetauscht werden. Eine Definition für einen Webdienst sieht wie folgt aus:

Definition 2.1.1 [Webdienst]
Ein Webdienst *ist ein Softwaresystem, welches durch eine URI identifizierbar ist und dessen öffentliche Schnittstellen mittels XML definiert und beschrieben sind. Andere Systeme können diesen Webdienst anhand seiner XML-Beschreibung auffinden und verwenden, indem XML-basierte Nachrichten wie in der Beschreibung angegeben über das Internet ausgetauscht werden [57].*

Der Ablauf einer Dienstnutzung entspricht daher dem in Abbildung 2.1 dargestellten *Dienstdreieck*:

[1] Tatsächlich leiten sich beide Sprachen von der Standard Generalized Markup Language (SGML) ab. HTML ist dabei eine Anwendung von SGML, XML eine Untermenge von SGML.

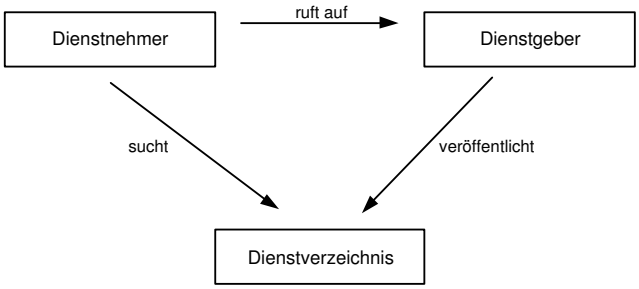


Abbildung 2.1.: Vereinfachte Darstellung einer Dienstnutzung im „Dienstdreieck“.

1. Der *Dienstgeber* beschreibt die Funktionalität seines Softwaresystems, welche er öffentlich zur Verfügung stellen möchte, mittels einer XML-basierten Dienstbeschreibung. Hierzu verwendet er die *Web Service Description Language* (WSDL, siehe Abschnitt 2.1.1).

2. Der Dienstgeber registriert diese Dienstbeschreibung in einem oder mehreren öffentlichen *Dienstverzeichnissen*. Hierzu verwendet er *Universal Description, Discovery, and Integration* (UDDI, siehe Abschnitt 2.1.3), ein Protokoll zum Veröffentlichen und Wiederfinden von Dienstbeschreibungen.

3. Möchte ein Entwickler einer Software eine Funktionalität über einen externen Dienst nutzen, wendet er sich an ein oder mehrere Dienstverzeichnisse und durchsucht diese nach einem geeigneten Dienst.

4. Der Entwickler verbindet seine Software mit dem gefundenen Dienstgeber als Teil des Entwicklungsprozesses (*statische Dienstbindung*).

5. Zur Laufzeit kann die Software auf die Funktionalität des eingebundenen Dienstgebers zurückgreifen. Hierzu verwendet sie das *Simple Object Access Protocol* (SOAP, siehe Abschnitt 2.1.2), das die Parameter und Rückgabewerte in XML-basierte Nachrichten kodiert.

2.1.1. Web Service Description Language (WSDL)

Die *Web Service Description Language* (WSDL) [27] ist die offizielle Beschreibungssprache für Webdienste. Ursprünglich von Microsoft und IBM entwickelt, wird sie heute vom W3C standardisiert und befindet sich zurzeit in Version 2.0. WSDL basiert auf XML, insbesondere werden zur Beschreibung von Nachrichten die atomaren Typen aus XML Schema herangezogen.

WSDL stellt einen Dienst als Sammlung von Kommunikationsendpunkten dar, über welche Operationen mittels Austausch von Nachrichten durchgeführt werden können.

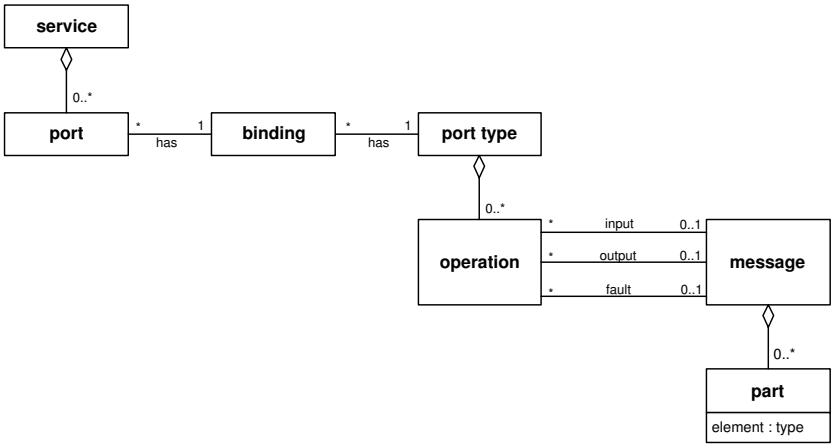


Abbildung 2.2.: Grundkonzepte von WSDL als UML-Diagramm.

Die Beschreibung teilt sich in die Definitions des Vokabulars, die Definition der Nachrichten sowie die Erfassung der Interaktionen. Im Vokabular können mit Hilfe des Typsystems aus XML Schema komplexe Typen definiert werden. Hiermit kann der Aufbau von auszutauschenden Nachrichten festgelegt werden. Eine Nachricht ist somit eine Komposition von Elementen des vordefinierten Vokabulars, d.h. atomaren oder komplexen Typen. Die Interaktion wird durch Operationen definiert. Typischerweise kann eine Operation eine Nachricht empfangen, diese verarbeiten und eine Ergebnisnachricht liefern. Aus diesem Grund wird WSDL als *nachrichtenbasierte Beschreibung* bezeichnet [85].

Die Beschreibungselemente von WSDL sind in Abbildung 2.2 als UML-Klassendiagramm dargestellt. Ein Dienst (engl. service) ist eine Sammlung von Kommunikationsendpunkten (engl. ports). Diese Kommunikationsendpunkte müssen über eine Bindung (engl. binding) an einen PortTyp gebunden werden. Ein PortTyp ist dabei eine Sammlung abstrakter Operationen (engl. operations), die vom zugehörigen Kommunikationsendpunkt angeboten werden. Eine Operation kann eingehende (engl. input), ausgehende (engl. output) und Fehlernachrichten (engl. fault messages) besitzen. WSDL unterscheidet daher:

- *Einweg-Operationen* (engl. one-way operations), d.h. Operationen, bei denen der zugehörige Endpunkt nur eine Nachricht empfängt.
- *Anfrage-Antwort-Operationen* (engl. request-response operations), d.h. Operationen, bei denen der zugehörige Endpunkt eine Nachricht empfängt, verarbeitet und eine Antwortnachricht aussendet.
- *Bewerbung-Antwort-Operationen* (engl. solicit-response operations), d.h. Operationen, bei denen der zugehörige Endpunkt zunächst aktiv eine Nachricht versendet, um anschließend eine eingehende Nachricht empfangen zu können.

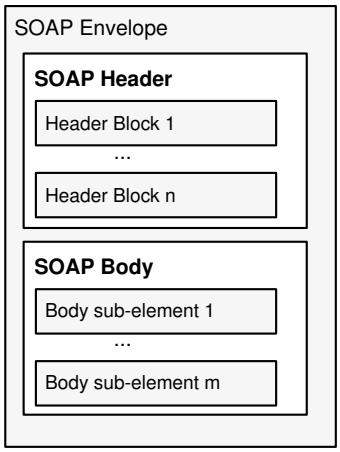


Abbildung 2.3.: Aufbau einer SOAP-Nachricht.

- *Ankündigungsoperationen* (engl. notification), d.h. Operationen, bei denen der zugehörige Endpunkt nur eine Nachricht aussendet.

Neben dieser abstrakten Beschreibung des Dienstes, wird in WSDL der Zusammenhang zur konkret zu verwendenden Kommunikation mit dem Dienst getrennt beschrieben. In der so genannten Bindung (engl. binding) werden daher die abstrakt definierten Nachrichten und Operationen mit der konkreten Nachrichtensyntax und den Details zum Protokollablauf verknüpft. Das wichtigste Protokoll, welches den entfernten Dienstaufruf über das Internet unterstützt, ist das ebenfalls XML-basierte SOAP (siehe Abschnitt 2.1.2). Daneben sind auch Bindungen zu einfacherem HTTP GET/POST möglich. Die Bindungsinformation wird direkt an den jeweiligen Stellen (port, operation, message) in die abstrakte Dienstbeschreibung eingetragen.

2.1.2. Simple Object Access Protocol (SOAP)

SOAP [104] (eigentlich nur bis Version 1.1 als Akronym für *Simple Object Access Protocol*) ist ein leichtgewichtiges Protokoll und eine Kodierungsvorschrift basierend auf XML, um Informationen in verteilten Umgebungen zwischen beliebigen Anwendungen wie etwa Webdiensten auszutauschen. Ursprünglich von Microsoft und Userland Software konzipiert, wurde SOAP mehrfach überarbeitet und liegt zurzeit in Version 1.2 vor, die vom W3C standardisiert werden soll.

Durch SOAP werden zwei Konzepte spezifiziert:

- *SOAP Envelope*, durch welchen der Aufbau einer SOAP-Nachricht festgelegt wird (Syntax) und wie eine solche beim Eintreffen in einem SOAP-Knoten verarbeitet werden muss (Semantik). Der Aufbau ist in Abbildung 2.3 dargestellt

und teilt sich in einen optionalen SOAP Header, welcher Metadaten über den Aufruf und Verarbeitungsvorschriften für den Zielknoten enthält, sowie einen SOAP Body, der die eigentlichen zu übermittelnden Daten enthält.

- *Abbildungsvorschriften*, die festlegen, wie applikationsspezifische Typen auf SOAP-Typen abgebildet werden.

SOAP liegt im Wesentlichen das Paradigma eines zustandslosen, unidirektionalen Nachrichtenaustauschs zugrunde. Allerdings können Anwendungen auch komplexere Interaktionsmuster definieren, indem mehrere unidirektionale Nachrichtenübermittlungen kombiniert werden. SOAP bleibt jedoch nach wie vor unabhängig von den zu übermittelnden Daten und dem zugrunde liegenden Transportmechanismus im Netz.

Das wichtigste Interaktionsmuster, das auch im Rahmen von Webdiensten Verwendung findet, ist der entfernte Funktionsaufruf (engl. remote procedure call, RPC). Hier werden zwei unidirektionale Nachrichtenübertragungen so verbunden, dass die erste vom Dienstnehmer zum Dienstgeber führt und Informationen über die aufzurufende Funktion sowie die nötigen Parameter enthält, während die zweite die entstandenen Ergebnisse vom Dienstgeber zum Dienstnehmer transportiert. Aufgrund der hohen Bedeutung von *SOAP RPC* sind die Details zu diesem Interaktionsmuster bereits in der Spezifikation von SOAP verankert. Es bildet die technische Grundlage für den Aufruf von Webdiensten über das Netz.

Der Austausch von SOAP-Nachrichten kann im Internet über verschiedenartige unterliegende Protokolle erfolgen, welche typischerweise auf der Applikationsebene zu finden sind. Die häufigsten Vertreter sind HTTP, HTTPS und SMTP. Das jeweilige Interaktionsmuster passt sich dabei dem verwendeten Protokoll in natürlicher Weise an.

2.1.3. Universal Description, Discovery, and Integration (UDDI)

Universal Description, Discovery, and Integration (UDDI[2]) ist neben WSDL und SOAP der dritte Bestandteil zur technischen Realisierung von Webdiensten. UDDI stellt ein webbasiertes, verteiltes Verzeichnis dar, in dem Teilnehmer ihre angebotenen Dienste registrieren und nach vorhandenen Diensten suchen können. Aus diesem Grund spricht man von UDDI auch als *Gelbe-Seiten-Dienst*. Tatsächlich erfolgt der Zugriff auf ein UDDI-Verzeichnis selbst wieder über eine Reihe von Webdiensten.

Jeder zu veröffentlichende Dienst wird in UDDI unter einer *UDDI Business Registration* abgelegt, einer Datenstruktur, welche auf XML basiert und in folgende Abschnitte unterteilt ist:

[2] `http://www.uddi.org`

- *BusinessEntity*, welche Informationen über den Anbieter des Dienstes enthält. Wird ein über WSDL beschriebener Dienst in einem UDDI-Verzeichnis veröffentlicht, so verweist die BusinessEntity auf das WSDL-Dokument.
- *BusinessService*, welche Informationen über die Funktionalität des Dienstes enthält. Wichtigstes Merkmal ist eine Kategorisierung des Dienstes durch Zuordnung zu einem oder mehreren Geschäftsfeldern. Die Taxonomie kann dabei von UDDI oder weiteren Teilnehmern stammen.
- *BindingTemplates*, welche technische Details der gegebenen Dienstimplementierung enthalten und beschreiben, wie auf den Dienst zugegriffen werden muss.
- *PublisherAssertion*, welche weitere Metadaten zum Dienst enthalten, etwa Beziehungen zu anderen Diensten oder Angaben zu Sicherheitsaspekten.

Folgende Funktionalität steht für UDDI-Verzeichnisse in Form von Webdiensten zur Verfügung:

- Veröffentlichung eines Dienstes durch Angabe einer Business Registration.
- Hinterlegung von stehenden Anfragen.
- Suche nach Diensten oder Informationen über Dienste in den hinterlegten Metadaten.

Weitere Funktionalität wie die Replikationsverwaltung bei mehreren Verzeichnissen, Schlüsselgenerierung, Autorisierung usw. erfolgen für den Endanwender transparent und können über eine spezielle Administrationsschnittstelle konfiguriert werden.

Die Spezifikation macht UDDI besonders für ein halbautomatisches Auffinden von Diensten im Geschäftsumfeld interessant. Eine vollautomatische Auswahl ist nicht angestrebt.

2.1.4. Werkzeuge für den Umgang mit Webdiensten

Der Erfolg von Webdiensten beruht neben der Einfachheit auch auf der Vielzahl von Werkzeugen, die mittlerweile existieren. Auch die nahtlose Einbettung von Webdiensten in Programmiersprachen zur Erstellung von dienstorientierten Anwendungen wird stark forciert. Insbesondere Microsofts *.NET Framework* und Suns J2EE-Umgebung zusammen mit dem *Java Web Service Developer Pack* (Java WSDP) vereinfachen die Verwendung von Funktionalität über Webdienste erheblich. Es ist abzusehen, dass in Zukunft alle Technologien zur verteilten Anwendungsentwicklung (wie DCOM,

RPC, RMI) über ein vereinheitlichtes Modell auf Basis von Webdiensten abgewickelt werden. Microsoft will etwa ein entsprechendes Rahmenwerk (*Indigo*) in die nächste Windows-Version *Windows Vista* integrieren [20]. Auch IBM versucht sich im wachsenden Markt um Webdienste mit den Produkten *WebSphere Application Server* und *WebSphere Studio* zu etablieren.[3]

Neben Produkten kommerzieller Anbieter existiert auch eine Reihe von Werkzeugen zum Umgang mit Webdiensten von nicht-profitorientierten Gruppen, allen voran das *Apache Web Services Project*[4]. Wichtigstes Produkt ist *Axis*, bestehend aus einem SOAP-Server und einer Reihe unterstützender Techniken, um Funktionalität als Webdienst anbieten zu können.

2.1.5. Erweiterungen zu WSDL, SOAP und UDDI

Die drei Technologien WSDL, SOAP und UDDI bieten die Grundlagen zur Nutzung von Webdiensten. Daneben entwickeln eine Reihe von Herstellern darunter Microsoft, IBM und SAP Erweiterungen, um diese Nutzung effektiver, sicherer und zuverlässiger zu machen. Zusammengefasst werden sie unter dem Namen *WS-*-Spezifikationen*. Diese Spezifikationen unterliegen jedoch noch starken Veränderungen und werden daher hier nur kurz vorgestellt:

- *WS-Addressing*. Erweiterung, um Referenzen zum Endpunkt eines Webdienstes eindeutig ausdrücken zu können.
- *WS-Policy*. Erweiterung, um Anforderungen und Zusicherungen von Webdiensten ausdrücken zu können. Die generischen Sprachkonstrukte konzentrieren sich dabei auf nicht-funktionale Aspekte wie Nachrichtengrößen, Verschlüsselung, Dienstgüte usw. Untergeordnete Spezifikationen regeln, wie solche Zusicherungen in WSDL-Dokumente und SOAP-Nachrichten eingebunden werden.
- *WS-Resource Framework* (WSRF). Erweiterung, die nötig ist, um Ressourcen (wie Speicherplatz, Rechenkapazität etc.) auf einem Dienstgeber beschreiben zu können. Hierzu ist insbesondere die Beschreibung der Zustände auf dem Dienstgeber nötig. WSRF ist aus Ideen des Grid Computings entstanden, insbesondere der *Open Grid Services Infrastructure* (siehe auch Abschnitt 3.6.3).
- *WS-Security* (WSS). Erweiterung, um die Nutzung von Webdiensten sicherer zu machen. WSS bietet unter anderem Möglichkeiten, die Integrität und Vertraulichkeit von ausgetauschten Nachrichten zu sichern sowie die Identität der teilnehmenden Parteien zu garantieren. Die nötigen Technologien für digitale

[3] `http://www.ibm.com/software/websphere`
[4] `http://ws.apache.org`

Signaturen und Verschlüsselung werden dabei aus den W3C-Standards *XML Signature* und *XML Encryption* entliehen.

- *WS-Coordination.* Generische Erweiterung, um Dienstbeschreibungen mit Protokollen anzureichern, welche die Aktionen mehrerer beteiligter Dienste koordinieren, etwa um zu einem global konsistenten Zustand zu gelangen. Die wichtigsten Implementierungen dieser Erweiterung sind *WS-AtomicTransaction* zur Definition kurz laufender Transaktionen mit garantierter Atomizität durch Sperren sowie *WS-BusinessActivity* zur Definition lang laufender Transaktionen mit eingeschränkter Atomizität mittels Kompensationsfunktionen.

Eine weitere wichtige Erweiterung, die nicht direkt zu den WS-*-Spezifikationen zählt, ist die *Business Process Execution Language for Web Services* (BPEL4WS[5]). Sie stellt eine Sprache zur Spezifikation der *Orchestrierung* dar, also des Zusammenspiels zwischen eigentlichem Dienstgeber und seinen Sub-Dienstgebern. BPEL4WS ermöglicht die Definition eines Prozesses, in den mehrere Webdienste integriert werden können, um so einen neuen Dienst zu erhalten. Angestrebt wird eine statische Komposition, d.h. eine dynamische Umkonfigurierung zur Laufzeit ist nicht angedacht.

2.1.6. Zusammenfassung

Webdienste stellen Funktionen dar, die öffentlich über das Internet genutzt werden können, indem Nachrichten ausgetauscht werden. Technische Grundlage bilden drei XML-basierte Sprachen: WSDL zur nachrichtenorientierten Beschreibung der Dienstfunktionalität, UDDI als Verzeichnis zum Ablegen und Auffinden von Dienstbeschreibungen und SOAP zum Nachrichtenaustausch zwischen Dienstnehmer und Dienstgeber. Angestrebt ist keine vollständige Automatisierung zur Laufzeit, sondern eine manuelle Suche und Auswahl aus dem Verzeichnis sowie eine statische Dienstbindung. Webdienste sind aufgrund ihrer Einfachheit und der Vielzahl der existierenden Werkzeuge weit verbreitet. Zur Erhöhung des Nützlichkeit sind eine Reihe von Erweiterungen in Arbeit.

2.2. Das Semantische Web

2.2.1. Idee des Semantischen Webs

Das *Semantische Web* (engl. Semantic Web) ist eine Erweiterung des WWW. Ziel ist es, die Ressourcen des Internets auch durch Rechner automatisch und sinnvoll verarbeitbar zu machen. Die Bestrebungen gehen auf Tim Berners-Lee zurück, der

[5] `http://www.ibm.com/developerworks/library/specification/ws-bpel`

erste Ideen hierzu im Jahr 2000 äußerte. In der Tat wächst das World Wide Web und die darin enthaltene Information mit einem enormen Tempo. Die Inhalte sind jedoch in Dokumente verpackt und nicht auf Weiterverarbeitbarkeit, sondern auf Präsentation ausgelegt und somit für die beteiligten Rechner weitestgehend opak. Der Rechner eines Nutzers dient daher oft nur als reines Anzeigegerät, welches keine komplexeren, inhaltlichen Operatoren auf den Dokumenten auszuführen vermag. Suche und Extraktion von Informationen verlangt vom Benutzer daher ein steigendes Maß an Handarbeit.

Der Ansatz zur Erreichung der Ziele des Semantic Web beruht auf einer Erweiterung um Zusatzinformationen. Die Dokumente im Netz (oder auch Teile davon) sollen durch Metadaten so beschrieben werden, dass ein Rechner in der Lage ist, diese mit inhaltlichen Operatoren auswerten zu können. Dies wird auch in einer Definition von Berners-Lee deutlich:

Definition 2.2.1 [Semantisches Web]
Das Semantische Web *ist eine Erweiterung des aktuellen Webs, in welchem den Informationen eine wohl definierte Bedeutung gegeben ist, um die Kooperation zwischen Computern und Menschen zu verbessern [16].*

Als erster einfacher Ansatz zu einer Annotation von Dokumenten im Web können *Meta Tags* in HTML-Dateien angesehen werden. Im Kopf eines solchen Dokuments kann beispielsweise durch die Angabe von

```
<META name="Creator" content="Michael Klein">
```

der Ersteller der Datei bekannt gegeben werden. HTML macht jedoch keine Angaben darüber, welche Schlüssel-Wert-Paare für ein Dokument bestimmt werden sollten. Auch die Bedeutung der Angaben ist offen und bleibt der Interpretation des Benutzers überlassen. Es besteht jedoch die Möglichkeit, die gültigen Schlüssel in einem *Profile* abzulegen. Das wichtigste und bekannteste Profil ist *Dublin Core*[6] zur einheitlichen Beschreibung von elektronischen Dokumenten. Hierin wird ein Satz von 15 Schlüsseln vorgeschlagen (wie `Title`, `Creator`, `Language`, `Rights` etc.) und deren Semantik verbal hinterlegt. Eine solche Beschreibung bietet zwar bereits einen Mehrwert, da sie von Suchmaschinen ausgewertet werden kann, hat jedoch den Nachteil, dass die Zusatzinformationen flach sind, d.h. in der Regel nur ein Literal in Form eines Strings, einer Zahl oder eines Datums darstellen.

[6] `http://dublincore.org`

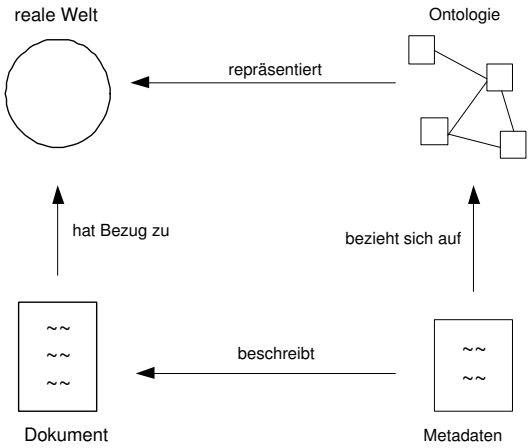


Abbildung 2.4.: Zusammenhang zwischen Metadaten und Ontologie.

2.2.2. Ontologien

Grundlage für eine allgemein verständliche Beschreibung mit Metadaten ist ein gemeinsames Vokabular. Interessant wird ein solches Vokabular jedoch erst dann, wenn es einen Bezug zu Objekten der realen Welt aufweist, welche selbst weitere Eigenschaften besitzen können. Beispiele hierfür sind:

- Der Autor eines Dokuments ist eine Person, die an einer Institution arbeitet und bestimmte Forschungsinteressen hat.
- Das Themengebiet einer Seite ist verwandt zu anderen Themengebieten.
- Die Rechte an einem Dokumente gehören einer Organisation, deren Sitz in den USA ist.

Beschreibungen dieser Art erlauben auch tiefer gehende Anfragen, wie etwa nach Dokumenten, die von relationalen Datenbanken oder verwandten Themen handeln, oder Dokumenten, die von einem Autor des IPD verfasst wurden.

Nötig hierzu ist eine Abbildung der realen Welt in eine formale, computerlesbare Form, eine so genannte *Ontologie*[7]. Die Metadaten können dann Referenzen auf Beschreibungselemente der Ontologie enthalten und so den Zusammenhang des Dokuments mit der realen Welt darstellen (siehe Abbildung 2.4).

Eine Ontologie dient somit hauptsächlich der Explizitmachung von Begriffen in einem Anwendungsgebiet (der *domain of interest*) durch eine Anwendergruppe (der

[7]Der Begriff ist aus der Philosophie entliehen, wo er die philosophische Disziplin beschreibt, die sich mit dem Seienden beschäftigt. Der Ausdruck ist von dem griechischen Partizip *on* (= Seiendes) abgeleitet.

community). Ein *Begriff* ist dabei die mehr oder weniger geteilte Vorstellung von einem konkreten oder abstrakten Gegenstand der realen Welt. Als *Konzept* bezeichnet man die explizite und einheitliche Benennung eines Begriffs in der Anwendergruppe.

Die wichtigste und meistzitierte Definition geht auf Tom Gruber aus dem Jahr 1993 zurück:

Definition 2.2.2 [Ontologie]
Eine Ontologie ist eine explizite und formale Spezifikation einer geteilten Konzeptualisierung [52].

Darin bedeutet

- *Konzeptualisierung* ein abstraktes Modell von Begriffen, die von der Anwendergruppe als wichtig für den Anwendungszweck herausgestellt wurden;
- *explizit*, dass die Begriffe nicht nur gedanklich vorliegen, sondern konkret als Konzepte notiert werden;
- *formal*, dass diese Konzepte maschinenlesbar sein sollen;
- *geteilt*, dass durch Einigung in der Anwendergruppe ein gemeinsames Verständnis der Konzepte erzielt wird.

Eine Ontologie überbrückt somit die semantische Lücke zwischen realer Welt, in welcher der Mensch „operiert“, und dem Informationssystem, in welchem Rechner operieren können. Hauptaufgabe ist daher die Einigung auf Konzepte und deren Bedeutung in der Anwendergruppe, um sich im Folgenden durch Bezug auf diese Konzepte semantisch eindeutig austauschen zu können:

> *Ontologies are models that represent an abstraction of a domain in a formal way, such that several parties are able to agree on the abstraction and reuse the model in their own application.* [63]

Um sinnvoll verwendet werden zu können, sollten Ontologien analog zur Erstellung von Software auf ingenieurmäßigem Weg erstellt und bestimmte Regeln beachtet werden. So sollten Ontologien unter anderem modular, intern kohärent und erweiterbar sein sowie ihre Konzepte um natürliche Kategorien anordnen (vgl. [141]). Zu beachten ist, dass eine Ontologie nicht an einen bestimmten Formalismus gebunden ist. Im Gegenteil: Es existiert eine Reihe konkurrierender *Ontologiesprachen*, welche versuchen, diesem Ziel in einem bestimmten Anwendungsfall möglichst gerecht zu werden. Sie unterscheiden sich im Wesentlichen darin, in welcher Art und Weise Konzepte

durch sie geordnet werden können, d.h. welchen *Ordnungsmechanismus* in Form von Modellierungsprimitiven sie verwenden. Einfache Ontologiesprachen erlauben nur die Auflistung von Konzepten, komplexere Sprachen ermöglichen auch das Einführen von Beziehungen zwischen Konzepten, das Definieren von Attributen usw.

Häufig können nicht alle Begriffe und deren Beziehungen explizit als Konzepte in der Ontologie festgehalten werden. Daher stellen Sprachen vordefinierte *ausgezeichnete Konstrukte* zur Verfügung, mit denen Wissen über die reale Welt zusammengefasst werden kann. Zum Beispiel könnte angegeben werden, dass die Konzepte Mann und Frau disjunkt sind oder dass aus der Beziehung istElternteil(x, y) die Beziehung istKind(y, x) abgeleitet werden kann. Dadurch enthält die Ontologie neben dem explizit ausgedrückten Wissen auch implizites Wissen. Zur eindeutigen Ableitung dieses impliziten Wissens müssen die vordefinierten Konstrukte eine wohldefinierte Semantik besitzen. Hierzu existieren prinzipiell zwei Vorgehensweisen:

- Direkt durch Angabe einer *modelltheoretischen Semantik*. Hierbei werden die Konzepte der Ontologie durch eine gedankliche Abbildung (die *Interpretation*) auf Begriffe der realen Welt definiert. Für ausgezeichnete Konstrukte ist angegeben, welche zusätzlichen Bedingungen sie an eine solche Interpretation stellen.
- Indirekt durch Angabe einer *axiomatischen Semantik*, indem die Konstrukte der Sprache formal auf eine andere Sprache abgebildet werden, für die bereits eine Semantik existiert.

Die Explizitmachung impliziten Wissens innerhalb einer Ontologie nennt man *logisches Schließen* (engl. reasoning). Typischerweise stehen für verschiedene Ontologietypen unterschiedliche Schlussfolgerungsoperationen zur Verfügung, etwa der Test, ob eine gegebene Instanz zu einem Konzept gehört, ein Konzept widersprüchlich ist, d.h. keine Instanzen enthalten kann, oder ein Konzept ein Teil eines anderen Konzeptes ist.

Zusammenfassend hat also eine Ontologiesprache folgende Bestandteile:

- Eine wohldefinierte, computerverarbeitbare *Syntax*.
- Einen *Ordnungsmechanismus*, der es den Teilnehmern der Anwendergruppe erlaubt, selbst Konzepte und gegebenenfalls deren Beziehungen und Eigenschaften zu definieren. Im Falle einer modelltheoretischen Semantik haben diese eine *Interpretation*, d.h. eine Entsprechung in der realen Welt, auf welche sich die Anwendergruppe verständigt hat. Die Interpretation ist nicht streng formalisiert, sondern bestenfalls menschenverständlich dokumentiert.

- Optional eine Reihe *vordefinierter Konstrukte*, mit denen die Teilnehmer der Anwendergruppe weiteres Wissen implizit definieren können. Für diese ist eine formale Semantik hinterlegt.
- Eine Reihe von *Schlussfolgerungsoperationen*, die es erlauben, Anfragen an das explizit und implizit vorhandene Wissen der Ontologie zu stellen.

Spezielle, einfache Arten von Ontologien sind:

- *Kontrolliertes Vokabular.* Ontologie ohne vordefinierte Konstrukte oder Schlussfolgerungsoperationen. Der Hauptfokus liegt hierbei auf der Einigung auf Konzepte innerhalb der Anwendergruppe. Alle Konzepte sind explizit vorhanden.
- *Thesaurus.* Ein kontrolliertes Vokabular mit zusätzlichem vordefinierten Konstrukt *istÄhnlichZu*.
- *Taxonomie.* Ein kontrolliertes Vokabular mit zusätzlichem vordefinierten Konstrukt *istUntertypVon*.

Im Folgenden werden die wichtigsten Ontologiesprachen des Semantischen Webs vorgestellt. Im Einzelnen sind dies F-Logic, RDF und RDFS sowie OWL. Einen umfangreichen Überblick über das Themengebiet der Ontologien liefert [137].

2.2.3. F-Logic

F-Logic (eigentlich Frame-Logic) [72] kann vereinfacht als eine formale, objektorientierte Modellierungssprache aufgefasst werden. Die **Ordnungsmechanismen** sind daher die aus der Welt der Objektorientierung bekannten Elemente: Klassen (in F-Logic so genannte *frames*), Attribute (in F-Logic so genannte *slots*), Methoden (in in F-Logic so genannte *signatures*), komplexe Objekte mit Klassenzugehörigkeit und Identität. Die wichtigsten **vordefinierten Konstrukte** sind Vererbung und die Möglichkeit, komplexe Regeln zu definieren.

Abbildung 2.5 zeigt eine Beispielontologie für Verwandtschaftsverhältnisse in F-Logic. Definiert sind drei Klassen für Frau, Mann und Person. Durch den zweifachen Doppelpunkt wird eine Vererbungsbeziehung ausgedrückt: Frauen und Männer sind also Personen. In eckigen Klammen werden Attribute definiert, wie hier, dass jede Person genau einen Vater, genau eine Mutter (einfacher Pfeil) und 0 oder mehrere Töchter und Söhne hat (doppelter Pfeil). Anschließend sind vier Regeln definiert, welche von rechts nach links zu lesen sind. Die erste Regel besagt beispielsweise, dass, falls Y ein Mann ist, dessen Vater X ist (rechte Seite), dann Y der Sohn von X ist (linke Seite). Abschließend sind drei Objekte definiert: Abraham vom Typ Mann, Sarah vom Typ

```
woman::person.
man::person.
person[father=>man].
person[mother=>woman].
person[daughter=>>woman].
person[son=>>man].

FORALL X,Y  X[son->>Y] <- Y:man[father->X].
FORALL X,Y  X[son->>Y] <- Y:man[mother->X].
FORALL X,Y  X[daughter->>Y] <- Y:woman[father->X].
FORALL X,Y  X[daughter->>Y] <- Y:woman[mother->X].

abraham:man.
sarah:woman.
isaac:man[father:abraham; mother:sarah].
```

Abbildung 2.5.: Beispiel-Ontologie in F-Logic, entnommen aus [4].

Frau und Isaac vom Typ Mann, dessen Attribute Vater und Mutter gefüllt sind. In F-Logic sind eine Reihe komplexerer Konstrukte möglich, die nicht vorgestellt werden sollen.

Als **Schlussfolgerungsoperation** stellt F-Logic Anfragen zur Verfügung. Anfragen entsprechen Regeln mit leerer linker Seite, welche auch Variablen enthalten können. Ein Beispiel könnte sein:

```
FORALL X,Y <- X:woman[son->>Y[father:abraham]]
```

Gesucht sind demnach alle Frauen X, die einen Sohn Y haben, dessen Vater Abraham ist. Als Ausgabe liefert eine geeignete Implementierung Belegungen für die Variablen X und Y, die diese Bedingung erfüllen.

Die **Semantik** von F-Logic kann axiomatisch durch Abbildung der Konstrukte in Prädikatenlogik erster Ordnung erfasst werden. Dabei werden alle Konstrukte durch Prädikate repräsentiert, etwa `C[A=>R]` durch $atttype(C, A, R)$ oder `A::B` durch $sub(A, B)$. Die eigentliche Semantik wird dann durch Abschlussregeln ausgedrückt:

- Die Vererbung ist transitiv:
  $\forall X, Y, Z : sub(X, Y) \wedge sub(Y, Z) \rightarrow sub(X, Z)$

- Attribute werden an Unterklassen vererbt:
  $\forall C_1, C_2, A, T : sub(C_1, C_2) \wedge atttype(C_2, A, T) \rightarrow atttype(C_1, A, T)$
- usw.

Die **Syntax** von F-Logic ähnelt objektorientierten Programmiersprachen und ist daher intuitiv zu verwenden.

2.2.4. RDF und RDFS

Das *Resource Description Framework* (RDF) [97] ist die erste Sprache, die speziell zur Beschreibung von Ressourcen im Web konzipiert wurde und daher häufig als Grundstein des Semantischen Webs angesehen wird. Neben der Beschreibung von Webressourcen lassen sich auch allgemeine Ontologiebeschreibungen der realen Welt mit RDF erfassen. RDF wird vom W3C standardisiert und liegt seit Februar 2004 als Empfehlung (W3C Recommendation) vor.

RDF beruht auf sehr einfachen **Ordnungsmechanismen**. Ressourcen des Webs sowie Entitäten und Beziehungen der realen Welt werden durch eindeutige *Uniform Resource Identifier* (URI) repräsentiert. Beschreibungen erfolgen durch *Aussagen* in Form von Triplets, in denen eine URI als Subjekt, eine URI als Prädikat und eine URI oder ein Literal als Objekt auftritt.

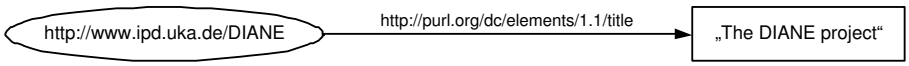


Abbildung 2.6.: Beispiel für eine Aussage in RDF.

Ein Beispiel für eine solche Aussage in graphischer Notation zeigt Abbildung 2.6. Hier ist ausgesagt, dass der Titel (festgelegt durch die Eigenschaft title aus dem Dublin Core Metadatensatz) der angegebenen Webseite (angegebenen durch deren URL) das String-Literal `"The DIANE project"` ist.

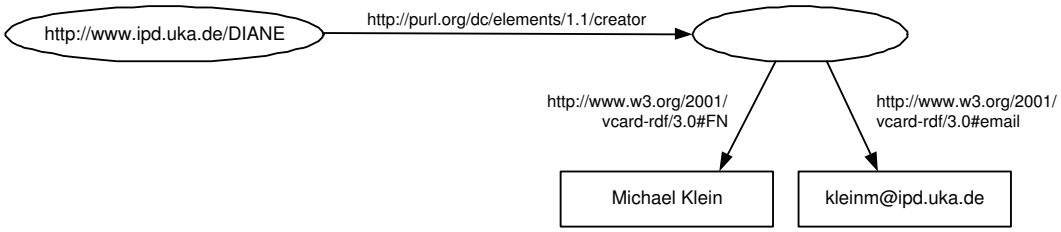


Abbildung 2.7.: Beispiel für eine Aussage mit leerem Knoten in RDF.

Zudem besteht die Möglichkeit, anonyme oder leere Knoten zu definieren, wie Abbildung 2.7 zeigt. Diese stellen Ressourcen dar, die nicht durch eine URI gekennzeichnet

sind, sondern allein durch die Angabe von Eigenschaften umschrieben sind. Im Beispiel ist der Autor der Webseite ein Individuum mit dem Namen `"Michael Klein"` und der angegebenen E-Mail-Adresse.

Prinzipiell unterliegt RDF keinen Beschränkungen hinsichtlich der verwendbaren Ressourcen, Literalen und Eigenschaften beim Aufstellen von Aussagen. Hierdurch wird das Ziel der Einigung auf Konzepte innerhalb der Anwendergruppe wesentlich erschwert. Aus diesem Grund existiert die *RDF Vocabulary Description Language – RDF Schema* (RDFS) [21] zur Definition von Vokabularien und Einschränkungen für Eigenschaften. Wichtigste Ordnungsmechanismen sind hier Klassen (`rdfs:Class`) und Eigenschaftstypen (`rdf:Property`)[8].

RDFS führt auch eine Reihe **ausgezeichneter Konstrukte** ein, etwa die Unterklassenbeziehung (`rdfs:subClassOf`), die Untereigenschaftsbeziehung (`rdfs:subPropertyOf`) sowie Möglichkeiten, den Definitionsbereich (`rdfs:domain`) sowie den Zielbereich (`rdfs:range`) für Eigenschaften einzugrenzen. RDF und RDFS werden zusammen als RDF(S) bezeichnet.

Typischerweise stellen existierende Modellierungssprachen Klassen in den Mittelpunkt des Entwurfs. Bei der Definition einer Klasse wird angegeben, welche Eigenschaften in Form von Attributen diese besitzt. Alle Änderungen an diesen Attributen erfordern das Einverständnis der Person bzw. der Gruppe, die die Klasse verwaltet. RDF wurde speziell zur Verwendung im Web entwickelt und verfolgt einen dezentralen Ansatz, bei dem Beziehungen gleichberechtigt und prinzipiell losgelöst neben Klassen stehen. Aus diesem Grund kann jeder Teilnehmer eigene Beziehungen definieren und für die Beschreibung seiner Ressourcen beliebige Beziehungen aus dem Internet verwenden, vorausgesetzt der Definitions- und Zielbereich sind passend.

Als Besonderheit von RDF kann die Fähigkeit zur Selbstreflexion angesehen werden. Zum einen besteht die Möglichkeit zur *Verdinglichung* (engl. reification), d.h. Aussagen in RDF selbst wieder als Ressourcen zu sehen, über die Aussagen getroffen werden können, andererseits ist das Metamodell zu RDF(S) selbst in RDF(S) definiert. So ist zum Beispiel die Eigenschaft `rdfs:subClassOf` selbst Instanz von `rdf:Property` und `rdfs:Class` als die Menge aller Klassen eine Instanz von sich selbst. Dieser Ansatz wird häufig kritisiert (siehe z.B. [117]), da Daten, Schema und Metaschema vermischt werden, was neben einer konfusen und verwirrenden Darstellung auch zu unendlichen Metamodellierungsketten führen kann. Mit dem Hinzufügen von Negation (wie in OWL, siehe Abschnitt 2.2.5) können dadurch leicht Paradoxien entstehen. Es existiert daher eine Reihe von Vorschlägen zur Vereinfachung und besseren Trennung der Ebenen, wie etwa in *RDFS(FA)* [51].

RDF(S) ist über einem graphbasierten Modell spezifiziert, es existiert daher eine abstrakte, graphbasierte **Syntax**, wie sie bereits oben vorgestellt wurde. Jede RDF(S)-

[8] Wie der Präfix zeigt, ist `rdf:Property` historisch bedingt bereits in RDF definiert, semantisch gesehen sollte es jedoch zu RDFS gehören.

Beschreibung ist dabei ein *RDF-Graph*, der aus einer Menge von *RDF-Triplets* besteht. Auf dieser abstrakten Syntax ist dann die Semantik einer Beschreibung definiert. Für die abstrakte Syntax existieren verschiedene konkrete Syntaxen. Am weitesten verbreitet ist *RDF XML* [13], eine auf XML basierende Notation, welche die Aussagen als verschachtelte Bäume darstellt, sowie *N3* [15], eine kompakte, gut lesbare Notation, die die Triplets ungeschachtelt auflistet.

Die **Semantik** von RDF(S) wird mit Hilfe der Modelltheorie festgelegt und liegt seit 2004 als Empfehlung des W3C vor [60]. Die Modelltheorie ist eine Theorie, die mittels einer Interpretation syntaktische Ausdrücke einer Sprache zu Entitäten der realen Welt zuordnet. Eine solche Interpretation ermöglicht es, die Bedeutung der ausgezeichneten Konstrukte zu definieren und damit Ableitungsregeln für das Schließen zu festzulegen. RDF(S) beschränkt sich dabei auf monotones Schließen, d.h. bereits abgeleitete Aussagen können durch Hinzunahme weiterer Aussagen nicht mehr falsch werden. Die Ableitungsregeln haben die Form *„Wenn Aussage der Form* $(s_1 p_1 o_1)$ *enthalten ist, dann füge Aussage* $(s_2 p_2 o_2)$ *hinzu“*. Sie umfassen im Wesentlichen

- die Vervollständigung um Aussagen zur Selbstreflexion, d.h. aufgrund der Aussage a aus Abbildung 2.6 würde unter anderem hinzugefügt, dass `http://www.ipd.uka.de/DIANE` Subjekt der Aussage a ist;
- die Vervollständigung um Aussagen, die die Transitivität und Reflexivität der Unterklassen- und Untereigenschaftsbeziehungen widerspiegeln.

Insgesamt stellt RDF(S) auf den ersten Blick eine einfache Ontologiesprache dar, mit der Ressourcen im Internet sowie andere Anwendungsgebiete beschreibbar sind. Die Sprache enthält jedoch eine Reihe von Tücken, die speziell aus der unsauberen Vermischung von Daten, Schema und Metaschema herrühren.

2.2.5. OWL

Auch die *Web Ontology Language* (OWL) [12] ist seit 2004 eine Empfehlung des W3C und kann als Erweiterung von RDF(S) um expressivere Konstrukte angesehen werden. Auch ihr Ziel ist eine formale, semantische Beschreibung von Webressourcen. Hervorgegangen ist OWL im Wesentlichen aus DAML+OIL, einer Kombination der *DARPA Agent Markup Language*[9] (DAML) und des *Ontology Inference Layer*[10] (OIL), die beide aus dem Gebiet der Agententechnologien stammen. OWL enthält im Vergleich zu RDF(S) bessere Schlussfolgerungsoperationen wie Konsistenzprüfung

[9] `http://www.daml.org`
[10] `http://www.ontoknowledge.org/oil`

und automatische Klassifizierung und soll so auch die weltweit verteilte Entwicklung größerer Ontologien ermöglichen.

Der **Ordnungsmechanismus** von OWL basiert auf Beschreibungslogik (siehe [8]), technisch ist OWL jedoch ein Aufsatz auf RDF(S), indem eine Reihe weiterer **eingebauter Konstrukte** eingeführt werden, welche in RDF(S) fehlen. Insbesondere sind dies Möglichkeiten zur Deklaration von Disjunktheit zweier oder mehrerer Klassen, die Möglichkeit, Klassen zu neuen Klassen zu kombinieren, die Gleichheit von Klassen anzugeben, enumerierte Klassen zu definieren, Kardinalitäten von Beziehungen zu spezifizieren und spezielle Eigenschaften von Beziehungen wie Transitivität, Symmetrie, Eindeutigkeit oder Inversität zu einer anderen Beziehung anzugeben. Die **Semantik** dieser Konstrukte ist wie in RDF(S) mit Hilfe der Modelltheorie festgelegt [120].

Kombiniert mit der Möglichkeit zur Selbstreflexion aus RDF(S), können die in OWL neu eingeführten ausgezeichneten Konstrukte leicht zu Problemen bei der Berechenbarkeit der Schlussfolgerungsoperationen führen. Aus diesem Grund wurden drei Klassen von OWL definiert, die sich im Grad ihrer Expressivität (insbesondere im Zusammenspiel mit RDF(S)) und der Effizienz der ausführbaren Schlussfolgerungsoperationen unterscheiden:

- *OWL Full.* Erlaubt den vollen Sprachumfang von OWL und beliebige Kombinationen zwischen OWL- und RDF(S)-Konstrukten. OWL Full ist vollständig abwärtskompatibel zu RDF(S). Aufgrund der hohen Expressivität sind die Operationen zur Schlussfolgerung jedoch nicht mehr entscheidbar. Auch in der Praxis existieren keine effizienten Implementierungen.

- *OWL DL.* Schränkt das Zusammenspiel zwischen OWL und RDF(S) so ein, dass insgesamt die Ausdrucksstärke der Beschreibungslogik (engl. description logic, DL) erreicht wird. Hierdurch sind effiziente Implementierungen der Schlussfolgerungsoperationen verfügbar. Allerdings ist die Kompatibilität mit RDF(S) nicht mehr gewährleistet, da nicht jedes gültige RDF(S)-Dokument auch ein gültiges OWL-DL-Dokument darstellt.

- *OWL Lite.* Wie OWL DL, jedoch sind einige Konstruktoren von OWL nicht erlaubt, etwa die Verwendung von enumerierten Klassen, die Angabe, dass zwei Klassen disjunkt sind oder die Verwendung beliebiger Kardinalitäten. Dies macht OWL Lite zwar vergleichsweise wenig ausdrucksstark, insgesamt jedoch einfacher zu verstehen und effizienter zu verarbeiten.

OWL bietet eine Reihe von **Schlussfolgerungsoperationen**, die aus der Beschreibungslogik bekannt sind:

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#madeFromGrape"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Abbildung 2.8.: Beispiel für eine typische OWL-Klassendefinition in RDF/XML-Syntax, entnommen aus [136]

- *Unterklassenbeziehung* (engl. subsumption). Testet, ob jedes Element der Klasse C auch Element der Klasse D ist.
- *Klassenzugehörigkeit.* Testet, ob eine Instanz x Element der komplex definierten Klasse D ist.
- *Klassenäquivalenz.* Testet, ob zwei Klassen D und E die gleichen Elemente enthalten.
- *Konsistenz.* Testet, ob eine Klasse D überhaupt ein Element enthält.
- *Klassifikation.* Bestimmt für eine Instanz x diejenigen Klassen, von denen es Element ist.

Für OWL existiert eine abstrakte **Syntax** [120], welche nicht zum Austauschen von Informationen gedacht ist. Vielmehr dient sie als Grundlage zur Definition der Semantik. Die abstrakte Syntax ist rahmenorientiert, d.h. anders als typische Syntaxen für Beschreibungslogik oder RDF(S) fasst sie Informationen über eine Klasse oder Eigenschaft in einem großen syntaktischen Konstrukt zusammen, anstatt sie auf viele atomare Einzeldefinitionen zu verteilen. Die eigentliche Austauschsyntax für OWL ist RDF/XML. Dazu wird durch eine Reihe von Transformationsvorschriften die abstrakte Syntax von OWL auf die graphbasierte Syntax von RDF(S) abgebildet [120], die in Form von RDF/XML ausgegeben werden kann.

Abbildung 2.8 zeigt ein Beispiel für eine OWL-Definition in RDF/XML-Syntax. Definiert wird die Klasse Wein (engl. Wine), als Unterklasse von Getränk (engl. PotableLiquid) und Unterklasse von Klassen, die die Eigenschaft erzeugtAusTraube (engl.

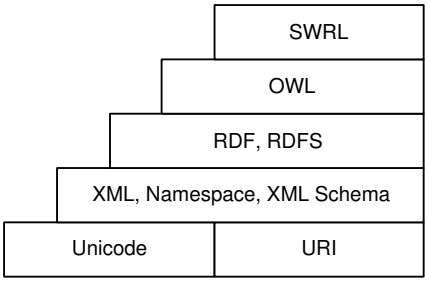


Abbildung 2.9.: Schichtung von Sprachen um OWL.

madeFromGrape) mindestens einmal definiert haben. Die eher verwirrende Mischung von Schlüsselwörtern aus OWL, RDF, RDFS, XML und XML Schema (XSD) ist gut zu erkennen.

Selbst die hohe Expressivität von OWL reicht in manchen Fällen nicht aus. Insbesondere für die Beschreibung von Diensten mit der OWL-basierten Sprache OWL-S (siehe Abschnitt 3.2) sind weitere Konstrukte in Form von Regeln nötig. Dies soll durch die *Semantic Web Rule Language* (SWRL) [66] erreicht werden, die auf OWL DL und OWL Lite aufgesetzt werden kann, um Horn-artige Regeln der Form $linkeSeite \rightarrow rechteSeite$ in die Ontologie einzubringen, etwa

```
parent(?x,?y) AND brother(?y,?z) IMPLIES uncle(?x,?z)
```

Um die Berechenbarkeit weiterhin zu gewährleisten, existiert eine Reihe von Empfehlungen, welchen Einschränkungen die Bestandteile der Regeln unterliegen sollten. SWRL soll durch das W3C standardisiert werden und liegt seit Mai 2004 als Einreichung (W3C Member Submission) vor.

Insgesamt ergibt sich bei der Verwendung von OWL die in Abbildung 2.9 dargestellte Schichtung von Sprachen. Auf der unteren Schicht sorgen Unicode und URIs für eine einheitliche Verwendung von Zeichen und Referenzen. Darüber sorgt XML mit dem Namespace-Mechanismus und den in XML Schema definierten Datentypen für eine einheitliche Syntax. Die grundlegende Semantik liefert dann RDF(S). Deren Expressivität kann dann mittels OWL und SWRL erhöht werden.

Als Aufsatz auf RDF(S) erbt OWL eine Vielzahl der Probleme. Insbesondere die unsaubere Syntax durch Vermischung verschiedenster Elemente und die Nichtentscheidbarkeit von OWL-Full müssen als problematisch angesehen werden. Modellierungen realer Anwendungsgebiete sind in OWL aufgrund der beschreibungslogischen Grundlage schwierig.

2.2.6. Zusammenfassung

Ziel des Semantischen Webs ist es, die sinngemäße Nutzung von Inhalten im Internet auch Rechnern zu ermöglichen. Grundtechnik zur Erreichung dieses Ziels stellt die Beschreibung der Dokumente mit Metadaten dar. Eine Nutzung dieser Annotationen ist jedoch nur sinnvoll, wenn diese Bezug auf Sachverhalte der realen Welt nimmt. Diese können durch Ontologien ausgedrückt werden, welche ein explizites, formales Vokabular bereitstellen und so eine Brücke zwischen realer Welt und Informationssystem schlagen. Sie werden durch Ontologiesprachen definiert, deren Bestandteile eine Syntax, einen Ordnungsmechanismus, vordefinierte Konstrukte sowie Schlussfolgerungsoperationen sind. Wichtige, im Rahmen des Semantischen Webs eingesetzte Ontologiesprachen sind F-Logic als formale Objektorientierung, RDF(S), welches die Welt in Form von Aussagentriplets erfasst, und OWL/SWRL als expressive Erweiterungen von RDF(S).

3. Stand der Forschung

Dieses Kapitel untersucht den Stand der Forschung auf dem Gebiet der Semantischen Webdienste. Es wird zunächst in Abschnitt 3.1 eine Liste von Anforderungen aufgestellt, welche ein Ansatz erfüllen muss, um tatsächlich für eine automatische und semantisch korrekte Dienstnutzung in dynamischen Umgebungen eingesetzt werden zu können. Diese Liste dient im Folgenden als Vergleichsgrundlage. Zunächst werden in Abschnitt 3.2 und 3.3 die zwei wichtigsten Vertreter semantischer Dienstbeschreibungssprachen, OWL-S und WSMO, im Detail vorgestellt und analysiert. Anschließend werden in Abschnitt 3.4 SWSF, METEOR-S, IRS-III, WSDL-S und weitere kleinere Ansätze untersucht. Eine Zusammenfassung, in welchem Maße die Anforderungen bei den einzelnen Ansätzen erfüllt sind, gibt dann Abschnitt 3.5 in Form einer Übersichtstabelle. Das Auffinden und automatische Nutzen von Funktionalität spielt in vielen Bereichen der Informatik eine Rolle, wie etwa im Bereich der Wiederverwendung von Komponenten, der Agentensysteme, des Grid Computings oder des Information Retrievals. Abschnitt 3.6 untersucht, welche relevanten Vorarbeiten in diesen Gebieten geleistet wurden. Ein Fazit beschließt das Kapitel. Es ist zu beachten, dass das Forschungsgebiet der semantischen Dienstbeschreibungen noch sehr dynamisch ist und sich vermutlich noch stark weiterentwickelt. Das Kapitel stellt somit eine Momentaufnahme des Standes vom Dezember 2005 dar.

3.1. Anforderungen an semantische Dienstbeschreibungssprachen

Semantische Webdienste stellen eine Vereinigung von Webdiensten und Konzepten des Semantischen Webs dar. Es handelt sich dabei um ein noch relativ junges Forschungsgebiet, das sich als Ziel gesetzt hat, die Nutzung jedweder Dienste (also auch kombinierter oder solcher mit komplexer Choreographie) zu automatisieren und dabei semantische korrekte Ergebnisse zu erzielen. Die Ziele sind daher eine Obermenge der Ziele dieser Arbeit. Grundlage ist in allen Ansätzen eine geeignete semantische Dienstbeschreibungssprache, welche genügend und eindeutige Information liefert, sodass ein Rechner in der Lage ist, die Dienstnutzung selbstständig durchzuführen. Um dieses Ziel erreichen zu können, muss die Dienstbeschreibungssprache eine Reihe von Anforderungen erfüllen. Diese Liste wird im Folgenden vorgestellt. Grundlage sind

A1	Konzeptionelles Modell
A2	Universal
A3	Inhaltlich eindeutig
A4	Angemessene Beschreibung für angebotene und benötigte Dienste
A5	Eindeutige funktionale Beschreibung angebotener Dienste
A6	Deterministische Beschreibung benötigter Dienste
A7	Formal fundierte, rechnerverständliche Semantik
A8	Effektiv vergleichbar
A9	Effizient vergleichbar
A10	Unabhängig erstellbar
A11	In Applikationen einbindbar
A12	Verarbeitbar durch Menschen
A13	Positiv evaluiert

Abbildung 3.1.: Liste der Anforderungen an semantische Dienstbeschreibungssprachen.

die Publikationen [34], [91] und insbesondere [82], die sich mit allgemeinen Anforderungen an semantische Dienstbeschreibungssprachen beschäftigen. Die Liste dient im Laufe des Kapitels als Metrik, um die Stärken und Schwächen der existierenden Ansätze objektiver überprüfen zu können.

Einen Überblick über die generellen Anforderungen an eine semantische Dienstbeschreibungssprache B zeigt die Tabelle in Abbildung 3.1. Im Detail sind das:

- **Anforderung A1: Konzeptionelles Modell.** B sollte ein konzeptionelles Modell zugrunde liegen, welches generelle Begriffe (wie *Dienst*, *Effekt*, *Dienstnehmer* etc.) definiert und welches den Entwurf der Sprache maßgeblich mitbestimmt. Dies ist wichtig, damit die Sprache nicht technologiegetrieben (wie können aktuelle Technologien für eine Dienstbeschreibung verwendet werden?), sondern problemgetrieben (was zeichnet einen Dienst aus und wie müsste eine Technik aussehen, die das zu beschreiben vermag?) entwickelt wird.
- **Anforderung A2: Universal.** Mittels B sollten prinzipiell Dienste aller Domänen beschrieben werden können, d.h. insbesondere darf sich B nicht auf einzelne Anwendungsgebiete festlegen. Dabei muss der initiale und fortlaufende Aufwand zur Integration und Wartung einer neuen Domäne angemessen sein. Dies ist wichtig, um nicht bereits im Voraus bestimmte Einsatzgebiete zu verhindern.
- **Anforderung A3: Inhaltlich eindeutig.** Beim Verarbeiten von Beschreibungen in B darf es keine inhaltlichen Mehrdeutigkeiten bezüglich der Begriffe zur Erfassung einer Domäne geben, in welcher der Dienst operiert. Diese Forderung ist essenziell, da sonst die automatische Verarbeitung in unerwünschter Weise ablaufen kann.

- **Anforderung A4: Angemessene Beschreibung für angebotene und benötigte Dienste.** Die Ziele bei der Beschreibung angebotener und benötigter Dienste sind grundsätzlich verschieden, was sich in einer Trennung der entsprechenden Beschreibungselemente in B widerspiegelt sollte.

- **Anforderung A5: Eindeutige funktionale Beschreibung angebotener Dienste.** Für Angebotsbeschreibungen in B muss klar ersichtlich sein, welche Funktionalität durch ihre Ausführung zu erwarten ist. Dies gilt insbesondere dann, wenn die Funktionalität nicht im Voraus eindeutig festgelegt ist, sondern durch den Austausch von Nachrichten konfiguriert werden kann. Der Zusammenhang zwischen Informationsfluss und Zustandsübergang muss daher eindeutig ablesbar sein, da eine automatische Verarbeitung sonst zu unerwünschtem Verhalten führen kann.

- **Anforderung A6: Deterministische Beschreibung benötigter Dienste.** Für Anfragebeschreibungen in B muss klar ersichtlich sein, welche Funktionalität der Dienstnehmer mit ihrer Absendung erwirkt haben möchte. Insbesondere muss B dem Dienstnehmer gestatten, seine Präferenzen bezüglich nicht exakt passender Dienstangebote in die Beschreibung einbringen zu können. Dies ist wichtig, da er nach Absenden der Anfragebeschreibung und während der automatischen Verarbeitung nicht mehr auf die Auswahl eines für ihn geeigneten Dienstgebers Einfluss nehmen kann.

- **Anforderung A7: Formal fundierte, rechnerverständliche Semantik.** Für Beschreibungen in B muss formal definiert sein, was sie bedeuten. Dies gilt insbesondere für die nicht-statischen Teile einer Beschreibung wie etwa die Beschreibung des Zustandsübergangs, des Nachrichtenflusses oder des Effektwunsches. Besondere Aufmerksamkeit gilt dabei der Semantik an „Nahtstellen“ zwischen unterschiedlichen, gemischt verwendeten Sprachen. Eine formale Semantik ist unerlässlich, um daraus die Operationen (insbesondere den Vergleich) auf Beschreibungen der Sprache korrekt ableiten zu können.

- **Anforderung A8: Effektiv vergleichbar.** Für Angebots- und Anfragebeschreibungen in B muss ein Vergleichsverfahren V zur Verfügung stehen, dessen Ergebnisse verwendet werden können, um daraus eine semantisch korrekte Ausführung eines angebotenen Dienstes einleiten zu können. Es ist daher wichtig, dass das Vergleichsergebnis für eine maschinelle Weiterverarbeitung bestimmt ist. Darüber hinaus sollte sich ein Vergleichsergebnis möglichst gut mit dem intuitiven Verständnis der Teilnehmer decken.

- **Anforderung A9: Effizient vergleichbar.** Das Vergleichsverfahren von V muss berechenbar sein und der benötigte Zeitaufwand zur Berechnung von Vergleichsergebnissen muss in einem akzeptablen Rahmen liegen. Dies ist insbeson-

dere dann wichtig, wenn wie im Falle des Geschäftsprozessszenarios eventuell mehrere hundert Dienstangebote gegen eine Anfrage verglichen werden müssen.

- **Anforderung A10: Unabhängig erstellbar.** Beschreibungen in B müssen zur Entwurfszeit unabhängig von existierenden oder noch unbekannten Beschreibungen erstellt werden können, ohne dass dadurch zur Laufzeit eigentlich passende Dienste unpassende Beschreibungen aufweisen. Dies ist wichtig, da für eine dynamische Dienstbindung von einer vorher unbekannten Dienstlandschaft ausgegangen werden muss, was insbesondere bei internetbasierten Geschäftsprozessen oder in mobilen Umgebungen der Fall ist.

- **Anforderung A11: In Applikationen einbindbar.** Dienstanfragebeschreibungen in B müssen bereits zur Entwurfszeit von Anwendungsentwicklern in Applikationen einbindbar sein, ohne dass bekannt ist, welcher konkrete Dienstgeber zur Laufzeit die Funktionalität wie erfüllen wird. Beschreibungen in B müssen sich daher eignen, zwischen den bereits zur Entwurfszeit festgelegten Erwartungen der Anwendung und dem konkret genutzten Dienst zur Laufzeit zu vermitteln.

- **Anforderung A12: Verarbeitbar durch Menschen.** Beschreibungen in B und auch dazu nötige grundlegende Beschreibungen der realen Welt müssen von menschlichen Benutzern fehlerfrei und unter mäßigem Zeitaufwand erstellbar sein.

- **Anforderung A13: Evaluiert.** Die Konzepte von B sollten nicht nur theoretischen Betrachtungen standhalten, sondern auch in praktischen und realistischen Szenarios evaluiert worden sein. Erst bei einem tatsächlichen Einsatz werden die Stärken und Schwächen einer Sprache sichtbar.

Die Tabelle wird in Abschnitt 3.5 wieder aufgegriffen, um eine Übersicht zu geben, in wie weit die vorgestellten Ansätze die Anforderungen erfüllen. Hierzu werden im Folgenden zunächst die beiden großen Ansätze OWL-S und WSMO, danach kleinere Ansätze vorgestellt und analysiert.

3.2. OWL-S

Die Web Ontology Language - Services (*OWL-S*) [5] stellt eine Ontologie auf Basis von OWL (siehe Abschnitt 2.2.5) dar, um Webdienste semantisch zu beschreiben. Eingeführt wurde sie im Jahr 2000 als Forschungsprojekt der *Defense Advanced Research Projects Agency* (DARPA) noch unter dem Namen DAML-S (für *DARPA Agent Markup Languages for Services*) und basierte zunächst auf dem OWL-Vorgänger DAML. Die Sprache wurde seit Beginn von einer relativ geschlossenen Gruppe bestehend aus

Forschungseinrichtungen an Universitäten und der Industrie entwickelt. Mit Erreichen der Version 1.1 im November 2004 wurde OWL-S als *Member Submission* zur Standardisierung beim World Wide Web Consortium eingereicht.

Ziel von OWL-S ist es, Werkzeuge und Technologien bereitzustellen, mit deren Hilfe die Dienstnutzung im Semantic Web automatisiert werden kann. Dazu stellt OWL-S eine grundlegende Menge von Sprachkonstrukten zur Verfügung, mit denen Eigenschaften und Fähigkeiten von Webdiensten eindeutig und computerverständlich beschreibbar sein sollen. Dadurch sollen die Dienstvermittlung, die Dienstausführung, die Kooperation von Dienstgebern, die Komposition von Diensten sowie die Überwachung des Dienstablaufs automatisiert werden.

3.2.1. Aufbau von OWL-S

OWL-S basiert im Wesentlichen auf drei Erweiterungen im Vergleich zu WSDL:

- *Ideen des Semantischen Webs.* OWL-S befreit Dienstbeschreibungen von allgemeinem Wissen über die Welt, indem es dieses in Form einer Beschreibungslogik-basierten Ontologie ablegt und referenziert.
- *Ideen aspektorientierter Softwareentwicklung.* OWL-S beschreibt einen Dienst aus drei quasi orthogonalen Sichtweisen: Die Beschreibung der Funktionalität im ServiceProfile, die Beschreibung der Choreographie[1] im ServiceModel sowie den technischen Zugangsweg im ServiceGrounding.
- *Ideen aus dem Bereich der Agententechnologie.* OWL-S beschreibt im Gegensatz zu WSDL nicht nur den Informationsfluss eines Diensten, sondern auch den Zustandsübergang, den dieser erzielen kann.

In OWL-S werden konkrete Dienstbeschreibungen von allgemeinem Wissen getrennt. Dies ermöglicht es, Wissen über die Welt in eigenständige, OWL-basierte Ontologien auszulagern. Auch OWL-S selbst ist eine in OWL definierte Ontologie; eine konkrete Dienstbeschreibung ist eine Instanz der dort definierten Konzepte sowie Referenzen auf Allgemeinwissen. Die OWL-Basis ermöglicht die Verwendung logischer Schlussfolgerungsoperationen auf Dienstbeschreibungen, etwa zum Durchführen von Vergleichen oder zur Konsistenzprüfung (siehe unten). Die Mächtigkeit von OWL alleine reicht jedoch nicht aus, um Dienste korrekt beschreiben zu können. Für OWL-S

[1] Choreographie bezeichnet den korrekten Ablauf zwischen der Kommunikation von Dienstnehmer und Dienstgeber.

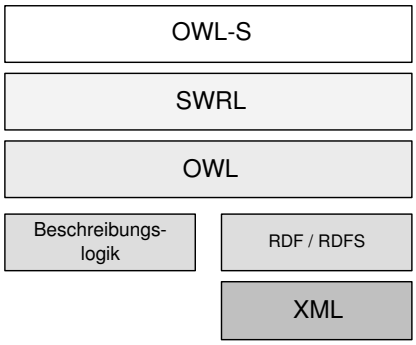


Abbildung 3.2.: Schichtung von Semantiksprachen unter OWL-S

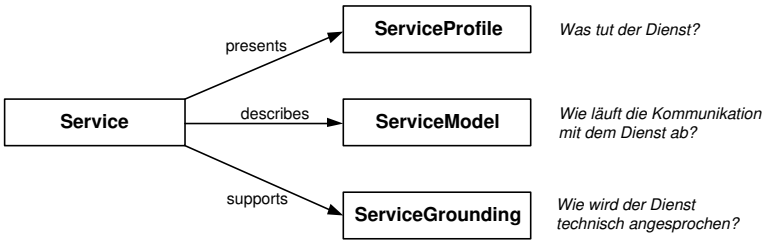


Abbildung 3.3.: Obere Dienstontologie von OWL-S.

wird zusätzlich eine Regelsprache benötigt, insbesondere um Zusammenhänge zwischen verschiedenen Parametern auszudrücken. Diese Aufgabe wird von SWRL übernommen (siehe Abschnitt 2.2.5), welche auf OWL aufgesetzt wird. Die komplette Sprachschichtung von OWL-S zeigt Abbildung 3.2.

Basis für Beschreibungen in OWL-S ist die domänenunabhängige *obere Dienstontologie*. Ausgehend von dem Konzept Service werden hier drei Aspekte des Dienstes getrennt dargestellt (siehe Abbildung 3.3):

- Im ServiceProfile (dt. Dienstprofil) wird auf hohem Abstraktionsniveau beschrieben, welche Leistung der Dienst erbringen kann. Der Teil ist daher insbesondere für die Suche von geeigneten Diensten wichtig.
- Im ServiceModel (dt. Dienstmodell) wird der Ablauf beschrieben, der aus Sicht des Dienstnehmers korrekt und vollständig durchgeführt werden muss, um die beschriebene Leistung zu erhalten (die so genannte *Choreographie*). Der Teil kann für die Auswahl eines Dienstgebers herangezogen werden, ist aber insbesondere in der Phase der Dienstausführung von Bedeutung.
- Im ServiceGrounding (dt. Dienstfundament) wird beschrieben, wie der Dienst technisch angesprochen werden kann. Der Teil ist daher in der Phase der realen Dienstausführung von Bedeutung.

Im Folgenden werden die drei Aspekte genauer vorgestellt.

Dienstprofil

Ziel des Dienstprofils ist es, die Fähigkeiten des Dienstes darzustellen, d.h. anzugeben, „was der Dienst macht“. Mit dieser Beschreibung muss ein Agent in der Lage sein festzustellen, ob der Dienst für seine Zwecke geeignet ist. Eine solche Beschreibung umfasst daher im Wesentlichen die Leistungen des Dienstes, seine Anforderungen, seine Einschränkungen und seine Qualität. Im Detail enthält sie die folgenden Bestandteile:

- *Menschenlesbare Informationen.* Dies sind Angaben, die nicht von einem Computeragenten im Rahmen der Dienstsuche einbezogen werden können oder sollen. Hierunter fallen der Name des Dienstes, eine textuelle Beschreibung und Informationen zum Anbieter des Dienstes (etwa dessen elektronische Visitenkarten *vCard*).
- *Klassifizierung.* Diese beschreibt die Einordnung des Wirtschaftszeigs des Dienstes (oder auch nur Teile des Dienstes, wie etwa die verkauften Produkte) in existierende, standardisierte Taxonomien wie UNSPSC[2] oder NAICS[3] oder selbstdefinierte Kategorisierungsschemata. OWL-S selbst definiert kein solches Schema.
- *Funktionale Beschreibung.* Diese erfasst die Funktionalität des Dienstes in maschinenverständlicher Form und bildet damit den Kern des Dienstprofils. Die Beschreibung erfolgt getrennt nach Informationstransformation (welche Eingaben erwartet der Dienst, welche Ausgaben liefert er?) und Zustandsübergang (welche Vorbedingungen benötigt der Dienst, welche Resultate in Form neuer Zustände liefert er in welchen Fällen?). Wie Abbildung 3.4 zeigt, stehen hierzu die Attribute input, output, precondition und result zur Verfügung (nach ihren

[2]United Nations Standard Products and Services Code, `www.unspsc.org`
[3]North American Industry Classification System, `www.census.gov/epcd/www/naics.html`

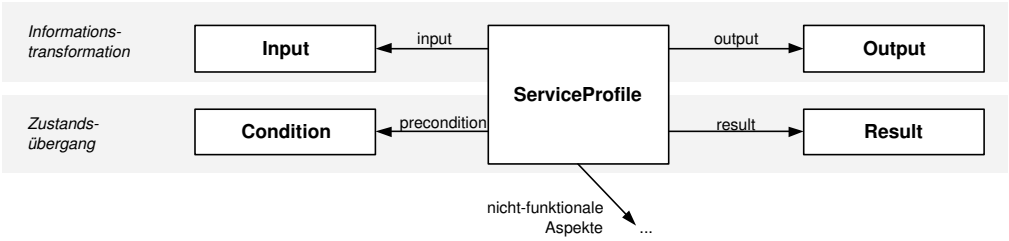


Abbildung 3.4.: Das Dienstprofil in OWL-S.

Anfangsbuchstaben auch kurz *IOPRs* genannt). In einer konkreten Dienstbeschreibung sind diese nicht im Dienstprofil gefüllt, sondern zeigen auf Instanzen im Dienstmodell. Dort findet auch die Verknüpfung der IOPRs untereinander statt.

- *Nicht-funktionale Beschreibung.* Diese erfasst im Wesentlichen Einschränkungen und die Qualität des Dienstes, etwa Sicherheitsgarantien, zu erwartende Leistung etc. In OWL-S ist dieser Teil jedoch noch sehr generisch gehalten.

Dienstmodell

Ziel des Dienstmodells ist es zu beschreiben, welche Schritte der Dienstnehmer zu unternehmen hat, um korrekt mit dem Dienstgeber zu interagieren. Angegeben sind daher Details, wann und wie die IOPRs aus dem Dienstprofil eine Rolle spielen. Dennoch ist das Dienstmodell nach wie vor eine abstrakte Beschreibung, die noch keine Details zu konkreten Nachrichtenformaten oder Kommunikationsprotokollen enthält.

Wichtigstes Konzept im Dienstmodell ist der *Prozess*. Ein Prozess repräsentiert eine Transformation von Daten und Zuständen. Dienste können aus einem oder mehreren Prozessen zusammengesetzt sein (siehe unten). Beschrieben wird ein Prozess durch die vier aus dem Dienstprofil bekannten IOPRs:

- hasInput. Beschreibt die Eingabedaten, die der Prozess benötigt. Typ von hasInput ist Parameter. Sein Attribut parameterType zeigt auf das Konzept, von welchem Instanzen als Eingabe verlangt werden.
- hasOutput. Beschreibt die Ausgabedaten, die im Prozess erzeugt und zurückgeliefert werden. Typ von hasOutput ist Parameter. Sein Attribut parameterType zeigt auf das Konzept, von welchem die ausgegebenen Instanzen sein werden.
- hasPrecondition. Beschreibt die Bedingungen an die Welt, die erfüllt sein müssen, damit der Prozess korrekt ablaufen kann. In OWL-S werden solche Bedingungen in SWRL ausgedrückt.
- hasResult. Beschreibt das Ergebnis des Dienstes in Abhängigkeit einer bestimmten Bedingung. inCondition enthält diese Bedingung als SWRL-Bedingung, withOutput legt fest, wie die Ausgabedaten in diesem Fall aussehen, hasEffect erfasst die Änderungen der realen Welt als SWRL-Bedingung, die aufgrund der Durchführung des Prozesses wahr wird.

Wichtiges Hilfsmittel zur Beschreibung von Prozessen ist daher das auf OWL aufgesetzte SWRL. Die Integration von SWRL in OWL-S ist erst seit OWL-S 1.1 definiert

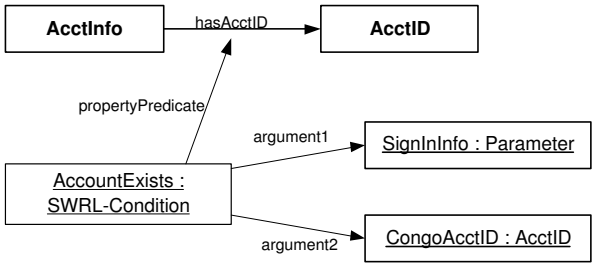


Abbildung 3.5.: Beispiel für eine SWRL-Regel aus einer OWL-S-Dienstbeschreibung, entnommen aus dem Gesamtbeispiel in Abbildung 3.6 (in eigener graphischer Darstellung).

und benötigt im Wesentlichen SWRL-Bedingungen als spezielle Regeln. Eine SWRL-Bedingung besteht aus einem Prädikat (predicate), welches mit einer Beziehung aus OWL belegt werden kann, und einem oder mehreren Argumenten (argument$i$), die mit belieben Instanzen gefüllt werden können, typischerweise mit Eingabe- oder Ausgabeparameterinstanzen der Dienstbeschreibung (siehe Abbildung 3.5 als Beispiel für eine Regel, die besagt, dass ein Account für die eingegebenen Werte existieren muss). Mittels SWRL ist es daher möglich, die IOPRs untereinander zu referenzieren und so in Beziehung zu setzen.

Ein größeres Gesamtbeispiel für einen Prozess zeigt Abbildung 3.6. Dieses wurde von der offiziellen Webseite entnommen und zur besseren Übersicht in eine eigene graphische Darstellung überführt. Dargestellt ist ein atomarer Prozess eines fiktiven Buchkaufdienstes *CongoBookBuy*. Der Prozess benötigt zwei Eingaben (ersichtlich an hasInput): die ISBN des Buches sowie Login-Informationen. Der Prozess hat die aus Abbildung 3.5 bekannte Vorbedingung, dass ein zu den Eingaben passender Account existieren muss. Das Ergebnis des Dienstes wird durch hasResult dargestellt. Der positive Fall tritt ein, wenn die spezifizierte ISBN ein Buch darstellt und dieses im Lager vorhanden ist (inConditions). In diesem Fall, tritt der ShippedEffect auf, der das gewünschte Buch im Rahmen eines Shipments an den angegebenen Account versendet, was durch zwei SWRL-Expressions zum Ausdruck kommt. Der negative Fall, der eintritt, wenn das Buch nicht verfügbar ist, wurde aus Platzgründen in der Zeichnung weggelassen.

Das Dienstmodell unterscheidet neben dem bereits vorgestellten atomaren Prozess weitere Arten von Prozessen (siehe Abbildung 3.7):

- *Atomare Prozesse* (AtomicProcess), die aus Sicht des Dienstnehmers in einem einzigen Schritt ausgeführt werden, d.h. sie erwarten einen einzelnen Nachrichtenaustausch zwischen Dienstnehmer und Dienstgeber. Atomare Prozesse sind direkt ausführbar, d.h. das genaue Nachrichtenformat muss im Dienstfundament spezifiziert werden (siehe unten).

- *Zusammengesetzte Prozesse* (CompositeProcess), die aus Sicht des Dienstnehmers mehrere Schritte erfordern. Dazu wird eine Reihe von Nachrichten zwischen Dienstnehmer und Dienstgeber ausgetauscht, wobei sich beide einen internen Zustand merken. Zusammengesetzte Dienste hinterlegen ihre konkreten technischen Details nicht im Dienstfundament, sondern werden auf einfache Prozesse zurückgeführt.

- *Einfache Prozesse* (SimpleProcess), dienen als Zwischenstück zwischen atomaren und zusammengesetzten Prozessen. Sie stellen zusammengesetzte Dienste als Einheit dar.

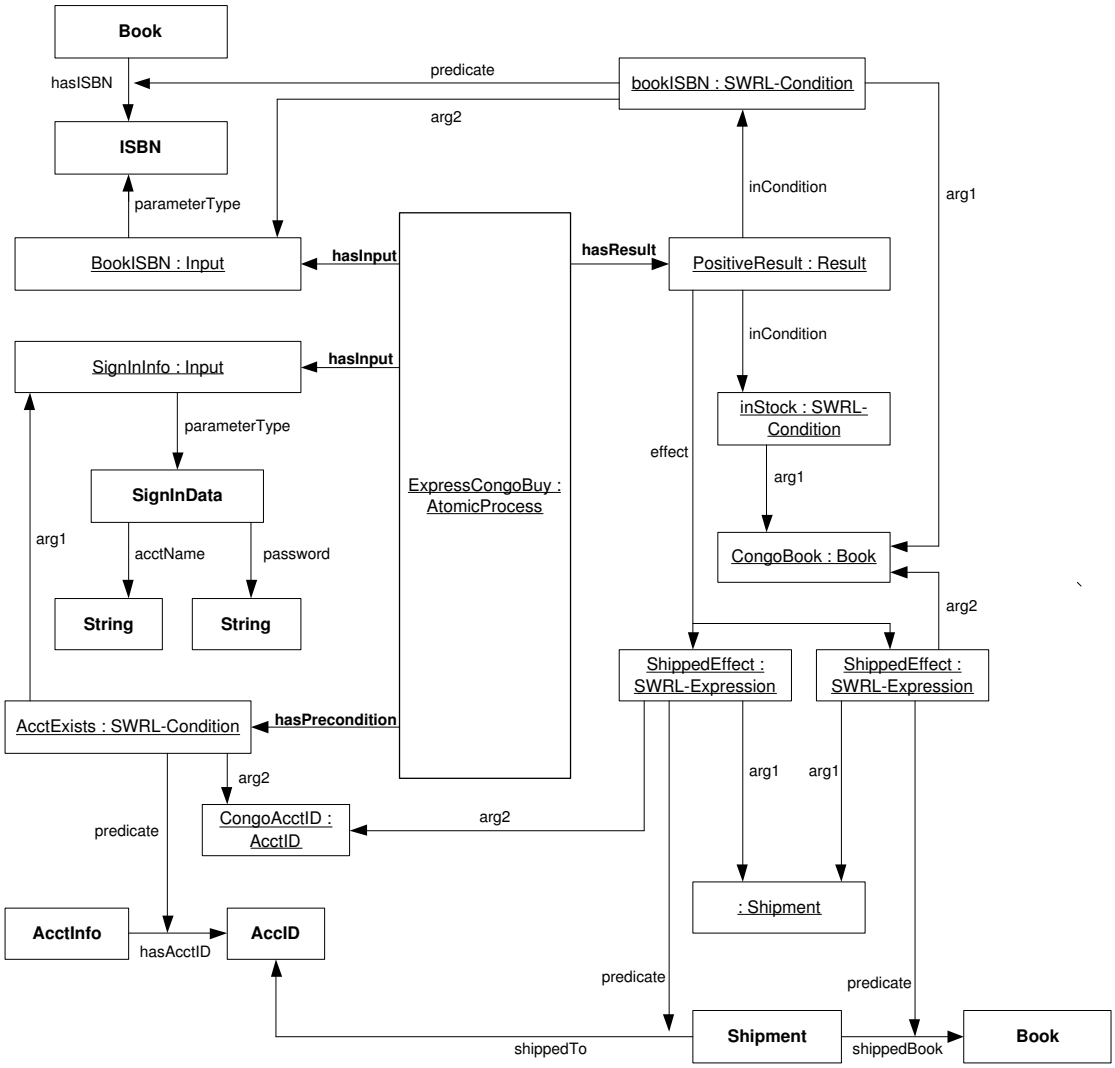


Abbildung 3.6.: Beispiel für einen atomaren Prozess in OWL-S (in eigener graphischer Darstellung).

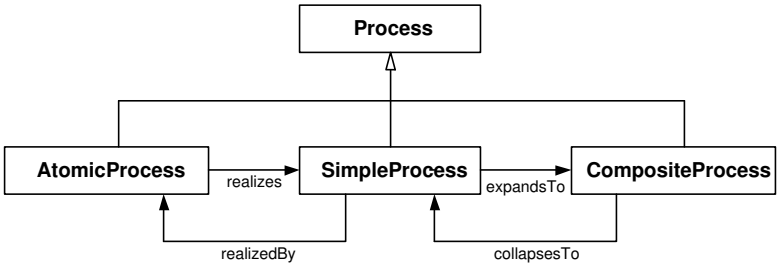


Abbildung 3.7.: Prozessarten in OWL-S.

Zusammengesetzte Prozesse spezifizieren den zeitlichen Ablauf ihrer Subprozesse durch Kontrollkonstrukte wie sie aus imperativen Programmiersprachen bekannt sind. Zur Verfügung stehen die Nacheinanderverarbeitung (Sequence) von Einzelprozessen, die Nebenläufigkeit (Split), die Nebenläufigkeit mit Synchronisationspunkt (Split-Join), die Ausführung in beliebiger Reihenfolge (Any-Order), die Auswahl eines beliebigen Einzelprozesses (Choice) und Schleifen (Iterate, Repeat-While und Repeat-Until) Die Beschreibung zusammengesetzter Prozesse ähnelt zwar Programmcode, darf jedoch nicht damit verwechselt werden. Programmcode kann selbstständig ausgeführt werden, eine Prozessbeschreibung muss vom Dienstnehmer befolgt werden, um zum beschriebenen Ergebnis zu gelangen. Das Dienstmodell bietet auch die Möglichkeit, die Datenflüsse zwischen den Einzelprozessen zu beschreiben.

Dienstfundament

Im Dienstfundament (ServiceGrounding) wird beschrieben, wie auf den Dienst technisch zugegriffen werden muss. Wichtigste Aufgabe ist, die abstrakten Nachrichten der atomaren Prozesse des Dienstmodells auf konkrete, über das Netz versendbare Nachrichten abzubilden. Des Weiteren werden Details über das zu verwendende Kommunikationsprotokoll, die eingesetzte Serialisierung etc. festgelegt. Ist der reale Dienst ein klassischer Webdienst kann er über ein Fundament auf Basis von WSDL eingebunden werden. Die abstrakten OWL-S-Nachrichten werden dann auf *WSDL:Messages* abgebildet, die ihrerseits über SOAP konkretisiert werden. Generell gilt folgende Abbildung: atomare Prozesse werden auf *WSDL:Operations* abgebildet, ihre Ein- und Ausgabeparameter auf *WSDL:Messages*. Die darin verwendeten komplexen Typen können prinzipiell direkt als OWL-XML-Code in WSDL integriert werden. In der Regel ist der reale Dienstgeber jedoch nicht in der Lage, die OWL-Typen der Domänenontologien zu verstehen. Aus diesem Grund existiert in OWL-S die Möglichkeit, die Typen mittels einer XSL-Transformation umzuwandeln.

Wichtiges Merkmal des Dienstfundaments von OWL-S ist die *direkte Kommunikation* zwischen Dienstgeber und Dienstnehmer nach der Dienstvermittlung. Dies erfordert einerseits, dass das Dienstfundament an den Dienstnehmer ausgeliefert wird,

andererseits, dass dieser in der Lage sein muss, im gewünschten Protokoll mit dem Dienstgeber zu kommunizieren.

3.2.2. Vergleich von OWL-S-Beschreibungen

Während die Sprache OWL-S im Wesentlichen von der abgeschlossenen Koalition festgelegt wird, erfolgt die Entwicklung von geeigneten Vergleichsverfahren hauptsächlich durch externe Gruppen. Es existiert daher kein empfohlener Vergleicher für OWL-S, sondern eine Reihe konkurrierender Ansätze.

Vergleicher für OWL-S sollten im Prinzip ausschließlich die Dienstprofile der betrachteten Beschreibungen in Betracht ziehen, da diese abstrakt beschreiben, welche Leistungen vom Dienst erbracht bzw. erwartet werden. Die meisten Vergleicher beruhen auf daher einem gemeinsamen Grundansatz, welcher der Vorgehensweise bei WSDL ähnelt. Dabei wird überprüft, ob die IOPRs eines angebotenen Dienstes zu denen des benötigten Dienstes passen. Die Algorithmen unterscheiden sich im Wesentlichen dadurch, was als „passend“ angesehen wird. Sie können in zwei Gruppen eingeteilt werden: typbasierte Ein-/Ausgabe-Vergleicher und beschreibungslogikbasierte Ein-/Ausgabe-Vergleicher.

Die reine Betrachtung des Dienstprofils erweist sich jedoch als schwierig, da das Profil bei den wichtigen funktionalen Parametern Verweise auf das Dienstmodell enthält. Es existieren daher auch Vergleicher, die weitere Informationen (z.B. aus dem Dienstmodell oder der impliziten Semantik) in Betracht ziehen, um so die Ergebnisse zu verbessern.

Im Folgenden werden typbasierte und beschreibungslogikbasierte Typvergleicher sowie Modellvergleicher und andere Ansätze vorgestellt. Vergleicher, die auch die formalen SWRL-Bedingungen der Vor- und Nachbedingungen berücksichtigen, existieren zurzeit noch nicht.

Typbasierte Ein-/Ausgabe-Vergleicher

Typbasierte Ein-/Ausgabe-Vergleicher ziehen von den IOPRs nur die Beschreibungen der Ein- und Ausgaben in Betracht und untersuchen, ob die Typen zueinander passen. Dabei gehen sie von folgender These aus: Generell passt eine Angebotsbeschreibung für einen Dienst o auf eine Anfragebeschreibung r, wenn alle Ausgaben von o auf die angeforderten Ausgaben passen und auch alle Eingaben für o mit denen von r übereinstimmen. Hierdurch soll garantiert werden, dass der Anfrager alle vom Dienst benötigten Eingaben liefern kann und die gelieferten Ausgaben seine Erwartungen erfüllen. Je nach konkretem Algorithmus werden verschiedene Passungsgrade unterschieden, die aus der Ober- bzw. Unterklassenbeziehung der Ein- und Ausgabe-Typen

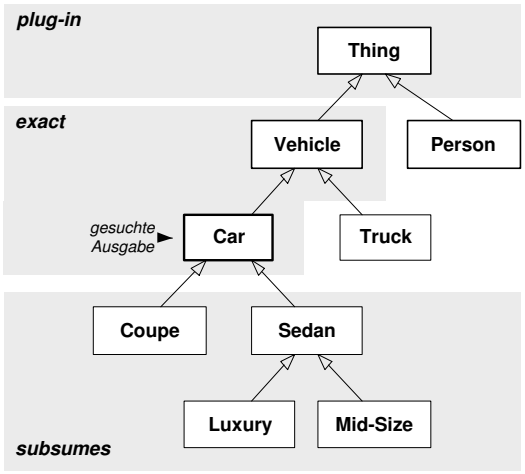


Abbildung 3.8.: Bildliche Darstellung der Ähnlichkeitsstufen des *Semantic Matchmakers* an einem Beispiel. In der Anfrage sei nach einer Ausgabe vom Typ Car gesucht. Die schraffierten Flächen geben an, welche Ähnlichkeitsstufe Typen als Ausgaben in Angebotsbeschreibungen zugewiesen würde.

bestimmt werden. Die funktionalen Parameter precondition und result werden nicht berücksichtigt.

Den bekanntesten Ansatz für einen solchen Vergleicher liefern Paolucci et al. mit dem **Semantic Matchmaker** in [118, 121]. Da eine exakte Übereinstimmung aller Typen in den seltensten Fällen möglich ist, wird der Übereinstimmungsgrad `exact` so definiert, dass der Ausgabeparameter out_r einer Nachfrage schon dann „exakt“ mit dem Ausgabeparameter out_o des Angebots übereinstimmt, wenn out_r zumindest eine direkte Unterklasse von out_o ist[4]. Dabei wird davon ausgegangen, dass ein Anbieter eines Dienstes nur dann einen Ausgabeparameter vom Typ der Oberklasse verwenden darf, wenn seine Ausgabe auch *alle* Typen der direkten Unterklassen unterstützt. In solchen Fällen offeriert der Anbieter also mehr als der Suchende verlangt.

Weitere Abschwächungen von Gleichheit werden durch die Einführung weiterer Ähnlichkeitsstufen erreicht: `plugIn`, `subsumes` und `fail` (siehe Abbildung 3.8). Für Ausgabeparameter out_r und out_o gilt `plugIn`, wenn out_r eine beliebige, nicht-direkte Unterklasse von out_o ist. In solchen Fällen könnten die Ausgaben möglicherweise nützlich sein: Der Anbieter offeriert zwar Parameter des gewünschten Obertyps, hierunter muss jedoch nicht zwangsläufig eine Instanz des in der Anfrage verlangten Typs sein. Für Ausgabeparameter out_r und out_o gilt `subsumes`, wenn out_r eine echte Oberklasse von out_o ist. Dann erfüllt der Anbieter keinesfalls die gesamte Anfrage, maximal Teile der Ausgabe könnten Verwendung finden. Besteht keine Ober- oder Unterklassenbeziehung zwischen den Parametern gilt die Ähnlichkeitsstufe `fail`.

[4]Bei Eingabeparametern in_r und in_o ist die Beziehung vertauscht, d.h. in_o muss direkte Unterklasse von in_r sein.

Ein weiterer Vergleicher für OWL-S-Beschreibungen wurde von Sirin et al. im Rahmen des **MIND SWAP-Projektes** entwickelt [133]. Hier wird ein zweistufiges Vorgehen vorgeschlagen: Im ersten Schritt wird überprüft, ob die Typen der Ein- und Ausgaben zueinander passen. Unterschieden werden dazu zwei Gleichheitsstufen: `exact`, also eine genaue Typübereinstimmung zwischen den Ein- und Ausgaben, sowie `generic`, wofür bereits eine Unterklassenbeziehung ausreichend ist. Auch hier werden precondition und result nicht berücksichtigt. Im zweiten Schritt findet ein Filtern auf Basis nicht-funktionaler Attribute statt. Hierbei sind jedoch keine Bedingungen in der Anfrage möglich.

Auch in den Arbeiten von **Constantinescu** und **Faltings** wird ein ähnliches Verfahren angewandt [29]. Sie definieren und benennen die Ähnlichkeitsstufen jedoch etwas anders. Für out_o und out_r gilt beispielsweise:

- `exact`, wenn die Typen von out_o und out_r gleich sind.
- `container`, wenn out_o ein echter Untertyp von out_r ist.
- `contained`, wenn out_o ein echter Obertyp von out_r ist.
- `overlap`, wenn out_o und out_r mehr als einen Obertyp besitzen und die Schnittmenge ihrer Obertypen nicht leer ist.[5]
- `fail`, wenn out_o und out_r zwar Untertyp eines gemeinsamen Obertyps sind, selbst jedoch nicht in einer Ober-/Untertyp-Beziehung stehen.

Der Ansatz von Michael Jaeger et al. der **TU Berlin** [67] geht einen Schritt weiter und berücksichtigt auch, ob die Eigenschaften input und output in Form von Untereigenschaften instanziiert wurden. Es werden drei Fälle unterschieden: `equivalent`, wenn die Eigenschaftstypen gleich sind, `subproperty`, wenn die Eigenschaftstypen in einer echten Untereigenschaftsbeziehung stehen, sowie `fail`, falls sie in keiner Untereigenschaftsbeziehung stehen. Daneben existieren drei Fälle für die Konzepte: `equivalent`, wenn die Typen exakt gleich sind, `subsumes`, wenn sie in einer echten Unterklassenbeziehung stehen, und `fail`, falls sie in keiner Unterklassenbeziehung stehen. Insgesamt entstehen somit 9 Kombinationen, die folgendermaßen bewertet werden: Im Falle von `(fail,*)` bzw. `(*,fail)` kann mindestens eine gewünschte Ausgabe nicht vom angebotenen Dienst geliefert werden. Es wird daher ein Vergleichsrang von 0 zugewiesen. In den restlichen Fällen sind alle gewünschten Ausgaben vorhanden. Für `(subproperty,subsumes)` wird 1 zugewiesen, `(subproperty,equivalent)` erhält 2, `(equivalent,subsumes)` erhält 3 und `(equivalent,equivalent)` den höchsten Wert 4. Eingaben werden in ähnlicher Weise überprüft.

Insgesamt haben die Ansätze dieser Kategorie viele schwer wiegende Nachteile, die einen Einsatz in realen Umgebungen erschweren:

[5] Die Ähnlichkeitsstufe `overlap` kann daher nur im Falle von Mehrfachvererbung auftreten.

- Der Vergleich zieht von den funktionalen Attributen nur Ein- und Ausgaben in Betracht. Vorbedingungen und Ergebnisse, die ein wesentlicher Bestandteil der funktionalen Semantik von OWL-S sind, werden nicht berücksichtigt. Essenzielle Informationen aus dem Dienstmodell, die besagen, wie der Zusammenhang zwischen Ein- und Ausgaben ist, werden ebenfalls nicht berücksichtigt.
- Der Vergleich findet nur auf dem Typ der Ein- und Ausgabeparameter statt. Im Falle komplexer Typen mit Attributen (wie beispielsweise Person mit Namen, Alter und Wohnsitz) werden diese nicht rekursiv weiter verglichen, sodass der eigentliche semantische Gehalt oft nicht ausgewertet wird.
- Die Ähnlichkeitsstufen scheinen willkürlich festgelegt und sind fest im Vergleicher verankert. Zudem verlangen sie im Falle des Semantic Matchmakers eine unübliche Unterscheidung in direkte und nicht-direkte Unterklassenbeziehung für `exact-` und `plugIn`-Ähnlichkeit oder die unübliche Verwendung von Untereigenschaften im zuletzt vorgestellten Algorithmus.
- Das Angeben von Eingaben in einer Anfragebeschreibung ist in der Regel nicht sinnvoll, da erst nach dem Auffinden eines passenden Dienstes bekannt ist, welche Eingaben er genau zur korrekten Ausführung benötigt. Daher wird sich der Vergleich häufig nur auf den Typ der Ausgabeparameter stützen.

Beschreibungslogikbasierte Ein-/Ausgabe-Vergleicher

Beschreibungslogikbasierte Ein-/Ausgabe-Vergleicher arbeiten ähnlich wie typbasierte. Sich unterscheiden sich dadurch, dass sie die Typen der Ein- und Ausgaben nicht nur auf einfache Ober- und Unterklassenbeziehungen untersuchen, sondern diese als komplexe Konzepte sehen, welche mit beschreibungslogischen Konstruktoren definiert wurden und mittels Schlussfolgerungsoperationen (siehe Abschnitt 2.2.5) überprüft werden müssen. Hierdurch werden auch eventuell vorhandene Eigenschaften der Konzepte in Betracht gezogen, sodass im Gegensatz zu typbasierten Vergleichern ein tiefer Vergleich durchgeführt wird.

Einen Vertreter dieses Typs stellt der Vergleicher von **Li** und **Horrocks** dar [94, 95].[6] Hier werden nicht nur die Ein- und Ausgaben als beschreibungslogische Konzepte aufgefasst, sondern die Dienstbeschreibung insgesamt stellt ein solches Konzept dar (im Unterschied zur Spezifikation von OWL-S, wo eine Dienstbeschreibung durch eine *Instanz* des Konzepts Service darstellt wird). Ein Vergleich findet dann statt, indem mittels Schlussfolgerungsoperation überprüft wird, in welchem Verhältnis das Angebotskonzept o zum Anfragekonzept r steht. Unterschieden werden 5 Passungsstufen:

- `Exact`, wenn $o \sqsubseteq r$ und $r \sqsubseteq o$, das heißt beide Konzepte gleich sind.

[6] Ein sehr ähnlicher Ansatz wird (jedoch wesentlich später) von Guo et al. in [56] vorgestellt.

- `PlugIn`, wenn $r \sqsubseteq o$, das heißt, das Angebot mehr beinhaltet als in der Anfrage gewünscht wird.
- `Subsume`, wenn $o \sqsubseteq r$, das heißt, das Angebot nur einen Teil dessen beinhaltet, was in der Anfrage gewünscht wird.
- `Intersection`, wenn $o \sqcap r \neq \bot$, das heißt, es eine nicht-leere Schnittmenge zwischen dem gibt, was im Angebot gefordert und in der Anfrage gewünscht wird.
- `Disjoint`, wenn $o \sqcap r \equiv \bot$, das heißt, wenn es keine Überschneidung zwischen Angebot und Anfrage gibt.

Problematisch ist bei diesem Ansatz, dass Ein- und Ausgaben gleich behandelt werden, obwohl sie ähnlich wie im Ansatz der typbasierten Vergleicher differenziert werden müssen: Betrachtet man ein Angebot, so sollten *mehr* lieferbar Ausgaben und *weniger* verlangte Eingaben zu einem positiven Vergleichsergebnis führen. Dies führt dazu, dass Dienste in der Regel nur durch ihre Ausgaben beschrieben werden. Da zusätzlich weder Informationen zu precondition oder result berücksichtigt werden, geht der Ansatz von der Annahme aus, dass die Ausgabe das liefert, was der Dienstnehmer durch die Dienstausführung besitzen wird, d.h. der Vergleicher ist auf E-Commerce-Dienste spezialisiert. Allgemeine Dienste kann er nicht korrekt vergleichen.

Di Noia et al. verfahren in [109] auf eine ähnliche Art. Sie unterteilen den Vergleichsvorgang jedoch in zwei Stufen: In der ersten Stufe bestimmen sie zu einer Dienstanfrage r potenziell passende Dienstangebote o, in dem sie überprüfen, ob $r \sqcap o$ erfüllbar ist. Wenn ja, überprüfen sie für o in der zweiten Stufe, in wie weit o im Bezug auf die Subsumption von r abweicht. Das Optimum stellt dabei die exakte Gleichheit dar, die mit 0 bewertet wird. Jede Abweichung davon wird mit einem Strafwert belegt, der standardmäßig 1 ist, aber auch durch persönliche Präferenzen angepasst werden kann. Vergleichswerte > 0 bezeichnen daher suboptimale Lösungen. Die Vergleichswerte der potenziell passenden Dienstangebote können dann zur Sortierung verwendet werden.

Experimente haben gezeigt, dass der Vergleicher zwar an intuitive, durch Menschen zuvor bestimmte Vergleichswerte heranreicht. Allerdings berücksichtigt der Vergleicher nur Ausgaben. Eingaben, Vor- und Nachbedingungen werden nicht einbezogen. Damit schränkt der Vergleicher sein Anwendungsgebiet auf einfache Dienste ein, die genau einen vorher festgelegten Effekt erwirken können und nicht konfigurierbar sind.

Modell-Vergleicher

Modell-Vergleicher berücksichtigen neben dem Dienstprofil auch Informationen über die Choreographie des Dienstes, welche im Dienstmodell zu finden ist. Einen solchen

Vergleicher stellen **Bansal** und **Vidal** in [10] vor. Die Prozesse des Dienstmodells einer Angebotsbeschreibung fassen sie als Baum auf, wobei die inneren Knoten die Kontrollflussoperationen wie Split oder Sequence darstellen, während die Blätter die atomaren Prozesse repräsentieren. Nur die atomaren Prozesse haben Ein- und Ausgaben. Ein benötigter Dienst wird hingegen nur über eine Menge abgebbarer Eingaben und gewünschter Ausgaben abgebildet. Ziel des Algorithmus ist es, aus dem Baum eine Menge von Blättern auszuwählen, sodass diese (1) über die Kontrollflussoperationen erreichbar sind, (2) die Ausgaben der Anfrage vollständig überdecken und (3) die abgebbaren Eingaben nicht übersteigen. Die Überprüfung, ob Ein- und Ausgaben zueinander passen, erfolgt wie bei den oben vorgestellten Algorithmen über einfache Typvergleiche. Da der Algorithmus im schlechtesten Fall alle Kombinationen von Überdeckungen überprüfen muss, hat er einen exponentiellen Aufwand.

Der Ansatz bezieht zwar explizit auch das Dienstmodell in den Vergleich mit ein, durch das einfache Vorgehen bei atomaren Prozessen bleiben die Probleme der oben vorgestellten Verfahren jedoch bestehen.

Weitere Ansätze

Daneben existiert eine Reihe von Ansätzen, die keine strengen Kriterien für eine Passung von Dienstbeschreibungen vorgeben und so die Vergleichsergebnisse in diskrete Kategorien einteilen, sondern für den Vergleich weitere Informationen heranziehen, was zu einem kontinuierlichen Vergleichsergebnis aus dem Intervall $[0, 1]$ führt:

- **Younas** et al. versuchen in [157] zu einem intuitiveren Vergleichsergebnis zu gelangen, indem sie unscharfe Logik einsetzen. Dazu verwenden sie einen Algorithmus, der vor dem Vergleich scharfe Werte innerhalb der Anfragebeschreibung durch linguistische Variablen ersetzt und so unscharf werden lässt. Ein konkrete Preisangabe wird so beispielsweise in einen *angemessenen Preis* umgewandelt werden, der Abweichungen nach oben und unten zulässt. Für den Vergleich werden solche linguistischen Variablen von einer zusätzlichen Komponente der Schließers ausgewertet. Das Einbringen von Unschärfe in die Beschreibung ist zwar ein sinnvolles Vorgehen, um zu einem differenzierteren Ergebnis zu gelangen, allerdings muss dies vom Benutzer selbst ausgehen. Für diesen ist sonst unklar, welche Werte wie angepasst wurden, sodass er das entstandene Vergleichsergebnis zunächst überprüfen will, bevor es zur Weiterverarbeitung verwendet wird. Für eine Automatisierung der Dienstnutzung ist das Verfahren daher nicht geeignet.

- **Klusch** et al. gehen mit *OWL-MX* in [86] einen anderen Weg. Ziel ist es, neben der expliziten Semantik einer Dienstbeschreibung, welche durch die zugrunde

liegende Logiksprache und gemeinsame Ontologie gegeben ist, auch deren implizite Semantik, welche zum Beispiel durch die Wahl der Worte in der Dienstbeschreibung mitbestimmt wird, einzubeziehen. In einem solchen *hybriden Vergleicher* kommen somit zusätzlich zu Schlussfolgerungsoperationen auch Techniken des Information Retrieval zum Einsatz, wodurch sich sowohl der Recall (d.h. wie viele eigentlich passende Dienste werden auch als solche erkannt) als auch die Präzision (d.h. wie viele eigentlich unpassende Dienste werden auch als solche erkannt) verbessert. Nach wie vor ist jedoch eine Präzision von nahezu 1.0, wie sie für die vollautomatische Weiterverarbeitung der Ergebnisse ohne Überprüfung durch einen menschlichen Benutzer benötigt würde, nur für sehr geringe Recall-Werte (< 0.4) möglich. Als ähnlicher Ansatz und Vorgänger kann der hybride Vergleicher für die Beschreibungssprache für Agentenfähigkeiten *LARKS* (siehe Abschnitt 3.6.2) gesehen werden.

3.2.3. Bewertung von OWL-S

OWL-S gilt als Vorreiter auf dem Gebiet der semantischen Dienstbeschreibungssprachen. Dieser Abschnitt fasst zusammen, wo die Stärken und Schwächen des Ansatzes liegen. Eine Übersicht zeigt auch die Tabelle in Abbildung 3.14 auf Seite 81. Folgende Stärken von OWL-S wurden in diesem Abschnitt festgestellt:

- Im Gegensatz zu WSDL beschreibt OWL-S Dienste nicht nur über ihren Informationsfluss, sondern ermöglicht auch die Beschreibung von Zustandsübergängen, die durch Vorbedingungen und Ergebnisse ausdrückbar sind. Auch Nebeneffekte (wie die Belastung einer Kreditkarte), die nicht zur eigentlichen Funktionalität des Dienstes gehören, können beschrieben werden.
- Seit Version 1.1 ist es in OWL-S im Unterschied zu WSDL möglich, die Beziehung der IOPRs untereinander durch SWRL-Regeln auszudrücken. Hierdurch soll der Einfluss der Ein- und Ausgaben auf die Vorbedingungen und Ergebnisse klarer werden.
- Durch die Möglichkeit eines WSDL-Dienstfundaments ist OWL-S vollständig kompatibel zu WSDL, d.h. jeder mittels WSDL beschriebene Webdienst kann direkt auch mit OWL-S beschrieben werden.
- OWL-S hat die Anfänge der Forschung um semantische Dienstbeschreibungssprachen entscheidend geprägt, hat noch heute eine große Forschungsgemeinschaft und lässt sich mit einer Vielzahl von Werkzeugen bearbeiten.

Demgegenüber steht eine Vielzahl von Problemen und Schwächen:

- Zunächst fehlt in OWL-S ein einheitliches konzeptionelles Modell. Deutlich wird das daran, dass mehrere unterschiedliche Definitionen für den zentralen Begriff *Dienst* existieren (vgl. dazu auch [102]). A1 (der Anforderungen in Tabelle in Abbildung 3.1 auf Seite 40) ist demnach nicht erfüllt.

- OWL-S beschreibt angebotene und benötigte Dienste auf die gleiche Art und Weise. Der Dienstnehmer hat dadurch nicht die Möglichkeit, seine Präferenzen bezüglich geeigneter Dienste anzugeben. A4 und A6 sind demnach nicht erfüllt.

- In OWL-S-Beschreibungen werden die Elemente Instanz, Menge und SWRL-Regel gemischt verwendet, ohne dass hierfür eine formale Semantik hinterlegt wäre. Was bedeutet es beispielsweise, dass das Attribut parameterType mit einem Konzept anstatt mit einer Instanz gefüllt ist oder SWRL-Regeln als Füllwerte von inCondition in Ergebnissen auftreten? Dieses Mischen erschwert ein formal korrektes Schlussfolgern über Dienstbeschreibungen, wie es für den Vergleich benötigt wird. Zudem sind wichtige Informationen in der Dienstbeschreibung über Profil und Modell verstreut, was zu Inkonsistenzen führen kann. A5 ist demnach nicht erfüllt.

- Für OWL ist zwar eine formale, modelltheoretische Semantik definiert, die von OWL-S als spezielle Ontologie übernommen wird, allerdings fehlt darüber hinaus eine formale, dienstspezifische Semantik für Beschreibungen in OWL-S. Was sagt es aus, dass eine Instanz vom Typ Result als Füllwert von hasResult in einem Prozess auftritt? Wo ist ersichtlich, wer im Laufe der Dienstnutzung welche Informationen liefern muss? Dies führt dazu, dass die genaue Aussage einer Dienstbeschreibung nicht definiert ist, wodurch das Aufstellen von Dienstbeschreibungen erschwert und Vergleicher auf selbstdefinierte Ähnlichkeitsmaße zurückgreifen müssen. Deutlich wird das an der großen Zahl unzureichender Ansätze für den Vergleich von OWL-S-Beschreibungen. A7 ist demnach nicht erfüllt.

- Existierende Vergleicher für OWL-S-Beschreibungen arbeiten nicht intuitiv oder erreichen durch die fehlende Berücksichtigung vieler Teile der Beschreibung eine zu geringe Präzision, um für die vollständige Automatisierung der Dienstnutzung eingesetzt werden zu können. A8 ist demnach nicht erfüllt.

- Beschreibungen in OWL-S sind überspezifiziert, etwa durch die Angabe von Eingaben und Vorbedingungen in der Anfragebeschreibung, durch Angabe von Fehlerbehandlungsroutinen, lokaler Variablen, genauer Rechenvorschriften oder nicht überprüfbarer Vorbedingungen (vgl. [9]). An anderen Stellen ist die Sprache sehr generisch, etwa indem sie kaum Bedingungen an die Beschreibung der IOPRs stellt. Das unabhängige Aufstellen passender Beschreibungen wird somit je nach Vergleichsverfahren entweder quasi unmöglich oder enthält einen

ungerechtfertigten Zusatzaufwand, da nur Teile der Beschreibung berücksichtigt werden. A10 ist somit nicht erfüllt.

- Die Einbindung von Diensten, die mittels OWL-S beschrieben wurden, in eine Applikation ist zwar möglich, setzt aber aufgrund des Dienstfundaments voraus, dass der angebotene und benötigte Dienst eine strukturell gleiche Signatur besitzen, was die Wahrscheinlichkeit einer Passung insbesondere bei einer zuvor unbekannten Dienstlandschaft extrem einschränkt (siehe dazu auch [119]). A11 ist daher nur teilweise erfüllt.

- Für OWL-S existieren nur sehr wenige Beispiele und diese haben eine Vielzahl von Schwächen. Was ist beispielsweise der Unterschied des Standardbeispiels CongoBookBuy zu anderen Buchkaufdiensten? Welche Bücher gibt es? Was kosten diese? Auch in Erfahrungsberichten zum praktischen Einsatz der Sprache werden diese Schwächen genannt (siehe [126]). Eine Evaluation der Tauglichkeit von OWL-S existiert nicht. A13 ist damit nicht erfüllt.

Insgesamt stellt OWL-S einen der ersten ernst zu nehmenden Ansätze dar, der die Probleme einer rein syntaktischen Beschreibung (wie beispielsweise durch WSDL) erkannte und durch Einbringen von Semantik und einer Reihe neuer Konzepte zu beheben versuchte. Trotz der großen Zahl an Problemen und Nachteilen hat OWL-S das Forschungsgebiet der semantischen Dienstbeschreibungen entscheidend geprägt und wichtige Ideen eingebracht, auf die spätere Ansätze (auch der Ansatz dieser Arbeit) aufbauen konnten.

3.3. WSMO

Die *Web Service Modelling Ontology* (WSMO) [124] ist die „europäische Antwort“ auf OWL-S. Ziel ist es, Verfahren bereitstellen, mit denen die semantische Dienstnutzung automatisiert werden kann. Dazu wird eine geeignete Beschreibung von Diensten in allen Aspekten angestrebt, welche auch die Integrationsproblematik berücksichtigt. Die Weiterentwicklung und Verbreitung von WSMO geschieht durch die *WSMO Working Group*, eine relativ offene Forschergruppe bestehend aus aktiven und inaktiven Mitglieder europäischer Forschungseinrichtungen. Ziel ist eine Standardisierung von WSMO. Hierzu wurde WSMO im April 2005 beim World Wide Web Consortium eingereicht.

Die drei wichtigsten Aspekte von WSMO werden in drei Arbeitsgruppen vorangetrieben:

- Aufgabe der **WSMO Working Group** ist die Bereitstellung eines abstrakten, konzeptionellen Modells zur Beschreibung von Diensten sowie die Entwicklung

der dazu nötigen Ontologien. Dieses Modell bildet die Grundlage für alle weiteren Arbeiten. Die Details zum konzeptionellen Modell von WSMO finden sich in Abschnitt 3.3.1.

- Aufgabe der **WSML Working Group** ist die Schaffung einer formalen Sprache als geeignete Repräsentation des konzeptionellen Modells. Die entstehende *Web Service Modelling Language* soll dabei einerseits an existierende Sprachen angelehnt sein, andererseits die Besonderheiten des konzeptionellen Modells erfassen können. Die Details zu WSML finden sich in Abschnitt 3.3.2.
- Aufgabe der **WSMX Working Group** ist die Schaffung einer Ausführungsumgebung für Dienste, die gemäß des konzeptionellen Modells von WSMO erstellt sind. Das entstehende *Web Service Modelling Execution Environment* muss dabei insbesondere auch die Algorithmen des konzeptionellen Modells (etwa zum Vergleich von Beschreibungen oder zur Anpassung von Ontologien) implementieren. Die Details zu WSMX finden sich in Abschnitt 3.3.3.

3.3.1. Konzeptionelles Modell von WSMO

Das konzeptionelle Modell von WSMO ist ein Meta-Modell, in welchem die wichtigsten Komponenten zur Automatisierung der Dienstnutzung bereits formal festgehalten sind [124]. Entstanden ist es aus dem 2002 von Fensel und Bussler publizierten *Web Service Modelling Framework* (WSMF) [37]. Ihm liegen unter anderem folgende Entwurfsprinzipien zugrunde:

- Alle WSMO-Beschreibungen sind *ontologiebasiert*, d.h. stützen sich auf das geteilte Verständnis der Anwendergruppe sowie die formale Semantik der zugrunde liegenden Sprache.
- WSMO-Beschreibungen werden *strikt voneinander entkoppelt*, d.h. unabhängig von ihrer Verwendung erstellt, um so der Arbeitsweise des Internets gerecht zu werden.
- Trotz der Heterogenität bei der Beschreibung von Ressourcen soll eine Kommunikation zwischen allen Teilnehmern möglich sein, was durch *explizite Mediation* erreicht werden soll.
- Angebotene und benötigte Dienste unterscheiden sich grundlegend, was durch *getrennte ontologische Rollen* zum Ausdruck gebracht werden soll.
- Vorrangiges Ziel von WSMO ist die Versorgung von Teilnehmern mit *Diensten*, welche anschließend (außerhalb der WSMO-Umgebung) verwendet werden können, um bestimmte Zustände zu erwirken.

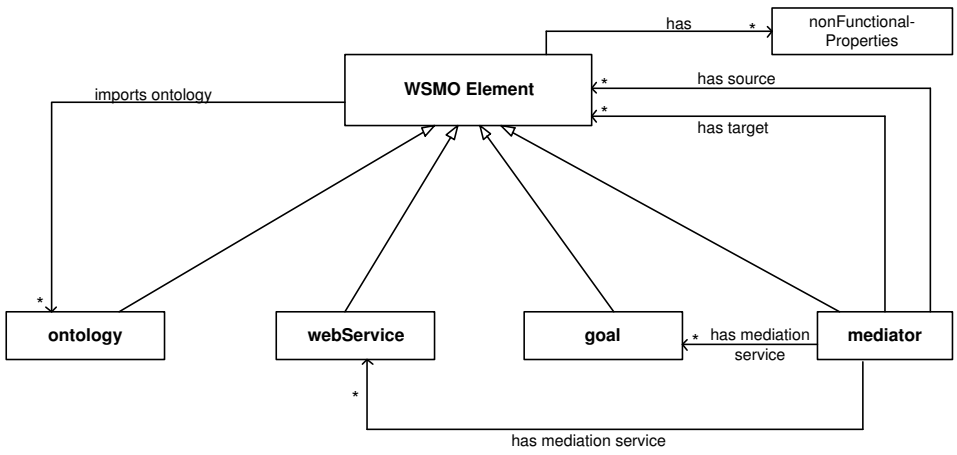


Abbildung 3.9.: Die vier Hauptelemente des konzeptionellen Modells von WSMO nach [124].

Das hieraus entwickelte konzeptionelle Modell von WSMO ist als Meta-Modell in MOF[7] spezifiziert. Es basiert auf vier Hauptelementen, wie die UML-Repräsentation in Abbildung 3.9 zeigt:

- *WSMO Element*. Stellt die abstrakte Oberklasse aller vier Hauptelemente dar. Jedes Hauptelement kann durch einen Satz nicht-funktionaler Attribute beschrieben sein, die im Wesentlichen den Attributen des Dublin Core Standards[8] ergänzt um ein Versionsattribut entsprechen. Jedes Hauptelement kann zu seiner Beschreibung Ontologien importieren. Die sprachliche Grundlage zur Erfassung der WSMO-Hauptelemente ist zunächst offen gelassen. Als Referenzimplementierung schlägt WSMO die Verwendung von WSML vor.

- *Ontologien* (engl. ontology). Ontologien liefern ein formal spezifiziertes Vokabular, das von allen anderen Komponenten verwendet wird. Sie stellen somit quasi das Datenmodell von WSMO dar, da alle Beschreibungen von WSMO ontologiebasiert sein müssen. Um Skalierbarkeit zu gewährleisten, können die Ontologien der einzelnen Teilnehmer unabhängig voneinander entwickelt werden (Entkopplung) und durch Importierung gegenseitig verwendet werden (Modularisierung). Ontologien in WSMO bestehen aus Konzepten, Attributen, Funktionen, Relationen, Instanzen sowie Axiomen in Form logischer Ausdrücke.

- *Ziele* (engl. goal). WSMO sieht anders als OWL-S eine explizite Trennung der Beschreibung von angebotenen und benötigten Diensten vor. Für benötigte Dienste wird ein *zielorientierter* Ansatz verfolgt, bei dem der Wunsch des

[7]MOF = Meta-Object Facility, `http://www.omg.org/technology/documents/formal/mof.htm`
[8]`http://dublincore.org/`

Dienstnehmers als Ziel formuliert wird, unabhängig davon, wie dieses durch einen konkreten Dienst ausgeführt werden könnte. Aufgabe der Ausführungsumgebung ist es dann, dieses Ziel durch Nutzung eines oder mehrerer Dienste zu erfüllen. In Zielen werden zwei Arten von Forderungen unterschieden: In den *Nachbedingungen* (engl. postconditions) ist erfasst, wie der Informationsraum nach einer Dienstnutzung aussehen soll, in den *Effekten* (engl. effects) ist erfasst, welche Zustände in der realen Welt erfüllt sein sollen.

- *Webdienste* (engl. web service). Angebotene Dienste werden mittels dieses Hauptelements von WSMO erfasst. Die Beschreibung unterteilt sich dabei in den funktionalen Aspekt der *Fähigkeiten* (engl. capabilities), welcher die möglichen Wirkungen des Dienstes erfasst, sowie den Aspekt der *Schnittstelle* (engl. interface), welcher die Zugriffsmöglichkeiten auf den Dienst darstellt (siehe Abbildung 3.10). Die Fähigkeiten werden durch Angabe von *Vorbedingungen* (engl. preconditions), *Annahmen* (engl. assumptions), Nachbedingungen (engl. postcondition) und Effekte (engl. effect) beschrieben (die zusammengefasst als APPE bezeichnet werden). Die Nachbedingungen und Effekte drücken wie bei den Zielen den Zustand des Informationsraumes bzw. der realen Welt nach Nutzung des Dienstes aus. Zusätzlich werden durch die Vorbedingungen Restriktionen an die Eingaben definiert und durch die Annahmen Bedingungen an den Zustand der realen Welt vor der Dienstausführung geknüpft. Durch die Verwendung gemeinsamer Variablen, die in der Fähigkeit deklariert werden, können die APPE untereinander in Beziehung gesetzt werden. Die Schnittstelle umfasst die Beschreibung der *Choreographie*, in der festgelegt ist, wie der Dienstnehmer mit dem Dienst zu kommunizieren hat, um zu den angegebenen Effekten und Nachbedingungen zu gelangen, sowie der Beschreibung der *Orchestration*, die den Zugriff des Dienstes auf Subdienstgeber erfasst. Die genaue Umsetzung dieser Konzepte erfolgt in WSML und wird in Abschnitt 3.3.2 genauer betrachtet.

- *Mediatoren* (engl. mediator). Bei der Nutzung von Webdiensten kann es zwischen Komponenten, die interagieren müssen, zu Heterogenität auf Daten-, Protokoll- und Prozessebene kommen, insbesondere dann, wenn diese in einer offenen Umgebungen stattfindet. Mit dem Hauptelement der Mediatoren besteht in WSMO ein Konzept, um explizit mit solchen Ungleichheiten umgehen zu können. Mediatoren dienen als Konnektoren zwischen den einzelnen Komponenten, die zunächst nicht zueinander passende Elemente so miteinander verbinden, dass die Heterogenität beherrschbar wird. In der Tat kommunizieren in WSMO zwei Komponenten nie ohne zwischengeschalteten Mediator. Man unterscheidet Ontologie-Ontologie-Mediatoren (OO-Mediatoren), welche die Datenintegration vornehmen, Ziel-Ziel-Mediatoren (GG-Mediatoren), mit deren Hilfe Einzelziele zu komplexeren Zielen verbunden und spezialisiert werden können, was eine Wiederverwendung von Zielen ermöglicht, Webdienst-Webdienst-Mediatoren (WW-Mediatoren), welche Prozess und Protokollintegrationen vornehmen, um

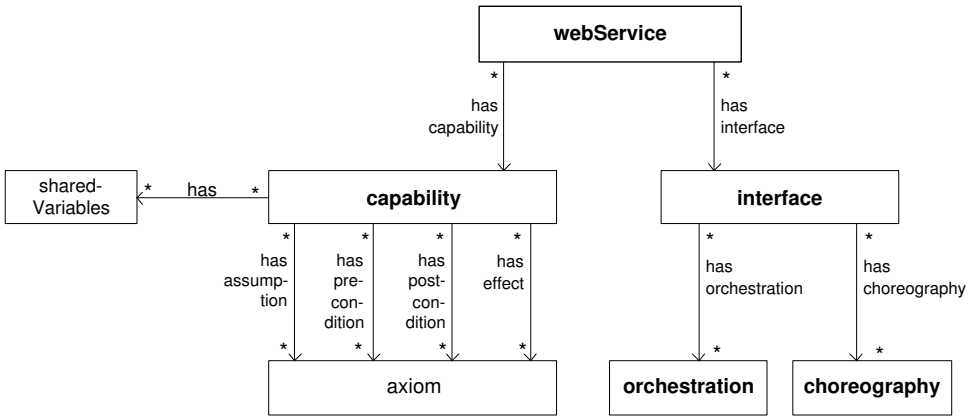


Abbildung 3.10.: Der Aufbau des WSMO-Hauptelement *web service* nach [124].

so mehrere Webdienste für eine Zusammenarbeit zusammenschalten zu können, sowie Webdienst-Ziel-Mediatoren (WG-Mediatoren), um einen semantisch passenden Webdienst selbst dann mit einem Ziel zu verbinden, wenn deren Schnittstellen, Protokolle und ausgetauschte Daten nicht direkt zueinander passen. Die Erstellung von Mediatoren ist jedoch nicht vollständig automatisierbar und erfordert in vielen Fällen Integrationsentscheidungen von einem menschlichen Benutzer.

3.3.2. Grundlegende Sprache: WSML

Ziel der *Web Service Modelling Language* (WSML) [23] ist die Bereitstellung einer Familie formaler Sprachen, in welcher einerseits das konzeptionelle Modell von WSMO optimal repräsentiert werden kann und für die andererseits eine kompakte, menschenlesbare Syntax und eine wohldefinierte Semantik vorhanden ist. WSML definiert daher Konstrukte, um die Hauptelemente von WSMO (also Ontologien, Webdienste, Ziele und Mediatoren) direkt notieren zu können. Dabei liegt WSML in verschiedenen Varianten vor, die sich aufgrund der zur Verfügung stehenden Konstrukte in Ausdrucksstärke und Effizienz der Berechenbarkeit unterscheiden. Am häufigsten wird *WSML-Rule* verwendet, die als Vereinigung aus F-Logic und HiLog angesehen werden kann. Generell ist die Syntax eine Mischung aus rahmenbasierten Ansätzen und logischen Formeln. Es existieren daher sowohl Konstrukte zur Datenmodellierung wie Klassen, Attribute, Relationen, Instanzen etc. als auch Konstrukte zur Definition von Axiomen wie Prädikate und logische Operatoren. Verbunden werden diese durch *Moleküle*. Diese speziellen Prädikate wie `memberOf` oder `subclassOf` machen Aussagen über Beziehungen im Datenmodell. WSML unterscheidet eine konzeptuelle Syntax (KS) und eine Syntax durch logische Formeln (FS). Beide sind teilweise ineinander

```
concept Human
  nonFunctionalProperties
    dc#description hasValue "concept of a human being"
  endNonFunctionalProperties
  hasName ofType foaf#name
  hasParent inverseOf(hasChild) impliesType Human
  hasChild impliesType Human

axiom humanDefinition
   definedBy
     ?x memberOf Human equivalent
     ?x memberOf Animal and
     ?x memberOf LegalAgent.
```

Abbildung 3.11.: Beispiel für eine Konzept- und eine Axiomdefinition in WSML, entnommen aus [23].

überführbar. Zum Austausch mit anderen Gruppen existieren zudem Ausgabesyntaxen in Form von XML, RDF oder OWL.

Ontologien in WSML

Ontologien werden meist in KS notiert und um einzelne Axiome in FS erweitert. Abbildung 3.11 zeigt ein Beispiel. Hier wird das Konzept Human definiert. Menschen haben einen Namen, eine Relation hasParent zu ihren Eltern und die dazu inverse Beziehung hasChild.[9] Das Axiom in FS drückt aus, dass Menschen als Schnittmenge von Lebewesen und handelnden Entitäten angesehen werden.

Webdienste und Ziele in WSML

Die Beschreibung von Webdiensten und Zielen erfordert die Beschreibung von Fähigkeiten und Schnittstellen. Dazu existieren in WSML die Konstrukte `capability` und `interface`, welche auf logische Formeln in FS verweisen. Abbildung 3.12 zeigt ein Beispiel für die Beschreibung einer *Fähigkeit* eines Registrierdienstes. Die Vorbedingung

[9] Die Unterscheidung in `ofType` und `impliesType` ist diffizil und kommt nur dann zum Tragen, wenn die Typzugehörigkeit(en) einer Instanz nicht vollständig bekannt ist. Im Falle von `A ofType B` führt ein Füllen mit einer Instanz `i`, von der nicht bekannt ist, ob sie vom Typ `B` ist, zu einer Verletzung der Konsistenz (also zu einer Fehlermeldung). Im Falle von `impliesType` führt dies dazu, dass für `i` zusätzlich eine Typzugehörigkeit zu `B` angenommen wird.

```
capability
   sharedVariables ?child
   precondition
       nonFunctionalProperties
               dc#description hasValue "The input has to be
                   boy or a girl with birthdate in the past
                   and be born in Germany."
       endNonFunctionalProperties
       definedBy
           ?child memberOf Child
                   and ?child[hasBirthdate hasValue ?birthdate]
                   and ?child[hasBirthplace hasValue ?location]
                   and ?location[locatedIn hasValue oo#de]
                   or (?child[hasParent hasValue ?parent] and
                       ?parent[hasCitizenship hasValue oo#de]).

   effect
           nonFunctionalProperties
               dc#description hasValue "After the registration
                       the child is a German citizen"
           endNonFunctionalProperties
           definedBy
               ?child memberOf Child
                   and ?child[hasCitizenship hasValue oo#de].
```

Abbildung 3.12.: Beispiel für die Beschreibung einer Fähigkeit in WSML, entnommen aus [23].

(ausgedrückt durch `precondition`) ist ein Kind, das in Deutschland geboren wurde oder das deutsche Eltern hat. Als Effekt erhält das Kind die deutsche Staatsbürgerschaft. Vorbedingung und Effekt sind über die gemeinsame Variable `?child`, welche zu Beginn der Fähigkeit deklariert wurde, miteinander verbunden. Allgemein stellen die APPEs also Axiome dar, die gemeinsame Variablen enthalten können.

Die Festlegung, wie **Schnittstellen** in WSML beschrieben werden, ist noch nicht endgültig abgeschlossen und liegt zum Zeitpunkt dieser Arbeit erst als vager Entwurf vor [130]. Die Beschreibung der *Choreographie* soll über Zustandsautomaten erfolgen, wobei festgelegt wird, welche Operationen des Dienstes zu welchen Zustandsübergängen führen. Zustände des Automaten werden dabei als WSMO-Instanzen definiert, Zustandsübergänge durch `if-then`-Konstrukte, deren `then`-Teil eine

Zustandsinstanzerzeugung, -änderung oder -löschung bewirkt. Insgesamt ist die Spezifikation nicht stimmig, da sie sich nicht in Beschreibungen der Fähigkeiten integrieren lässt. Zur Beschreibung der *Orchestrierung* existieren noch keine konkreten Ansätze, sondern nur erste Ideen.

Mediatoren in WSML

Für die Beschreibung von Mediatoren in WSML existiert das sprachliche Konstrukt `mediator`, welches in der vorliegenden Spezifikation nur einen Verweis auf einen extern definierten Mediator zulässt und keine inhaltliche Beschreibung erlaubt. Als Grund wird die Schwierigkeit einer solchen Mediatorbeschreibung angegeben, was in [68] zum Ausdruck kommt. Insgesamt ist dieses Vorgehen sehr fragwürdig, da WSMO Mediatoren in seinen Entwurfsprinzipien explizit erwähnt und zu einem der vier Hauptelemente ernennt. Die Mediation ist in WSMO daher nur theoretisch besser integriert als in anderen Sprachen, praktisch muss sie auch hier von externen Komponenten erledigt werden.

Semantik von WSML

Die Semantik von WSML wird in [23], Kapitel 8 definiert. Zurzeit geschieht dies jedoch nur für die traditionellen Ontologieelemente von WSML, wie etwa Konzept, Instanz, Relation usw. Gerade für die interessanten weiteren Hauptelemente von WSMO wie insbesondere Fähigkeiten, Schnittstellen und Mediatoren liegt (noch) keinerlei Semantikdefinitionen vor. WSML muss daher als gewöhnliche Ontologiebeschreibungssprache angesehen werden, welche Hauptelemente nur mit Schlüsselworten markiert, deren Semantik aber nicht formal erfasst und somit für verarbeitende Einheiten offen lässt. Es muss sich daher der Kritik aussetzen, keinen Neuigkeitswert für den Bereich der Dienstbeschreibung zu bieten und somit durch klassische Ontologiesprachen wie etwa OWL+SWRL austauschbar zu sein.

3.3.3. Die Ausführungsumgebung: WSMX

Das *Web Service Modelling Execution Environment* (WSMX) [58] kann als Referenzimplementierung des konzeptionellen Modells von WSMO angesehen werden. Es soll zeigen, ob sich der Ansatz von WSMO tatsächlich für die Automatisierung der semantischen Dienstnutzung eignet.

Die vorliegende Implementierung von WSMX stellt ein System zur Verfügung, an das sich Dienstgeber und Dienstnehmer dynamisch (eventuell über Adapter) anschließen können, um so Dienste anbieten oder nutzen zu können. Dienstgeber und -nehmer

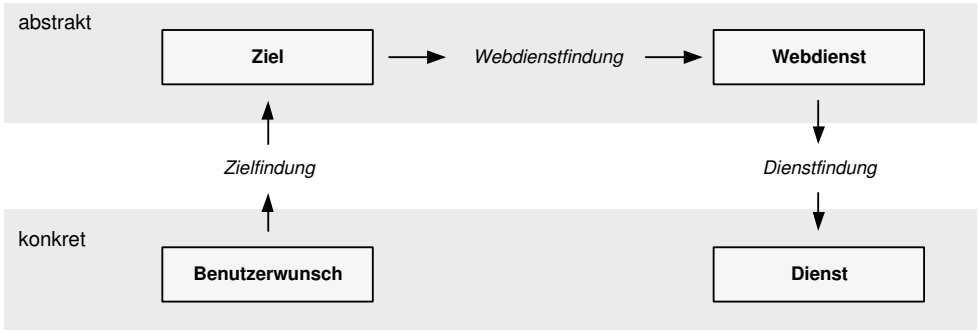


Abbildung 3.13.: Ablauf und Begrifflichkeiten der Dienstfindung in WSMO.

bleiben so entkoppelt, da sie nicht direkt miteinander in Kontakt treten. Passende Paare von Dienstgebern und -nehmern werden im Rahmen der *WSMX Discovery* ausfindig gemacht, welche den Vergleich von Webdiensten und Zielen durchführt. Zurzeit findet hier jedoch nur ein einfacher Stringvergleich statt. Ausgefeiltere Algorithmen, wie sie im nächsten Abschnitt vorgestellt werden, die auch die Bedeutung der Beschreibungen in Betracht ziehen, sollen zukünftig integriert werden. Auch die nötige Mediation ist nur rudimentär integriert. Die Datenintegration im Rahmen der OO-Mediation erfolgt durch Auswertung konkreter Abbildungsvorschriften zwischen den zu vereinheitlichenden Ontologien. Für die Mediation von Prozessen ist vorgesehen, die ausgetauschten Nachrichten mittels einer gegebenen Abbildung anzupassen. Der innere Ablauf der Ausführungsumgebung ist dabei durch eine Reihe von Petrinetzen spezifiziert und somit für die Teilnehmer offen gelegt. Die WSMX-Umgebung kann über zwei Zugangspunkt genutzt werden (siehe dazu [150]): Die Einwegausführung soll nach Absenden eines Ziels ohne Rückfrage einmalig einen Effekt erwirken, die Webdienstfindung lieft einem Dienstnehmer nach Absenden eines Zieles die Daten für den Zugriff auf einen Webdienst, den er auch mehrmals ausführen kann.

3.3.4. Vergleich von Beschreibungen in WSMO

Der Vergleich von Dienstbeschreibungen wird in WSMO in zwei Phasen unterteilt [69]: in der *Findungsphase* (engl. discovery phase) werden Dienstbeschreibungen miteinander verglichen und festgestellt, ob ein Webdienst generell in der Lage ist, die gegebenen Ziele zu erfüllen. In der anschließenden *Verhandlungsphase* (engl. contracting phase) werden dann für ein ausgewähltes Dienstangebot konkrete Parameter ermittelt, die zum gewünschten Ergebnis führen. WSMO konzentriert sich im Wesentlichen auf die erste Phase und verschiebt die zweite Phase in direkte, von der Ausführungsumgebung unabhängige Verhandlungen zwischen Dienstnehmer und Dienstgeber.

Zur Nutzung eines Dienstes ist also ein Ablauf wie in Abbildung 3.13 dargestellt nötig. Am Anfang steht ein konkreter *Benutzerwunsch*. Ein Beispiel für einen sol-

chen Benutzerwunsch könnte sein, ein Telefon eines bestimmten Modells zu besitzen. Dieser konkrete und detaillierte Wunsch muss dann zunächst im Rahmen der *Zielfindung* zu einem allgemeineren *Ziel* abstrahiert werden, im Beispiel also zu dem Ziel, ein Telefon zu erwerben. In einem zweiten Schritt wird dann dieses Ziel im Rahmen der *Webdienstfindung* mit den veröffentlichten Beschreibungen der Webdienste verglichen. Zum Aufruf des ausgewählten geeigneten Dienstgebers müssen die vorherigen Abstraktionen wieder rückgängig gemacht werden, was im dritten Schritt, der *Dienstfindung* geschieht. Im Beispiel muss der *Webdienst* daraufhin überprüft werden, ob er den *Dienst* bereitstellt, mit dem das konkret gewünschte Telefon bezogen werden kann.

Zurzeit wird in WSMO nur der zweite Schritt unterstützt, bei dem allgemeine Dienstbeschreibungen miteinander verglichen werden. Es bleibt daher auch bei einem positiven Vergleichsergebnis unsicher, ob das gefundene Ergebnis zum konkreten Wunsch des Dienstnehmers passend ist. Der Vergleich in WSMO muss daher eher als *grober Vorvergleich* angesehen werden, der durch einen in WSMO nicht näher betrachteten Feinvergleich unterstützt werden muss. In [70] werden drei Möglichkeiten vorgestellt, wie ein Feinvergleich im Rahmen der Dienstfindung aussehen könnte: erstens durch eine testweise Ausführung des Dienstes, zweitens durch eine Überprüfung der Durchführbarkeit anhand der Dienstbeschreibung oder drittens durch Bezug von weiteren Informationen über die an der Dienstausführung konkret beteiligten Instanzen. Für WSMO lehnen die Autoren diese Vorgehensweisen jedoch als zu applikationsspezifisch ab und konzentrieren sich auf den Webdienstvergleich.

Zum vom WSMO betrachteten zweiten Schritt, dem Webdienstvergleich, sind drei generelle Vorgehensweisen denkbar:

- *Schlüsselwortbasierter Ansatz.* Die Beschreibungen werden als Text von Schlüsselworten aufgefasst, die mittels Techniken des Textvergleichs verglichen werden. Typischerweise wird dabei untersucht, ob wichtige Schlüsselworte der Anfrage auch im Angebot vorkommen. Der volle semantische Gehalt der Beschreibungen wird dabei nicht berücksichtigt, was zu sehr ungenauen Vergleichsergebnissen führt (sowohl Präzision als auch Recall sind wesentlich kleiner als 1). Der Ansatz kann daher bestenfalls als vorläufige Zwischenlösung angesehen werden.

- *Mengenbasierter Ansatz.* Ziel und Webdienst werden als Mengen von Objekten angesehen, die um eine *Intention* in Form eines der beiden Quantoren $\exists$ oder $\forall$ ergänzt werden. Aus den unterschiedlichen Möglichkeiten zur Überlappung zwischen diesen Mengen können die verschiedenen Passungsstufen abgeleitet werden. Eine Vorgehensweise zur Erfassung und Verarbeitung der Mengen bietet die Beschreibungslogik.

- *Ableitungsbasierter Ansatz.* Die Fähigkeiten von Ziel und Webdienst werden als logische Formeln angesehen. Ein Webdienst passt dann zu einem Dienst, wenn

(1) aus den Effekten und Nachbedingungen des Webdienstes die gewünschten Effekte und Nachbedingungen des Ziels logisch gefolgert werden können und (2) aus der Erfüllung der Eingaben des Webdienstes die Erfüllung seiner Annahmen abgeleitet werden kann. Als Logiksprache wird vornehmlich F-Logic (mit Erweiterungen) verwendet.

Ein generelles Problem bei allen drei Ansätzen besteht darin, dass das Ziel zwar den Wunsch des Dienstnehmers erfasst, nicht jedoch seine Präferenzen im Falle eines mehr oder weniger stark abweichenden Webdienstes enthält. Der Vergleich muss daher in solchen Fällen Heuristiken anwenden, was zu verzerrten, vom Dienstnehmer häufig unbeabsichtigten Vergleichsergebnissen führt. Ein vollständig automatischer Ablauf wird so in der Regel nicht akzeptiert.

Im Folgenden werden Beispiele für die beiden letzten, semantischen Verfahren vorgestellt.

Mengenbasierte Ansätze

Im mengenbasierten Ansatz (siehe [69]) für den Vergleich von WSMO-Beschreibungen werden sowohl Ziel als auch Webdienst als Menge von erbringbaren Effekten aufgefasst. Dargestellt wird unabhängig von gegebenen Ein- oder Ausgaben, welche Effekte generell erbringbar bzw. gewünscht sind. Zusätzlich muss die *Intention* des Teilnehmers mit dieser Beschreibung festgehalten werden: Möchte bzw. liefert er *alle* Effekte oder möchte bzw. liefert er *nur einen einzelnen*. Die Intention wird in Form eines Quantors festgehalten. Ein Webdienst W kann die Wünsche eines Ziels G erfüllen, wenn die zugehörigen Mengen R_W und R_G und die Intentionen I_W und I_R gleich sind. Ansonsten entscheidet der Grad der Überlappung zwischen R_W und R_G den Grad der Passung. Folgende Beziehungen zwischen R_W und R_G sind denkbar:

- $R_W = R_G$, d.h. die Mengen sind gleich.
- $R_W \subseteq R_G$, d.h. der Webdienst ist eine Teilmenge des Ziels.
- $R_W \supseteq R_G$, d.h. der Webdienst ist eine Obermenge des Ziels.
- $R_W \cap R_G \neq \emptyset$, d.h. es gibt gemeinsame Elemente zwischen Webdienst und Ziel.
- $R_W \cap R_G = \emptyset$, d.h. es gibt keine gemeinsamen Elemente zwischen Webdienst und Ziel.

Aus der Beziehung zwischen R_W und R_G und den angegebenen Interpretationen I_W und I_G lässt sich auf intuitive Weise der Grad der Passung zwischen W und G ableiten. Es werden drei Stufen unterschieden:

- *Passend* (engl. match). Der Webdienst kann das gewünschte Ziel vollständig erfüllen (und evtl. sogar noch mehr bieten). Diese Stufe gilt beispielsweise für $I_W = \forall$, $I_G = \forall$ und $R_W = R_G$ oder $R_W \supseteq R_G$.

- *Teilweise passend* (engl. partial match). Der Webdienst kann nur dazu beitragen, das Ziel zu erfüllen, es jedoch alleine nicht erzielen. Diese Stufe gilt beispielsweise für $I_W = \exists$, $I_G = \forall$ und $R_W = R_G$.

- *Unpassend* (engl. no match). Der Webdienst kann nicht dazu beitragen, das Ziel zu erfüllen. Diese Stufe gilt beispielsweise immer dann, wenn $R_W \cap R_G = \emptyset$.

Bewertend kann über den mengenbasierten Vergleichsansatz Folgendes ausgesagt werden:

- Die Auffassung von Diensten als Menge von Effekten, welche sie erwirken können bzw. welche erwirkt werden sollen, ist *intuitiv*.

- Der Ansatz ist *generisch*, d.h. in weiten Teilen von der zugrunde liegenden Sprache unabhängig. Eine Möglichkeit zur Realisierung des Verfahrens bietet zum Beispiel die Erfassung der Mengen R_G und R_W in einer Logiksprache wie etwa Beschreibungslogik. Die Bestimmung der Überlappung zwischen den Mengen wird dann auf die entsprechenden Schlussfolgerungsoperationen dieser Sprache zurückgeführt.

- Der Ansatz vollzieht nur einen *groben Vergleich*, da nicht betrachtet wird, wie die in die Dienste eingehenden und ausgehenden Informationen die erwirkten oder benötigten Effekte bestimmen oder verändern. Der Ansatz ist daher nur der erste Schritt (die Webdienstfindung) auf dem Weg der vollständigen Automatisierung gesehen werden, dem eine weiterer Schritt (die Findung des konkreten Dienstes) folgen muss. Vorschläge, wie dies zu erreichen wäre, finden sich in [69].

- Je nach Expressivität der zugrunde liegender Sprache ist die Berechnung der Überlappung zwischen R_G und R_W sehr *aufwändig* oder gänzlich *unberechenbar*.

Ableitungsbasierte Ansätze

Insgesamt unterscheiden sich die ableitungsbasierten Ansätze nur wenig von den mengenbasierten. Auch sie betrachten die eingehenden und ausgehenden Nachrichten nicht oder nur unvollständig und können nur im Rahmen des groben Webdienstvergleichs verwendet werden. Grundlage für ableitungsbasierte Verfahren ist die Auffassung der APPEs in Webdienst W und Ziel G als logische Formeln. Generell passt ein Webdienst zu einem Ziel, wenn aus den Effekten und Nachbedingungen

von W die gewünschten Effekte und Nachbedingungen von G logisch gefolgert werden können.

Ein solches ableitungsbasiertes Verfahren stellen **Keller, Stollberg und Fensel** in [70] vor. Zunächst werden vier einstellige Prädikate eingeführt: `g-post`(x) und `g-eff`(x) sind für die Instanzen bzw. Effekte x wahr, die vom Dienstnehmer im Ziel G gewünscht sind, `ws-post`(x) und `ws-eff`(x) hingegen für Instanzen bzw. Effekte x, die vom angebotenen Dienst W geliefert werden können. Die Definition der Prädikate kann aus den entsprechenden Dienstbeschreibungen abgelesen und als prädikatenlogische Formeln erster Stufe notiert werden. Dies ist möglich, da Ausdrücke in allen Varianten von WSML in Prädikatenlogik überführt werden können.

Für die Passung von G und W können dann drei Stufen abgeleitet werden:

- W kann G *vollständig erfüllen*, wenn
  $\forall x.(\texttt{g-post}(x) \rightarrow \texttt{ws-post}(x)) \wedge \forall x.(\texttt{g-eff}(x) \rightarrow \texttt{ws-eff}(x))$.
- W kann zur Erfüllung von G *beitragen*, wenn
  $\forall x.(\texttt{ws-post}(x) \rightarrow \texttt{g-post}(x)) \wedge \forall x.(\texttt{ws-eff}(x) \rightarrow \texttt{g-eff}(x))$,
  d.h. der Webdienst liefert in jedem Fall von G gewünschte Effekte, jedoch müssen evtl. weitere Dienste konsultiert werden, um G vollständig zu erfüllen.
- W kann zur Erfüllung von G *teilweise beitragen*, wenn
  $\exists x.(\texttt{ws-post}(x) \wedge \texttt{g-post}(x)) \wedge \exists x.(\texttt{ws-eff}(x) \wedge \texttt{g-eff}(x))$,
  d.h. der Webdienst liefert zum Teil von G gewünschte Effekte, jedoch müssen evtl. weitere Dienste konsultiert werden.

Je nach Intention von G bzw. W (d.h. je nachdem, ob alle oder nur ein einzelner Effekt gewünscht ist bzw. geliefert wird) muss überprüft werden, ob eine geeignete Passungsstufe erreicht ist.

Die Ableitung selbst wird in [70] von VAMPIRE übernommen, einem Schließer für Prädikatenlogik erster Stufe mit Gleichheit, welcher nicht-monotones Schließen nicht beherrscht. Die Umsetzung von WSML in Prädikatenlogik benötigt dies momentan nicht.

Bewertend lässt sich sagen, dass der Ansatz ähnliche Eigenschaften wie die mengenbasierten Ansätze aufweist. In der Tat könnte das Verfahren als Implementierung des mengenbasierten Ansatzes gesehen werden, wenn die vier Prädikate als charakteristische Funktionen der Mengen angesehen werden. Problematisch ist das Verfahren durch die Abbildung auf allgemeine Prädikatenlogik: Ein einzelner Vergleich ist somit sehr aufwändig und die Berechenbarkeit ist im allgemeinen Fall nicht garantiert. Zur Abmilderung des Problems schlagen die Autoren vor, Vergleichsergebnisse im Dienstverzeichnis vorzuberechnen, was allerdings ein mehr oder weniger

statisches Ziel-Verzeichnis voraussetzt, auf welches Dienstnehmer dann zurückgreifen müssen. Die Basierung auf reine Logik macht zudem die Verwendung von konkreten Domänen (wie Zeichenketten, Zahlen) und spezifischen Operatoren darauf schwierig. Diese müssten als Axiome formalisiert werden, was eine stark verminderte Effizienz zur Folge hätte.

Ein weiteres ableitungsbasiertes Verfahren wird von **Kifer et al.** in [71] vorgeschlagen[10]. Als Erweiterung wird hier die Integration von WG-Mediatoren vorgeschlagen. WG-Mediatoren erfüllen nach Meinung der Autoren zwei Aufgaben:

- Sie generieren aus einem gegebenen Ziel G die Eingaben, die zur korrekten Nutzung eines Dienstes W nötig sind. Dies wird durch die Formel $In_{wg}(\mathrm{G})$ beschrieben. Dies weicht von der ursprünglichen Idee eines solchen Mediators stark ab.
- Sie transformieren das Ziel G in eine Nachbedingung, die gegen die Effekte des Webdienstes W verglichen werden kann. Dies wird durch die Formel $Post_{wg}(G)$.

Zur Untersuchung, ob W geeignet ist, um G zu erfüllen, ist folgende Formel zu verifizieren:

$$O \wedge In_{wg}(G) \wedge W_{eff} \rightarrow Post_{wg}(G) \tag{3.1}$$

wobei O die verwendeten Ontologien und W_{eff} die Effekte des Webdienstes W darstellen. Es wird also überprüft, ob aus der Ontologie, den vom Mediator generierten Eingaben und den vom Webdienst erzeugten Effekten gefolgert werden kann, dass die (vom Mediator sprachlich angepassten) Nachbedingungen von G gelten.

Technisch realisiert wird diese Überprüfung in [71] durch Umsetzung der in G und W auftretenden WSML-Formeln in F-Logic. Die Schlussfolgerung kann dann mit $\mathcal{F}$-lora, einem speziell für F-Logic ausgelegten Schließer, erfolgen. Um diesen Schließer in der Praxis nicht nur für einen Webdienst W, sondern auch für ein ganze Menge von angebotenen Webdiensten einsetzen zu können, schlagen die Autoren die Verwendung von *transaktionaler Logik* vor. Hierbei wird für jeden angebotenen Webdienst angenommen, seine Effekte seien erfüllt und das entsprechende Axiom temporär in die Wissenbasis hinzugefügt. Nach Beendigung des Vergleichs von W wird das Axiom wieder entfernt.

Bewertend lässt sich sagen, dass der Gebrauch des WG-Mediators auf diese Weise als sehr kritisch anzusehen ist. Die Autoren verwenden ihn, um ausgehend von einem Ziel G passende Eingaben für einen eigentlich zuvor unbekannten Webdienst bereitzustellen. Das ist insbesondere problematisch, da der Mediator in der Praxis manuell

[10] Die Ideen finden sich auch in [69].

erstellt werden muss. Darüber hinaus dient der extern bereitzustellende Mediator nur als Transformator, sodass kein Mehrwert gegenüber der Lösung von Keller et al. entsteht.

3.3.5. Bewertung von WSMO

WSMO gilt als die „europäische Antwort“ auf OWL-S.[11] Dieser Abschnitt fasst zusammen, wo die Stärken und Schwächen von WSMO liegen (siehe dazu auch die Übersichtstabelle in Abbildung 3.14 auf Seite 81). Folgende Vorteile sind für WSMO zu nennen:

- WSMO macht eine explizite Unterscheidung in ein abstraktes *konzeptionelles Modell* und konkrete Umsetzungen. Dies hat den Vorteil, dass unterschiedliche Forschungsgruppen ihre eigenen Vorstellungen und Ansätze bezüglich WSMO umsetzen können. Mit der vorgeschlagenen Grundlagensprache WSML sowie der Referenzimplementierung der Ausführungsumgebung WSMX soll demonstriert werden, dass das konzeptionelle Modell generell tragbar ist. A1 (der Anforderungen auf Seite 40) ist also im Gegensatz zu OWL-S erfüllt.
- WSMO trennt bereits in seinen Hauptelementen in Beschreibungen von *angebotenen und benötigten Diensten* und unterscheidet sich darin gegenüber den meisten anderen semantischen Dienstbeschreibungssprachen wie auch OWL-S. Grund ist die Erkenntnis, dass zur Notation benötigter Dienste (Ziele) andere Aspekte im Vordergrund stehen als bei der Beschreibung angebotener Dienste (Webdienste). A4 ist also im Gegensatz zu OWL-S erfüllt.
- WSMO schlägt bereits in seinen Hauptelementen die Verwendung von *Mediatoren* vor. Diese Komponenten soll speziell die im Rahmen offener Umgebungen auftretende Heterogenität zwischen den zusammenarbeitenden Teilsystemen ermöglichen, indem sie Daten und Prozesse angleichen.

Demgegenüber steht eine Reihe von Problemen und offenen Fragen:

- Viele der vorgestellten Konzepte sind noch unausgereift oder unvollständig. Dies trifft insbesondere auf das Konzept der *Mediatoren* zu. Diese werden zwar explizit als eines der vier Hauptelemente von WSMO vorgeschlagen, dann jedoch als externe Komponenten angesehen, die ausgelagert und über einen Verweis in Beschreibungen eingebunden werden sollen. Über den inneren Aufbau der Mediatoren macht WSMO daher keine Angaben, räumt aber ein, dass diese

[11] Ein direkter Vergleich zwischen beiden Sprachen findet sich in [92].

nicht vollständig automatisch erstellbar sind. Beschreibungen ohne solche Mediatoren sind demnach inhaltlich nicht eindeutig (wodurch A3 nur als teilweise erfüllt angesehen werden kann). Insbesondere die WG- und GG-Mediatoren als Komponenten, die zwischen Webdiensten und Zielen vermitteln, sind noch größtenteils undefiniert. Es bleibt beispielsweise unklar, wie diese in Umgebungen mit zuvor unbekannten Diensten und Zielen definiert werden sollen, ohne zu einer statischen Bindung zu führen. Auch die Festlegung von *Schnittstellen* ist noch vage und zum Teil nicht mit der Definition von Fähigkeiten kompatibel.

- Als zweites großes Problem ist die *Semantik* von WSMO-Beschreibungen zu nennen. Zwar stellt WSML eine Sprache dar, die speziell zur Realisierung von WSMO gedacht ist und explizit Konstrukte zur Umsetzung der vier Hauptelemente enthält, dennoch ist hier nur eine Semantik für das Hauptelement der Ontologie definiert. Die Bedeutung von Webdienst-, Ziel- oder Mediatorbeschreibungen fehlt und damit auch die Semantik von Dienstbeschreibungen. Somit stellt WSML streng genommen keine wirkliche Neuerung dar und könnte auch durch andere Ontologiesprachen (wie etwa OWL+SWRL) ersetzt werden. Die eigentlichen Vorteile von WSML liegen in der Vereinigung zwischen Modellierungs- und Formelsprache. A7 ist demnach (noch) nicht erfüllt.

- Der *Vergleich* von WSMO-Beschreibungen ist ebenso problematisch. Die Aufteilung in einen zweistufigen Vergleich ist zwar prinzipiell wünschenswert, WSMO konzentriert sich aber ausschließlich auf die erste Stufe, den so genannten Webdienstvergleich. Hierbei werden nur abstrahierte Versionen der Ziele und Webdienste miteinander verglichen, bei denen beispielsweise die Ein- und Ausgaben der zu vergleichenden Dienste unberücksichtigt bleiben. Ein positives Vergleichsergebnis bietet daher keine Garantie für eine erfolgreiche Diensterbringung, sodass der Vergleich als grober Vorvergleich angesehen werden muss. A8 ist demnach (noch) nicht erfüllt. Existierende Vergleicher beruhen auf einem mengen- oder ableitungsbasierten Ansatz, welcher typischerweise sehr aufwändig oder für den allgemeinen Fall nicht berechenbar ist (A9 nicht erfüllt). Auch die Einbindung der Mediatoren in der Vergleichsprozess ist noch unklar. Streng genommen ist durch die fehlende Semantik das Ergebnis eines Vergleiches nicht eindeutig definiert, wodurch unterschiedliche Auffassungen für einen korrekten Vergleich existieren.

- WSMO unterscheidet zwar zwischen angebotenen und benötigten Diensten, der Grund für diese Unterscheidung liegt jedoch hauptsächlich in der Wiederverwendbarkeit von Zielen, nicht zwangsläufig in einer veränderten oder mächtigeren Beschreibung von benötigten Diensten, was dadurch zum Ausdruck kommt, dass Ziele und Webdienste quasi den gleichen inneren Aufbau besitzen, d.h. mit den gleichen Elementen beschrieben werden. Insbesondere ist es dem Dienstnehmer nicht möglich, seine Präferenzen bezüglich mehrerer unterschiedlicher

Dienste auszudrücken. Das Vergleichsergebnis ist also auch hier ähnlich wie bei OWL-S verzerrt, da es von allgemeinen Heuristiken innerhalb des Vergleichers abhängt und keine individuelle Bewertung der Ergebnisse zulässt. A6 ist demnach nicht erfüllt.

- Die Eignung von WSMO in der Praxis ist (wie bei fast allen semantischen Dienstbeschreibungssprachen) nicht evaluiert (A13 nicht erfüllt). Eine Untersuchung müsste klären, ob WSMO tatsächlich zur dynamischen und automatischen Dienstbindung verwendet werden kann, insbesondere im typischen Fall, wenn die Dienstlandschaft zum Zeitpunkt der Beschreibungserstellung eine andere ist wie zum Zeitpunkt der Dienstnutzung. Es ist fraglich, ob WSMO hier für eigentlich passende Dienste Beschreibungen nahe legt, die später von einem Vergleicher als passend angesehen werden. Ob A10 erfüllt ist, ist somit zweifelhaft.

Insgesamt bietet WSMO einer Reihe neuer Konzepte und kann somit als Verbesserung gegenüber OWL-S angesehen werden. Dennoch scheitert die Erfüllung des Gesamtziels noch an einer Reihe ernst zu nehmender Probleme (wie die Mediation, der Vergleich, die mangelhafte formale Semantik usw.). Nötig wäre eine Evaluation, mit welcher gezielt Schwächen aufgedeckt und behoben werden könnten.

3.4. Weitere Forschungsansätze zu semantischen Dienstbeschreibungssprachen

Dieser Abschnitt beschreibt weitere Forschungsansätze zu semantischen Dienstbeschreibungssprachen. Grundlage bilden meist die beiden großen Ansätze OWL-S und WSMO, die kombiniert oder projektspezifisch erweitert werden. Vorgestellt werden SWSF, METEOR-S, IRS-III und WSDL-S, die jeweils gegen die Anforderungen aus Abschnitt 3.1 verglichen werden. Das Ergebnis ist in der Tabelle in Abbildung 3.14 auf Seite 81 zusammengefasst.

3.4.1. SWSF

Die *Semantic Web Services Initiative*[12] (SWSI) ist eine spontane Initiative verschiedener Forscher, die sich zum Ziel gesetzt hat, die Techniken aus dem Bereich des Semantischen Webs und aus dem Bereich der Webdienste zu kombinieren, um so zu einer maximalen Automatisierung und Dynamisierung bei der Suche, Auswahl, Komposition und Ausführung von Diensten zu gelangen. Dabei sollen bestehende

[12] `http://www.swsi.org/`

Forschungsergebnisse (insbesondere aus OWL-S) fortgeführt und der Öffentlichkeit durch Einreichung beim W3C zugänglich gemacht werden. Die SWSI untergliedert sich in fünf Unterkomitees, von denen sich das *Semantic Web Services Language*[13] (SWSL) Committee mit der Erstellung einer geeigneten Dienstbeschreibungssprache befasst. Im Mai 2005 wurde von diesem Komitee die erste Version des *Semantic Web Services Framework* (SWSF) [11] vorgestellt. Es besteht (ähnlich wie WSML und WSMO) aus zwei Bestandteilen:

- Die *Semantic Web Services Language* (SWSL) als zugrunde liegende Ontologie- und Logiksprache. SWSL existiert in zwei Ausprägungen: als SWSL-FOL basierend auf Prädikatenlogik und SWSL-Rules basierend auf Horn-Regeln mit Erweiterungen. Die Sprache ist generisch, soll sich aber speziell für die Beschreibungen von Diensten eignen.
- Die *Semantic Web Services Ontology* (SWSO) als SWSL-Ontologie, mit deren Hilfe Dienste erfasst werden können. Die Ontologie liegt in zwei Ausprägungen vor: die *First-order Logic Ontology for Web Services* (FLOWS) ist in SWSL-FOL spezifiziert, die *Rules Ontology for Web Services* (ROWS) in SWSL-Rules.

Trotz der Ähnlichkeit zu WSMO muss die Sprache als Fortführung der Ideen aus OWL-S angesehen werden. Eine Dienstbeschreibung in SWSO ähnelt daher mit *Service Description*, *Process Model* und *Grounding* der von OWL-S bekannten Dreiteilung. Die Auftrennung ist jedoch konsequenter: Die Service Description enthält nur nicht-funktionale Aspekte des beschriebenen Dienstes, während das Process Model die funktionale Semantik erfasst. Dies geschieht durch eine reiche Prozessdefinitionssprache auf Basis der *Process Specification Language* (PSL) [53]. Grundlage ist eine Beschreibung der Dienstdomäne durch zeitveränderliche Prädikate, so genannte *fluents*. Dienste werden dann über Aktivitäten beschrieben, welche diese Prädikate (und somit die beschriebene Welt) verändern oder Nachrichten über Kanäle austauschen, wobei der Zusammenhang zwischen Nachrichten und Zuständen über spezielle Prädikate erfasst wird. Die Semantik der Sprache wird über eine Axiomatisierung in Prädikatenlogik unter Verwendung eines *Knows*-Konstrukts, welches das Wissen eines Teilnehmers ausdrückt, sowie des zeitlichen Prädikats *prior* formalisiert.

Insgesamt kann OWL-S als Vorgänger von SWSO gesehen werden und stellt somit eine Teilmenge dar. Insbesondere durch die Verwendung von Prädikatenlogik im Vergleich zu Beschreibungslogik bei OWL-S, reichhaltigere Modellierungsmöglichkeiten bei der Choreographie und Konstrukte zur Beschreibung von Ausnahmefällen entsteht eine größere Ausdrucksmächtigkeit. Aus WSMO werden die Ideen der getrennten Anfragebeschreibung als Ziele sowie die Vermittlung durch Mediatoren und der damit verbundene Vergleich von Beschreibungen übernommen. Somit „erbt“ SWSO eine

[13] `http://www.daml.org/services/swsl/`

Reihe der Probleme dieser Ansätze, was sich insbesondere in den noch unvollständigen und wenig überzeugenden Beispielbeschreibungen widerspiegelt: Einerseits sind die verwendeten Domänenontologien häufig durch die zeitveränderlichen Prädikate und die Verwendung von IDs auf einen speziellen Dienst angepasst, wie etwa

```
Book_inventory [
     ISBN => xsd#string
     warehouse_id => xsd#string,
     quantity_on_hand => xsd#integer
]
```

welches ausdrückt, dass ein Buch in einer bestimmten Anzahl in einem Warenhaus des Dienstgebers mit der gegebenen ID vorhanden ist. Da diese IDs aus dem internen Datenmodell des Dienstgebers stammen, welches dem Dienstnehmer nicht bekannt ist, wird eine unabhängige Erstellung von Dienstbeschreibungen nahezu unmöglich (A10 ist daher nicht erfüllt). Andererseits findet durch die Verwendung von Mediatoren, welche quasi eine explizite Abbildung von Zielen auf Dienstbeschreibungen vornehmen müssen, kein echter Vergleich von Dienstbeschreibungen statt (A8 ist nicht erfüllt).

3.4.2. METEOR-S

METEOR-S [149], entstanden aus dem Projekt *Managing End-To-End OpeRations* (METEOR), befasst sich mit der Einbringung von Semantik in alle Bereiche von Webdiensten wie etwa die Annotation der Beschreibungen, die Findung, Komposition und Ausführung von Diensten. METEOR-S stellt dabei keine eigene Sprache zur Verfügung, sondern verfolgt einen Ansatz, bei dem gewöhnliche Webdienste auf Basis von WSDL und UDDI um zusätzliche Semantik erweitert werden [134]. Konkret werden folgende Annotationen vorgenommen:

- In WSDL-Beschreibungen werden allen auftretenden Nachrichtentypen (also Ein- und Ausgaben) Konzepte einer externen Domänenontologie zugeordnet.
- Weiterhin werden in WSDL-Beschreibungen allen auftretenden Operationen Konzepte einer speziellen Operationsontologie zugewiesen und zusätzlich mit Vorbedingungen und Effekten in Form von logischen Formeln versehen.
- UDDI-Dienstverzeichnisse werden mit Konzepten aus einer speziellen Verzeichnisontologie versehen, die festlegt, welche Art von Diensten das Verzeichnis aufnehmen kann. Durch eine Vererbungs- und Partnerschaftsbeziehung zwischen den Konzepten entsteht eine Hierarchie von Dienstverzeichnissen, die Verweise auf andere Verzeichnisse enthalten können.

- Dienstanfragen stellen ebenso annotierte WSDL-Dokumente dar, enthalten aber zusätzlich eine Angabe, welche Verzeichnisse verwendet werden sollten.

Der Vergleich von mit METEOR-S annotierten WSDL-Beschreibungen läuft daher in verschiedenen Stufen (siehe dazu [151]). Zunächst wird die Anfrage an ein entsprechendes UDDI-Verzeichnis gesandt. Dort werden mithilfe von logischen Schließern getrennt voneinander zunächst die annotierten Operationen, dann die Ein- und Ausgaben und zuletzt die Vorbedingungen und Effekte verglichen.

Insgesamt stellt METEOR-S einen schnellen und praktischen Ansatz dar, wie mit moderatem Zusatzaufwand existierende WSDL-Beschreibungen semantisch angereichert werden können. Den hohen Anforderungen aus Abschnitt 3.1 ist METEOR-S jedoch nicht gewachsen: Erstens fehlt ein konzeptionelles Modell (A1 nicht erfüllt) und eine Formalisierung der Semantik (A5 nicht erfüllt). Durch die starke Anlehnung an WSDL und die Nähe zu OWL-S 1.0 ergibt sich insgesamt nur eine geringe Steigerung der Ausdruckskraft, insbesondere eine durch die unverbundenen IOPEs nach wie vor uneindeutige funktionale Beschreibung (A5/A6 nicht erfüllt) sowie daraus resultierend für die Automatisierung nicht ausreichend effektive Vergleichsverfahren (A8 nicht erfüllt). Die Verwendung von spezialisierten Operationsontologien verhindert darüber hinaus eine unabhängige Erstellung von Beschreibungen (A10 nicht erfüllt).

3.4.3. IRS-III

Der *Internet Reasoning Service* (IRS) [33] stellt eine Infrastruktur dar, um Semantische Webdienste veröffentlichen, auffinden und ausführen zu können und beschäftigt sich auch mit der Komposition von Diensten. Die Weiterentwicklung findet vorwiegend an der Open University in Milton Keynes statt. Die aktuelle Version IRS-III kann als Umsetzung von WSMO angesehen werden, wodurch sie die meisten Vorteile und Probleme dieses Ansatzes übernimmt. Sie basiert jedoch nicht auf WSML, sondern auf der *Unified Problem Solving Method Development Language* (UPML) [38]. Weitere Besonderheiten sind die Möglichkeit des *One-Click-Publishings*, d.h. die Erstellung eines Webdienstes aus einer Java- oder Lisp-Methode ohne Konfigurationsaufwand, sowie die *fähigkeitsbasierte Dienstausführung*. Hierbei verwendet der Dienstnehmer zur Entwurfszeit eine mit Ein- und Ausgaben angereicherte Zielbeschreibung, für die die Ausführungsumgebung zur Laufzeit dann einen geeigneten Dienstgeber bereitstellen soll (A11 ist somit im Gegensatz zu WSMO erfüllt). Der Vergleich von Beschreibungen findet über den *IRS-III Broker* statt, welcher (nicht genauer spezifizierte) Mediatoren zum Vergleich einsetzen soll. Die Anforderung A8 kann daher auch für IRS nicht als erfüllt angesehen werden.

3.4.4. WSDL-S

WSDL-S [2] stellt eine relativ neue Möglichkeit zur semantischen Beschreibung von Diensten dar. Zunächst von der Universität Georgia entwickelt, ist WSDL-S seit November 2005 zur Standardisierung beim W3C eingereicht. Es handelt sich dabei um einen eher praktischen Ansatz, bei dem ähnlich wie bei METEOR-S gewöhnliche, existierende WSDL-Beschreibungen semantisch angereichert werden, indem die in ihnen verwendeten Operationen und Konzepte durch Verweise auf eine externe Ontologie mit Semantik versehen werden. Die Art der verwendeten Ontologie ist dabei bewusst offen gelassen, um eine hohe Flexibilität und Praxistauglichkeit zu erreichen. In diesem Sinne stellt WSDL-S keinen wirklich eigenständigen Ansatz dar, da nur vorhandene Sprachen kombiniert werden.

Bei der Entwicklung von WSDL-S standen folgende Prinzipien im Vordergrund:

- Soweit wie möglich sollten *existierende Standards verwendet* werden, um so den Einstieg in die Sprache und deren Verbreitung in der Praxis zu erleichtern.
- Die Festlegung der Semantik sollte *unabhängig von bestimmten Ontologiesprachen* sein, um so ein hohen Maß an Flexibilität zu erreichen und möglichst viele existierende Domänenontologien unverändert wiederverwenden zu können.
- Zur Verknüpfung der Konzepte innerhalb einer WSDL-Beschreibung mit den Konzepten der Ontologie sollten *ausdrucksstarke Abbildungsoperationen* verwendet werden können.

In WSDL-S werden daher eine Reihe neuer Elemente definiert, die in gewöhnliche WSDL-Beschreibungen integriert werden können, um so die Semantik der Beschreibung zu erfassen. Im Einzelnen sind das:

- `modelReference` für direkte Abbildungen von WSDL-Konzepten auf Konzepte der Ontologie. Möglich sind diese Referenzen für die WSDL-Elemente `operation`, `input` und `output`.
- `schemaMapping` für komplexere Transformationen zwischen dem Datenmodell des Webdienstes und dem Schema der Ontologie. Notiert werden kann die Abbildung beispielsweise mithilfe der XML-Transformationssprache XSLT.
- `precondition` und `effect` als zusätzliche Elemente zur Erfassung von Vor- und Nachbedingungen des Dienstes. Ausgedrückt werden diese durch einen Verweis auf eine logische Formel, etwa in SWRL. Details, insbesondere der Zusammenhang zu den Ein- und Ausgaben der WSDL-Beschreibung, sind noch nicht definiert.

- `category` als zusätzliches Element zur Erfassung des wirtschaftlichen Zweiges des Dienstes. Diese kann einer der bereits existierenden Taxonomien entnommen.

Der wichtigste Vorteil dieses Ansatzes gegenüber den bereits vorgestellten Sprachen ist sicher seine Einfachheit und die große Akzeptanz der grundlegenden Sprache WSDL. Zudem stellt IBM eine Reihe von Werkzeugen zur Verfügung, um die Erstellung und Wartung solcher Beschreibungen zu erleichtern (A12 erfüllt). Durch WSDL als Basis ist eine Integration in Applikationen leicht möglich (A11 erfüllt). Es ist daher zu erwarten, dass der Ansatz eine große Verbreitung finden wird.

Dennoch kann dieser insgesamt eher einfache Ansatz nicht die Anforderungen aus Abschnitt 3.1 erfüllen: Ihm liegt kein konzeptionelles Modell zugrunde (A1 nicht erfüllt) und er behandelt angebotene und benötigte Dienste auf die gleiche Weise (A4 und A6 nicht erfüllt). Die Semantik einer Dienstbeschreibung ist zwar teilweise für die einzelnen Bestandteile durch die Zielsprachen gegeben, die Gesamtsemantik ist jedoch nicht formalisiert und bleibt damit unklar (A5 und A7 sind nur teilweise erfüllt). Auch ist bisher kein Vergleichsalgorithmus für WSDL-S veröffentlicht (A8 und A9 nicht erfüllt). WSDL als Ausgangssprache lässt jedoch vermuten, dass hier ein getrennter Vergleich der IOPEs und der Dienstkategorie angestrebt wird, wozu dann Spezialvergleicher der jeweiligen Sprachen herangezogen werden. In diesem Falle wären Beschreibungen nicht unabhängig voneinander erstellbar, da Signatur und Semantik isoliert voneinander übereinstimmen müssten. Auch andere Gründe sprechen gegen die unabhängige Erstellbarkeit von WSDL-S-Beschreibungen: Werden Beschreibungen mit Ontologien unterschiedlicher Ontologiesprachen (etwa WSML und RDF) annotiert, so ist es quasi unmöglich, einen sinnvollen Vergleich durchzuführen (A10 nicht erfüllt). Bisher existiert auch noch keine Evaluation des Ansatzes (A13 nicht erfüllt).

3.4.5. Weitere Ansätze

Neben den vorgestellten Ansätzen existiert noch eine Reihe kleinerer und älterer Ansätze verschiedener Einzelinstitutionen. Zu nennen sind hier noch:

- **DReggie** [26] der *eBiquity Research Group*[14] aus dem Jahr 2001. Idee war die Erweiterung der Jini-Dienstsuche um semantische Elemente aus DAML. Der Vergleich von Beschreibungen sollte mit einem Prolog-Schließer durchgeführt werden. Die Ansätze blieben jedoch vage. Auch im Bereich Bluetooth versuchte die Gruppe, die Dienstvermittlung durch Verwendung von Technologien des

[14] `http://ebiquity.umbc.edu/research/`

Semantischen Webs zu verbessern. In [7] stellten AVANCHA et al. Erweiterungen des Service Discovery Protocols (SDP) vor, welche die Dienstbeschreibung um RDF-Konstrukte anreichern. Hierbei handelt es sich jedoch ausschließlich um nicht-funktionale Attribute zu Dienstqualitäten oder -kosten, die zudem nur atomare Werte zulassen. Funktionale Aspekte wurden im Ansatz nicht berücksichtigt.

- Dienstbeschreibungen basierend auf **Golog** [100, 99], einer Logiksprache zur Abbildung komplexer Aktionen in dynamischen Domänen basierend auf dem Situationskalkül. Die Sprache erlaubt insbesondere eine Planung von Handlungen, die für ein Ziel benötigt werden und ist daher speziell zur Kombination von Diensten geeignet.
- **Entish** [3] als Beschreibungssprache, die Dienste über die Nachrichten beschreibt, welche sie versenden und empfangen können, und dabei das Austauschprotokoll explizit angibt. Besonderes Augenmerk schenkt dieser Ansatz ebenfalls der Komposition von Diensten.
- Dienstbeschreibungen der **Universität Paderborn** [59], dargestellt als Operationen auf einem Graphen, der die Domäne des Dienstes repräsentiert. Die eigentlich gute Idee, die im Jahr 2004 vorgestellt wurde, wurde jedoch nicht fortgeführt.

Daneben existiert eine Vielzahl an Ansätzen, die die beiden großen Dienstbeschreibungssprachen OWL-S und WSMO verwenden und nur marginale Weiterentwicklungen bieten. Aus diesem Grund werden diese hier nicht vorgestellt.

3.5. Zusammenfassung der Erfüllung der Anforderungen

Die Tabelle in Abbildung 3.14 zeigt eine Zusammenfassung, in wie weit die vorgestellten Ansätze OWL-S, WSMO, SWSF, METEOR-S, IRS-III und WSDL-S die nötigen Anforderungen erfüllen. Dabei bedeutet „•“, dass die Anforderung erfüllt ist, „○“, dass die Anforderung teilweise erfüllt ist, und „-“, dass die Anforderung nicht erfüllt.

Insgesamt lässt sich festhalten, dass alle betrachteten Ansätze noch eine Reihe von Schwächen haben. Insbesondere eine deterministische Beschreibung benötigter Dienste (A6), einen effektiven *und* effizienten Vergleich (A8 und A9), die unabhängige Erstellbarkeit (A10) sowie eine Evaluation (A13) kann keines der Verfahren aufweisen (fettgedruckte Anforderungen der Tabelle). Auch eine formale Semantik aller relevanten Aspekte (A7) ist in den meisten Ansätzen nicht definiert – lediglich das neuere SWSF erfüllt diese Anforderung.

		OWL-S	WSMO	SWSF	METEOR-S	IRS-III	WSDL-S
A1	Konzeptionelles Modell	-	●	●	-	●	-
A2	Universal	●	●	●	●	●	●
A3	Inhaltlich eindeutig	●	○	○	●	○	●
A4	Angeb. vs. benöt. Dienste	-	●	●	●	●	-
A5	Eindeutige Angebotsbeschr.	○	●	●	-	●	○
A6	Deterministische Anfrageb.	-	-	-	-	-	-
A7	Formale Semantik	-	○	●	-	-	○
A8	Effektiv vergleichbar	○	-	-	○	-	-
A9	Effizient vergleichbar	●	-	-	●	-	-
A10	Unabhängig erstellbar	-	-	-	-	-	-
A11	In Applikationen einbindbar	○	○	○	●	●	●
A12	Durch Menschen verarbeitbar	●	●	●	●	●	●
A13	Positiv evaluiert	-	-	-	-	-	-

Abbildung 3.14.: Übersicht über die Erfüllung der Anforderungen für wichtige Ansätze zur semantischen Dienstbeschreibung (● = Anforderung erfüllt, ○ = Anforderung teilweise erfüllt, - = Anforderung nicht erfüllt).

3.6. Arbeiten in verwandten Gebieten

Die automatische und dynamische Nutzung von Funktionalität spielt in vielen Gebieten der Informatik eine wichtige Rolle. Somit sind Beschreibungssprachen zur Erfassung von Funktionalität auch in anderen Bereichen Objekt der Forschung. Im Folgenden sollen einige dieser Forschungsgebiete vorgestellt werden. Dabei wird auf die jeweiligen Schwerpunkte und Unterschiede im Vergleich zur Beschreibung Semantischer Webdienste eingegangen.

3.6.1. Wiederverwendung von Komponenten

Das Auffinden von Funktionalität ist nicht erst seit dem Aufkommen von Webdiensten Thema der Forschung. Bereits Mitte der 1980er Jahre wurden Verfahren erforscht, mit denen Softwarekomponenten zum Aufbau einer größeren Applikation aus einer Bibliothek aufgefunden und so wiederverwendet werden konnten. Ein *Komponente* stellt dabei eine wiederverwendbare, abgeschlossene Softwareeinheit dar, die eine klar definierte Funktionalität erbringt. Bekannte kommerzielle Komponentensysteme mit großer Verbreitung sind die *Common Object Request Broker Architecture*[15] (CORBA) der OMG, Suns *Enterprise Java Beans*[16] (EJB) sowie Microsofts *Distributed*

[15] `http://www.corba.org/`
[16] `http://java.sun.com/products/ejb/`

Component Object Model[17] (DCOM) und .NET[18].

Trotz einiger Gemeinsamkeiten zwischen Komponenten und Diensten, überwiegen doch die Unterschiede (siehe dazu [131]): Hauptziel von Komponenten ist die Wiederverwendung von Funktionalität, um so schneller und günstiger große Applikationen erstellen zu können. Dazu sucht der Programmierer zur Entwicklungszeit aus einer Bibliothek für seine Anforderungen geeignete Komponenten heraus und integriert sie in die Gesamtsoftware. Es entsteht eine *enge Kopplung* und eine *statische Bindung*. Bei der Suche nach Komponenten wird der Anwender von Werkzeugen unterstützt, die ihm auf Anfrage möglicherweise passende Komponenten vorschlagen. Die Komponenten müssen dabei mit einer Beschreibung versehen werden, die meist auf den inneren Aufbau oder ein lokales Datenmodell Bezug nimmt.

Beim Einsatz von Diensten wird hingegen ein anderes Ziel verfolgt. Sie werden verwendet, um ein Softwaresystem um eine Funktionalität zu erweitern, die lokal nicht bereitgestellt werden kann oder soll, und die aufgrund ihres Umfangs oder Komplexität von fallweise wechselnden externen Dienstgebern erbracht werden soll. Der Anwendungsentwickler integriert somit zur Entwicklungszeit eine Beschreibung des benötigten Dienstes in die Gesamtsoftware. Erst zur Laufzeit werden pro Nutzung geeignete Dienstgeber gesucht und verwendet. Man spricht von *loser Kopplung* und *dynamischer Bindung*. Die Suche nach solchen Dienstgebern muss im Gegensatz zur Suche von Komponenten daher vollständig automatisch ablaufen, da kein menschlicher Benutzer mehr beteiligt ist, was eine entsprechend genaue Dienstbeschreibung erfordert. Weiterhin wird das Problem dadurch erschwert, dass Dienste meist im Unterschied zu Komponenten auf großen, nur beim Dienstgeber verfügbaren Datenbeständen basieren und auch Wirkungen außerhalb des Informationssystems erbringen können. Da sie unternehmensübergreifend Verwendung finden sollen, darf sich ihre Beschreibung daher nicht an lokalen Modellen orientieren, sondern muss ausschließlich mit anwendungsunabhängigem Vokabular auskommen.

Ansätze zum Auffinden von Komponenten können daher zwar als Grundlage zur semantischen Beschreibung von Webdiensten angesehen werden, jedoch nicht direkt übernommen werden. In der Tat finden sich viele der Verfahren, die insbesondere einen Vergleich der beteiligten Schnittstellenbeschreibungen (der *Signatur*) sowie der formalen Beschreibung ihrer Wirkung (der *Spezifikation*) durchführen, als erste Ansätze zum Vergleich von OWL-S- oder WSMO-Beschreibungen wieder. Einen guten Überblick über die verschiedenen Verfahren geben [153, 160, 159]. Für eine dynamische Bindung weltverändernder, unternehmensübergreifender Dienste, die anwendungsneutral und unabhängig voneinander beschrieben werden müssen, sind die Verfahren jedoch nicht ausreichend. Umgekehrt können semantische Dienstbeschreibungen nicht direkt zur Erfassung von Komponenten verwendet werden, da die Suche

[17] `http://www.microsoft.com/com/`
[18] `http://www.microsoft.com/net/`

hier zur Entwicklungszeit stattfindet, d.h. die konkrete Parameterbelegung wie bei der Dienstsuche zur Laufzeit noch nicht feststeht.

3.6.2. Agentensysteme

Ein zunehmend wichtiger werdendes Forschungsgebiet stellen Agentensysteme dar. *Agenten* sind Softwareeinheiten, welche im Gegensatz zu gewöhnlichen Komponenten oder Objekten autonom handeln und versuchen, ein Ziel zu erreichen. Dazu haben sie eine mehr oder weniger korrekte und vollständige Sicht auf die sie umgebende Welt. Agenten handeln selbstständig, d.h. nach Erhalt ihrer Aufgabe verrichten sie ihre Arbeit ohne von einem menschlichen Benutzer gesteuert zu werden.[19] Von großer Bedeutung sind *Multi-Agenten-Systeme*, in denen mehrere Agenten zusammenwirken, um eine Aufgabe zu erfüllen.[20] Für diese Kollaboration ist eine Kommunikation zwischen den verschiedenen Agenten nötig, wobei zwei grundsätzliche Ansätze unterschieden werden können:

- Der *dienstorientierte Ansatz* ist als klassisch anzusehen: Agenten in einem System bieten Funktionalität in Form von Diensten an, welche von anderen Agenten genutzt werden kann [24, 49]. Man spricht hier auch von *Agentenfähigkeiten*. Ablauf und Rollenverteilung orientieren sich am klassischen Dienstdreieck (bestehend aus Dienstgeber, -nehmer und -verzeichnis) und sind demnach eher streng festgelegt. Um ein selbstständiges Arbeiten der Agenten zu ermöglichen, ist für die Beschreibung der von Agenten angebotenen und benötigten Dienste eine semantische Dienstbeschreibungssprache erforderlich. Zwei dieser Sprachen (LARKS und eine RDF-basierte) werden weiter unten vorgestellt.

- Der *konversationsbasierte Ansatz* kann als Obermenge des dienstorientierten Ansatzes gesehen werden, der insbesondere für komplexe Problemstellungen zum Einsatz kommt, in denen der Agent auch selbstständig entscheiden muss, welche Funktionalität er wann anbieten oder nutzen möchte. Die Agenten besitzen dazu die Fähigkeit, im Rahmen von *Konversationen*, Informationen und Ansichten auszutauschen, Fragen an andere Agenten zu stellen oder Funktionalitätswünsche zu äußern. Die Semantik dieser *Sprechakte* (engl. speech acts, zuerst erwähnt in [6]) ist dabei formal hinterlegt und beinhaltet in der Regel eine (einfachere) Logiksprache, mit der der Inhalt eines Sprechakts erfasst werden kann. Absicht und Inhalt einer Nachricht sind damit getrennt. Die Rollenverteilung bei der Kollaboration ist somit flexibler und kann auch mehrere Agenten

[19] Die Definition eines Agenten ist ein viel diskutiertes Thema. Gebräuchlich ist die Definition nach [154], an der sich auch dieser Abschnitt orientiert.

[20] Die Vereinheitlichung und Weiterentwicklung der Forschungsergebnisse wird hauptsächlich von der *Foundation for Intelligent Physical Agents* (FIPA, `http://www.fipa.org`) koordiniert.

umfassen. Die beiden wichtigsten Sprachen zur Erfassung solcher Sprechakte sind die *Knowledge Query and Manipulation Language* (KQML) [90], die *Interagent Communication Language* (ICL) [98] und die *FIPA Agent Communication Language* (FIPA ACL) [135]. Den Inhalt ihrer Nachrichten beschreiben sie in einer Logiksprache wie etwa dem *Knowledge Interchange Format*[21] (KIF). Die Zuordnung von Sprechaktnachrichten zu Partnern übernehmen spezielle Agenten, so genannte *Vermittlungsagenten* (engl. broker agents). Diese überprüfen, welcher der angemeldeten Agenten die ausgesandten Nachrichten verstehen könnten. Verfahren, die das ermöglichen, sind beispielsweise der *RETSINA Matchmaker* [142], der *Open Agent Architecture Facilitator* [98] sowie der *Infosleuth Broker* [47].

Aufgrund der Nähe zu Beschreibungssprachen für Webdienste werden im Folgenden zwei Beschreibungssprachen für Agentenfähigkeiten vorgestellt.

Ein weit verbreiteter, aber auch schon etwas älterer Ansatz zur Beschreibung von Agentenfähigkeiten stellt die *Language for Advertisement and Request for Knowledge Sharing* **LARKS** [143, 144] dar, die analog zu OWL-S die Funktionalität eines Agenten über die vier Parameter Ein- und Ausgabe, Vor- und Nachbedingung erfasst. Darüber hinaus bietet LARKS die Möglichkeit, spezielle Datentypen zu definieren und textuelle Beschreibungen für den Kontext, verwendete Begriffe und Funktionalität einzufügen. Ein Vergleich zwischen zwei Beschreibungen dieser Sprache erfolgt durch Anwendung von bis zu fünf Filtern, die aufsteigend geordnet immer mehr Semantik der Beschreibungen berücksichtigen:

- *1. Filter: Kontext.* Dieser Filter testet, ob die beiden Beschreibungen generell in derselben Domäne operieren. Dazu werden die Schlagworte der Kontexte auf Ähnlichkeit verglichen.
- *2. Filter: Profile.* Hierzu wird die gesamte Beschreibung als Volltext aufgefasst und die textuelle Ähnlichkeit mit einer Variante der TF/IDF-Technik überprüft. Bei dieser Technik werden insbesondere die Worte untersucht, die in Vergleichstexten relativ oft, in anderen Texten jedoch eher selten vorkommen.
- *3. Filter: Similarity.* Mit diesem Filter werden Worte, die in der Beschreibung der Eingaben vorkommen, miteinander vergleichen. Analog werden die Worte aus Ausgabe, Vor- und Nachbedingung analysiert.
- *4. Filter: Signature.* In diesem Filterschritt wird eine Variante des *Semantic Matchmakers* für OWL-S verwendet (vgl. Abschnitt 3.2.2). Generell sind zwei Parameter ähnlich, wenn einer ein Untertyp des anderen ist.

[21] `http://logic.stanford.edu/kif`

- *5. Filter: Constraint.* Im letzten Filter werden die Vor- und Nachbedingungen analysiert. Dazu werden diese als Hornklauseln aufgefasst und versucht, eine logische Ableitung zwischen diesen herzustellen.

Bei den ersten drei Filtern dieses Ansatzes besteht das Problem, dass der inhaltliche Gehalt des Dienstes nur geraten, nicht aber korrekt verstanden wird. Sie können daher allenfalls als Hinweis für die Ähnlichkeit der Dienste dienen, die auf jeden Fall aber durch einen menschlichen Benutzer überprüft werden muss. Filter 4 leidet unter den bereits erwähnten Problemen. Weiterhin kann durch die vom Nachrichtenfluss getrennte Betrachtung des Zustandsübergang in Filter 5 die Semantik des Dienstes nicht eindeutig erfasst werden.

Auch der von Hewlett Packard im Rahmen des **agents@HP Lab** entwickelte Ansatz [147] stammt bereits aus dem Jahr 2001 und gilt heute als überholt. Agentenfähigkeiten werden hier durch RDF-Dokumente erfasst, die einem vordefinierten RDF-Schema folgen. Als Vergleichsverfahren für solche Dienstbeschreibungen wird ein einfacher Graphenvergleich vorgeschlagen. Es gilt daher: (1) Zwei Dienstbeschreibungen sind ähnlich, wenn ihre Wurzelelemente ähnlich sind und (2) zwei Elemente a und b sind ähnlich, wenn (a) a Unterklasse von b ist und (b) jede Eigenschaft in a, die als Eigenschaft oder Untereigenschaft auch in b vorkommt, auf ähnliche Knoten verweist. Insgesamt werden also durch den Algorithmus die beiden Dienstbeschreibungsgraphen simultan durchlaufen und rekursiv auf Typgleichheit überprüft. Um den Vergleich zu verfeinern, ist es möglich, speziellen Vergleichscode (in Form von Javamethoden) an die entsprechende Klasse des RDF-Schemas zu hängen. Diese Möglichkeit wird jedoch nicht genauer betrachtet. Zudem können auch einfache Bedingungen wie `größerOderGleich 300` als Text in den Graph der Anfragebeschreibung eingebracht werden. Diese werden dann von einem speziellen Parser heraus gelöst und mit dem konkreten Wert in der Angebotsbeschreibung überprüft.

Das Verfahren hat eine Reihe von Nachteilen: Es fehlt ein formale Semantik, die Beschreibungen sind nicht eindeutig, der Vergleich liefert keine Abstufungen, der Umgang mit strukturell unterschiedlichen Beschreibungen bliebt undefiniert und nicht zuletzt ist die Mischung von RDF und Programmcode unsauber.

Die heutige Forschung auf dem Gebiet der Beschreibung von Agentenfähigkeiten zeichnet sich durch ein Zusammenwachsen mit der Forschung auf dem Gebiet der Semantischen Webdienste aus. Aktuelle Projekte verwenden in der Regel die mittlerweile stärker etablierten Sprachen WSMO oder OWL-S[22] zur Erfassung ihrer Agentenfähigkeiten. Beispielhaft zu nennen ist hier das noch junge *Semantic Web FRED Projekt*[23], deren Beschreibungssprache *Semantic Web Fred* (SWF) den Grundaufbau

[22] Dabei ist zu beachten, dass OWL-S ursprünglich als *DARPA Agent Markup Language for Services* (DAML-S) genau zu diesem Ziel entwickelt wurde.

[23] `http://www.deri.at/research/projects/swf/`

und viele Details aus WSMO übernimmt [140] oder das bereits etabliertere Projekt *Agentcities*[24], das zur Beschreibung der Fähigkeiten seiner FIPA-Agenten OWL-S einsetzt [25].

3.6.3. Grid Computing

Grid Computing stellt ein weiteres Forschungsgebiet dar, in dem Webdienste zum Einsatz kommen. Ziel des Grid Computings ist der Zusammenschluss von Rechnern und Datenspeichern im Internet, um so zu einer hoher Rechenleistung, einem hohen Speichervolumen oder großen Angebot an Funktionalität zu gelangen, welche von der Teilnehmergemeinschaft genutzt werden kann. Die Ursprünge der Idee finden sich in [43]. Allgemein wird eine Virtualisierung von Ressourcen im Netz angestrebt. Hintergrund ist die Beobachtung, dass viele Geräte im Internet einen Großteil der Zeit nicht voll ausgelastet sind und so von anderen Nutzern verwendet werden könnten. Anwendungen des Grid Computings sind daher fast ausschließlich rechen- oder datenintensive Probleme aus dem wissenschaftlichem Umfeld (so genanntes *number crunching*), wie beispielsweise die aufwändige Berechnung von Proteinfaltungen oder die Auswertung von Radioteleskopdaten. Die Hauptaufgabe innerhalb eines Grids ist die Beantwortung der Fragen nach Konnektivität der Teilnehmer, nach Ressourcenfindung und Ergebnissammlung.

Aufgrund dieser Nutzung von Ressourcen im Grid existieren viele Gemeinsamkeiten mit dem Forschungsgebiet der Webdienste. Beide besitzen das Konzept eines Dienstes und gehen von einer dienstorientierten Architektur aus. Allgemein können also Ressourcen im Grid als Webdienste aufgefasst und beschrieben werden. Dennoch nennt [50] einige Unterschiede zwischen der Nutzung von Grid-Diensten und Webdiensten:

- Die Dienstlandschaft innerhalb eines Grids ist *dynamischer* als die des Webs, da Dienste häufig nur durch überschüssige Ressourcen einzelner Rechner erbracht werden können.
- Die Zahl der Dienstanbieter ist weitaus *größer*, da nicht nur Unternehmen, sondern häufig auch Einzelpersonen Grid-Dienste zur Verfügung stellen.
- Die Dienstnutzung im Grid ist *zustandsbehaftet* und kann dadurch auch eine *lange Lebensdauer* besitzen.

Neuere Forschungsansätze versuchen daher, Grid und Webdienste zu vereinen, indem sie die Ressourcen des Grids ausnahmslos als Dienste auffassen, mit bekannten

[24] `http://www.agentcities.org/`

Mitteln aus dem Gebiet der Webdienste beschreiben, auffinden und nutzen und nur punktuell um Grid-spezifische Besonderheiten erweitern.

Ein Beispiel stellt die *Open Grid Services Architecture* (OGSA) [46, 44] dar. Darin ist ein Grid-Dienst ein gewöhnlicher Webdienst, der um eine standardisierte Klientenschnittstelle erweitert ist, mittels derer der Dienstnehmer auf wohldefinierte Weise mit dem Dienst interagieren kann. Die Schnittstelle umfasst vor allem Funktionen zur Lebenszyklusverwaltung und Benachrichtigung des Dienstes. Eine mögliche Umsetzung von OGSA stellt die *Open Grid Services Infrastructure* (OGSI) [148] dar. Grid-Dienste werden darin mit erweitertem WSDL beschrieben und über SOAP genutzt. Insbesondere langlaufende Dienste und die gleichzeitige Nutzung mehrere Dienstinstanzen werden unterstützt. Implementiert wurde OGSI im Rahmen des *Globus Toolkits* [42]. Eine Fortführung und Generalisierung der Ideen von OGSA findet im *Web Services Resource Framework* (WSRF) [30] statt.

Ähnlich wie bei Webdiensten bieten Grid-Dienste zwar die Möglichkeit einer verteilten dienstorientierten Nutzung von Funktionalität, dennoch müssen diese zur Entwurfszeit vorausgewählt werden. Auch hier verspricht die Verbindung mit Techniken des Semantischen Webs eine Automatisierung der Dienstnutzung. Dadurch wäre ein selbsttätig agierendes Grid denkbar, an das sich Nutzer zur Laufzeit mit verschiedensten Ressourcenwünschen wenden könnten. Eine wesentlich einfachere und flexiblere Zusammenarbeit zwischen den Teilnehmer könnte so erreicht werden. Man spricht dann vom *Semantischen Grid* [25]. [125]

Bei der Realisierung des Semantischen Grids verwenden die meisten Projekte zur Erfassung der Ressourcen existierende semantische Dienstbeschreibungssprachen. So wird in [146] das Einbringen von WSMO als Beschreibungssprache für Grid-Dienste analysiert, während im *myGrid*-Projekt[26] die semantische Grundlage durch die Verwendung von OWL und OWL-S erreicht werden soll [155]. In beiden Fällen sind jedoch Anpassungen an der Sprache nötig, um die Besonderheiten von Grid-Diensten erfassen zu können.

Daneben existiert eine kleine Zahl von Ansätzen für Beschreibungssprachen, die speziell zur Erfassung von Grid-Diensten entwickelt wurden. In der Regel handelt es sich dabei jedoch um rein syntaktische Verfahren, die Grid-Dienste anhand von einfachen Zeichenkettenvergleichen oder Operatorauswertungen auswählen (ein typisches Beispiel zeigt [45]). Semantische Ansätze existieren nur wenige. Ein Beispiel ist in [145] vorgestellt: Die Beschreibung von Grid-Diensten erfolgt hier mittels RDF, wobei für angebotene und benötigte Dienste unterschiedliche Beschreibungen verwendet werden. Neben den Dienstbeschreibungen werden zusätzlich ontologische Beschreibungen des Anwendungsgebiets sowie domänenspezifische Passungsregeln benötigt. Diese werden für den Vergleich zweier Beschreibungen dieser Domäne herangezogen, um

[25] Eine Bündelung der Forschungsansätze auf diesem Gebiet findet sich unter `http://www.semanticgrid.org`

[26] `http://www.mygrid.org.uk/`

festzustellen, ob ein angebotener Dienst verwendet werden kann, um die Forderungen eines benötigten Dienstes zu erfüllen. Verglichen mit den generischen Vergleichern, die im Rahmen Semantischer Webdienste zum Einsatz kommen, ist dieses Verfahren jedoch sehr aufwändig. Generell stellt dieses und andere Verfahren (wie z.B. [96]) kaum Neuerungen auf dem Gebiet der semantischen Dienstbeschreibungen dar.

3.6.4. Information Retrieval

Information Retrieval (IR) beschäftigt sich mit dem inhaltsorientierten Auffinden von Informationen in großen Datenbeständen. Das Konzept unterscheidet sich in zwei Punkten von klassischen Datenbankanfragen:

1. Das System hat nur *mangelhafte Kenntnisse* über die im Datenbestand abgelegten Dokumente. So kann beispielsweise der Inhalt eines Textes oder das Motiv eines Bildes unbekannt sein.
2. Die Anfrage nach Informationen ist *vage*, da der Anfrager die genauen Elemente des Datenbestands im Voraus nicht kennt.

Diese beiden Probleme spiegeln sich in den beiden Schritten des IR wider: Zunächst extrahiert das System aus den Elementen des Datenbestands Merkmale und legt diese als Metadaten zusammen mit den Daten ab. Beispiele für solche Merkmale sind inhaltsbeschreibende Schlagworte für einen Text, die Hintergrundfarbe eines Bildes etc. Dieser Schritt ist äußerst komplex, nach wie vor ein aktives Forschungsgebiet und dementsprechend ungenau und fehlerbehaftet. Im zweiten Schritt kann ein Benutzer Anfragen an das System stellen, die anhand der extrahierten Merkmale beantwortet werden. Hierbei wird jedoch keine vollständige Automatisierung der Suche angestrebt, sondern eine Unterstützung der vom Benutzer durchgeführten manuellen Auswahl.

Interessant für diese Arbeit ist insbesondere der zweite Schritt, da auch im Rahmen der Dienstsuche vage Anfragen an Metadaten gestellt werden müssen. Seit Mitte der 1980er Jahre wurde eine Reihe von Anfragesprachen entwickelt, insbesondere für die Abfrage von Multimediadatenbanken [93, 107, 127, 28] oder Objektdatenbanken [19]. Als wichtigste Erkenntnis lässt sich ableiten, dass getrennte Sprachen zur Beschreibung der abgelegten Informationen und zur Darstellung von Anfragen unumgänglich sind, um die Wünsche des Anfragers genau erfassen zu können. Die meisten Anfragesprachen ermöglichen die Gewichtung von Anfrageteilen, die Angabe eines Schwellwerts für die Mindestähnlichkeit, die Verrechnung mehrerer Anfrageteile sowie die Verwendung linguistischer Variablen. Auch zur Abfrage von durch Metadaten beschriebener Dienste ist dies wichtig (vgl. Anforderung A6).

Dennoch können die Ansätze des IR aus folgenden Gründen nicht direkt auf die Suche nach Diensten übertragen werden:

- Das zugrunde liegende Datenmodell ist eingeschränkt (für Dokumente etwa auf die 15 Beschreibungselemente des *Dublin Core*) und für alle Anfragen gleich. Beschreibungssprachen für Dienste müssen nach A2 jedoch universal sein, d.h. Dienste aller Domänen erfassen können, wodurch ein festgelegter Satz an Merkmalen ausscheidet.
- Angebotene Dienste können in der Regel vor ihrer Verwendung konfiguriert werden (vgl. A5), während die Datenelemente eines IR-Systems parameterlos abrufbar sind.
- Die Nutzung eines Dienstes erfordert in der Regel eine Vermittlung zwischen der vom Dienstnehmer gewünschten und der vom Dienstgeber bereitgestellten Schnittstelle (vgl. A11). Im Gegensatz dazu können die Elemente eines IR-Systems direkt, d.h. ohne Anpassung genutzt werden.
- Die Suche nach Informationen in einem IR-System ist für den menschlichen Konsum bestimmt, d.h. eine Präzision < 1 führt nur zu einem höheren Aufwand, nicht aber zu ungewollten Effekten. Bei einer vollautomatischen Dienstnutzung ist jedoch eine Präzision von nahezu 1 unerlässlich.

Der Bereich des Information Retrieval kann demnach nur generelle Hinweise für das Themengebiet der semantischen Dienstbeschreibungen liefern, da die entwickelten Techniken nicht direkt übernommen werden können.

3.7. Fazit

Die Anforderungen an eine semantische Dienstbeschreibungssprache, die als Grundlage für eine inhaltlich korrekte und vollständig automatisierte Dienstnutzung in dynamischen Umgebungen verwendet werden kann, sind hoch. In Abschnitt 3.1 wurden 13 Anforderungen festgestellt. Die Untersuchung existierender Ansätze ergab, dass diese viele der Anforderungen noch nicht oder nur unzureichend erfüllen können. Keiner der betrachteten Ansätze verfügt beispielsweise über eine Möglichkeit zur deterministischen Beschreibung benötigter Dienste (A6), stellt einen effektiven *und* effizienten Vergleicher zur Verfügung (A8 und A9), ist darauf ausgelegt, dass Beschreibungen unabhängig voneinander erstellt werden können (A10) oder wurde innerhalb einer wissenschaftlichen Studie evaluiert (A13). Eine formale Semantik für die relevanten Aspekte einer Dienstbeschreibung (A7) bietet zurzeit nur SWSF.

Die Gründe für diese große Zahl an Schwächen können nicht alleine bei den Ontologien zur Beschreibung gesucht werden, sondern muss auch die zugrunde liegende Ontologie- oder Logiksprache mitberücksichtigen. These dieser Arbeit ist, dass sie zu wenig auf die speziellen Bedürfnisse einer Beschreibung von *Diensten* eingeht und

diese somit nur unzureichend erfasst. Vermischungen der Ontologiesprache mit Regelsprachen (wie etwa OWL und SWRL) zeigen das Dilemma: Die Ausdrucksstärke (und damit Probleme bei der Berechenbarkeit) steigt extrem an und dennoch bleibt es schwierig, realistische Beispieldienste zu beschreiben. Oft ist auch die Semantik für die gemischt verwendeten Elemente undefiniert. Insgesamt führt diese unzureichende Sprachgrundlage zu einer Reihe von Problemen:

- Operationen auf der (häufig mehrfach in der Expressivität erweiterten) Ontologiesprache sind meist nicht mehr effizient berechenbar. Insbesondere die Schichtung aus RDF, OWL und SWRL ist als kritisch anzusehen. Es entstehen so zwar extrem ausdrucksstarke Sprachen, die Ausdrucksstärke liegt jedoch „an der falschen Stelle“.
- Der Vergleich von Dienstbeschreibungen orientiert sich stets an den generischen Schlussfolgerungsoperationen, die für die zugrunde liegende Ontologie- oder Logiksprache zur Verfügung stehen. Hierdurch entstehen unintuitive Vergleichsergebnisse, die vom Dienstnehmer nicht nachvollzogen werden können.
- Die formale Semantik der generischen Ontologiesprache beschränkt sich meist auf die Beschreibung statischer Domänenontologien, umfasst aber nicht die dynamischen Teile einer Dienstbeschreibung.
- Die Einfachheit und Intuitivität bei der Modellierung von Domänen und Beschreibung von Diensten geht aufgrund der Komplexität der Ontologiesprache verloren. Das Erstellen von Beispielbeschreibungen sowie die Anwendung in der Praxis werden somit für Normalanwender unmöglich.
- Die hohe Ausdrucksstärke der grundlegenden Ontologiesprache in Kombination mit einer hohen Generizität der Dienstontologie führt zu einem gewaltigen Modellierungsspielraum für Dienstbeschreibungen. Es wird daher quasi unmöglich, dass zwei Parteien unabhängig voneinander passende Dienstbeschreibungen erstellen, wie dies für den Einsatz in dynamischen Umgebungen unabdingbar ist.

Im nächsten Teil der Arbeit wird daher ein Ansatz zur semantischen Beschreibung von Diensten vorgestellt, der auch die zugrunde liegende generische Ontologiesprache neu definiert, sodass diese speziell für die Beschreibung der Besonderheiten von Diensten ausgelegt ist und damit die 13 Anforderungen sowie die Ziele der Arbeit erfüllen kann.

Teil II.

Eigener Ansatz

4. Überblick über den eigenen Ansatz

Der Stand der Forschung hat gezeigt, dass existierende Ansätze große Mängel bei der Erfüllung der gestellten Anforderungen aufweisen. Die Gründe hierfür können nicht nur in unzureichenden Dienstontologien zu finden sein, sondern müssen tiefer liegen und auch die Ontologiesprache selbst betreffen. Der Ansatz dieser Arbeit schlägt daher insbesondere dienstspezifische Erweiterungen auf dieser Ebene vor.

Das Vorgehen gründet auf der Überlegung, dass eine einfache Kombination der Konzepte aus dem Bereich des Semantischen Webs und der Webdienste, wie sie die Forschung um Semantische Webdienste nahe legt, nicht zum Erfolg führt. Dies liegt vor allem daran, dass das Semantische Web vorwiegend auf die Beschreibung statischer Informationen und Daten abzielt, was für eine Beschreibung dynamischer Webdienste unzureichend ist. Die These, die in dieser Arbeit aufgestellt wird, ist daher die, dass Dienste im Vergleich zu anderen Individuen der realen Welt grundsätzlich andere Eigenschaften besitzen, sodass sie nicht über die typischerweise vorhandenen, generischen Elemente einer Ontologiesprache erfasst werden können, sondern neue, dienstspezifische Elemente eingeführt werden müssen.

Eine Untersuchung grundsätzlicher Charakteristika von Diensten führt zu folgender Liste:

- (C1) Im Gegensatz zu anderen Individuen der realen Welt ist der Zweck eines Dienstes eine *Wirkung* und somit gezielte Änderung der realen Welt.
- (C2) Im Gegensatz zu anderen Individuen der realen Welt ist die Nutzung eines Dienstes mit einer *Konfigurierung* und *Instanziierung* verbunden.
- (C3) Im Gegensatz zu anderen Individuen der realen Welt werden Dienste in *dynamischen Umgebungen* verwendet, d.h. die (einmalige) Aufstellung und (mehrfache) Verwendung einer Beschreibung für einen Dienst findet nicht zur gleichen Zeit und nicht im gleichen Kontext statt.

Der Ansatz dieser Arbeit besteht darin, die Besonderheiten bei der Erfassung von Diensten durch eine spezielle neue Dienstbeschreibungssprache umzusetzen, die nicht nur eine weitere Dienstontologie darstellt, sondern auch die zugrunde liegende Ontologiesprache anpasst (siehe Abbildung 4.1). Dies geschieht einerseits dadurch, dass

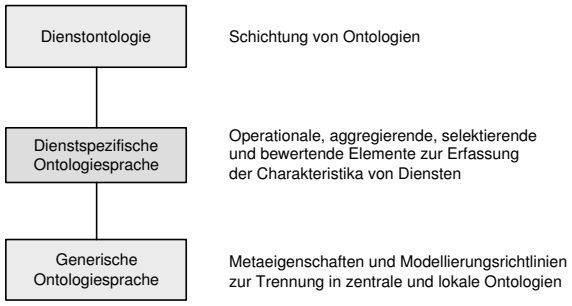


Abbildung 4.1.: Übersicht über den eigenen Ansatz.

zwischen der generischen Ontologiesprache und der konkreten Dienstontologie eine *dienstspezifische Ontologiesprache* mit neuen, zur Beschreibung von Diensten essentiellen Beschreibungselementen eingefügt wird. Andererseits wird die generische Ontologiesprache so angepasst, dass sie auch in den für Dienste typischen dynamischen Umgebungen verwendet werden kann.

Die fundamentalen Eigenschaften C1 und C2 werden in der **dienstspezifischen Ontologiesprache** erfasst, indem vier neue Elementtypen eingeführt werden:

- Der Zweck eines Dienstes besteht darin, eine Operation durchzuführen, d.h. eine Änderung in der realen Welt zu bewirken. Da die Ontologie die Welt beschreibt, muss eine Dienstbeschreibung folglich darstellen, wie sich die Operation in der Ontologie (insbesondere an den Instanzen darin) widerspiegelt. Aus diesem Grund müssen in der Ontologiesprache *operationale Elemente* zur Verfügung stehen, welche die Änderung erfassen können.

- Angebotene Dienste können mehrfach instanziiert werden und sind so Lage, mehrere ähnliche Effekte zu erbringen, andererseits sind Dienstnehmer bei der Anfrage nach Diensten häufig nicht nur mit genau einem Effekt zufrieden. Aus diesem Grund müssen *aggregierende Elemente* zur Ontologiesprache hinzugefügt werden, mit welcher die Sammlung von Effekten ausgedrückt werden kann.

- Angebotene Dienste sind in der Regel konfigurierbar und erlauben es dem Dienstnehmer, die tatsächlich zu erwirkenden Effekten vor der Dienstnutzung auszuwählen oder zumindest einzuschränken. Aus diesem Grund müssen *selektierende Elemente* in die Ontologiesprache integriert werden, mit welcher die Auswahlmöglichkeiten einzelner oder mehrerer Effekte erfasst werden können.

- Nicht alle Effekte, die ein Dienst erwirken kann, sind für einen Dienstnehmer gleich geeignet. Aus diesem Grund müssen *bewertende* Elemente in die Ontologiesprache integriert werden, mit welcher die Präferenzen des Dienstnehmers unter mehreren Effekten ausdrückbar werden.

Die Umsetzung der fundamentalen Eigenschaft C3 ist schwieriger, da sie nicht durch zusätzliche Elemente abgedeckt werden kann. Um den Einsatz von Dienstbeschreibungen auch für dynamische Umgebungen möglich zu machen, ist es nötig, dass diese *unabhängig voneinander erstellbar* sind. Nur so kann eine Dienstbeschreibung zum Zeitpunkt t aufgestellt und zu einem späteren Zeitpunkt bei veränderter Dienstlandschaft genutzt werden. Idee ist es daher, eine gemeinsame Ontologie zu verwenden, die eine *ordnende Wirkung* auf die unabhängige Erstellung von Beschreibungen ausübt. Da die zu beschreibende Welt jedoch umfangreich und zudem zeitlich variabel ist, wäre der Aufwand zur Erstellung und Wartung einer solchen Ontologie immens.

Der Ansatz dieser Arbeit löst das Problem durch einen Kompromiss zwischen ordnender Wirkung und Einigungsaufwand, indem die Ontologie in einen statischen und mehrere dynamische Teile zerlegt wird. Die statischen Teile werden in einer einheitlichen zentralen Ontologie verwaltet; für die dynamischen Teile erfolgt die Verwaltung lokal und ohne Einigung. Auf Seiten der **generischen Ontologiesprache** werden daher Anpassungen nötig, die eine solche Aufteilung unterstützen. Dies geschieht einerseits durch Einbringen von Modellierungsrichtlinien, die dafür sorgen, dass die zentrale Ontologie möglichst stabil und langlebig ist und somit ein nur geringer Wartungsaufwand anfällt, andererseits durch Bereitstellen von Metaeigenschaften, mit denen Konzepte der Ontologie annotiert werden müssen, um daraus später Wissen über nicht-sichtbare Konzepte in lokalen Ontologien ableiten zu können.

Die **Dienstontologie** muss dieser neuen Ontologiesprache Rechnung tragen. Bei der Erstellung eines Schemas für Dienstbeschreibungen muss sie sowohl klassische Elemente aus der generischen Ontologiesprache als auch neue Elemente aus der dienstspezifischen Ontologiesprache kombinieren.

Für die Ontologiesprache wird zudem eine **formale Semantik** definiert, die insbesondere auch die neuen, dienstspezifischen Elemente erfasst. Auf deren Basis kann dann ein **Vergleicher** konzipiert werden, mit dessen Hilfe Beschreibungen effektiv (d.h. für eine anschließende automatische und semantisch korrekte Nutzung) und effizient (d.h. mit vertretbarem Aufwand) verarbeitet werden können.

Der übergeordnete Ansatz dieser Arbeit ist daher:

> **EIGENER ANSATZ:**
> Erstelle eine neue Ontologiesprache mit formaler Semantik, welche durch die Einführung dienstspezifischer Elemente, Metaeigenschaften und Modellierungsrichtlinien speziell auf die fundamentalen Charakteristika von Diensten abgestimmt ist und definiere auf deren Basis eine Dienstontologie sowie effiziente und effektive Vergleichsoperatoren.

Im weiteren Verlauf des Kapitels wird die Vorgehensweise beim Umsetzen des Ansatzes überblicksartig vorgestellt. Die Details der Umsetzung finden sich dann in den Kapiteln 5 bis 9. Es wird in folgenden Schritten vorgegangen:

1. In Abschnitt 4.1 wird zunächst das **konzeptionelle Modell** der Arbeit vorgestellt, in dem die für den Ansatz wichtigen Begriffe definiert und in Beziehung gesetzt werden.

2. Die Konzepte bei der Entwicklung der **generische Ontologiesprache** *DIANE Elements I* (DE-I) werden in Abschnitt 4.2 vorgestellt. Die Syntax zu DE-I findet sich in Kapitel 5.

3. Die Einführung der neuen Sprachelemente der **dienstspezifischen Ontologiesprache** als *DIANE Elements II* (DE-II) erfolgt in Abschnitt 4.3. Hiermit sollen die operationalen, aggregierenden, selektierenden und bewertenden Eigenschaften darstellbar werden. Die Details zu DE-II finden sich in Kapitel 6.

4. Die Einführung der **Dienstbeschreibungssprache** *DIANE Service Description* (DSD) erfolgt in Abschnitt 4.4. Beschreibungen in dieser Sprache fußen auf einem gemeinsamen Vokabular aus DE-I und den neuen Sprachelementen aus DE-II. Die Details zu DSD finden sich in Kapitel 7.

5. Kapitel 8 definiert die formale, axiomatische **Semantik** für die Konzepte aus DE-I, DE-II und DSD. Insbesondere die dienstspezifischen Elemente werden erfasst.

6. Verfahren zum **Vergleich** von DSD-Beschreibungen werden dann in Abschnitt 4.5 vorgestellt. Der Vergleich arbeitet exakt gemäß der formalen Semantik und liefert Informationen, die eine automatisierte und semantisch korrekte Dienstnutzung ermöglichen. Die Details zum Vergleichen finden sich in Kapitel 9.

4.1. Konzeptionelles Modell des Ansatzes

Dieser Abschnitt stellt das konzeptionelle Modell des Ansatzes dar. Hierin werden die im Ansatz verwendeten Begriffe eingeführt. Dies ist wichtig, um die Entwurfsentscheidungen und die Semantik der zu entwickelnden Dienstbeschreibungssprache auf eine definierte und nachvollziehbare Basis zu stellen. Als Grundlage für das Modell dienen Arbeiten von Preist [123], Dumas et al [34], Keller/Kifer et al. [68, 69] und des W3C [18].

Angebotene Funktionalität

Zunächst werden die Begriffe bezüglich angebotener Funktionalität eingeführt. Eine Übersicht über diese Begriffe in Form eines UML-Klassendiagramms findet sich in Abbildung 4.2.

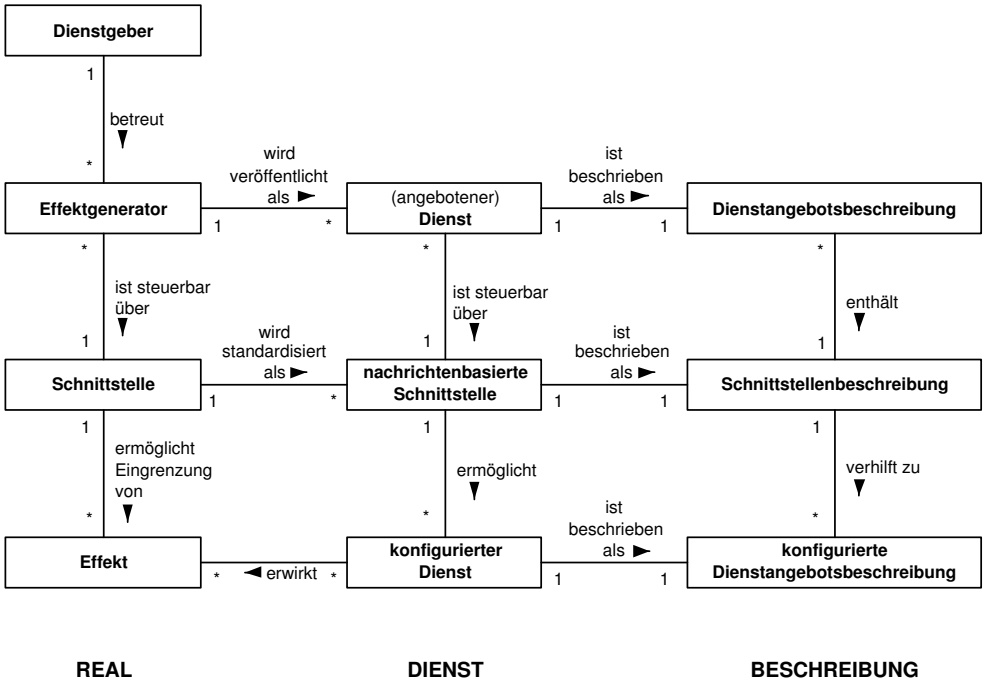


Abbildung 4.2.: Begriffe bezüglich angebotener Funktionalität und deren Zusammenhang.

Wichtigstes Konzept ist der *Effekt*. Diesem liegt zugrunde, dass bestimmte Aktionen den Zustand der Welt verändern können, d.h. einen Effekt erwirken können. Es gilt:

Definition 4.1.1 [Effekt]
Ein Effekt *ist eine einzelne Wirkung in der realen oder virtuellen Welt, welche durch die Ausführung einer Reihe zusammengehöriger Bearbeitungsschritte erreicht wird. Hierdurch wird ein neuer Zustand erreicht.*

Beispiele für Effekte können daher sein:

- Person p_1 erfährt den Namen des Autors, der das Buch mit dem Titel x geschrieben hat.
- Person p_2 erhält Zugriff auf ein Dokument, welches den Wetterbericht für den Tag t am Ort o enthält.
- Person p_3 erlangt Besitz über eine Kinokarte für die Vorstellung um x in Kino y.
- Person p_4 wird von 16 bis 17 Uhr in die Lage versetzt, mit Person p_5 zu chatten.

Unterschieden werden daher Effekte, deren erwirkter Zustand wie in den Beispielen a) bis c) *persistierend* ist, d.h. so lange anhält, bis er durch einen anderen Effekt verändert wird, oder wie im Beispiel d) *temporär* ist, d.h. auch ohne Fremdeinwirkung nach einer gewissen Zeit endet. Wie zu sehen, muss sich ein Effekt nicht zwangsläufig wie in Beispiel c) auf die physische Welt auswirken, sondern kann einen Wissenszustand verändern (Beispiel a) oder Wirkungen in der virtuellen Welt des Informationssystems haben (Beispiele b und d).

Effekte werden von *Effektgeneratoren* erbracht:

Definition 4.1.2 [Effektgenerator]
Ein Effektgenerator *bezeichnet eine Komponente, die in der Lage ist, eine Reihe von Effekten zu erbringen. Ein Effektgenerator kann daher ein Computerprogramm, eine Maschine oder ein Mensch sein.*

Beispiele für Effektgeneratoren könnten sein

- Ein Computerprogramm, das Währungsbeträge umrechnen kann.
- Eine Webanwendung, mit deren Hilfe man Informationen über die Wetterbedingungen in Europa erhalten kann.
- Eine Webseite, auf der man Flüge buchen kann.
- Ein Drucker.
- Ein Techniker, der in der Lage ist, Telefonanschlüsse im Bereich Karlsruhe freizuschalten.

Effektgeneratoren werden von einem menschlichen *Dienstgeber* betreut und haben für gewöhnlich eine *Schnittstelle*, über die gleichzeitig der zu erbringende Effekt ausgewählt, eingegrenzt und angestoßen werden kann. Die Schnittstelle ist je nach Typ des Effektgenerators von einem Rechner direkt oder indirekt aufrufbar (wie z.B. im Falle einer Webanwendung oder eines Druckers) oder nicht (wie z.B. im Falle des Technikers). Der Auswahl eines bestimmten Effektes über die Schnittstelle wird als *Konfiguration* des Effektgenerators bezeichnet.

Ist ein Effektgenerator in der Lage Effekte zu generieren, die für andere Teilnehmer nützlich sein könnten, so besteht die Möglichkeit, ihn öffentlich verfügbar zu machen. Hierzu muss der Besitzer bzw. Verwalter des Effektgenerators diesen durch eine *nachrichtenbasierte Schnittstelle* ansteuerbar machen. Eine solche Schnittstelle unterscheidet sich von der direkten Schnittstelle des Effektgenerators dadurch, dass sie einerseits rechnergestützt verwendet werden kann und die auszutauschenden Nachrichten in dem Format ausgetauscht werden müssen, auf das sich die Teilnehmer geeinigt haben.

Definition 4.1.3 [Nachrichtenbasierte Schnittstelle]
Eine nachrichtenbasierte Schnittstelle *ist eine Schnittstelle, die rechnergestützt verwendet werden kann und deren Aufruf über Nachrichten erfolgt, auf deren grundlegendes Format sich die teilnehmenden Einheiten geeinigt haben.*

Beispiele für Technologien, die verwendet werden, um nachrichtenbasierte Schnittstellen bereitzustellen, sind die von Webdiensten bekannten Sprachen WSDL und SOAP (siehe Abschnitt 2.1) sowie ältere Schnittstellenbeschreibungssprachen wie sie aus CORBA oder DCOM bekannt sind.

Ein Effektgenerator, für den eine nachrichtenbasierte Schnittstelle zur Verfügung steht, kann als *Dienst* angeboten werden. Ein solcher angebotener Dienst kann durch Austausch von Nachrichten öffentlich genutzt werden.

Definition 4.1.4 [Dienst]
Ein Dienst *ist ein Effektgenerator, dessen Schnittstelle über eine nachrichtenbasierte Schnittstelle öffentlich zugänglich gemacht wurde. Andere Teilnehmer können den Dienst und damit den Effektgenerator konfigurieren und anstoßen, indem sie standardisierte Nachrichten austauschen.*

Über die Schnittstelle eines Dienstes kann also durch Senden von Nachrichten der zugrunde liegende Effektgenerator so konfiguriert werden, dass dieser auf keinen, einen oder mehrere erbringbare Effekte eingeschränkt wird. Einen solchen Dienst nennt man *konfigurierten Dienst*. Im ersten Fall lehnt der Dienst die Ausführung mit einer Fehlermeldung ab, im zweiten und dritten Fall wählt der Dienst bzw. der Effektgenerator selbstständig einen den Einschränkungen genügenden Effekt aus, erbringt diesen und versendet Informationen hierüber in Form von Nachrichten.

Die Schnittstelle eines Dienstes hat demnach zwei Aufgaben. Durch Senden von Nachrichten an die Schnittstelle kann der Dienst konfiguriert werden (*Konfigurationsmöglichkeiten*), durch Empfangen von Nachrichten von der Schnittstelle nach Ausführung des Dienstes können Informationen über den tatsächlich erbrachten Effekt erlangt werden (*Informationsmöglichkeiten*).

Um einen Dienst öffentlich verwendbar zu machen, ist es nötig, in einem Dokument festzuhalten, welche Funktionalität er über seine Schnittstelle bereitstellt. Eine solche *Dienstangebotsbeschreibung* hat daher das Ziel, die Effektmöglichkeiten und die dazu nötigen Konfigurationen darzulegen. Wichtigster Teil ist daher die Beschreibung der Menge der erbringbaren Effekte sowie die Konfigurations- und Informationsmöglichkeiten.

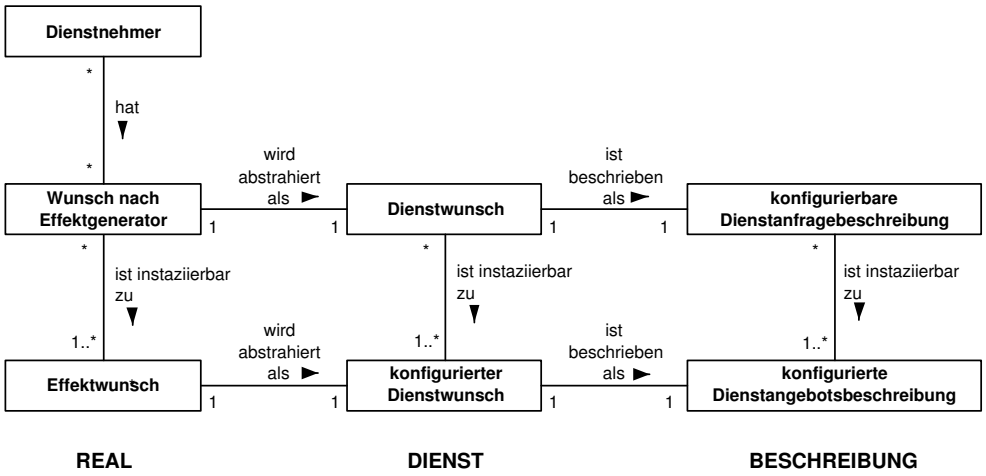


Abbildung 4.3.: Begriffe bezüglich benötigter Funktionalität und deren Zusammenhang.

Definition 4.1.5 [Dienstangebotsbeschreibung]
Eine Dienstangebotsbeschreibung *ist ein Dokument, welches die von einem Dienst erbringbaren Effekte und die damit verbundenen ausgegebenen Nachrichten in Abhängigkeit von der Konfiguration über die Schnittstelle so beschreibt, dass andere Teilnehmer den Dienst in ihrem Sinne verwenden können, ohne den zugrunde liegenden Effektgenerator betrachten zu müssen.*

Benötigte Funktionalität

Im Folgenden werden die Begriffe bezüglich benötigter Funktionalität definiert. Eine graphische Übersicht gibt Abbildung 4.3.

Neben Teilnehmern, die Funktionalität ihrer Effektgeneratoren als Dienste bereitstellen, existieren Teilnehmer, die zur Erreichung ihrer Ziele an einem oder mehreren Effekten interessiert sind. Bei ihnen existiert ein *Effektwunsch*. In der Hoffnung diesen durch einen öffentlich angebotenen Dienst erzielen zu können, wird dieser in Form einer *Dienstanfragebeschreibung* beschrieben, welche die geeigneten Effekte sowie den zugehörigen Informationsbedarf enthält. Diese Dienstanfragebeschreibung wird dann in Auftrag gegeben.

Für benötigte Funktionalität muss unterschieden werden, ob diese spontan benötigt und sofort erbracht werden soll, oder ob diese als Bestandteil einer größeren Anwendung später, evtl. auch mehrmals in unterschiedlichen Varianten erbracht werden soll. Ziel im ersten Fall ist die direkte Erzielung eines Effekts, im zweiten Fall die Bereitstellung eines *virtuellen Diensts*, der später durch entsprechende Konfiguration die

jeweils von der Anwendung benötigten Effekte durch Aufruf korrekt konfigurierter angebotener Dienste erbringen kann.

Definition 4.1.6 [Dienstanfragebeschreibung]
Eine Dienstanfragebeschreibung *ist ein Dokument, welches die von einem Teilnehmer gewünschten Effekte beschreibt. Man unterscheidet*

- *Spontane Anfragen, d.h. Anfragen, die auf die direkte Erzielung eines einzelnen Effekts abzielen. In solchen Fällen handelt es sich um (konfigurierte) Dienstanfragebeschreibungen.*
- *Anfragen nach virtuellen Diensten, d.h. Anfragen, die auf die Bereitstellung eines virtuellen Dienstes abzielen. Eine solche Beschreibung kann später mehrfach unterschiedlich konfiguriert werden und zur Erreichung der benötigten Effekte beitragen. Solche Anfragen bezeichnet man als konfigurierbare Dienstanfragebeschreibungen.*

Der Teilnehmer kann also je nach Art der Anfrage ein menschlicher Benutzer oder eine Applikation sein. Er wird in jedem Fall als *Dienstnehmer* bezeichnet.

Definition 4.1.7 [Dienstnehmer]
Ein Dienstnehmer *ist ein Teilnehmer, der mittels einer Dienstanfragebeschreibung nach geeigneter Funktionalität sucht. Ein Dienstnehmer kann ein menschlicher Benutzer sein, der die Anfrage spontan stellt, oder eine Applikation sein, in der eine konfigurierbare Anfrage hinterlegt ist, welche bei einem konkreten Effektwunsch konfiguriert und in Auftrag gegeben wird.*

Menschliche Teilnehmer an einem dienstorientierten System lassen sich in drei Gruppen aufteilen, die sich dahingehend unterscheiden, wann und auf welche Art sie zur Realisierung des Gesamtsystems beitragen:

- *Endanwender* verwenden den Geschäftsprozess oder die (mobile) Anwendung *zur Laufzeit* als Blackbox, die sie meist über eine webbasierte oder graphische Oberfläche bedienen. Für sie geschieht die Verwendung externer Dienste transparent, das heißt für sie scheint alle Funktionalität lokal verfügbar zu sein. Im optimalen Fall ist die Dienstorientierung daher zur Laufzeit nicht zu erkennen.
- *Anwendungsentwickler* implementieren Geschäftsprozesse oder mobile Anwendungen durch Nutzung oder Veröffentlichung externer Funktionalität in Form von Diensten. Ausgehend von den konkreten Anforderungen in der zu entwickelnden Applikation erstellen sie daher *zur Entwurfszeit* Dienstbeschreibungen, welche sie in die Anwendung einbetten.

- *Domänenexperten* sind Spezialisten in einem Anwendungsgebiet. Zwar haben sie eine Übersicht über existierende Dienstangebote und -wünsche in ihrem Gebiet, kennen jedoch nicht jeden einzelnen Dienst im Detail. Sie einigen sich mit anderen Domänenexperten auf eine gemeinsame, dienstunabhängige Beschreibung ihres Anwendungsgebiets.

4.2. Generische Ontologiesprache DE-I

Um zu einer semantisch sinnvollen und automatischen Verarbeitung von Dienstbeschreibungen zu gelangen, ist es notwendig, dass sich die Teilnehmer auf ein gemeinsames Vokabular einigen. Wie schon in Abschnitt 2.2.2 beschrieben, überbrückt eine Ontologie die semantische Lücke zwischen realer Welt und dem Informationssystem. Ihre Aufgabe ist daher die Schaffung eines geteilten Verständnisses der Konzepte, um den Teilnehmern zu ermöglichen, sich durch Bezug auf diese Konzepte semantisch eindeutig auszutauschen. Dies ist insbesondere in den für Webdiensten besonders wichtigen dynamischen Umgebungen von Bedeutung, da die Dienstbeschreibungen hier unabhängig voneinander erstellt werden müssen und einzig die gemeinsame Ontologie[1] eine ordnende Wirkung ausüben kann.

Problematisch bei der Erstellung einer gemeinsamen Ontologie ist jedoch, dass sich die zu erfassenden Konzepte verändern können (insbesondere auch durch die Wirkung von Diensten) und kontextabhängige Eigenschaften besitzen (die insbesondere durch die Verteilung von Dienstnehmern und Dienstgebern ausgelöst werden). Eine Einigung auf ein zentrales, vollständiges und zu jedem Zeitpunkt konsistentes Abbild der Welt ist daher sehr aufwändig und nicht skalierbar. Zudem ist durch häufige Anpassungen der Ontologie die ordnende Wirkung auf unabhängig erstellte Dienstbeschreibungen hinfällig.

Um den Konflikt zwischen unabhängiger Erstellbarkeit und akzeptablen Einigungsaufwand zu lösen, wird in dieser Arbeit mit *DIANE Elements I* (DE-I) eine generische Ontologiesprache vorgestellt, die den Ansatz verfolgt, statische und dynamische Ontologieteile zu trennen: Die Beschreibung der Welt zerfällt somit in eine gemeinsam verwaltete, zentrale Ontologie bestehend aus Schema und Instanzen und mehrere lokal verwaltete Ontologien, die nur aus privaten Instanzen bestehen. Die generische Ontologiesprache muss diesem Umstand Rechnung tragen, indem sie einerseits Modellierungsregeln vorgibt, andererseits entsprechende Konstrukte und Metaeigenschaften zur Verfügung stellt, um die Trennung umsetzen zu können.

[1] Unter Ontologie wird im Folgenden stets sowohl das Schema als auch die Instanzen verstanden.

Zentrale Ontologie

Ziel der zentralen Ontologie ist es, ein gemeinsam verwaltetes Vokabular bereitzustellen und so eine ordnende Wirkung auf unabhängig erstellte Dienstbeschreibungen auszuüben. Eine solche Ontologie strebt daher vor allem nach Langlebigkeit und Stabilität, versucht also die Anzahl der Änderungen im Laufe der Zeit zu minimieren. Da sie auch als Grundlage für bisher noch unbekannte Dienste zur Verfügung stehen soll, muss sie möglichst kontextfrei und anwendungsunabhängig sein, um nicht im Voraus bestimmte Dienste auszuklammern.

In DE-I werden diese Eigenschaften erreicht, indem spezielle Anforderungen an die zentrale Ontologie gestellt und mit neuen Konstrukten durchgesetzt werden. Allgemein zielen sie darauf ab, die Bestandteile der Ontologie möglichst gut gegen Änderungen zu schützen. Im Einzelnen sind das:

EIGENSCHAFT 4.1
Intuitive Modellierung

Um eine stabile zentrale Ontologie zu erhalten, sollte die Ontologiesprache auf einem Ordnungsmechanismus beruhen, der von den Teilnehmern intuitiv verwendet werden kann. Als Ordnungsmechanismus dient daher bei DE-I Objektorientierung. Begründet ist dies durch die weite Verbreitung und Akzeptanz dieses Paradigmas und die Möglichkeit einer natürlichen Modellierung. Mit der Entscheidung für Objektorientierung ist eine Trennung in Schema und Instanzen verbunden, die auch in DE-I durchgeführt wird. Die Instanzen stehen dabei stellvertretend für Individuen der realen Welt, während das Schema das Anwendungsgebiet strukturiert, indem artgleiche Instanzen zu Klassen zusammengefasst werden.[2] Im Schema sind zudem über Attribute die Beziehungsmöglichkeiten der Instanzen untereinander festgehalten. Eine besondere Beziehung stellt die Vererbung dar, mit der eine is-a-Semantik ausgedrückt werden kann.

EIGENSCHAFT 4.2
Einheitliches, öffentliches, modularisiertes Schema

Für die ordnende Wirkung einer Ontologie sorgt vor allem ein zentrales Schema. Es sollte für alle Teilnehmer der Anwendergruppe einheitlich und zentral verfügbar sein und spiegelt somit die Einigung innerhalb der Gruppe wider. Lokale Änderungen oder Erweiterungen am Schema sind zwar denkbar, bringen jedoch erhebliche Probleme bei unabhängig erstellten Dienstbeschreibungen und werden daher in dieser

[2] In der Praxis erweist sich die Unterscheidung zwischen Klasse und Instanz nicht immer als trivial. Siehe dazu auch [152].

Arbeit nicht betrachtet. Änderungen und Erweiterungen am gemeinsamen Schema dürfen nur nach Absprache mit der Anwendergruppe durchgeführt werden. Beruhten die Dienstbeschreibungen auf mehreren unterschiedlichen Schemata, wäre eine wiederholte Anpassung der Beschreibungen vor jedem Vergleich nötig, die in der Regel jedoch nur halbautomatisch geschehen kann. Eine einmalige Einigung auf ein gemeinsames Schema zu Beginn sowie eine fortlaufende, möglichst minimale Wartung während der Laufzeit des Systems ist daher weitaus effizienter. Die Wartbarkeit wird durch die Modularisierung der Ontologie in disjunkte Teilontologien erleichtert.

EIGENSCHAFT 4.3
Anwendungsneutrales Schema

Bei dienstorientierten Architekturen wird von einer variablen Dienstlandschaft ausgegangen, die beim Aufstellen der konkreten Dienstbeschreibungen zur Entwurfszeit noch unbekannt ist. Aus diesem Grund müssen die Schemata der Teilontologien zwar domänenspezifisch sein, dabei aber neutral gegenüber konkreten Diensten, Anwendungen, Implementierungen und lokalen Datenmodellen bleiben. Die zentrale Ontologie ist daher eine Beschreibung der Welt unabhängig vom späteren Verwendungszweck, um nicht bereits auf bestimmte Fälle einzuschränken. Aus diesem Grund enthält sie insbesondere keine Beschreibungen von Nachrichten (wie PhoneOrderingMessage), keine Beschreibungen von verarbeitenden Klassen (wie AvailabilityChecker), keine programmiersprachlichen Konzepte (wie PhoneHashtable) und keine Methoden. Eine solche Ontologie kann dann zur Beschreibung unterschiedlicher und vorher unbekannter Dienste genutzt werden sowie von Domänenexperten ohne Programmierkenntnisse erstellt und gewartet werden.

EIGENSCHAFT 4.4
Unterscheidung in öffentliche und private Instanzen

Da Instanzen für Individuen der realen Welt stehen, existieren sehr viele Instanzen. Viele davon sind jedoch nur von lokalem Interesse, z.B. nur innerhalb eines Dienstes. In DE-I ist es daher möglich, neben *öffentlichen* auch *private* Instanzen zu vereinbaren. Öffentliche Instanzen werden dabei ähnlich wie Schemaelemente von der gesamten Anwendergruppe unter global eindeutigem Namen im öffentlichen Instanzenpool festgelegt. Sie unterliegen damit dem Einigungsprozess. Dies sollte für Instanzen durchgeführt werden, die von allgemeinem Interesse sind (wie etwa alle Währungen, bestimmte Städte, bestimmte Personen etc.). Private Instanzen sind nur für einen Teilnehmer unter einem lokal eindeutigen Namen sichtbar und liegen nur in dessen privaten Instanzenpool. Für den Vergleicher von Dienstbeschreibungen ist wichtig zu wissen, ob zu einer Klasse nicht-öffentliche Instanzen existieren können, was zu einem

unerwünschten Effekt führen könnte. In DE-I werden daher Klassen mittels einer Metaeigenschaft als *öffentlich* markiert, wenn all ihre Instanzen öffentlich sind, und als *teilöffentlich markiert*, wenn es sowohl öffentliche als auch private Instanzen geben kann.

EIGENSCHAFT 4.5
Ausschließlich rigide Klassen im Schema

Da das Schema von allen Teilnehmern der Anwendergruppe geteilt wird, sollte es nur Klassen enthalten, die von allgemeinem Interesse und stabil sind, d.h. deren Objekte dauerhaft zu dieser Klasse gehören. Im Schema sollten daher nur *rigide* Konzepte enthalten sein. Nach der *OntoClean*-Methode [55] ist ein Konzept rigide, wenn dessen Elemente aufhören zu existieren, sobald sie nicht mehr zum Konzept gehören. Beispielsweise ist die Klasse Person rigide, die Klassen Student oder CompanyInKarlsruhe hingegen nicht. Nicht-rigide Konzepten sollten kritisch betrachtet werden. Es könnten wie im Falle von Student Rollen sein, die dann über Zustände zu modellieren (etwa eine Person im Zustand Studying) und in den lokalen Ontologien abzulegen sind, oder wie im Falle von CompanyInKarlsruhe Abfragen von temporärem oder lokalem Interesse sein, die dann über Mengen (siehe Abschnitt 6.1) abzubilden sind.

EIGENSCHAFT 4.6
Spezielle Behandlung extrinsischer Eigenschaften.

Für Klassen können intrinsische und extrinsische Eigenschaften unterschieden werden (siehe [36]). *Intrinsische Eigenschaften* hat eine Entität „von sich aus“. Sie sind daher in jedem Kontext anwendbar und können nicht von ihr entfernt werden, wie etwa isbn und title bei Book. *Extrinsische Eigenschaften* hingegen entstehen durch Beziehungen zu anderen Entitäten. Sie sind daher kontextabhängig gefüllt, wie etwa price oder owner bei CopyOfBook[3]. Gerade diese extrinsischen Eigenschaften werden jedoch von Diensten verändert (wie etwa ein neuer Besitzer) oder kontextabhängig abgefragt (wie etwa der Anbieter mit dem günstigsten Preis). Für die in der zentralen Ontologie abgelegten öffentlichen Instanzen ist es daher nicht sinnvoll, auch ihre aktuellen extrinsischen Eigenschaften festzuhalten. Klassen im zentralen Schema können daher zwar intrinsische und extrinsische Attribute besitzen, für Instanzen sind jedoch im zentralen Schema nur die intrinsischen Attribute gefüllt. Die extrinsischen Attribute werden in lokalen Ontologien über Zustandsklassen festgehalten (siehe Eigenschaft 4.7).

[3]Objekte von Book bezeichnen abstrakte Ausgaben eines Buches, Objekte von CopyOfBook die konkreten greifbaren Exemplare bestehend aus Papier.

Lokale Ontologien

Ziel der lokalen Ontologien ist es, die stärker veränderlichen, kontextabhängigen oder nur lokal interessierenden Teile der Weltbeschreibung zu erfassen. Da das Schema aufgrund seiner ordnenden Wirkung vollständig in der zentralen Ontologie zu finden ist, enthalten lokale Ontologien keine zusätzlichen Schemaelemente, sondern ausschließlich *private Instanzen*. Zudem muss die Möglichkeit bestehen, die extrinsischen (und damit häufig veränderlichen und kontextabhängigen) Eigenschaften von öffentlichen und privaten Instanzen zusammen mit ihrem Kontext abzulegen.

Da Dienstbeschreibungen auch private Instanzen beinhalten können, ist es wichtig, dass ein Vergleicher auch dann noch gewisse Rückschlüsse über diese unbekannten Instanzen ziehen kann, wenn er diese nicht explizit beschaffen kann. Es sollte daher grob ersichtlich sein, welche Instanzen zu einer Klasse gehören könnten und welche nicht.

In DE-I werden all diese Eigenschaften erreicht, indem zusätzliche Konstrukte und Metaeigenschaften eingeführt werden. Im Detail sind das:

EIGENSCHAFT 4.7
Extrinsische Eigenschaften über Zustandsklassen verdinglicht.

Um die extrinsischen Eigenschaften von Instanzen lokal ablegen zu können, existiert in DE-I das Konzept der Zustandsklassen. Von diesen werden lokale Instanzen erzeugt, um extrinsische Eigenschaften einer Instanz auszudrücken. Zustandsklassen stellen somit verdinglichte extrinsische Eigenschaften dar. Diese Zustandsklasse enthält dann den eigentlichen Wert, den Kontext und einen Verweis auf die beschriebene Entität. Beispielsweise entsteht aus dem extrinsischen Attribut price die Zustandsklasse Priced mit den Attributen price (der eigentliche Wert), vendor (der Kontext) und entity (als Verweis auf die beschriebene Entität). In der ursprünglichen Klasse wird die extrinsische Eigenschaft dann durch ein *orthogonales Attribut* mit dem Zieltyp der Zustandsklasse ersetzt. Orthogonale Attribute werden in DE-I mithilfe einer Metaeigenschaft explizit gekennzeichnet. Sie können nicht aus den intrinsischen Attributen abgeleitet werden (daher orthogonal) und sind in der zentralen Ontologie nie gefüllt (siehe Eigenschaft 4.6). Sie dienen nur als Hinweis auf mögliche Zustandsklassen.

EIGENSCHAFT 4.8
Markierung von wertbestimmten und Entitätsklassen;
Unterscheidung in anonyme und benamte Instanzen

Klassen müssen dahingehend unterschieden werden, ob jede Kombination von Attributfüllwerten ein existierendes Individuum der realen Welt repräsentiert oder ob

es Kombinationen gibt, die kein Individuum repräsentieren. Beispielsweise könnte ein Dienst als Eingabe einen Preis (bestehend aus Betrag und Währung) verlangen. Hierbei führen alle Kombinationen zu gültigen Preisen. Ein anderer Dienst könnte als Eingabe ein Telefonmodell (durch Angabe von Name und Hersteller) verlangen. Hier sind nur bestimmte Kombinationen von Füllwerten gültig. Bei der Konfiguration von angebotenen Diensten muss dies beachtet werden. Betrachtet man für eine Klasse den durch ihre Attribute aufgespannten Raum aller möglicher Instanzen, so spricht man von einer *wertbestimmten Klasse*, wenn jeder Punkt im Raum ein eigenes, existierendes Individuum repräsentiert. Die Gleichheit von Individuen ist dann einfach über die Gleichheit der Attributkombination feststellbar. Ein Beispiel hierfür ist Price. Eine Benennung der Instanz ist nicht nötig, weshalb wertbestimmte Klassen nur *anonyme Instanzen* besitzen. Man spricht von einer *Entitätsklasse*, wenn nicht alle Punkte des Raums existierende Individuen repräsentieren und auch mehrere Punkte dasselbe Individuum beschreiben können. Die Individuen stellen somit eigenständigen Entitäten dar, deren Attribute sich im Laufe der Zeit leicht ändern können, ohne dass die Identität verloren geht. Die Gleichheit solcher Instanzen ist komplex und wird über eine zusätzliche Benamung der Instanzen definiert. Ein Beispiel hierfür ist PhoneType. Entitätsklassen haben daher nur *benamte Instanzen*.

EIGENSCHAFT 4.9
Markierung von definierenden und ableitbaren Attributen

Attribute können für Entitätsklassen von unterschiedlicher Bedeutung sein. Als Beispiel dienen die Attribute isbn und author von Book. Durch Ersteres wird ein Buch eindeutig selektiert, durch Zweiteres nicht. Zur korrekten Auswahl eines Dienstes ist dieses Wissen jedoch zwingend erforderlich, insbesondere dann, wenn von privaten Instanzen auszugehen ist. Für den Vergleicher ist daher eine Unterscheidung in *definierende* und *ableitbare Attribute* unerlässlich. Die definierenden Attribute sind eindeutig. Durch sie ist eine Entität eindeutig spezifiziert, insbesondere ist der Name und die Füllwerte der ableitbaren Attribute festgelegt. In DE-I werden sie durch Metaeigenschaften explizit gekennzeichnet.

4.3. Dienstspezifische Ontologiesprache DE-II

Im zweiten Schritt wird die generische Ontologiebeschreibungssprache DE-I um neue dienstspezifische Sprachelemente zu *DIANE Elements II* (DE-II) erweitert, mit deren Hilfe sich die Besonderheiten von Diensten beschreiben lassen. Unterschieden werden operationale, aggregierende, selektierende und bewertende Elemente. Während die Semantik der Konstrukte in DE-I explizit dienstneutral ist, haben die neuen Elemente eine dienstspezifische Semantik, sind also nur sinnvoll innerhalb von Dienstbeschreibungen zu verwenden. Die Details zu DE-II finden sich in Kapitel 6.

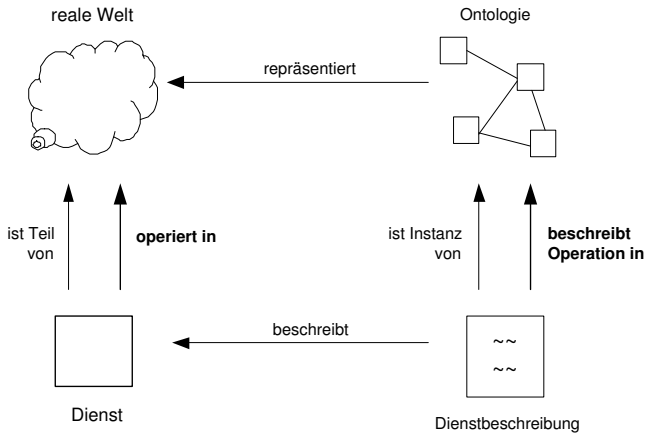


Abbildung 4.4.: Zusammenhang zwischen realer Welt, Ontologie, Dienst und Dienstbeschreibung.

Operationale Elemente

Eine fundamentale Eigenschaft von Diensten, die sie von anderen Individuen der realen Welt wesentlich unterscheidet, ist ihr Zweck zur Operation in der Welt, d.h. die Welt von einem Ausgangszustand in einen neuen Zustand zu transformieren. Dass dies Auswirkungen auf die Beschreibungssprache haben muss, wird in Abbildung 4.4 deutlich, die den Zusammenhang zwischen vier für semantische Dienstbeschreibungen wesentlichen Konzepten zeigt: *Ontologien* stellen eine rechnerverarbeitbare Beschreibung der *realen Welt* dar. Sie bestehen in DE-I aus einem Schema zur Strukturierung der Domäne und Instanzen, welche reale Individuen der Welt repräsentieren. *Dienste* stellen ebenfalls einen Teil der realen Welt dar. Demnach sollte auch die *Dienstbeschreibung* ein Teil der Ontologie sein. Konkret heißt das, die Beschreibung eines Dienstes als Instanz eines Dienstschemas anzugeben. Andererseits operieren Dienste in der Welt. Da die Ontologie die Welt beschreibt, muss eine Dienstbeschreibung folglich auch darstellen, wie sich die Operation in der Ontologie (insbesondere an den Instanzen darin) widerspiegelt. In DE-II werden dazu zwei *operationale Sprachelemente* eingeführt:

- <<precondition>> *o*. Dient zur Beschreibung eines Ausgangszustandes. Das Element hat innerhalb einer Dienstbeschreibung folgende Semantik: Unmittelbar vor Ausführung des beschriebenen Dienstes muss bzw. wird das durch den Operand *o* beschriebene Individuum existieren.[4] Ist dies nicht der Fall, bleibt der Ausgang der Dienstausführung undefiniert. Beispielsweise kann die

[4]Mit der Einführung aggregierender Elemente im nächsten Abschnitt wird auch die Verwendung deklarativer Mengen als Operand möglich.

Durchführung vom Dienstgeber abgelehnt werden oder ein fehlerhaftes Ergebnis liefern. Das Konstrukt hat daher eine *exists*-Semantik. Als wichtigste Typen für den Operanden *o* kommen verdinglichte Zustände der Klasse State (siehe Abschnitt 4.4) sowie teilöffentliche Entitätsklassen in Frage.

- <<effect>> *o*. Dient zur Beschreibung eines Folgezustandes. Das Element hat innerhalb einer Dienstbeschreibung folgende Semantik: Unmittelbar nach der erfolgreichen Ausführung des beschriebenen Dienstes soll bzw. wird das durch den Operanden *o* beschriebene Individuum existieren, entweder indem es neu erzeugt wird oder indem ein bereits vorhandenes ersetzt wird. Das Konstrukt hat daher eine *new/replace*-Semantik. Als wichtigste Typen für den Operanden *o* kommen verdinglichte Zustände der Klasse State in Frage.

Es ist wichtig zu betonen, dass es sich bei den neuen Elementen nicht um gewöhnliche Attribute (mit Füllwerten) auf Schemaebene handelt, sondern um Operatoren (mit Operanden) auf Ebene des Metaschemas, denen eine spezielle, dienstspezifische Semantik zugewiesen wurde, welche sie exklusiv zur Beschreibung von Diensten einsetzbar macht.

Die beiden Elemente sind bewusst sehr einfach gehalten. Denkbar wären auch komplexere Operationen, die Füllwerte von Instanzen direkt eintragen, ändern oder löschen oder auch Berechnungen, Verschiebungen und Kopien von Füllwerten vornehmen. Die Beschränkung auf einfache Grundoperationen bietet jedoch zwei wichtige Vorteile: Erstens ist die Gleichheit auf den Operatoren sehr einfach zu definieren. Zwei <<effect>>-Operatoren bzw. zwei <<precondition>>-Operatoren sind gleich, wenn sie die gleichen Operanden haben. Der Vergleich ist somit wohldefiniert und kann auf den als Operanden gegebenen Instanzen (oder konfigurierbaren Mengen) durchgeführt werden. Durch diese Zerlegung der Gesamtoperation in einfache Grundoperationen und Operanden ist es nicht nötig, komplexe ontologische Operation auf Gleichheit oder Ähnlichkeit zu untersuchen. Die Evaluation des Ansatzes zeigt zudem, dass die Ausdrucksstärke ausreicht, um hiermit quasi alle realistischen Dienste beschreiben zu können. Zweitens schafft die Gegensätzlichkeit der beiden Operatoren die Grundlage für die Berechnung von Dienstkompositionen. Eine nicht erfüllte *exists*-Vorbedingung eines Dienstes könnte direkt durch einen entsprechenden *new*-Effekt eines anderen Dienstes erfüllt werden. Bei der Verwendung komplexer Operatoren ist eine solche Dienstverkettung oft nicht direkt ableitbar.

Aggregierende Elemente

Als zweite fundamentale Eigenschaft von Diensten kann angesehen werden, dass diese oft nicht nur einen einzelnen unveränderlichen Effekt anbieten, sondern eine ganze Reihe von Effektmöglichkeiten beinhalten können. Beispielsweise kann ein Druckdienst

verschiedene Dokumente in den Zustand Printed überführen, wenn er die entsprechenden Dateien erhält. Daher kann eine Dienstbeschreibung als *Familie von Effekten* angesehen werden. Solche Aggregationen kommen bei Diensten in unterschiedlichen Situationen vor:

- Wie gesehen kann ein angebotener Dienst in der Regel nicht nur genau einen Effekt erwirken, sondern verschiedene ähnliche.
- Ein angebotener Dienst wird in der Regel nicht nur mit genau einer Vorbedingung arbeiten können, sondern mit verschiedenen ähnlichen.
- Ein Dienstnehmer wird in der Regel nicht nur mit einem einzelnen Effekt zufrieden sein, sondern akzeptiert für seinen Anwendungszweck verschiedene ähnliche.

Diesem Umstand werden *aggregierende Sprachelementen* gerecht. Mit diesen Elementen wird es möglich, *deklarative Mengen* von Instanzen zu bilden, insbesondere auch Mengen von Zustandsinstanzen. Das erlaubt es, einen angebotenen sowie benötigten Dienst auf kompakte Art und Weise in seiner Gesamtheit darzustellen, ohne sich auf einen typischen Effekt oder eine typische Vorbedingung beschränken zu müssen.

Mengen stellen ein Mittelding zwischen Klassen und Instanzen dar. Klassen stehen stellvertretend für *alle* Individuen eines Typs. Allgemein wichtige Klassen werden von der Anwendergruppe bestimmt und für alle Teilnehmer festgeschrieben. Instanzen stehen stellvertretend für *ein* Individuum der realen Welt. Auch Instanzen können unter einem eindeutigen Namen öffentlich festgeschrieben werden. Mengen hingegen stehen für eine temporäre Sammlung von Instanzen, die nur für den Gebrauch innerhalb von Dienstbeschreibungen gedacht sind und daher nicht in Domänenbeschreibungen auftauchen. Sie können als Abfragen einzelner Benutzer gesehen werden, sind nur von lokalem Interesse und werden nicht öffentlich festgeschrieben.

DE-II bietet vor allem die Möglichkeit zur Definition *deklarativer Mengen*, d.h. keine explizite Festlegung der Mengenelemente durch Auflistung, sondern über eine Reihe von Bedingungen. Diese Bedingungen können jedoch nicht beliebig definiert werden, sondern müssen einer durch Konstruktoren vorgeschriebene Struktur folgen. Dies ermöglicht es, die Gleichheit und Teilmengeneigenschaft zwischen solchen Mengen effizient berechnen zu können.

Wichtig ist die Kombination zwischen operationalen und aggregierenden Elementen. Deklarative Mengen können als Operanden für <<precondition>> und <<effect>> angegeben werden und so die Ausdrucksstärke wesentlich erhöhen. Es gilt dann folgende geänderte dienstspezifische Semantik: Im Falle einer Vorbedingung wird bzw. muss ein beliebiges Element der Menge unmittelbar vor Dienstausführung vorhanden sein, im Falle eines Effektes soll bzw. wird genau ein Element der Menge durch die Dienstausführung erzeugt werden. Da ein Vergleich operationaler Elemente über den

Vergleich ihrer Operanden geschieht, wird die Überprüfung von Mengen auf Gleichheit und Teilmengenbeziehung beim Vergleich von Dienstbeschreibungen eine wesentliche Rolle spielen.

Selektierende Elemente

Die dritte fundamentale Eigenschaft von Diensten bezieht sich darauf, dass angebotene Dienste zwar in der Regel durch Aggregation mehrerer Effekte definiert sind, diese von Dienstnehmer jedoch nicht unverändert angenommen werden müssen, sondern vor der Dienstausführung *konfiguriert* werden können. Dazu empfängt der Dienst Informationen, mit deren Hilfe der erbrachte Effekt ausgewählt oder konkretisiert werden kann. Umgekehrt liefert der Dienstgeber nach Abschluss der Dienstausführung Informationen über den tatsächlich erbrachten Effekt. Benötigt werden also neue Sprachelemente, mit denen die Auswahl von Instanzen aus Mengen beschreibbar wird. Diese Auswahl kann sowohl direkt (über den Namen der Instanz) als auch indirekt (über die Eigenschaften der Instanz) geschehen. Als neues Sprachelement werden in DE-II dazu *Variablen* eingeführt.

Durch Einbringen von Variablen an bestimmten Stellen von Mengendefinitionen kann ausgedrückt werden, dass dem Dienstnehmer an bestimmten Stellen Auswahl- oder Konfigurationsmöglichkeiten zur Verfügung stehen. Mit weiteren Variablen erhält der Dienstgeber die Möglichkeit, Informationen über den erwirkten Zustand bekannt zu geben. Hierdurch wird der Einfluss der ausgetauschten Informationen semantisch eindeutig in der Beschreibung festgehalten.

Variablen werden in Kategorien eingeteilt, die ihre Semantik bestimmen. Die Unterscheidung in IN- und OUT-Variablen legt fest, wer die Variable mit einem Wert zu belegen hat. IN-Variablen sind vom Dienstnehmer zu füllen, OUT-Variablen vom Dienstgeber. Variablen können sowohl in Anfrage- als auch Angebotsbeschreibungen auftreten und haben dann folgende Bedeutung:

- IN-Variablen in Angebotsbeschreibungen (so genannte OffIN-Variablen) geben an, welche Informationen der Dienst zur korrekten Ausführung benötigt. Über diese kann der zu erbringende Effekt vom Dienstnehmer konfiguriert werden.
- OUT-Variablen in Angebotsbeschreibungen (so genannte OffOUT-Variablen) geben an, welche Informationen der Dienst nach der Dienstausführung liefert, um so den tatsächlich erbrachten Effekt genauer zu spezifizieren oder Berechnungsergebnisse zu liefern.
- IN-Variablen in Anfragebeschreibungen (so genannte ReqIN-Variablen) legen fest, welche Einstellungen an der Anfrage der Dienstnehmer variabel halten und erst bei einem konkreten Aufruf einfüllen will.

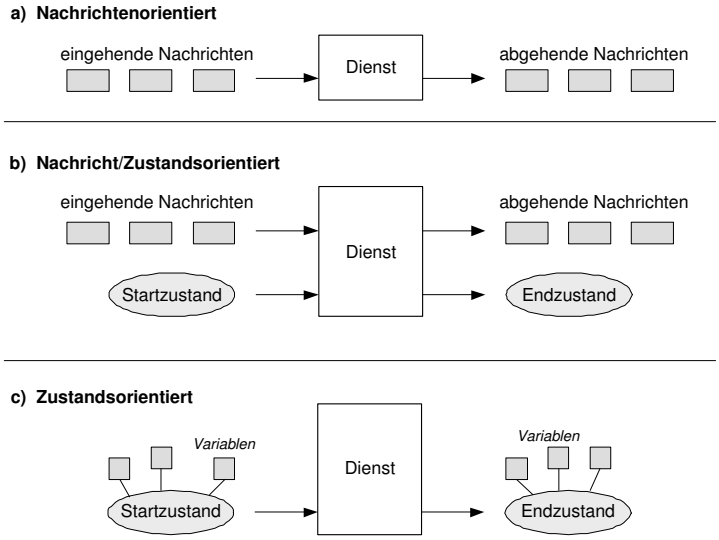


Abbildung 4.5.: Typen von Dienstbeschreibungen.

- OUT-Variablen in Anfragebeschreibungen (so genannte ReqOUT-Variablen) geben an, welche Informationen der Dienstnehmer nach Dienstausführung wissen möchte.

Durch Einbringen von Variablen in die Dienstbeschreibung wird also die tatsächliche bzw. gewünschte Informationsschnittstelle zum Dienst sichtbar. Mengen, die in ihrer Definition Variablen enthalten, werden als *konfigurierbare Mengen* bezeichnet und stellen ein zentrales Konzept des Ansatzes dar.

Insgesamt kann die funktionale Semantik eines Dienstes über eine Kombination operationaler, aggregierender und selektierender Elemente erfasst werden. Neben der eigentlichen Wirkung des Dienstes wird so auch die Konfigurationssemantik dargestellt. Ergebnis ist eine *zustandsorientierte Beschreibung* [85], d.h. eine Beschreibung, bei der die Funktionalität eines Dienstes allein über die Zustände vor und nach der Dienstausführung erfasst wird (siehe Abbildung 4.5c). Die Beschreibung dieser Zustände erfolgt dabei über konfigurierbare Mengen, wodurch im Gegensatz zu anderen Beschreibungstypen (siehe Abbildung 4.5a/b) auf eine explizite und getrennte Nachrichtenbeschreibung verzichtet werden kann. Als Vorteil wird bei diesem Ansatz der Einfluss der ausgetauschten Informationen durch die direkte Integration der Variablen klar ersichtlich, andererseits können auch Dienste zueinander passen, deren Informationsflüsse nicht direkt kompatibel sind.

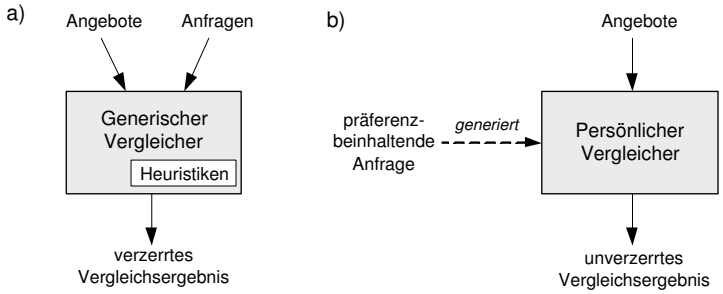


Abbildung 4.6.: Unterschied zwischen generischen und persönlichen Vergleichern.

Bewertende Elemente

Keine der betrachteten Dienstbeschreibungssprachen der Literatur ermöglicht die Integration aller Benutzerpräferenzen in eine Anfragebeschreibung (vgl. Anforderung A6). Der Dienstnehmer beschreibt daher die benötigte Funktionalität als perfekten Wunschdienst, welcher dann durch einen allgemeinen Vergleicher mit den Angebotsbeschreibungen auf Ähnlichkeit verglichen wird. Hierdurch kommt es jedoch zu einem beeinflussten Vergleichsergebnis, da der Vergleicher auf vordefinierte, dem Dienstnehmer unbekannte Heuristiken zurückgreifen muss, um Abweichungen zwischen Anfrage und Angebot bewerten zu können (siehe Abbildung 4.6a). Man stellt fest, dass der Dienstnehmer in der Regel nicht bereit ist, ein solches Vergleichsergebnis und die zugehörige automatische Auswahl des Dienstgebers ohne Rückfragen hinzunehmen, sondern er sich die besten Lösungen präsentieren lässt, sie manuell überprüft und den geeignetsten Dienstgeber auswählt.

Wie in [79, 81] ausgeführt, kann das Problem gelöst werden, wenn ein prinzipieller Unterschied zwischen Angebots- und Anfragebeschreibungen erkannt wird:

- *Angebotsbeschreibungen.* Typischerweise kennt der Dienstanbieter alle Details seines Dienstes. Da der Dienst in der Regel mehrere Effekte erwirken kann, die zum Teil vom Dienstnehmer ausgewählt und konfiguriert werden können, ist die Dienstbeschreibung wie oben erwähnt eine durch Variablen konfigurierbare Menge von Effekten und Vorbedingungen.
- *Anfragebeschreibungen.* Der Dienstnehmer will in der Regel eine bestimmte Funktionalität erbracht wissen und denkt dabei nicht zwangsläufig an einen bestimmten einzelnen existierenden Dienst. Auch geringfügige Abweichungen ist er eventuell bereit hinzunehmen. Typischerweise sind mehrere unterschiedliche Dienste zur Erbringung dieser Funktionalität geeignet. Es ist daher wenig sinnvoll, eine einzelne Dienstinstanz zu notieren. Vielmehr wäre es für ihn interessant, eine Menge aller geeigneter Dienste aufstellen zu können und dabei zu markieren, welche hiervon im Zweifelsfall präferiert würden.

DE-II trägt diesem Umstand durch Einführung *bewertender Elemente* Rechnung. Diese erlauben es, dass Instanzen einen kontinuierlichen Zugehörigkeitsgrad (ausgedrückt durch einen Zahlenwert aus dem Intervall $[0, 1]$) zu einer Menge besitzen können. Damit kann der Dienstnehmer seine Präferenzen bezüglich mehrerer Effekte ausdrücken: Je höher der Zugehörigkeitswert zur Effektmenge, desto stärker bevorzugt der Dienstnehmer diesen Effekt. Technisch geschieht dies durch die Erweiterung der Mengen zu *unscharfen Mengen*.

Dem Dienstnehmer steht damit ein Mittel zur Verfügung, mit dem er seine Präferenzen bezüglich der benötigten Funktionalität *im Voraus*, *vollständig* und *exakt* in die Anfrage integrieren kann. So entstehen *präferenzbeinhaltende Anfragebeschreibungen*. Der Vergleicher selbst ist dann nicht mehr auf die Verwendung allgemeiner Heuristiken angewiesen, sondern kann exakt nach der Anfragebeschreibung und somit nach den Vorstellungen des Dienstnehmers agieren. Die Vergleichsergebnisse eines solchen *persönlichen Vergleichers* sind daher unbeeinflusst von eingebauten Heuristiken und werden in der Regel vom Dienstnehmer ohne Rückfragen akzeptiert (siehe Abbildung 4.6b).

4.4. Dienstbeschreibungssprache DSD

Die eigentliche Beschreibung von Diensten erfolgt mit der *DIANE Service Description* (DSD). Sie basiert auf der Ontologiesprache DE-I und verwendet zudem die neuen Sprachelemente aus DE-II. Ausgangspunkt ist eine **obere Dienstontologie**, die ein grundlegendes Schema für Dienstbeschreibungen aufstellt. In Anlehnung an OWL-S (ohne das Dienstmodell für die Erfassung einer komplexen Choreographie) untergliedert sich dieses in zwei Teile:

- Das *Dienstprofil* beschreibt die Funktionalität des Dienstes auf abstrakte Art und Weise, d.h. was der Dienst macht. Dieser Teil dient der Dienstfindung.
- Das *Dienstfundament* beschreibt, wie die Funktionalität des Dienstes konkret in Anspruch genommen werden kann, d.h. wie der Dienst technisch aufzurufen ist. Dieser Teil dient der Dienstausführung.

Für diese Arbeit ist vor allem das Dienstprofil interessant. Es ist in DSD zweigeteilt, was die zwei grundlegenden Sichtweisen auf einen Dienst widerspiegelt: Nichtfunktionale Aspekte des Dienstes werden durch Instanzen aus DE-I beschrieben.[5] Funktionale Aspekte des Dienstes werden durch die neuen Sprachelemente aus DE-II ausgedrückt.

[5] Auf diese nichtfunktionalen Aspekte wird in dieser Arbeit jedoch nicht weiter eingegangen.

Im Dienstprofil wird die Grundstruktur von Dienstbeschreibungen festgelegt. Allerdings ist diese noch zu generisch, um daraus einheitlich aufgebaute Beschreibungen ableiten zu können. Es fehlen Vorgaben, welche Typen die Operanden von <<precondition>> und <<effect>> annehmen dürfen. Eine Auflistung und Kategorisierung der durch Dienste veränderbaren Zustände in der Welt wird innerhalb der **Kategorieontologien** vorgenommen.

Jede Kategorieontologie enthält eine Sammlung von Zustandsklassen. Zustandsklassen repräsentieren verdinglichte (engl. *reified*) orthogonale Eigenschaften. Beispielsweise verdinglicht die Zustandsklasse Owned die Besitzbeziehung zwischen einer Entität und einer Person, wie sie auch durch das orthogonale Attribut owner in Entity ausdrückbar ist. Die Verdinglichung von Attributen zu Klassen ermöglicht die Verwendung als Typen für Operanden von operationalen Elemente. Die Zustände sind domänenunabhängig, d.h. nur Attribute, die übergreifend für viele unterschiedliche Konzepte gültig sind, sind als Zustände verdinglicht. Wie andere Klassen stehen Zustandsklassen in einer Vererbungshierarchie. Jede erbt zumindest von der allgemeinen Oberklasse State. Ähnliche Zustände finden sich so auf dem selben Pfad zur Wurzel, wodurch ähnlich wirkende Dienste später leichter gefunden werden können.

Insgesamt entstehen 9 Kategorieontologien, die sich in vier große Gruppen einteilen lassen:

- Kategorieontologien mit *Realweltzuständen*, Zuständen also, die Entitäten der realen Welt einnehmen können.
- Kategorieontologien mit *Informationszuständen*, Zuständen also, die Informationseinheiten wie Dateien oder Datenbankeinträge annehmen können.
- Kategorieontologien mit *Wissenszuständen*, Zuständen also, die das Wissen eines Agenten (menschlich oder nicht-menschlich) über eine Entität beschreiben.
- Kategorieontologien mit *Befähigungszuständen*, Zuständen also, die beschreiben, welche Handlungen ein Agent in der Lage ist zu tun.

Die Zustände der Kategorieontologien sind zwar domänenunabhängig, müssen innerhalb einer Dienstbeschreibung jedoch Bezug auf konkrete Entitäten aus bestimmten Anwendungsgebieten nehmen. Deren Beschreibung wird durch **Domänenontologien** vorgenommen. Eine Domänenontologie stellt ein vereinheitlichtes Vokabular für ein inhaltlich abgeschlossenes Themengebiet zur Verfügung. Domänenontologien für DSD werden in DE-I beschrieben, sind *dienstunabhängig* und erreichen eine Strukturierung der Begriffe durch Bereitstellung eines Schemas, einer Sammlung von Instanzen als Repräsentanten von Individuen der realen Welt sowie möglichen domänenspezifischen Gleichheits- und Ähnlichkeitsfunktionen. Zudem sind Verweise auf Konzepte in anderen Ontologien möglich.

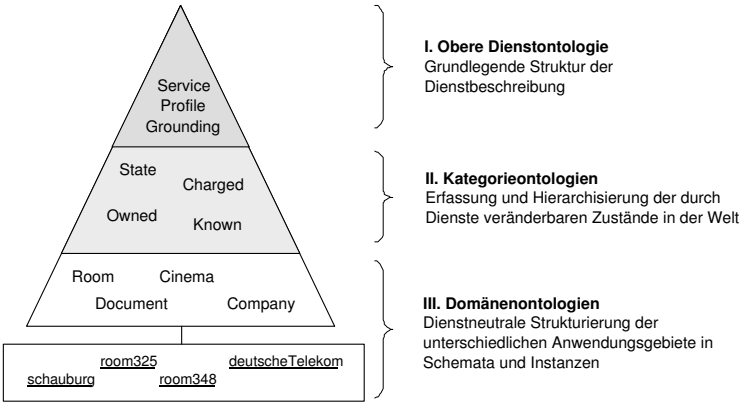


Abbildung 4.7.: Schichtung von Ontologien in DSD.

Insgesamt ergibt sich für DSD eine *Schichtung von Ontologien* wie sie in Abbildung 4.7 dargestellt ist: Eine einheitliche und kleine obere Dienstontologie legt dabei den Grundaufbau einer Dienstbeschreibung fest. Nichtfunktionale Teile werden dabei mit DE-I, funktionale Teile mit DE-II beschrieben. Zustände, die für diese Beschreibung benötigt werden, werden in Kategorieontologien als Klassen verdinglichter Attribute erfasst, strukturiert und hierarchisiert. Den Bezug zu konkreten Anwendungsgebieten stellen Domänenontologien her. Diese sind nicht Bestandteil von DSD, sondern werden von den jeweiligen Domänenexperten bereitgestellt und liegen daher meist in großer Zahl vor. Neben der Strukturierung durch Schemaelemente wird die reale Welt insbesondere auch durch einen Instanzenpool erfasst. Die darin enthaltenen, mit Namen versehenen Instanzen werden mit DE-I beschrieben und stehen stellvertretend für Individuen der realen Welt. Dienstbeschreibungen stellen eine besondere Form von Instanzen dar, die zusätzliche Elemente aus DE-II enthalten.

4.5. Vergleich von DSD-Beschreibungen

Semantische Beschreibungen von Diensten sind nur ein Mittel zum Zweck. Durch sie soll der gesamte Prozess der Dienstnutzung automatisiert werden, ohne die semantische Korrektheit zu vermindern. Wichtig sind also Funktionen, mit denen Beschreibungen zu diesem Zweck verarbeitet werden können. Der wichtigste Schritt während des Prozesses ist der *Vergleich* von Dienstbeschreibungen. Um die semantische Grundlage der Beschreibungen ausnutzen zu können, sollte der Vergleichsalgorithmus von den Schlussfolgerungsoperationen der zugrunde liegenden Ontologiesprache unterstützt werden. Dies darf jedoch nicht so missverstanden werden, dass der Vergleicher zwingend aus vorhanden Schlussfolgerungsoperationen aufgebaut werden muss. Es muss vielmehr zunächst festgelegt werden, wie ein korrektes Vergleichsergebnis intuitiv aussehen sollte, und danach überprüft werden, mittels welcher (vor-

handenen oder neu zu erstellenden) Schlussfolgerungsoperation dies erreicht werden könnte.

Im Falle von zustandsorientiertem DSD hat der Vergleicher für eine gegebene Anfragebeschreibung und eine gegebene Angebotsbeschreibung folgende Grundfrage zu lösen: Liefert der angebotene Dienst in jedem Fall einen Effekt, der zur Anfrage passt?[6] Da sowohl Angebote als auch Anfragen durch Effektmengen beschrieben sind, hat der Vergleicher zu untersuchen, ob das Angebot eine Teilmenge der Anfrage ist. Der Vergleicher verfolgt also ein *konservatives Vorgehen*, d.h. ein Dienstangebot passt nur dann zu einer Anfrage, wenn auch im schlechstest möglichen Fall ein noch gewünschtes Ergebnis erwartet werden kann. Der Vergleicher könnte dazu eine Schlussfolgerungsoperation `subset` zum Test auf Teilmengeneigenschaft zwischen Mengen heranziehen, wie er in Beschreibungslogik z.B. durch `subsumes` zur Verfügung steht.

Im allgemeinen Fall ist die Berechnung des Vergleichsergebnisses jedoch wesentlich schwieriger. Angebotsbeschreibungen enthalten konfigurierbare Mengen, Anfragebeschreibungen unscharfe Mengen. Zur korrekten Verarbeitung dieser Konstrukte benötigt der Vergleicher eine spezifische Schlussfolgerungsoperation: `best-subset`. Seine Aufgabe ist: Gegeben sei eine deklarativ definierte, konfigurierbare Menge o (die Menge der erzielbaren Effekte in der Dienstangebotsbeschreibung) und eine deklarativ definierte, unscharfe Menge r (die Menge der gewünschten Effekte mit Präferenzen in der Dienstanfragebeschreibung). Konfiguriere o in der Art, dass es die *beste Teilmenge* von r ist, wobei „beste“ über die Zugehörigkeitsfunktion von r definiert wird.

Eine direkte Implementierung dieser Schlussfolgerungsoperation mittels Iteration über alle Instanzen ist zwar theoretisch vorstellbar, jedoch praktisch nicht durchsetzbar, da die Zahl der Instanzen sehr groß bis quasi unendlich werden kann und eventuell nicht alle Instanzen zum Zeitpunkt des Vergleichs bekannt sind. Aufgrund von drei Eigenschaften von DSD ist jedoch ein effizienter symbolischer Vergleich möglich: (1) Die Struktur passender Angebots- und Anfragebeschreibungen ist aufgrund des gemeinsamen Schemas ähnlich, (2) die implizit verwendeten Instanzen können teilweise aus den Metaeigenschaften von DE-I abgeleitet und entsprechend verarbeitet werden und (3) die Konstrukte in DE-II zur Definition von Bedingungen für Mengen sind so limitiert, dass eine effiziente Berechenbarkeit stets gewährleistet ist. Die Details zum Vergleicher finden sich in Kapitel 9.

4.6. Zusammenfassung

Der Ansatz dieser Arbeit ist nicht die Definition einer neuen Ontologie zur Beschreibung von Diensten, sondern operiert tief greifender: Auch die zugrunde liegende

[6] Zudem muss überprüft werden, ob alle Vorbedingungen des angebotenen Dienstes erfüllt sind oder erfüllt werden können.

Ontologiesprache wird angepasst. Dazu werden spezielle Charakteristika von Diensten herausgestellt, um daraus angepasste Konstrukte abzuleiten, mit denen exklusiv Dienstbeschreibungen erstellt werden können, die zudem effizient verarbeitbar sind. Der Ansatz operiert dazu auf drei Ebenen:

- Auf Ebene der *generischen Ontologiesprache* entsteht ein Kompromiss zwischen Strukturierung, die für eine unabhängige Erstellbarkeit von Dienstbeschreibungen benötigt wird, und akzeptablem Einigungsaufwand. Hierzu stellt DE-I Möglichkeiten zur Verfügung, um Ontologien in zentrale, langlebige sowie lokale, dynamischere Teile zu zerlegen. Eine Reihe von Modellierungsregeln (etwa „nur rigide Klassen in der zentralen Ontologie“) und zusätzliche Konstrukte (Zustandsklassen, Metaeigenschaften) sollen dabei helfen. Kapitel 5 präsentiert die Details zu DE-I.
- Die Ebene der *dienstspezifischen Ontologiesprache* entsteht neu, indem innerhalb von DE-II vier neue Elemente eingeführt werden, die speziell aus den besonderen Eigenschaften von Diensten abgeleitet wurden. Operationale, aggregierende und konfigurierbare Elemente führen zu zustandsorientierten Dienstbeschreibungen, die sich durch ihre eindeutige funktionale Semantik auszeichnen; mit bewertenden Elementen ergeben sich präferenzbeinhaltende Anfragebeschreibungen, welche zur Definition persönlicher Vergleicher herangezogen werden können. Kapitel 6 präsentiert die Details zu DE-I.
- Auf Ebene der Ontologien wird basierend auf den Elementen aus DE-I und -II der generelle Aufbau von Dienstbeschreibungen im Rahmen der oberen Dienstontologie festgelegt. Diese separiert funktionale und nicht-funktionale Aspekte und trennt zwischen abstrakter und technischer Beschreibung. Durch eine Schichtung von Kategorieontologien mit Zuständen sowie Domänenontologien entsteht eine flexible, universale und dennoch strukturierte Grundlage für Dienstbeschreibungen. Kapitel 7 präsentiert die Details zu DSD.

Grundlage für die Entwurfsentscheidungen liefert ein konzeptionelles Modell, das die wesentlichen Begriffe definiert und in Beziehung setzt. Die axiomatische Semantik von DE-I, -II und DSD wird dann in Kapitel 8 eingeführt.

Auch für den Vergleichsvorgang werden keine generischen Operationen der Ontologiesprache herangezogen. Vielmehr wird aus intuitiv korrekten Vergleichsergebnissen gefolgert, wie diese zu erreichen und welche Operationen dazu nötig wären. Im Falle von DSD ist dies im Wesentlichen die Berechnung der Teilmengeneigenschaft zwischen konfigurierbaren und unscharfen Mengen. Aufgrund der einschränkenden Konstrukte von DE kann für diese eine effiziente Implementierung angeboten werden. Den genauen Ablauf des Vergleichers präsentiert Kapitel 9.

5. Generische Ontologiesprache: DIANE Elements I

Aufgabe einer generischen Ontologiesprache ist es, Elemente zur Aufstellung der zugrunde liegenden Domänenontologien zur Verfügung zu stellen. Einerseits soll damit ein geteiltes Verständnis der verwendeten Konzepte erreicht werden, um so eine ordnende Wirkung auf unabhängig erstellte Dienstbeschreibungen zu erhalten, andererseits darf der Einigungsaufwand gerade im Hinblick einer sich ständig ändernden Ontologie nicht über alle Maßen wachsen.

Nicht jede beliebige Ontologie kann daher als Grundlage für die Beschreibung von Diensten verwendet werden. Wie im letzten Kapitel gezeigt wurde, muss eine Reihe von Eigenschaften erfüllt sein. Die Erreichung von Ontologien mit diesen Eigenschaften wird durch die generische Ontologiesprache *DIANE Elements I* (DE-I) möglich. In diesem Kapitel wird ihre genaue Syntax und verbale Semantik eingeführt (siehe dazu auch [73]). Die formale Semantik ihrer Elemente findet sich in Kapitel 8. Abschnitt 5.1 zeigt eine Übersicht über die Repräsentationsformen für DE. In Abschnitt 5.2 werden die Elemente zur Definition des Schemas, in Abschnitt 5.3 die zur Definition der Instanzen eingeführt.

5.1. Repräsentationsformen

DE ist nicht an eine Syntax gebunden, sondern es existieren mehrere *Repräsentationsformen*, die jeweils unterschiedliche Vor- und Nachteile bieten und daher in verschiedenen Situationen zum Einsatz kommen. Abbildung 5.1 zeigt deren Zusammenhang:

- Im Mittelpunkt steht die *formale Hauptrepräsentation* (kurz f-dsd). Diese ist durch eine formale Grammatik definiert, deren Ziel es ist, möglichst kompakte und gut lesbare Beschreibungen zu erhalten. Da die Beschreibungen rein textbasiert sind, können sie leicht zwischen verschiedenen Computersystemen ausgetauscht werden. Als Hauptrepräsentation strebt f-dsd nach Vollständigkeit, d.h. in ihr sind alle Konzepte und Beschreibungsmöglichkeiten der DE enthalten und ausdrückbar. Die Syntax der f-dsd ist angelehnt an F-Logic und

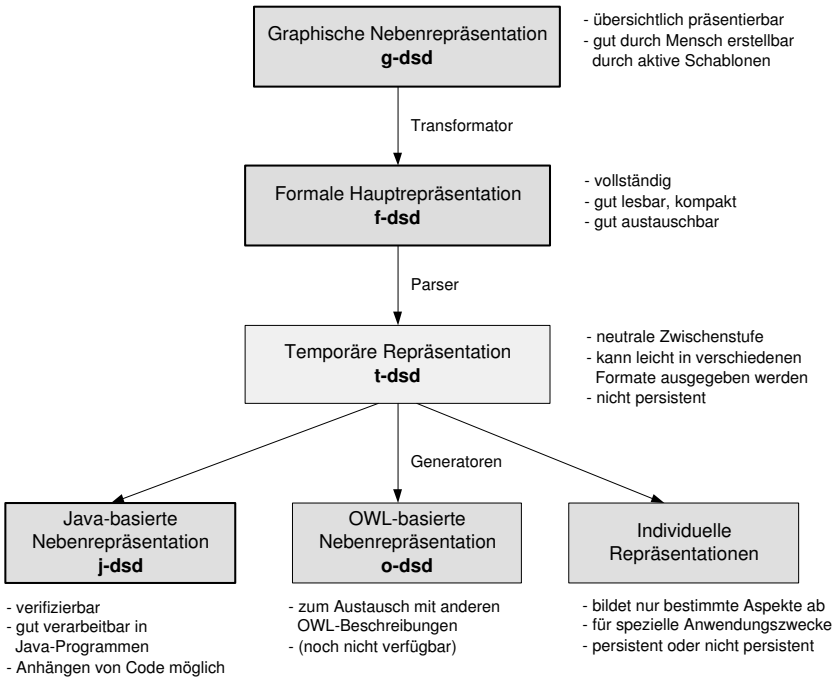


Abbildung 5.1.: Repräsentationsformen für DE.

übernimmt die Darstellung primitiver Werte aus XML Schema. Um eine übersichtlichere Darstellung zu erhalten, wurde auf Verwendung von XML verzichtet [65].

- Eine anschauliche Darstellung liefert die *graphische Nebenrepräsentation* (kurz g-dsd). Sie wurde als visuelle Sprache konzipiert, sodass sich menschliche Benutzer in ihr schnell zurechtfinden, um so existierende Beschreibungen gut verstehen und neue Beschreibungen effizient aufbauen zu können. Editiert werden können g-dsd-Beschreibungen mit einem Werkzeug, welches durch Schablonen einem menschlichen Benutzer aktiv bei der Erstellung von Beschreibungen behilflich ist [64]. Ein Transformator kann Beschreibungen von g-dsd nach f-dsd umwandeln. Die Notation von g-dsd ist an UML-Klassendiagramme angelehnt.

- Um Beschreibungen in formaler Repräsentation verarbeiten zu können, werden sie zunächst von einem speziellen Parser eingelesen und in *temporärer Repräsentation* (kurz t-dsd) nicht-persistent im Hauptspeicher abgelegt. Diese neutrale Zwischenstufe ist darauf ausgelegt, in eine der weiteren Nebenrepräsentation persistent ausgegeben zu werden. Auch eigene Spezialrepräsentationen können hieraus leicht abgeleitet werden.

- Eine wichtige Repräsentation ist die *Java-basierte Nebenrepräsentation* (kurz j-dsd). In ihr wird das Schema zu einem Satz von Java-Klassen, welche durch ihren

Quellcode dargestellt sind. Instanzen hingegen sind gewöhnliche Java-Objekte, die durch statische Erzeugungsklassen persistent gemacht werden. Die neuen Sprachelemente aus DE-II wie Mengen oder Variablen werden durch besondere Klassen umgesetzt. Diese Darstellungsform hat eine Reihe von Vorteilen: Zunächst können durch Kompilierung des Quellcodes automatisch semantische Fehler entdeckt werden, deren Überprüfung in f-dsd komplexe Zusatzprogramme erfordert hätte. Weiterhin können die übersetzten Klassen direkt in allen Java-Programmen verwendet werden. Drittens kann die Semantik der Sprachelemente wie Menge oder Variable durch bestimmte Methoden hinterlegt werden. Auch der Vergleicher und die Middleware verarbeiten Beschreibungen in dieser Darstellung.

- Zum Austausch mit anderen Forschungsgruppen wäre eine (vermutlich verlustbehaftete) *OWL-* oder *WSML-basierte Nebenrepräsentation* denkbar. Hierin wären die Konzepte der DE mit Mitteln der Zielsprache dargestellt.

- Neben den vorhandenen Darstellungsformen können auch *individuelle Repräsentationen* entwickelt werden. Diese sind häufig verlustbehaftet, d.h. sie stellen nur die Aspekte einer Beschreibung dar, die für den jeweiligen Anwendungszweck von Bedeutung sind und blenden unwichtige Teile aus. Es sind sowohl persistente als auch nicht-persistente Repräsentierungen denkbar.

In dieser Arbeit wird durchgängig die visuelle g-dsd-Repräsentation verwendet. Die Syntax von f-dsd und j-dsd wird in Anhang B dargestellt. Die Grammatik für f-dsd findet sich in Anhang C.

5.2. Schemata

Zur Definition von Schemata stehen zwei Metakonzepte zur Verfügung: Vordefinierte Datentypen (auch eingebaute oder primitive Datentypen) sowie selbstdefinierbare Datentypen (auch komplexe Datentypen oder Klassen).

5.2.1. Primitive Datentypen

Primitive Datentypen stellen Typen dar, die bereits vordefiniert sind. Durch sie werden allgemein gültige Sachverhalte wie Zahlen, Daten, Wahrheitswerte, Zeichenketten etc. abgebildet. Jeder dieser Typen besteht aus einer Menge gültiger Werte (seinem Wertebereich) sowie einer oder mehrerer lexikalischer Repräsentationen dieser Werte. Von DE werden zurzeit acht wichtige primitive Datentypen unterstützt, die sich von den in XML Schema [17] definierten primitiven Datentypen ableiten:

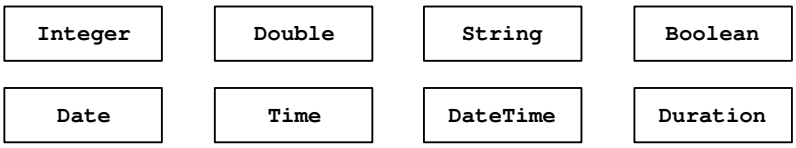


Abbildung 5.2.: Primitive Datentypen in g-dsd.

- `Integer`. Repräsentiert die Menge der ganzen Zahlen.
- `Double`. Repräsentiert die Menge der Fließkommazahlen.
- `String`. Repräsentiert die Menge der Zeichenketten, d.h. endliche Ketten von Einzelzeichen.
- `Boolean`. Repräsentiert die Menge der Wahrheitswerte, d.h. die Menge bestehend aus `wahr` und `falsch`.
- `Date`. Repräsentiert die Menge der Datumswerte. Ein Datum steht für einen vergangenen, jetzigen oder zukünftigen Tag in der Zeit, also ein nicht-periodisches Ereignis. Ein Tag beginnt um 0:00 Uhr und endet mit dem Beginn des folgenden Tages.
- `Time`. Repräsentiert die Menge der Zeitpunkte an einem Tag. Ein solcher Zeitpunkt wiederholt sich an jedem Tag. Seine Dauer ist null.
- `DateTime`. Repräsentiert die Menge der einzelnen Punkte in der Zeit. Die Dauer eines Zeitpunkts ist null. `DateTime` kann als Kombination von `Date` und `Time` angesehen werden.
- `Duration`. Repräsentiert die Menge der Zeitdauern. Eine Zeitdauer hat keinen bestimmten Anfangs- und Endzeitpunkt, sondern steht stellvertretend für jedes Zeitintervall dieser Länge.

In g-dsd wird ein primitiver Datentyp durch ein weißes Kästchen dargestellt, das den Namen des Typs in fetter Schreibmaschinenschrift trägt (siehe Abbildung 5.2).

Auf jedem primitiven Datentyp ist eine **totale Ordnungsrelation** definiert, mit deren Hilfe je zwei Literale des selben Typs verglichen werden können. Für die zahlen- und zeitbezogenen Typen ist die Relation klassisch definiert, `String`s werden lexikographisch verglichen, für `Boolean` gilt, dass `falsch` kleiner als `wahr` ist. Die Ordnungsrelationen ermöglichen es, folgende Vergleiche auf dem Wertebereich durchzuführen: $==, <=, >=, <, >, != $.

Neben der scharfen Ordnungsrelation sind noch *unscharfe Ordnungsrelationen* definiert, die ein Wertepaar nicht binär vergleichen, sondern gewisse Toleranzen zulassen.

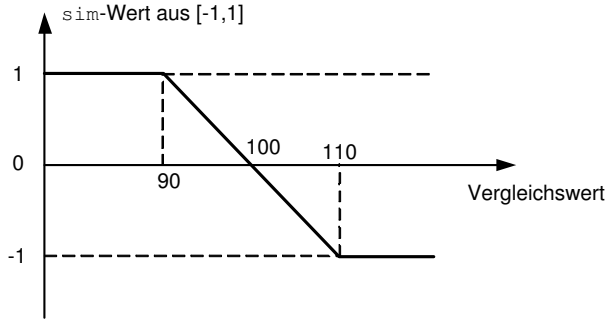


Abbildung 5.3.: Unscharfer Vergleich des Referenzwertes 100.

Beispielsweise sind die Zahlen 100 und 100.1 nicht exakt gleich, in manchen Fällen ist es jedoch sinnvoll festzuhalten, dass sie *fast* gleich sind und die Ähnlichkeit `sim` mit einem Fließkommawert auszudrücken. Für die einzelnen konkreten Datentypen ist `sim` wie folgt definiert: Zahlenbezogene Werte dürfen um bis zu 10% nach oben oder unten vom Referenzwert abweichen, erst danach gelten sie als wirklich kleiner (+1) oder wirklich größer (-1). Dazwischen ändert sich der Rückgabewert kontinuierlich und linear von +1.0 über 0.0 zu -1.0. Abbildung 5.3 veranschaulicht diese Beziehung für den beispielhaften Referenzwert 100.

Für datumsbezogene Werte gelten folgende Festlegungen: Für `Date` und `DateTime` ist keine standardmäßige Abweichung definiert. Für `Time` ist eine Abweichung von +/- 10% einer Tageslänge, für Duration eine Abweichung von +/- 10% von der Referenzdauer zugelassen. Für `String` und `Boolean` ist keine unscharfe Vergleichsfunktion definiert; hier sind unscharfer und scharfer Vergleich gleich.

Mit Hilfe von `sim` können nun die bekannten Vergleichsoperatoren so geändert werden, dass sie nicht nur streng gelten (1) oder nicht gelten (0), sondern auch den Zwischenbereich von 0.0 und 1.0 annehmen können. Die Notation erfolgt durch ein vorangestelltes $\sim$:

- Berechne $a \sim== b$ durch $1 - |$`sim(a,b)`$|$
- Berechne $a \sim<= b$ durch `sim(a,b) >= 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim< b$ durch `sim(a,b) > 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim>= b$ durch `sim(a,b) <= 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim> b$ durch `sim(a,b) < 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim! = b$ durch $|$`sim(a,b)`$|$

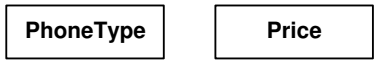


Abbildung 5.4.: Beispielhafte Klassendefinition in g-dsd.

Neben den vordefinierten unscharfen Vergleichsoperatoren, die eine 10%-ige Abweichung zulassen, kann die erlaubte Abweichung auch selbst definiert werden. Hierzu wird in eckigen Klammer die minimal und/oder maximal erlaubte prozentuale oder absolute Abweichungsgrenze angegeben. Ein Beispiel könnte sein $x \sim< [20\%]y$ oder $100 \sim== [75, 125]y$. Im ersten Fall darf der Vergleichswert y auch noch um bis zu 20% größer sein als der Referenzwert x um noch als *ungefähr kleiner* zu gelten, im zweiten Fall gilt der Wert von y noch als *ungefähr gleich* zu 100, falls er im Bereich zwischen 75 und 125 liegt. Der Ähnlichkeitswert fällt jeweils linear.

5.2.2. Klassen

Im Gegensatz zu primitiven Typen können Klassen selbst definiert werden.

Konzeption

Klassen sind Datentypen, die nicht von DE fest vorgegeben werden, sondern von der Community selbst definiert werden können. Ihre Anzahl ist daher prinzipiell nicht beschränkt. Klassen stehen stellvertretend für die Sammlung aller Individuen der realen Welt einer bestimmten Art. Beispielsweise steht die Klasse PhoneType für die Sammlung aller Telefonmodelle, die Klasse Price für alle Preise. Klassen sind als Teil des Schemas für alle Systemteilnehmer von Bedeutung und stehen damit öffentlich zur Verfügung (vgl. Eigenschaft 4.2 auf Seite 103).

Als Modellierungsrichtlinie gilt:

- Klassen sind stets rigide Konzepte. (Eigenschaft 4.5)
- Das Schema ist anwendungsneutral zu modellieren. (Eigenschaft 4.3).

Klassen werden in g-dsd durch ein weißes Kästchen mit dem Klassennamen in fetter Normalschrift dargestellt, wie Abbildung 5.4 beispielhaft zeigt.

Intrinsische Eigenschaften

Haben die durch die Klasse vertretenen Individuen gemeinsame Eigenschaften bzw. können diese mit anderen Individuen in Beziehung stehen, so wird dies durch Attribute repräsentiert. Nach Eigenschaft 4.6 werden nur intrinsische Attribute direkt notiert. Ein Attribut hat einen Namen und einen Zieltyp, d.h. einen Typ der angibt, durch welche Literale oder Instanzen das Attribut „gefüllt“ werden kann. Der Zieltyp kann sowohl ein primitiver Datentyp als auch eine Klasse sein. Im Beispiel könnte die Klasse PhoneType die Attribute name vom Typ `String`, availableSince vom Typ `Date` und manufacturer vom Typ Company besitzen. Der Name eines Attributes beginnt immer mit einem Kleinbuchstaben. Auf Instanzebene nennt man den Wert, der ein Attribut ausfüllt, *Füllwert*.

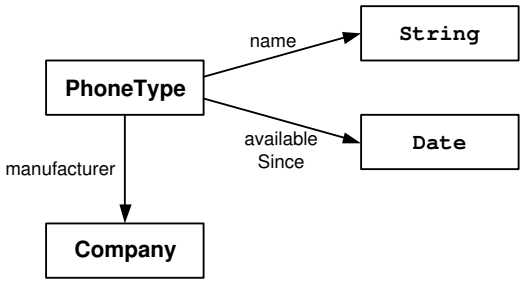


Abbildung 5.5.: Beispielhafte Attributdefinition in g-dsd.

In g-dsd werden Attribute durch Pfeile dargestellt, die von der Klasse zum Zieltyp zeigen und mit dem Namen des Attributes beschriftet sind. Abbildung 5.5 zeigt das am Beispiel.

Grundsätzlich sind alle Attribute optional und einwertig, d.h. in UML hätten sie die Kardinalität 0..1. Durch die Verwendung definierender Attribute kann dies geändert werden.

Definierende und ableitbare Attribute

Attributarten stellen eine Metaeigenschaft dar, um die Art eines Attributs genauer erfassen zu können. Attributarten sind daher eine Erweiterung klassischer Objektorientierung. Nach Eigenschaft 4.9 soll die Markierung mit Attributarten erfassen, welche Attribute einer Klasse voneinander abhängen und welche nicht. Ein ähnliches Konzept verkörpern funktionale Abhängigkeiten aus dem Bereich relationaler Datenbanken. Zudem legt die Attributart fest, welche Attribute verpflichtend sind, d.h. für jede Instanz gefüllt sein müssen.

Intrinsische Attribute werden in DSD in zwei Arten unterschieden: definierende und ableitbare. Dabei gilt folgender Grundsatz: Sind von einer Instanz die Füllwerte aller

definierenden Attribute bekannt, so sind dadurch auch die Füllwerte der ableitbaren Attribute bestimmt. Anders ausgedrückt sind zwei Instanzen mit definierenden Attributen gleich, wenn sie gleiche Füllwerte in ihren definierenden Attributen besitzen. In der Beispielklasse PhoneType wäre das Attribut name definierend, während availableSince und manufacturer ableitbar sind (in der Annahme, dass es keine zwei gleich heißenden Telefonmodelle gibt).

In g-dsd werden definierende Attribute durch einen schwarz ausgefüllten Kreis am Pfeilende markiert. Ableitbare Attribute bleiben ohne Markierung (siehe Abbildung 5.6).

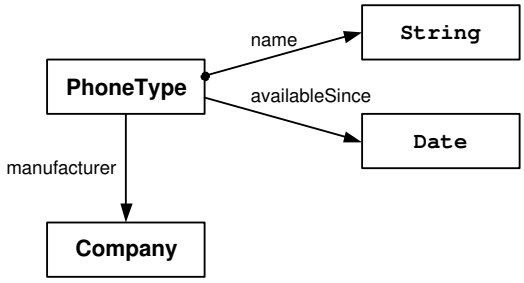


Abbildung 5.6.: Unterscheidung von definierenden und ableitbaren Attributen in g-dsd.

Die formale Semantik von definierenden und ableitbaren Attributen findet sich in den Regeln 8.4 bis 8.18 in Kapitel 8.

Vererbung

Eine besondere Art der Beziehung zwischen Klassen stellt die Vererbung dar. Sie drückt eine *is-a*-Beziehung aus. Eine Klasse K sollte von einer Klasse L erben, wenn L alle Instanzen enthält, die auch K enthält, K also eine Spezialisierung von L ist. Es gilt, dass K alle Attribute, die in L definiert sind, übernimmt und zusätzliche weitere definieren kann. Beispielsweise sollte die Klasse Company von LegalPerson erben, da jedes Unternehmen eine juristische Person ist. In DE kann jede Klasse nur von maximal einer Oberklasse erben, d.h. das Konzept der Mehrfachvererbung existiert nicht. Erbt eine Klasse (außer Thing) von keiner Klasse, so erbt sie implizit von Thing.

In g-dsd wird die Vererbung durch den aus UML bekannten unausgefüllten Pfeil dargestellt, der von der Unter- zur Oberklasse zeigt. Ein Beispiel zeigt Abbildung 5.7.

Für die Vererbung gelten Einschränkungen, wenn die Klassen mit bestimmten Metaeigenschaften versehen sind. Details hierzu finden sich in Abschnitt 5.2.6. Die formale Semantik der Vererbungsbeziehung findet sich in den Regeln 8.20 bis 8.30.

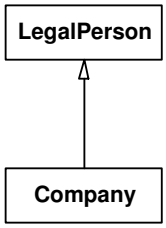


Abbildung 5.7.: Beispiel für eine Vererbungsbeziehung in g-dsd.

Extrinsische Eigenschaften

Extrinsische Eigenschaften einer Klasse entstehen durch Beziehungen mit anderen Klassen. Solche Eigenschaften sind daher nicht inhärenter Bestandteil der Klasse, sondern von extern angetragen. Die Füllwerte extrinsischer Attribute sind daher kontextabhängig gefüllt. Ein Beispiel ist der Preis eines Telefonmodells, der nur im Kontext eines Verkaufsangebots einen Sinn macht und vom konkreten Verkäufer abhängt.

Wie in Eigenschaft 4.6 dargelegt, ist es daher nicht sinnvoll, extrinsische Eigenschaften direkt über Attribute anzugeben, da so der Kontext des Füllwerts nicht erfasst werden kann. Aus diesem Grund wird in DE-I eine extrinsische Eigenschaft als Zustand aufgefasst, in dem sich die beschriebene Entität befinden kann. Dies führt zu einer Verdinglichung der Eigenschaft in eine Zustandsklasse. Diese Klasse kann dann durch zusätzliche Attribute den Kontext der extrinsischen Eigenschaft erfassen.

Abbildung 5.8 zeigt ein Beispiel für die extrinsische Eigenschaft des Preises eines Telefonmodells. In (a) ist diese wie häufig in der Literatur zu finden direkt als ableitbares Attribut price dargestellt, was ungeeignet ist, da der Verkaufskontext nicht dargestellt werden kann. Darunter ist die für DE-I korrekte Darstellung zu sehen: In (b) wird der neue Zustand Priced (dt. bepreist) eingeführt, welcher als Attribute den eigentlichen Preis in price, die betroffene Entität in entity sowie den Verkaufskontext durch den Verkäufer in valuer und den Gültigkeitszeitraum in startsAt und endsAt enthält. In PhoneType wird lediglich ein Hinweis auf einen möglichen Zustand Priced vermerkt. Dies geschieht in (c) durch die Angabe eines *orthogonalen Attributs*, gekennzeichnet durch den unausgefüllten Kreis. Als Name des Attributs sollte die Konvention isState (hier z.B. isPriced) eingehalten werden, wobei State den Zieltyp des Attributs darstellt.

Orthogonale Attribute haben nur eine hinweisende Funktion. Sie können für eine konkrete Instanz nicht gefüllt werden.[1] Vielmehr muss eine entsprechende Instanz der zugehörigen verdinglichten Zustandsklasse angelegt werden.

[1] Zur Definition von Mengen können orthogonale Attribute jedoch als Attributbedingungen verwendet werden (siehe dazu Abschnitt 6.1.3).

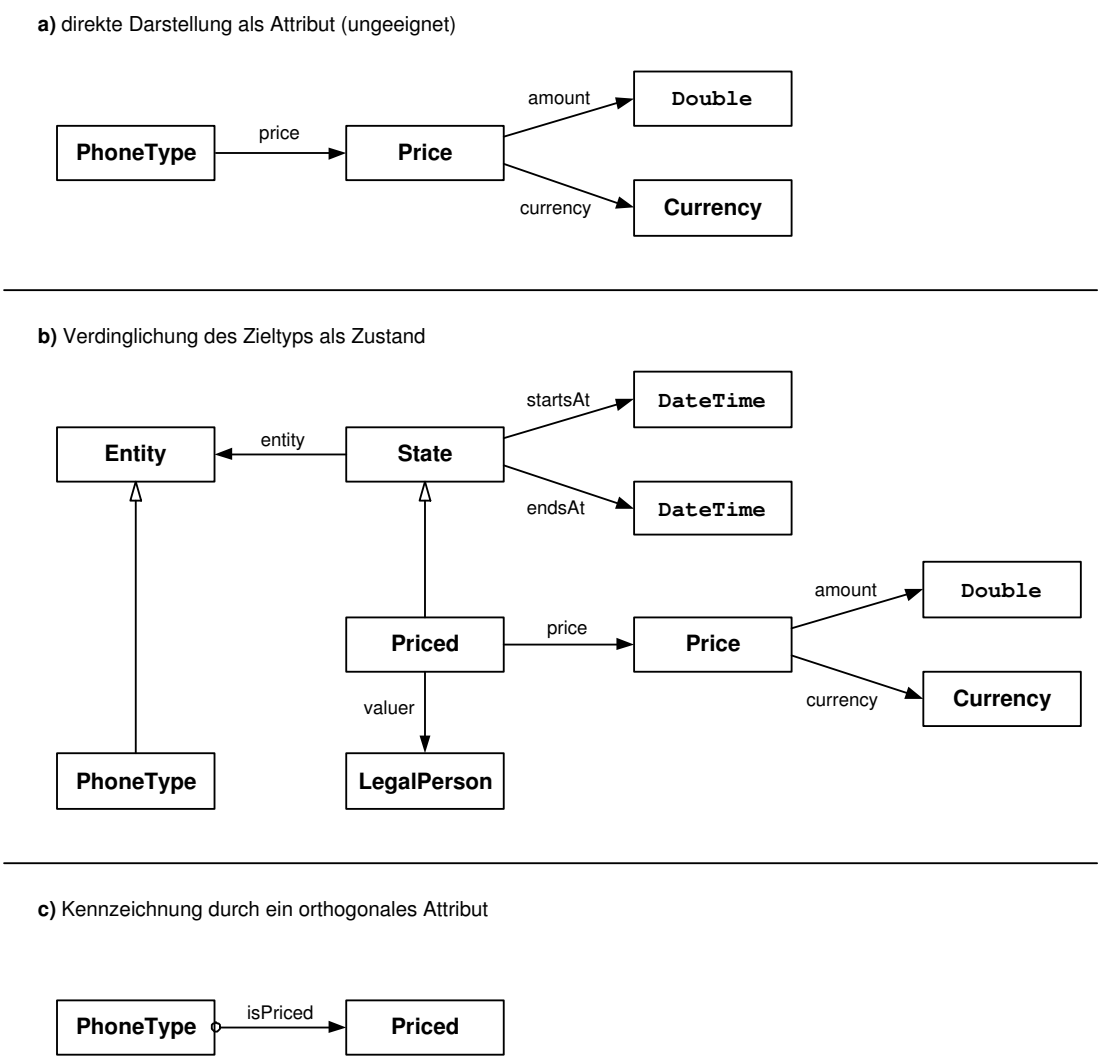


Abbildung 5.8.: Darstellung einer extrinsischen Eigenschaft in g-dsd: (a) ungeeignet über ein direktes Attribut; (b) in DE über einen verdinglichten Zustand und (c) ein orthogonales Attribut.

5.2.3. Ontologien

Um den Raum der Klassen übersichtlich zu halten, ist es wichtig, diesen zu strukturieren (Eigenschaft 4.2). Hierzu wird in DE das Konzept der *Ontologien* verwendet. Eine Ontologie ist eine Sammlung von Klassen (und Instanzen, siehe später), die thematisch zusammengehören und so von einer Reihe von Domänenexperten gewartet werden kann.

Ontologienamen

Ontologien werden durch einen eindeutigen Namen unterschieden. Generell unterscheidet man Ontologien auf den drei durch DSD vorgegebenen Ebenen (siehe Abschnitt 4.4), was auch den ersten Bestandteil dieses Namens bestimmt: Die obere Dienstontologie (bezeichnet durch `upper`), Kategorieontologien (bezeichnet durch `category`) und Domänenontologien (bezeichnet durch `domain`). In diesen Gruppen können hierarchische Untergruppen eröffnet werden, die jeweils thematisch abgeschlossen sind. Ihre Namen werden analog zum Namensraum-Mechanismus aus XML oder dem Package-Mechanismus aus Java durch Punkte strukturiert. Start ist immer der Bezeichner der Hauptgruppe. Um beispielsweise die Domäne der Telekommunikation zu beschreiben, könnte die Ontologie `domain.telecommunications` verwendet werden, die Welt der Fahrzeuge könnte man in `domain.transportation.vehicles` modellieren. Die Kategorie der Besitzzustände findet sich in `category.possession`, die Grundkonzepte zum Dienstprofil in `upper.profile`. Zu bemerken ist, dass durch die Punktnotation keine Inklusion definiert wird: Konzepte aus `a.x.y` sind nicht automatisch auch in `a` und `a.x` enthalten.

Neuerstellung von Ontologien

Die Sammlung der Klassen und Instanzen einer zu definierenden Ontologie wird in g-dsd auf einer Zeichenfläche zusammengefasst. Diese hat den Markierer ONTOLOGY <Name: >, mit dem die Neudefinition der Ontologie mit dem angegebenen Namen eingeleitet wird. Auf der Zeichenfläche selbst kann eine beliebige Anzahl von Klassen und Instanzen definiert werden, welche alle in der angegebenen Ontologie abgelegt werden. Wichtig ist, dass generell alle in einer Ontologiedefinition verwendeten Typen entweder primitiv sind oder selbst in dieser Ontologie definiert werden. Klassen aus anderen Ontologien müssen gekennzeichnet werden.

Verwenden vorhandener Ontologien

Nach Eigenschaft 4.2 gilt für Schemata der Grundsatz, dass diese als Ergebnis eines Einigungsprozesses in der Anwendergruppe erstanden sind und öffentlich zur Verfügung gestellt werden. Daraus folgt die Forderung, dass keine zwei Klassen existieren, welche für dieselben Individuenmengen der realen Welt stehen. Beispielsweise ist es unzulässig, gleichzeitig die Klassen Company und Unternehmen zu besitzen, die dasselbe bedeuten, aber andere Namen oder andere Attribute und Oberklassen besitzen. In einem solchen Fall fehlt die angesprochene Einigung in der Anwendergruppe. Diese müsste sich für eine Klasse entscheiden und in Zukunft ausschließlich diese nutzen.

Konsequenterweise erstreckt sich diese Forderung auch auf ganze Ontologien. Ontologien müssen disjunkt sein, d.h. keine Klasse (oder Instanz) darf in mehr als einer Ontologie definiert werden.[2]

Die Wiederverwendung von Klassen und Instanzen aus bereits existierenden Ontologien ist daher nicht nur eine Möglichkeit, den Aufwand einer Neuerstellung zu sparen, sondern verpflichtend. Bei der Einbindung solcher externer Klassen in die eigene Ontologie muss die ontologische Herkunft der Klasse angegeben werden.

In g-dsd erfolgt das durch Setzen der bereits extern definierten Klasse als Kästchen mit gestrichelter Umrandung. Zudem steht der Name der Herkunftsontologie in kleiner, kursiver Normalschrift oberhalb des Klassennamens. Abbildung 5.9 zeigt, wie die Klasse LegalPerson aus der externen Ontologie `domain.legal` verwendet wird, um in der Ontologie `domain.economy` die Klasse Company zu definieren.

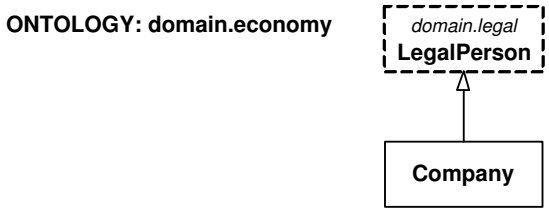


Abbildung 5.9.: Verwendung der in `domain.legal` definierten Klasse LegalPerson zur Neudefinition von Company in `domain.economy`

5.2.4. Wertbestimmte und Entitätsklassen

Nach Eigenschaft 4.8 ist es unerlässlich, zwischen wertbestimmten und Entitätsklassen zu unterscheiden. Es ist daher wichtig, für jede Klasse bei ihrer Definition anzugeben, von welchem Typ sie ist:

- **Wertbestimmte Klassen** (engl. value-based classes). Klassen, bei denen jede beliebige, gültige Kombination aus Füllwerten zu Instanzen führt, die existierende Individuen der realen Welt beschreiben, gelten als wertbestimmte Klassen. Ausgehend von einer beliebigen Instanz einer solchen Klasse führt eine

[2] In der Praxis erweist sich die Durchsetzung dieser Forderungen als nicht immer einfach: Einerseits kann die Anwendergruppe sehr groß werden, was eine direkte Einigung erschwert oder unmöglich macht, andererseits kann die Anwendergruppe sehr dynamisch sein, etwa in einem mobilen Netz, wo ständig Teilnehmer der Anwendergruppe beitreten oder diese verlassen. In solchen Fällen ist es üblich, die Forderung in gewissen Maßen abzuschwächen und „persönliche Ontologien“ zu erlauben. Diese müssen jedoch vor der Verwendung durch Verfahren der Ontologieanpassung (engl. ontology alignment) (semi-)automatisch aufeinander abgebildet werden, um zu einer Vereinheitlichung zu gelangen.

(kleine) Änderung an einem der Füllwerte bereits zu einer neuen Instanz, die ein anderes Individuum der Welt beschreibt. Beispiele für solche Klassen sind Price oder WeightMeasure. Für solche Klassen sind nur anonyme Instanzen sinnvoll (siehe Abschnitt 5.3.2). Aufgrund der Orthogonalität der Attribute ist die Unterscheidung in definierende und orthogonale Attributen nicht sinnvoll (alle Attribute können als definierend angesehen werden). Die Identität der Instanzen ist vollständig durch die Füllwerte bestimmbar. Häufig sind dazu jedoch domänenspezifische Gleichheits- oder Ähnlichkeitsfunktionen nötig (siehe Abschnitt 5.2.7).

- **Entitätsklassen** (engl. entity classes). Klassen, bei denen nur bestimmte Kombinationen von Füllwerten zu Instanzen führen, die Individuen der Welt beschreiben, gelten als Entitätsklassen. Jede solche Wertekombination beschreibt eine eigenständige *Entität*, die durch einen global eigenständigen Namen gekennzeichnet werden kann. Für solche Klassen sind daher nur benamte Instanzen sinnvoll (siehe Abschnitt 5.3.2). Beispiele für solche Klassen sind Phone, Person und Company. Die Identität der Entitäten ist nicht immer einfach aus den Attributen abzuleiten, denn kleine Änderungen an den Attributen führen nicht zwangsläufig zu neuen Entitäten. Daher sollte der Vergleich den Namen der Entität in Betracht ziehen.

In g-dsd erfolgt die Markierung durch einen tiefgestellten Index am Namen der Klasse: E für Entitätsklassen, V für wertbestimmte (engl. value-based) Klassen. Auch hier wird als Standard eine Markierung mit E angenommen. Abbildung 5.10 zeigt ein Beispiel.

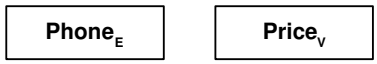


Abbildung 5.10.: Differenzierung zwischen wertbestimmten und Entitätsklassen in g-dsd.

Die formale Semantik wertbestimmter und Entitätsklassen findet sich in den Regeln 8.38 bis 8.42.

5.2.5. Öffentliche und teilöffentliche Entitätsklassen

Nach Eigenschaft 4.4 müssen Entitätsklassen in öffentliche und teilöffentliche Entitätsklassen unterteilt werden:

- **Öffentliche Entitätsklassen.** In öffentlichen Entitätsklassen ist jede Instanz von allgemeinem Interesse und muss der Öffentlichkeit zur Verfügung gestellt werden. Beispiele für solche Klassen sind PhoneType und Currency.

- **Teilöffentliche Entitätsklassen.** In solchen Entitätsklassen existieren auch Instanzen von privatem Interesse, die der Öffentlichkeit nicht zur Verfügung gestellt werden brauchen. Beispiele für solche Klassen sind Person und Phone. Definierende Attribute sind in solchen Klassen von besonderer Bedeutung, da häufig nur deklarativ über die Füllwerte dieser Attribute angefragt werden kann.

In g-dsd erfolgt die Markierung durch Anhängen des Index P an den Klassennamen, wie das Beispiel in Abbildung 5.11 zeigt. Als Standard für Entitätsklassen ohne Markierung gilt, dass sie teilveröffentlicht sind.

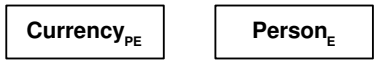


Abbildung 5.11.: Markierung von öffentlichen Entitätsklassen in g-dsd.

Die formale Semantik von öffentlichen und teilöffentlichen Klassen findet sind in den Regeln 8.47 bis 8.48. Die Unterscheidung wird in Abschnitt 5.3.4 über öffentliche und private Instanzen noch einmal aufgegriffen.

5.2.6. Vererbungsbeschränkungen

Aufgrund der großen semantischen Unterschiede zwischen wertbestimmten und Entitätsklassen und den daraus resultierenden Forderungen an die Instanziierung sowie der Öffentlichkeit von Entitätsklassen, entstehen auch Einschränkungen bei der Vererbung von Klassen:

- Eine Vererbung zwischen einer wertbestimmten Oberklasse und einer Unterklasse als Entitätsklasse oder umgekehrt ist nicht erlaubt (formal in den Regeln 8.43 und 8.44). Dies ist damit begründet, dass die beiden Klassentypen Instanzen enthalten, die vom Wesen her unterschiedlich sind. Durch die Inklusionssemantik der Vererbung wären so anonyme Instanzen in Entitätsklassen bzw. benamte Instanzen in wertbestimmten Klassen enthalten.
- Eine teilveröffentlichte Klasse darf nicht von einer öffentlichen Klasse erben (formal in Regel 8.48). Dies ist damit begründet, da sonst aufgrund der Inklusionssemantik der Vererbung nicht mehr garantiert wäre, dass alle Instanzen der öffentlichen Oberklasse bekannt wären.

Generell gilt also, dass die Markierer `entityclass`, `valueclass` und `public` an die Unterklassen „vererbt“ werden. Durch die Überprüfung dieser Eigenschaft können auch prinzipielle Modellierungsfehler beim Erstellen einer Ontologie entdeckt werden.

5.2.7. Domänenspezifische Berechnungen auf Instanzen

Für Klassen besteht die Möglichkeit, eigene Relationen zu definieren. Hierdurch wird auf eine domänenspezifische Weise festgelegt, wann zwei Instanzen gleich sind, wie ähnlich sie sind, in welcher Ordnung sie stehen oder sonstiges. Wie bei allen Schemaelementen muss sich die Anwendergruppe bei der Definition solcher Relationen einigen. Dennoch ist die Verwendung dieser Relationen besonders in Anfragebeschreibungen nicht zwingend. Hier können über deklarative Mengen jederzeit persönliche Vergleichsfunktionen definiert werden. Domänenspezifische Relationen sind insbesondere für wertbestimmte Klassen von Bedeutung.

Die Definition von domänenspezifischen Relationen kann nur als Codefragment in j-dsd erfolgen; die Verwendung ist jedoch in allen Repräsentationsformen möglich. Häufig benötigte und daher bereits spezifizierte Beispiele können sein:

- `Double _distance(Location loc1, Location loc2)`
  Berechnet den Abstand zwischen den Lokationen `loc1` und `loc2` in Metern.
- `DateTime _after(DateTime dt, Duration dur)`
  Berechnet den Zeitpunkt, der `dur` nach `dt` liegt.
- `DateTime _before(DateTime dt, Duration dur)`
  Berechnet den Zeitpunkt, der `dur` vor `dt` liegt.
- `[0,1] _near(Location loc1, Location loc2)`
  Bestimmt die Nähe zwischen zwei Orten. Sollte in den Unterklassen von Location überschrieben werden.
- `Boolean _equalWeight(WeightMeasure wm1, WeightMeasure wm2)`
  Bestimmt, ob die zwei gegebenen Gewichtsangaben das gleiche Gewicht bezeichnen (wie etwa 1000g und 1kg).

5.3. Instanzen

Instanzen gehören gleichberechtigt zu einer Ontologie und stellen das zweite grundsätzliche Modellierungselement von DE-I dar. Unterschieden werden Werte primitiver Typen (Literale) und Instanzen von Klassen.

5.3.1. Werte primitiver Typen

Werte primitiver Typen (auch Literale) sind Elemente aus dem Wertebereich primitiver Datentypen. Die Gleichheit auf den Elementen ist durch eine spezielle Funktion definiert, welche ausschließlich die Struktur des Werts in Betracht zieht. Für jeden primitiven Wert existiert zumindest eine lexikalische Hauptrepräsentation, die Standardrepräsentation genannt wird. Die Literale in DE lehnen sich dabei an die Repräsentation von XML Schema an:

- `Integer.` Werte sind die ganzen Zahlen.
  Syntax: `(PLUS|MINUS)? ZIFFER+`
  Beispiele: `-4, 0, 42`

- `Double.` Werte sind Fließkommazahlen.
  Syntax: `(PLUS|MINUS)? ZIFFER+ PUNKT ZIFFER+`
  Beispiele: `-0.22, 0.0, 42.1`

- `String.` Werte sind endliche Zeichenketten, die durch doppelte Anführungszeichen terminiert sind.
  Syntax: `ANFUEHRUNGSZEICHEN ZEICHEN* ANFUEHRUNGSZEICHEN`
  Beispiel: `"Dies ist eine Zeichenkette"`

- `Boolean.` Werte sind die beiden Wahrheitswerte in englischer Sprache, begrenzt durch spitzen Klammern.
  Syntax: <`true`> | <`false`>

- `Date.` Werte sind Datumsangaben im Format <YYYY-MM-DD>.
  Syntax: < `ZIFFER ZIFFER ZIFFER ZIFFER MINUS ZIFFER ZIFFER MINUS ZIFFER ZIFFER` >
  Beispiele: <`1976-03-12`>, <`2005-01-01`>

- `Time.` Werte sind Zeitangaben im Format <HH:MM:SS.sss>.
  Syntax: < `ZIFFER ZIFFER DOPPELPUNKT ZIFFER ZIFFER (DOPPELPUNKT ZIFFER ZIFFER (PUNKT ZIFFER ZIFFER ZIFFER)?)?` >
  Beispiele: <`10:15`>, <`20:15:10.43`>

- `DateTime.` Werte sind Zeitpunktangaben im Format bestehend aus einem `Date`- und einem `Time`-Werte (ohne spitze Klammern), welche durch ein `T` getrennt sind und zusammen in spitze Klammern gesetzt werden.
  Beispiel: <`1985-09-12T10:15:10`>

- `Duration.` Werte sind Zeitlängenangaben im Format <PyYm_1MdDThHm_2MsS>. Hierbei steht y für die Anzahl der Jahre, m_1 für die Anzahl der Monate, d für

die Anzahl der Tage, h für die Anzahl der Stunden, m_2 für die Anzahl der Minuten, s für die Anzahl der Sekunden und Bruchteile von Sekunden. s ist eine Fließkommazahl, alle anderen sind Ganzzahlen. Werte, die 0 sind, können zusammen mit ihrem großbuchstabigen Markierer ausgelassen werden. Alle Werte beginnen mit dem Startsymbol `P` (für engl. period) und haben das Trennzeichen `T`.
Beispiele: <`P1M10DT`> für 1 Monate und 10 Tage, <`PT10M`> für 10 Minuten, <`PT1M10.5S`> für 1 Minute, 10 Sekunden und 500 Millisekunden.

Die formale Semantik von Literalen findet sich in den Regeln 8.2 bis 8.3.

5.3.2. Instanzen von Klassen

Jede Instanz steht stellvertretend für ein Individuum der realen Welt und gehört eindeutig zu einer speziellsten Klasse K und damit implizit zu all ihren Oberklassen (siehe formale Semantik in Regel 8.1). Instanzen müssen nach Eigenschaft 4.8 in zwei Arten eingeteilt werden: benamte und anonyme Instanzen.

- **Benamte Instanzen.** Instanzen von Entitätsklassen sind stets benamte Instanzen (siehe Regel 8.45). Sie stehen stellvertretend für reale Individuen, welche als Ganzes bzw. als Einheit angesehen werden. Wie alle Instanzen sind auch benamte Instanzen eindeutig von einer speziellsten Klasse abgeleitet, welche auch die Ontologiezugehörigkeit bestimmt. Anfragen nach benamten Instanzen sind über deren Namen („genau die“) oder deklarativ über deren ausgefüllte Attribute („so eine“) möglich.

- **Anonyme Instanzen.** Instanzen von wertbestimmten Klassen sind stets anonyme Instanzen (siehe Regel 8.46). Sie stehen stellvertretend für reale Individuum, die über ihre Attribute definiert sind und nicht als Einheit angesehen werden. Sie haben keinen Namen und werden in keinem Instanzenpool abgelegt, sondern können zu jeder Zeit angelegt werden. Wie jede Instanz sind auch anonyme Instanzen eindeutig von einer speziellsten Klasse abgeleitet. Anfragen nach anonymen Instanzen sind nur über deren ausgefüllte Attribute („so eine“) möglich.

Die formale Semantik von benamten und anonymen Instanzen findet sich in den Regeln 8.31 bis 8.37.

Definition benamter Instanzen

In g-dsd können benamte Instanzen der speziellsten Klasse k nur innerhalb der Ontologie definiert werden, welche die Klasse k enthält. Sie werden als weißes Kästchen dargestellt, welches den Namen der Instanz und die Klassennamen abgetrennt durch einen Doppelpunkt enthält. Es wird eine nicht-fette, unterstrichene Normalschrift verwendet. Abbildung 5.12 zeigt die Definition von Instanzen in g-dsd.

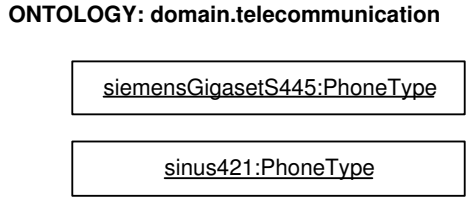


Abbildung 5.12.: Definition zweier benamter Instanzen in g-dsd.

Definition anonymer Instanzen

Da anonyme Instanzen nicht unter einem Namen veröffentlicht werden, können sie überall definiert werden (z.B. auch während der Aufstellung einer Dienstbeschreibung). Sie besitzen keinen Namen, jedoch einen eindeutigen Typ.

Die Definition einer anonymen Instanz in g-dsd ähnelt der Definition einer benamten Instanz, nur wird der Name der Instanz weggelassen. Abbildung 5.13 zeigt ein Beispiel.

Abbildung 5.13.: Definition zweier anonymer Instanzen in g-dsd.

Füllen von Attributen

Instanzen sind eindeutig von einer speziellsten Klasse abgeleitet, d.h. für sie ist eindeutig eine Menge von Attributen definiert, die ausgefüllt werden können. Attribute werden stets mit Instanzen oder Werten des Zieltyps oder eines Untertyps davon gefüllt. Diese werden als *Füllwert* bezeichnet. Nicht ausgefüllte Attribute sind möglich; sie gelten dann als unbekannt.

Orthogonale Attribute dürfen nie gefüllt werden. Für sie muss eine Instanz des zugehörigen verdinglichten Zustands angelegt werden (siehe Abschnitt 5.2.2), welche dann implizit den Füllwert definiert (siehe Regel 8.19).

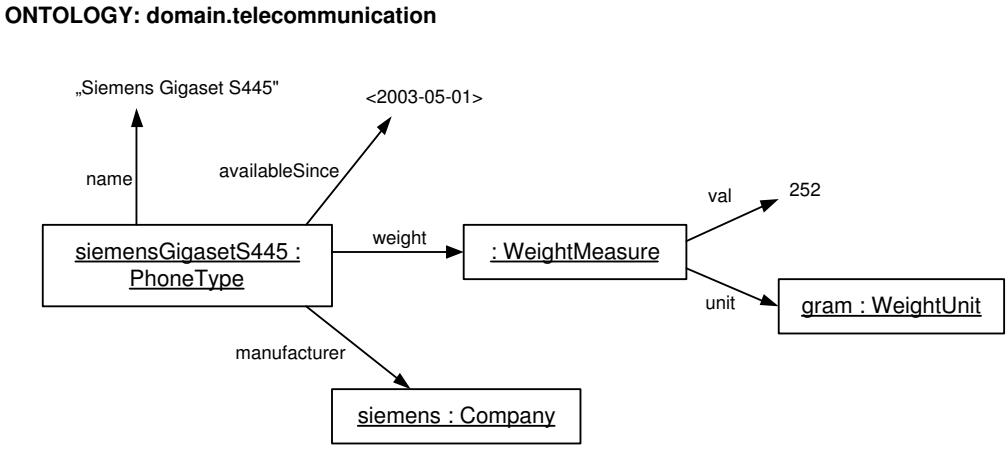


Abbildung 5.14.: Ausfüllen von Attributen in g-dsd.

In g-dsd wird das Füllen von Attributen durch Pfeile notiert, welche von der Instanz auf den Füllwert zeigen und mit dem Namen des Attributs beschriftet sind. Die Quellontologie muss nur dann angegeben werden, wenn sie sich nicht eindeutig aus dem Schema ableiten lässt. Abbildung 5.14 zeigt das Beispiel von oben in g-dsd. Es gilt, dass das Auftauchen einer benamten Instanz nur dann zu einer Neudefinition führt, wenn diese nicht als Füllwert dient (d.h. keine Pfeile auf sie weisen).

5.3.3. Anforderungen an das Füllen von Attributen

Beim Anlegen neuer benamter und anonymer Instanzen gelten bestimmte Anforderungen an das Füllen der Attribute:

- Da wertbestimmte Klassen allein durch die Füllwerte ihrer Attribute bestimmt sind, müssen bei der Definition anonymer Instanzen *alle* Attribute gefüllt werden. Hierdurch kann beispielsweise für OUT-Variablen wertbestimmter Typen erwartet werden, dass die gelieferte Instanz in allen Attributen definiert sein wird.

- Bei der Definition von benamten Instanzen müssen alle definierenden Attribute gefüllt werden, abgeleitete Attribute können gefüllt werden. Orthogonale Attribute dürfen nie gefüllt werden. Für sie können Instanzen des zugehörigen verdinglichten Zustands angelegt werden, welche dann implizit die Füllwerte definiert.

5.3.4. Öffentliche und private benamte Instanzen

Bisher stellten benamte Instanzen ein gedankliches Konstrukt dar, mit denen Individuen der realen Welt perfekt abgebildet werden. Probleme durch eine verteilte Speicherung wurden dabei zunächst ignoriert. Es wurde angenommen, dass jede benamte Instanz ein Individuum korrekt, vollständig und eindeutig repräsentiert. Sie hat einen global eindeutigen Namen und ist für alle Teilnehmer des Systems jederzeit bekannt. Mit diesen Annahmen kann die Gleichheit von Instanzen auf ihre Identität zurückgeführt werden: Zwei benamte Instanzen sind gleich, genau dann wenn sie identisch sind. Die folgende fundamentale *Bijektionsforderung* ist demnach für diese perfekten Instanzen erfüllt:

> **Bijektionsforderung**
> Zwei benamte Instanzen sind genau dann gleich, wenn sie dasselbe Individuum der realen Welt beschreiben.

Für die Praxis sind diese Annahmen jedoch unrealistisch. Durch die hohe Anzahl von Individuen entstünde ein enormer Einigungsaufwand innerhalb der Anwendergruppe, der die Skalierbarkeit gefährden würde. Typischerweise wird daher die Aufgabe der Wissensbeschaffung und der Instanzspeicherung (zumindest teilweise) unter den Mitgliedern des Systems aufgeteilt. Von einer gedanklich perfekten benamten Instanz existieren dann an verschiedenen Orten und zu verschiedenen Zeiten mehrere mehr oder weniger unvollständige und fehlerhafte und damit nicht-identische *Kopien*.

Die oben vorgestellte Bijektionsforderung ist für Kopien von benamten Instanzen nicht mehr gültig, da es zu einem Individuum mehrere beschreibende Kopien geben kann. Benötigt wird demnach eine Definition von Gleichheit auf Kopien, die über die strikte Identität hinausgeht und die so beschaffen ist, dass zwei Kopien genau dann gleich sind, wenn sie dasselbe Individuum beschreiben.

Die Definition von Gleichheit wird im nächsten Abschnitt vorgestellt. Sie basiert auf folgender Unterscheidung zwischen Instanzen von öffentlichem und privatem Interesse (vgl. Eigenschaft 4.4):

- *Öffentliche Instanzen* sind für alle Mitglieder der Anwendergruppe von Bedeutung. Es ist daher sinnvoll, eine gemeinsame *zentrale Kopie* der Instanz im *öffentlichen Instanzenpool* abzulegen und diese ist mit einem global eindeutigen Namen zu versehen. Alle definierenden Attribute müssen gefüllt sein, weitere ableitbare Attribute können gefüllt sein. Name und Füllwerte unterliegen dem Einigungsprozess der gesamten Anwendergruppe.
- *Private Instanzen* hingegen sind nur für einzelne Mitglieder der Anwendergruppe von Interesse. Für sie wird *keine zentrale Kopie* im öffentlichen Instanzenpool hinterlegt.

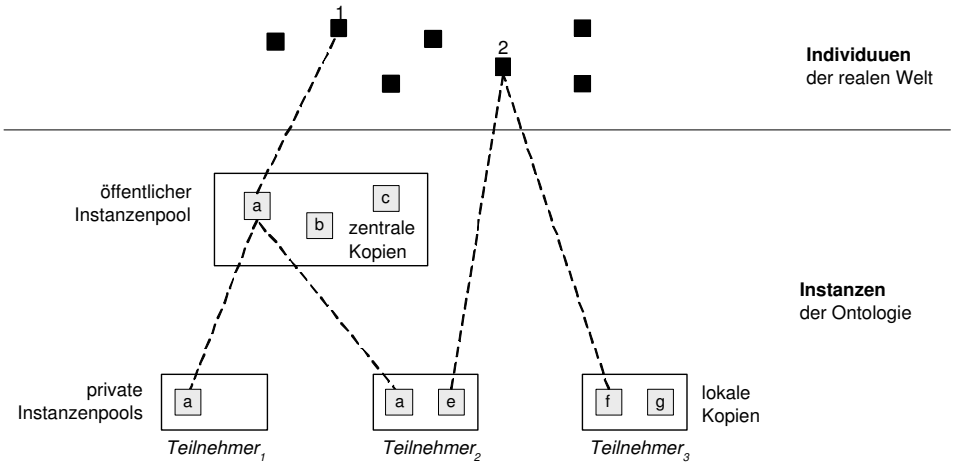


Abbildung 5.15.: Unterscheidung von öffentlichen und privaten Instanzenpools bzw. zentralen und lokalen Kopien.

Generell können von allen Instanzen *lokale Kopien* im privaten Instanzenpool abgelegt werden, wie in Abbildung 5.15 beispielhaft dargestellt. Existiert keine zentrale Kopie, unterliegt dies keinen Einschränkungen: einerseits ist der Name frei wählbar, solange er im persönlichen Pool eindeutig ist, andererseits können die Attribute beliebig gefüllt werden. Der Teilnehmer sollte dabei jedoch nach bestem Wissen handeln. Im Beispiel ist die Instanz für Individuum 2 nicht von globalem Interesse. Teilnehmer 2 und 3 legen daher unabhängig voneinander die unterschiedlichen private Kopien `e` und `f` an. Ist hingegen eine zentrale Kopie im öffentlichen Instanzenpool enthalten, unterliegt eine lokale Kopie für dasselbe Individuum Restriktionen: sie muss den Namen und die existierenden Füllwerte übernehmen und darf nur die noch ungefüllten ableitbaren Attribute frei füllen. Im Beispiel ist die Instanz für Individuum 1 von allgemeinem Interesse. Sie liegt daher als zentrale Kopie `a` im öffentlichen Instanzenpool vor. Die Teilnehmer 1 und 2 haben davon lokale Kopien mit gleichem Namen erstellt.

Die Unterscheidung zwischen privaten und öffentlichen benamten Instanzen hängt eng mit der Unterscheidung in öffentliche und teilöffentliche Klassen zusammen (siehe Abschnitt 5.2.5). Es gilt, dass öffentliche Klassen nur öffentliche Instanzen enthalten dürfen, während teilöffentliche Klassen sowohl öffentliche als auch private Instanzen enthalten können (siehe Regel 8.49).

5.3.5. Gleichheit von Instanzen

Da Instanzen in der Praxis in Form zentraler oder lokaler Kopien vorkommen, muss die Definition ihrer Gleichheit auf diesen Kopien erfolgen. Die Gleichheit muss so definiert werden, dass sie die in Abschnitt 5.3.4 eingeführte Bijektionsforderung möglichst

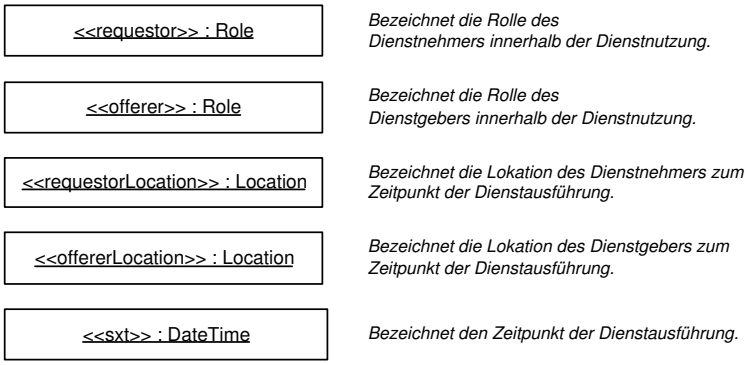


Abbildung 5.16.: Einige vordefinierte Instanzen in g-dsd.

gut erfüllt. Sie wird durch das Symbol `==` notiert und hängt von der Art der Instanz ab:

- Die Gleichheit von Literalen (als Instanzen primitiver Typen) wird durch die vordefinierte Gleichheitsfunktion bestimmt. Dabei ist nicht unbedingt eine zeichengenaue Gleichheit nötig, z.B. gilt auch `<PT1H> == <PT60M>` oder `1.00 == 1.0` (Regel 8.50).

- Zwei Kopien benamter öffentlicher Instanzen sind genau dann gleich, wenn sie den gleichen Namen haben. Die erfüllt die Bijektionsforderung, da alle lokalen Kopien, die sich auf dasselbe Individuum beziehen, den von der zentralen Kopie abgeleiteten gleichen Namen besitzen (Regel 8.51).

- Die Gleichheit benamter privater Instanzen ist komplexer, da sie nicht auf einen gleichen Namen zurückgeführt werden kann. Sie wird daher durch eine klassenspezifische Gleichheitsfunktion, die die Füllwerte der Attribute heranzieht, bestimmt. In dieser Arbeit ist die Gleichheit wie folgt definiert: Zwei Kopien benamter privater Instanzen sind genau dann als gleich anzusehen, wenn sie den gleichen Typ besitzen und die jeweils gleichen Füllwerte für die definierenden Attribute haben (Regel 8.52). Dies ist problematisch, da die Kopien unabhängig voneinander gewartet werden und so nicht immer gleich sind, obwohl die für dasselbe Individuum stehen.

- Zwei anonyme Instanzen sind genau dann gleich, wenn sie den gleichen Typ besitzen und die Füllwerte der korrespondierenden Attribute gleich sind (Regel 8.53).

- Eine anonyme Instanz und eine benamte Instanz sind immer ungleich, da sie nach Abschnitt 5.2.4 von unterschiedlichen Typen sein müssen (Regel 8.54).

Nr.	Eigenschaft	Umsetzung
4.1	Intuitive Modellierung	Sprachbasis: Objektorientierung
4.2	Modularisiertes Schema	Möglichkeit zur Zerlegung des Schemas
4.3	Anwendungsneutrales Schema	als Modellierungsrichtlinie
4.4	Öff. und priv. Instanzen	durch verschiedene Instanzenpools
4.5	Nur rigide Klassen im Schema	als Modellierungsrichtlinie
4.6	Extrinsische Eigenschaften	durch Metaeigenschaft `orthprop`
4.7	Zustandsklassen	als Unterklassen von State
4.8	Wertbestimmte und Entitätsklassen	durch Metaeigenschaften `V` und `E`
4.9	Def. und ableitbare Attribute	durch Metaeigenschaften `def-`/`incprop`

Abbildung 5.17.: Eigenschaften für DE-I und ihre Realisierung.

Die Definition der Gleichheit für Kopien benamter privater Instanzen ist problematisch, da sie nicht in jedem Fall die Bijektionsforderung erfüllt. Beim Vergleich von Dienstbeschreibungen kann es so zu Fehlinterpretationen kommen, die das Vergleichsergebnis verfälschen. Ein möglicher Ausweg kann sein, neben dem Ergebnis der Gleichheitsuntersuchung auch einen Konfidenzwert für die Korrektheit abzuschätzen. Dieser könnte bei der Auswahl eines Dienstes berücksichtigt werden.[3] Das Problem einer unzuverlässigeren Gleichheitsbestimmung existiert für öffentliche benamte Instanzen nicht, was durch einen höheren Wartungsaufwand aufgrund der gemeinsam betreuten zentralen Kopie erkauft werden muss.

Da benamte Instanzen nur in Form von Kopien existieren, sind mit dem Ausdruck „Instanz“ im Folgenden immer auch die konkreten Kopien gemeint.

5.3.6. Vordefinierte Instanzen

In Dienstbeschreibungen muss teilweise auf konkrete Teilnehmer, Lokationen und Zeitpunkte der Dienstnutzung eingegangen werden. Hierzu steht eine Reihe von vordefinierten Instanzen zur Verfügung. Sie sind durch doppelte spitze Klammern gekennzeichnet. Einige Beispiele zeigt Abbildung 5.16.

5.4. Zusammenfassung

In diesem Kapitel wurde *DIANE Elements I* vorgestellt, eine generische, objektorientierte Ontologiesprache, die den für Dienstbeschreibungen wichtigen Konflikt zwi-

[3] Dieser Konfidenzwert darf jedoch nicht mit der Präferenz des Benutzers verwechselt oder gar mit ihr verrechnet werden, wie dies einige Vergleicher in der Literatur tun, sondern muss als orthogonale Größe angesehen werden.

schen unabhängiger Erstellbarkeit und akzeptablem Einigungsaufwand angeht, indem sie eine Trennung zwischen zentralen und lokalen Ontologienteilen vornimmt. Dazu wurde beim Entwurf der Sprache darauf geachtet, dass diese die Eigenschaften aus Abschnitt 4.2 besitzt. Die Tabelle in Abbildung 5.17 zeigt im Überblick, wie (und in welchem Abschnitt) diese konkret umgesetzt wurden: Einerseits wurden in DE-I Modellierungsrichtlinien, andererseits konkrete sprachliche Mittel in Form von Metaeigenschaften eingeführt, um die Eigenschaften zu erzielen. Zu nennen ist insbesondere die Trennung in definierende, ableitbare und orthogonale Attribute, die Trennung in öffentliche Instanzen mit zentralen Kopien und private Instanzen mit lokalen Kopien sowie die Trennung in wertbestimmte und Entitätsklassen. Alle Elemente sind bewusst dienst- und anwendungsneutral. Mit der Einführung von DE-II im nächsten Kapitel wird DIANE Elements um dienstspezifische Elemente erweitert.

6. Dienstspezifische Ontologiesprache: DIANE Elements II

In Kapitel 4 wurde dargelegt, dass eine geeignete Beschreibung von Diensten nur mit einer Reihe von Spracherweiterungen möglich ist, die auf die besonderen Eigenschaften von Diensten abgestimmt sind. Hierzu wird in dieser Arbeit die Sprache *DIANE Elements II* (DE-II) eingeführt. Sie setzt auf DE-I auf und enthält die dienstspezifischen Spracherweiterungen.

Für die neuen Sprachelemente ist eine klar definierte Semantik unerlässlich. Einerseits muss der Benutzer beim Aufstellen von Dienstbeschreibungen deren Aussage genau kennen, andererseits müssen die Komponenten des Systems (insbesondere der Vergleicher) eine genaue Beschreibung davon haben, wie die Elemente zu verarbeiten sind. Eine solche formale Semantik ist für die Sprachelemente in DE-II durch eine axiomatische Abbildung auf Prädikatenlogik definiert und findet sich in Kapitel 8. Die Semantik ist zunächst dienstneutral, d.h. enthält keine Verweise auf bestimmte Zeitpunkte und Teilnehmer einer bestimmten Dienstnutzung. Erst die Verwendung innerhalb einer konkreten Dienstbeschreibung verleiht ihnen eine dienstspezifische Semantik.

Im folgenden Kapitel werden die neuen Sprachelemente im Detail vorgestellt. Keines dieser Elemente wird zum Aufstellen von Domänenontologien benötigt. Sie wurden speziell zum Einsatz in Dienstbeschreibungen entworfen:

- *Aggregierende Elemente* zur Erfassung der Tatsache, dass Dienste häufig mehrere ähnliche Effekte zusammenfassen. Diese werden in DE-II durch das Konzept deklarativer *Mengen* erfasst. Die Details finden sich in Abschnitt 6.1.
- *Bewertende Elemente* zur Erfassung der persönlichen Präferenzen eines Teilnehmers unter verschiedenen wählbaren Effekten. In DE-II wird dies auf das Konzept *unscharfer Mengen* abgebildet. Die Details finden sich in Abschnitt 6.2.
- *Selektierende Elemente* zur Erfassung der Tatsache, dass die genaue Wirkung eines Dienstes vom Dienstnehmer konkretisiert werden muss. In DE-II wird

dies durch das Konzept der *Variablen* erreicht. Die Details finden sich in Abschnitt 6.3.

- *Operationale Elemente* zur Darstellung einer bedingten Wirkung in der Ontologie. In DE-II werden sie durch die beiden *Operatoren* <<precondition>> und <<effect>> dargestellt. Die Details finden sich in Abschnitt 6.4.

6.1. Mengen

Mengen von Instanzen sind ein wichtiges Konzept in DE-II. Sie verwirklichen die Idee aggregierender Sprachelemente, die eingesetzt werden, um mehrere Instanzen zusammenzufassen. Im Rahmen von Dienstbeschreibungen ist dies allgegenwärtig und tritt an folgenden Stellen auf:

- *In Angebotsbeschreibungen.* Oft kann ein Dienst nicht nur einen einzigen Effekt erzielen, sondern ist prinzipiell in der Lage, eine Reihe von Effekten zu erreichen, die durch einen oder mehrere Parameter konkretisiert werden können. Aus diesem Grund muss eine Dienstbeschreibung als Menge von Instanzen angesehen werden, aus der vor und während der Dienstnutzung Elemente ausgewählt werden.

- *In Anfragebeschreibungen.* Der Dienstnehmer will in der Regel eine bestimmte Funktionalität erbracht wissen und denkt dabei nicht zwangsläufig an einen bestimmten, einzelnen Dienst. Typischerweise sind auch mehrere unterschiedliche Dienste zur Erbringung dieser Funktionalität geeignet. Es ist daher für ihn wichtig, eine Menge geeigneter Dienste angeben zu können.

Mengen stellen ein Zwischending zwischen Klassen und Einzelinstanzen dar: Klassen stehen stellvertretend für *alle* Individuen eines Typs, Instanzen stehen für *genau ein* Individuum. Instanzmengen hingegen enthalten eine beliebige Sammlung von Instanzen und können daher keine, eine, mehrere oder alle Instanzen eines Typs enthalten. Im Gegensatz zu Klassen, welche Instanzen zu rigiden Konzepten von allgemeinem Interesse bündeln, sind Mengen meist temporär, nicht-rigide und nur von lokalem Interesse. Wie alle Sprachelemente aus DE-II werden Mengen daher nicht zur Definition von Domänenontologien verwendet, sondern treten ausschließlich in Dienstbeschreibungen auf.[1] Abbildung 6.1 zeigt den Zusammenhang am Beispiel der Klasse Company graphisch. In g-dsd werden Mengen wie Klassen notiert, die einen kleinen Querstrich in der linken oberen Ecke besitzen. Ihre formale Semantik findet sich in Abschnitt 8.3.

[1] Denkbar ist jedoch, dass Mengen von allgemeinem Interesse auch in der Domänenontologien unter einem Namen veröffentlicht werden und so allen Teilnehmern zur Verfügung stehen.

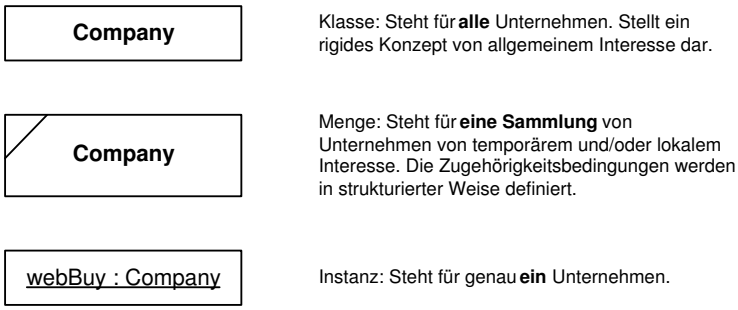


Abbildung 6.1.: Mengen als Zwischending zwischen Klassen und Instanzen.

Die Bestimmung, welche Instanzen zu einer Menge gehören, erfolgt in DE-II deklarativ, d.h. durch Angabe einer Funktion, welche für jede Instanz die Zugehörigkeit zur Menge bestimmt. Lässt man für eine solche *Zugehörigkeitsfunktion* beliebige Funktionen zu, so erhält man einerseits zwar die größtmögliche Ausdruckskraft, andererseits führt dies jedoch zu enormen Problemen: Erstens ist das Aufstellen solcher Funktionen für menschliche Benutzer sehr schwierig, da keinerlei Vorgaben existieren, zweitens kann der Vergleicher mit beliebigen Funktionen nur schwer umgehen, insbesondere wenn Funktionen in Anfragebeschreibungen auf Funktionen in Angebotsbeschreibungen treffen.

In DE-II wurde daher bewusst eine Strukturierung (und damit auch Einschränkung) der möglichen Zugehörigkeitsfunktionen vorgenommen, indem bestimmte Beschreibungselemente zur Definition von Mengen vorgegeben wurden. Unterschieden werden Bedingungen, die die prinzipielle Elementzugehörigkeit regeln, und Strategien, mit denen Ausnahmen bei leicht abweichenden Instanzen formuliert werden können.

6.1.1. Typbedingungen

Die Typbedingung bestimmt den Typ, den die Elemente der Menge haben müssen, d.h. eine Instanz kann nur dann Element der Menge sein, wenn sie von der in der Bedingung angegebenen Klasse abgeleitet ist. Die Angabe einer Typbedingung ist für eine Menge verpflichtend. Sie findet sich als Name der Klasse im Kästchen. Stammt die Klasse aus einer fremden Ontologie, wird deren Name wie gewohnt in kleiner, kursiver Schrift über dem Klassennamen notiert. Abbildung 6.2 zeigt drei Beispiele.

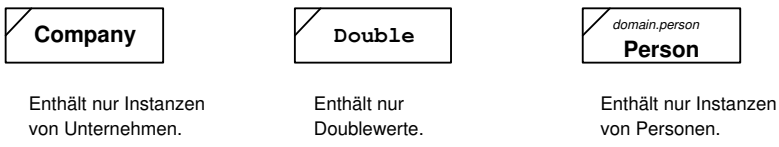


Abbildung 6.2.: Notation von Mengen und Typbedingung in g-dsd.

Die formale Semantik von Typbedingungen findet sich in Regel 8.72.

Man bezeichnet Mengen,

- ...die einen primitiven Typ als Typbedingung besitzen, auch als primitive Menge (kurz *Prim-Menge*),
- ...die eine wertbestimmte Klasse als Typbedingung besitzen, auch als wertbestimmte Menge (kurz *V-Menge*),
- ...die eine teilöffentliche Entitätsklasse als Typbedingung besitzen, auch als Entitätsmenge (kurz *E-Menge*),
- ...die eine öffentliche Entitätsklasse als Typbedingung besitzen, auch als öffentliche Entitätsmenge (kurz *PE-Menge*).

6.1.2. Direkte Bedingungen

Direkte Bedingungen einer Menge bezeichnen Bedingungen, die direkt an die potenziellen Elemente (d.h. nicht an deren Attribute) gerichtet sind. Hierzu stehen alle Vergleichsoperatoren wie `==`, `<=` etc. zur Verfügung. Zudem kann mit dem Operator `in` auf Enthaltensein in einer aufgelisteten Menge verglichen werden. Zur Berechnung des Vergleichswerts können auch domänenspezifische Operationen und vordefinierte Instanzen verwendet werden. Insgesamt können null, eine oder mehrere direkte Bedingungen angegeben werden. Eine Instanz kann nur dann zur Menge gehören, wenn sie alle diese Bedingung erfüllt, d.h. die Bedingungen werden konjunktiv verknüpft.

In g-dsd werden direkte Bedingungen einer Menge in das Kästchen aufgenommen und durch eine Linie abgetrennt. Die Syntax ist dabei `<Operator> <Vergleichsinstanz/-wert>`. Abbildung 6.3 zeigt drei Beispiele.

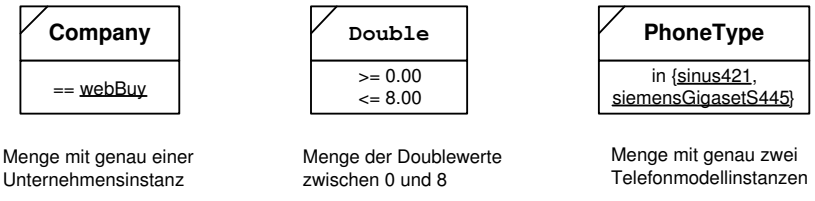


Abbildung 6.3.: Beispiele für Mengen mit direkten Bedingungen.

Die formale Semantik von direkten Bedingungen findet sich in den Regeln 8.55 bis 8.56 und 8.74 bis 8.75.

6.1.3. Attributbedingungen

Für Mengen von nicht-primitiven Typen stehen neben direkten Bedingungen auch Attributbedingungen zur Verfügung. Diese überprüfen potenzielle Elemente anhand ihrer ausgefüllten Attribute. Attributbedingungen werden immer aus definierenden, ableitbaren oder orthogonalen[2] Attributen der Klasse ihrer Typbedingung erstellt. Sie übernehmen daher den Namen des Attributes und haben zudem eine *Zielmenge*. Es gilt: Eine Instanz kann nur dann Element der Menge sein, wenn das Attribut der Bedingung bei ihr gefüllt ist und dieser Füllwert Element der Zielmenge ist. Hierdurch können verschachtelte, deklarative Mengen aufgebaut werden. Attributbedingungen sind optional. Treten mehrere auf, werden diese standardmäßig konjunktiv verknüpft. Dies kann durch eine alternative Verbindungsstrategie geändert werden (siehe dazu auch Abschnitt 6.1.5).

In g-dsd werden Attributbedingungen durch einen Pfeil notiert, der den Namen des Attributs trägt und auf die Zielmenge zeigt. Abbildung 6.4 zeigt ein Beispiel.

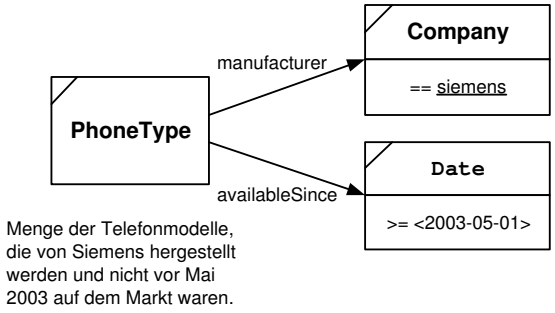


Abbildung 6.4.: Beispiele für eine Menge mit Attributbedingungen.

Die formale Semantik von Attributbedingungen findet sich in den Regeln 8.76 bis 8.77.

6.1.4. Fehlstrategien

Es gilt, dass eine Instanz nur dann eine Attributbedingung erfüllen kann, wenn bei ihr das geforderte Attribut auch gefüllt ist. Eine alternative Fehlstrategie modifiziert dieses Verhalten. Es gibt drei Arten von Fehlstrategien:

[2] Obwohl orthogonale Attribute einer Instanz nicht direkt gefüllt werden können, sind deren Füllwerte doch indirekt über die zugehörigen verdinglichten Zustandsklasse definiert (siehe Regel 8.19). Zusammengefasst gilt daher: x ist dann Element einer Menge mit der Attributbedingung über ein orthogonales Attribut mit der Zielmenge z, wenn gilt, dass es ein Element e der Zielmenge z gibt, welches das entity-Attribut mit x gefüllt halt.

- `assume_failed`. Bezeichnet den Standardfall. Wenn der entsprechende Füllwert fehlt, wird die Attributbedingung als fehlgeschlagen angesehen, d.h. der Wahrheitswert `false` wird angenommen.
- `assume_fulfilled`. Wenn der entsprechende Füllwert fehlt, wird die Attributbedingung als erfüllt angesehen, d.h. der Wahrheitswert `true` wird angenommen.
- `ignore`. Wenn der entsprechende Füllwert fehlt, wird die Attributbedingung ignoriert, d.h. der zusätzliche Wahrheitswert `neutral` wird angenommen.[3]

Unterschiede zwischen den Rückgabewerten `neutral` und `true` treten nur bei geänderter Verbindungsstrategie auf (siehe Abschnitt 6.1.5).

Aufgrund der Forderungen aus Abschnitt 5.3.3 müssen bestimmte Attribute stets gefüllt werden. Die Angabe von Fehlstrategien für Bedingungen an solche Attribute ist daher nicht sinnvoll. Daher sind Fehlstrategien nur für Bedingungen über abgeleitete oder orthogonale Attribute von Entitätsklassen sinnvoll.

In g-dsd wird die Fehlstrategie durch eine Markierung am Pfeil der Attributbedingung kenntlich gemacht. Hierbei steht ein Minuszeichen (oder keine Markierung) für den Standardfall `assume_failed`, eine Pluszeichen für `assume_fulfilled` und ein ausgefüllter Kreis für `ignore`. Abbildung 6.5 zeigt ein Beispiel für die `ignore`-Strategie.

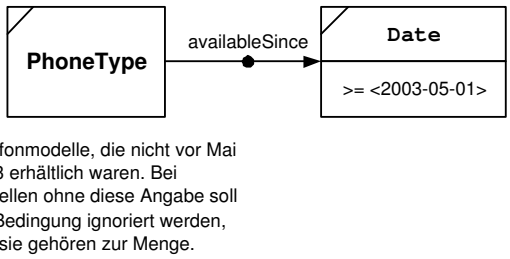


Abbildung 6.5.: Beispiel für die Fehlstrategie `ignore`.

Die formale Semantik von Fehlstrategien findet sich in den Regeln 8.57 bis 8.59 und 8.78 bis 8.80.

6.1.5. Verbindungsstrategien

Die Verbindungsstrategie modifiziert das Verhalten, wie die einzelnen Attributbedingungen verknüpft werden. Standardmäßig werden diese konjunktiv verbunden. Die

[3] `neutral` ist das neutrale Element für alle booleschen Operationen. Es gilt dabei: $x \wedge$ `neutral` $= x$, $x \vee$ `neutral` $= x$, $\neg$`neutral` $=$ `neutral` mit $x \in \{$`true`, `false`, `neutral`$\}$.

Strategie stellt einen booleschen Ausdruck bestehend aus den als Attributbedingung auftretenden Attributnamen und den Operatoren `and` und `or` dar. Es gilt: Eine Instanz kann nur Element einer Menge sein, wenn der Ausdruck der Verbindungsstrategie nicht zu `false` evaluiert, wobei die Attributnamen durch die Einzelergebnisse der entsprechenden Attributbedingungen (also `true`, `false` oder `neutral`) ersetzt werden.

Auf die Negation `not` wurde bei der Definition von Verbindungsstrategien bewusst verzichtet, da dies die Komplexität des Vergleichsvorgangs erheblich erhöhen würde. Ausgehend von einem Knoten könnten dann die als Attributbedingungen abgehenden Kanten nicht mehr gleich behandelt werden, sondern deren Einfluss auf das Gesamtergebnis hinge davon ab, ob sie durch eine modifizierte Verbindungsstrategie negiert würden. Die Negation ist jedoch durch den `!=`-Operation bei direkten Bedingungen für Blattknoten möglich.

In g-dsd wird eine alternative Verbindungsstrategie in kursiver Schrift in das Kästchen der Menge geschrieben und durch eine horizontale Linie abgetrennt. Abbildung 6.6 zeigt ein Beispiel.

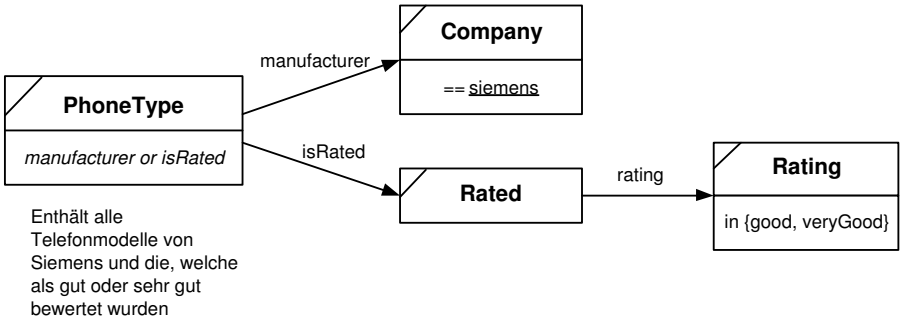


Abbildung 6.6.: Beispiel für eine alternative Verbindungsstrategie.

Die formale Semantik der Verbindungsstrategie findet sich in den Regeln 8.60, 8.61, 8.81 und 8.82.

6.1.6. Typvergleichsstrategien

Standardmäßig müssen die Elemente einer Menge vom Typ der Menge oder einem Untertyp von diesem sein. Durch Angabe einer alternativen Typvergleichsstrategie kann dies so abgeschwächt werden, dass auch Oberklassen erlaubt sind. Folgende Strategien stehen zur Verfügung:

- =. Bezeichnet den Standard, bei dem nur eine exakte Typübereinstimmung bzw. eine Unterklassenbeziehungen erlaubt ist.

- `super`. Elemente der Menge dürfen auch von einer Oberklasse der Mengenklasse instanziiert sein.
- `super[`n`,1]`. Elemente der Menge dürfen auch von einer Oberklasse der Mengenklasse instanziiert sein, aber maximal über n Vererbungsbeziehungen. Die 1 ist nur für den Fall unscharfer Mengen relevant und gibt die Abschwächung des Zugehörigkeitswerts pro Vererbungsbeziehung an (siehe Abschnitt 6.2.4).

In g-dsd wird eine optionale Typvergleichsstrategie direkt im rechten unteren Bereich des Kästchens notiert. Die Strategie = kann dabei als Standard weggelassen werden. Abbildung 6.7 zeigt ein Beispiel.

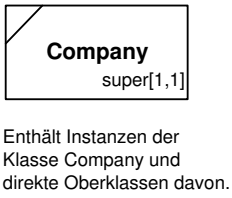


Abbildung 6.7.: Beispiel für eine alternative Typvergleichsstrategie.

Zu beachten ist der Zusammenhang zwischen einer geänderten Typvergleichsstrategie und Attributbedingungen. Es gilt Folgendes: Sei M eine Menge vom Typ T mit einer Typvergleichsstrategie, die Obertypen zulässt. Sei S ein Obertyp von T. Gibt es in M Attributbedingungen bezüglich Attributen, die nicht in S vorkommen, so werden diese für Instanzen mit dem speziellsten Typ S nicht überprüft. Für diese kommt nicht die Fehlstrategie zum Einsatz. Siehe hierzu Regel 8.77. Sei weiterhin R ein Untertyp von S und damit eine Schwesterklasse von T. Ist in R und T ein gleichlautendes Attribut a definiert, das nicht in S vorkommt, so wird dieses nicht herangezogen, falls in M hieran eine Attributbedingung geknüpft ist, die auf einer Instanz aus R nicht überprüft werden kann.

Die formale Semantik der Typvergleichsstrategie findet sich in Regel 8.73.

6.1.7. Test auf Mengenzugehörigkeit

Es gilt, dass genau die Instanzen bzw. Literale zu einer Menge gehören, die alle Bedingungen unter Berücksichtigung der Strategien erfüllen. Dabei wird das Ergebnis der Typbedingung, das aus der konjunktiven Verknüpfung der direkten Bedingungen und das aus den laut Verbindungsstrategie verrechneten Attributbedingungen immer konjunktiv verknüpft. Am Ende steht daher auf jeden Fall einer der booleschen Werte `true` oder `false`.[4] Die formale Semantik findet sich in Regel 8.83.

[4] `neutral` kann nicht mehr vorkommen, da zumindest die Typbedingung nicht `neutral` liefert.

6.2. Unscharfe Mengen

Bei den bisher vorgestellten Mengen handelt es sich um *scharfe* Mengen, d.h. für jede Instanz kann angegeben werden, ob sie sich in der Menge befindet oder nicht. Zum Aufstellen von Anfragebeschreibungen ist dies nicht ausreichend. Hier will der Dienstnehmer über den Grad der Mengenzugehörigkeit seine Präferenzen unter den Elementen der Menge ausdrücken können. Aus diesem Grund sind in DE-II solche *bewertenden Elemente* durch das Konzept der *unscharfen Menge* realisiert. Unscharfe Mengen lassen sich durch eine kontinuierliche Zugehörigkeitsfunktion definieren, die jeder Instanz einen Wert aus dem Intervall $[0.0, 1.0]$ zuweist. Dabei bedeutet 0, dass die Instanz nicht zur Menge gehört, wohingegen ein Wert > 0 eine graduelle Zugehörigkeit angibt. Scharfe Mengen sind ein Spezialfall unscharfer Mengen, die nur die Zugehörigkeiten 0 und 1 kennen. Beide Mengentypen können daher gemischt verwendet werden.

Um unscharfe Mengen definieren zu können, enthält DE-II zusätzliche Beschreibungselemente im Bereich der direkten Bedingungen sowie der drei Strategietypen. Diese werden im Folgenden vorgestellt. Da sich die Definition der formalen Semantik nicht wesentlich von der Semantik scharfer Mengen unterscheidet, ist diese nicht explizit aufgeführt (siehe Abschnitt 8.4).

6.2.1. Unscharfe direkte Bedingungen

Unscharfe direkte Bedingungen für Mengen primitiver Typen werden möglich, indem die in Abschnitt 5.2.1 eingeführten unscharfen Vergleichsoperatoren ∼==, ∼<= etc. als Operatoren zugelassen werden. Auch eine selbstdefinierte Abweichung durch eine Wertangabe in eckigen Klammern ist erlaubt. Die Notation in den drei Repräsentationsformen ändert sich dabei nicht.

Für Mengen von komplexen Typen existieren zwei Möglichkeiten, unscharfe Bedingungen zu formulieren: direkte Angabe von Zugehörigkeitswerten in aufgelisteten Mengen beim `in`-Operator und domänenspezifische Funktionen. Im ersten Fall wird jedem Element der Vergleichsmenge konkret ein Wert aus $[0, 1]$ zugewiesen, der in eckigen Klammern hinter dem Element notiert wird.

Ein Beispiel in g-dsd zeigt Abbildung 6.8. Die Menge in a) enthält Double-Werte zwischen 7 und 8 mit einem Zugehörigkeitswert von 1.0 sowie die Randwerte bis 6 und 9 mit verringerter Zugehörigkeit. b) zeigt eine unscharfe Enumerierung von Telefonmodellen, c) die Verwendung einer domänenspezifischen Funktion. In diesem Beispiel geht es um Städte, die sich in der Nähe von Karlsruhe befinden. Die Ähnlichkeit wurde dabei von einem Domänenexperten festgelegt und könnte etwa die Entfernung per Luftlinie in Betracht ziehen.

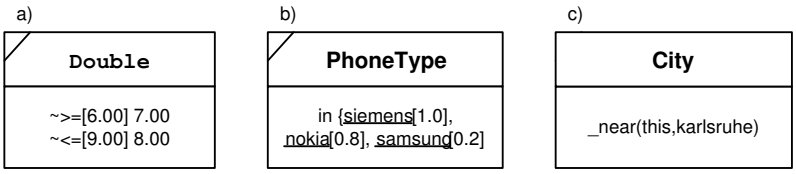


Abbildung 6.8.: Unscharfe direkte Bedingungen in g-dsd.

6.2.2. Unscharfe Fehlstrategien

Für unscharfe Mengen wurden die Fehlstrategien um eine weitere Strategie ergänzt: `assume_value[`n`]`. Im Falle eines fehlenden Füllwerts wird der in Klammern angegebene Wert n als Erfüllungsmaß herangezogen. n stammt dabei aus dem Intervall $[0, 1]$, wobei $n = 0$ der Strategie `assume_failed`, $n = 1$ der Strategie `assume_fulfilled` entspricht.

In g-dsd erfolgt die Notation durch ein Pluszeichen mit dem daneben gestellten Wert n in eckigen Klammern. Ein Beispiel zeigt Abbildung 6.9.

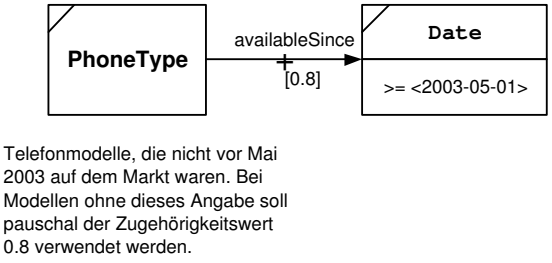


Abbildung 6.9.: Unscharfe Fehlstrategie `assume_value` in g-dsd.

6.2.3. Unscharfe Verbindungsstrategien

Da die Einzelergebnisse der Attributbedingungen in unscharfen Mengen keine booleschen Werte, sondern Zahlen aus dem Intervall $[0, 1]$ sind, muss auch die Vorschrift zu ihrer Verrechnung angepasst werden. `and`-Verknüpfungen werden durch die Multiplikation „$\cdot$“ ersetzt. Für den Spezialfall scharfer Werte, d.h. 0 für `false` und 1 für `true` liefert sie dasselbe Ergebnis. `or`-Verknüpfungen werden durch die in der Literatur häufig verwendete modifizierte Addition „$\oplus$“ ersetzt. Sie ist definiert als $a \oplus b := a + b - ab$ für $a, b \in [0, 1]$ und gewährleistet, dass das Ergebnis stets im Bereich $[0, 1]$ liegt. Auch sie liefert für den Spezialfall scharfer Werte das korrekte Ergebnis. Insgesamt stehen folgende unscharfe Operatoren für Werte $x_1, \ldots, x_n \in [0, 1]$ zur Verfügung. Alle Funktionen sind so gewählt, dass sie monoton steigend in allen Parametern $x_1, \ldots, x_n$ sind. Dadurch wird es möglich, in einem Vergleich die verbundenen Attributbedingungen einzeln zu maximieren:

- *Konjunktion*: (x_1 `and` x_2) bzw. (x_1 `mul` x_2). Steht für die gewöhnliche Multiplikation, d.h. das Ergebnis berechnet sich zu $x_1 \cdot x_2$.
- *Disjunktion*: (x_1 `or` x_2) bzw. (x_1 `add` x_2). Steht für die modifizierte Addition, d.h. das Ergebnis berechnet sich zu $x_1 \oplus x_2$.
- *Extremale Konjunktion*: `min(`x_1`, ..., `x_n`)`. Steht für die Minimalwertbildung, die für scharfe Werte dasselbe Ergebnis liefert wie die Konjunktion.
- *Extremale Disjunktion*: `max(`x_1`, ..., `x_n`)`. Steht für die Maximalwertbildung, die für scharfe Werte dasselbe Ergebnis liefert wie die Disjunktion.
- *Gewichtete Summe*: (λ_1 `*` x_1 `+ ... +` λ_n `*` x_n) wobei $\lambda_1 + \ldots + \lambda_n = 1$. Gewichtet die Einzelergebnisse unterschiedlich und addiert die Werte. Die gewichtete Summe ist dadurch als disjunktive Verrechnung zu werten. Das Ergebnis berechnet sich zu: $\lambda_1 \cdot x_1 + \ldots + \lambda_n \cdot x_n$.
- *Exponentielle Verstärkung*: `exp(`x_1`, `λ`)`. Verstärkt die „Wichtigkeit“ des Wertes x_1 oder verringert sie. $\lambda = 1$ bewirkt keine Änderung, $\lambda < 1$ eine Abschwächung, $\lambda > 1$ eine Verstärkung. Die Berechnung erfolgt zu $(x_1)^\lambda$.

Wird einer der Operanden x_i aufgrund der Fehlstrategie `ignore` zu `neutral` ausgewertet, so wird er als neutrales Element der jeweiligen Operation aufgefasst. Bei `mul`, `add`, `min`, `max` bzw. der gewichteten Summe werden alle auftretenden `neutral`-Operanden ignoriert, außer alle Operanden sind `neutral`; in dem Fall liefert die Operation selbst `neutral`. Bei der gewichteten Summe werden die Gewichte der `neutral`-Operanden gleichmäßig auf die anderen Operanden verteilt. `exp` liefert immer `neutral`, falls der Operand `neutral` ist. Mit der gleichen Begründung wie für scharfe Verbindungsstrategien wurde auf einen unscharfen Negationsoperator bewusst verzichtet.

Aus diesen Operationen kann ein beliebiger Ausdruck aufgebaut und als Verbindungsstrategie eingesetzt werden. Die Notation erfolgt in g-dsd unverändert.

6.2.4. Unscharfe Typvergleichsstrategien

Für die Typvergleichsstrategie existiert eine unscharfe Form, die den Abstand der zu vergleichenden Typen in Betracht zieht:

- `super[`n`,`f`]`. Elemente der Menge dürfen auch von einer Oberklassen der Mengenklasse instanziiert sein, maximal jedoch über n Vererbungsbeziehungen, sonst ist der Zugehörigkeitswert 0. Für eine exakte Typübereinstimmung ergibt sich ein Zugehörigkeitswert von 1, für jede dazwischenliegende Beziehung wird der Zugehörigkeitswert durch eine Multiplikation mit $f \in [0, 1]$ abgeschwächt.

In g-dsd wird eine unscharfe Typvergleichsstrategie direkt in der rechten unteren Ecke des Kästchens notiert.

6.2.5. Test auf unscharfe Mengenzugehörigkeit

Wie beim Test auf scharfe Mengenzugehörigkeit werden die Ergebnisse der Einzelbedingungen (Typ-, direkte und Attributbedingungen) in jedem Fall konjunktiv, d.h. im unscharfen Fall multiplikativ verrechnet. Zur Menge gehören genau die Instanzen, deren so errechneter Zugehörigkeitswert > 0 ist. Der Wert gibt den Grad der Zugehörigkeit an. Instanzen mit einer Zugehörigkeit von 0 gelten als nicht zur Menge gehörig.

6.3. Variablen

Einerseits werden zur Beschreibung von Diensten Sprachelemente benötigt, mit denen mehrere Instanzen aggregiert werden können, andererseits muss ausdrückbar sein, dass aus diesen Mengen bestimmte Instanzen ausgewählt werden können bzw. müssen. In DE-II werden solche *selektierenden Elemente* durch das Konzept der *Variablen* umgesetzt.

Variablen sind immer an eine bestimmte Menge (die so genannte *Grundmenge*) gebunden und können mit einem konkreten Wert gefüllt werden. Im Rahmen von Dienstbeschreibungen kann hiermit beispielsweise ausgedrückt werden, dass bestimmte Informationen vom Dienstnehmer bzw. Dienstgeber bereitzustellen sind. Variablen können als Zielmengen von Attributbedingungen eingesetzt werden. Nach dem Füllen ändert sich dann deren Semantik: Sie wirken dann wie eine einelementige Menge, die genau diesen Wert enthält. Die Semantik von Variablen findet sich in Abschnitt 8.5.

In g-dsd werden Variablen ähnlich wie Mengen dargestellt: Als Kästchen mit einem Querstrich in der linken oberen Ecke. Zur Unterscheidung werden Variablen hellgrau hinterlegt. Zwei Beispiele in g-dsd zeigt Abbildung 6.10. Im ersten Fall hat die Variable die gesamte Klasse der Datumswerte als Grundmenge, im zweiten Fall kann die Variable mit der Instanz eines Telefonmodells von Siemens oder Nokia gefüllt werden.

6.3.1. Bindungszustand

Der *Bindungszustand* einer Variablen gibt an, ob und wie die Variable mit einem konkreten Wert gefüllt ist. Generell kann eine Variable nur mit Instanzen bzw. Literalen aus ihrer Grundmenge gefüllt werden (siehe Regel 8.84). Man unterscheidet vier Bindungszustände (siehe Abbildung 6.11):

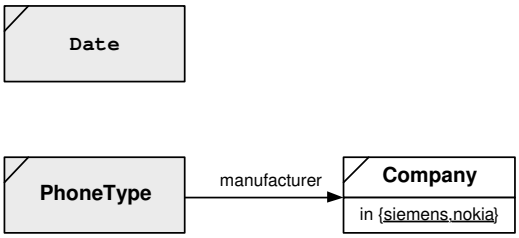


Abbildung 6.10.: Beispiele für Variablen in g-dsd.

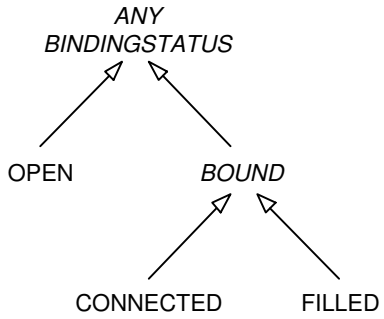


Abbildung 6.11.: Bindungszustände von Variablen.

- **OPEN.** Die Variable ist noch ungebunden, das heißt sie wurde noch mit keinem Wert gefüllt.
- **BOUND.** Bezeichnet das Gegenteil von `OPEN`. Die Variable hat bereits einen Wert zugewiesen bekommen. Dieser Zustand ist eine abstrakte Oberklasse von `FILLED` und `CONNECTED` und tritt selbst nicht auf (siehe Regeln 8.85 bis 8.87).
- **FILLED.** Eine Möglichkeit von `BOUND`. Bezeichnet eine Variable, der ein Literal bzw. eine konkrete Instanz zugewiesen wurde.
- **CONNECTED.** Eine Möglichkeit von `BOUND`. Bezeichnet eine Variable, die mit einem noch unbekannten Wert gefüllt wurde, d.h. diesen Wert übernimmt, sobald er feststeht und daher mit einer anderen Variablen verbunden ist.

Eine wichtige Eigenschaft von Variablen ist, dass sie auch als Zielmenge von Attributbedingungen verwendet werden können. Es gilt dann folgende Semantik:

- Eine ungebundene Variable wirkt als Zielmenge einer Attributbedingung wie ihre Grundmenge (siehe Regel 8.88).
- Eine gefüllte Variable wirkt wie eine einelementige Menge, die nur das eingefüllte Literal bzw. die eingefüllte Instanz als Element enthält (siehe Regel 8.89).

- Eine verbundene Variable wirkt wie die Variable, mit der sie verbunden ist (siehe Regel 8.90).

Die Mischung von aggregierenden Elementen mit selektierenden Elementen führt zu so genannten **konfigurierbaren Mengen**, einem zentralem Konzept von DE. Eine konfigurierbare Menge ist eine Menge, deren Wertebereich durch Füllen der auftretenden Variablen eingeschränkt oder auf ein Element konkretisiert werden kann. Insbesondere zur Beschreibung von Diensten spielt diese Kombination von Mengen und Variablen eine große Rolle, da hierdurch die Wirkung der zwischen Dienstnehmer und Dienstgeber ausgetauschten Informationen eindeutig beschrieben ist.

6.4. Operatoren

Im Gegensatz zu anderen Individuen ist die Hauptaufgabe von Diensten eine Wirkung in der realen Welt. Ausgehend von initialen Zuständen, die zu einem bestimmten Zeitpunkt erfüllt sein müssen, stellt ein Dienst eine Reihe von neuen Zuständen her. DE-II trägt diesem Umstand Rechnung und führt speziell zur Beschreibung von Diensten *operationale Sprachelemente* ein, mit denen einerseits ein Existenztest für Zustände, andererseits das Entstehen neuer Zustände formal korrekt erfasst werden kann:

- x <<precondition>> s mit einer Instanz x und einer Menge s. Beschreibt einen Test, der überprüft, ob die Menge s ein Element enthält. Typischerweise ist s eine Menge vom Typ State, formal jedoch nicht darauf beschränkt. Mit <<precondition>> wird also ein *Existenztest auf Instanzen* ausgedrückt.
- x <<effect>> s mit einer Instanz x und einer Menge s. Beschreibt eine Wirkung, bei der ein Element neu erzeugt wird, das dann zur s gehört. Auch hier ist o gewöhnlich vom Typ State, formal jedoch nicht darauf beschränkt. Mit <<effect>> wird also eine *Erzeugung von Instanzen* ausgedrückt.

Über die Ausgangsinstanz x können mehrere Vorbedingungen und Effekte verbunden werden. Die Semantik ist dann wie folgt definiert: Wenn alle Existenztests gegeben durch <<precondition>> an x zu einem Zeitpunkt t_1 erfüllt sind, dann entstehen durch alle Effekte gegeben durch <<effect>> an x neue Instanzen zum Zeitpunkt t_2. Die formale, dienstneutrale Semantik der beiden Operatoren findet sich in Regel 8.91. Eine dienstspezifische Semantik ergibt sich, wenn die Operatoren innerhalb von Dienstbeschreibungen verwendet werden (siehe Abschnitt 7.2.2).

Repräsentiert werden die beiden Operatoren wie Attribute, die als Füllwert eine Menge besitzen. In g-dsd geschieht dies durch Pfeile, die mit precondition bzw. effect

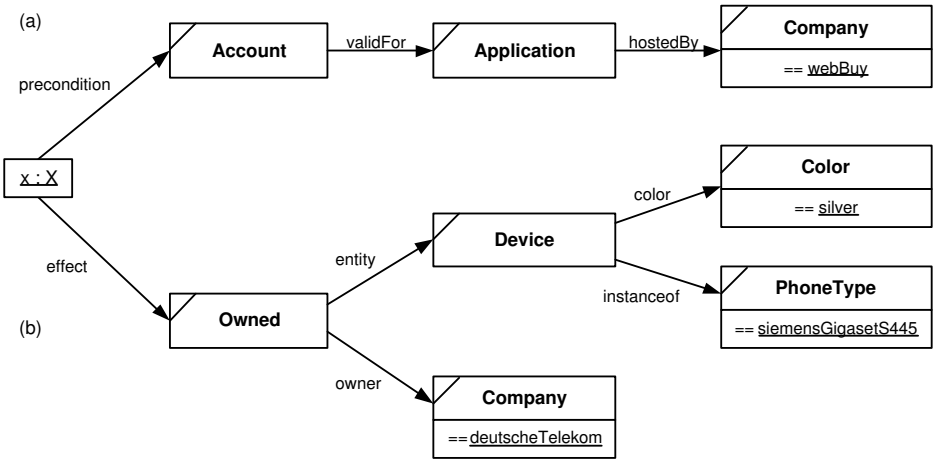


Abbildung 6.12.: Beispiele für die operationalen Elemente in g-dsd. (a) beschreibt einen Test, bei dem überprüft wird, ob ein Account bei webBuy vorliegt; (b) beschreibt eine Wirkung, bei der die Deutsche Telekom ein bestimmtes Telefon erwirbt.

beschriftet sind und von der gemeinsamen Instanz x auf die jeweiligen Operandenmengen s zeigen. Abbildung 6.12 zeigt ein Beispiel. (a) beschreibt einen Test, der überprüft, ob ein Account bei webBuy existiert, (b) beschreibt eine Wirkung, bei der die Deutsche Telekom ein bestimmtes Telefon erwirbt.

6.5. Zusammenfassung

In diesem Kapitel wurde die Sprache *DIANE Elements II* vorgestellt. Diese setzt auf DIANE Elements I auf und führt vier speziell für Dienstbeschreibungen wichtige Sprachelemente ein: aggregierende Elemente in Form von Mengen, bewertende Elemente in Form von unscharfen Mengen, selektierende Elemente in Form von Variablen und operationale Elemente in Form von Operatoren. Neben der Syntax von g-dsd wurde ihre dienstneutrale Semantik durch Verweis auf die entsprechenden formalen Regeln in Kapitel 8 definiert. Mit der Verwendung der Elemente in konkreten Dienstbeschreibungen im nächsten Kapitel gewinnen sie eine dienstspezifische Semantik.

7. Dienstbeschreibungssprache: DIANE Service Description

Zum Beschreiben von Diensten wird in dieser Arbeit die *DIANE Service Description* (DSD) vorgestellt, die auf den Sprachelementen aus DE-I und -II basiert. In diesem Kapitel wird deren Aufbau in Abschnitt 7.1 und die zugehörige Bedeutung in Abschnitt 7.2 vorgestellt.

7.1. Struktur von Dienstbeschreibungen in DSD

DSD ist eine Sprache zur Beschreibung von Diensten und besteht aus einer Reihe von Teilen:

- DSD enthält ein *klassisches Schema* für Dienste. Dieses ist in DE-I definiert und dient als grundlegende Schablone, um Dienste als Teil der Welt in Form von Instanzen zu erfassen.
- DSD bietet Möglichkeiten, um die *neuen Sprachelemente* aus DE-II in Dienstbeschreibungen integrieren zu können. Mit diesen kann unter anderem die Wirkung des Dienstes in der Welt erfasst werden. Diese Sprachelemente erhalten dann neben der bereits vorgestellten dienstneutralen Semantik eine zusätzliche dienstspezifische Semantik.
- DSD enthält eine Reihe weiterer vordefinierter Schemata, die einerseits den *Raum der Zustände* strukturieren und andererseits einen *Rahmen für Domänenontologien* liefern.

Die einzelnen Teile finden sich in einer Schichtung von Ontologien. Wie bereits in Abschnitt 4.4 vorgestellt, existieren drei solcher Stufen [78], die allesamt unter einer oberen Ontologie eingeordnet werden (siehe Abbildung 7.1):

- *Die obere Ontologie* teilt die allgemeinste Klasse Thing in grundlegende Konzepte auf. Jede andere Klasse sollte sich durch Vererbung in diese Strukturierung einordnen.

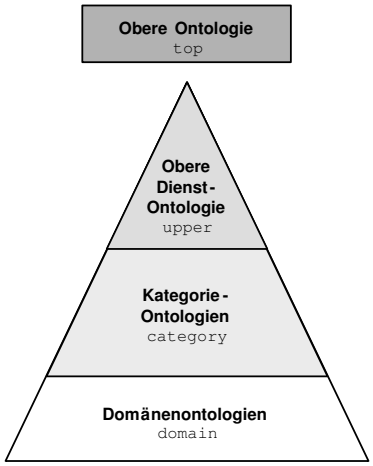


Abbildung 7.1.: Schichtung verschiedener Ontologien in DSD.

- *Die obere Dienstontologie* legt den Grundaufbau von Dienstbeschreibungen fest, indem sie ein Dienstschema mit Möglichkeiten zur Integration neuer Sprachelemente bereitstellt.
- *Kategorieontologien* erfassen und strukturieren den Raum der Zustände, welche durch Dienste verändert werden können.
- *Domänenontologien* repräsentieren die Anwendungsgebiete der realen Welt.

Die einzelnen Stufen werden in den folgenden Abschnitten im Detail vorgestellt.

7.1.1. Obere Ontologie

Die obere Ontologie `top` hat die Aufgabe, die generische Klasse Thing in grundlegende Konzepte zu unterteilen. Diese können als abstrakt angesehen werden, da sie keine direkten Instanzen besitzen. Konkrete Konzepte in Domänenontologien sollten stets als Unterklasse einer Klasse aus `top` definiert werden. `top` orientiert sich an oberen Ontologien der Literatur wie DOLCE [48], SUMO [108] oder den generischen Teilen von WordNet [114]. In der OntoClean-Methode [54] werden diese Grundlagenkonzepte auch als *Kategorien* bezeichnet. Sie zeichnen sich dadurch aus, dass sie zwar rigide sind, jedoch keine Identität tragen.

Abbildung 7.2 zeigt die Ontologie `top`. Folgende Konzepte sind enthalten:

- Entity und Value. Jedes Ding gehört zu einer der beiden Klassen. Die Unterscheidung entspricht der Aufteilung in Entitäts- und wertbestimmte Klassen. Wertbestimmte Klassen wie Price haben eine einfache Identität, Entitätsklassen wie Person oder Book haben eine komplexe Identität.

- PhysicalEntity und AbstractEntity. Jede Entität gehört zu einer der beiden Klassen. Physische Entitäten wie Person bestehen aus Materie, „werfen einen Schatten“ und haben eine Lokation, während abstrakte Entitäten wie Trip oder Document nicht-materielle, gedankliche Konzepte darstellen.

- Agent. Eine besondere Art physischer Entitäten sind Agenten. Diese zeichnen sich dadurch aus, dass sie selbstständig handeln können wie etwa Personen (Person) und Unternehmen (Company).

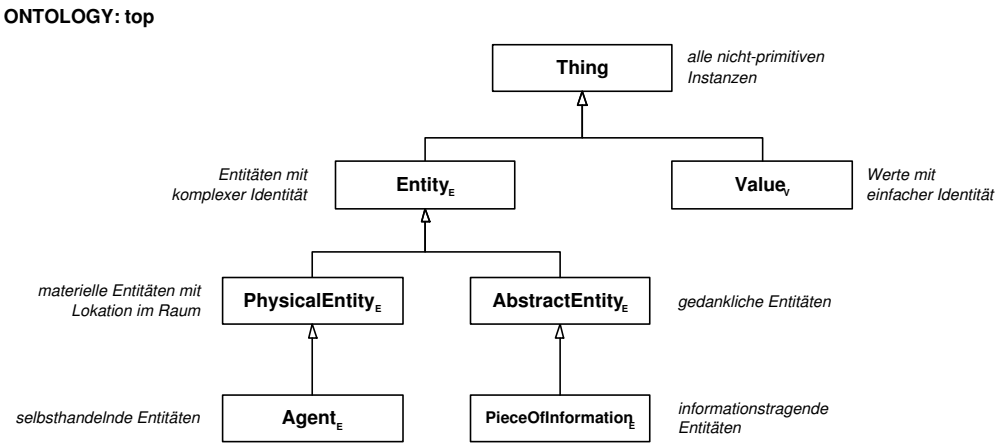


Abbildung 7.2.: Die obere Ontologie `top`.

- PieceOfInformation. Eine besondere Art abstrakter Entitäten sind Informationseinheiten wie etwa Dateien (File) oder Datenbankeinträge (DatabaseEntry), d.h. Entitäten die Informationen repräsentieren.

Die Verwendung einer oberen Ontologie bringt zwei Vorteile. Einerseits zwingt sie den Ersteller von Ontologien, sich über die Semantik ihrer Konzepte klar zu werden, wenn diese in die obere Ontologie einsortiert werden müssen. Andererseits wiederholen sich bestimmte orthogonale Attribute für eine Vielzahl von Klassen. Diese können dann ausfaktorisiert und an den entsprechenden Konzepten der oberen Ontologie definiert werden.

7.1.2. Obere Dienstontologie

Die obere Dienstontologie ist ein Schema, welches das allgemeine Grundgerüst für Dienstbeschreibungen festlegt. Abbildung 7.3 zeigt diese Ontologie `upper`. Das Schema ist sehr einfach und übernimmt im Wesentlichen die Idee aus OWL-S, die unterschiedlichen Aspekte eines Dienstes getrennt zu beschreiben. In DSD erfolgt das in zwei Teilen: Das ServiceProfile enthält eine abstrakte Beschreibung der Dienstleistungen, d.h. was der Dienst macht. Im ServiceGrounding ist festgehalten, wie die Verbindung vom abstrakten zum realen Dienst aussieht (siehe dazu Anhang A).

Abbildung 7.4 zeigt das ServiceProfile in der Ontologie `upper.profile`. Einerseits sieht es vor, nicht-funktionale Aspekte mit klassischen Elementen aus DE-I zu beschreiben. Sie sind nicht Teil dieser Arbeit und daher hier nicht vorgegeben. Eine Möglichkeit sie zu beschreiben findet sich in [115, 116]. Andererseits bietet das ServiceProfile eine gemeinsame Instanz für die operationalen Elemente precondition und effect und damit auch für die anderen neuen Sprachelemente aus DE-II. Der `*` soll deutlich machen, dass die Operatoren beliebig oft aufgeführt werden können. Die genaue

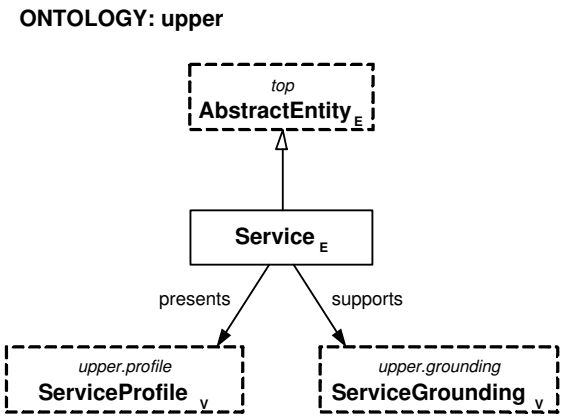


Abbildung 7.3.: Die obere Dienstontologie von DSD.

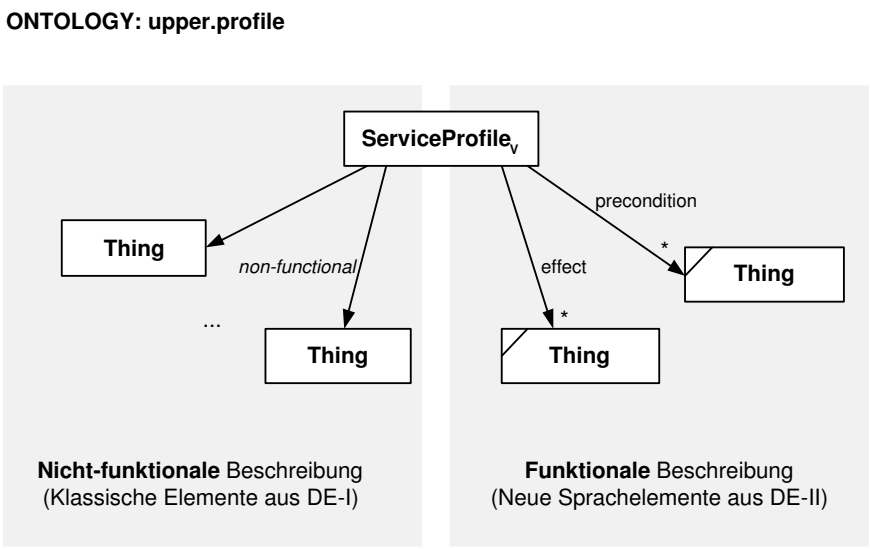


Abbildung 7.4.: ServiceProfile von DSD. Nicht-funktionale Aspekte werden mit klassischen Elementen aus DE-I beschrieben; für neue Sprachelemente bietet es eine gemeinsame Instanz, um hieran die funktionale Semantik des Dienstes zu beschreiben.

dienstspezifische Semantik der neuen Sprachelemente findet sich in Abschnitt 7.2.2. Vereinfacht gilt:

- precondition (Vorbedingung): Beschreibt Bedingung der Welt, die erfüllt sein muss, damit der Dienst erfolgreich ausgeführt werden kann. Wird der Dienst aufgerufen, obwohl eine Bedingung nicht erfüllt ist, so ist der Effekt undefiniert. Die Bedingungen gelten als erfüllt, wenn sich unmittelbar vor Dienstausführung jeweils mindestens eine Instanz in den von precondition referenzierten Mengen befindet.
- effect (Effekt): Beschreibt die Zustände der Welt nach einer erfolgreichen Durchführung des Dienstes. Die Menge als Operand von effect in einer Angebotsbeschreibung sagt aus, dass durch die erfolgreiche Dienstausführung genau ein Effekt der Menge neu entsteht, d.h. durch den Dienstgeber erwirkt wird.

Für precondition und effect ist angegeben, dass ihre Operanden Mengen von Typ Thing sind. In konkreten Beschreibungen werden hier jedoch meist Mengen von Unterklassen der Klasse State zu finden sein, da Dienste häufig den Zustand von Entitäten überprüfen bzw. verändern. Die Mengen selbst werden in der Regel konfigurierbar sein, indem Variablen in der Definition der Operanden verwendet werden. Auch unscharfe Mengen sind möglich.

Dienstprofile von angebotenen und benötigten Diensten unterscheiden sich in zwei Punkten voneinander:

- *Keine unscharfen Mengen in Angebotsbeschreibungen.* Angebotsbeschreibungen beschreiben die Familie der Effekte, die der angebotene Dienst erbringen kann. Dabei gilt, dass der Dienstanbieter keine Präferenzen unter diesen Effekten hat, zumindest keine, die er in der Dienstbeschreibung bekannt gibt. Mengen, die in Angebotsbeschreibungen eingebracht werden, sind daher immer scharfe Mengen. Dadurch sind insbesondere nur die beiden Operatoren `and` und `or` bzw. `add` und `mul` im Rahmen von Verbindungsstrategien zulässig.

- *Keine Vorbedingungen in Anfragebeschreibungen.* Da der Anfrager in der Regel nicht weiß, welche Vorbedingung ein konkretes Dienstangebot benötigen wird, enthält eine Anfragebeschreibung typischerweise keine Beschreibung der vom Dienstnehmer erfüllbaren Vorbedingungen. Die Prüfung oder Erfüllung der Vorbedingungen einer Angebotsbeschreibung finden erst dann statt, wenn für die Effekte bereits ein positives Vergleichsergebnis festgestellt wurde.

Ein weiterer Teil einer Dienstbeschreibung in DSD stellt das *Grounding* dar. Das Grounding beschreibt den Zusammenhang zwischen der abstrakten, formellen Beschreibung des Dienstes im Dienstprofil und der konkreten, ausführbaren Funktion des Effektgenerators, welche als Dienst beschrieben ist. Im Gegensatz zu anderen Sprachen existiert in DSD sowohl für Angebots- als auch für Anfragebeschreibungen ein Grounding:

- Das Grounding eines angebotenen Dienstes beschreibt den Zusammenhang zu der *real existierenden* Funktion des Effektgenerators, die als Dienst bereitgestellt werden soll. Der Effektgenerator kann hierbei beispielsweise eine Javaklasse mit ihren Methoden, ein Web Service oder ein Wrapper zu einer Webseite sein.

- Das Grounding einer Dienstanfrage beschreibt den Zusammenhang zu einer *gewünschten* Funktion, die beispielsweise innerhalb einer Applikation verwendet und über einen dynamischen Dienstaufruf realisiert werden soll.

Das Grounding hat in beiden Fällen zwei Aufgaben. Zum einen beschreibt es den konkreten Informationsfluss, d.h. es legt fest, wie der durch die IN- und OUT-Variablen spezifizierte Nachrichtenfluss im zugrunde liegenden Effektgenerator ausgeführt werden muss. Konkret wird also angegeben, wie die Werte der IN-Variablen als Parameter an den Effektgenerator übergeben werden und wie die Daten der OUT-Variablen als Rückgabewerte des Effektgenerator abgelesen werden. Zum zweiten beschreibt das Grounding den Zusammenhang zum Zustandsübergang, d.h. es gibt an, wie die Funktion konkret verwendet muss, damit die im Profil angegebenen Effekte tatsächlich entstehen. Die Details zum Grounding finden sich in Anhang A.

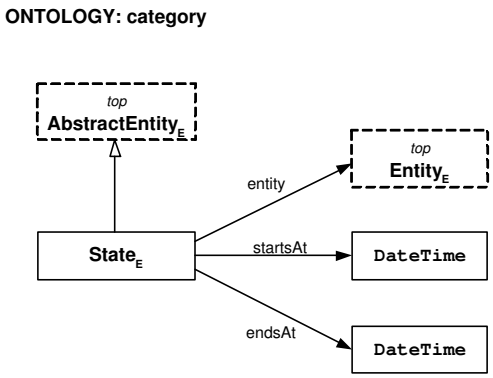


Abbildung 7.5.: Die Ontologie `category`, in der die Oberklasse State definiert ist.

7.1.3. Kategorieontologien

Die zweite Schicht von Ontologien zur Beschreibung von Diensten stellen Kategorieontologien dar. Diese beschreiben und strukturieren den Raum der möglichen Zustände. Ontologien aus dieser Schicht beginnen mit dem Präfix `category`.

Zustände stellen in DE verdinglichte extrinsische Eigenschaften dar. Wie Abbildung 7.5 zeigt, werden sie durch die Entitätsklasse State in der Ontologie `category` repräsentiert. State hat ein Attribut entity, das auf die beschriebene Entität zeigt, sowie die Attribute startsAt und endsAt, die angeben, in welchem Zeitraum der Zustand gilt. Durch Anlegen konkreter Instanzen von State und deren Unterklassen werden orthogonale Attribute implizit gefüllt, wie in Regel 8.19 festgehalten.

Abbildung 7.6 zeigt ein Beispiel. Die Klasse Device hat die extrinsische Eigenschaft, einen Besitzer zu haben, was durch das orthogonale Attribut isOwned ausgedrückt wird. Zieltyp von isOwned ist die Zustandsklasse Owned. Der Zustand Owned stellt somit die verdinglichte Eigenschaft dar, welcher von der allgemeinen Klasse State erbt. Zusätzlich zu den geerbten Attributen wird in Owned das Attribut owner definiert, welches den Besitzer als Agent enthält.

In Kategorieontologien sollen jedoch keine domänenspezifischen Zustände aufgeführt

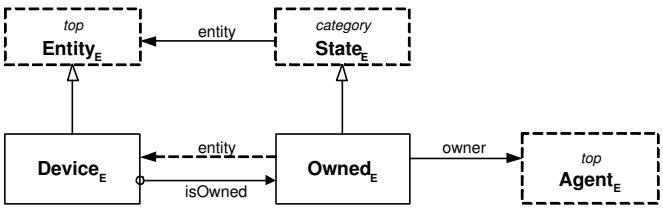


Abbildung 7.6.: Owned als Beispiel für eine Zustandsklasse, die eine extrinsische Eigenschaft verdinglicht.

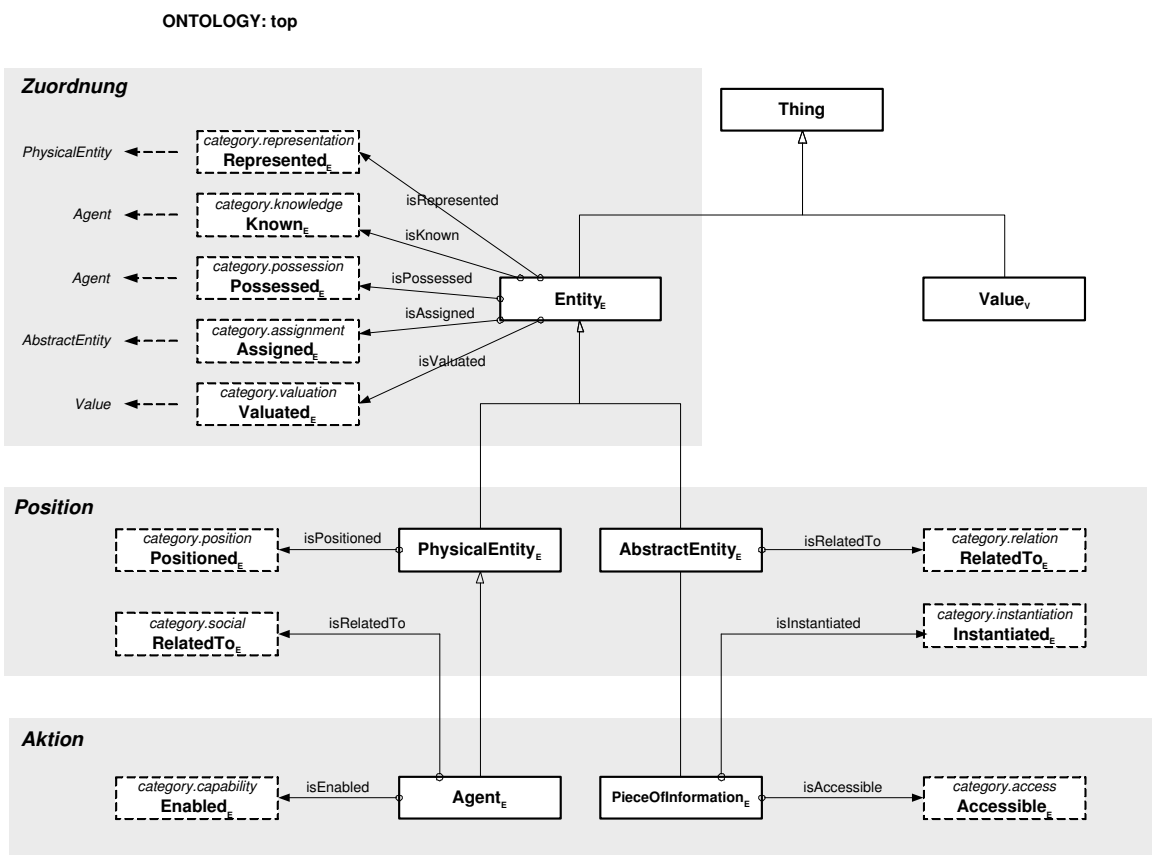


Abbildung 7.7.: Erfassung generischer orthogonaler Attribute und Zustände in der Ontologie `top`

werden, sondern Ziel ist es, möglichst allgemeine Zustände abzudecken, die für eine große Zahl von Konzepten möglich sind. Die Zustände werden daher bereits in der oberen Ontologie `top` durch orthogonale Attribute definiert. Dabei ist festzuhalten, dass nur Entitäten orthogonale Attribute besitzen können; bei wertbestimmten Klassen (Value) sind per Definition alle Attribute definierend.

Die wichtigsten, am häufigsten verwendeten orthogonalen Attribute können in drei Stufen festgemacht werden, was auch in Abbildung 7.7 verdeutlicht wird:

1. Die generischsten orthogonalen Attribute finden sich an der allgemeinen Oberklasse Entity. Sie drücken nur eine generische **Zuordnung** zu anderen Instanzen aus.

2. Für Unterklassen von Entity kann je nach Typ ein orthogonales Attribut angegeben werden, welches die aktuelle **Position** einer Instanz ausdrückt. Je nach Konzept handelt es sich hierbei um eine physische oder nicht-physische Position.

3. Die speziellen Unterklassen Agent und PieceOfInformation besitzen weitere charakteristische Attribute, die etwas über die **Aktion** einer Instanz aussagen: Agenten können handeln, Informationen können verarbeitet werden.

Im Einzelnen sind das folgende orthogonale Attribute:

Entity umfasst sehr viele verschiedene Konzepte, wodurch die orthogonalen Attribute nur sehr generisch sein können. Für eine Instanz i vom Typ Entity ist beispielsweise die allgemeine *Zuordnung* einer anderen Instanz j erfassbar. Die einzelnen Arten von Zuordnungen entstehen durch die unterschiedlichen Möglichkeiten von Typen für j:

- j kann vom allgemeinen Typ PhysicalEntity sein, d.h. der Entität wird eine konkrete Entität aus Materie zugeordnet. Diese kann die Entität i im weitesten Sinne repräsentieren oder sie vertreten. Beispielsweise wird das abstrakte Recht, ein Kinovorstellung besuchen zu dürfen, durch eine physikalische Kinokarte dargestellt. Ausgedrückt wird dies durch das orthogonale Attribute isRepresented, welches auf einen Zustand Represented aus `category.representation` zeigt.

- j kann vom speziellen Typ Agent sein, d.h. der Entität wird ein selbstständig handelnder Agent zugeordnet. Dieser kann etwas über die Instanz i wissen oder sie in irgendeiner Form besitzen. Ersteres wird durch das orthogonale Attribut isKnown ausgedrückt, welches auf den Zustand Known zeigt. Zweiteres wird durch das orthogonale Attribut isPossessed ausgedrückt, welches auf einen Zustand Possessed aus `domain.possession` zeigt.

- j kann vom allgemeinen Typ AbstractEntity sein, d.h. der Entität wird ein abstraktes gedankliches Konzept zugeordnet, wie etwa eine Belastung oder ein Gebot. Ausgedrückt wird das durch das Attribut isAssigned. Hierzu gehört auch der Fall, dass j ein PieceOfInformation, d.h. zum Beispiel eine Datei oder ein Datenbankeintrag ist.

- j kann vom Typ Value sein, d.h. der Entität wird ein Wert zugewiesen, etwa ein Preis, eine Beschriftung oder eine Bewertung. Ausgedrückt wird dies durch das orthogonale Attribut isValuated.

Ist die genaue Unterklasse von Entity bekannt, kann zusätzlich eine *Position* angegeben werden:

- Eine PhysicalEntity hat definitionsgemäß eine Lokation im Raum, die entweder absolut oder relativ zu anderen physischen Entitäten angegeben werden kann. Ausgedrückt wird das durch das orthogonale Attribut isPositioned.

- Ein Agent hat zudem eine soziale Position in der Gesellschaft. Ausgedrückt wird das durch das orthogonale Attribut isRelatedTo.

- Eine AbstractEntity hat eine semantische Position bezüglich anderer abstrakter Entitäten, steht etwa in einer Enthaltenseinsbeziehung. Dies wird allgemein durch das orthogonale Attribut isRelatedTo ausgedrückt.

- Ein PieceOfInformation kann zudem eine (oder mehrere) Ausprägungen an verschiedenen Positionen haben, etwa durch Transformation, durch Versand, durch Visualisierung usw. Dies wird durch das orthogonale Attribut isInstantiated ausgedrückt.

Für die speziellsten top-Klassen Agent und PieceOfInformation stehen charakteristische orthogonale Attribute zur Verfügung, die etwas über die *Aktion* einer Instanz aussagen. Agenten zeichnen sich durch ihre Handlungsfähigkeit aus. Über das orthogonale Attribut isEnabled können daher ihre (zeitlich variablen) Fähigkeiten ausgedrückt werden. Informationseinheiten zeichnen sich durch ihre Eigenschaft aus, von Agenten verarbeitet und interpretiert werden zu können. Welche Agenten auf sie zugreifen können, wird durch (isAccessible) ausgedrückt.

Wichtigste Aufgabe der Kategorieontologien ist es, die vorgestellten Zustände zu strukturieren, d.h. ihre Unterklassen zu konkretisieren. Die Beschreibung und Strukturierung umfasst (1) die Hierarchisierung der Zustände, d.h. die Einordnung in Vererbungsbeziehungen, um so ähnliche Dienste durch eine gemeinsame Oberklasse kenntlich zu machen, (2) die Erweiterung konkreter Zustandsklassen um Attribute, mit denen der Kontext des Zustandes erfasst werden kann und (3) die Modularisierung des Zustandsraums in einzelne Ontologien.

Kategorieontologien streben zwar nach Vollständigkeit, in der Praxis müssen jedoch von Zeit zu Zeit neue Zustände eingefügt werden. Dabei gilt generell:

- Neue domänenspezifische Zustände werden in der entsprechenden Domänenontologie definiert und sollten einen generischen Zustand aus einer Kategorieontologie als Oberklasse haben.

- Neue generische Zustände sollten (wenn möglich) nach Absprache mit der Anwendergruppe in eine der existierenden Kategorieontologien eingefügt werden und dort von einem existierenden Zustand erben.

- Nur in Ausnahmefällen sollten neue Kategorieontologien erstellt werden, um einen neuen Zustand erfassen zu können.

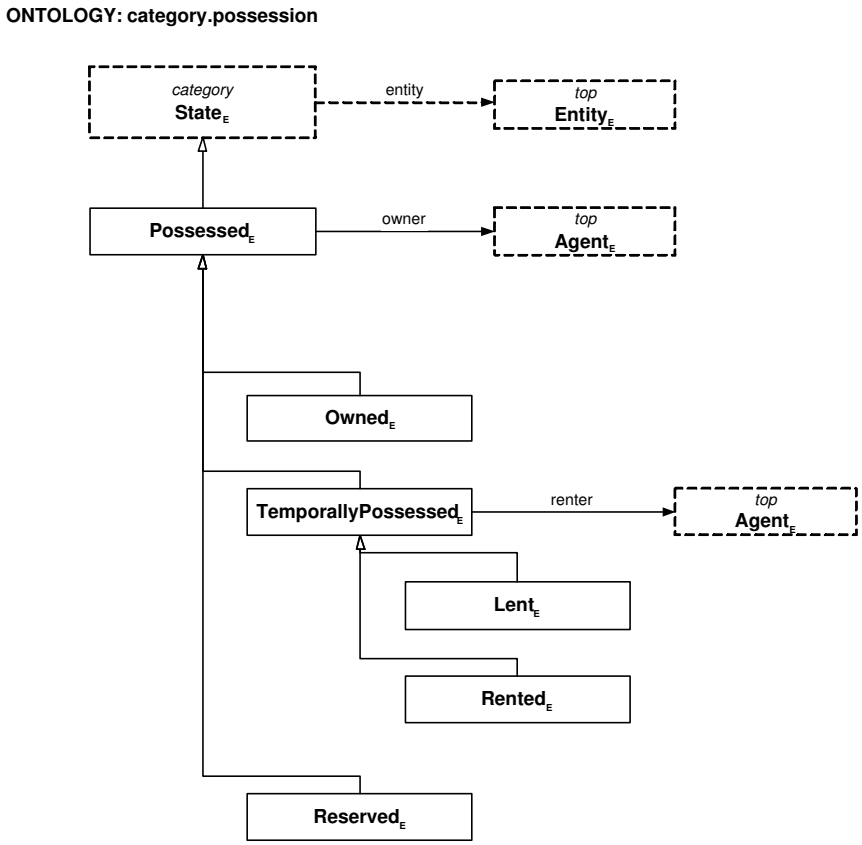


Abbildung 7.8.: Die Ontologie `category.possession` zur Darstellung von Zuständen, die etwas über den Besitzzustand einer Entität aussagen.

Im Folgenden wird die Kategorieontologie `category.possession` beispielhaft vorgestellt. Sie verfeinert zwar die vorgeschlagene Zustandsklasse Possessed, alle ihre Unterklassen sind jedoch nach wie vor domänenunabhängig. Spezielle, domänenspezifische Zustände sollten in entsprechenden Domänenontologien aufgenommen werden und einen allgemeinen Zustand aus einer Kategorieontologie als Oberklasse haben. Weitere Kategorieontologien finden sich in Anhang D.3.

Die Ontologie `category.possession` (siehe Abbildung 7.8) drückt den Besitzzustand einer Entität aus. Oberklasse ist Possessed, die aussagt, dass einer Entität entity ein Agent (also z.B. eine Person oder ein Unternehmen) zugewiesen ist (der owner), der ein Nutzungsrecht besitzt. Unterschieden werden folgende Abstufungen:

- Owned als echtes Eigentumsverhältnis. Dem Besitzer gehört die Entität. Er darf dieses Recht auch an Dritte weitergeben.
- Lent als Leihverhältnis. Der „Besitzer“ darf die Entität zeitweise und unentgeltlich nutzen. Dieses Recht darf er nicht an Dritte weitergeben.

- Rented als Leihverhältnis. Der „Besitzer“ darf die Entität gegen ein Entgelt zeitweise nutzen. Dieses Recht darf er nicht an Dritte weitergeben. Lent und Rented werden zu TemporallyOwned zusammengefasst. Dabei drückt das Attribut renter den Verleiher aus.

- Reserved als geplantes Nutzungsrecht. Dem „Besitzer“ wird zugesichert, die Entität im angegebenen Zeitraum nutzen zu dürfen.

Weitere Besitzzustände sind denkbar und können an entsprechender Stelle in die Ontologie integriert werden.

7.1.4. Domänenontologien

Die dritte Schicht der Ontologien stellen Domänenontologien dar. Sie konzeptualisieren ein bestimmtes Anwendungsgebiet wie Bücher, Orte, Dateien usw. Im Gegensatz zu den wenigen und eher kleinen Ontologien auf den oberen Schichten existieren viele und auch größere Domänenontologien. Sie sind nicht Teil von DSD, sondern werden von Experten der Anwendergruppe eingebracht und liegen daher im Allgemeinen verteilt vor. In der Regel enthalten Domänenontologien sowohl Klassen, die das Anwendungsgebiet inhaltlich strukturieren, als auch Instanzen, welche für konkrete, reale Individuen aus diesem Bereich stehen. Zusätzlich können domänenspezifische Vergleichsfunktionen eingebracht werden, falls diese von allgemeinem Interesse sind. Domänenontologien beginnen mit dem Präfix `domain`.

Domänenontologien für DSD sind in DE-I spezifiziert und haben die Eigenschaften 4.1 bis 4.4 aus Kapitel 4. Insbesondere sind extrinsische Eigenschaften über Zustände verdinglicht, auf die mittels orthogonaler Attribute verwiesen wird. Die Zustände können direkt aus einer der Kategorieontologien stammen oder in der Domänenontologie als Unterklasse einer existierenden Zustandsklasse definiert werden.

Neben der oberen Ontologie, die grundlegende Oberklassen für alle Konzepte bietet, treten bei Attributen in Domänenontologien häufig auch wiederkehrende Zieltypen wie Maßangaben, Preise, Personen, Dateien, Lokationen etc. auf. Diese werden in DSD in so genannten *unteren Domänenontologien* bereitgestellt und brauchen nicht selbst entwickelt zu werden. Wichtige untere Domänenontologien finden sich in Anhang D. Insgesamt werden Domänenontologien in DSD also mithilfe von Konzepten aus unteren Domänenontologien definiert und in die obere Ontologie eingeordnet (vgl. [103]).

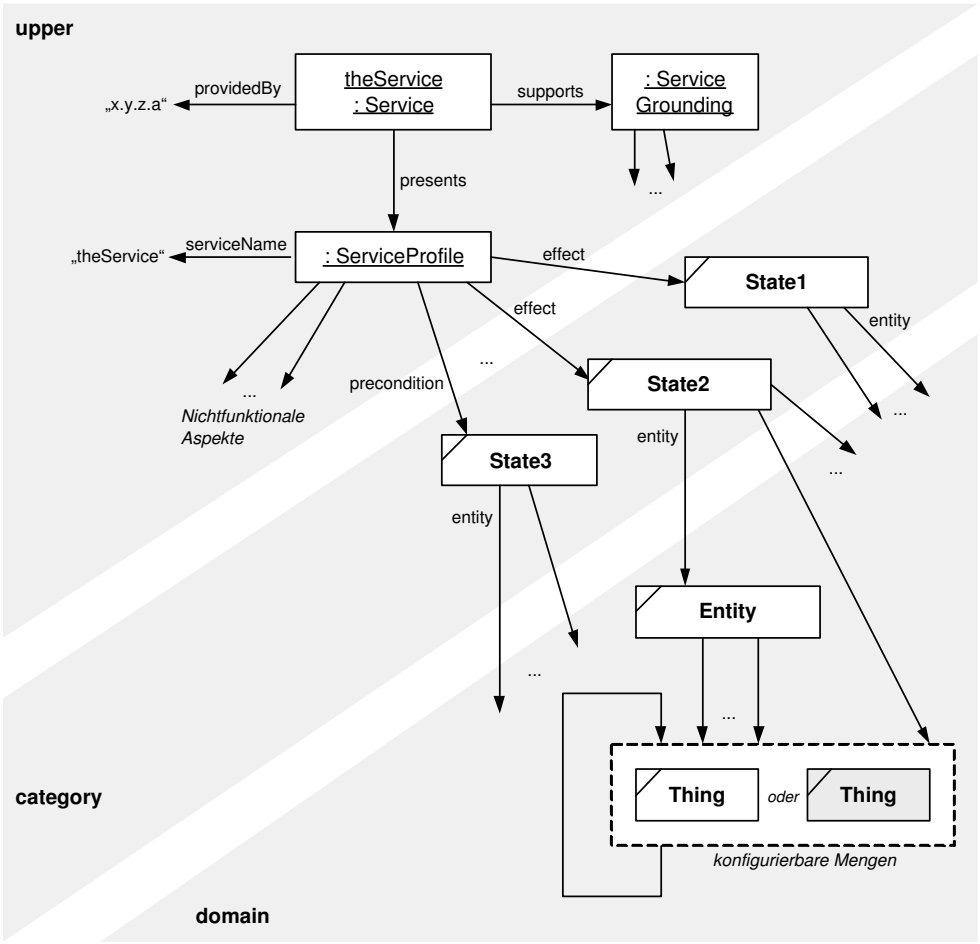


Abbildung 7.9.: Prinzipieller Aufbau von Dienstbeschreibungen in DSD.

7.1.5. Prinzipieller Aufbau von Dienstbeschreibungen

Abbildung 7.9 zeigt den resultierenden prinzipiellen Aufbau einer Dienstbeschreibung in DSD. Beginnend mit DE-I-Instanzen aus der oberen Dienstontologie `upper` wird der Kopf der Beschreibung aufgebaut. Er enthält Informationen zum Dienstgeber, zum technischen Zugangsweg im ServiceGrounding sowie zum Inhalt des Dienstes im ServiceProfile. Es beschreibt neben nichtfunktionalen Aspekten die Funktionalität des Dienstes über die DE-II-Operatoren precondition und effect. Diese zeigen auf Mengen von Zuständen aus Kategorieontologien `category.*`. Jeder dieser Zustände hat neben anderen Bedingungen zumindest eine Attributbedingung zur beschriebenen Entität. Die Entität selbst stammt aus einer Domänenontologie `domain.*` und ist eine konfigurierbare Menge nach DE-II, d.h. über verschachtelte Attributbedingungen definiert, die auch Variablen enthalten kann. Die Blätter werden dabei meist von Mengen primitiver Typen gebildet. In Anfragebeschreibungen können zudem unscharfe Mengen

auftreten, dafür entfällt in diesen die Notation von Vorbedingungen. Dienstbeschreibungen werden explizit als solche ausgezeichnet. Insgesamt sind die Graphen der Dienstbeschreibungen Bäume mit einer Instanz vom Typ Service als Wurzel.

7.2. Bedeutung von Dienstbeschreibungen in DSD

In Dienstbeschreibungen, die nach Abschnitt 7.1 aufgebaut sind, erhalten die neuen Sprachelemente aus DE-II zusätzlich zu der bereits bestehenden dienstneutralen Semantik eine erweiterte, *dienstspezifische Semantik*. Die Semantik ist dienstspezifisch, da sie direkt auf die Teilnehmer und Zeitpunkte innerhalb der Dienstbereitstellung und -nutzung Bezug nimmt. Im folgenden Abschnitt 7.2.1 wird dieser Ablauf von Dienstbereitstellung und Dienstnutzung vorgestellt.

7.2.1. Ablauf von Dienstbereitstellung und Dienstnutzung

Der (menschliche) Anbieter eines Dienstes (der *Dienstgeber*) betreut den Nachrichtenfluss zwischen der Schnittstelle des Dienstes und dem eigentlichen Effektgenerator in der Regel nicht selbst, sondern überlässt dies einem rechnergestützten *Angebotsagenten*. Hierfür enthält die Dienstangebotsbeschreibung im ServiceGrounding Informationen, die dem Angebotsagenten erläutern, wie dieser den Effektgenerator technisch korrekt anzusprechen hat.

Auch der *Dienstnehmer* betreut den Ablauf der Dienstnutzung nicht selbst. Er übergibt die Aufgabe in der Regel einem *Anfrageagenten*, der eine Reihe von Aktionen für ihn durchführt: Er sucht nach einem geeigneten angebotenen Dienst, konfiguriert diesen, indem die zu sendenden Nachrichten festgelegt werden, ruft ihn über dessen Angebotsagenten aus, empfängt vom Dienst ausgegebene Nachrichten und generiert daraus die vom Dienstnehmer benötigten Informationen. Im Falle konfigurierbarer Dienstanfragebeschreibungen wirkt der Anfrageagent für den Dienstnehmer somit wie ein lokal verfügbarer, virtueller Dienst.

Im Folgenden wird der Ablauf von Dienstbereitstellung und Dienstnutzung mittels DSD im Detail vorgestellt. Eine Übersicht darüber gibt Abbildung 7.10. Hierin kennzeichnen ausgefüllte Pfeile synchrone Aufrufe, bei denen Rückgabewerte erwartet werden, während gestrichelte Pfeile asynchrone Aufrufe ohne Datenrückgabe darstellen.

Dienstbereitstellung

Zunächst *beauftragt* der Dienstgeber seinen Angebotsagenten mit der Bereitstellung eines Dienstes (Schritt 1). Dazu übergibt er diesem die Dienstangebotsbeschreibung

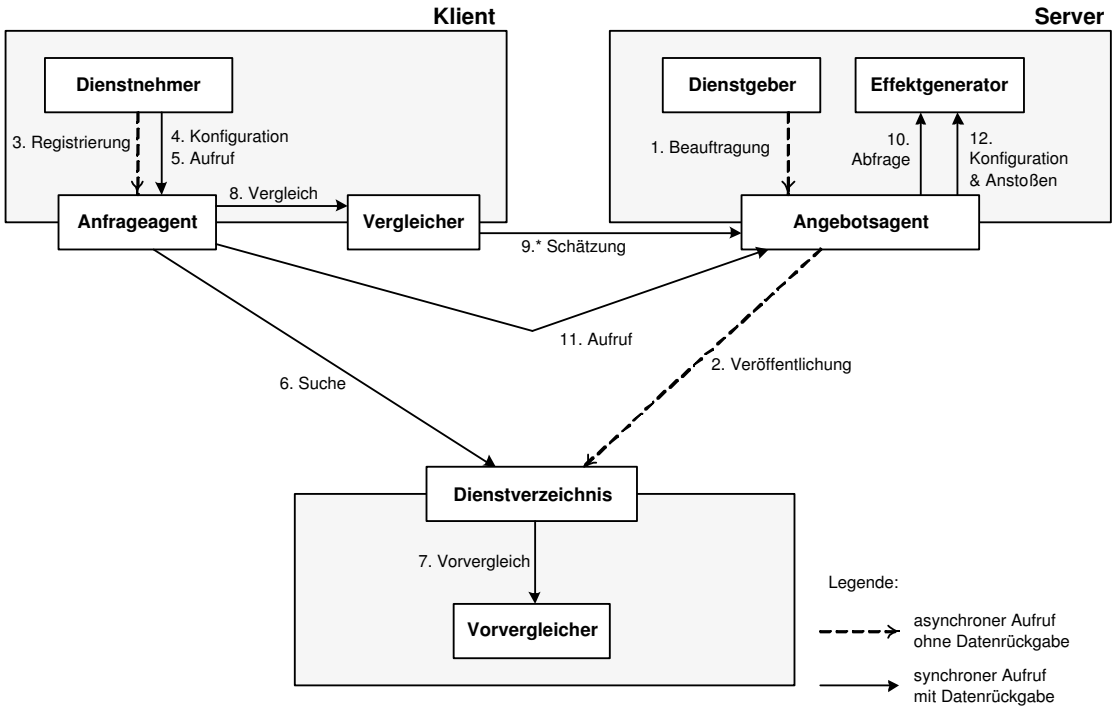


Abbildung 7.10.: Ablauf der Dienstbereitstellung und -nutzung mittels DSD.

seines angebotenen Dienstes. Der Angebotsagent unternimmt daraufhin zwei Aktionen:

- Er *veröffentlicht* die Dienstangebotsbeschreibung in einem öffentlich zugänglichen Dienstverzeichnis[1] (Schritt 2).
- Er wartet auf eingehende Nachrichten, die den angebotenen Dienst konfigurieren und anstoßen würden.

Der gesamte Prozess von der Beauftragung des Angebotsagenten bis zum Warten auf eingehende Nachrichten wird als *Dienstbereitstellung* bezeichnet.

Dienstnutzung

Typischerweise agiert ein menschlicher Benutzer nicht selbst als Dienstnehmer, sondern verwendet eine graphische Anwendung, die zur Bereitstellung ihrer Funktionalität auf externe Dienste zurückgreift. Aus diesem Grund stellt die Anwendung den

[1] Dieses kann zentralisiert oder verteilt sein. Eine genauere Betrachtung hierzu findet sich in Abschnitt 11.3.2.

Dienstgeber dar. Bei ihrem Start *registriert* sie zunächst konfigurierbare Dienstanfragebeschreibungen bei ihrem Anfrageagenten (Schritt 3). Der Anfrageagent wartet dann auf einen konkreten Aufruf einer der registrierten Anfragen durch den Dienstnehmer. Für den Dienstnehmer erscheint der Anfrageagent wie ein virtueller Dienstgeber.

Für eine konkrete Verwendung wird die Dienstanfragebeschreibung zunächst *konfiguriert* (Schritt 4) und dann aufgerufen (Schritt 5). Der Anfrageagent initiiert daraufhin eine *Suche*, indem er die konfigurierte Anfragebeschreibung an das Dienstverzeichnis schickt (Schritt 6).

Das Dienstverzeichnis hat Zugriff auf einen Vorvergleicher. Diesen beauftragt es, die Anfragebeschreibung mit allen hinterlegten Angebotsbeschreibungen *vorzuvergleichen* (Schritt 7). Der Vergleicher liefert daraufhin eine Reihe von möglicherweise passenden Dienstangebotsbeschreibungen. Ein Dienst passt dabei möglicherweise, wenn der Vorvergleicher nicht ausschließen kann, dass er passend sein könnte.

Der Anfrageagent sammelt alle eintreffenden Dienstvorschläge und leitet sie in Schritt 8 an den Vergleicher. Die endgültige Entscheidung, welcher Dienst ausgewählt, wie konfiguriert und ausgeführt wird, wird dann von diesem lokalen Vergleicher ausgeführt. Diesem stehen zum einen persönliche Daten des Dienstnehmers zur Verfügung, die nicht in die Anfrage integriert werden sollen, zum anderen kann sich der Vergleicher mit Schätzaufrufen in Schritt 9 direkt an potenzielle Dienste wenden. Der Angebotsagent fragt dann den zugehörigen Effektgenerator in Schritt 10 ab und liefert aktuelle und weiter gehende Informationen über den zu erwartenden Effekt. Der Vergleicher hat die Möglichkeit, mehrere Schätzanfragen zu stellen, was durch den Stern an Schritt 9 repräsentiert wird. Hieraus berechnet der Vergleicher geeignete Dienstangebote in Form von Vergleichsergebnissen. Diese bestehen aus (a) der Adresse des angebotenen Dienstes, (b) der nötigen Konfiguration, (c) den bereits aus der Angebotsbeschreibung bekannten Rückgabewerte für den Dienstnehmer sowie (d) dem resultierenden Vergleichswert > 0, der den Grad der Eignung zur Anfrage ausdrückt. Für einen angebotenen Dienst kann es daher mehrere Vergleichsergebnisse mit unterschiedlicher Konfiguration geben. Das Vergleichsergebnis mit dem höchsten Vergleichswert liefert der Vergleicher an den Anfrageagenten zurück.[2]

Der Anfrageagent sendet entsprechend der angegebenen Konfiguration Nachrichten an den betreuenden Angebotsagenten des bestgeeigneten Dienstangebots. Hierdurch wird der Dienst *aufgerufen* (Schritt 11). Der Angebotsagent *konfiguriert* den zugrunde liegenden Effektgenerator gemäß der erhaltenen Nachrichten und *stößt* seine

[2]Eignet sich keiner der angebotenen Dienste direkt, um die Anfrage zu erfüllen, so kann der Anfrageagent einen Planer zu Rate ziehen. Dieser zerlegt (evtl. unter Zuhilfenahme des Dienstverzeichnisses) den Effektwunsch des Dienstnehmers in mehrere Teileffekte, für welche entsprechende Dienste gefunden und entsprechend kombiniert ausgeführt werden können. Die *Dienstkombination* ist jedoch nicht Teil dieser Arbeit und wird daher nur kurz in Abschnitt 11.1 aufgegriffen.

Ausführung *an* (Schritt 12). Die dabei entstehende Informationen kapselt er in Nachrichten und sendet sie zurück an den Anfrageagenten (Rückgabe von Schritt 11). Dieser generiert daraus sowie aus dem Vergleichsergebnis die vom Dienstnehmer verlangte Information und liefert sie an ihn zurück (Rückgabe von Schritt 5).

Der gesamte Prozess vom Aufruf durch den Dienstnehmer bis zum Erhalt des Ergebnisses wird als *Dienstnutzung* bezeichnet.

Anhand des Ablaufs lassen sich die folgenden dienstspezifischen Konzepte festlegen, mit deren Hilfe die erweiterte Semantik von DE-II-Sprachelementen in Dienstbeschreibungen definiert werden kann:

- Als *Klient* fasst man alle Komponenten auf Seiten des Dienstnehmers zusammen, d.h. den Dienstnehmer selbst, seinen Anfrageagenten und den Vergleicher.
- Als *Server* fasst man alle Komponenten auf Seiten des Dienstgebers zusammen, d.h. den Dienstgeber selbst, seinen Angebotsagenten und den Effektgenerator.
- Der Zeitpunkt t_{ra} stellt den Moment des Aufrufs einer Dienstanfrage durch den Dienstnehmer dar, also den Zeitpunkt zu Beginn von Schritt 5.
- Der Zeitpunkt t_{re} stellt den Moment dar, in dem die Ergebnisse der Dienstanfrage an den Dienstnehmer geliefert werden, also der Zeitpunkt am Ende von Schritt 5.
- Der Zeitpunkt t_{ea_k} stellt den Moment dar, in dem die k-te Schätzanfrage abgesendet wird, also ein Zeitpunkt zu Beginn von Schritt 9.
- Der Zeitpunkt t_{ee_k} stellt den Moment dar, in dem die Ergebnisse der k-ten Schätzanfrage geliefert werden, also ein Zeitpunkt am Ende von Schritt 9.
- Der Zeitpunkt t_{xa} stellt den Moment dar, in dem die Ausführung des Dienstes verlangt wird, also der Zeitpunkt zu Beginn von Schritt 11.
- Der Zeitpunkt t_{xe} stellt den Moment dar, in dem die Ausführung des Dienstes beendet ist und Rückgabewerte geliefert werden, also der Zeitpunkt am Ende von Schritt 11.

Es gilt daher: $\forall k \in \mathbb{N} : t_{ra} < t_{ea_k} < t_{ee_k} < t_{xa} < t_{xe} < t_{re}$

7.2.2. Erweiterte Semantik der Sprachelemente aus DE-II innerhalb von Dienstbeschreibungen

Sprachelemente aus DE-II erwerben eine erweiterte, dienstspezifische Semantik, wenn sie innerhalb von Dienstbeschreibungen eingesetzt sind. Hierzu werden Instanzen, die als Dienstbeschreibungen fungieren sollen, explizit als Angebots- oder Anfragebeschreibung ausgezeichnet (siehe Regel 8.92 bis 8.94). In g-dsd geschieht dies durch Angabe des Markierers `{offer}` bzw. `{request}` in der linken oberen Ecke des Zeichenblattes.

Mengen

Elemente einer Menge zeichnen sich dadurch aus, dass sie alle Bedingungen der Menge wie im letzten Abschnitt angegeben erfüllen. Liegen von einer Instanz jedoch mehrere unterschiedliche Kopien vor, so ist nicht definiert, auf welcher Kopie einer Instanz die Bedingungen überprüft werden müssen. In der Tat können verschiedene Kopien derselben Instanz teilweise zur Menge gehören und teilweise nicht. Bei der Definition einer Menge ist daher die Angabe der *Grundgesamtheit* nötig, d.h. die Kollektion von Instanzkopien, für welche die Mengenzugehörigkeit überprüft wird.

Für Mengen, die im Rahmen von Dienstangebotsbeschreibungen eingesetzt werden, unterscheidet man vier Grundgesamtheiten. Sie werden durch folgende Bezeichner markiert:

- `|pubpool|`. Grundgesamtheit der Menge ist der öffentliche Instanzenpool, d.h. die darin abgelegten zentralen Kopien. Hierdurch ist gewährleistet, dass Dienstnehmer und Dienstgeber über dasselbe Wissen verfügen.
- `|offpool|`. Grundgesamtheit der Menge ist der private Instanzenpool des Dienstgebers, d.h. seine darin abgelegten lokalen Kopien. Alle zentralen Kopien von Instanzen im öffentlichen Instanzenpool, die nicht explizit im lokalen Pool des Dienstgebers angelegt wurden, gelten als unverändert übernommen.
- `|reqpool|`. Grundgesamtheit der Menge ist der private Instanzenpool des Dienstnehmers, d.h. seine darin abgelegten lokalen Kopien. Auch hier gilt, dass alle zentralen Instanzen übernommen werden, falls sie nicht explizit im privaten Pool angelegt sind.

Bei fehlendem Markierer gilt der Standardwert `|offpool|`.

Bei Mengen in Dienstanfragebeschreibungen ist eine Unterscheidung nicht sinnvoll, da nur eine Bewertung der angebotenen Dienste im Mittelpunkt steht.

Variablen

Variablen, die in Dienstbeschreibungen auftreten, besitzen eine erweiterte Semantik, die festlegt, wann und von wem diese im Rahmen der Dienstnutzung gebunden werden müssen. Hierzu werden diese in *Kategorien* eingeteilt und entsprechend annotiert. Generell unterscheidet man zwei Arten: IN-Variablen und OUT-Variablen:

- IN-*Variable.* Diese muss im Verlauf der Dienstnutzung vom *Klienten* ausgefüllt werden. IN-Variablen in einer Anfrage (so genannte ReqIN-Variablen) sind dann sinnvoll, wenn der Dienstnehmer die Anfrage häufiger in unterschiedlicher Parametrisierung ausführen will. Sie werden vor dem Absenden der Anfrage ausgefüllt. IN-Variablen im Angebot (so genannte OffIN-Variablen) müssen vom Klienten ausgefüllt werden, bevor der Dienst geschätzt oder zur Ausführung aufgerufen werden kann, um ihn zu konfigurieren.
- OUT-*Variable.* Diese muss im Verlauf der Dienstnutzung vom *Server* ausgefüllt werden. OUT-Variablen in der Anfrage (so genannte ReqOUT-Variablen) stehen für Informationen zur Dienstausführung, an denen der Dienstnehmer interessiert ist. Diese werden während der Schätzphase oder nach der Dienstausführung eingetragen. OUT-Variablen im Angebot (so genannte OffOUT-Variablen) stehen für Informationen, die der Server auf Nachfrage bzw. nach Dienstausführung bekannt gibt.

Für Variablen in der Angebotsbeschreibung kann zudem konkretisiert werden, wann sie ausgefüllt werden müssen. Man unterscheidet hier Variablen der *Schätzphase* und Variablen der *Ausführungsphase*.

- Variablen, die in der Schätzphase auszufüllen sind, werden mit dem Index e (für estimate, engl. schätzen) gekennzeichnet. Eine zusätzliche Angabe einer Schrittnummer i gibt an, welche IN_e- und OUT_e-Variablen zusammengehören. Es gilt: Um die Informationen der $\mathsf{OUT}_{e,i}$-Variable vom Dienstgeber zu erhalten, müssen zunächst vom Dienstnehmer alle $\mathsf{IN}_{e,i}$-Variablen spezifiziert werden.
- Variablen, die in der Ausführungsphase auszufüllen sind, tragen den Index x (für execute, engl. ausführen). Dabei gilt, dass IN_x-Variablen vor der Dienstausführung vom Dienstnehmer ausgefüllt werden müssen, während OUT_x-Variablen nach der Dienstausführung vom Dienstgeber bekannt gegeben werden. Wird kein Index gegeben, so gilt die Ausführungsphase als Standard.

Eine Variable gehört immer zu mindestens einer Kategorie, kann jedoch auch zu mehreren Kategorien gehören, etwa wenn ein Wert sowohl in der Schätz- als auch in der Ausführungsphase benötigt wird.

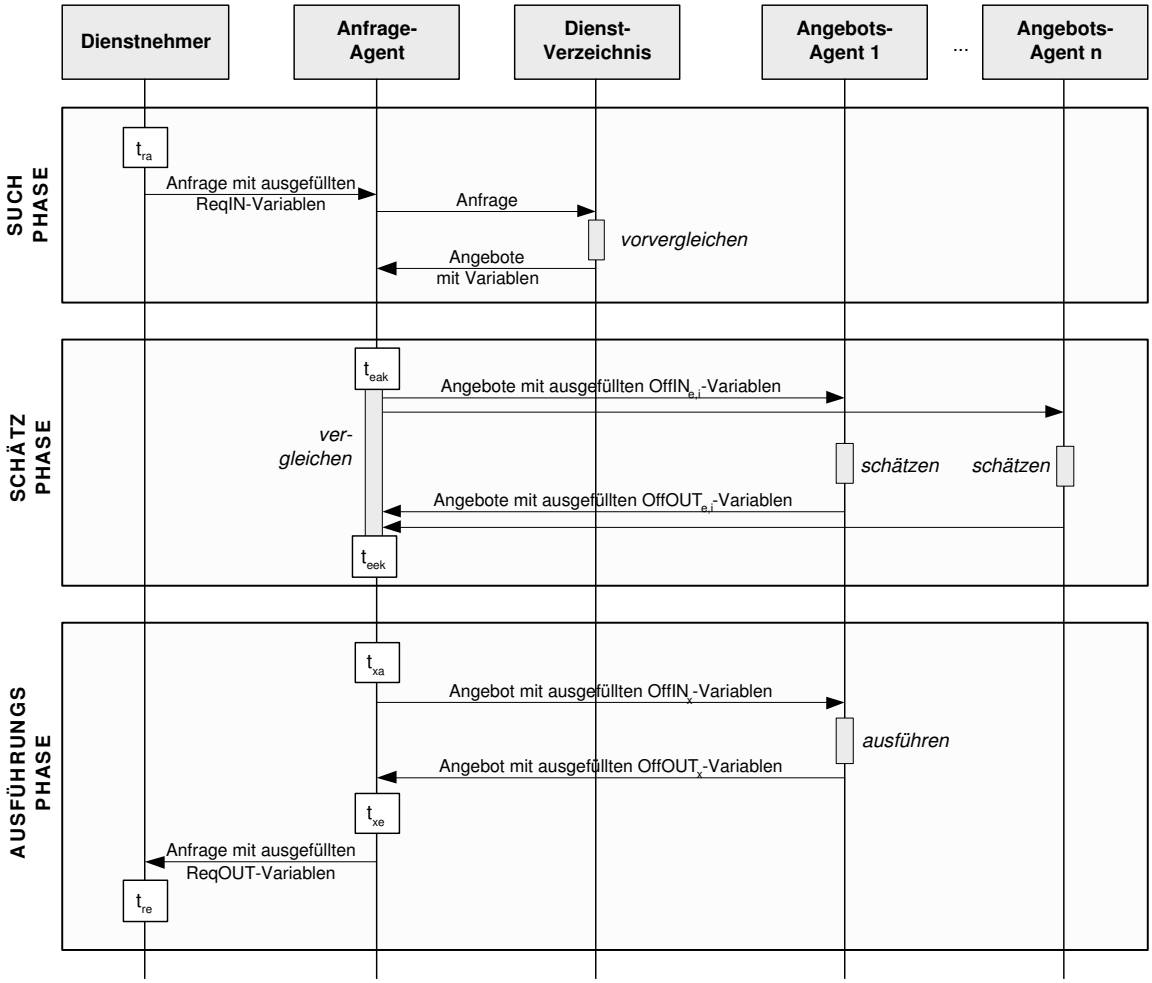


Abbildung 7.11.: Verlauf der Variablenbindung während einer Dienstnutzung.

Insgesamt sind die verschiedenen Variablen zu folgenden Zeitpunkten zu füllen (siehe auch Abbildung 7.11):

- ReqIN-Variablen sind vor dem Aufruf der Anfrage zu binden, d.h. vor Zeitpunkt t_{ra} müssen alle Variablen dieses Typs gefüllt sein (Regel 8.103).
- $\mathsf{OffIN}_{e,i}$-Variablen sind vor Absenden einer Schätzanfrage des Schritts i zu binden, d.h. vor einem Zeitpunkt t_{ea_k} müssen alle $\mathsf{OffIN}_{e,i}$ gefüllt sein (Regel 8.96).
- $\mathsf{OffOUT}_{e,i}$-Variablen sind nach Beendigung einer Schätzanfrage des Schritts i gebunden, d.h. vor einem Zeitpunkt t_{ee_k} müssen alle $\mathsf{OffOUT}_{e,i}$ gefüllt sein (Regel 8.97).
- OffIN_x-Variablen sind vor dem Absenden des Dienstaufrufs zu binden, d.h. vor Zeitpunkt t_{xa} müssen alle Variablen dieses Typs gefüllt sein (Regel 8.98).

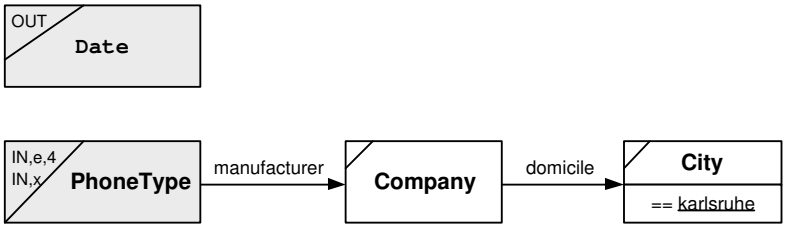


Abbildung 7.12.: Beispiele für Variablen mit Kategorieangaben in g-dsd.

- OffOUT$_x$-Variablen sind nach Beendigung einer Dienstausführung gebunden, d.h. vor Zeitpunkt t_{xe} müssen alle Variablen dieses Typs gefüllt sein (Regel 8.100).
- ReqOUT-Variablen sind nach Beendigung des Aufrufs einer Anfrage gebunden, d.h. vor Zeitpunkt t_{re} müssen alle Variablen dieses Typs gefüllt sein (Regel 8.105).

In g-dsd werden die Kategorien einer Variable untereinander in der Form `IN|OUT, e|x,` i in die linke obere Ecke eingetragen. Hierbei sind Abkürzungen möglich: x bzw. 1 kann ausgelassen werden. Beispiele in g-dsd zeigt Abbildung 7.12.

Operatoren

Auch die operationalen Sprachelemente erhalten innerhalb von Dienstbeschreibungen eine erweiterte Semantik. Für sie gilt, dass sich die Zeitpunkte, wann die Vorbedingungen überprüft und die Effekte erwirkt werden, am Verlauf der Dienstnutzung orientieren:

- Für Operatoren in einer Angebotsbeschreibung *off* gilt: Wenn alle Existenztests gegeben durch <<precondition>> an der Instanz von ServiceProfile in *off* zum Zeitpunkt t_{xa} erfüllt sind (Zustandsbedingung, siehe Regel 8.99), dann wird zum Zeitpunkt t_{xe} durch jeden Effekt gegeben durch <<effect>> o an der Instanz von ServiceProfile eine neue Instanz erzeugt, die zur Operandenmenge o gehört (Zustandsleistung, siehe Regel 8.101).
- Für Operatoren in einer Anfragebeschreibung *req* gilt: Es sollen zum Zeitpunkt t_{xe} durch jeden <<effect>>-Operator mit dem Operanden o an der Instanz von ServiceProfile in *req* eine neue Instanz, die zur Operandenmenge o gehört, entstanden sein, jedoch keine weiteren (Zustandswunsch, siehe Regel 8.104).

Im Vergleich zur dienstneutralen Semantik werden demnach die Zeitpunkte t_1 und t_2 zu t_{xa} und t_{xe} konkretisiert.

Unscharfe Mengen

Unscharfe Mengen sind nur in Anfragebeschreibungen möglich und treten dort als Operanden von <<effect>>-Operatoren auf. Sie haben dort eine erweiterte Semantik, indem sie die Präferenzen des Dienstnehmers ausdrücken. Es gilt: Sei o eine unscharfe Menge als Operand eines Effektoperators in einer Anfragebeschreibung. Stehen zwei Effekte e_1 und e_2 zur Verfügung, die beide Elemente von o sind (d.h. Zugehörigkeitswerte > 0 besitzen), so präferiert der Dienstnehmer denjenigen mit dem höheren Zugehörigkeitswert zur Menge. Effekte mit einem Zugehörigkeitswert von 0 gehören nicht zur Menge und sind daher unerwünscht. Unter Effekten mit gleichem Zugehörigkeitswert hat der Dienstnehmer keine Präferenzen. Die Präferenz für einen Effekt e wird daher direkt auf die Zugehörigkeit von e in o abgebildet.

7.2.3. Zusammenfassung: Gesamtsemantik von Dienstbeschreibungen

Insgesamt kann die Semantik einer **Dienstangebotsbeschreibung** als *Vertrag* aufgefasst werden, der aus drei Bestandteilen besteht: einem Schätzungsteil, der festlegt, wie zusätzliche Information über den Dienst angefordert werden kann, einem Bedingungsteil, der die Vorbedingungen auflistet, die zu einer korrekten Dienstausführung nötig sind, sowie einem Leistungsteil, der beschreibt, welche Wirkung bei einer erfolgreichen Dienstausführung entsteht.

Der *Schätzungsteil* wird über die $\mathsf{OffIN}_{e,i}$- und $\mathsf{OffOUT}_{e,i}$-Variablen beschrieben. Der Klient hat die Möglichkeit, beliebig viele Schätzungen über den Dienst einzuholen. Durch Anfordern und Berechnung eines Schätzergebnisses wird keiner der beschriebenen Effekte erbracht. Die Schätzphase ist daher generell seiteneffektfrei.

Der *Bedingungsteil* beschreibt die Anforderungen des Dienstes als *Informations-* und *Zustandsbedingungen*. Für eine erfolgreiche Dienstausführung müssen beide Bedingungen erfüllt sein. Die Informationsbedingungen werden implizit über OffIN_x-Variablen ausgedrückt. All diese Variablen müssen zu Beginn der Ausführung gefüllt sein und so den angebotenen Dienst mit nötigen Informationen zu versorgen. Die Zustandsbedingungen werden explizit über <<precondition>>-Operatoren notiert. Der Dienst kann nur dann erfolgreich arbeiten, wenn zu Beginn der Dienstausführung keine der zugehörigen Operandenmengen leer ist.

Der *Leistungsteil* beschreibt, welche Wirkung der Dienst erbringen kann, wenn er unter gültigen Bedingungen aufgerufen wird. Zunächst gilt, dass der Dienst die Ausführung ablehnen kann, obwohl alle Bedingungen erfüllt sind, etwa aufgrund einer unterspezifizierten Dienstbeschreibungen (siehe Abschnitt 7.3.2). In diesem Fall wird *keine* Leistung (auch keine Teilleistung) erbracht. Lehnt der Dienst die Ausführung nicht ab,

so werden *alle* beschriebenen *Informations-* und *Zustandsleistungen* erbracht. Informationsleistungen sind implizit über OffOUT_x-Variablen beschrieben, deren Füllung nach der Dienstausführung zugesichert wird. Daneben sichert der Dienst Zustandsleistungen zu: Durch jeden Effektoperator der Dienstbeschreibung entsteht eine neue Instanz in der Operandenmenge (und somit auch eine Wirkung in der realen Welt).

Die Semantik einer **Dienstanfragebeschreibung** kann als *Wunsch* aufgefasst werden, der aus zwei Teilen besteht: einem *Informationswunsch*, der ausdrückt, über welche Informationen der Dienstnehmer am Ende der Dienstausführung verfügen möchte, sowie einem *Zustandswunsch*, der ausdrückt, an welchen Folgezuständen der Dienstnehmer interessiert ist. Der Informationswunsch wird implizit über ReqOUT-Variablen ausgedrückt, die nach Dienstausführung gefüllt vorliegen sollen. Der Zustandswunsch wird über die Effektoperatoren ausgedrückt, für deren Operandenmengen je ein neues Element erzeugt werden soll. Sind diese Mengen unscharf, so werden Instanzen mit höherem Zugehörigkeitswert zur Menge präferiert.

7.3. Besondere Eigenschaften von Dienstbeschreibungen

Dienste und ihre Beschreibungen können besondere Eigenschaften besitzen:

- Es muss unterschieden werden, ob angebotene Dienste *eindeutig* oder *mehrdeutig* spezifizierbar sind, d.h. ob der Dienstnehmer den zu erbringen Effekt eindeutig auswählen oder lediglich einschränken kann.

- Dienstangebotsbeschreibungen können *unterbestimmt* sein, d.h. sie beschreiben die im angebotenen Dienst erzielbaren Effekte nicht exakt, sondern lediglich eine Obermenge davon. Bei solchen Dienstbeschreibungen kann sich der Dienstnehmer nicht sicher sein, dass ein Aufruf des Dienstes mit einer gültigen Belegung der Eingabevariablen nicht vom Dienstgeber abgelehnt wird.

- Für Vorbedingungen in Dienstangebotsbeschreibungen muss unterschieden werden, ob diese von Dienstnehmer *beeinflussbar* oder unbeeinflussbar sind bzw. ob sie für den Dienstnehmer oder Dienstgeber *auswertbar* sind oder nicht.

Im Folgenden sollen diese Eigenschaften genauer untersucht werden.

7.3.1. Eindeutig und mehrdeutig spezifizierbare Dienste

Angebotene Dienste können in der Regel eine Familie von ähnlichen Effekten erbringen. Die Menge aller von einem Dienst d tatsächlich erbringbarer Effekte soll mit $\mathcal{D}$ bezeichnet werden. Es sei zudem s die Dienstbeschreibung dieses Dienstes mit den IN_x-Variablen IN^1, IN^2, ..., IN^n. Der Raum aller nach s gültigen IN-Variablenbelegungen sei $\mathcal{I}$, ein Element hieraus wird mit $(i_1, i_2, ..., i_n)$ bezeichnet.

Typischerweise kann der Dienst d bei einer gegebenen IN-Variablenbelegung eine Reihe möglicher Effekte erbringen, die konform zu s sind. Die *Effektmöglichkeiten* von d können daher durch eine Funktion ep_d (ep = effect possibilities) beschrieben werden:

$$ep_d : \mathcal{I} \longrightarrow \mathcal{P}(\mathcal{D}) \tag{7.1}$$

Die Kardinalität der jeweiligen Menge der Effektmöglichkeiten bestimmt dann, ob ein Dienst eindeutig oder mehrdeutig spezifiziert ist (siehe dazu Abbildung 7.13):

- Ein Dienst ist genau dann *eindeutig spezifizierbar*, wenn für alle $i \in \mathcal{I}$ gilt: $|ep_d(i)| \leq 1$. Bei so spezifizierten Diensten ist also der zu erwartende Effekt nach Füllung aller IN-Variablen eindeutig festgelegt, sofern der Dienst mit dieser Variablenbelegung überhaupt ausgeführt werden kann (vgl. unterbestimmte Dienstbeschreibungen im nächsten Abschnitt). Beispiel für einen solchen Dienst ist ein Druckdienst, bei dem der Dienstnehmer über IN-Variablen alle Eigenschaften des Ausdrucks festlegen kann. Der zu erwartende Effekt ist also bekannt.
- Ein Dienst ist genau dann *mehrdeutig spezifizierbar*, wenn ein $i \in \mathcal{I}$ existiert, sodass gilt: $|ep_d(i)| > 1$. Bei so spezifizierten Diensten liegt der tatsächlich zu erbringende Effekt teilweise im Ermessen des Dienstgebers, indem er selbstständig

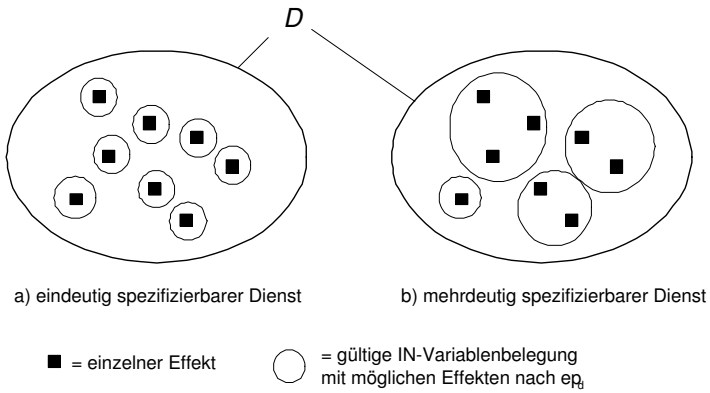


Abbildung 7.13.: Unterscheidung zwischen (a) eindeutig und (b) mehrdeutig spezifizierbaren Diensten.

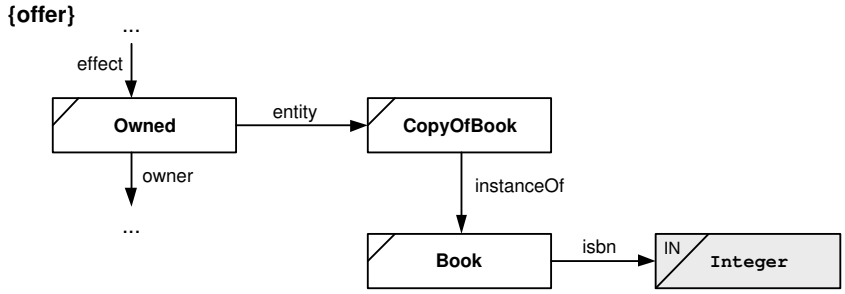


Abbildung 7.14.: Ausschnitt aus einer Beschreibung für einen Dienst zum Verkauf von Büchern.

einen der möglichen Effekte aus der Menge $ep_d(i)$ der Effektmöglichkeiten auswählt. Der Dienstnehmer kann daher nur begrenzt Einfluss auf die Wirkung des Dienstes nehmen. Beispiel für einen solchen Dienst ist ein Dienst zur Reservierung von Kinokarten, bei dem der Dienstnehmer über IN-Variablen nur den gewünschten Film und eine Startzeit angeben kann. Die genaue Platz- und Saalnummer entscheidet der Dienstgeber jedoch eigenständig.

7.3.2. Unterbestimmte Dienstbeschreibungen

Bestimmte Dienste bieten an, einen Effekt für eine große Menge möglicher Entitäten erbringen zu können. Ein Beispiel hierfür könnte ein Dienst sein, über den ein Buch gekauft werden kann. Bei solchen Diensten ist es oft der Fall, dass die Entität, auf der der Dienst operiert, nicht zu einer öffentlichen Klasse gehört. Dies kann mehrere Gründe haben: Zum einen könnte es zu aufwändig sein, die Vielzahl der Instanzen allgemein zu veröffentlichen (z.B. alle existierenden Bücher), zum anderen könnte sich die Menge der Instanzen häufig ändern. Auch Situationen, in denen Informationen über Instanzen selbst einen Wert für den Dienstanbieter darstellen, können dafür ausschlaggebend sein, Klassen nicht als öffentliche Entitätsklassen zu definieren.

Abbildung 7.14 zeigt einen Ausschnitt aus dem Schema und einer beispielhaften Beschreibung eines Dienstes zum Kauf eines Buches. Die Auswahl des Buches (als teilöffentliche Entitätsklasse) findet dabei nicht über Angabe der Book-Instanz statt, sondern erfolgt eindeutig über das definierende Attribut isbn. Dennoch kann es aus zwei Gründen dazu kommen, dass der Dienst nicht ausgeführt werden kann:

- Der Dienstnehmer wählt zwar typgerecht einen korrekten Füllwert für die IN-Variablen aus, dieser führt aber zu keiner allgemein veröffentlichen oder privat hinterlegten Instanz. Im Beispiel tritt dieser Fall ein, wenn der Dienstnehmer einen Integerwert angibt, der keiner ISBN-Nummer eines Buches entspricht,

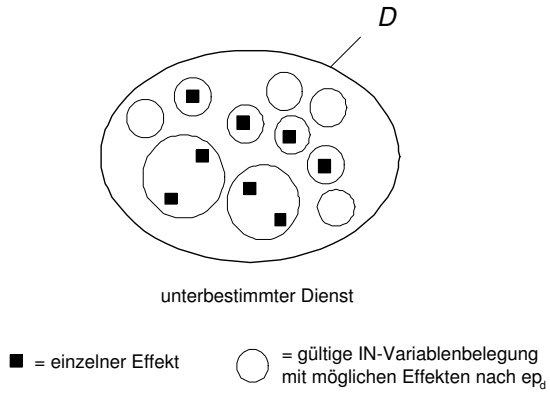


Abbildung 7.15.: Darstellung einer unterbestimmten Dienstbeschreibung.

z.B. 18. In diesem Fall ist die Book-Menge leer; der Dienst kann nicht ausgeführt werden.

- Der Dienstnehmer wählt die Füllwerte für die IN-Variablen so, dass hierdurch eine nicht-leere Menge von Entitäten entsteht. Dies kann möglich sein, wenn er Werte von allgemein veröffentlichten oder Instanzen aus seinem privaten Instanzenpool beziehen kann. Im Beispiel könnte der Dienstnehmer eine ISBN-Nummer eines Buches angeben, dessen Instanz im allgemeinen Instanzenpool hinterlegt ist. Trotzdem könnte der Dienst die Ausführung ablehnen, da er genau dieses Buch nicht liefern kann.

In beiden Fällen führt also eine eigentlich gültige Belegung der IN-Variablen zu leeren Mengen von Effektmöglichkeiten für den Dienstgeber, d.h. der Dienst kann nicht ausgeführt werden. Wir definieren wie folgt (siehe Abbildung 7.15):

- Eine Dienstbeschreibung ist genau dann *unterbestimmt*, wenn ein $i \in \mathcal{I}$ existiert, sodass $ep_d(i) = \emptyset$. Dienste, die mit einer solchen Variablenbelegung i aufgerufen werden, weisen daher den Aufruf vor der eigentlichen Ausführung ab.

Beim Vergleich von Dienstbeschreibungen muss der Vergleicher daher gegebenenfalls zusätzlich bestimmen, mit welcher Wahrscheinlichkeit ein Dienst mit einer bestimmten IN-Variablenbelegung auch ausgeführt werden kann. Diese *Ausführungswahrscheinlichkeit* ist ein Maß dafür, wie hoch die Chance ist, einen bestimmten Effekt tatsächlich erwirkt zu bekommen. Bei mehreren gleich gut passenden Dienstangeboten sollte daher der Dienst mit der höchsten Ausführungswahrscheinlichkeit bevorzugt werden. Weiterhin sollten Dienste mit sehr geringer Ausführungswahrscheinlichkeit nicht weiter betrachtet werden.

Der Vergleicher kann die Ausführungswahrscheinlichkeit nicht definitiv berechnen, sondern nur grob abschätzen. Hierbei können ihm zusätzliche Angaben in der Angebotsbeschreibung behilflich sein. Der Dienstgeber sollte daher bei jeder Menge vom Typ einer Entitätsklasse, deren Elemente über Pfade von Attributbedingungen durch IN-Variablen spezifiziert werden können, angeben, wie viele Instanzen der Dienst unterstützt. Es existieren folgende *Kardinalitätsmarkierer*:

- `all values`. Gibt an, dass jede beliebige Belegung der IN-Variablen vom Dienst verarbeitet werden kann. Die Ausführungswahrscheinlichkeit ist in einem solchen Fall immer 1.

- `all entities`. Gibt an, dass der Dienst immer ausgeführt werden kann, sofern die Belegung der IN-Variablen zu einer nicht-leeren Menge von Entitäten führt. Dabei können die enthaltenen Instanzen allgemein veröffentlicht oder im privaten Instanzenpool von Dienstnehmer oder -geber zu finden sein. Führt die Belegung zu einer leeren Menge von Entitäten, kann der Dienst nicht ausgeführt werden. Dies ist der Standardfall, falls kein Markierer spezifiziert ist.

- n `entities`. Gibt an, dass der Dienst auch dann nicht immer ausgeführt werden kann, wenn die Belegung der IN-Variablen zu einer nicht-leeren Menge von Entitäten führt, etwa weil der Dienst die spezifizierten Entitäten entgegen der Dienstbeschreibung nicht unterstützt. Die Zahl n gilt dann als Übersicht für die Dienstnehmer, wie viele Entitäten der Dienstgeber insgesamt unterstützt.

7.3.3. Eigenschaften von Vorbedingungen

Für eine Angebotsbeschreibung werden zwei Arten von Vorbedingungen unterschieden:

- *Informationsvorbedingungen* sind Bedingungen, die sicherstellen, dass der Dienstgeber mit allen Informationen versorgt wird, die er zum korrekten Ausführen des Dienstes benötigt. Sie werden durch OffIN-Variablen ausgedrückt.

- *Zustandsvorbedingungen* sind Bedingungen, die sicherstellen, dass zu Beginn der Dienstausführung bestimmte Eigenschaften der Welt gegeben sind, die zum korrekten Ablauf des Dienstes erforderlich sind. Sie werden durch den Operator <<precondition>> ausgedrückt.

Beeinflussbarkeit und Auswertbarkeit von Vorbedingungen

Bei Vorbedingungen wird unterschieden, ob diese vom Dienstnehmer beeinflussbar sind oder nicht, d.h. man untersucht, ob es für den Dienstnehmer möglich ist, unerfüllte Vorbedingungen gezielt zu erfüllen. Man definiert daher:

- Eine Vorbedingung heißt *beeinflussbar*, wenn der Dienstnehmer prinzipiell die Möglichkeit hat, sie gezielt zu erfüllen. Typischerweise sind alle Informationsvorbedingungen beeinflussbar.
- Eine Vorbedingung heißt *nicht beeinflussbar*, wenn der Dienstnehmer keinen Einfluss darauf hat, ob diese zu Beginn der Dienstausführung erfüllt ist oder nicht.

Zudem werden Zustandsvorbedingungen dahingehend unterschieden, ob und von wem sie ausgewertet werden können:

- Eine Zustandsvorbedingung heißt *vom Dienstgeber direkt auswertbar*, falls dieser in der Lage ist, die Leere oder Nichtleere der zugehörigen Menge allein aufgrund seines ontologischen Wissens (bestehend aus allgemeinem Wissen über Schema und öffentliche Instanzen und Wissen über persönliche Instanzen im privaten Instanzenpool) festzustellen.
- Eine Zustandsvorbedingung heißt *vom Dienstgeber indirekt auswertbar*, falls dieser in der Lage ist, die Leere oder Nichtleere der zugehörigen Menge mittels zusätzlicher, auch externer Wissensquellen und Dienste festzustellen. Diese Feststellung ist typischerweise mit einer gewisser Fehlerwahrscheinlichkeit behaftet.
- Eine Zustandsvorbedingung heißt *vom Dienstnehmer auswertbar*, falls dieser in der Lage ist, lokal bestimmen zu können, ob die zugehörige Menge leer ist oder nicht, ohne zuvor die Dienstausführung anstoßen zu müssen. Auch hier kann zwischen direkter und indirekter Auswertbarkeit unterschieden werden.

Die Auswertbarkeit hat Einfluss darauf, welche Ausführungswahrscheinlichkeit für den Dienst besteht. Ist eine Bedingung zwar vom Dienstgeber auswertbar, vom Dienstnehmer aber nicht, so kann es vorkommen, dass der Dienst so aufgerufen wird, dass diese Bedingung vor der Dienstausführung nicht erfüllt ist. Dies wird jedoch erst vom Dienstgeber erkannt, der die Ausführung daraufhin abweist. Für den Dienstnehmer hat der Dienst dann eine Ausführungswahrscheinlichkeit kleiner als 1.0.

Ist eine Bedingung vom Dienstgeber nicht oder nur ungenau auswertbar, kann der Fall eintreten, dass die Dienstausführung vom Dienstgeber nicht zurückgewiesen wird, obwohl die Bedingung vor der Dienstausführung nicht erfüllt war. In diesem Fall läuft die Dienstausführung unter falschen Voraussetzungen, was zu fehlerhaften, unvollständigen oder unerwünschten Ergebnissen führen kann. Wenn sie nachträglich erkannt werden, können sie gegebenenfalls durch Kompensationsfunktionen rückgängig gemacht werden.

7.4. Kommentierte Beispiele für Dienstbeschreibungen

Dieser Abschnitt zeigt drei beispielhafte Dienstbeschreibungen: ein *Wissensdienst* zur Erlangung von Informationen über Telefonmodelle, ein *Informationsdienst* zur Beschaffung einer bestimmten Datei sowie ein *Realweltdienst*, welcher den Kauf eines Telefons ermöglicht.

7.4.1. Beispielbeschreibung eines Wissensdienstes

Eine Beispielbeschreibung für einen Wissensdienst ist in Abbildung 7.16 gegeben. Durch Ausführung des Dienstes erfährt der Dienstnehmer die Füllwerte einer Instanz, hier die einer Zustandsinstanz vom Typ Rated, die sich auf ein Telefonmodell bezieht. Mit Hilfe des Dienstes können also Bewertungen von Telefonmodellen abgefragt werden. Als Eingabe (OffIN-Variable) ist der Name eines neueren Telefonmodells nötig, das seit 2001 verfügbar sein muss. Als Ausgaben (OffOUT-Variablen) und damit zur Erbringung des Effekts liefert der Dienst den Bewerter (valuer) und seine Bewertung (value) im Fünf-Sterne-Bewertungsschema.

7.4.2. Beispielhafte Anfragebeschreibung nach einem Informationsdienst

Abbildung 7.17 zeigt eine beispielhafte Dienstanfragebeschreibung. Gesucht wird nach einem Dienst, der eine Datei mit einer Abbildung von einem Telefon zugänglich macht, etwa um dieses im Produktkatalog einer Webapplikation darstellen zu können. Der Wunsch wird durch die Zustandsmenge vom Typ Accessible ausgedrückt wird. Die Eigenschaften der Informationseinheit sind auf verschiedenen Ebenen zu finden[3]:

- Auf unterster Ebene ist die Information lediglich ein uninterpretierter Bytestrom (ByteStream), der vom Nutzer mittels der ReqOUT-Variable als Rückgabe verlangt wird.

[3]siehe dazu die Ontologie `domain.information` in Anhang D

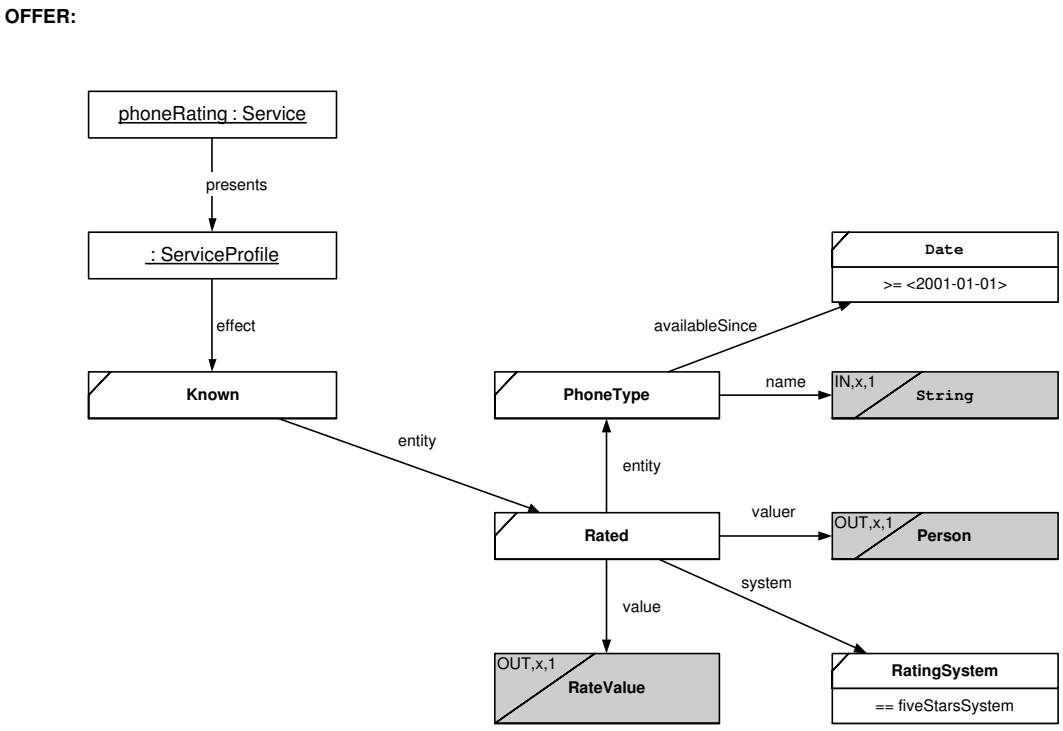


Abbildung 7.16.: Beispiel für einen Wissensdienst, mit dessen Hilfe die Bewertung von Telefonmodellen abgefragt werden kann.

- Eine Materialisierung des Bytestroms stellt eine Datei (File) dar. Im Beispiel wird eine Datei in einem für das Internet geeignete Format wie jpg oder gif verlangt. Das Format png wird zwar auch akzeptiert, jedoch nur mit einer Präferenz von 0,5.

- Eine Datei repräsentiert ein Dokument, hier vom Untertyp Photo. Gewünscht ist ein großes Foto mit Abmessungen von mindestens 800×600 Bildpunkten. Bilder mit geringeren Abmessungen sind bis zu einer Größe von 200×200 Bildpunkten mit abnehmender Präferenz noch zufrieden stellend. Durch die alternative Typvergleichsstrategie sind auch Instanzen der direkten Oberklasse von Photo, hier also Image, und damit andere Bildtypen wie Zeichnungen unter einer verminderten Präferenz von 0,4 möglich.

- Der Inhalt des Fotos soll eine Abbildung (Depiction) sein. Gezeigt werden soll ein Telefon, möglichst von vorne. Fehlt in einer Angebotsbeschreibung die Angabe dieses Blickwinkels, soll dieser dennoch mit einer verminderten Präferenz von 0,8 betrachtet werden.

- Die gezeigte Entität, hier das Telefon, soll von einem Modell sein, dass der Dienstnehmer vor dem jeweiligen Absenden der Anfrage durch die ReqIN-Variable

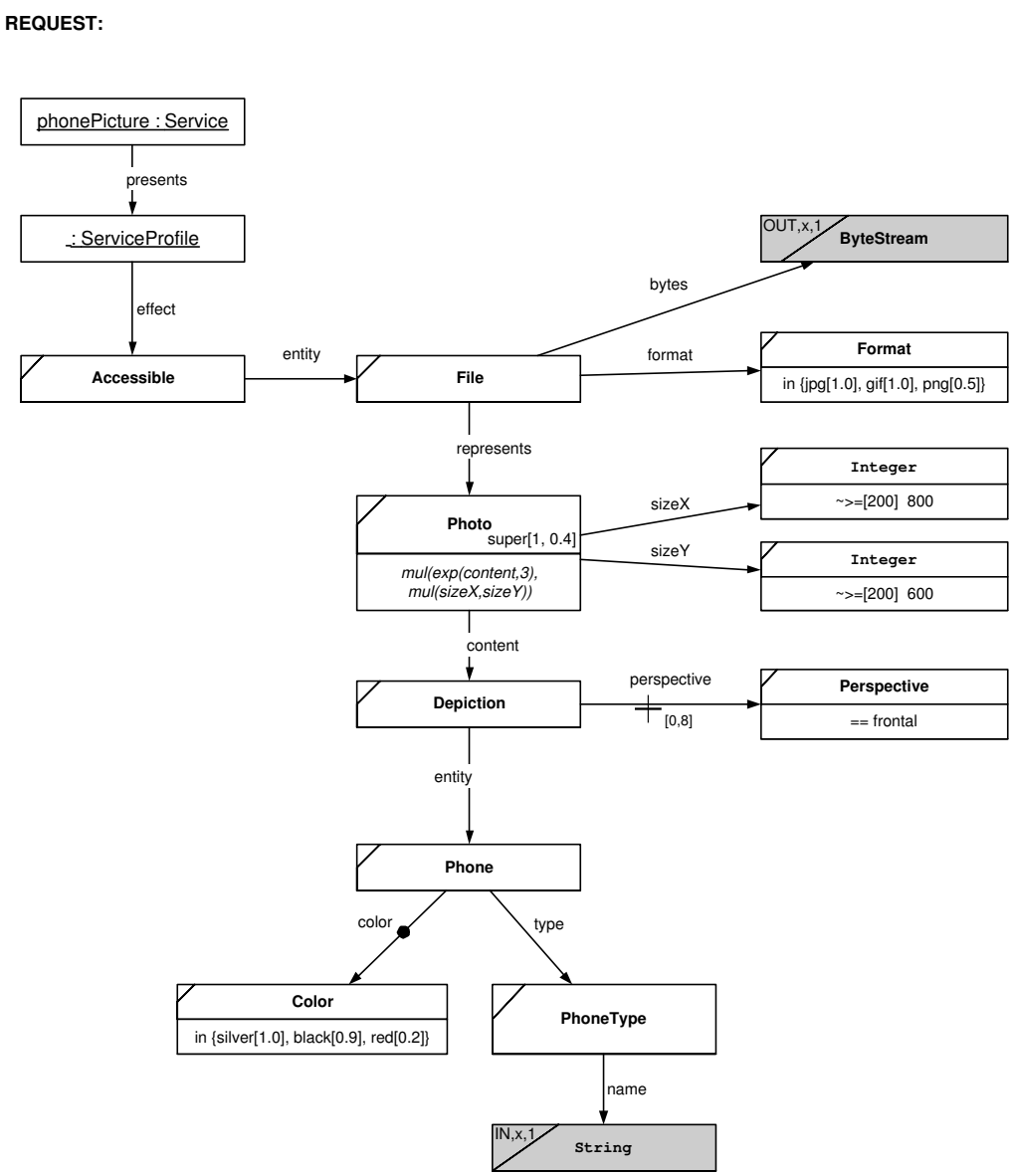


Abbildung 7.17.: Beispielhafte Anfragebeschreibung nach einem Dienst, der eine Abbildung eines Telefons liefern kann.

bestimmen kann, und eine der angegebenen Farben haben. Durch die geänderte Fehlstrategie ist festgelegt, dass im Falle einer im Angebot nicht angegebenen Farbinformation die Bedingung ignoriert werden darf.

Interessant ist die veränderte Verbindungsstrategie in der Photo-Menge. Sie besagt, dass die Präferenzen für Inhalt und Größe des Fotos zwar konjunktiv verknüpft sind (`mul`-Operator), die Präferenzen nach einem geeigneten Inhalt aufgrund des Exponenten 3 dabei jedoch stärker sind.

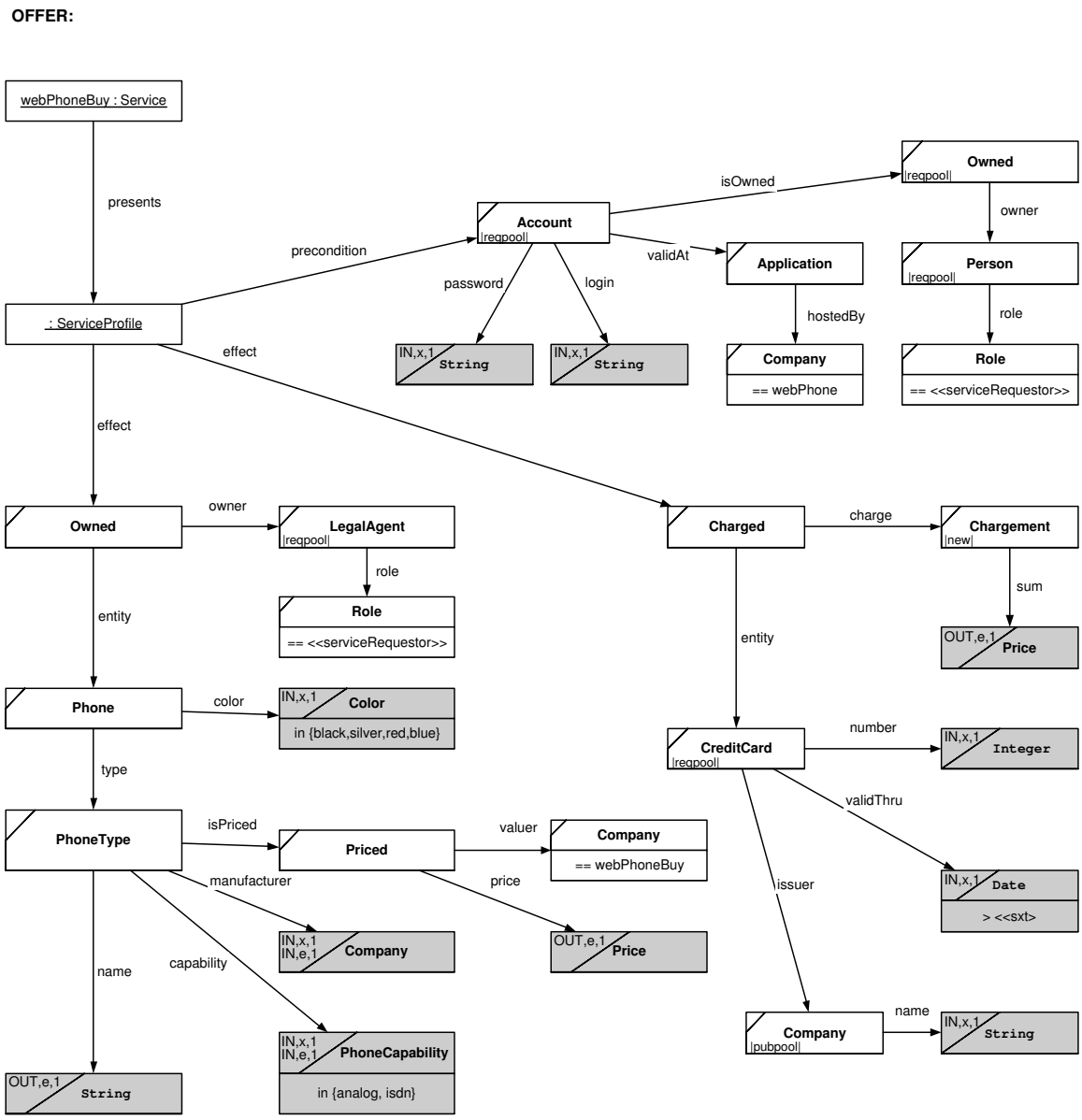


Abbildung 7.18.: Beschreibung eines Realweltdienstes, über den Telefone erworben werden können.

7.4.3. Beispielbeschreibung eines Realweltdienstes

Abbildung 7.18 beschreibt einen angebotenen Realweltdienst, mit dem Telefongeräte erworben werden können. Das Dienstprofil zeigt dazu zwei Effekte und eine Vorbedingung:

- Der wichtigste Effekt verweist auf eine Zustandsmenge vom Typ Owned, d.h. ein

neuer Besitzzustand wird geschaffen. Der neue Besitzer (owner) ist die Person in der Rolle des Dienstnehmers; die erworbene Entität (entity) ist ein Gerät eines bestimmten Telefonmodells. Es kann vom Klienten indirekt über zwei OffIN_x-Variablen festgelegt werden: über die gewünschte Farbe, welche schwarz, silber, rot oder blau sein kann, sowie den Modellnamen.

- Der zweite Effekt beschreibt die Bezahlung durch Belastung einer Kreditkarte. Bei Dienstausführung entsteht also eine neue Instanz der Zustandsklasse Charged. Die belastete Entität ist eine Kreditkarte, die vor der Dienstnutzung über drei OffIN_x-Variablen vom Klienten bekannt gegeben werden muss: über die Nummer der Kreditkarte, ihr Gültigkeitsdatum sowie über den Namen der ausstellenden Institution. Der Belastungsbetrag in sum lässt sich dank der $\mathsf{OffOUT}_{e,1}$-Variable vorab durch eine Schätzanfrage ermitteln. Dazu sind vom Klienten alle $\mathsf{OffIN}_{e,1}$-Variablen, also hier Firma und Fähigkeiten des gewünschten Telefonmodells zu füllen; die Farbe ist für den Preis offensichtlich unerheblich. Neben dem Belastungsbetrag der Kreditkarte wird in diesem Schätzschritt auch der Grundpreis und der Name des ausgewählten Telefons bekannt gegeben.
- Eine Vorbedingung zur Nutzung des Dienstes ist die Existenz eines bestimmten Accounts, dargestellt durch die Menge vom Typ Account. Der Zugang muss vor dem Dienstaufruf durch Angabe von Login und Passwort vom Klienten spezifiziert werden. Er muss dem Dienstnehmer gehören und bei der Firma webPhone gültig sein.

Interessant sind folgende Punkte der Dienstbeschreibung:

- Die Vorbedingung, dass der angegebene Account dem Dienstnehmer selbst gehört, ist vom Dienstgeber nur indirekt auswertbar.
- Der Dienst ist unterbestimmt, da nicht alle existierenden, durch Namen bestimmbare Telefonmodelle geliefert werden können. Nur der Dienstanbieter kennt alle Telefonmodelle (PhoneType ist eine nicht-öffentliche Entitätsklasse).
- Die Daten zur Befüllung der IN-Variablen zur Bestimmung des Accounts und der Kreditkarte sind typischerweise nicht in einer passenden Anfragebeschreibung aufgeführt. Sie werden vom Vergleicher aus dem privaten Instanzenpool des Dienstnehmers herangezogen (falls dieser das wünscht).
- Wird der Dienst erfolgreich ausgeführt, werden stets *beide Effekte* erwirkt, d.h. es findet immer ein Besitzwechsel *und* eine Kreditkartenbelastung statt.
- Explizit nicht beschrieben sind das Verhalten im Fehlerfall, die interne Abwicklung des Kaufvorgangs sowie die Berechnung des neuen Belastungsstandes der Kreditkarte. All diese Informationen sind zum Auffinden des Dienstes unerheblich oder sogar schädlich, da sie von der eigentlichen Funktionalität ablenken.

7.5. Zusammenfassung

In diesem Kapitel wurde die Dienstbeschreibungssprache *DIANE Service Description* vorgestellt, die auf DIANE Elements I und II basiert. Zur Beschreibung von Diensten greift DSD auf eine Schichtung von Ontologien zurück: Die obere Dienstontologie legt die Grundstruktur fest, Kategorieontologien konzeptualisieren den Raum der Zustände, die durch Dienste verändert werden können, und in eine obere Ontologie eingebettete Domänenontologien bestimmen das Vokabular für die verschiedenen Anwendungsgebiete. Die Semantik einer Dienstbeschreibung leitet sich im Wesentlichen aus den in ihr verwendeten neuen Sprachelementen aus DE-II ab, die hierdurch eine erweiterte, dienstspezifische Semantik erlangen. Diese ist durch bestimmte Zeitpunkte und Teilnehmer während der Dienstnutzung definiert.

8. Axiomatische Semantik von DE und DSD

Ziel dieses Kapitels ist es, die Semantik von DE und DSD formal zu definieren.[1] Hierzu existieren im Wesentlichen zwei Möglichkeiten: (1) ein modelltheoretischer Ansatz, bei dem eine (gedachte) Abbildung zwischen Instanzen, Klassen und Beziehungen von DSD auf die reale Welt eingeführt wird, mit deren Hilfe die Bedeutung der weiteren Konstrukte erklärt werden kann, (2) ein axiomatischer Ansatz, bei dem die Sprache auf eine andere Sprache abgebildet wird, für die bereits eine formale Semantik definiert ist. Für DSD wurde der zweite Ansatz gewählt und eine Abbildung auf eine Form der modalen, temporalen Prädikatenlogik erster Ordnung mit Identität vorgenommen.

8.1. Modale, temporale Prädikatenlogik

Die temporale Prädikatenlogik bedient sich folgender Bausteine:

- Eine Menge von Individualkonstanten $\mathcal{K}$, die für verschiedene Sachverhalte wie etwa Klassennamen, Instanznamen, Mengennamen usw. stehen können. Teilmenge der Individualkonstanten ist die Menge der Literale $\mathcal{L}$.
- Eine Menge von Individualvariablen $\mathcal{X}$. Jede Individualvariable steht stellvertretend für ein Element aus $\mathcal{K}$. Im Folgenden werden Individualvariablen mit Kleinbuchstaben notiert.
- Eine Menge von Prädikaten $\mathcal{P}$. Jedes Prädikat hat eine festgelegte Stelligkeit. Beispiele sind `instanceof`, `class` und `contains`. Prädikate unterteilen sich in die Menge der konstanten Prädikate $\mathcal{P}_{const}$ und in die Menge der zeitlich veränderlichen Prädikate $\mathcal{P}_{var}$.
- Die bekannten aussagenlogischen Junktoren $\wedge$, $\vee$, $\neg$, $\rightarrow$ und $\leftrightarrow$.
- Die Quantoren $\forall$ und $\exists$.

[1] Die formale Semantik ist für das weitere Verständnis der Arbeit nicht zwingend erforderlich.

- Eine Gleichheitsrelation auf Individualkonstanten (und somit auch auf Literalen), dargestellt durch $\doteq$. Zwei Individualkonstanten sind genau dann gleich, wenn sie identisch sind. Die zugehörige Ungleichungsrelation wird mit $\neq$ notiert.
- Eine Menge von Zeitpunkten $\mathcal{T} = \{t_0, t_1, \ldots\}$, die über die totale Ordnungsrelation $\preceq$ geordnet sind. $\mathcal{T}$ stellt somit den „Fluss der Zeit“ dar. Während konstante Prädikate ihre Wahrheitswerte unabhängig von t_i bestimmen, hängt der Wahrheitswert eines variablen Prädikats von t_i ab.
- Die temporalen Operatoren $\bigcirc$, $\Box$ und $\Diamond$ nach [88], wobei gilt:
  - $\bigcirc F$: F trifft im unmittelbar folgenden Zeitpunkt zu (*nexttime*-Operator).
  - $\Box F$: F trifft in allen nachfolgenden Zeitpunkten zu (*always*-Operator).
  - $\Diamond F$: F trifft in (mindestens) einem nachfolgenden Zeitpunkt zu (*sometime*-Operator).
- Der modale *Soll*-Operator $\rhd$ (vgl. dazu [132], Abschnitt 8.2.3). $\rhd F$ besagt, dass gewünscht ist, dass F zutrifft.

Für die Operatoren gelten folgende Prioritäten bei der Bindung:

- $\forall$, $\exists$, $\neg$, $\bigcirc$, $\Box$, $\Diamond$ und $\rhd$ binden stärker als $\doteq$.
- $\doteq$ bindet stärker als $\wedge$ und $\vee$.
- $\wedge$ und $\vee$ binden stärker als $\rightarrow$.
- $\rightarrow$ bindet stärker als $\leftrightarrow$.

8.2. Klassen und Instanzen

Jeder eindeutige Name für einen **primitiven Datentyp** ist in $\mathcal{K}$ enthalten. Es existiert ein einstelliges Prädikat `datatype`. `datatype`(D) besagt, dass D einen Datentyp darstellt. Es existieren 8 Datentypen: `Integer`, `Double`, `String`, `Boolean`, `Date`, `Time`, `DateTime` und `Duration`.

Jeder eindeutige **Klassenname** ist in $\mathcal{K}$ enthalten. Es existiert ein einstelliges Prädikat `class`. `class`(C) besagt, dass C eine Klasse darstellt. Es existiert eine ausgezeichnete Klasse `Thing`, die auch in $\mathcal{K}$ enthalten ist.

Jeder Name einer **benamten Instanz** ist in $\mathcal{K}$ enthalten. Auch anonyme Instanzen erhalten einen künstlichen, eindeutigen Namen und sind in $\mathcal{K}$ enthalten. Es existiert

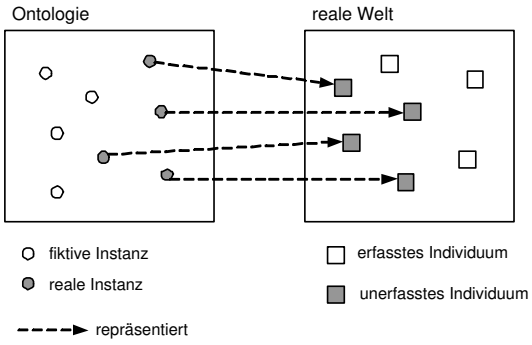


Abbildung 8.1.: Unterscheidung zwischen realen und fiktiven Instanzen zur Bestimmung des zeitlich variablen Prädikats `real`.

ein zweistelliges Prädikat `instanceof`. $\texttt{instanceof}(I, C)$ besagt, dass Instanz I vom Typ C ist.

Für Instanzen existiert ein zeitlich variables Prädikat `real`. $\texttt{real}(I)$ sagt aus, ob es zu der Instanz I (abhängig vom Zeitpunkt der Auswertung) ein zugehöriges Individuum in der realen Welt gibt. Abbildung 8.1 verdeutlicht diesen Zusammenhang: Zu jedem Zeitpunkt existieren *reale Instanzen* (mit zugehörigen Individuen) und *fiktive* Instanzen (ohne zugehörige Individuen) sowie *erfasste Individuen* (mit abbildenden Instanzen) und *unerfasste Individuen* (ohne abbildende Instanzen).

Jede Instanz hat einen Typ:

$$\forall i.\exists c.\texttt{instanceof}(i, c) \tag{8.1}$$

Jedes **Literal** ist in $\mathcal{L} \subseteq \mathcal{K}$ enthalten. Es existiert ein zweistelliges Prädikat `instanceof`. $\texttt{instanceof}(L, D)$ besagt, dass L ein Literal des Datentyps D ist.

Jedes Literal gehört zu mindestens einem Datentyp:

$$\forall l.\exists d.\texttt{instanceof}(l, d) \tag{8.2}$$

Jedes Literal gehört zu höchstens einem Datentyp:

$$\texttt{instanceof}(l, d) \wedge \texttt{instanceof}(l, e) \rightarrow d \doteq e \tag{8.3}$$

Jeder **Attributname** ist in $\mathcal{K}$ enthalten. Es existieren folgende Prädikate:

- $\texttt{defprop}(C, A, D)$. Besagt, dass A ein definierendes Attribut von Klasse C mit dem Zieltyp D ist.

- $\texttt{incprop}(C, A, D)$. Besagt, dass A ein ableitbares Attribut von Klasse C mit dem Zieltyp D ist.
- $\texttt{orthprop}(C, A, D)$. Besagt, dass A ein orthogonales Attribut von Klasse C mit dem Zieltyp D ist.
- $\texttt{prop}(C, A, D)$. Besagt, dass A irgendein Attribut von Klasse C mit dem Zieltyp D ist.

Es gilt, dass jede Art von Attribut ein allgemeines Attribut ist:

$$\texttt{defprop}(c, a, d) \rightarrow \texttt{prop}(c, a, d) \tag{8.4}$$
$$\texttt{incprop}(c, a, d) \rightarrow \texttt{prop}(c, a, d) \tag{8.5}$$
$$\texttt{orthprop}(c, a, d) \rightarrow \texttt{prop}(c, a, d) \tag{8.6}$$

Jedes Attribut ist von einer dieser drei Arten:

$$\texttt{prop}(c, a, d) \rightarrow \texttt{defprop}(c, a, d) \vee \texttt{incprop}(c, a, d) \vee \texttt{orthprop}(c, a, d) \tag{8.7}$$

Kein Attribut kann von verschiedenen Typen sein:

$$\texttt{defprop}(c, a, d) \rightarrow \neg\texttt{incprop}(c, a, d)) \tag{8.8}$$
$$\texttt{defprop}(c, a, d) \rightarrow \neg\texttt{orthprop}(c, a, d) \tag{8.9}$$
$$\texttt{incprop}(c, a, d) \rightarrow \neg\texttt{defprop}(c, a, d) \tag{8.10}$$
$$\texttt{incprop}(c, a, d) \rightarrow \neg\texttt{orthprop}(c, a, d) \tag{8.11}$$
$$\texttt{orthprop}(c, a, d) \rightarrow \neg\texttt{defprop}(c, a, d) \tag{8.12}$$
$$\texttt{orthprop}(c, a, d) \rightarrow \neg\texttt{incprop}(c, a, d) \tag{8.13}$$

Kein Attribut kann mehrere Zieltypen haben:

$$\texttt{prop}(c, a, d) \wedge \texttt{prop}(c, a, e) \rightarrow d \doteq e \tag{8.14}$$

Attribute von Instanzen können mit konkreten Instanzen oder Literalen gefüllt werden. Dies drückt das zweistellige Prädikat `filler` aus. $\texttt{filler}(I, A, J)$ besagt, dass für Instanz I das Attribut A mit dem Füllwert J belegt ist.

Für einen Füllwert muss ein Attribut mit passendem Zieltyp existieren:

$$\texttt{filler}(\texttt{i}, \texttt{a}, \texttt{j}) \wedge \texttt{instanceof}(i, c) \rightarrow \exists d.(\texttt{prop}(c, a, d) \wedge \texttt{instanceof}(j, d)) \quad (8.15)$$

Eine Instanz kann für ein definierendes oder ableitbares Attribut nur maximal einen Füllwert besitzen:

$$\begin{aligned} \texttt{filler}(i, a, j) \wedge \texttt{filler}(i, a, k) \\ \wedge(\texttt{defprop}(c, a, d) \vee \texttt{incprop}(c, a, d)) \rightarrow j \doteq k \end{aligned} \quad (8.16)$$

Definierende Attribute müssen immer gefüllt sein:

$$\texttt{instanceof}(i, c) \wedge \texttt{defprop}(c, a, d) \rightarrow \exists j.\texttt{filler}(i, a, j) \quad (8.17)$$

Definierende Attribute bestimmen die ableitbaren Attribute:

$$\begin{aligned} \texttt{instanceof}(i, c) \wedge \texttt{instanceof}(j, c) \\ \wedge\forall a.(\texttt{defprop}(c, a, d) \wedge \texttt{filler}(i, a, k) \wedge \texttt{filler}(j, a, l) \rightarrow k \doteq l) \\ \wedge\texttt{incprop}(c, b, e) \wedge \texttt{filler}(i, b, m) \wedge \texttt{filler}(j, b, n) \\ \rightarrow m \doteq n \end{aligned} \quad (8.18)$$

Orthogonale Attribute haben nur hinweisende Funktion und können nicht direkt gefüllt werden. Ihre Füllwerte sind indirekt über die zugehörigen verdinglichten Zustandsklassen definiert, die über das Attribut entity auf die beschriebene Instanz zeigen (siehe dazu auch Abschnitt 5.2.2):

$$\begin{aligned} \texttt{instanceof}(i, c) \wedge \texttt{orthprop}(c, a, d) \\ \rightarrow (\texttt{filler}(i, a, j) \leftrightarrow \exists j.(\texttt{instanceof}(j, d) \wedge \texttt{filler}(j, \mathsf{entity}, i))) \end{aligned} \quad (8.19)$$

Klassen können in einer **Vererbungsbeziehung** stehen. Dies wird durch das zweistellige Prädikat `subclass` ausgedrückt. $\texttt{subclass}(C, D)$ drückt aus, dass C eine Unterklasse von D ist.

`subclass` ist reflexiv:

$$\texttt{subclass}(c, c) \tag{8.20}$$

`subclass` ist transitiv:

$$\texttt{subclass}(c, d) \wedge \texttt{subclass}(d, e) \rightarrow \texttt{subclass}(c, e) \tag{8.21}$$

Alle Klassen haben Thing als Oberklasse:

$$\texttt{subclass}(c, \mathsf{Thing}) \tag{8.22}$$

Alle Instanzen werden nach oben vererbt:

$$\texttt{instanceof}(i, c) \wedge \texttt{subclass}(c, d) \rightarrow \texttt{instanceof}(i, d) \tag{8.23}$$

Attribute werden nach unten vererbt und behalten dabei Typ und Zieltyp:

$$\texttt{defprop}(c, a, d) \wedge \texttt{subclass}(e, c) \rightarrow \texttt{defprop}(e, a, d) \tag{8.24}$$
$$\texttt{incprop}(c, a, d) \wedge \texttt{subclass}(e, c) \rightarrow \texttt{incprop}(e, a, d) \tag{8.25}$$
$$\texttt{orthprop}(c, a, d) \wedge \texttt{subclass}(e, c) \rightarrow \texttt{orthprop}(e, a, d) \tag{8.26}$$

Eine direkte Oberklasse einer Klasse heißt *Vaterklasse*, was durch das Prädikat `father` zum Ausdruck kommt:

$$\begin{aligned} \texttt{father}(c, d) \\ \leftrightarrow [\texttt{subclass}(c, d) \wedge c \neq d \\ \wedge \neg\exists e.(\texttt{subclass}(e, d) \wedge \texttt{subclass}(c, e) \wedge e \neq d \wedge c \neq e)] \end{aligned} \tag{8.27}$$

Jede Klasse hat höchstens eine Vaterklasse:[2]

$$\texttt{father}(c, d) \wedge \texttt{father}(c, e) \rightarrow d \doteq e \tag{8.28}$$

Eine Instanz hat den *speziellsten Typ* C, wenn sie nicht auch Instanz einer Unterklasse von C ist. Diese Definition gilt auch für Literale und wird durch das Prädikat `type` ausgedrückt:

[2]Insgesamt hat dadurch jede Klasse außer Thing genau eine eindeutige Vaterklasse.

$$\begin{aligned} &\texttt{type}(i,c) \\ &\leftrightarrow [\texttt{instanceof}(i,c) \\ &\wedge\neg\exists d.(\texttt{subclass}(d,c) \wedge d \neq c \wedge \texttt{instanceof}(i,d))] \end{aligned} \tag{8.29}$$

Jede Instanz hat höchstens einen speziellsten Typ:

$$\texttt{type}(i,c) \wedge \texttt{type}(i,d) \rightarrow c \doteq d \tag{8.30}$$

Zur Unterscheidung von **Instanzarten** existieren zwei zweistellige Prädikate: `namedinstanceof` und `anoninstanceof`. $\texttt{namedinstanceof}(I,C)$ besagt, dass I eine benamte Instanz von C ist. $\texttt{anoninstanceof}(I,C)$ besagt, dass I eine anonyme Instanz von C ist.

Sowohl benamte als auch anonyme Instanzen sind allgemeine Instanzen:

$$\texttt{namedinstanceof}(i,c) \rightarrow \texttt{instanceof}(i,c) \tag{8.31}$$
$$\texttt{anoninstanceof}(i,c) \rightarrow \texttt{instanceof}(i,c) \tag{8.32}$$

Jede Instanz ist von einem der beiden Arten:

$$\texttt{instanceof}(i,c) \rightarrow \texttt{anoninstanceof}(i,c) \vee \texttt{namedinstanceof}(i,c) \tag{8.33}$$

Die Arten schließen sich gegenseitig aus:

$$\texttt{namedinstanceof}(i,c) \rightarrow \neg\texttt{anoninstanceof}(i,c) \tag{8.34}$$
$$\texttt{anoninstanceof}(i,c) \rightarrow \neg\texttt{namedinstanceof}(i,c) \tag{8.35}$$

Wie das Prädikat `instanceof` werden diese an Oberklassen vererbt:

$$\texttt{namedinstanceof}(i,c) \wedge \texttt{subclass}(c,d) \rightarrow \texttt{namedinstanceof}(i,d) \tag{8.36}$$
$$\texttt{anoninstanceof}(i,c) \wedge \texttt{subclass}(c,d) \rightarrow \texttt{anoninstanceof}(i,d) \tag{8.37}$$

Auch für **Klassenarten** existieren zwei einstellige Prädikate: `valueclass` und `entityclass`. `valueclass`(C) besagt, dass C eine wertbestimmte Klasse ist; `entityclass`(C) besagt, dass C eine Entitätsklasse ist.

Sowohl wertbestimmte als auch Entitätsklassen sind allgemeine Klassen:

$$\texttt{valueclass}(c) \rightarrow \texttt{class}(c) \tag{8.38}$$
$$\texttt{entityclass}(c) \rightarrow \texttt{class}(c) \tag{8.39}$$

Die Arten schließen sich gegenseitig aus:[3]

$$\texttt{valueclass}(c) \rightarrow \neg\texttt{entityclass}(c) \tag{8.40}$$
$$\texttt{entityclass}(c) \rightarrow \neg\texttt{valueclass}(c) \tag{8.41}$$

Alle Attribute wertbestimmter Klassen sind definierend:

$$\texttt{valueclass}(c) \wedge \texttt{prop}(c, a, d) \rightarrow \texttt{defprop}(c, a, d) \tag{8.42}$$

Ist eine Klasse wertbestimmt, so auch ihre Unterklassen:

$$\texttt{valueclass}(c) \wedge \texttt{subclass}(d, c) \rightarrow \texttt{valueclass}(d) \tag{8.43}$$

Ist eine Klasse eine Entitätsklasse, so auch ihre Unterklassen:

$$\texttt{entityclass}(c) \wedge \texttt{subclass}(d, c) \rightarrow \texttt{entityclass}(d) \tag{8.44}$$

Wertbestimmte Klassen haben nur anonyme Instanzen, Entitätsklassen nur benamte Instanzen:

$$\texttt{valueclass}(c) \wedge \texttt{instanceof}(i, c) \rightarrow \texttt{anoninstanceof}(i, c) \tag{8.45}$$
$$\texttt{entityclass}(c) \wedge \texttt{instanceof}(i, c) \rightarrow \texttt{namedinstanceof}(i, c) \tag{8.46}$$

Entitätsklassen können zusätzlich öffentlich sein. Dafür existiert das einstellige Prädikat `public`. `public`(C) besagt, dass C eine öffentliche Klasse ist.

[3] hierdurch sind die Vererbungsbeschränkungen nach Abschnitt 5.2.6 bereits erfasst

Nur Entitätsklassen können öffentlich sein:

$$\texttt{public}(c) \rightarrow \texttt{entityclass}(c) \tag{8.47}$$

Ist eine Klasse öffentlich, so auch ihre Unterklassen:

$$\texttt{public}(c) \wedge \texttt{subclass}(d, c) \rightarrow \texttt{public}(d) \tag{8.48}$$

Eine Instanz ist öffentlich, wenn im öffentlichen Instanzenpool eine zentrale Kopie hinterlegt ist. Dies wird durch das einstellige Prädikat `publicinst` ausgedrückt. $\texttt{publicinst}(I)$ besagt also, dass I eine öffentliche Instanz ist.

Öffentliche Klassen haben nur öffentliche Instanzen:

$$\texttt{public}(c) \wedge \texttt{instanceof}(i, c) \rightarrow \texttt{publicinst}(i) \tag{8.49}$$

Teilöffentliche Klassen können sowohl öffentliche als auch private Instanzen besitzen.

Anmerkung: Für diese Abbildung zur Definition der Semantik ist die Unterscheidung in öffentliche und teilöffentliche Klassen bzw. öffentliche und private Instanzen nicht direkt von Belang, da von einem *allwissenden Ansatz* ausgegangen wird, d.h. jede Instanz (und auch jedes Literal) ist durch eine eindeutige Individualkonstante ausgezeichnet und allgemein sichtbar. Hierdurch vereinfachen sich die Regeln erheblich, ohne dass die Exaktheit der Semantik leidet. Für die Praxis muss jedoch stets bedacht werden, dass nicht zu jedem Zeitpunkt allen Teilnehmern alle Instanzen zur Verfügung stehen.

Die zweistellige **Gleichheitsrelation** `equals` für Literale und Instanzen ist wie folgt definiert:

Für Literale leitet sich `equals` von ihrer natürlichen Gleichheit ab:

$$\begin{aligned}\texttt{datatype}(d) \wedge \texttt{instanceof}(l, d) \wedge \texttt{instanceof}(m, d) \\ \rightarrow [\texttt{equals}(l, m) \leftrightarrow l \doteq m]\end{aligned} \tag{8.50}$$

Zwei benamte öffentliche Instanzen sind gleich, wenn sie den gleichen Namen haben:

$$\begin{aligned}\texttt{namedinstanceof}(i, c) \wedge \texttt{namedinstanceof}(j, c) \\ \wedge \texttt{publicinst}(i) \wedge \texttt{publicinst}(j) \\ \rightarrow [\texttt{equals}(i, j) \leftrightarrow i \doteq j]\end{aligned} \tag{8.51}$$

Zwei benamte private Instanzen sind gleich, wenn sie den gleichen speziellsten Typ besitzen und die Füllwerte der korrespondierenden definierenden Attribute gleich sind:[4]

$$\begin{aligned} \texttt{namedinstanceof}(i,c) \wedge \texttt{namedinstanceof}(j,c) \quad (8.52)\\ \wedge \neg\texttt{publicinst}(i) \wedge \neg\texttt{publicinst}(j)\\ \rightarrow [\texttt{equals}(i,j) \leftrightarrow \exists e.(\texttt{type}(i,e) \wedge \texttt{type}(j,e))\\ \wedge\forall a.(\texttt{defprop}(c,a,d) \wedge \texttt{filler}(i,a,k) \wedge \texttt{filler}(j,a,l) \rightarrow \texttt{equals}(l,k))] \end{aligned}$$

Zwei anonyme Instanzen sind gleich, wenn sie den gleichen speziellsten Typ besitzen und die Füllwerte der korrespondierenden Attribute gleich sind:

$$\begin{aligned} \texttt{instanceof}(j,c) \wedge \texttt{instanceof}(i,d) \wedge \texttt{valueclass}(c) \wedge \texttt{valueclass}(d) \quad (8.53)\\ \rightarrow [\texttt{equals}(i,j) \leftrightarrow \exists e.(\texttt{type}(i,e) \wedge \texttt{type}(j,e))\\ \wedge\forall a.(\texttt{filler}(i,a,k) \wedge \texttt{filler}(j,a,l) \rightarrow \texttt{equals}(l,k))] \end{aligned}$$

Zwei Instanzen/Literale verschiedener Typen sind immer ungleich (somit auch z.B. Instanzen verschiedener Klassenarten oder eine Instanz und ein Literal):

$$\neg\exists c.(\texttt{type}(i,c) \wedge \texttt{type}(j,c)) \rightarrow \neg\texttt{equals}(i,j) \quad (8.54)$$

Durch diese Definitionen ist `equals` bereits reflexiv, transitiv und symmetrisch.

8.3. Mengen

8.3.1. Aufbau von Mengen

Zur Definition von **Mengen** sind eine Reihe weiterer Individualkonstanten nötig:

- Bezeichner für Mengen. Es existiert ein einstelliges Prädikat `set`. $\texttt{set}(S)$ besagt, dass S eine Menge ist.
- Bezeichner für Typbedingungen. Es existiert ein dreistelliges Prädikat `typecondition`. $\texttt{typecondition}(TC, S, C)$ besagt, dass TC eine Typbedingung ist, die für die Menge S Instanzen der Klasse C verlangt.

[4] Hier wären auch komplexere, domänenspezifische Definitionen aus dem Gebiet der Informationsintegration denkbar.

- Bezeichner für direkte Bedingungen. Es existiert ein zweistelliges Prädikat `directcondition`. `directcondition`(DC, S) besagt, dass DC eine direkte Bedingung der Menge S ist.

- Bezeichner für enumerierte Mengen. Es existiert ein einstelliges Prädikat `enumset`. `enumset`(ES) besagt, dass ES eine enumerierte Menge ist. Die Elemente der Menge werden durch das zweistellige Prädikat `element` definiert. `element`(ES, K) besagt, dass die Instanz oder das Literal K Element der Menge ES ist.

- Bezeichner für Attributbedingungen. Es existiert ein dreistelliges Prädikat `propertycondition`. `propertycondition`(PC, A, T) besagt, dass PC eine Attributbedingung des Attributs A in die Zielmenge T ist.[5].

- Bezeichner für Verbindungsstrategien. Es existiert ein vierstelliges Prädikat `connectingstrategy`. `connectingstrategy`$(CS, T0, T1, T2)$ besagt, dass CS eine Verbindungsstrategie darstellt, die zwei Attributbedingungen oder weitere Verbindungsstrategien $T1$ und $T2$ zusammenfasst und an die Menge oder Verbindungsstrategie $T0$ anhängt.

Die Typvergleichsstrategie *super* wird durch das vierstellige Prädikat `typecondition_super` ausgedrückt. `typecondition_super`$(TC, S, C, STEP)$ besagt, dass TC eine Typbedingung ist, die für die Menge S die Klasse C oder eine Oberklasse, die maximal $STEP$ Schritte entfernt ist, verlangt.

Für direkte Bedingungen existieren zwei Spezialisierungen, die durch die zwei dreistelligen Prädikate `directcondition_eq` und `directcondition_in` ausgedrückt werden. `directcondition_eq`(DC, S, J) besagt, dass zur Menge S die direkte Bedingung „$== J$“ hinzugefügt wird. `directcondition_in`(DC, S, ES) besagt, dass zur Menge S die direkte Bedingung „in X“ hinzugefügt wird, wobei X die Elemente der enumerierten Menge ES enthält. Da beides Spezialformen sind, gilt:

$$\texttt{directcondition\_eq}(dc, s, j) \rightarrow \texttt{directcondition}(dc, s) \tag{8.55}$$

$$\texttt{directcondition\_in}(dc, s, es) \rightarrow \texttt{directcondition}(dc, s) \tag{8.56}$$

Für Attributbedingungen existieren drei Spezialisierungen abhängig von der gewählten Fehlstrategie, die durch drei dreistellige Prädikate ausgedrückt werden `propertycondition_failed`, `propertycondition_ignore` und `propertycondition_fulfilled`. Es gilt:

[5]Die Verbindung zur Menge geschieht indirekt über Verbindungsstrategien.

$$\texttt{propertycondition\_failed}(pc, a, t) \rightarrow \texttt{propertycondition}(pc, a, t) \quad (8.57)$$
$$\texttt{propertycondition\_ignore}(pc, a, t) \rightarrow \texttt{propertycondition}(pc, a, t) \quad (8.58)$$
$$\texttt{propertycondition\_fulfilled}(pc, a, t) \rightarrow \texttt{propertycondition}(pc, a, t) \quad (8.59)$$

Für Verbindungsstrategien existieren zwei Spezialisierungen, die durch zwei vierstellige Attribute ausgedrückt sind: `connectingstrategy_and` sowie `connectingstrategy_or`. Es gilt:

$$\texttt{connectingstrategy\_and}(cs, t0, t1, t2) \rightarrow \texttt{connectingstrategy}(cs, t0, t1, t2) \quad (8.60)$$
$$\texttt{connectingstrategy\_or}(cs, t0, t1, t2) \rightarrow \texttt{connectingstrategy}(cs, t0, t1, t2) \quad (8.61)$$

Zur Vereinfachung wird das Attribut `flatcs` eingeführt. `flatcs`(S,T) sammelt alle Verbindungsstrategien und Attributbedingungen T, die direkt oder indirekt an der Menge S hängen.

$$\texttt{connectingstrat}(cs, s, t1, t2) \rightarrow \texttt{flatcs}(s, t1) \wedge \texttt{flatcs}(s, t2) \quad (8.62)$$
$$\texttt{flatcs}(s, t) \wedge \texttt{connectingstrat}(cs, t, t3, t4) \rightarrow \texttt{flatcs}(s, t3) \wedge \texttt{flatcs}(s, t4) \quad (8.63)$$

8.3.2. Elemente einer Menge, contains

Wichtig für eine Menge S ist die Semantik, ob eine Instanz I Element der Menge ist oder nicht. Dies wird durch das zweistellige Prädikat `contains` ausgedrückt. Hierfür sind eine Reihe von Hilfsprädikaten nötig, die im Folgenden eingeführt werden. Eine entscheidende Rolle spielt das zweistellige Prädikat `fulfilled`. `fulfilled`(B, I) dass besagt, dass Bedingung B für die Instanz I erfüllt ist.

Da Bedingungen im Zuge der Fehlstrategie *ignore* auch ignoriert werden können, kann nicht immer eindeutig angegeben werden, ob `fulfilled` gilt oder nicht. Es wird daher zu einer **dreiwertigen Logik** erweitert, indem drei weitere zweistellige Prädikate eingeführt werden: `fulfilled_true`, `fulfilled_false` und `fulfilled_neutral`. Für eine Bedingung und eine Instanz sind diese disjunkt:

$$\texttt{fulfilled\_true}(b, i) \rightarrow \neg\texttt{fulfilled\_false}(b, i) \quad (8.64)$$
$$\texttt{fulfilled\_true}(b, i) \rightarrow \neg\texttt{fulfilled\_neutral}(b, i) \quad (8.65)$$

$$\texttt{fulfilled\_neutral}(b,i) \rightarrow \neg\texttt{fulfilled\_true}(b,i) \tag{8.66}$$
$$\texttt{fulfilled\_neutral}(b,i) \rightarrow \neg\texttt{fulfilled\_false}(b,i) \tag{8.67}$$
$$\texttt{fulfilled\_false}(b,i) \rightarrow \neg\texttt{fulfilled\_true}(b,i) \tag{8.68}$$
$$\texttt{fulfilled\_false}(b,i) \rightarrow \neg\texttt{fulfilled\_neutral}(b,i) \tag{8.69}$$

Zwischen der dreiwertigen und zweiwertigen Logik gelten folgende Zusammenhänge:

$$\texttt{fulfilled}(b,i) \rightarrow \texttt{fulfilled\_true}(b,i) \tag{8.70}$$
$$\neg\texttt{fulfilled}(b,i) \rightarrow \texttt{fulfilled\_false}(b,i) \tag{8.71}$$

Folgende Regeln legen fest, wann **Typbedingungen** erfüllt sind. Eine Typbedingung ist für eine Instanz bzw. ein Literal i genau dann erfüllt, wenn i vom Typ der angegebenen Klasse bzw. des angegebenen Datentyps ist:

$$\texttt{typecondition}(tc,s,c) \rightarrow \forall i.[\texttt{fulfilled}(tc,i) \leftrightarrow \texttt{instanceof}(i,c)] \tag{8.72}$$

Im Falle einer Typvergleichsstrategie *super* gilt:[6]

$$\begin{aligned}\texttt{typecondition\_super}(tc,s,c) \rightarrow \\ (step \doteq 0 \rightarrow \\ \forall i.[\texttt{fulfilled}(tc,i) \leftrightarrow \texttt{instanceof}(i,c)]) \\ \wedge(step \doteq 1 \rightarrow \\ \forall i.[\texttt{fulfilled}(tc,i) \leftrightarrow \texttt{instanceof}(i,c) \\ \vee\exists d.(\texttt{father}(c,d) \wedge \texttt{instanceof}(i,d))]) \\ \wedge(step \doteq 2 \rightarrow \ldots\end{aligned} \tag{8.73}$$

Folgende Regeln legen fest, wann **direkte Bedingungen** erfüllt sind. Im Falle einer _eq-Bedingung erfüllt nur diese Instanz die Bedingung:

$$\texttt{directcondition\_eq}(dc,s,j) \rightarrow \forall i.[\texttt{fulfilled}(dc,i) \leftrightarrow i \doteq j] \tag{8.74}$$

Im Falle einer _in-Bedindung kann ein beliebiges Element der enumerierten Menge die Bedingung erfüllen:

[6] Damit eine geschlossene Formel angegeben werden kann, muss die maximale Schrittzahl *step* beschränkt werden, was für die Praxis unkritisch ist.

$$\texttt{directcondition\_in}(dc, s, es) \rightarrow \forall i.[\texttt{fulfilled}(dc, i) \leftrightarrow \exists k.(\texttt{element}(es, k) \wedge i \doteq k)] \tag{8.75}$$

Folgende Regeln legen fest, wann **Attributbedingungen** erfüllt sind. Existiert der entsprechende Füllwert, so wird überprüft, ob er in der Zielmenge enthalten ist:

$$\begin{aligned}\texttt{propertycondition}(pc, a, t) \wedge \exists k.(\texttt{filler}(i, a, k)) \rightarrow \\ (\forall i.[\exists j.(\texttt{filler}(i, a, j) \wedge \texttt{contains}(t, j)) \rightarrow \texttt{fulfilled\_true}(pc, i)] \wedge \\ \forall i.[\forall j.(\texttt{filler}(i, a, j) \wedge \neg\texttt{contains}(t, j)) \rightarrow \texttt{fulfilled\_false}(pc, i)])\end{aligned} \tag{8.76}$$

Existiert für eine Instanz *das Attribut* gar nicht, auf dem eine Attributbedingung definiert ist (etwa durch eine geänderte Typvergleichsstrategie), so wird diese Bedingung ignoriert:

$$\begin{aligned}\texttt{propertycondition}(pc, a, s) \wedge \texttt{instance}(i) \\ \wedge\texttt{type}(i, t) \wedge \neg\exists d.\texttt{prop}(t, a, u) \\ \rightarrow \forall i.\texttt{fulfilled\_neutral}(pc, i)\end{aligned} \tag{8.77}$$

Existiert zwar das Attribut, jedoch *der Füllwert* nicht, so hängt der Grad der Erfüllung von der Fehlstrategie ab:

$$\begin{aligned}\texttt{propertycondition\_failed}(pc, a, t) \wedge \texttt{flatcs}(s, pc) \\ \wedge\texttt{typecondition}(tc, s, c) \wedge \exists d.\texttt{prop}(c, a, d) \\ \rightarrow \forall i.[\neg\exists j.\texttt{filler}(i, a, j) \rightarrow \texttt{fulfilled\_false}(pc, i)]\end{aligned} \tag{8.78}$$

$$\begin{aligned}\texttt{propertycondition\_ignore}(pc, a, t) \wedge \texttt{flatcs}(s, pc) \\ \wedge\texttt{typecondition}(tc, s, c) \wedge \exists d.\texttt{prop}(c, a, d) \\ \rightarrow \forall i.[\neg\exists j.\texttt{filler}(i, a, j) \rightarrow \texttt{fulfilled\_neutral}(pc, i)]\end{aligned} \tag{8.79}$$

$$\texttt{propertycondition\_fulfilled}(pc, a, t) \wedge \texttt{flatcs}(s, pc) \tag{8.80}$$

$$\wedge \texttt{typecondition}(tc, s, c) \wedge \exists d.\texttt{prop}(c, a, d)$$
$$\rightarrow \forall i.[\neg\exists j.\texttt{filler}(i, a, j) \rightarrow \texttt{fulfilled\_true}(pc, i)]$$

Folgende Regeln legen (rekursiv) fest, wann eine **Verbindungsstrategie** erfüllt ist. Für _and darf keiner der beiden Terme false sein. Sind beide neutral, ist auch das Ergebnis neutral:

$$\begin{aligned}
&\texttt{connectingstrategy_and}(c2, t0, t1, t2) \rightarrow \\
&\forall i.[\texttt{fulfilled_false}(t1, i) \vee \texttt{fulfilled_false}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_false}(t0, i)] \\
\wedge&\forall i.[\texttt{fulfilled_neutral}(t1, i) \wedge \texttt{fulfilled_neutral}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_neutral}(t0, i)] \\
\wedge&\forall i.[\texttt{fulfilled_true}(t1, i) \wedge \neg\texttt{fulfilled_false}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_true}(t0, i)] \\
\wedge&\forall i.[\neg\texttt{fulfilled_false}(t1, i) \wedge \texttt{fulfilled_true}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_true}(t0, i)]
\end{aligned} \tag{8.81}$$

Für _or reicht es, wenn einer der beiden Terme true ist. Sind beide neutral, ist auch das Ergebnis neutral:

$$\begin{aligned}
&\texttt{connectingstrategy_or}(c2, t0, t1, t2) \rightarrow \\
&\forall i.[\texttt{fulfilled_true}(t1, i) \vee \texttt{fulfilled_true}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_true}(t0, i)] \\
\wedge&\forall i.[\texttt{fulfilled_neutral}(t1, i) \wedge \texttt{fulfilled_neutral}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_neutral}(t0, i)] \\
\wedge&\forall i.[\texttt{fulfilled_false}(t1, i) \wedge \neg\texttt{fulfilled_true}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_false}(t0, i)] \\
\wedge&\forall i.[\neg\texttt{fulfilled_true}(t1, i) \wedge \texttt{fulfilled_false}(t2, i) \\
&\qquad \rightarrow \texttt{fulfilled_false}(t0, i)]
\end{aligned} \tag{8.82}$$

Das Prädikat fulfilled erlaubt es, die Semantik von **contains** zu definieren. Es gilt, dass eine Instanz i zu einer Menge s gehört, wenn i alle ihre Typ-, direkte und Attributbedingungen (entsprechend der Verbindungsstrategie) erfüllt:

$$\begin{aligned}\texttt{set}(s) \rightarrow (\texttt{contains}(i,s) \leftrightarrow \\ (\forall tc.\forall c.[\texttt{typecondition}(tc,s,c) \rightarrow \texttt{fulfilled\_true}(tc,i)] \\ \wedge\forall dc.\forall j.[\texttt{directcondition}(dc,s,j) \rightarrow \texttt{fulfilled\_true}(dc,i)] \\ \wedge\forall cs.\forall t1.\forall t2.[\texttt{connectingstrategy}(tc,s,t1,t2) \\ \rightarrow \neg\texttt{fulfilled\_false}(cs,i)]))\end{aligned} \tag{8.83}$$

8.4. Unscharfe Mengen

Wichtig zur Beschreibung von Anfragen in DSD ist das Konzept unscharfer Mengen. Diese erlauben im Gegensatz zu scharfen Mengen einen fließenden Zugehörigkeitswert aus dem Interval $[0, 1]$. Zur Abbildung ihrer Semantik benötigt man unscharfe Prädikatenlogik, d.h. eine Prädikatenlogik, bei der der Erfüllungsgrad eines Prädikats nicht nur `false` oder `true` annehmen kann, sondern kontinuierliche Werte aus $[0, 1]$ erlaubt. Diese Übersetzung soll hier nicht durchgeführt werden, da sie sich nicht wesentlich von der oben vorgestellten Abbildung unterscheidet. Die Vorgehensweise wäre die folgende:

- Die meisten der oben vorgestellten Regeln können für den unscharfen Fall übernommen werden, indem mathematische Operationen für die booleschen Operatoren $\wedge$, $\vee$ und $\neg$ festgelegt werden.
- Die unscharfen Typvergleichs-, Fehl- und Verbindungsstrategien müssen durch geeignete Prädikate abgebildet werden.

8.5. Variablen

Für Variablen werden zusätzliche Individualkonstanten eingeführt. Es existiert zudem ein zweistelliges Prädikat `variable`. $\texttt{variable}(V, S)$ besagt, dass V eine Variable mit der Grundmenge S ist.

Der **Bindungszustand** einer Variable wird durch vier zeitlich veränderliche Prädikate festgelegt:

- Das einstellige Prädikat `open`. $\texttt{open}(V)$ besagt, dass die Variable V noch ungebunden ist.

- Das zweistellige Prädikat `filled`. $\texttt{filled}(V, J)$ besagt, dass die Variable V mit der Instanz bzw. dem Literal J gefüllt ist.
- Das zweistellige Prädikat `connected`. $\texttt{connected}(V, W)$ besagt, dass die Variable V mit der Variable W verbunden ist.
- Das einstellige Prädikat `bound`. $\texttt{bound}(V)$ besagt, dass die Variable V auf irgendeine Weise gebunden ist.

Variablen können nur mit Werten aus ihrer Grundmenge gefüllt werden:

$$\texttt{variable}(v, s) \wedge \texttt{filled}(v, j) \rightarrow \texttt{contains}(s, j) \tag{8.84}$$

Es gilt: `bound` ist eine Zusammenfassung von `filled` und `connected`:

$$\begin{aligned} \texttt{filled}(v, j) &\rightarrow \texttt{bound}(v) \\ \texttt{connected}(v, w) &\rightarrow \texttt{bound}(v) \end{aligned} \tag{8.85}$$

Eine gebundene Variable ist entweder gefüllt oder verbunden:

$$\texttt{bound}(v) \rightarrow [\exists j.\texttt{filled}(v, j) \leftrightarrow \neg\exists w.\texttt{connected}(v, w)] \tag{8.86}$$

Entweder ist eine Variable ungebunden oder gebunden:

$$\texttt{variable}(v) \rightarrow [\texttt{open}(v) \leftrightarrow \neg\texttt{bound}(v)] \tag{8.87}$$

Durch die Erweiterung von **contains** auf Variablen können Variablen auch als Zielmenge von Attributbedingungen verwendet werden und somit semantisch eindeutig in Dienstbeschreibungen integriert werden. Die Semantik von $\texttt{contains}(V)$ hängt entscheidend vom Bindungszustand von V ab.

Eine ungebundene Variable wirkt wie ihre Grundmenge:

$$\texttt{variable}(v, s) \wedge \texttt{open}(v) \rightarrow \forall i.[\texttt{contains}(i, v) \leftrightarrow \texttt{contains}(i, s)] \tag{8.88}$$

Eine gefüllte Variable wirkt wie eine einelementige Menge, die nur den Füllwert enthält:

$$\texttt{variable}(v, s) \wedge \texttt{filled}(v, j) \rightarrow \forall i.[\texttt{contains}(i, v) \leftrightarrow \texttt{equals}(i, j)] \tag{8.89}$$

Eine verbundene Variable wirkt wie die Variable, mit der sie verbunden ist:

$$\texttt{variable}(v, s) \wedge \texttt{connected}(v, w) \rightarrow \forall i.[\texttt{contains}(i, v) \leftrightarrow \texttt{contains}(i, w)] \quad (8.90)$$

8.6. Operatoren

Operationen werden durch die beiden zweistelligen Attribute `precondition` und `effect` gekennzeichnet. Der erste Parameter gibt die Zugehörigkeit zu einer Instanz an, der zweite Parameter den Operanden als Menge. Die Semantik ist für die `precondition`- und `effect`-Elemente gemeinsam definiert, die über eine gemeinsame Instanz x verbunden sind. Ob die Operation angestoßen wurde, wird durch das einstellige, zeitlich variable Prädikat `invoked` ausgedrückt. $\texttt{invoked}(X)$ überprüft, ob die Operation an der gemeinsamen Instanz X gestartet wurde. Ist dies der Fall und sind alle Vorbedingungen an x erfüllt sind, so entstehen zu einem zukünftigen Zeitpunkt (ausgedrückt durch den temporalen Operator $\Diamond$) neue reale Instanzen durch alle Effekte an x:

$$\begin{aligned} \texttt{invoked}(x) \wedge \forall s.[\texttt{precondition}(x, s) \rightarrow \exists i.(\texttt{contains}(i, s) \wedge \texttt{real}(i))] \quad (8.91) \\ \rightarrow \Diamond \forall t.[\texttt{effect}(x, t) \rightarrow \exists j.(\texttt{contains}(j, t) \wedge \texttt{real}(j))] \end{aligned}$$

8.7. Dienstbeschreibungen

Dienstbeschreibungen werden aus Elementen von DE-I und -II zusammengesetzt. Sie bestehen aus

- einem *Kopf* bestehend aus Informationen über den Dienstnehmer bzw. Dienstgeber und weiterer nicht-funktionaler Aspekte des Dienstes.
- einem *Profil* bestehend aus Effekten E als new-Operationen auf der Ontologie sowie Vorbedingungen P als exists-Tests auf der Ontologie.
- einem *Grounding* zur Registrierung beim Agenten.

E und P stellen jeweils Reihen von DE-II-Mengen dar, in deren Attributbedingungen auch Variablen auftreten können. E und P sind daher Reihen konfigurierbarer Mengen.

Dienstbeschreibungen werden durch weitere Individualkonstanten ausgezeichnet. Es existiert ein einstelliges Prädikat `servicedescription`. $\texttt{servicedescription}(D)$ besagt, dass D eine Dienstbeschreibung darstellt. Dienstbeschreibungen werden in Angebots- und Anfragebeschreibungen unterschieden. Dazu existieren zwei einstellige Prädikate `offer` und `request`. $\texttt{offer}(O)$ besagt, dass es sich bei O um eine Dienstangebotsbeschreibung handelt; $\texttt{request}(R)$ besagt, dass es sich bei R um eine Dienstanfragebeschreibung handelt. Es gilt:

$$\texttt{offer}(d) \rightarrow \texttt{servicedescription}(d) \tag{8.92}$$

$$\texttt{request}(d) \rightarrow \texttt{servicedescription}(d) \tag{8.93}$$

$$\texttt{servicedescription}(d) \rightarrow [\texttt{offer}(d) \leftrightarrow \neg\texttt{request}(d)] \tag{8.94}$$

Variablen, die in Dienstbeschreibungen auftreten, besitzen eine erweiterte Semantik, die festlegt, wann und von wem diese gebunden werden müssen. Es existieren dazu vier **Kategorien von Variablen**: IN_x, OUT_x, IN_e und OUT_e. Variablen können auch zu mehreren Kategorien gehören. Diese werden durch vier Prädikate unterschieden:

- $\texttt{in\_x\_variable}(V)$ besagt, dass die Variable V eine IN_x-Variable ist.
- $\texttt{out\_x\_variable}(V)$ besagt, dass die Variable V eine OUT_x-Variable ist.
- $\texttt{in\_e\_variable}(V, STEP)$ besagt, dass die Variable V eine $\mathsf{IN}_{e,STEP}$-Variable ist.
- $\texttt{out\_e\_variable}(V, STEP)$ besagt, dass die Variable V eine $\mathsf{OUT}_{e,STEP}$-Variable ist.

Wichtig für die Semantik von Dienstbeschreibungen ist die Überprüfung, ob alle Variablen einer Kategorie gefüllt sind. Für IN_x-Variablen existiert dazu das Prädikat `all_in_x_filled`. Es ist genau dann für die Menge S wahr, wenn keine direkt oder indirekt über eine Attributbedingung von S erreichbare IN_x-Variable ungefüllt ist:

$$\begin{aligned}\forall s.\texttt{all\_in\_x\_filled}(s) \leftrightarrow{}& \\ [\texttt{in\_x\_variable}(s) \rightarrow \exists j.\texttt{filled}(x, j)] \wedge{}& \\ [\forall pc.\forall a.\forall t.\texttt{flatcs}(s, pc) \wedge \texttt{propertycondition}(pc, a, t) \rightarrow \texttt{all\_in\_x\_filled}(t)]&\end{aligned} \tag{8.95}$$

Das Prädikat ist zeitlich veränderlich, da es sich vom zeitlich veränderlichen Prädikat `filled` ableitet. Die Prädikate für die drei anderen Variablenkategorien werden analog definiert.

Dienstbeschreibungen sind nicht statisch, sondern durchlaufen einen **Lebenszyklus**. Daher ist die Semantik einer Beschreibung zeitabhängig, was durch die Verwendung von temporalen Operatoren ausgedrückt werden kann.

8.7.1. Dienstangebotsbeschreibungen

Die **Semantik einer Dienstangebotsbeschreibung** O (d.h. $\texttt{offer}(O) \equiv \mathit{true}$) kann dann als Vertrag aufgefasst werden, der aus drei Teilen besteht: einem *Schätzungsteil*, der festlegt, welche weiteren Details über den Vertrag angefordert werden können, einem *Bedingungsteil*, der festlegt, welche Anforderungen erfüllt sein müssen, damit der Dienst ausgeführt werden kann, sowie einem *Leistungsteil*, der festlegt, welche Leistungen der Dienst bei erfolgreicher Ausführung erbringt.

Der **Schätzungsteil** erlaubt es dem Dienstnehmer, vor der Dienstausführung weitere Details über die Dienst zu erfahren.

V1. [Vorbereitung der Schätzung]
In O müssen alle IN_e-Variablen mit der gleichen Schrittnummer $STEP$ gefüllt sein, damit der Dienstgeber die Schätzung mit der Schrittnummer $STEP$ korrekt durchführen kann. Daher wird der Wahrheitsgehalt des folgenden Ausdrucks fortlaufend ermittelt und in est_{STEP} abgelegt.

$$\begin{aligned} \mathrm{est}_{STEP} :\equiv \quad & (8.96) \\ \forall s.[\texttt{effect}(O,s) \vee \texttt{precondition}(O,s) \rightarrow \texttt{all\_in\_e\_filled}(s, STEP)] & \end{aligned}$$

V2. [Durchführung der Schätzung]
Ist est_{STEP} erfüllt, kann die Schätzung mit der Schrittnummer $STEP$ angestoßen werden, was durch das zweistellige, zeitlich veränderliche Prädikat `e_invoked` ausgedrückt wird. $\texttt{e\_invoked}(O, STEP)$ drückt aus, dass der angegebene Schätzschritt der Beschreibung O aktiviert wird. Der Dienstnehmer führt die Schätzung dann bis zu einem der nächsten Zeitpunkte durch, indem er die OUT_e-Variablen füllt, welche die Schrittnummer $STEP$ tragen. Dadurch werden keine Effekte erwirkt. Es gilt daher:

$$\begin{aligned} & \mathrm{est}_{STEP} \wedge \texttt{e\_invoked}(O, STEP) \\ \rightarrow \Diamond \forall s.[& \texttt{effect}(O,s) \vee \texttt{precondition}(O,s) \\ & \rightarrow \texttt{all\_out\_e\_filled}(s, STEP)] \end{aligned} \tag{8.97}$$

Der **Bedingungsteil** gliedert sich in Informationsbedingungen und Zustandsbedingungen. Es gilt:

B1. [Informationsbedingung]
In O müssen alle IN_x-Variablen gefüllt sein, damit der Dienst seine Leistungen korrekt erbringen kann. Der zeitlich veränderliche Wahrheitsgehalt des folgenden Ausdrucks wird daher fortlaufend ermittelt und in *xinfo* abgelegt:

$$\text{xinfo} :\equiv \forall s.[\texttt{effect}(O,s) \vee \texttt{precondition}(O,s) \rightarrow \texttt{all\_in\_x\_filled}(s)] \quad (8.98)$$

B2. [Zustandsbedingung]
Weiterhin muss jede Menge, die von O's Vorbedingungen referenzierten wird, eine reale Instanz enthalten, damit der Dienst seine Leistungen korrekt erbringt kann. Der Wahrheitsgehalt des folgenden Ausdrucks wird daher fortlaufend ermittelt und in *xstate* abgelegt:

$$\text{xstate} :\equiv \forall s.[\texttt{precondition}(O,s) \rightarrow \exists i.(\texttt{contains}(i,s) \wedge \texttt{real}(i))] \quad (8.99)$$

Der **Leistungsteil** umfasst Informationsleistungen und Zustandsleistungen. Sind sowohl *xinfo* als auch *xstate* erfüllt, so kann der Dienstnehmer die Ausführung des Dienstes explizit anstoßen, was durch das einstellige, zeitlich veränderliche Prädikat `x_invoked` ausgedrückt wird. $\texttt{e\_invoked}(O)$ drückt aus, dass die Ausführung des Dienstes mit der Beschreibung O aktiviert wird. Nach der Anstoßung existieren zwei Möglichkeiten: (1) Der Dienstgeber kann die Ausführung des Dienstes ablehnen (z.B. im Falle einer Unterspezifizierung von O, siehe Abschnitt 7.3.2) und erbringt *keine* der Leistungen; oder (2) der Dienstgeber erbringt *alle* der angegebenen Leistungen. In diesem Fall gilt:

L1. [Informationsleistung]
Bis zu einem zukünftigen Zeitpunkt sind alle OUT_x-Variablen in O gefüllt.

$$\begin{aligned} &\text{xinfo} \wedge \text{xstate} \wedge \texttt{x\_invoke}(O) \quad (8.100)\\ \rightarrow \Diamond\, &\forall s.[\texttt{effect}(O,s) \vee \texttt{precondition}(O,s) \rightarrow \texttt{all\_out\_x\_filled}(s)] \end{aligned}$$

L2. [Zustandsleistung]
Zudem hat der Dienstgeber zu einem zukünftigen Zeitpunkt für jede Menge, die in O's Effekten referenziert wird, eine reale Instanz geschaffen, welche zur Menge gehört.

$$\begin{aligned} &\text{xinfo} \wedge \text{xstate} \wedge \texttt{x\_invoke}(O) \quad (8.101)\\ \rightarrow \Diamond\, &\forall s.[\texttt{effect}(O,s) \rightarrow \exists i.(\texttt{real}(i) \wedge \texttt{contains}(i,s))] \end{aligned}$$

8.7.2. Dienstanfragebeschreibungen

Die **Semantik einer Dienstanfragebeschreibung** R (d.h. $\texttt{request}(R) \equiv \mathit{true}$) kann als Wunsch aufgefasst werden, der aus drei Teilen besteht: einer *Spezialisierungsmöglichkeit*, mit der die allgemeine Anfrage für den konkreten Zweck angepasst werden kann, einem *Zustandswunsch*, der angibt, an welchem neuen Zustand der Dienstnehmer interessiert ist, sowie einem *Informationswunsch*, der angibt, welche Informationen der Dienstnehmer nach Ausführung eines Dienstes benötigt.

Dienstanfragebeschreibungen enthalten keine Vorbedingungen:

$$\forall r.\texttt{request}(r) \rightarrow \neg\exists s.\texttt{precondition}(r, s) \tag{8.102}$$

Die **Spezialisierungsmöglichkeiten** erlauben dem Dienstnehmer, die Anfragebeschreibung vor der Versendung zum Zeitpunkt t_{ra} zu konkretisieren.

S1. [Spezialisierung]
Um die Anfrage R absenden zu können, müssen in R alle IN_x-Variablen gefüllt sein. Dies wird fortlaufend überprüft und in *rinfo* festgehalten:

$$\text{rinfo} :\equiv \forall s.[\texttt{effect}(R, s) \rightarrow \texttt{all\_in\_x\_filled}(s)] \tag{8.103}$$

Der **Zustandswunsch** gibt an, welche Zustände vom Dienstnehmer angestrebt werden, nachdem die Anfrage angestoßen wurde. Dies wird durch das zeitlich veränderliche Prädikat `r_invoked` ausgedrückt wird. $\texttt{r\_invoked}(R)$ drückt aus, dass die Anfrage R vom Dienstnehmer abgesandt wurde.

W1. [Zustandswunsch]
Mit dem Absenden der Anfrage drückt der Dienstnehmer einen Wunsch aus. Zu einem kommenden Zeitpunkt *soll* für jede Menge, die in R's Effekten referenziert wird, je eine reale Instanz erzeugt werden, die zur Menge gehört. Ausgedrückt wird das durch:

$$\begin{aligned} &\text{rinfo} \wedge \texttt{r\_invoked}(R) \\ \rightarrow\ &\triangleright\Diamond\forall s.[\texttt{effect}(R, s) \rightarrow \exists i.(\texttt{contains}(i, s) \wedge \texttt{real}(i))] \end{aligned} \tag{8.104}$$

Zudem drückt der Dienstnehmer einen **Informationswunsch** aus, indem er angibt, welche Informationen vom Dienstnehmer benötigt werden, d.h. welche Daten nach der Dienstausführung bekannt sein sollen.

W2. [Informationswunsch]
Zu einem kommenden Zeitpunkt *soll* jede OUT_x-Variable der Anfragebeschreibung gefüllt sein. Ausgedrückt wird das durch:

$$\begin{aligned} &\text{rinfo} \wedge \texttt{r\_invoked}(R) \\ \rightarrow \triangleright\Diamond\forall s.[&\texttt{effect}(R, s) \rightarrow \texttt{all\_out\_x\_filled}(s)] \end{aligned} \tag{8.105}$$

Der Vergleicher bzw. die Middleware hat die Aufgabe, einen Dienst zu finden und geeignet zu konfigurieren, sodass diese Wünsche erfüllt werden.

8.8. Zusammenfassung

In diesem Kapitel wurde die Semantik von DE-I, DE-II und DSD formal definiert. Als Technik diente eine Abbildung auf eine existierende Sprache, für die bereits eine formale Semantik hinterlegt ist, in diesem Falle die modale, temporale Prädikatenlogik 1. Ordnung mit Identität. Es entstand die *axiomatische Semantik* für DE/DSD. Wichtig war für DE-I insbesondere die Formalisierung der Metaeigenschaften, für DE-II die Zugehörigkeit zu einer Menge (ausgedrückt durch `contains`) und für DSD die Auffassung von Dienstbeschreibungen als Verträge bzw. Wünsche mit entsprechenden Bedingungen und Leistungen.

9. Vergleich von Dienstbeschreibungen

Der Vergleicher für semantische Dienstbeschreibungen stellt die wichtigste Komponente einer Architektur zur Automatisierung der Dienstnutzung dar. Tatsächlich ist eine semantische Dienstbeschreibungssprache nur dann sinnvoll und nützlich, wenn ein Vergleichsalgorithmus zur Verfügung steht, der konkrete Beschreibungen in dieser Sprache korrekt und effizient verarbeiten kann. In diesem Kapitel wird ein solcher Vergleicher für DSD vorgestellt.

DSD zeichnet sich dadurch aus, dass zwischen Beschreibungen für angebotene und benötigte Dienste unterschieden wird. Als Besonderheit sind Dienstanfragebeschreibungen durch Verwendung von unscharfen, deklarativen Mengen *präferenzbeinhaltend*, d.h. in ihnen ist eine Bewertung der Wunscheffekte bereits enthalten. Aufgabe eines Vergleichers für DSD ist daher *nicht* wie in der Literatur üblich die Berechnung der Ähnlichkeit zwischen der Anfrage- und einer Angebotsbeschreibung, sondern die Bestimmung der persönlichen Präferenz des Dienstnehmers für ein gegebenes Angebot allein aus den Angaben der Anfragebeschreibung. Dieser Wert wird als *Vergleichswert* bezeichnet. Ein solcher Vergleich ist damit ausschließlich durch die hinterlegte Semantik der Dienstbeschreibungen definiert und daher unverzerrt, da er keine dem Dienstnehmer unbekannten Heuristiken zur Auswahl des Dienstgebers verwendet. Auf diese Weise entsteht für jede Anfrage ein *persönlicher Vergleicher* [79, 81], der sich von in den in der Literatur eingesetzten generischen Allzweckvergleichern erheblich unterscheidet. Vorteil eines solchen persönlichen Vergleichers ist, dass die von ihm berechneten Vergleichsergebnisse vom Dienstnehmer tatsächlich ohne Rückfragen akzeptiert werden können, da sie exakt den Vorstellungen entsprechen.

Ein weiterer Unterschied des Vergleichers für DSD ist die kombinierte Überprüfung von Signatur und Spezifikation [80]. Ein Dienstangebot wird nicht allein deshalb verworfen, weil seine ein- und ausgehenden Informationen nicht mit denen in der Anfrage übereinstimmen. Die fehlenden Informationen können unter Umständen aus den beteiligten Zuständen abgeleitet werden. Ebenso wird ein Angebot nicht verworfen, wenn die erwirkbaren Zustande nicht direkt zur Anfrage passen – denkbar ist, dass sie über eine entsprechende Konfiguration der Eingabevariablen eingestellt werden können.

Im folgenden Kapitel wird die Umsetzung des Vergleichers präsentiert, welche auch als Java-Implementierung vorliegt. In Abschnitt 9.1 wird zunächst der Vergleich von Effekten vorgestellt; in Abschnitt 9.2 folgt dann die Beschreibung, wie Vorbedingungen überprüft werden. In Abschnitt 9.3 wird auf die Trennung von Vor- und Hauptvergleich eingegangen. In Abschnitt 9.4 wird das Vorgehen des Vergleichers an einem Gesamtbeispiel vorgeführt. Abschnitt 9.5 fasst abschließend den Vergleichsvorgang zusammen.

9.1. Vergleich von Effekten

Im letzten Kapitel wurde deutlich, dass ein angebotener Dienst durch eine konfigurierbare Menge von erzielbaren Zuständen beschrieben werden kann, während ein benötigter Dienst durch eine unscharfe Menge von gewünschten Zuständen erfasst wird. Zunächst wird daher der Fall betrachtet, dass Dienstbeschreibungen genau einen <<effect>>-Operator enthalten. Mehrerer Effekte und Vorbedingungen werden zum besseren Verständnis erst später berücksichtigt. Intuitiv passen eine Dienstangebots- und eine -anfragebeschreibung dann zusammen, wenn die Menge im Angebot durch Konfiguration so eingeschränkt werden kann, dass sie zu einer Teilmenge der Anfragemenge wird. In diesem Fall ist der Effekt, der durch die Dienstausführung erwirkt werden kann, auch im ungünstigsten Fall vom Dienstnehmer gewünscht.

Der Vergleicher hat daher neben der Berechnung des Vergleichswerts für DSD noch weitere Aufgaben: (1) Er muss den ausgewählten Dienstgeber für die Ausführung geeignet konfigurieren, d.h. er muss die benötigten Informationen bereitstellen, indem die entsprechenden Variablen gebunden werden; (2) er muss berechnen, ob und wie die vom Dienstnehmer gewünschten Informationen über die Effekte bereitgestellt werden können. Als Ergebnis eines Vergleichs zwischen einer Anfrage- und einer Angebotsbeschreibung entsteht somit ein zusammengesetztes *Vergleichsergebnis* (engl. matching result, MR) mit folgenden Teilen:

- Die *Konfiguration der Eingabevariablen* j legt fest, wie die IN-Variablen der Angebotsbeschreibung zu belegen sind, wenn der Dienst aufgerufen wird.
- Der *Vergleichswert* mv (für engl. matching value) aus dem Intervall $[0, 1]$ repräsentiert eine untere Schranke für die *Präferenz* des Dienstnehmers für den mit j konfigurierten Dienst.
- Die *Belegung der Ausgabevariablen* k legt fest, wie die OUT-Variablen der Anfragebeschreibung zu belegen sind, nachdem der Dienst ausgeführt wurde.

Um eine exakte Beschreibung dieser Aufgabe zu bekommen, wird für Mengen und Variablen folgender Formalismus eingeführt: Sei $\mathcal{S}$ die Menge aller scharfen, variablenfreien (d.h. nicht-konfigurierbaren) Mengen, die mittels der Sprachelemente aus DE-II definiert werden können. Eine Menge ist *variablenfrei*, wenn zu ihrer Definition keine IN-Variablen verwendet wurden. $\tilde{\mathcal{S}}$ dagegen sei die Menge aller unscharfen, variablenfreien Mengen aus DE-II. Es gilt dann natürlich, dass $\mathcal{S} \subset \tilde{\mathcal{S}}$. Für jede Menge $s \in \tilde{\mathcal{S}}$ ist eine *charakteristische Funktion* definiert

$$\chi_s : I \longrightarrow [0, 1] \tag{9.1}$$

welche jeder Instanz i aus der Menge aller Instanzen I einen Zugehörigkeitswert zur Menge s zuweist. I enthält alle Literale, benamte und anonyme Instanzen der Grundgesamtheit der Menge. Diese Definition erlaubt es, eine Teilmengenbeziehung `subset` zwischen einer scharfen Mengen $s_1 \in \mathcal{S}$ und einer unscharfen Menge $s_2 \in \tilde{\mathcal{S}}$ zu definieren. `subset` ist durch das Element i aus s_1 definiert, das am schlechtesten zu s_2 passt:

$$\texttt{subset}(s_1, s_2) = \min_{i \in s_1} \chi_{s_2}(i) \tag{9.2}$$

Typischerweise sind Mengen in Angebotsbeschreibungen jedoch nicht variablenfrei, sondern sind über die Verwendung von OffIN-Variablen konfigurierbar. Wir definieren daher $\mathcal{S}^n$ als die Menge aller scharfen Mengen aus DE-II, die mittels n IN-Variablen konfiguriert werden können. Eine solche Menge $s \in \mathcal{S}^n$ kann also durch ein Instanztupel $j \in I^n$ konfiguriert werden. Die konfigurierte Menge wird dann als $s[j]$ notiert und ist wieder Element aus $\mathcal{S}$.

Diese Definitionen helfen uns, die Aufgabe des Vergleichers formal zu fassen. Sei $o \in \mathcal{S}^n$ die konfigurierbare Effektmenge der Dienstangebotsbeschreibung, die n OffIN-Variablen enthält, und $r \in \tilde{\mathcal{S}}$ die unscharfe, variablenfreie Effektmenge der Anfragebeschreibung.[1] Der Vergleichswert mv zwischen der Anfragebeschreibung r und der mit dem Instanztupel j konfigurierten Angebotsbeschreibung o ist dann wie folgt definiert:

$$mv(o, r, j) \;:=\; \texttt{subset}(o[j], r) \tag{9.3}$$

[1] Die Anfragebeschreibung kann zum Zeitpunkt des Vergleichs als variablenfrei angesehen werden, da ihre ReqIN-Variablen bereits vor Absenden der Anfrage vom Dienstnehmer gefüllt wurden und wie einelementige Mengen wirken (vgl. den Ablauf der Dienstnutzung in Abbildung 7.11 auf Seite 178).

Es ist ersichtlich, dass es nicht *den* Vergleichswert zwischen einer Anfrage- und einer Angebotsbeschreibung gibt, sondern dass dieser von der Konfiguration der Eingabevariablen abhängt. Der optimale Vergleichswert mv_{opt} ist durch die Variablenkonfiguration bestimmt, die zum größten Wert für die Teilmengeneigenschaft führt:

$$mv_{opt}(o, r) := \max_{j \in I^n} \texttt{subset}(o[j], r) \tag{9.4}$$

Mit der Definition von `subset` von oben gilt also:

$$mv_{opt}(o, r) = \max_{j \in I^n} \min_{i \in o[j]} \chi_r(i) \tag{9.5}$$

Die optimale Konfiguration der OffIN-Variablen j_{opt} kann dann mithilfe der argmax-Funktion bestimmt werden:

$$j_{opt}(o, r) = \arg\max_{j \in I^n} \min_{i \in o[j]} \chi_r(i) \tag{9.6}$$

Die Formeln 9.5 und 9.6 liefern auch einen theoretischen Ansatz, wie der Vergleichswert und die optimale Konfiguration mittels einer geschachtelten Schleife berechnet werden kann. Die äußere Schleife enumeriert dabei alle gültigen Variablenkonfigurationen j, während die innere Schleife alle Elemente i der Menge $o[j]$ auflistet. Für jede dieser Instanzen i wird der Zugehörigkeitswert zu r bestimmt. In der Praxis ist dieses Vorgehen nur teilweise durchführbar, da es nicht immer möglich ist, die Elemente von o zu enumerieren. Es muss daher auf eine Berechnung der `subset`-Operation zurückgeführt werden, bei der die Symbole zur Definition der Mengen ausgewertet werden. Dies ist möglich, da die Sprachelemente aus DE-II zur Definition von konfigurierbaren Mengen so entwickelt wurden, dass sie aufgrund von Orthogonalitäts- und Monotonieeigenschaften eine schrittweise, rekursive Berechnung von `subset` zulassen.

Der Vergleichsalgorithmus für Effekte wird in drei Schritten vorgestellt. Zunächst wird in Abschnitt 9.1.1 vereinfachend angenommen, dass die zu vergleichenden Angebotsbeschreibungen variablenfrei ist, und erklärt, wie `subset` in diesem Fall implementiert werden kann. Anschließend wird in Abschnitt 9.1.2 gezeigt, wie eine variablenbehaftete Beschreibung durch optimale Füllung der OffIN-Variablen in eine variablenfreie Beschreibung umgewandelt werden kann. Im letzten Schritt in Abschnitt 9.1.3 wird erklärt, wie abschließend die ReqOUT-Variablen gebunden werden können. Die technischen Details des Vergleichs finden sich in [105].

9.1.1. Vergleich variablenfreier Mengen

In diesem Abschnitt wird zunächst angenommen, dass die zu vergleichenden Mengen o und r variablenfrei sind, d.h. bei der Berechnung des Vergleichswertes nach Formel 9.5 entfällt die Konfigurierung der Angebotsmenge o. Der Vergleichswert kann in diesem Fall durch Berechnung von `subset` bestimmt werden:

$$mv_{opt}(o, r) = \texttt{subset}(o, r), \quad \text{falls } o \text{ variablenfrei} \tag{9.7}$$

Die Berechnung von `subset` hängt entscheidend davon ab, ob die Elemente von o enumeriert werden können. In einem solchen Fall ist eine direkte *iterative Berechnung* nach der Grundformel 9.2 möglich. Ist eine Auflistung der Elemente von o nicht möglich, so muss auf eine symbolische *rekursive Berechnung* zurückgegriffen werden.

Die Berechnung von $\texttt{subset}(o, r)$ hängt daher von der Beschaffenheit und dem Kenntnisstand über die Menge o ab. Da der Vergleich auf der Seite des Klienten durchgeführt wird, sind an Instanzen nur die zentralen Kopien aus dem öffentlichen Pool und die lokalen Kopien aus dem privaten Pool des Dienstnehmers bekannt. Die lokalen Kopien des Dienstgebers stehen nicht zur Verfügung. Man unterscheidet daher drei Fälle für die Menge o:

- **Stufe 0: o ist vollständig**: Jede fiktive Instanz, die die Bedingungen der Menge o erfüllt, ist auch Element von o. Die Elemente brauchen dann nicht aufgelistet werden zu können, da die Elemente über ihre Attributbedingungen aufgespannt werden. o wird dann auch kurz als *S0-Menge* bezeichnet. Dies gilt in den Fällen:
  - o ist eine Prim-Menge[2].
  - o ist eine V-Menge.
  - o ist eine E- oder PE-Menge, die von einem effect-Operator der Angebotsbeschreibung referenziert wird. Der Dienstnehmer erstellt dann eine neue Instanz, die anschließend zur Menge o gehört.
- **Stufe 1: o ist unvollständig, aber vollzählig enumerierbar**: Nicht jede fiktive Instanz, die die Bedingungen der Menge o erfüllt, ist auch Element von o, jedoch können o's Elemente vollzählig enumeriert werden. o wird dann auch kurz als *S1-Menge* bezeichnet. Dies gilt in folgenden Fällen:
  - o ist eine PE-Menge ohne Referenz von einem effect-Operator. Alle Elemente von o können dann dem zentralen Pool entnommen werden.

[2]d.h. eine Menge mit Elementen eines primitiven Typs, vergleiche Abschnitt 6.1.1.

  - o ist eine E-Menge mit der Markierung `|pubpool|` oder `|reqpool|`. In dem Fall sind die Elemente des Pools bekannt und keine weiteren, unbekannten Elemente können in o enthalten sein.
  - o ist eine E-Menge mit der Markierung `|offpool|` und mit einer direkten Bedingung mit dem Operator `==` oder `in`. Die Element von o können dann dieser Bedingung entnommen werden.
  - o ist eine E-Menge mit der Markierung `|offpool|`, deren Elemente in einem vorhergehenden Schritt durch Auswertung weiterer Informationen bestimmt werden können. Beispielsweise kann ein weiterer Wissensdienst voran geschaltet werden.

- **Stufe 2: o ist nicht vollständig und nicht vollzählig enumerierbar**: Die Elemente von o können nicht oder nur teilweise enumeriert werden, obwohl o nicht vollständig ist, d.h. es fiktive Instanzen gibt, die zwar alle Bedingungen der Menge erfüllen, jedoch trotzdem kein Element von ihr sind. o wird dann auch kurz als *S2-Menge* bezeichnet. Dies gilt im Fall:
  - o ist eine E-Menge mit der Markierung `|offpool|` ohne direkte Bedingung mit dem Operator `==` oder `in`, für deren Elemente kein Wissensdienst zur Verfügung steht.

Zwar sind S1-Mengen so definiert, dass es dem Klienten möglich ist, ihre Elemente vollzählig aufzulisten, jedoch ist nicht garantiert, dass die einzelnen aufgelisteten Elemente dem Klienten in allem Füllwerten bekannt sind. Beispielsweise könnte bei einer PE-Menge mit der Markierung `|offpool|` der Dienstgeber von einigen Elementen lokale Kopien mit erweiterten Füllwerten angelegt haben, die dem Klienten unbekannt sind. Im Folgenden wird daher zunächst angenommen, dass die Füllwerte stets vollständig bekannt sind. In Abschnitt 9.1.4 wird dann beschrieben, wie sich der Vergleich durch unbekannte Füllwerte ändert.

Die Berechnung von `subset`(o, r) unterscheidet sich für verschiedene Stufen der Menge o. Ist o eine S0-Menge wird Rekursion verwendet, ist o eine S1-Menge wird eine Iteration über die Elemente verwendet, ist o eine S2-Menge steht für eine exakte Berechnung zu wenig Information zur Verfügung. Hier erfolgt eine Abschätzung der unteren Grenze durch Rekursion. Die genauen Verfahren werden im Folgenden vorgestellt.

Subset für eine Menge der Stufe 0

Die Berechnung von `subset`(o, r) für eine Menge o der Stufe 0 kann nicht iterativ erfolgen, da die Instanzen von o nicht aufgelistet werden können. Da die Menge jedoch

vollständig ist, d.h. die Attributbedingungen orthogonal sind, kann die Teilmengeneigenschaft *symbolisch* durch Überprüfung der einzelnen Bedingungen von o und r überprüft werden. Dies übernehmen drei Funktionen $\texttt{test}_{tc}$, welche die Typbedingung und Typvergleichsstrategien überprüft, $\texttt{test}_{dc}$, welche die direkten Bedingungen untersucht sowie $\texttt{test}_{cs}$, welche die durch Verbindungsstrategien verknüpften Attributbedingungen vergleicht. Ihre jeweilige Berechnung wird im Folgenden vorgestellt.

Typbedingungen. $\texttt{test}_{tc}(o, r)$ überprüft die Typbedingungen (inklusive Typvergleichsstrategien) von o und r. Ausschlaggebend ist der maximale Typ t, der in o noch zulässig ist. Für ihn wird der Zugehörigkeitswert in r überprüft. Für $\texttt{test}_{tc}$ entsteht so ein Wert aus [0,1]: 1, wenn der t zu r passt, 0, sonst. Zwischenwerte können durch unscharfe Typvergleichsstrategien in r entstehen.

Direkte Bedingungen. $\texttt{test}_{dc}(o, r)$ überprüft die direkten Bedingungen von o und r. Hierzu wird zunächst die gesamte direkte Bedingung aus o als Zugehörigkeitsfunktion $b_o : I \to \{0, 1\}$ und die gesamte direkte Bedingung aus r als Zugehörigkeitsfunktion $b_r : I \to [0, 1]$ aufgefasst. Da die Teilmengeneigenschaft $o \subset r$ überprüft werden soll, kann $\texttt{test}_{dc}$ wie folgt berechnet werden:

$$\texttt{test}_{dc}(o, r) = \forall i : b_o(i) \to b_r(i) \tag{9.8}$$

Die Berechnung der Implikation im Falle unscharfer Werte kann durch folgende Umformung im scharfen Fall abgeleitet werden. Im scharfen Fall gilt:

$$a \to b \equiv \neg a \vee b \equiv \neg(a \wedge \neg b) \tag{9.9}$$

Formel 9.8 kann also wie folgt berechnet werden:

$$\texttt{test}_{dc}(o, r) = \min_i \; b(i) \;\; \text{mit} \;\; b(i) := 1 - [b_o(i) \cdot (1 - b_r(i))] \tag{9.10}$$

Die Berechnung hängt davon ab, wie der genaue Typ der Menge aussieht. Ist o ein **primitiver Typ**, so kann sein Wertebereich auf einen Zahlenwert umgerechnet werden (siehe dazu [105]). In diesem Fall kann das Minimum mit Hilfe der Differenzialrechnung bestimmt werden. Als notwendige Bedingung für die Existenz eines globalen Minimums gilt, dass es als Extremwert der Funktion b immer an einer Stelle der Ableitung b' zu finden ist, die 0 oder undefiniert ist.[3]. Solche Stellen entstehen durch die Schnittpunkte und singulären Punkte der einzelnen Bedingungen aus o und r.

[3] vgl. Bronstein et al. [22], Abschnitt 6.1.6

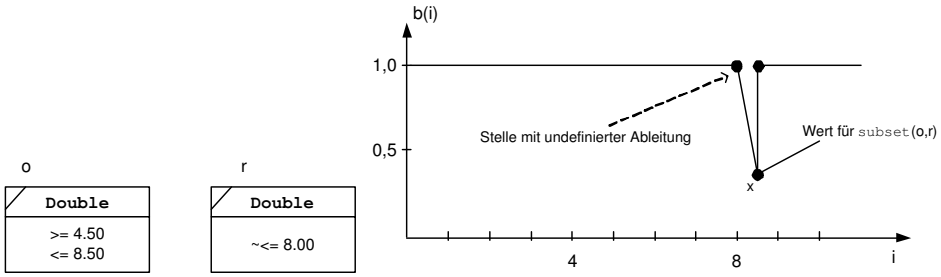


Abbildung 9.1.: Berechnung von $\texttt{test}_{dc}$ für Mengen primitiver Typen.

Einerseits existiert eine solche Stelle immer[4], andererseits kann b' nur endlich viele solche Stellen besitzen. Das globale Minimum kann daher durch Berechnung der Funktionswerte von b auf diesen Stellen gefunden werden.

Abbildung 9.1 zeigt zwei Mengen vom primitiven numerischen Typ `Double`. In r verlangt der Dienstnehmer einen Wert von 8.0 oder kleiner, akzeptiert durch die unscharfe Bedingungen jedoch auch Werte darüber, allerdings mit entsprechend geringerer Präferenz. In o bietet der Dienstgeber Werte zwischen 4.5 und 8.5 an, ohne sich im Voraus auf einen festzulegen. Die Abbildung zeigt die resultierende Funktion b mit den in Frage kommenden Stellen, an denen ihre Ableitung unstetig ist. Punkt x stellt mit 0.375 den niedrigsten Wert und damit den Wert von $\texttt{test}_{dc}(o, r)$ dar.

Ist o hingegen ein **Entitätstyp**, so können nur direkte Bedingungen mit den positiven Vergleichsoperatoren `==` oder `in` sowie dem negativen Operator `!=` auftreten. Im Falle eines positiven Operators in o kann die Überprüfung für die angegebenen Instanzen iterativ wie für S1-Mengen erfolgen, indem deren Zugehörigkeitswerte zu r sukzessive ermittelt werden. Eine weitere Betrachtung der Attributbedingung ist dann nicht mehr nötig. Im Falle eines negativen Operators in r muss sichergestellt werden, dass die aufgelisteten Elemente nicht Teil von o sind. Auch das kann durch iteratives Überprüfen der Elemente erfolgen. Anschließend ist noch eine weitere Betrachtung der Attributbedingungen nötig.

Abbildung 9.2 zeigt zwei Mengen o und r des Entitätstyps Company: Der Dienstgeber liefert eine deutsche Firma außer Siemens oder der Deutschen Telekom, der Dienstnehmer wünscht eine deutsche oder französische Firma, jedoch nicht Siemens oder France Telecom. Da hier r direkte Bedingungen mit negativem Operator enthält, wird überprüft, ob die zugehörigen Operanden siemens und franceTelecom in o liegen, was nicht der Fall ist. $\texttt{test}_{dc}(o, r)$ liefert demnach 1.0. Die Überprüfung der Attributbedingung muss jedoch noch erfolgen.

[4]Direkte Bedingungen setzen sich nur aus linearen Funktionen mit der Wertebereich [0,1] zusammen. Daher ist die einzige Ableitungsfunktion ohne Unstetigkeitsstellen die Nullfunktion $b' \equiv 0$, da sonst b den Wertebereich [0,1] verlassen müsste. In diesem Fall ist b konstant ($b \equiv c$), wobei c das globale Minimum darstellt.

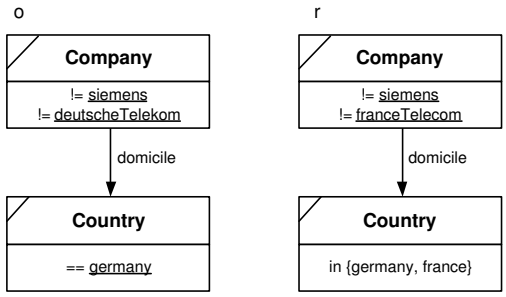


Abbildung 9.2.: Berechnung von $\texttt{test}_{dc}$ für Mengen von Entitätstypen.

Attributbedingungen. Da die Attributbedingungen von o und r über veränderte Verbindungsstrategien verbunden sein können, steht die Berechnung von $\texttt{test}_{cs}(o, r)$ im Mittelpunkt. Seien cs_o und cs_r die Verbindungsstrategien von o und r, welche die Ergebnisse der einzelnen Attributbedingungen für eine Instanz i verrechnen. Wenn o eine Teilmenge von r sein soll, muss gelten

$$\forall i \in I : cs_o(i) \rightarrow cs_r(i) \tag{9.11}$$

Hierbei enthält cs_o zum Beispiel eine Attributbedingung a_o aus o, während cs_r die zugehörige Attributbedingung a_r aus r enthält. Jede Attributbedingung kann als Funktion $a_o : I \rightarrow [0, 1]$ bzw. $a_r : I \rightarrow [0, 1]$ aufgefasst werden, die für eine Instanz berechnet, in wie weit diese die Attributbedingung erfüllt.

Als Rechenvorschrift gilt (mit der Begründung aus Formel 9.9):

$$cs_o(i) \rightarrow cs_r(i) = 1 - (cs_o(i) \cdot (1 - cs_r(i))) =: cs_{o,r}(i) \tag{9.12}$$

$\texttt{test}_{cs}$ berechnet sich daher zu

$$\texttt{test}_{cs}(o, r) = \min_{i \in I} cs_{o,r}(i) \tag{9.13}$$

Da die Attributbedingungen von o und r orthogonal sind, kann zunächst jedes vorkommende Attribut a einzeln untersucht werden. Dabei können drei Gruppen von Instanzen i unterschieden werden, die zusammen den gesamten Raum I ergeben:

- Instanzen i, deren Füllwert für a *in der Zielmenge* der Attributbedingung $o.a$ liegt, d.h. i mit $i.a \in o.a$. Für solche i gilt also $a_o(i) = 1$. $a_r(i)$ ist variabel, im schlechtesten Fall jedoch genau der minimale Zugehörigkeitswert zu $r.a$, was durch $\texttt{subset}(o.a, r.a)$ ausgedrückt wird. Hierzu ist also ein rekursiver Aufruf von `subset` erforderlich.

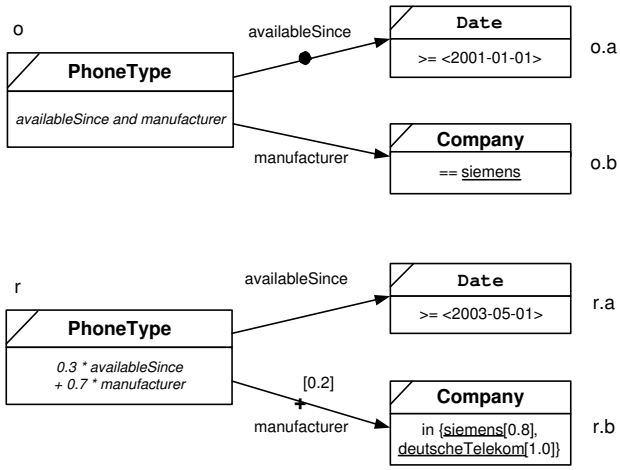


Abbildung 9.3.: Beispiel für die Berechnung von $\texttt{test}_{cs}$.

- Instanzen i, deren Füllwert für a *nicht in der Zielmenge* der Attributbedingung $o.a$ liegt, d.h. i mit $i.a \notin o.a$. Für solche i gilt also $a_o(i) = 0$. Für i muss dann im Normalfall auch angenommen werden, dass es nicht zu $a.r$ gehört, d.h. $a_r(i) = 0$ gilt, außer $a.r$ fehlt oder enthält keinerlei Bedingungen, dann gilt $a_r(i) = 1$.
- Instanzen i, die *keinen Füllwert* für a besitzen, d.h. i mit $i.a = \perp$. Je nach Fehlstrategie auf a_o ist $a_o(i)$ also $0, 1$, `neutral` oder ein angegebener Wert. Analog ist $a_r(i)$ über die Fehlstrategie auf a_r definiert.

Insgesamt entstehen so für jedes Attribut a für die drei Gruppen von Instanzen je zwei konstante Funktionen: $a_o(i)$ und $a_r(i)$, die per Kreuzprodukt zu allen möglichen Instanzgruppen zusammengefügt werden. In jeder wird mittels der Formel 9.12 der Wert für $cs_{o,r}(i)$ direkt aus den Werten für $a_o(i)$ und $a_r(i)$ ermittelt, da ihr Wert in den jeweiligen Abschnitten konstant ist. Insgesamt ergibt sich so der Wert für $\texttt{test}_{cs}$ nach Gleichung 9.13.

Abbildung 9.3 zeigt ein Beispiel für die Berechnung von $\texttt{test}_{cs}$. Der Dienstgeber bietet in o Telefonmodelle von Siemens, die nach 2001 auf den Markt kamen. Der Füllwert zu availableSince kann jedoch aufgrund der Fehlstrategie `ignore` auch fehlen. Der Anfrager verlangt in r entweder ein neueres Telefonmodell *oder* eines von Siemens oder der Deutschen Telekom, was durch die gewichtete Summe in der Verbindungsstrategie zum Ausdruck kommt. Der Hersteller ist ihm aufgrund des Faktors 0.7 wichtiger. Ist der Hersteller nicht gegeben, gilt für diese Bedingung die Präferenz 0.2.

Für die Attribute $a =$availableSince und $b =$manufacturer gelten für eine Instanz i je nach Gruppe:

- Wenn $i.a \in o.a$, dann $a_o(i) = 1$ und $a_r(i) = \texttt{subset}(o.a, r.a) = 0$, da $o.a$ keine Teilmenge von $r.a$ ist.

- Wenn $i.a \notin o.a$, dann $a_o(i) = 0$ und $a_r(i) = 0$, da $r.a$ Einschränkungen besitzt.
- Wenn $i.a = \perp$, dann $a_o(i) = n$ (aufgrund der Fehlstrategie `ignore`) und $a_r(i) = 0$ (aufgrund der standardmäßigen Fehlstrategie `assume_failed`).
- Wenn $i.b \in o.b$, dann $b_o(i) = 1$ und $b_r(i) = \texttt{subset}(o.b, r.b) = 0.8$, da siemens mit der Zugehörigkeit 0.8 in $r.b$ enthalten ist.
- Wenn $i.b \notin o.b$, dann $b_o(i) = 0$ und $b_r(i) = 0$, da $r.b$ Einschränkungen besitzt.
- Wenn $i.b = \perp$, dann $b_o(i) = 0$ und $b_r(i) = 0.2$ aufgrund der Fehlstrategie `assume_value` für b in r.

Interessant für das Minimum von $cs_{o,r}(i)$ sind nur die Fälle, in denen $cs_o(i)$ nicht 0 wird, da sich sonst nach Formel 9.12 der Funktionswert 1 ergibt. Aufgrund der Verbindungsstrategie `and` in o ist das in den zwei Fällen: $(i.a \in o.a \ \wedge \ i.b \in o.b)$ sowie $(i.a \in o.a \ \wedge \ i.b = \perp)$. Im ersten Fall gilt $cs_r(i) = 0.3 \cdot a_r(i) + 0.7 \cdot b_r(i) = 0.3 \cdot 0 + 0.7 \cdot 0.8 = 0.56$. Auch im zweiten Fall gilt $cs_r(i) = 0.56$. Als Minimum und damit als Wert für $\texttt{test}_{dc}(o, r)$ ergibt sich demnach 0.56.

Verrechnung der Testergebnisse. $\texttt{subset}(o, r)$ ergibt sich dann durch Multiplikation der einzelnen Tests, also

$$\texttt{subset}(o, r) = \texttt{test}_{tc}(o, r) \cdot \texttt{test}_{dc}(o, r) \cdot \texttt{test}_{cs}(o, r) \tag{9.14}$$

Subset für eine Menge der Stufe 1

Die Berechnung von $\texttt{subset}(o, r)$ für eine Menge o der Stufe 1 kann iterativ erfolgen, da der Vergleicher auf Seiten des Klienten alle Instanzen von o vollzählig enumerieren kann. Für jede Instanz aus o wird ihr Zugehörigkeitswert zur Menge r bestimmt. Das Ergebnis der `subset`-Operation ist dann der kleinste dieser Zugehörigkeitswerte. Die Operation kann beschleunigt werden, indem ein geeigneter Index verwendet wird, der die Elemente von o direkt und effizient bereitstellt.

Subset für eine Menge der Stufe 2

Die Berechnung von $\texttt{subset}(o, r)$ für eine Menge o der Stufe 2 stellt den schwierigsten Fall dar. Einerseits kann o nicht enumeriert werden, was die Anwendung des iterativen Verfahrens verhindert, andererseits ist die Menge nicht vollständig und damit ihre Attributbedingungen nicht orthogonal. Das Wissen auf Seiten des Dienstnehmers ist daher nicht ausreichend, um den genauen Wert für `subset` zu bestimmen. Als

Möglichkeit bleibt, eine untere Schranke für den Wert zu berechnen, d.h. zu ermitteln, welcher Zugehörigkeitswert zu r durch ein Element aus o sicher nicht unterschritten wird. Diese untere Grenze ist für die Auswahl eines Dienstgebers trotzdem interessant, da sie den Vergleichswert *im schlechtesten Fall* angibt. Möglich ist jedoch, dass dieser Wert bei einer konkreten Ausführung nie erreicht wird, da die dazu nötige Instanz nicht Element von o ist.

Die Abschätzung wird dadurch erreicht, dass die Vollständigkeit von o angenommen wird und `subset` wie im Fall für S0-Mengen berechnet wird. Da das Minimum gesucht ist, sind teilweise vorhandene zentrale Kopien nicht von Bedeutung, da sie den Wert für `subset` nur erhöhen können.

Beim Füllen von Variablen stellen S2-Mengen weitere Schwierigkeiten dar (siehe nächster Abschnitt). Bei ihrem Auftreten reicht es in der Regel nicht mehr aus, nur noch das beste Vergleichsergebnis zu berechnen, sondern alle Möglichkeiten müssen betrachtet werden, da die Ausführung vom Dienstgeber abgelehnt werden kann.

Aufgrund der Probleme mit S2-Mengen sollten diese bei der Erstellung von Angebotsbeschreibungen vermieden werden. Dies kann im Normalfall durch die Bereitstellung von vorschaltbaren Wissensdiensten erreicht werden.

9.1.2. Optimale Konfiguration der Eingabevariablen des Angebots

Typischerweise sind Angebotsbeschreibungen konfigurierbar, d.h. sie enthalten OffIN-Variablen. Aus diesem Grund kann `subset` nicht direkt verwendet werden, um den Vergleichswert zwischen einer Angebotsbeschreibung o und einer Anfragebeschreibung r zu berechnen. Es ist daher nötig, in einem vorgelagerten Schritt o variablenfrei zu machen, indem die OffIN-Variablen mit optimalen Werten gefüllt werden. Hierbei muss jedoch auf Abhängigkeiten zwischen den Variablen geachtet werden. Daher wird im Folgenden das Konzept des *Variablenkontexts* definiert. Ein Variablenkontext ist ein Ausschnitt einer Dienstbeschreibung, innerhalb dessen enthaltene Variablen zusammen verarbeitet und optimiert werden müssen. Verschiedene Kontexte hingegen könnten getrennt betrachtet werden.

Es gilt, dass zwei OffIN-Variablen dann nicht unabhängig sind, wenn sie direkt oder indirekt zur Definition von Attributbedingungen derselben Menge o vom Typ einer Entitätsklasse e verwendet wurden, genauer, wenn o eine unvollständige Menge der Stufen 1 oder 2 darstellt. Ist o eine S1-Menge, so sind die Abhängigkeiten über die enumerierbaren Instanzen bekannt und können somit vom Vergleicher in Betracht gezogen werden. In diesem Fall werden die OffIN-Variablen in einem gemeinsamen Kontext gruppiert. Ist o jedoch eine S2-Menge, sind die Abhängigkeiten nicht vollständig bekannt und können vom Vergleicher nicht in die Berechnung einbezogen werden. Der Vergleicher nimmt dann Unabhängigkeit der Attribute an, und weist an, für diesen Teilbaum *alle* möglichen Variablenbelegungen zu berechnen. Dies ist nötig, falls

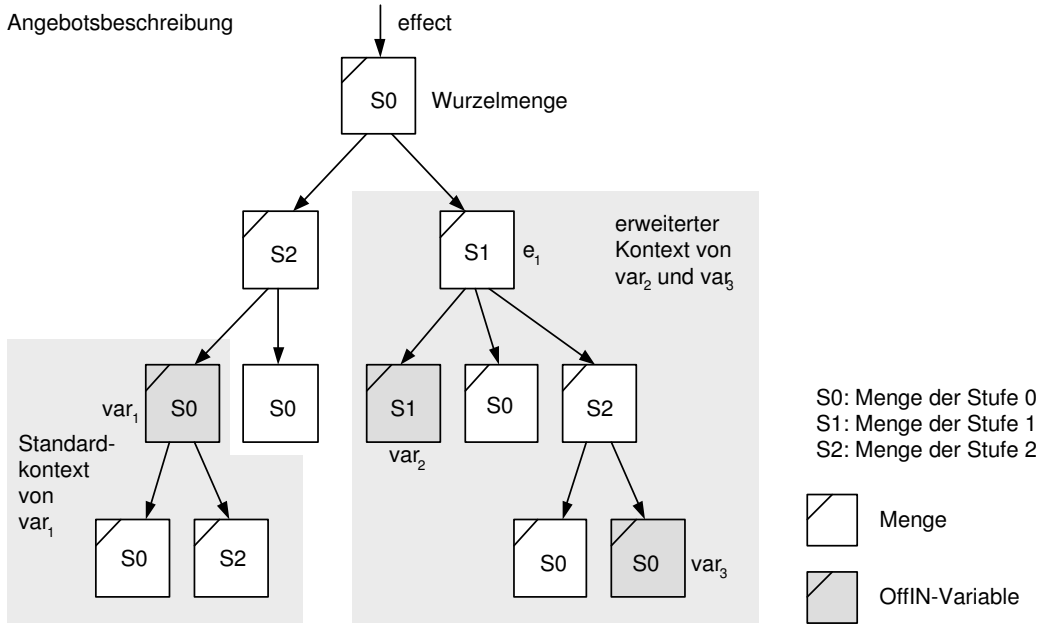


Abbildung 9.4.: Beispiel für einen Standard- und einen erweiterten Variablenkontext.

der Server die Dienstausführung mit der optimalen IN-Variablenbelegung aufgrund einer unterspezifizierten Dienstbeschreibung ablehnt und der Dienstaufruf mit der nächstbesten Belegung wiederholt werden muss.

In einer konkreten Angebotsbeschreibung kann der Kontext einer IN-Variable also wie folgt bestimmt werden:

- Der *Standardkontext* einer IN-Variable var_1 enthält die Variable var_1 selbst sowie alle Mengen und Variablen, die direkt oder indirekt über Attributbedingungen von var_1 aus erreichbar sind.
- Der Standardkontext einer Variable var_2 muss zu einem *erweiterten Kontext* ausgedehnt werden, falls auf dem Pfad zwischen der Wurzelmenge der Effektbeschreibung und var_2 eine Menge der Stufe 1 auftritt. Der erweiterte Kontext umfasst dann diese S1-Menge e_1, die auf dem Pfad zuerst auftrat sowie alle Mengen und Variablen, die direkt oder indirekt über Attributbedingungen von e_1 aus erreichbar sind. Ein solcher erweiterter Kontext kann auch mehrere *OffIN*-Variablen enthalten.

Ein Beispiel für die Kontexte von OffIN-Variablen zeigt Abbildung 9.4. Zu sehen ist eine generische Effektmenge aus einer Dienstangebotsbeschreibung bestehend aus Mengen (weiße Kästchen) und OffIN-Variablen (graue Kästchen). Sie haben verschiedene Stufen, gekennzeichnet durch S0 bis S2. Im Beispiel findet sich keine S1-Menge

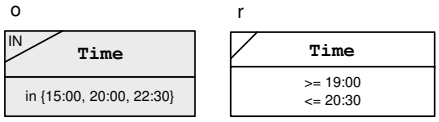


Abbildung 9.5.: Beispiel für die Optimierung einer Variable im Standardkontext.

auf dem Pfad zwischen der Wurzel der Beschreibung und der OffIN-Variablen var_1. Für var_1 wird daher ein Standardkontext ausgewählt, der die Variable selbst und ihre Nachfolger enthält (graue Hinterlegung links). Im Gegensatz dazu sind die OffIN-Variablen var_2 und var_3 im selben erweiterten Kontext, da sie beide e_1 als S1-Menge auf ihren Pfaden von der Wurzel finden (graue Hinterlegung rechts).

Nach Definition kann die optimale Variablenbindung und somit auch der Vergleichswert getrennt für jeden Kontext bestimmt werden. Die Berechnung ist jedoch davon abhängig, ob es sich um einen Standard- oder erweiterten Kontext handelt.

Variablen im Standardkontext

In einem Standardkontext ist nur genau eine OffIN-Variable enthalten. Da der Klient den Füllwert für die IN-Variable selbst bestimmen kann und nicht auf die Auswahl des Dienstgebers angewiesen ist, kann der Vergleicher hier eine *Maximierung* des Wertes anstreben. Der optimale Füllwert für eine solche OffIN-Variable mit der Grundmenge o in der Angebotsbeschreibung und der zugehörigen unscharfen Menge r in der Anfragebeschreibung ist dann die Instanz i_{opt}, welche den größten Wert für $\chi_r(i)$ erzielt:

$$i_{opt}(o, r) = \arg\max_{i \in o} \ \chi_r(i) \tag{9.15}$$

Der optimale Vergleichswert für o und r ist dann durch die Funktion `maxsubset` bestimmt, die wie folgt definiert ist:

$$mv_{opt}(o, r) = \texttt{maxsubset}(o, r) = \max_{i \in o} \ \chi_r(i) \tag{9.16}$$

Da der Standardkontext bis auf die Wurzel variablenfrei ist, kann die Berechnung von `maxsubset` analog zu den Fällen für `subset` von oben erfolgen, jedoch wird stets versucht, ein maximales Ergebnis zu erzielen. Eine Besonderheit ergibt sich bei der Optimierung von solchen Variablen mit einer Grundmenge der Stufe S2. Da hier der Name einer Instanz verlangt wird, können nur Instanzen verwendet werden, für die zentrale Kopien im öffentlichen Instanzenpool vorliegen. Im Unterschied zu `subset` kann also hier das iterative Verfahren verwendet werden.

Ein Beispiel zeigt Abbildung 9.5, bei dem angenommen wird, dass die OffIN-Variable o im Standardkontext zu optimieren ist. Eine Iteration über die in o aufgelisteten Literale liefert $mv_{opt} = 1.0$ und $j_{opt} = 20:00$.

Variablen im erweiterten Kontext

In einem erweiterten Kontext müssen die Füllwerte für die enthaltenen OffIN-Variablen gemeinsam optimiert werden, da sie voneinander abhängen. Die Abhängigkeit rührt daher, da sie verwendet werden, um auf indirekte Weise ein Element aus der Wurzelmenge des Kontext c_o auszuwählen. In einem Beispiel für einen solchen eindeutig spezifizierbaren Dienste könnte ein Telefonmodell über seinen Namen ausgewählt werden. Im Falle mehrdeutig spezifizierbarer Dienste erlaubt die Bindung der OffIN-Variablen nur eine Eingrenzung der Elemente in der Wurzelmenge, wodurch die endgültige Entscheidung der Auswahl beim Dienstgeber bleibt. Im Beispiel könnte das Telefonmodell über OffIN-Variablen vom Klienten durch Hersteller und Fähigkeiten eingegrenzt werden – das konkrete Modell wählt jedoch der Dienstgeber aus. Daher ist die optimale Bindung der OffIN-Variablen im Kontext das Instanztupel $j \in I^n$, welches zum konfigurierten Kontext $c_o[j]$ mit dem höchsten `subset`-Wert im Vergleich zum korrespondierenden Kontext c_r der Anfragebeschreibung führt:

$$j_{opt}(c_o, c_r) = \arg\max_{j \in I^n} \texttt{subset}(c_o[j], c_r) \tag{9.17}$$

Die Berechnung kann effizient erfolgen, indem in einem ersten Schritt festgestellt wird, welche Variablenbelegungen $j \in I^n$ möglich sind, in einem zweiten Schritt, welche Instanzen in jeder so konfigurierbaren Menge $c_o[j]$ enthalten sind und in einem dritten Schritt zu welchem Wert von $\texttt{subset}(c_o[j], c_r)$ das führt. Die Bestimmung der möglichen Variablenbelegungen erfolgt über einen protokollierten Test auf Mengenzugehörigkeit zu c_o für alle Elemente von c_o. Hierbei werden jeweils die Wertekombination $j \in I^n$ gespeichert, die sich jeweils an den OffIN-Variablen ergeben. In einer zweiten Iteration über die Instanzen i von c_o wird dann bestimmt, in welcher dieser Variablenbelegungen sie auftreten. So entstehen Äquivalenzklassen von Instanzen. Die Instanz mit dem schlechtesten Vergleichswert bestimmt den Vergleichswert jeder Äquivalenzklasse; die Klasse mit dem besten Wert gibt die Belegung der Variablen vor. Daher bestimmt ein dritter Durchlauf schließlich über die iterative Grundformel 9.2 den Wert von $\texttt{subset}(c_o[j], c_r)$ für jede Äquivalenzklasse gegeben durch die Variablenbelegung j. Die Belegung mit dem maximalen Wert wird zu j_{opt}.

Abbildung 9.6 zeigt ein Beispiel für einen erweiterten Kontext mit einer S1-Wurzelmenge vom Typ PhoneType. Der Dienstgeber liefert ein Telefonmodell, welches über die Angabe des Herstellernamens und der Fähigkeit eingegrenzt werden kann. Der Dienstnehmer sucht ein sehr neues Modell, ist aber mit geringerer Präferenz bereit,

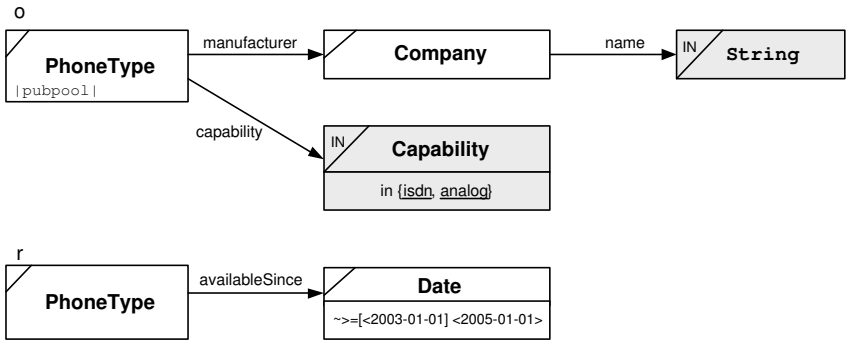


Abbildung 9.6.: Beispiel für die Optimierung einer Variable im erweiterten Kontext.

auch ältere Modelle zu akzeptieren. Im ersten Schritt iteriert der Vergleicher durch alle Elemente von o und merkt sich die auftretenden Werte an den OffIN-Variablen, hier als Tupel der Form (Herstellername, Fähigkeit). Im zweiten Schritt überprüft er jedes Element von o erneut und untersucht in welche Klasse(n) es fällt. Hier könnten beispielsweise in die Klasse (`"Siemens"`, isdn) drei Modelle fallen. Im dritten Schritt wird für jede dieser Klassen k $\texttt{subset}(k, r)$ bestimmt. Für die Klasse (`"Siemens"`, isdn) könnte er > 0 sein, da keines der drei Modelle vor 2003 auf den Markt kam. Unter allen Klassen wird diejenige mit dem größten `subset`-Wert ausgewählt. Sie bestimmt die Werte für j_{opt} und mv_{opt}.

Füllen von Variablen in Gegenwart von S2-Mengen

Wie bereits oben erwähnt, stellt das Auftreten von S2-Mengen innerhalb von Angebotsbeschreibungen weitere Schwierigkeiten dar, da ihre Elemente nicht vollzählig aufgelistet werden können. Die möglichen Füllwerte für OffIN-Variablen, die direkt oder indirekt zur Definition von Attributbedingungen solcher S2-Menge auftreten, sind daher nicht bekannt; eine unterbestimmte Dienstbeschreibung liegt vor. In einem solchen Fall müssen *alle* möglichen IN-Variablenbelegungen j und die dazugehörigen Vergleichswerte bestimmt werden, um sicherzustellen, dass der Dienst mit einer der Konfigurationen tatsächlich ausgeführt werden kann.

Es gilt also: Wenn eine S2-Menge auf dem Pfad zwischen der Wurzel der Effektbeschreibung und der Wurzel eines Kontexts liegt, berechnet der Vergleicher in diesem Kontext nicht nur die beste Belegung der Variablen, sondern alle, die zu einem Vergleichswert > 0 führen. Insgesamt können sich so für eine Anfrage- und eine Angebotsbeschreibung eine Reihe von Vergleichsergebnissen mit unterschiedlicher Variablenbelegung und unterschiedlichem Vergleichswert ergeben.

Da nicht sicher ist, welche dieser Vergleichsergebnisse tatsächlich ausführbar sind, wird zu jedem eine *Ausführungswahrscheinlichkeit* bestimmt, die abschätzt, wie wahr-

scheinlich es ist, dass der Dienstgeber den Dienst in dieser Konfiguration nicht ablehnt. Bei mehreren gleich gut passenden Angeboten sollte daher das Vergleichsergebnis mit der höchsten Ausführungswahrscheinlichkeit bevorzugt werden. Vergleichsergebnisse mit sehr geringer Ausführungswahrscheinlichkeit hingegen sollten verworfen werden, auch wenn sie einen hohen Vergleichswert besitzen. Die Ausführungswahrscheinlichkeit ist somit Bestandteil eines jeden Vergleichsergebnisses.

Die Ausführungswahrscheinlichkeit kann üblicherweise nicht definitiv berechnet, sondern nur abgeschätzt werden. Hierbei helfen die in Abschnitt 7.3.2 eingeführten *Kardinalitätsmarkierer*, die zumindest angeben, wie viele Instanzen der S2-Menge der Dienstgeber tatsächlich unterstützt (siehe dazu [105]).

Abschließen der Konfiguration der Eingabevariablen

Nachdem alle Kontexte analysiert wurden, sind jedem eine oder mehrere OffIN-Konfigurationen sowie die zugehörigen Vergleichswerte zugeordnet. Das Gesamtergebnis des Vergleichs kann jetzt berechnet werden, indem jeder Kontext durch einen Spezialmenge ersetzt wird, die genau die Elemente aus der Wurzelmenge des Kontexts enthält, die durch die optimale Bindung selektiert wurden. Es entsteht so eine variablenfreie Angebotsbeschreibung, die dann mit dem Verfahren aus Abschnitt 9.1.1 weiterverarbeitet werden kann. Wenn dieses auf eine solche Spezialmenge trifft, wird direkt der bereits errechnete Vergleichswert verwendet, anstatt diesen erneut zu berechnen. Enthält der Kontext mehrere Vergleichswerte für verschiedene OffIN-Konfigurationen, entstehen auch im Gesamtvergleich mehrere Vergleichsergebnisse mit unterschiedlichen OffIN-Konfigurationen. Falls später die Dienstausführung für die optimale Konfiguration abgelehnt wird, kann auf die zweitbeste zurückgegriffen werden usw.

9.1.3. Optimale Konfigurationen der Ausgabevariablen der Anfrage

Um den Vergleichsprozess abzuschließen, sind noch die OUT-Variablen der Anfragebeschreibung zu binden. Diese ReqOUT-Variablen stellen die Informationsanforderungen des Dienstnehmers dar. Er ist nur dann an einem bestimmten angebotenen Dienst interessiert, wenn durch dessen erfolgreiche Ausführung die durch die ReqOUT-Variablen spezifizierten Informationen bekannt sind. Daher muss auch ein Vergleichsergebnis mit hohem Vergleichswert verworfen werden, wenn es die Informationswünsche des Dienstnehmers nicht erfüllt.

Jede ReqOUT-Variable r wird gebunden, indem sie mit der entsprechenden Menge o der konfigurierten Angebotsbeschreibung verglichen wird. Dies ist möglich, wenn die

o eine einelementige Menge darstellt, deren Element bekannt ist. Dies ist in verschiedenen Situationen der Fall:

- o liegt innerhalb eines Standardkontexts einer OffIN-Variable. Dann ist der Füllwert für r eindeutig und kann aus dem Füllwert für die OffIN-Variable des Kontexts abgeleitet werden. r wird also mit einem konkreten Wert *gefüllt*.
- o liegt im Standardkontext einer OffOUT-Variable. Dann ist der Füllwert für r nach der Dienstausführung eindeutig, wenn der Füllwert der OffOUT-Variable bekannt ist. r wird also zunächst mit dem Standardkontext *verbunden* (vgl. die Bindungszustände in Abschnitt 6.3.1).
- o liegt im Kontext einer einelementigen Menge.
- o hat eine direkte Bedingung vom Typ ==.
- Über allen definierenden Attributen des Typs von o existieren Attributbedingungen mit einelementigen Zielmengen.
- o liegt im Kontext einer S1-Menge. Die Iteration über alle Elemente liefert nur ein Element, das in r auftreten kann.

Kann r nicht eindeutig gefüllt werden, so wird das Vergleichsergebnis verworfen.

9.1.4. Unbekannte Füllwerte

S1-Mengen in einem Angebot sind so definiert, dass es dem Klienten möglich ist, ihre Elemente vollzählig zu enumerieren. Garantiert ist jedoch nicht, dass die einzelnen aufgelisteten Elemente dem Klienten in allen Füllwerten bekannt sind. Bei einer PE-Menge mit der Markierung `|offpool|` könnte der Dienstgeber von einigen Elementen lokale Kopien mit erweiterten Füllwerten angelegt haben, die dem Klienten unbekannt sind. Eine weitere Quelle unbekannter Füllwerte sind orthogonale Attribute. Ihr Füllwert ist indirekt über Instanzen verdinglichter Zustandsklassen definiert, die häufig nur im Instanzenpool des Dienstgebers zu finden sind.

Die nicht bekannten Füllwerte können jedoch nötig sein, um den Zugehörigkeitswert zu einer Menge im Angebot oder der Anfrage zu bestimmen bzw. um eine OffIN- oder ReqOUT-Variable zu konfigurieren. Es kann dabei nicht einfach auf die angegebene Fehlstrategie zurückgegriffen werden, da der Füllwert nicht *fehlt*, sondern dem Klienten nur *unbekannt* ist. Für eine Instanz i mit unbekanntem Attribut a muss also angenommen werden, dass a ungefüllt oder mit jedem beliebigen Füllwert gefüllt ist. Das hat folgende Auswirkungen:

- Das standardmäßige iterative Vorgehen bei der Berechnung von $\texttt{subset}(o, r)$, bei der für jedes i zunächst die Zugehörigkeit zu o und anschließend $\chi_r(i)$ bestimmt wird, kann nicht mehr durchgeführt werden, da benötigte Füllwerte unbekannt sind. Stattdessen wird i als Menge i_m aufgefasst, wobei die bekannten Füllwerte von i als Attributbedingungen mit einelementigen Zielmengen dienen, die unbekannten Attribute hingegen durch eine fehlende Attributbedingung unbeschränkt bleiben. Die Berechnung von $\chi_r(i)$ erfolgt dann über $\texttt{subset}(i_m, r)$ wie für eine S2-Menge auf rekursive Weise. $\texttt{subset}(o, r)$ stellt das Minimum der Berechnungen über die enumerierten Instanzen i dar.
- Kann bei der Optimierung eines erweiterten Kontext eine OffIN-Variablen nicht gefüllt werden, da der entsprechende Füllwerte für ein Elemente der Wurzelmenge der Stufe 1 nicht bekannt ist, so muss das Ergebnis für *alle möglichen* Variablenbelegungen berechnet werden, falls die dabei auftretende Ausführungswahrscheinlichkeit nicht zu gering wird. Es entstehen somit mehrere Vergleichsergebnisse mit unterschiedlichen Konfigurationen für die OffIN-Variablen, unterschiedlichen Vergleichswerten und Ausführungswahrscheinlichkeiten < 1.
- Soll eine ReqOUT-Variable mit einem Wert gefüllt werden, der unbekannt ist, so ist das Vergleichsergebnis zu verwerfen.

9.1.5. Einbeziehen der Schätzphase

Beim Vergleich von Dienstbeschreibungen können Variablen der Schätzphase helfen, ein genaueres und aktuelleres Vergleichsergebnis zu bestimmen. Sie können einbezogen werden, indem die OffIN_e-Variablen wie gewöhnliche OffIN_x-Variablen in der Phase der Eingabevariablenkonfiguration optimal belegt werden. Trifft der weitere Vergleich dann auf eine $\mathsf{OffOUT}_{e,i}$-Menge o, so wird der dazugehörige Schätzwerte durch Senden der Konfiguration der $\mathsf{OffIN}_{e,i}$-Variablen beim entsprechenden Dienstnehmer angefordert. Hierdurch wird o auf detailliertere Informationen eingeschränkt, was die Chance auf einen Vergleichswert > 0 erhöht.

9.1.6. Vergleich mehrerer Effekte

Sowohl Dienstanfrage- als auch -angebotsbeschreibungen können mehrere effect-Operatoren enthalten. Dabei gilt folgende Semantik: Der Dienstnehmer wünscht, dass zu jeder in der Anfrage beschriebenen Effektmenge genau ein Element neu entsteht, jedoch keine weiteren Effekte erwirkt werden. Der Dienstgeber hingegen sichert zu, dass der Dienst zu jeder in der Angebotsbeschreibung erfassten Effektmenge genau ein Element schafft, jedoch keine weiteren. Eine Angebotsbeschreibung mit den Effektmengen o_1 bis o_n und eine Anfragebeschreibung mit den Effektmengen r_1 bis r_m passen also genau dann zusammen, wenn

Abbildung 9.7.: Dreistufiger Ablauf des Effektvergleichs.

- $n = m$, das heißt, die Anzahl der gewünschten gleich der Anzahl der erbrachten Effekte ist und
- es eine bijektive Zuordnung zwischen den o_i und den r_j gibt, sodass diese nach der Konfiguration der o_i in Teilmengenbeziehung stehen.

In diesem Fall entstehen n Mengen von Vergleichsergebnissen v_1 bis v_n für die einzelnen Effekte, die per Kreuzprodukt miteinander zu einem oder mehreren Gesamtvergleichsergebnissen verbunden werden. Dabei werden die Vergleichswerte und Ausführungswahrscheinlichkeiten multipliziert und die Konfigurationen für OffIN- und ReqOUT-Variablen konkateniert.

9.1.7. Gesamtablauf des Effektvergleichs

Zusammenfassend lässt sich sagen, dass der Effektvergleich in drei Schritten durchgeführt wird (siehe Abbildung 9.7):

1. Bestimmung der *optimalen Konfiguration* für die $\mathsf{OffIN}_{x/e}$-Variablen durch Optimierung der Standard- und erweiterten Kontexte. Dabei werden für Kontexte unterhalb von S2-Mengen *alle* sinnvollen Belegungen zusammen mit ihrer Ausführungswahrscheinlichkeit berechnet. Die Kontexte werden abschließend durch ihre optimale Konfiguration ersetzt.
2. Berechnung der *Vergleichswerte* für die verschiedenen Konfigurationen durch Berechnung von `subset` auf den jetzt variablenfreien Mengen. Hierzu werden die Mengen im Angebot in drei Stufen unterschieden und davon abhängig iterativ oder rekursiv mit denen in der Anfrage verglichen. Im Gegensatz zu Ansätzen in der Literatur, die Schwellwerte zur Unterscheidung von passenden und unpassenden Ergebnissen einsetzen, ist für DSD jeder Vergleichswert > 0 als passend anzusehen. Dies rührt daher, dass jedes Element einer Effektmenge (d.h. Zugehörigkeitswert > 0) in der Anfragebeschreibung einen vom Dienstnehmer gewünschten Effekt darstellt.

3. Bereitstellung der vom Dienstnehmer geforderten Informationen durch *Konfiguration der* ReqOUT-*Variablen*. Je nach Verfügbarkeit werden die Variablen direkt gefüllt oder zunächst mit entsprechenden OffOUT-Variablen verbunden.

9.2. Überprüfung von Vorbedingungen

Eine weitere Aufgabe des Vergleichers ist es, für jedes Vergleichsergebnis zu überprüfen, ob die Vorbedingungen zur Ausführung vollständig erfüllt sind. Nach Abschnitt 7.3.3 unterscheidet DSD in Angebotsbeschreibungen zwei Arten von Vorbedingungen: (1) Informationsvorbedingungen, welche durch die OffIN-Variablen repräsentiert werden, und (2) Zustandsvorbedingungen, welche direkt durch die precondition-Operatoren spezifiziert sind.

Die *Informationsvorbedingungen* eines angebotenen Dienstes gelten als erfüllt, wenn alle OffIN-Variablen seiner Angebotsbeschreibung zu Beginn der Dienstausführung eindeutig belegt sind. Die OffIN-Variablen innerhalb von effect-Operanden werden bereits durch den im letzten Abschnitt vorgestellten Vergleich optimal gefüllt. Da auch OffIN-Variablen innerhalb von precondition-Operanden auftreten können, sind auch diese vom Vergleicher optimal zu binden. Dazu werden diese mit demselben Vorgehen in Standard- und erweiterte Kontexte eingeteilt und optimale Füllwerte bestimmt. Da jedoch in der Anfragebeschreibung keine precondition-Operatoren auftreten, bestehen von Seiten des Dienstnehmers keine Präferenzen – einziges Ziel des Vergleichers ist es also, eine *gültige* Belegung dieser Variablen zu finden. Häufig stehen solche Variablen im Kontext von E-Mengen, die mit dem Markierer `|reqpool|` versehen sind. Der Vergleicher wird daher meist auf die persönlichen Daten im privaten Instanzenpool des Klienten zurückgreifen, um die Variablen korrekt belegen zu können. Im Beispiel in Abbildung 7.18 auf Seite 190 waren das etwa Account-Daten bestehend aus Login und Passwort.

Die *Zustandsvorbedingungen* eines angebotenen Dienstes gelten als erfüllt, wenn keine durch einen precondition-Operator referenzierte Menge zu Beginn der Dienstausführung leer ist. Aufgabe des Vergleichers ist es, dies für jedes gelieferte Vergleichsergebnis sicherzustellen. Kann eine solche Zustandsvorbedingung durch OffIN-Variablen konfiguriert werden, gilt die Bedingung als *vom Dienstnehmer beeinflussbar*, d.h. bereits die Optimierung der Variablenbelegung sorgt für eine nichtleere Menge. Ist die Bedingung *nicht beeinflussbar*, so versucht der Vergleicher lediglich festzustellen, ob die referenzierte Menge ein Element enthält. Dies kann direkt durch Untersuchung des privaten oder öffentlichen Instanzenpools oder indirekt durch Nutzung weiterer Dienste geschehen.

Beim Vergleich von Dienstbeschreibungen auf Seiten des Dienstnehmers ist zu beachten, dass Bedingungen, die für ein Vergleichsergebnis nicht erfüllt sind, nicht direkt

zur Verwerfung dieses Ergebnisses führen müssen. Es kann zunächst versucht werden, die Bedingung auf anderem Wege zu erfüllen, bevor die Dienstausführung angestoßen wird. Ansätze hierzu werden zwar in dieser Arbeit nicht vorgestellt, dennoch sind mittels DSD beschriebene Dienste bereits für eine solche *verkettende Dienstkombination* vorbereitet:

- Fehlende Informationen zum Binden von OffIN_x-Variablen können durch Nutzung von Wissensdiensten beschafft werden.
- Bei Auftreten leerer Mengen kann versucht werden, ein geeignetes Element über einen anderen Dienst zu erzeugen. Hierzu können Realwelt-, Informations-, oder Befähigungsdienste dienen.

Durch Nutzung von weiteren Diensten zur Erfüllung von Vorbedingung des eigentlichen Dienstes entsteht eine Kette von Diensten, die insgesamt den gewünschten Effekt erbringt. Diese Eigenschaft ermöglicht es, einfache Dienste automatisch und zur Laufzeit zu neuen komplexeren Diensten zu kombinieren (vgl. dazu [139]).

9.3. Trennung von Vor- und Hauptvergleich

Im Gegensatz zu Ansätzen der Literatur erfolgt der Vergleich, die Konfiguration und Auswahl von Diensten nicht im Dienstverzeichnis, sondern findet lokal beim Klienten statt (vgl. dazu Abbildung 7.10 auf Seite 173). Das Dienstverzeichnis selbst wendet nur einen Vorvergleich an, um möglicherweise passende Angebotsbeschreibungen zu bestimmen und zum Anfrageagenten zu senden. Dies ist aus folgenden Gründen nötig:

1. Das Verzeichnis hat nicht alle Informationen, um einen Vergleich vollständig durchführen zu können. Insbesondere fehlt der Zugriff auf den persönlichen Instanzenpool des Dienstnehmers.
2. Typischerweise ist das Verzeichnis verteilt und kennt daher nur einen Teil der angebotenen Dienste. Es ist ihm daher nicht möglich, den bestgeeigneten Dienstgeber auszuwählen.[5]

Im Dienstverzeichnis wird daher nur ein vereinfachter *Vorvergleich* auf dort abgelegten Angebotsbeschreibungen durchgeführt. Sein Ziel ist, auf effiziente Weise möglichst viele nicht passende Dienste auszusortieren, wobei auf keinen Fall ein passender Dienst

[5] Auch im Hinblick auf Dienstkombination ist dies wichtig: Hier müssen alle angebotenen Dienste bekannt sein, um diese so kombinieren zu können, dass sie die vom Dienstnehmer gewünschten Effekte gemeinsam erbringen.

verworfen werden darf. Konkret ist also seine Aufgabe, die Zahl der *false positives* zu minimieren, ohne dabei *false negatives* entstehen zu lassen. Das Ergebnis ist eine Liste *möglicherweise passender Angebotsbeschreibungen*, die an den Anfrageagenten übermittelt werden. Der Anfrageagent sammelt alle eintreffenden Listen und übergibt diese an den lokalen Vergleicher zum Hauptvergleich.

Der Vorvergleich kann auf mehrere Arten realisiert werden. Folgende drei Beispiele sollen dies verdeutlichen (aufsteigend nach Präzision geordnet):

- Trivialerweise durch eine Methode, die *alle* Angebotsbeschreibungen liefert. Hierbei werden definitiv keine passenden Dienste übersehen, jedoch ist die Zahl der später nicht passenden Dienste sehr hoch.
- Durch eine Methode, welche nur die *Typen der Effektmengen* einer Angebotsbeschreibung berücksichtigt. Passen diese nicht zur Anfragebeschreibung, kann der angebotene Dienst sicher ausgeschlossen werden.
- Durch eine Methode, die *Angebotsbeschreibungen als variablenfrei* auffasst, indem sie alle OffIN-Variablen durch ihre Grundmengen ersetzt. Untersucht wird dann, ob es eine bijektive Abbildung zwischen nicht-disjunkten Effektmengen im Angebot und in der Anfrage gibt.

9.4. Gesamtbeispiel

Abschließend soll ein größeres Gesamtbeispiel die Vorgehensweise des Vergleichers verdeutlichen. Als Angebotsbeschreibung dient der Realweltdienst aus Abschnitt 7.4.3, der in Abbildung 9.8 wieder aufgegriffen wurde. Für die Beschreibung wurden bereits die Stufen der Mengen sowie die Kontexte der Variablen bestimmt und eingetragen.

Abbildung 9.9 zeigt eine beispielhafte Anfragebeschreibung, für die untersucht werden soll, ob das Angebot zu ihr passt. Wir nehmen dazu an, dass sie ihre ReqIN-Variable an capability vor dem Aufruf mit dem Wert isdn belegt wurde.

Im ersten Schritt werden die **Eingabevariablen des Angebots optimal belegt**. Die drei Variablen in den Standardkontexten können dabei isoliert betrachtet werden, durch die S2-Menge Phone auf dem Pfad von der Wurzel zu den Variablen müssen jedoch *alle* möglichen Belegungen bestimmt werden. Im Falle der Color-Variablen sind das die Instanzen black und silver, da diese nicht durch die Anfragebeschreibung ausgeschlossen wurden. Auch für die Company-Variable müssen beide Belegungen siemens und deutscheTelekom übernommen werden. Die Variable an PhoneCapability kann hingegen eindeutig mit isdn gefüllt werden. Die OffIN-Variablen im erweiterten Kontext von CreditCard werden gemeinsam gefüllt. Da die Instanzen aus dem persönlichen

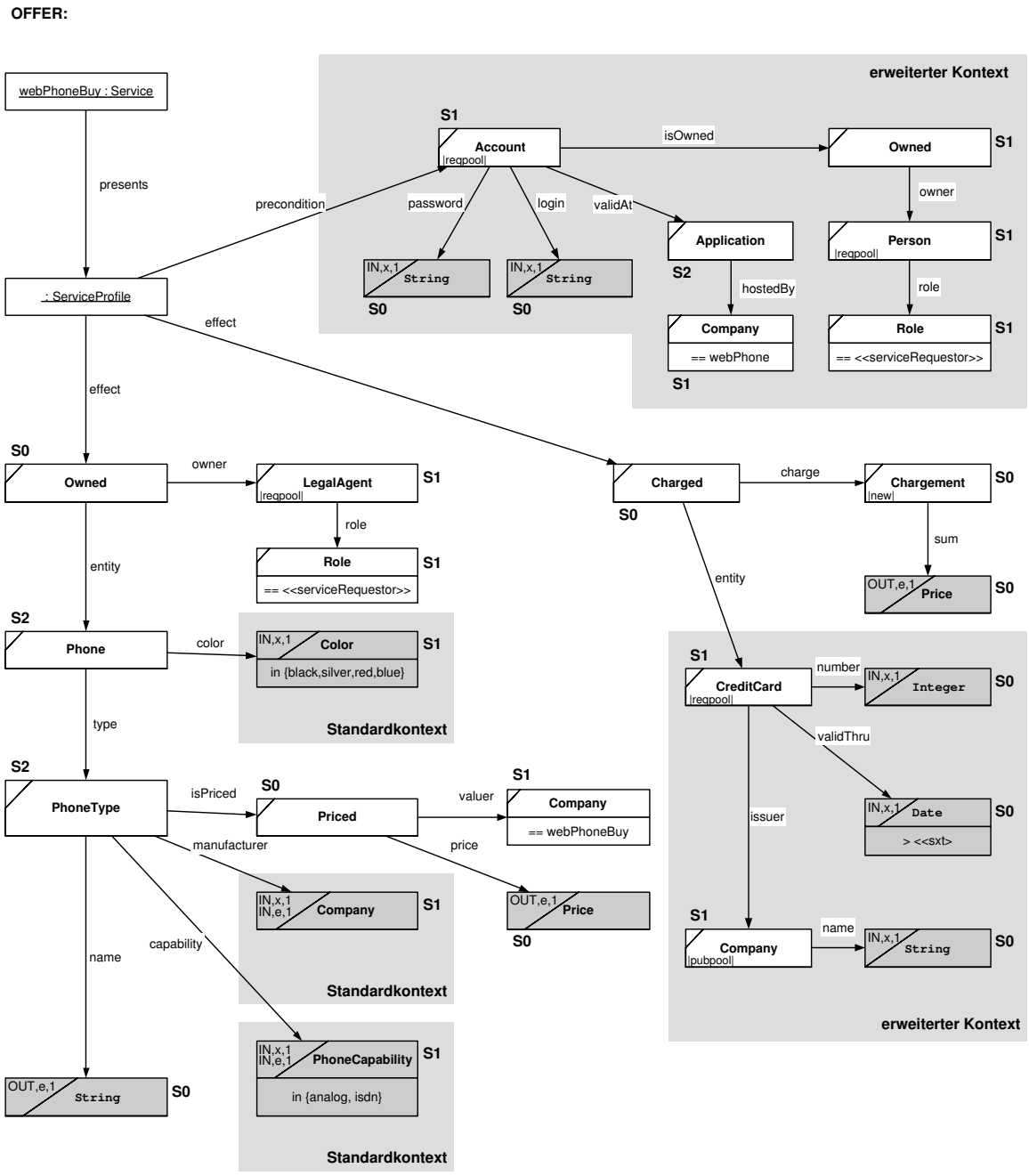


Abbildung 9.8.: Beispielhafte Dienstangebotsbeschreibung aus Abschnitt 7.4.3. Eingetragen sind die Stufen der Mengen sowie die Variablenkontexte.

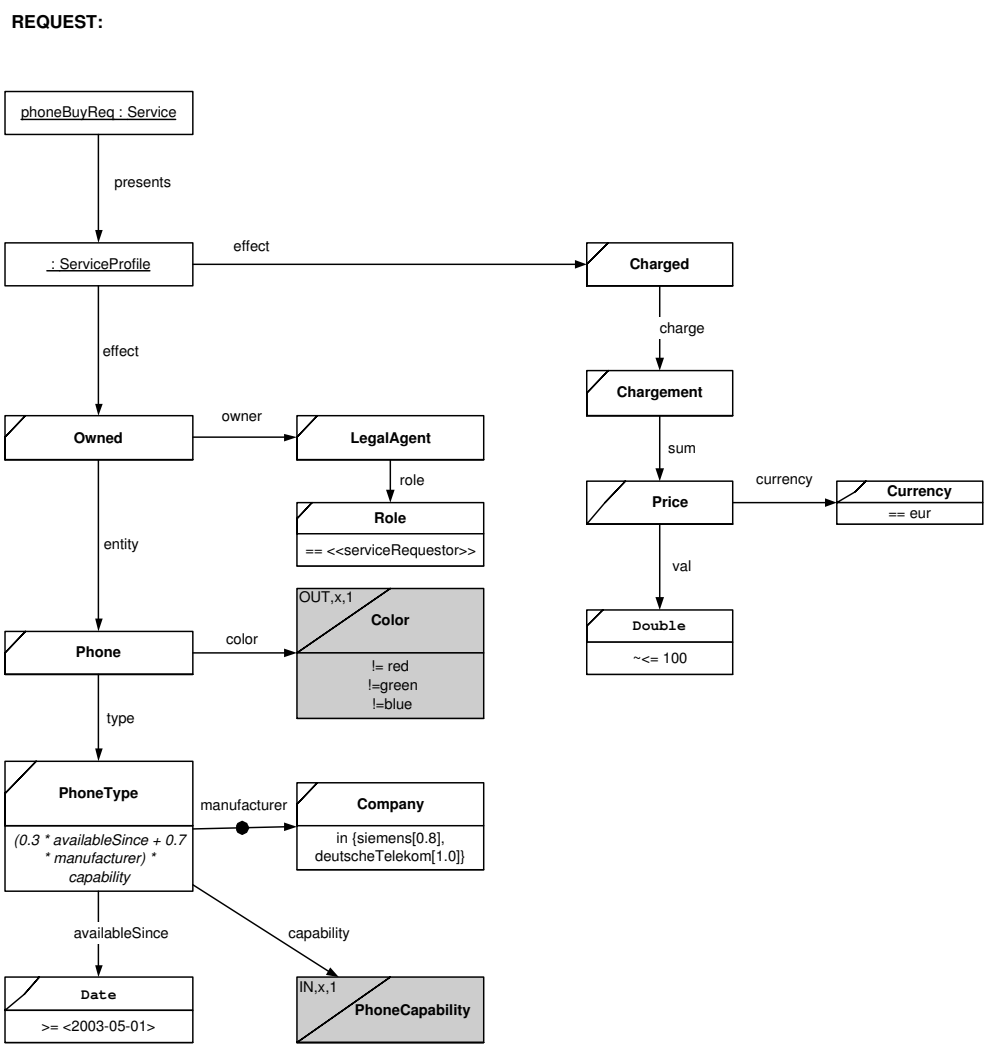


Abbildung 9.9.: Beschreibung eines benötigten Dienstes.

Pool des Dienstnehmers entnommen werden dürfen (`|reqpool|`), kann die optimale Belegung durch Iteration bestimmt werden. Wir nehmen an, dass der Dienstnehmer zwei gültige Kreditkarten in seinem Pool hinterlegt hat. In diesem Fall wird eine davon zufällig ausgewählt und ihre Werte in die OffIN-Variablen eingetragen. Analog werden die OffIN-Variablen im erweiterten Kontext von Account bestimmt. Insgesamt entstehen so also 4 mögliche Variablenbelegungen:

- mr_1 = [black, siemens, isdn, (Kreditkarte), (Account)]
- mr_2 = [silver, siemens, isdn, (Kreditkarte), (Account)]
- mr_3 = [black, deutscheTelekom, isdn, (Kreditkarte), (Account)]

- mr_4 = [silver, deutscheTelekom, isdn, (Kreditkarte), (Account)]

Für diese werden im zweiten Schritt die **Vergleichswerte** bestimmt. Kritisch ist insbesondere der Wert für die S2-Menge PhoneType. Hier wird Orthogonalität der beteiligten Attributbedingungen angenommen und der Vergleichswert durch einen Vergleich der Attribute nach unten abgeschätzt: capability liefert dabei einen Wert von 1, availableSince von 0 (aufgrund des Fehlens im Angebot), manufacturer von 0.8 für mr_1 und mr_2 sowie 1.0 für mr_3 und mr_4. Insgesamt ergibt sich durch die Verbindungsstrategie ein Vergleichswert von 0.56 für mr_1 und mr_2 sowie 0.7 für m_3 und m_4. Da die Bedingung in Color in allen vier Fällen erfüllt ist, ändern sich diese Werte nicht mehr. Im Charged-Effekt muss insbesondere die Bedingung an Price überprüft werden. Dieser steht in der Angebotsbeschreibung jedoch nicht direkt zur Verfügung, sondern muss zunächst über eine Schätzanfrage ermittelt werden. Dazu werden die Belegungen der entsprechenden $\mathsf{OffIN}_{e,1}$-Variablen an den Dienstgeber geschickt, was hier zu zwei Anfragen führt: [siemens, isdn] und [deutscheTelekom, isdn]. Wir nehmen an, dass der Dienstgeber die Schätzwerte 95€ bzw. 105€ als Belastungsbetrag liefert. Durch die unscharfe direkte Bedingung $\sim<= 100$ ergeben sich dadurch folgende Vergleichswerte: $0.56 \cdot 1 = 0.56$ für mr_1 und mr_2 sowie $0.7 \cdot 0.5 = 0.35$ für m_3 und m_4. Alle vier Vergleichsergebnisse sind daher vom Dienstnehmer prinzipiell gewünscht, jedoch auch mit einer Ausführungswahrscheinlichkeit < 1 versehen, da sie Belegungen der Variablen unterhalb der S2-Menge Phone beinhalten.

Im dritten Schritt werden die **Ausgabevariablen der Anfragebeschreibung** belegt. Im Beispiel ist dies nur die Farbe des Telefons, welche direkt aus dem Vergleichsergebnis abgelesen werden kann. Interessant ist, dass Color in der Anfrage eine OUT-Variable, im Angebot dagegen eine IN-Variable darstellt, die Dienste dennoch zueinander passen.

Anschließend überprüft der Vergleicher, ob die Vorbedingungen erfüllt sind. Im Beispiel ist dies gegeben, da die Account-Menge genau den Account des Dienstnehmers enthält.

Die vier Vergleichsergebnisse werden abschließend zurückgegeben. Zunächst wird versucht, den Dienst in der Konfiguration mr_1 anzustoßen. Lehnt der Dienstgeber das ab, können anschließend die Ergebnisse in der Reihenfolge mr_2, mr_3 und mr_4 angestoßen werden.

9.5. Zusammenfassung

In diesem Kapitel wurde ein Vergleicher für DSD vorgestellt. Dieser hat die Aufgabe zu überprüfen, ob eine gegebene Angebotsbeschreibung zu einer Anfragebeschreibung

passt. Da die Anfragebeschreibung präferenzbeinhaltend ist, untersucht er, wie die Effektmengen des Angebots konfiguriert werden müssen, um möglichst gute Teilmengen der Effektmengen der Anfrage zu sein. Durch die Teilmengeneigenschaft wird sichergestellt, dass der Dienst auch im schlechtesten Fall ein noch gewünschtes Ergebnis erzielt. Das Resultat des Vergleichers ist eine Liste von Vergleichsergebnissen jeweils bestehend aus dem Vergleichswert, der Konfiguration der Eingabevariablen des Angebots, der Konfiguration der Ausgabevariablen der Anfrage sowie einer Abschätzung der Ausführungswahrscheinlichkeit.

Zur Berechnung des Ergebnisses geht der Vergleicher wie folgt vor:

- In einem ersten Schritt wird die Angebotsbeschreibung variablenfrei gemacht, indem die enthaltenen Eingabevariablen optimal belegt werden. Hierbei wird unterschieden, ob eine Variable alleine (d.h. im Standardkontext) oder gemeinsam mit anderen (d.h. im erweiterten Kontext) optimiert werden muss.
- Im zweiten Schritt wird der Vergleichswert auf der jetzt variablenfreien Angebotsbeschreibung berechnet. Dazu werden die Mengen der Angebotsbeschreibung in Stufen eingeteilt, je nachdem ob sie vollständig oder vollzählig enumerierbar sind, und entsprechend verarbeitet. Zudem hat der Vergleicher die Möglichkeit, Schätzanfragen an den Angebotsagenten des Dienstes zu richten, um detailliertere Informationen über den Dienst zu erhalten.
- Im dritten Schritt werden die Ausgabevariablen der Anfragebeschreibung belegt, indem die entsprechenden Informationen aus der Angebotsbeschreibung oder den zu erwartenden Rückgabewerten der Dienstausführung abgeleitet werden.
- Abschließend überprüft der Vergleicher, ob alle Zustandsvorbedingungen zur Nutzung eines Vergleichsergebnisses erfüllt sind. Ist dies nicht direkt der Fall, können diese evtl. durch vorgeschaltete weitere Dienste noch erreicht werden.

Sind dem Vergleicher auf Seiten des Klienten Instanzen oder Füllwerte unbekannt, so kann es zu mehreren Vergleichsergebnissen für einen angebotenen Dienst kommen. Diese sind dann mit geschätzten Ausführungswahrscheinlichkeiten versehen, die angeben, in welchem Prozentsatz der Fälle die Ausführung nicht vom Dienstgeber abgelehnt wird.

Dem Vergleicher ist ein Vorvergleicher auf Seiten des Dienstverzeichnisses vorgeschaltet. Dieser wählt auf effiziente Weise aus den in diesem Verzeichnis vorhandenen Dienstangebotsbeschreibungen diejenigen aus, die möglicherweise zur Anfrage passen. Hierbei werden keine passenden Angebote übersehen, jedoch evtl. auch unpassende Angebote geliefert. Das Dienstverzeichnis ist aufgrund fehlender Informationen nicht in der Lage, den Hauptvergleich durchzuführen.

Der Vergleicher stellt die wichtigste Komponente einer dienstorientierten Architektur auf Basis von DSD dar. Seine Existenz beweist, dass die Mächtigkeit von DE/DSD trotz der zahlreichen neuen Sprachelemente beherrschbar bleibt. Dass die Verarbeitung zudem effizient abläuft, wird in Kapitel 10 gezeigt. Insgesamt liefert der Vergleicher wertvolle Informationen für die nachfolgenden Komponenten, mit denen eine Automation der gesamten Dienstnutzung möglich wird.

Teil III.

Schluss

10. Evaluation

Die Evaluation soll zeigen, ob die Ziele erreicht wurden und die vorgestellten Szenarios mit der entwickelten Lösung realisierbar sind. Ziel der Arbeit war nach Abschnitt 1.3 die Entwicklung einer Beschreibungssprache für Dienste, mit deren Hilfe die Dienstnutzung

- universal (d.h. bzgl. aller Domänen),
- vollständig automatisch,
- semantisch korrekt,
- mit akzeptablem Einigungsaufwand
- und unter der Randbedingung einer dynamischen Umgebung

durchgeführt werden kann. Die Untersuchung des Stands der Forschung hat gezeigt, dass existierende Ansätze nicht in der Lage sind, all diese Anforderungen zu erfüllen. Die in dieser Arbeit entwickelte Lösung würde somit einen qualitativen Fortschritt auf dem Gebiet der Dienstorientierung darstellen, so die Ziele erfüllt wären.

Eine Evaluation von Beschreibungssprachen muss immer holistisch durchgeführt werden, d.h. sie darf nicht nur einzelne Aspekte isoliert betrachten, sondern muss stets ihren gesamten Lebenszyklus betrachten. Entsprechende Benchmarks existieren zurzeit nicht in der Literatur – vorhandene semantische Dienstbeschreibungssprachen wurden allenfalls innerhalb einiger Fallstudien und damit nur unvollständig überprüft (siehe z.B. [9, 126]).

Abbildung 10.1 zeigt die einzelnen Stufen einer Nutzung von Diensten im Ansatz dieser Arbeit. Am Anfang stehen die real angebotenen und benötigten Dienste. Diese werden zunächst mittels DSD in Form einer Dienstbeschreibung erfasst, welche an die dienstorientierte Middleware übergeben wird. Dort berechnet der Vergleicher dann daraus Vergleichsergebnisse, die unter anderem den Vergleichswert und die benötigte Konfiguration enthalten. Die Vergleichsergebnisse können dazu verwendet werden, um die Nutzung des Dienstes in einer bestimmten Konfiguration automatisch durchzuführen und so einen (oder mehrere) Effekte zu erwirken.

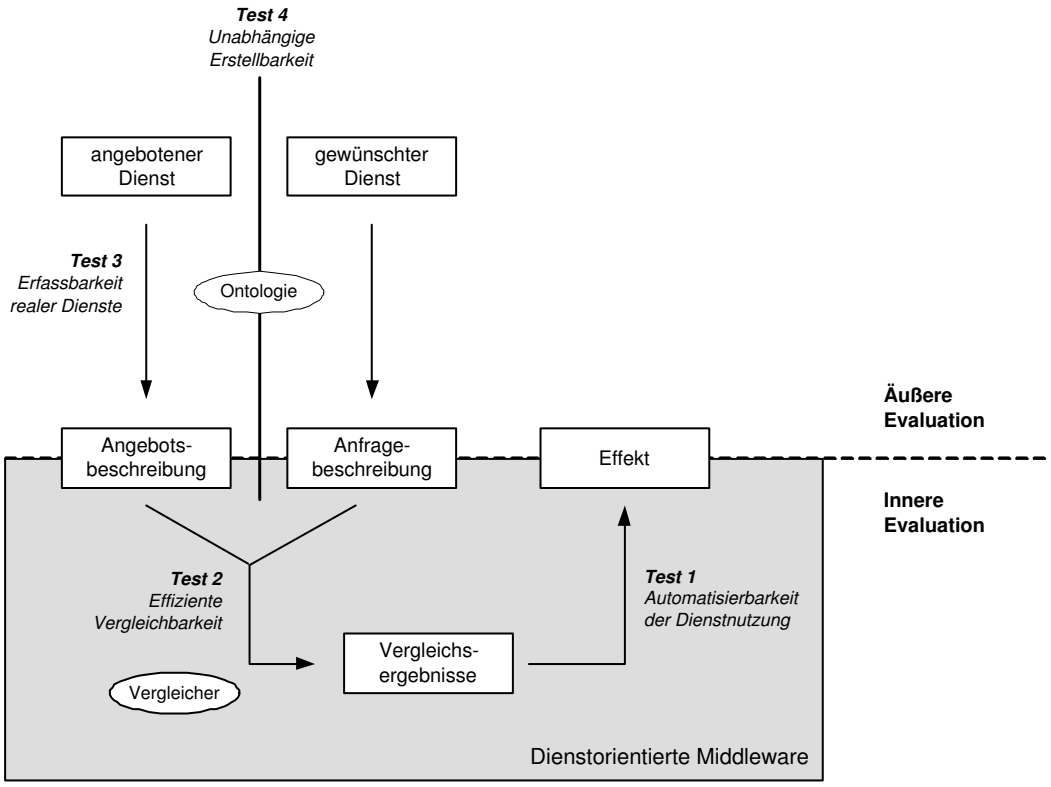


Abbildung 10.1.: Zusammenhang zwischen den vier Tests der Evaluation.

Die Evaluation dieser Arbeit kann entlang dieser Schritte zerlegt werden. Sie wird in umgekehrter Reihenfolge durchgeführt, wodurch im Schritt n Annahmen getroffen werden, die durch Schritt $n + 1$ erfüllt werden. Insgesamt wird untersucht, ob die einzelnen Schritte mit der Dienstbeschreibungssprache DSD durchführbar sind. Beginnend mit dem Zielpunkt, der automatischen Dienstnutzung, ergeben sich folgende aufeinander aufbauende Tests der Evaluation, die zusammen genommen die Erreichung des oben genannten Ziels überprüfen:

- **Test 1: Automatisierbarkeit der Dienstnutzung**. Ausgangspunkt für die Evaluation ist die Zielsetzung der Arbeit: die automatische und semantische korrekte Dienstnutzung. Es muss daher untersucht werden, ob die vom Vergleicher gelieferten Vergleichsergebnisse ausreichen, um hiermit eine automatisch ablaufende Dienstnutzung innerhalb der *Middleware* zu erzielen.

- **Test 2: Effiziente Vergleichbarkeit**. Im zweiten Test muss untersucht werden, ob das in Test 1 benötigte Vergleichsergebnis aus zwei gegebenen korrekten Dienstbeschreibungen effizient bestimmt werden kann. In diesem Test muss also der Aufwand des *Vergleichers* aus Kapitel 9 vermessen werden. Getestet wird

demnach implizit, ob die Ausdrucksstärke der Sprache genügend begrenzt ist, um noch eine effiziente Verarbeitung zuzulassen.

- **Test 3: Vollständige und effiziente Erfassbarkeit realer Dienste**. Dieser Schritt geht von realen Diensten aus und untersucht, ob es möglich ist, für jeden von diesen eine korrekte Dienstbeschreibung wie in Test 2 benötigt mit akzeptablem Aufwand zu erstellen. Hierzu sind *Experimente* mit realistischen Diensten nötig, die von einem Experten der Sprache umgesetzt werden müssen. Insgesamt wird überprüft, ob die Ausdrucksstärke und Flexibilität der Sprache genügend hoch ist, um reale Dienste in Form korrekter und vollständiger Dienstbeschreibungen zu erfassen.

- **Test 4: Unabhängige Erstellbarkeit von Beschreibungen**. Dieser Schritt untersucht, ob für Paare passender Dienste auch passende Beschreibungen erstellt werden, selbst wenn – aufgrund von räumlich und zeitlich verteilten Diensten – keine direkte Absprache zwischen den Erstellern der Beschreibungen möglich ist (abgesehen von den vorher gemeinsam definierten Ontologien und der allgemein bekannten Semantik der verwendeten Sprachkonstrukte). Dieser Test ist wichtig, da insbesondere in verteilten, dynamischen Umgebungen mit variabler Dienstlandschaft zukünftig passende Dienste bei der Erstellung einer Dienstbeschreibung nicht bekannt sein müssen. Auch hierzu ist die Durchführung von *Experimenten* nötig, allerdings nicht mit Experten, sondern mit durchschnittlichen Benutzern. Insgesamt wird überprüft, ob die Sprache genügend strukturiert ist, d.h. ob die Sprache für zueinander passenden Diensten einheitliche Beschreibungen nahe legt.

Bezeichnend ist, dass an die Dienstbeschreibungssprache in den einzelnen Tests alternierende Anforderungen gestellt werden: Test 4 verlangt eingeschränkte Flexibilität, Test 3 hohe Flexibilität und hohe Ausdruckskraft, Test 2 hingegen wieder eingeschränkte Ausdruckskraft. Für Test 1 wird wieder ein hoher Informationsgehalt benötigt. Dies zeigt deutlich, dass das Problem nicht durch eine einseitige Verbesserung in einem Punkt (etwa die stetige Erhöhung der Ausdruckskraft) gelöst werden kann, sondern nur ein geschicktes Ausbalancieren der Sprache zum Erfolg in *allen* Tests führt. In der Tat kann die Gesamtevaluation als konjunktive Verknüpfung der Einzeltests angesehen werden: Sie kann nur dann als erfolgreich angesehen werden, wenn *alle* Tests positiv ausfallen.

Die vier Tests können zu einer inneren und einer äußeren Evaluation gruppiert werden. Zur *inneren Evaluation* gehören die Untersuchungen der Automatisierbarkeit der ersten beiden Tests. In ihr wird die externe, reale Welt nicht betrachtet – Ausgangspunkt sind bereits gegebene korrekte Dienstbeschreibungen, die innerhalb der Middleware verarbeitet werden. Es wird untersucht, ob diese effizient und effektiv zur Erreichung der Ziele eingesetzt werden. Die *äußeren Evaluation* untersucht den Zusammenhang

zu Diensten der realen Welt; zu dieser Evaluation gehören die letzten beiden Tests. Hier wird untersucht, ob solche korrekten DSD-Beschreibungen tatsächlich für alle realistischen Dienste erstellt werden können und ob dies eindeutig genug ist, um auch in verteilten, dynamischen Umgebungen zu funktionieren. Für die äußere Evaluation sind Experimente nötig.

Alle vier Tests wurden für DSD durchgeführt und werden im Folgenden getrennt nach innerer und äußerer Evaluation vorgestellt.

10.1. Innere Evaluation

Ausgangspunkt für die innere Evaluation ist die Existenz semantisch korrekter Dienstbeschreibungen in DSD. Für sie sind zwei Tests durchzuführen: (1) Ist das daraus gewinnbare Vergleichsergebnis ausreichend, um die Dienstnutzung zu automatisieren und (2) kann ein solches Vergleichsergebnis effizient berechnet werden?

10.1.1. Test 1: Automatisierbarkeit der Dienstnutzung

Nach Kapitel 9 entstehen bei einem Vergleich zwischen einer Anfrage- und einer Angebotsbeschreibung ein oder mehrere Vergleichsergebnisse v, jeweils bestehend aus

- dem zugehörigen Dienstanbieter: v.offerer,
- der Konfiguration der Eingabevariablen der Angebotsbeschreibung: v.filledOffINs,
- dem Vergleichswert aus dem Intervall $[0, 1]$: $v.mv$,
- der Konfiguration der Ausgabevariablen der Anfragebeschreibung: v.boundReqOUTs,
- sowie der geschätzten Ausführungswahrscheinlichkeit: $v.ep$.

Zur Bestätigung, dass Test 1 erfüllt ist, wird im Folgenden eine Middleware vorgestellt, die ausgehend von einer Liste solcher Vergleichsergebnisse eine automatische Dienstnutzung durchführen kann. Kernpunkt der Middleware bilden die bereits vorgestellten Anfrage- und die Angebotsagenten. Ihre Schnittstellen, Hilfskomponenten und genaue Arbeitsweise ist in Abbildung 10.2 dargestellt und wird im Folgenden genauer erläutert.

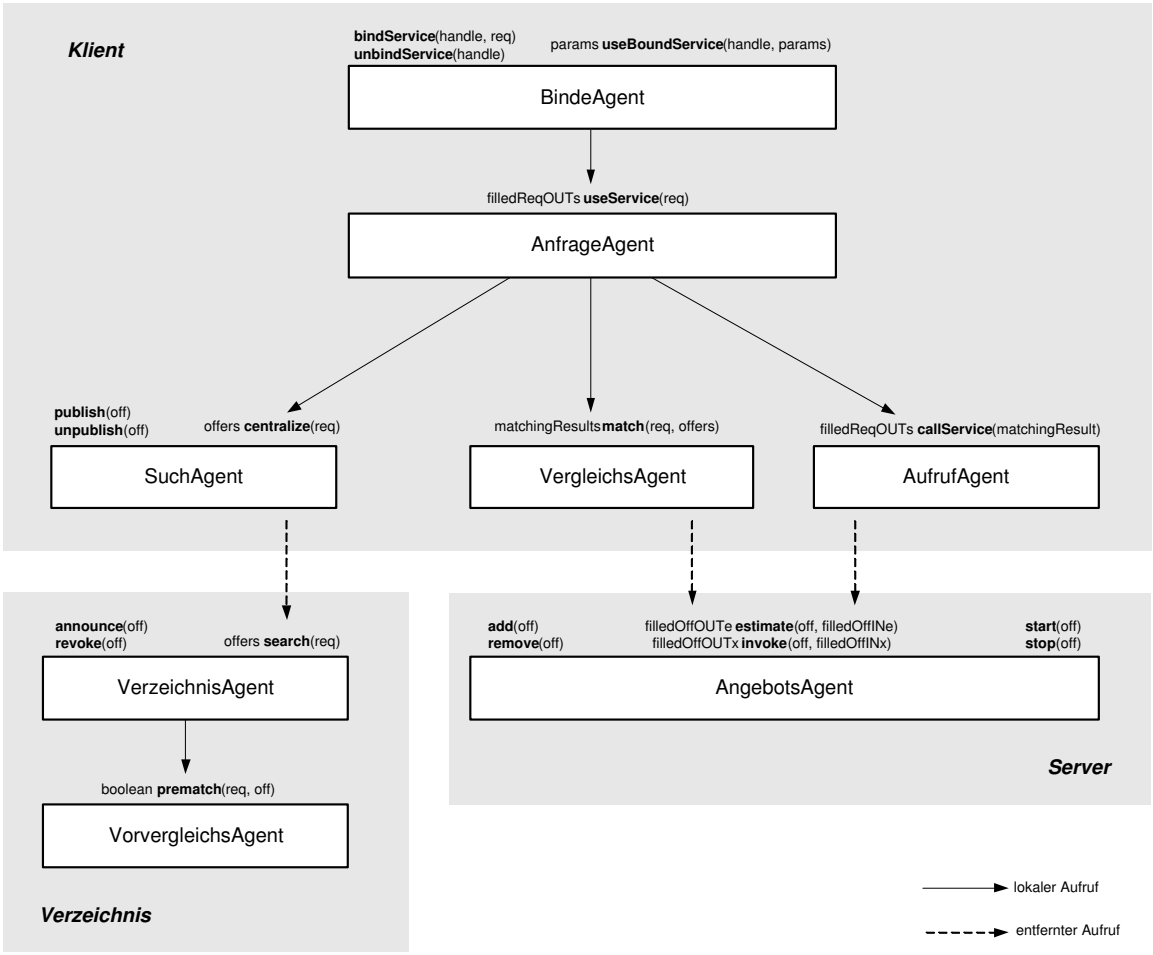


Abbildung 10.2.: Middleware basierend auf indirekter Kommunikation über Agenten, mit welcher die Dienstnutzung auf Basis von DSD-Beschreibungen automatisiert werden kann.

Wir gehen davon aus, dass der Dienstnehmer bereits einen zuvor eingebundenen Dienst (`bindService`) mit einer konkreten Parametrisierung aufgerufen hat (`useBoundService`), wodurch der Anfrageagent über den Suchagenten alle möglicherweise passenden Dienstangebote im Netz aufgesucht (`centralize`) und sich vom Vergleichsagenten die zugehörigen Vergleichsergebnisse hat berechnen lassen (`match`).

Der Anfrageagent verwirft zunächst alle die Vergleichsergebnisse w mit $w.mv = 0$. Falls kein Ergebnis übrig bleibt, liefert er eine Fehlermeldung an den Bindeagent und damit an den Dienstnehmer, anderenfalls wählt er das Vergleichsergebnis v mit dem höchsten Wert $v.mv$ aus. Bei Gleichheit entscheidet eine höhere Ausführungswahrscheinlichkeit $v.ep$. Er beauftragt den Aufrufagent für v eine Ausführung anzustoßen (`callService`). Dieser sendet zunächst an den Angebotsagenten von v.offerer die Konfiguration der OffIN-Variablen v.filledOffINs und stößt damit die Ausführung des

Dienstes an (`invoke`). Das Senden der Werte über ein Netzwerk wird möglich, indem Werte primitiver Typen direkt verschickt werden, während komplexe Instanzen zunächst serialisiert werden. Dazu kann ein DSD-spezifisches oder anderes Übertragungsprotokoll wie SOAP verwendet werden.

Der Angebotsagent [39] auf Seiten des Servers transformiert die Parameter anhand des in der Dienstangebotsbeschreibung hinterlegten Groundings (siehe Anhang A) in einen oder mehrere Aufrufe an den zugrunde liegenden Effektgenerator. Dieser könnte beispielsweise als Javamethode oder SOAP-fähiger Webdienst zur Verfügung stehen. Eventuell sind auch aufwändigere Typumwandlungen nötig, für die dann eine auf eine externe Transformationsvorschrift im Grounding verwiesen wird. In jedem Fall sind alle für den Aufruf benötigten Informationen vorhanden, da die OffIN-Variablen eindeutig belegt sein müssen und diese Informationsvorbedingung vom Vergleicher eingehalten wird.

Der Effektgenerator überprüft daraufhin, ob alle Zustandsvorbedingungen erfüllt sind und ob (im Falle einer unterspezifizierten Beschreibung) die Belegung der Eingabevariablen gültig ist. Falls dies nicht der Fall ist, lehnt er die Ausführung mit einer Fehlermeldung ab. Der Anfrageagent wird daraufhin die Anstoßung des Dienstgebers mit dem nächstbesten Vergleichsergebnisses beim Aufrufagenten veranlassen. Anderenfalls wird der Aufruf des Effektgenerators vollständig durchgeführt.

Nach Beendigung des Effektgenerators empfängt der Angebotsagent eventuelle Ergebnisse, transformiert diese anhand des Groundings in der Angebotsbeschreibung in Füllwerte für die OffOUT-Variablen und sendet diese über das Netz an den Aufrufagenten zurück. Dieser konstruiert aus diesen Füllwerten und der bereits berechneten Konfiguration v.boundReqOUTs die Füllwerte der ReqOUT-Variablen. Dies ist immer möglich, da alle ReqOUT-Variablen in der Konfiguration von v gebunden, d.h. gefüllt oder jetzt mit den Werten der OffOUT-Variablen füllbar sind. Anhand des Groundings in der Anfragebeschreibung werden die Rückgabewerte in den ReqOUT-Variablen vom Bindeagenten noch an die Wünsche des Dienstnehmers angepasst und an ihn zurückgeliefert.

Der Ablauf zeigt, dass Vergleichsergebnisse, wie sie vom Vergleicher aus Kapitel 9 geliefert werden, verwendet werden können, um einen automatischen Ablauf der Dienstnutzung durchführen zu können. Möglich wird dies durch eine dienstorientierte Middleware, die auf eine indirekte Kommunikation zwischen Dienstnehmer und Dienstgeber via Agenten setzt. Die Funktionsfähigkeit der gesamten Middleware nach Abbildung 10.2 wurde durch eine prototypische Implementierung belegt.

10.1.2. Test 2: Effiziente Vergleichbarkeit

Im letzten Abschnitt wurde gezeigt, dass ein DSD-Vergleichsergebnis ausreichend ist, um zu einer automatisierten Dienstnutzung zu gelangen. In Kapitel 9 wurde mit der

Präsentation der Algorithmen eines Vergleichers bereits gezeigt, dass es theoretisch möglich ist, solche Vergleichsergebnisse zu erstellen. In diesem Test 2 wird darüber hinaus untersucht, ob dies auch *effizient*, d.h. in einem vertretbaren Zeitraum machbar ist. Nur wenn gewährleistet werden kann, dass der Vergleich auch bei einer großen Anzahl von Angebotsbeschreibungen in einem angemessenen Zeitraum durchführbar ist, kann der Vergleicher auch in realistischen Szenarios eingesetzt werden.

Zur Messung wurde der Vergleicher aus Kapitel 9 in Java implementiert, wobei zur Steigerung der Effizienz in einigen Punkten von der dort vorgestellten Vorgehensweise abgewichen wurde [105]. Weiterhin wurde in [138] ein Generator für fiktive, aber syntaktisch korrekte Dienstbeschreibungen entwickelt. Dieser erzeugt jeweils eine Anfragebeschreibung und dazu N mehr oder weniger gut passende Angebotsbeschreibungen.[1] Bei der Generierung der Angebote wurde darauf geachtet, dass sie in weiten Teilen der Anfragebeschreibung ähnelten, um so nicht nur den Trivialfall unpassender Beschreibungen zu vermessen. Jede Beschreibung bestand aus genau einem Effekt und enthielt keine Zustandsvorbedingungen. Der Generator lieferte zudem eine Reihe fiktiver privater Instanzen im Pool des Dienstgebers, deren Existenz vom Vergleicher berücksichtigt werden musste.

Laufzeit eines einzelnen Vergleichs

Für den Test wurden 500 Anfragebeschreibungen mit jeweils 30 Angeboten erzeugt, sodass der Vergleicher insgesamt 15.000 Vergleiche durchzuführen hatte. Die Laufzeitmessungen wurden auf einem Athlon XP2000+ mit 256MB Hauptspeicher durchgeführt. Die Ergebnisse werden im Folgenden vorgestellt und dabei mit einigen charakteristischen Eigenschaften der verglichenen Dienstbeschreibungen in Beziehung gesetzt.

Abbildung 10.3 zeigt die Laufzeit eines Vergleichs in Abhängigkeit von der Anzahl der Knoten in der Anfragebeschreibung. Auf der x-Achse ist diese Knotenanzahl aufgetragen, auf der y-Achse der Mittelwert über alle zugehörigen Laufzeiten in Millisekunden.[2] Im Diagramm ist eine leicht steigende Berechnungszeit mit steigender Knotenzahl zu beobachten, die im Bereich von 30 Knoten ihren Höhepunkt erreicht. Anschließend sinkt die durchschnittliche Laufzeit trotz weiteren Wachsens der Knotenzahl. Die anfängliche schrittweise Zunahme ist verständlich, da die Implementierung grundsätzlich einen anfragegetriebenen Lauf durch die Beschreibung vornimmt.

[1] Gleichzeitig berechnet der Generator bei der Erzeugung der Dienstbeschreibungen das jeweilige Vergleichsergebnis, welches ein korrekter Vergleicher liefern sollte. Dadurch war es möglich, die Ergebnisse des Vergleichers mit den Ergebnissen des Generators zu vergleichen und so die Korrektheit dieses Vergleichers sicherzustellen.

[2] Für die fehlenden Punkte lagen keine Messwerte vor, da keine entsprechenden Anfragen vom Generator erzeugt wurden.

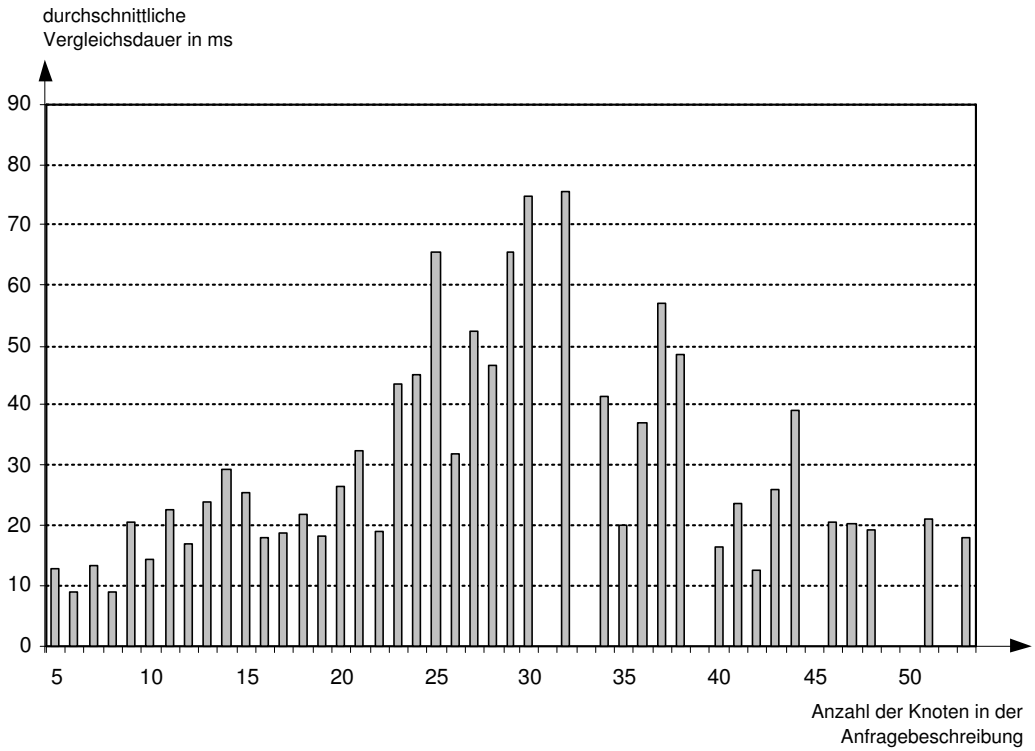


Abbildung 10.3.: Durchschnittliche Laufzeit eines Vergleichs zwischen Anfrage- und Angebotsbeschreibung in Abhängigkeit von der Knotenanzahl in der Anfrage.

Je mehr Knoten in der Anfrage vorhanden sind, desto aufwändiger ist dieser Vorgang. Im Bereich von $x = 30$ scheint sich dies zu wenden: Die Anzahl der Elemente in der Anfragebeschreibung, die kein passendes Element in der Angebotsbeschreibung finden, beginnt zu überwiegen. Durch dieses frühe Scheitern des Vergleichs kann ein weiterer zeitaufwändiger Graphabstieg vermieden werden. Für die Praxis ist dies ein sehr günstiges Verhalten, welches eine hohe Skalierbarkeit des Vergleichers ermöglicht. Insgesamt ist festzuhalten, dass die Laufzeit in keinem der Fälle über 80 ms ansteigt.

In Abbildung 10.4 wird die Vergleichsdauer im Verhältnis zur Anzahl der Mengen vom Typ öffentlicher Entitätsklassen in der Anfragebeschreibung betrachtet. Erkennbar ist ein Trend, bei dem bei steigender Anzahl von PE-Mengen auch die durchschnittliche Vergleichsdauer steigt. Dies ist verständlich, da der Vergleicher hier über die öffentlichen Instanzen iteriert und jeweils mit entsprechenden Menge der Anfrage vergleicht. Der öffentliche Instanzenpool war mit ca. 700 Instanzen eher klein – es ist daher anzunehmen, dass sich Zeitaufwand in praktischen Beispielen bei mehr zu vergleichenden Instanzen erhöht. In der Praxis kann zur Steigerung der Effizienz zudem ein Index zum gezielten Zugriff auf die Instanzen verwendet werden.

Weiterhin wurde die Vergleichsdauer in Abhängigkeit von der Anzahl der E-Mengen

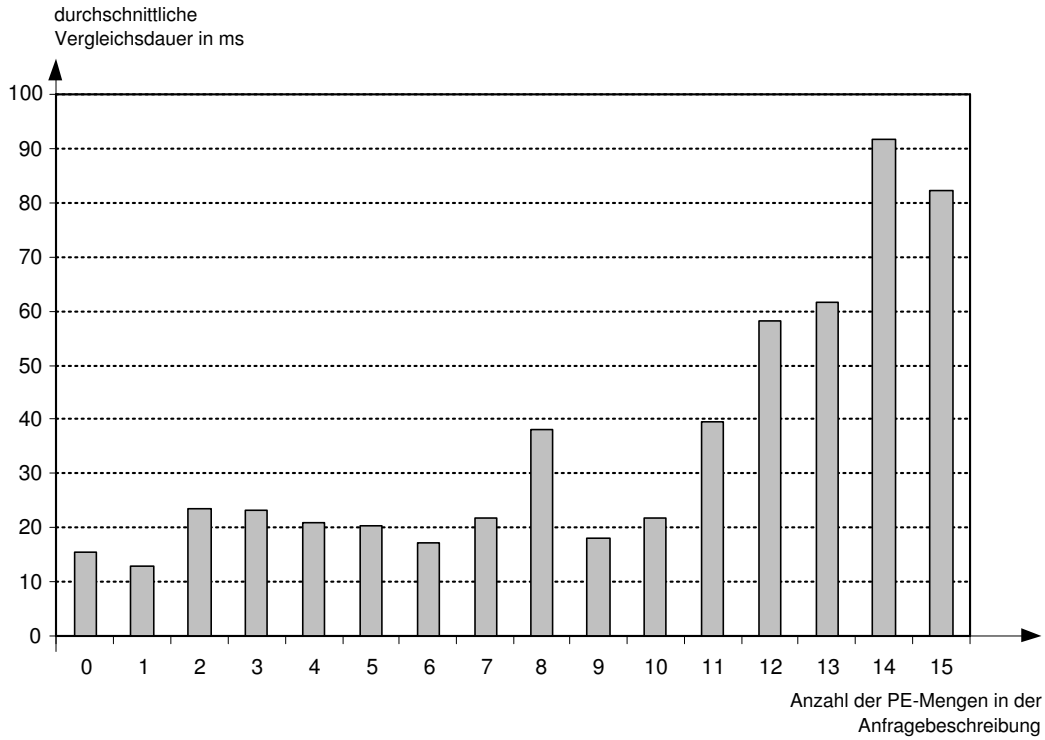


Abbildung 10.4.: Durchschnittliche Laufzeit des Vergleichs in Abhängigkeit der Anzahl von öffentlichen Entitätsklassen in der Anfrage.

oder Variablen in der Anfragebeschreibung untersucht. Hier war jedoch kein Zusammenhang zu erkennen.

Kumulierte Laufzeit für mehrere Vergleiche

Im vorherigen Fall wurde die (durchschnittliche) Laufzeit für einen einzelnen Vergleichsvorgang bestimmt. Dabei wurde vom Testgenerator eine Anfrage- und *nur dazu passende* Angebotsbeschreibungen erzeugt. In der Praxis wird eine Anfrage jedoch mit den verschiedensten Angeboten verglichen, von denen der Großteil einen Vergleichswert von 0 liefert. Für den Anfrageagenten ist jedoch interessant, wie lange er nach dem Absenden einer Anfrage auf die Ergebnisse des Vergleichers warten muss. Aus diesem Grund wurde ein Szenario untersucht, bei dem verschiedene Anfragebeschreibungen mit jeweils 1.000 Angebotsbeschreibungen verglichen wurden. Der Generator wurde so eingestellt, dass 1% der Angebotsbeschreibung als zur Anfrage passend generiert wurden, während 99% der Angebote zufällig, d.h. ohne Bezug zur Anfrage erstellt wurden und damit quasi nie passend waren.[3] Von den 1.000 Vergleichsvorgängen pro

[3]Die Annahme einer solchen Verteilung spiegelt den schlechtesten Fall dar, bei dem nur eine triviale Vorvergleichsstrategie (siehe Abschnitt 9.3) verwendet wurde und somit alle vorhandenen

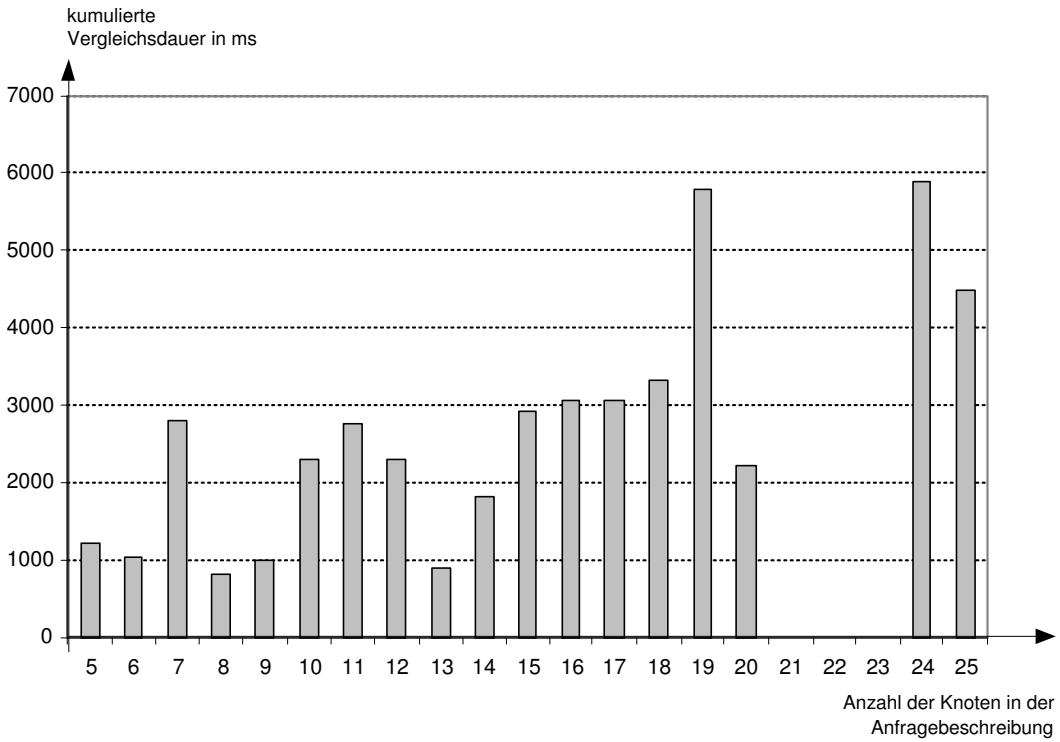


Abbildung 10.5.: Kumulierte Vergleichszeiten beim Vergleich einer Anfrage mit 1.000 Angeboten.

Anfrage lieferten also mindestens 10 Fälle einen positiven Vergleichswert. Die 1.000 einzelnen Vergleichszeiten wurden nicht gemittelt, sondern aufsummiert und bildeten so einen realistischen Wert für die Überprüfung einer Anfrage im Vergleicher. Im Diagramm in Abbildung 10.5 wurden diese kumulierten Vergleichszeiten (y-Achse) in Abhängigkeit von der Knotenanzahl in der Anfragebeschreibung (x-Achse) aufgetragen. Es zeigte sich, dass trotz der großen Zahl von Vergleichen der Anfrageagent im Schnitt nur 2 bis 3 Sekunden auf die Ergebnisse des Vergleichers warten muss. Auch der Maximalwert, der für große Anfragen mit 19 und mehr Knoten erreicht wurde, war mit unter 6 Sekunden noch akzeptabel. Insgesamt ist ersichtlich, dass die in den vorherigen Diagrammen gezeigten Zeiten für einen Einzelvergleich für dieses Szenario nicht einfach mit 1.000 multipliziert werden dürfen, da bei nicht passenden Beschreibungen der Vergleich sehr schnell abgebrochen werden kann. Vergleicht man die Werte der beiden Experimente, so ergibt sich eine durchschnittliche Zeit für den Vergleich zweier unpassender Beschreibungen von ca. 0,5 bis 5,5 ms, im Durchschnitt von ca. 2,5 ms.

Angebotsbeschreibungen verglichen werden müssen. In der Praxis können die Ergebnisse durch den Einsatz intelligenter Vorvergleichsstrategien besser ausfallen.

Da typische Beschreibungsgrößen im Bereich von 7 bis 30 Knoten liegen[4], ist der Vergleicher somit auch in realistischeren Umgebungen in der Lage, die für Test 1 benötigten Vergleichsergebnisse in einem akzeptablen Zeitrahmen bereitzustellen.

Schon die existierende Implementierung erzielt eine hohe Effizienz. Durch eine Reihe von Verbesserungen kann eine weitere Effizienzsteigerung erzielt werden:

- Der Zugriff auf zentrale oder lokale Kopien von Instanzen, die für die Iteration über S1-Mengen benötigt werden, kann durch entsprechende Indices beschleunigt werden.
- Durch eine nicht-triviale Vorvergleichsstrategie können bereits auf Seiten des Dienstverzeichnisses vollkommen unpassende Angebote herausfiltert werden, wodurch die Zahl der zu vergleichenden Ergebnisse drastisch gesenkt werden kann.
- Der Vergleichsvorgang kann parallelisiert werden: Zum einen ist eine datenparallele Implementierung denkbar, die gleichzeitig mehrere Angebote vergleicht, zum anderen können in einer funktionsparallelen Implementierung die einzelnen Stufen des Vergleichsvorgangs pipelineartig verarbeitet werden. Letzteres ist insbesondere dann sinnvoll, wenn die zu vergleichenden Angebotsbeschreibungen von verschiedenen, verteilten Dienstverzeichnissen zeitlich versetzt eintreffen.

10.2. Äußere Evaluation

Die äußere Evaluation stellt die Frage, ob die in der inneren Evaluation angenommene Existenz semantisch korrekter Dienstbeschreibungen für alle realistischen Dienste umsetzbar ist. Konkret überprüft sie, ob (1) typische reale Dienste durch DSD erfasst werden können und (2) ob diese Erfassung auch dann sinnvoll gelingt, wenn die Beschreibungen unabhängig voneinander erstellt werden. Zweiteres ist besonders in dynamischen Umgebungen wichtig, da Beschreibungen von zukünftig möglicherweise passenden Diensten aufgrund der variablen Dienstlandschaft noch unbekannt sind.

10.2.1. Test 3: Vollständige und effiziente Erfassbarkeit realer Dienste

Ausgangspunkt für den dritten Test sind reale Dienste. Es wird untersucht, ob es möglich ist, für alle auf effiziente Weise semantisch korrekte DSD-Beschreibungen zu

[4]Hierzu wurden 29 Dienstbeschreibungen, die innerhalb des DIANE-Projekts eingesetzt wurden, untersucht: nur zwei hatten mit 33 bzw. 59 eine Größe über 30 Knoten, der Mittelwert der Größe lag bei 16,8, der Median bei 15, die Standardabweichung bei 9,7 Knoten.

erstellen, die Ausgangspunkt für Test 2 sind. Für die Untersuchung existieren zwei Möglichkeiten:

Einerseits können *real vorhandene Dienste* (wie beispielsweise Webdienste im Internet) herangezogen und beschrieben werden. XMethods[5] bietet zwar eine Liste von tatsächlich angebotenen Webdiensten, viele davon sind jedoch ähnlich und oder trivial. In der Tat gibt es nur sehr wenige ernsthafte Dienstangebote wie etwa die von Google[6], Amazon[7] oder ebay[8], während reale Dienstanfragen quasi komplett fehlen. Die vorhandenen Dienste sind zudem oft für eine Benutzung durch einen Menschen konzipiert. Diese Art der Evaluation scheidet daher in der momentanen Situation aus.

Alternativ kann ein *synthetischer Benchmark* verwendet werden, der zwar fiktive, aber dennoch realistische, praxisnahe und typische Dienste vorgibt, die mit Hilfe von DSD zu beschreiben sind. Problematisch hierbei ist, dass ein solcher Benchmark für semantische Dienstbeschreibungssprachen zurzeit nicht existiert. Aus diesem Grund wurde zunächst ein generisches Testverfahren in einer unabhängigen Arbeit entwickelt und anschließend auf DSD angewandt.

Erstellung eines Testverfahrens

Um zu wissenschaftlich haltbaren Ergebnissen zu kommen, wurde in [40] ein Testverfahren für semantische Dienstbeschreibungssprachen entwickelt. Es hat folgende Eigenschaften:

- Es wurde *unabhängig* von den Fähigkeiten von DSD entwickelt.
- Es ist *generisch* und daher neben DSD auch auf andere semantische Dienstbeschreibungssprachen anwendbar.
- Die enthaltenen Testfälle sind *realistisch*, d.h. sie sind zwar fiktiv, könnten so aber in einem realen Szenario auftauchen.
- Das Testverfahren kommt zu einem *quantitativen* Ergebnis, d.h. einem Zahlenwert, der den Erfüllungsgrad der Ziele einschätzt.

Realistische Testfälle in Form von umzusetzenden Diensten können nur aus realen Quellen stammen. Sie müssen daher von typischen Benutzern des Systems erstellt werden, beispielsweise in Experimenten. Die Teilnehmer sollten dabei – um die Unabhängigkeit zu wahren – keine Vorkenntnisse im Bereich Semantischer Webdienste

[5] `http://www.xmethods.com`
[6] `http://www.google.com/apis`
[7] `http://www.amazon.com/gp/aws/landing.html`
[8] `http://developer.ebay.com/soap`

besitzen. Zur Erstellung des Testverfahrens wurden daher sechs Informatikstudierende im Hauptdiplom als wissenschaftliche Hilfskräfte angeworben. Ihre Aufgabe war es, benötigte Dienste zu entwerfen, die sie für nützlich und sinnvoll halten. Da der Entwurf von benötigten Diensten in der Regel leichter fällt als der Entwurf von angebotenen Diensten, waren von den Teilnehmern Anfragen zu erstellen. Die Anfragen stammten aus zwei generellen Bereichen:

- *Endnutzeranfragen.* Anfragen dieser Art sind typisch für direkt von menschlichen Endbenutzern gestellte Anfragen, d.h. der Dienstnehmer ist ein Mensch. Sie legen wenig Wert auf eine Wiederverwendbarkeit oder Parametrisierbarkeit, sondern versuchen ein einmaliges Ziel möglichst eindeutig zu erfassen. Bezeichnend für solche Anfragen ist ein tiefes inhaltliches Einbeziehen einer Domäne. Die Bestellung eines Buches oder einer Bahnfahrkarte sind repräsentative Vertreter solcher Anfragen.

- *Applikationsanfragen.* Anfragen dieser Art sind typisch für von Programmierern gestellte Anfragen, die in eine Anwendung integriert werden. Der Dienstnehmer ist später also ein Applikation. Sie repräsentieren den Wunsch nach ausgelagerter Funktionalität, der von einer Anwendung nicht selbst bereitgestellt und daher über einen Dienst beschafft werden soll. Wiederverwendbarkeit und Parametrisierbarkeit aus einem Programm stehen im Mittelpunkt. Der Fokus von DSD liegt auf solchen Dienstbeschreibungen, da sie zur Umsetzung der vorgestellten Szenarios benötigt werden.

Um die Generizität zu wahren, wurden die Teilnehmer aufgefordert, die Anfragen zunächst in natürlicher Sprache bzw. als Methodensignatur in Pseudocode zu erfassen. Sie arbeiteten dabei isoliert und wurden anfänglich nicht über die weitere Verwendung ihrer Anfragen informiert. Um eine maschinelle und automatische Nutzung im Rahmen Semantischer Webdienste zu emulieren, wurden sie explizit darauf hingewiesen, dass die Anfragen so zu formulieren seien, dass sie *ohne weitere Rückfragen* für einen Ausführer verständlich sind. Folgende konkrete Aufgabenstellungen waren in mehreren Experimenten zu bearbeiten:

- **Benötigter Buchkaufdienst** (Endnutzeranfrage): „Beschreiben Sie innerhalb weniger, kurzer Sätze einen Wunsch nach einem einzelnen Buch. Dieser ist an einen Freund gerichtet, der das gewünschte Buch in einer Buchhandlung für Sie einkaufen soll. Nach der Formulierung des Wunsches ist keinerlei Rückfrage oder weitere Absprache möglich.“

- **Benötigter Fahrkartendienst** (Endnutzeranfrage): „Beschreiben Sie innerhalb weniger Sätze einen Wunsch nach einer Fahrkarte für die Deutsche Bahn

Nr	Beschreibung
2	Bruce Schneier: „Angewandte Kryptographie“, gebundene Ausgabe, deutsch, Preis < 70 €.
3	Bring mir bitte ein cooles Fantasy-Buch mit, das am besten in den Bestsellerlisten ist.
4	Ein Fantasy Roman mit Science Fiction Elementen. Nicht mehr als 300 Seiten.
5	Ich brauche einen großen gebundenen Bildband über Meeresfische. Am besten mit bunten Bildern.
6	Ein Nachschlagewerk für Informatik, welches recht gut verkauft wird.

Abbildung 10.6.: Auszug aus den generierten Anfragen nach Buchkaufdiensten.

(1 Person, nur Hinfahrt). Dieser ist an einen Freund gerichtet, der die gewünschte Karte für Sie kaufen soll. Nach der Formulierung des Wunsches ist keinerlei Rückfrage oder weitere Absprache möglich.“

- **Funktionalität für Reisebüro** (Applikationsanfrage): „Sie sind Anwendungsprogrammierer für eine Reisebüro-Software und möchten bestimmte Funktionalität von externen Anbietern nutzen. Die von Ihnen in als kommentierte Methodensignatur zu formulierende Anfrage soll dabei eine solche abgeschlossene und klar umrissene Funktionalität beschreiben.“

Als Ergebnis entstanden 100 Anfragen nach Büchern, 45 Anfragen nach Bahnfahrkarten sowie 50 Anfragen nach einer Funktionalität für ein Reisebüro. Einen kleinen Auszug hieraus zeigen die Tabellen in den Abbildungen 10.6 bis 10.8. Die vollständigen Listen finden sich in [40]. Bezeichnend ist, dass die Anfragen nach Büchern häufig auf deren Inhalt eingehen, aber auch typische Eigenschaften wie ISBN, Titel oder Seitenzahl enthalten. Bei Anfragen nach Bahnfahrkarten waren Abfahrtsort, Zielort und Zeiten die meistgenannten Merkmale. Die Applikationsanfragen umfassten verschiedenste Funktionalitäten wie hier die Belastung einer Kreditkarte sowie die Bestimmung der Durchschnittstemperatur in einer gegebenen Stadt.

Als Grundlage für die Beschreibung der Dienste in einer semantischen Dienstbeschreibungssprache sind entsprechende Domänenontologien nötig. Um keine Logik zu bevorzugen werden diese Grundlagenontologien vom Testverfahren in generischer Form als 10 allgemein verständliche UML-Klassendiagramme zur Verfügung gestellt und müssen bei der Durchführung des Testverfahrens zunächst in die zugrunde liegende Ontologiesprache transformiert und dazu eventuell erweitert werden.

Somit ergeben sich für die Durchführung folgende zwei Aufgaben:

Nr	Beschreibung
103	Fahrt von Karlsruhe nach Berlin, mit Nachtzug, Donnerstag 12. Mai. Abfahrt ab 23:30 Uhr, Bahncard 50, alle Züge (auch IC, ICE).
104	Bring mir bitte ein Zugticket von Karlsruhe nach Hamburg am 03.07.05 mit, sodass ich etwa um 12:00 in Hamburg bin. Schau, ob es irgendwelche Sonderangebote gibt, jedoch kein Ticket für einen Bummelzug.
105	Ich will von Baden-Baden nach München, ohne Umsteigen und ohne den blöden Umweg über Mannheim wie beim letzen Mal.
106	Ich muss von Karlsruhe nach Stuttgart. Am 15.06.2005. Nachtzüge. Uhrzeit: vormittags.

Abbildung 10.7.: Auszug aus den generierten Anfragen nach Fahrkartendiensten.

1. Umsetzung der Grundlagenontologien in die Ontologiesprache der zu bewertenden semantischen Dienstbeschreibungssprache. Dabei darf und soll die Ontologie an die Erfordernisse der Sprache angepasst und um beliebige Konzepte und Regeln erweitert werden. Auch dürfen und sollen bereits vorhandene Ontologien genutzt und integriert werden. Die Anfragen sind in diesem Schritt jedoch noch unbekannt. Gemessen wird die Zeit, die zur Umsetzung der Ontologie benötigt wird.

2. Umsetzung der Anfragen auf Basis der in Schritt 1 entwickelten Ontologien und Regeln in eine Dienstbeschreibung der zu bewertenden Sprache durch einen Experten. Die Ontologien selbst dürfen jetzt nicht mehr angepasst werden, um eine Spezialisierung auf die konkreten Dienste zu vermeiden. Je nach Erfolg werden die Dienste dann in eine von drei Kategorien unterteilt:

   - *Grün.* Der Dienst kann mit den gegebenen Ontologien in der Dienstbeschreibungssprache vollständig und korrekt erfasst werden.
   - *Gelb.* Der Dienst könnte theoretisch mit den in der Beschreibungssprache zur Verfügung stehenden Konstrukten beschrieben werden. Fehlende Konzepte in den umgesetzten Grundlagenontologien verhindern jedoch eine korrekte und vollständige Erfassung. Mit einer Erweiterung der Ontologie wäre eine erfolgreiche Beschreibung möglich.
   - *Rot.* Der Dienst konnte nicht umgesetzt werden, da benötigte Konstrukte in der Dienstbeschreibungssprache fehlten, d.h. die Ausdrucksmächtigkeit nicht ausreichte.

Durch diese beiden Schritte (Umsetzung der Grundlagenontologien und Beschreiben der Dienste) lässt sich demnach ermitteln, ob sich mit der Beschreibungssprache ty-

Nr	Beschreibung

222	`/**`
	`* Belastet eine Kreditkarte um einen Betrag`
	`* @param number Nummer der Kreditkarte`
	`* @param date Gültigkeitsdauer der Kreditkarte`
	`* @param owner Name des Kreditkartenbesitzers`
	`* @param company Firma, die die Kreditkarte ausgestellt hat`
	`* @param sum Belastungsbetrag`
	`* @param cur Belastungswährung`
	`* @param text Text, der in der Buchung erscheinen soll`
	`*/`
	`void belasteKreditkarte(Integer number, Date date,`
	`String owner, Company company, Double sum, Currency cur,`
	`String text)`

229	`/**`
	`* Gibt die Durchschnittstemperatur einer Stadt zurück`
	`* @param city Name der Stadt`
	`* @return unit Temperaturmaßeinheit`
	`* @return value Temperaturwert`
	`*/`
	`(TempUnit unit, TempValue value) gibDurchschnittTempStadt`
	`(String city)`

Abbildung 10.8.: Beispiele für die generierten Applikationsanfragen.

pische realistische Dienste erfassen lassen.

Anwendung des Testverfahrens

In ersten Schritt wurden die 10 UML-Klassendiagramme der **Grundlagenontologie** in DIANE Elements übertragen. Aufgrund des einfachen, natürlichen und objektorientierten Modellierungsparadigmas von DE konnte die Grundlagenontologie schnell (in ca. 30 Minuten) übersetzt werden. Einen Auszug aus der Umsetzung für das Thema *Buch* zeigen die Abbildungen 10.9 und 10.10; die gesamte Umsetzung findet sich in [41]. Trotz der Ähnlichkeit zwischen UML und DE traten an folgenden Stellen Besonderheiten auf:

- In DE wurde das Konzept CopyOfBook hinzugefügt, das ein physisches Buch aus Papier im Gegensatz zur abstrakten Idee als Book darstellt.
- Einige Konzept in DE werden durch Vererbung von existierenden Konzepten (meist aus der oberen Ontologie) in die bereits vorhandene Weltbeschreibung integriert. Hier geschieht dies für Book und CopyOfBook.
- Klassen werden mit Metaeigenschaften versehen. Book wird beispielsweise als teilöffentliche Entitätsklasse ausgezeichnet.

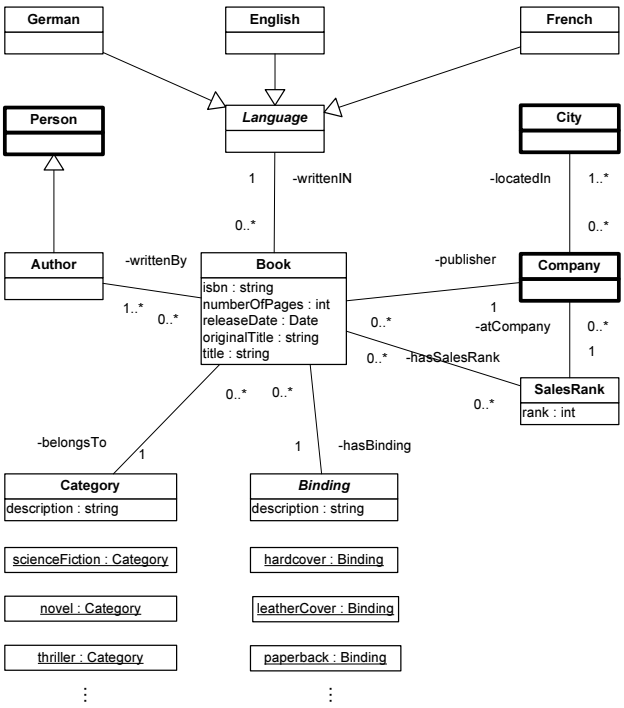


Abbildung 10.9.: Grundlagenontologie zum Thema *Buch* als UML-Klassendiagramm.

- Attribute werden mit Metaeigenschaften versehen. Hier wird beispielsweise das Attribut isbn als definierendes Attribut ausgezeichnet.

Problematisch war einzig die Umsetzung von listenwertigen Attributen wie hier die Liste von Autoren, da ein entsprechendes Konstrukt in DE fehlt.

Im zweiten Schritt wurde versucht, die **Dienste** aus den drei Gruppen mit Hilfe dieser Ontologien in semantische korrekte DSD-Anfragebeschreibungen umzusetzen.

Die Verteilung der Buchkaufdienste ist in Abbildung 10.11a zu sehen. Es fällt auf, dass mit 54% ein Großteil der Dienste als *gelb* eingestuft wird, d.h. prinzipiell umsetzbar wäre, wenn benötigte Konzepte in der Domänenontologie vorhanden gewesen wären. Grund ist die häufige Bezugnahme auf den Inhalt in der Anfrage. Im Gegensatz zu den einfachen, charakteristischen Merkmalen eines Buches wie etwa seine ISBN oder sein Titel, war der gewünschte Inhalt in den seltensten Fällen durch die Ontologie abgedeckt. Dienstanfragen ohne solchen komplexen Inhaltsbezug konnten problemlos abgedeckt werden, was zu 35% *grüner* Dienste führte. 11% der Dienste war nicht modellierbar und mussten der *roten* Kategorie zugeordnet werden. Dies lag im Wesentlichen am fehlenden Listenkonstrukt sowie der nicht vorhandenen Möglichkeit, unscharfe Vergleiche zwischen Zeichenketten durchzuführen.

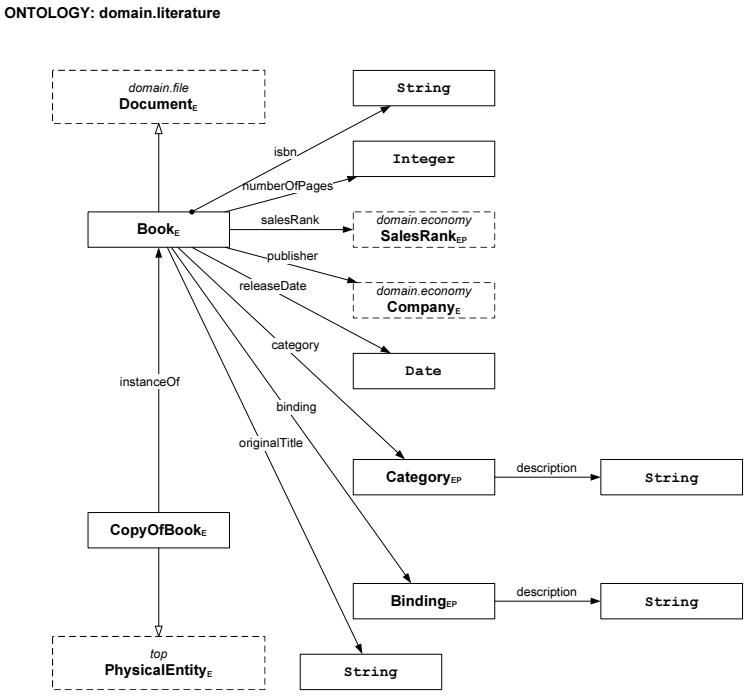


Abbildung 10.10.: Umsetzung der Grundlagenontologie in DE als Ontologie `domain.literature`.

Andere Ergebnisse lieferte die Umsetzung der Fahrkartendienste (siehe Abbildung 10.11b). Im Gegensatz zu den sich durch inhaltliche Beschreibungen auszeichnenden Buchkaufdiensten, konzentrieren sich die Anfragen nach Bahnfahrkarten eher auf direkte Merkmale wie Abfahrtszeiten und Zielorte, was den mit 78% hohen Anteil *grüner* Dienste erklärt. In einigen Fällen (22%) war jedoch auch hier die umgesetzte Ontologie nicht umfangreich genug, sodass diese in die Kategorie *gelb* einsortiert werden mussten. Grund waren jedoch hier oft exotische Wünsche wie Plätze neben den Toiletten oder Anfragen nach zuvor unberücksichtigten Sondertarifen. Keiner der benötigten Dienste musste als *rot* eingestuft werden, da die angebotenen Sprachkonstrukte für alle Anfragen eine ausreichende Mächtigkeit besaßen.

Die besten Ergebnisse wurden im Bereich der Applikationsanfragen erzielt (siehe Abbildung 10.11c). Dies erstaunt nicht, da der Fokus bei der Entwicklung von DSD auf der Unterstützung repetitiver und konfigurierbarer Dienstbeschreibungen lag. Mit einem Anteil von 90% konnten nahezu alle Anfragen dieser Art umgesetzt werden. Nur 2% waren aufgrund fehlender ontologischer Konzepte nicht umsetzbar, 8% scheiterten an einem erforderlichen Listenkonstrukt oder am Fehlen eines Operators, der in der Lage ist, einen Zustand zu negieren (also ein Operator mit *delete-* anstatt mit *new/replace*-Semantik).

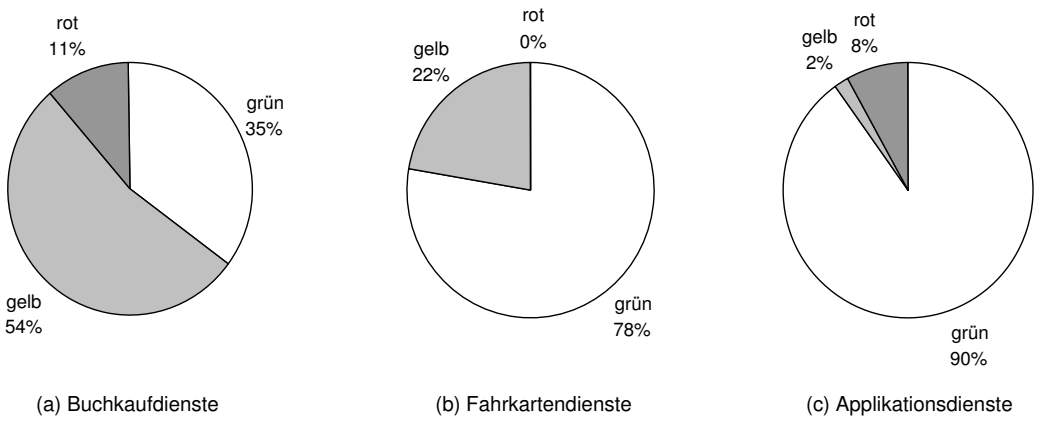


Abbildung 10.11.: Ergebnisse der Erfassbarkeit von realistischen Diensten in DSD: (a) für Endnutzeranfragen nach Buchkaufdiensten, (b) für Endnutzeranfragen nach Fahrkartendiensten, (c) für Anfragen nach Applikationsdiensten.

Insgesamt hat also Test 3 gezeigt, dass die meisten realistischen Dienste von DSD erfasst werden können, und das mit einem akzeptablen initialen Einigungsaufwand. Insbesondere für die wichtigen Applikationsdienste wurden sehr gute Ergebnisse erzielt.

10.2.2. Test 4: Unabhängige Erstellbarkeit von Beschreibungen

Im vierten Test wird untersucht, ob Beschreibungen unabhängig voneinander erstellt werden können, sodass genau für Paare passender Dienste auch Paare passender Beschreibungen entstehen. Diese Anforderung ist wichtig, da der Einsatz der Sprache auch in dynamischen Umgebungen möglich sein muss. Tatsächlich ist eine semantische Dienstbeschreibungssprache, die diese Forderung nicht erfüllt, nutzlos, da mit ihr genau wie bei nicht-semantischen Beschreibungen nur eine statische Bindung von Diensten unterstützt wird.

Dynamische dienstorientierte Umgebungen zeichnen sich dadurch aus, dass ihre Dienstlandschaft zeitlich variabel ist: Dienstgeber treten dem System im Laufe der Zeit bei oder verlassen es; auch ändern sie ihre angebotenen Dienste und deren Konfigurationsmöglichkeiten. Ebenso fluktuiert die Menge der Dienstnehmer, wodurch im Laufe der Zeit unterschiedlichste Dienste benötigt werden. Darüber hinaus kann sich die Umgebung ändern, in der die Dienstnutzung stattfindet, was sich durch Änderungen an den Instanzen in den verschieden Pools widerspiegelt.

Als Eigenschaft ergibt sich, dass bei der Erstellung einer Dienstbeschreibung für einen

angebotenen oder benötigten Dienst in einem dynamischen System die möglicherweise dazu passenden Dienste noch nicht sichtbar sind. Beschreibungen müssen daher isoliert von anderen erstellbar sein, d.h. ohne konkrete Absprache mit den Dienstnehmern bzw. Dienstgebern eventuell passender Dienste. Als gemeinsame Grundlage dienen nur die einheitlichen Ontologien sowie die festgelegte Semantik der Dienstbeschreibungssprache.

Erstellung eines Testverfahrens

Ein dazu geeigneter Test wurde ebenfalls in [40] entwickelt. Die Probanden müssen unabhängig voneinander 10 gegebene Dienste beschreiben. Die Teilnehmer stellen typische Endbenutzer dar, die zuvor eine Schulung in der Dienstbeschreibungssprache (hier DE und DSD) erhalten haben. Während des Experiments dürfen die Teilnehmer die Experten zur Syntax von DE befragen. Die gemeinsame Ontologie wird vorgegeben; auch hier dürfen die Teilnehmer bei den Experten Informationen über die Bedeutung einzelner Konzepte einholen, was den Prozess der Ontologieeinigung simulieren soll. Zur Erstellung der Beschreibungen haben die Teilnehmer prinzipiell keine Zeitbeschränkung und es stehen ihnen alle für die Sprache verfügbaren Werkzeuge zur Verfügung. Um nicht die Fähigkeiten einzelner Teilnehmer in den Vordergrund rücken zu lassen, werden die Teilnehmer in zufällige Dreierteams eingeteilt, die die Aufgabe gemeinsam bewältigen müssen. Jedes Team fasst 10 unterschiedliche Beschreibungen entweder als benötigte oder angebotene Dienste auf und verfasst so 10 Anfrage- (Gruppe I) oder 10 Angebotsbeschreibungen (Gruppe II).

Nach Durchführung des Experiments ist folgende Frage zu klären: Für wie viele der gleichen Dienste wurden auch passende Paare von Angebots- und Anfragebeschreibungen erstellt bzw. für wie viele der ungleichen Dienste waren auch die zugehörigen Beschreibungen unpassend? Dazu wird jede Anfragebeschreibung von Gruppe I mit jeder Angebotsbeschreibung aus Gruppe II verglichen, was zu insgesamt 100 Vergleichen führt. So entstehen 4 Kenngrößen:

- Die Anzahl der *korrekten Passungen* (engl. correct matches, cm), d.h. die Anzahl der Paare passender Angebots- und Anfragebeschreibungen (Vergleichswert $>$ 0), die denselben Dienst beschreiben.
- Die Anzahl der *falschen Passungen* (engl. false matches, fm), d.h. die Anzahl der Paare passender Angebots- und Anfragebeschreibungen (Vergleichswert $>$ 0), die unterschiedliche Dienste beschreiben.
- Die Anzahl der *korrekten Nicht-Passungen* (engl. correct no-matches, cnm), d.h. die Anzahl der Paare nicht-passender Angebots- und Anfragebeschreibungen (Vergleichswert $= 0$), die unterschiedliche Dienste beschreiben.

- Die Anzahl der *falschen Nicht-Passungen* (engl. false no-matches, fnm), d.h. die Anzahl der Paare nicht-passender Angebots- und Anfragebeschreibungen (Vergleichswert = 0), die denselben Dienst beschreiben.

Hieraus lassen sich die *Precision* p mittels

$$p := \frac{cm}{cm + fm} \tag{10.1}$$

und der *Recall* r mittels

$$r := \frac{cm}{cm + fnm} \tag{10.2}$$

bestimmen. Beide Werte liegen im Intervall $[0, 1]$. Kritisch für eine Sprache ist eine Precision $p < 1$, da dann ungewünschte Dienste ausgeführt werden können. Ein Wert von $r < 1$ ist zwar ebenso unerwünscht, jedoch weitaus weniger kritisch, da in diesem Fall nur mögliche Dienstpaarungen übersehen werden.

Anwendung des Testverfahrens

Zur Durchführung des Experiments für DSD standen 6 Teilnehmer zur Verfügung. Es handelte sich dabei um Studierende der Informatik aus dem Hauptdiplom. Aufgrund dieser eher geringen Teilnehmerzahl sind die entstanden Messwerte nur von geringer Signifikanz und sollten nur als Richtwert angesehen werden. Für besser fundierte Ergebnisse empfiehlt der Benchmark aus [40] mindestens 30 Probanden zu verwenden, was jedoch aus finanziellen Gründen nicht möglich war.

Folgende Vorbereitungen wurden vor der Durchführung des Experiments vorgenommen:

- Die Teilnehmer wurde in zwei Sitzungen von einem Experten der Sprache geschult. Sie erhielten dabei eine ausführliche theoretische und praktische Einführung in DE und DSD.
- Die Teilnehmer wurden in zwei Dreierteams eingeteilt. Team I hatte die 10 Dienste in Form von DSD-Anfragebeschreibungen zu erfassen, Team II in Form von DSD-Angebotsbeschreibungen.
- Jedes Team wurde mit einem Laptop ausgestattet. Auf diesem waren Visio sowie geeignete Schablonen zur Editierung von g-dsd vorinstalliert [64].

- Jedem Team standen alle Ontologien im g-dsd-Format als einfach navigierbare HTML-Seiten zur Verfügung [74].
- Jedes Team verfügte über einen Transformator, mit dem es in der Lage war, seine erstellten Beschreibungen auf syntaktische Korrektheit sowie auf Schemakonsistenz zu überprüfen.
- Den Teams wurden verschiedene Räume zugeteilt, sodass sie sich nicht teamübergreifend absprechen konnten. Fragen zur Syntax von DE sowie zur Bedeutung einzelner Ontologiekonzepte durften an einen Sprachexperten gestellt werden.

Das Experiment brachte folgende Ergebnisse:

$$\begin{aligned} cm &= 7 \\ cnm &= 90 \\ fm &= 0 \\ fnm &= 3 \end{aligned} \tag{10.3}$$

Ein Beispiel für eine korrekte Passung zeigen die Abbildungen 10.12 und 10.13. Es handelt sich um die Umsetzung von Dienst 1. Beide Gruppen beschreiben ihn nahezu identisch über die Erreichung des Zustands Visualized eines Dokument, das eine Flugverbindung beschreibt. Einziger Unterschied bei diesen Beschreibungen ist lediglich die spezifischere Angabe des Transportmittels im Dienstangebot, was sich jedoch nicht negativ auf das Vergleichsergebnis auswirkt.

Ein Beispiel für eine falsche Nicht-Passung zeigen die Abbildungen 10.14 und 10.15. Der Unterschied entsteht hier direkt bei der Wahl des vom Dienst erzielbaren Zustandes. Während Gruppe I den Effekt des Dienstes als Reserved sieht, fasst ihn Gruppe II als Owned auf, was der Vergleicher aufgrund der unterschiedlichen Typbedingung mit einem Vergleichswert von 0 bewertet. Solche strukturelle Abweichungen können leicht an den Stellen entstehen, wo sehr generische Zieltypen wie State einen großen Spielraum zulassen. Es ist daher insbesondere wichtig, die entsprechenden Unterklassen sehr gut zu dokumentieren, damit bei der Erstellung von Beschreibungen die Bedeutung der verwendeten Konzepte für alle Beteiligten eindeutig ist.

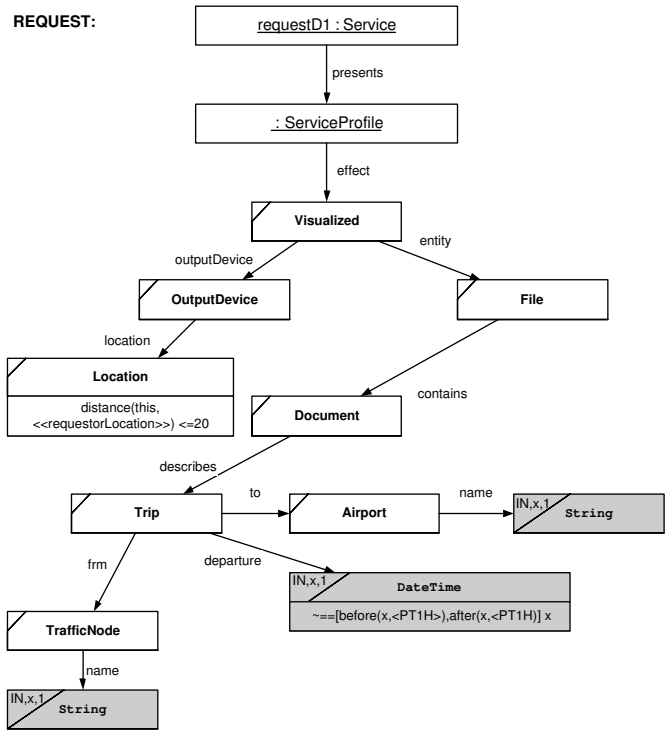


Abbildung 10.12.: Beschreibung von Dienst 1 als Anfrage durch Gruppe I.

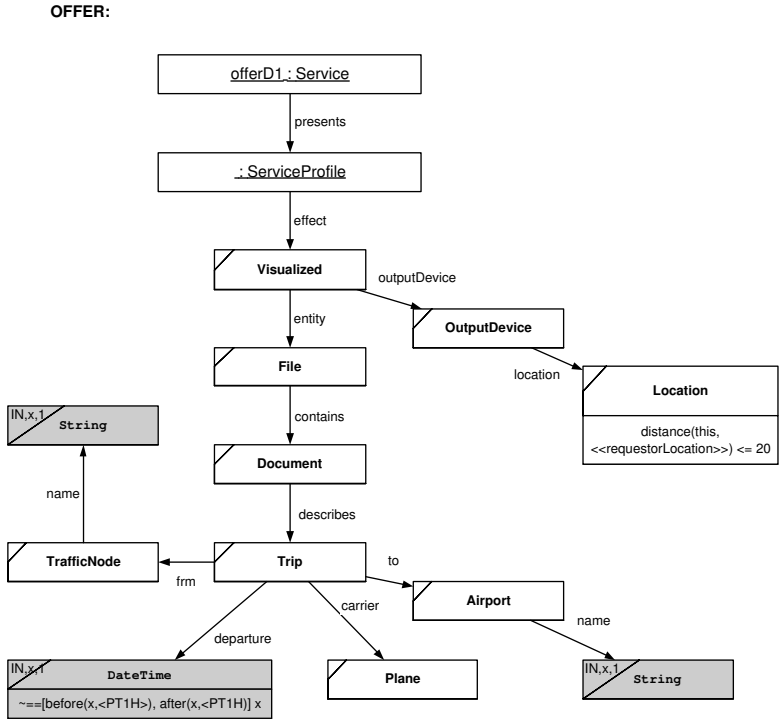


Abbildung 10.13.: Beschreibung von Dienst 1 als Angebot durch Gruppe II.

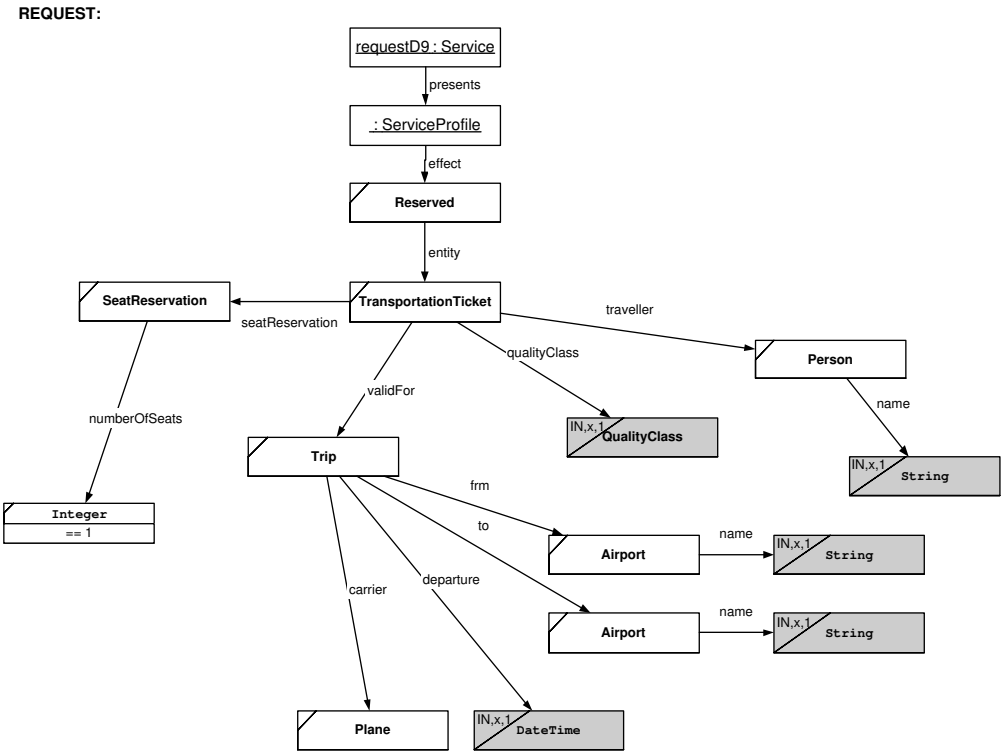


Abbildung 10.14.: Beschreibung von Dienst 9 als Anfrage durch Gruppe I.

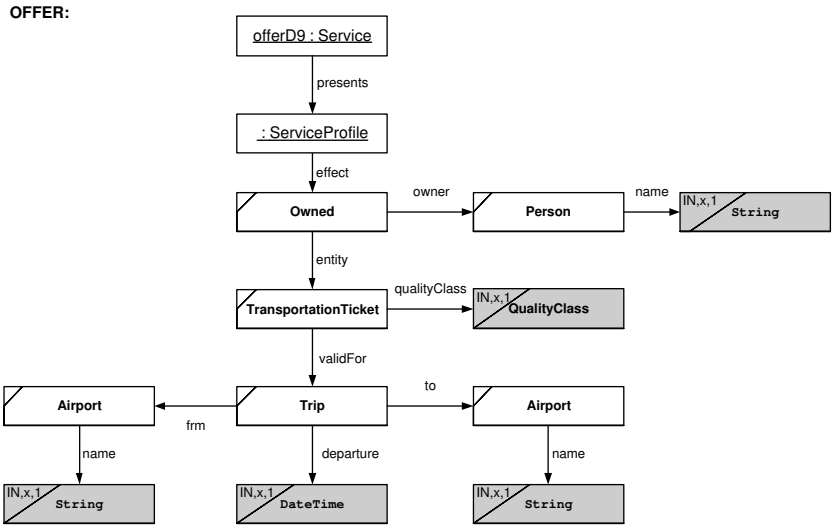
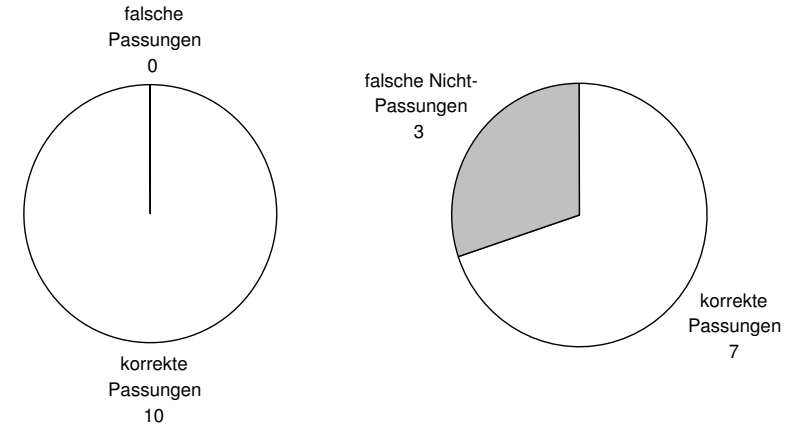


Abbildung 10.15.: Beschreibung von Dienst 9 als Angebot durch Gruppe II.



(a) Bestimmung von Precision p (b) Bestimmung von Recall r

Abbildung 10.16.: Ergebnisse für Precision und Recall.

Insgesamt ergibt sich so eine Precision von $p = 1.0$ und einen Recall von $r = 0.7$ (siehe Abbildung 10.16). Es lässt sich daher (unter Berücksichtigung der geringen Teilnehmerzahl) sagen, dass Dienstbeschreibungen in DSD trotz unabhängiger Erstellung einheitlich aufgebaut sind, was im Wesentlichen auf die direkte Anlehnung an die Schemata der Domänen- und Kategorieontologien sowie die der oberen Dienstontologie zurückzuführen ist. Dies führt zu einer guten Vergleichbarkeit. DSD eignet sich daher auch für eine automatische Dienstbindung in dynamischen Umgebungen mit variabler Dienstlandschaft.

10.3. Fazit

Zur Überprüfung, ob DSD die gesteckten Ziele erfüllt, wurden vier Tests durchgeführt. In den beiden ersten Tests (der so genannten inneren Evaluation) wurde bestätigt, dass korrekte Dienstbeschreibungen in DSD, deren Existenz in diesem Fall vorausgesetzt wird, zur effizienten Automatisierung der Dienstnutzung geeignet sind:

- **Test 1** zeigte, dass ein Vergleichsergebnis, wie es ein Vergleicher nach Kapitel 9 liefert, genug Informationen enthält, um damit die Nutzung eines Dienstes zu automatisieren. Hierzu wurde eine prototypische Middleware vorgestellt. Ihre Grundidee ist es, die Kommunikation zwischen Dienstgeber und Dienstnehmer indirekt über Agenten abzuwickeln. Die Vergleichsergebnisse v helfen dieser bei der Auswahl des bestgeeigneten Dienstgebers (v.offerer, $v.mv$, $v.ep$), bei

dessen Konfiguration (v.filledOffINs) und ermöglichen, die vom Dienstnehmer gewünschten Rückgabewerte zu bestimmen (v.boundReqOUTs). Zu bemerken ist, dass die ermöglichte Dienstnutzung auf bestimmte Kooperationsmuster zwischen Dienstnehmer und Dienstgeber beschränkt ist. Erweiterungen werden im nächsten Kapitel dargestellt.

- **Test 2** zeigte, dass ein solches Vergleichsergebnis auf effiziente Weise von einem Vergleicher nach Kapitel 9 beschafft werden kann, wenn korrekte Dienstbeschreibungen vorliegen. Messungen ergaben, dass passende Angebote von einer prototypischen Implementierung auf einem Standardrechner in weniger als 80 ms verglichen werden konnten. In praxisnahen Beispielen, in denen 1.000 realistische Vergleiche für eine Anfrage zu erledigen waren, konnte dies im Mittel in 2 bis 3 Sekunden durchgeführt werden, selbst in Ausnahmen besonders großer Anfragebeschreibungen stieg der Wert nicht über 6 Sekunden. Obwohl für die Praxis bereits verwendbar, kann eine weitere Effizienzsteigerung durch den Einsatz von Indices erreicht werden.

In den letzten beiden Tests (der so genannten äußeren Evaluation) wurde gezeigt, dass die Annahmen der inneren Evaluation gültig sind, nämlich dass Beschreibungen in DSD für fast alle realen Dienste erstellbar sind, auch unabhängig voneinander in dynamischen Umgebungen.

- **Test 3** zeigte, dass realistische Dienste von einem Experten in Form semantisch korrekter DSD-Beschreibungen erfasst werden können und zwar für die relevanten Applikationsdienste nahezu umfassend und mit einem akzeptablen Einigungsaufwand. Messungen ergaben, dass der nötige initiale Aufwand durch Festlegen von Domänenontologien in DE gering war, was mit den intuitiv verwendbaren, objektorientierten Konstrukten von DE zusammenhängt. Obwohl nicht direkt im Fokus von DSD, konnten von den Dienstanfragen, die direkt von Endbenutzern gestellt wurden, mehr als 30% (Anfrage nach Buchkaufdienst) bzw. 75% (Anfrage nach Fahrkartendienst) direkt umgesetzt werden, weitere 54% bzw. 22% wären mit erweiterten Ontologien möglich gewesen. Bei den für die Szenarios besonders wichtigen benötigten Applikationsdiensten, welche integriert innerhalb einer Anwendung verwendet werden, waren mit 90% fast alle direkt und weitere 8% nach Ontologieerweiterungen umsetzbar. Hauptursache für unmögliche Umsetzungen war ein fehlenden Listenkonstrukt in DE.

- **Test 4** zeigte, dass unabhängig voneinander erstellte Beschreibungen dennoch strukturell ähnlich sind, was einen semantisch korrekten Vergleich erlaubt. Messungen mit 6 Testpersonen ergaben, dass niemals ungewünschte Dienste zur Auswahl kamen (Precision $p = 1$) und nur in 3 von 10 Fällen Dienstangebote übersehen wurden (Recall $r = 0.7$). Obschon dies schon einen für die Praxis

		OWL-S	WSMO	SWSF	**DSD**
A1	Konzeptionelles Modell	-	●	●	●
A2	Universal	●	●	●	●
A3	Inhaltlich eindeutig	●	○	○	●
A4	Angebotene vs. benötigte Dienste	-	●	●	●
A5	Eindeutige Angebotsbeschr.	○	●	●	●
A6	Deterministische Anfrageb.	-	-	-	●
A7	Formale Semantik	-	○	●	●
A8	Effektiv vergleichbar	○	-	-	●
A9	Effizient vergleichbar	●	-	-	●
A10	Unabhängig erstellbar	-	-	-	○
A11	In Applikationen einbindbar	○	○	○	●
A12	Durch Menschen verarbeitbar	●	●	●	●
A13	Positiv evaluiert	-	-	-	○

Abbildung 10.17.: Vergleich der Erfüllung von Anforderungen zwischen DSD und wichtigen Ansätzen der Literatur (● = Anforderung erfüllt, ○ = Anforderung teilweise erfüllt, - = Anforderung nicht erfüllt).

geeigneten Wert darstellt, ist eine weitere Verbesserung des Recalls wünschenswert. Hierzu muss im Wesentlichen die Dokumentation der Zustände in den Kategorieontologien verbessert werden. Insgesamt ist zu vermuten, dass sich DSD auch in dynamischen Umgebungen, d.h. wenn die Dienstlandschaft variabel und bei Erstellung der Dienstbeschreibungen noch unbekannt ist, verwenden lässt. Zur endgültigen Verifizierung sind jedoch weitere Tests mit größeren Teilnehmerzahlen nötig.

Insgesamt wurde durch die vier Tests der Evaluation die Erreichung der Ziele bestätigt: Fast alle relevanten realistischen Dienste können unter akzeptablem initialen Aufwand als korrekte DSD-Beschreibungen erfasst werden, auch unter der Randbedingung dynamischer Umgebungen. Aus diesen Beschreibungen lassen sich mit akzeptablem Zeitaufwand Vergleichsergebnisse berechnen. Diese eignen sich, um im Rahmen einer dienstorientierten Middleware eine vollständig automatische Nutzung der ausgewählten Dienste zu erreichen.

Mit den Ergebnissen der Evaluation wird es möglich, die Anforderungen aus Kapitel 3 erneut aufzugreifen und DSD entlang dieser Anforderungen gegenüber wichtigen Ansätzen der Literatur zu vergleichen. Die Tabelle in Abbildung 10.17 zeigt das Ergebnis. DSD erfüllt nahezu alle Anforderungen vollständig:

- A1: DSD folgt dem konzeptionellen Modell aus Abschnitt 4.1, wonach Dienste als *Menge von Effektmöglichkeiten* angesehen werden.

- A2: DSD verwendet eine *Schichtung von Ontologien* und ist somit durch die Möglichkeit zur Einbindung beliebiger Domänenontologien universal, was auch durch Test 3 bestätigt wurde.
- A3: DE-I verlangt ein *gemeinsames Vokabular* innerhalb der zentralen Ontologie, daher sind darauf aufbauende Dienstbeschreibungen inhaltlich eindeutig.
- A4: DSD hat spezielle Beschreibungen für angebotene und benötigte Dienste.
- A5: DSD verwendet rein *zustandsorientierte Dienstangebotsbeschreibung*. Diese sind eindeutig, unter anderem aufgrund des klar ersichtlichen Einflusses des Informationsflusses auf den Zustandsübergang durch die Verwendung *konfigurierbarer Mengen*.
- A6: DSD verwendet *präferenzbeinhaltende Dienstanfragebeschreibungen*. Diese führen zu einem *persönlichen Vergleicher* und damit zu einem deterministischen Vergleichsergebnis.
- A7: Nach Kapitel 8 existiert für DE-I, -II und DSD eine formale, axiomatische Semantik, die auch die dienstspezifischen Teile einbezieht.
- A8: Test 2 hat gezeigt, dass der Vergleicher effektiv arbeitet.
- A9: Test 1 hat gezeigt, dass der Vergleicher effizient arbeitet.
- A10: Test 4 hat gezeigt, dass Dienstbeschreibungen in gewissen Grenzen unabhängig erstellbar sind. Hier sind jedoch noch Verbesserungen nötig.
- A11: Durch die Verwendung eines Dienstfundaments in Anfragebeschreibungen und der indirekten Kommunikation zwischen Dienstnehmer und -geber zur Laufzeit wird eine Einbindung von Diensten in Applikationen zur Entwurfszeit möglich.
- A12: DE liegt ein objektorientiertes Modell zugrunde, was menschlichen Benutzern eine einfache und intuitive Erstellung von Beschreibungen ermöglicht.
- A13: Wie dieses Kapitel zeigt, hat sich DSD im Rahmen einer Evaluation behauptet. Die Signifikanz der Ergebnisse muss jedoch noch in weiteren Experimenten mit größeren Teilnehmerzahlen untermauert werden.

Das Ziel der Arbeit, also die Schaffung einer Sprache, mittels derer die semantisch korrekten Dienstnutzung in dynamischen Umgebungen (unter den in Abschnitt 1.5 gegebenen Einschränkungen) automatisiert werden kann, ist also erreicht. Mögliche Erweiterungen im Bereich der Dienstausführung, -vermittlung und -komposition sowie an der unterstützenden Infrastruktur zeigt das nächste Kapitel.

11. Weiterführende Arbeiten

Das letzte Kapitel hat gezeigt, dass sich DSD als Dienstbeschreibungssprache als Basis für eine dienstorientierte Middleware eignet, in der Dienste automatisch und semantisch korrekt genutzt werden können. Dennoch existiert eine Reihe von Punkten, an denen das Konzept noch verbessert und erweitert werden sollte, um auch in der Praxis verwendbar zu sein. Konkret sind das

- die *Kombination von Diensten*, d.h. die Erwirkung eines Effektwunschs durch mehr als einen Dienstgeber. Dies ist wichtig, da in der Praxis ein benötigter Dienst häufig nur von mehreren Dienstgebern gemeinsam erbracht werden kann.
- die *Ausführung von Diensten* durch komplexere Kommunikationsparadigmen zwischen Dienstnehmer und Dienstgeber bzw. Dienstgeber und Subdienstgebern. Insbesondere die asynchrone Kommunikation zwischen den Partnern während einer länger dauernden Dienstnutzung ist zu betrachten.
- der Einsatz der Lösung in einer *mobilen Umgebung*. Gerade in solchen Umgebungen bringt die Verwendung einer vollständig automatisierten Dienstnutzung einen entscheidenden Vorteil, da mobile Geräte so ihre mangelnde Leistungsfähigkeit durch externe Dienste transparent verbessern können.
- die Erweiterung um zusätzliche infrastrukturelle Komponenten zur Unterstützung der Dienstnutzung, etwa die Bereitstellung von Ontologieinformationen, das Wachen über die Korrektheit der Beschreibung sowie Werkzeuge zur Unterstützung beim Umgang mit DSD-Beschreibungen.

Diese Bereiche werden im Folgenden genauer untersucht und erste Lösungsmöglichkeiten aufgezeigt.

11.1. Kombination von Diensten

Von *Dienstkombination* spricht man dann, wenn ein Effektwunsch in einer Anfrage nicht von einem einzelnen angebotenen Dienst erbracht werden kann, sondern verschiedene Dienstausführungen eventuell verschiedener Dienstgeber dazu nötig sind,

allgemein also dann, wenn mehrere Vergleichsergebnisse herangezogen werden müssen. Zwar existiert eine Reihe von Ansätzen, die eine semiautomatische Komposition von Diensten anstreben [1, 133], aufgrund der hohen Anzahl zu betrachtender Dienstgeber ist eine vollständige Automatisierung jedoch auch hier erstrebenswert. Es sind dabei mehrere Fälle zu unterscheiden, in denen eine Kombination von Diensten nötig sein kann (vgl. auch [89]):[1]

- **Verkettung.** Zu einer gegebenen Anfrage kann zwar ein einzelner Dienstgeber gefunden werden, der den Effektwunsch vollständig erfüllt, jedoch sind die für seine Ausführung benötigten Vorbedingungen nicht gegeben. In diesem Fall können weitere Dienstgeber herangezogen werden, welche die Vorbedingungen zu erfüllen versuchen. Insgesamt entsteht so eine Kette von Dienstgebern.
- **Mehrere Effekte.** Die Anfragebeschreibung enthält mehrere Effektoperatoren, die von einem einzelnen Dienstgeber nicht vollzählig erzielt werden können. Die Erwirkung der Effekte kann dann auf mehrere Dienstgeber verteilt werden. Der Fall ist dann schwierig, wenn die benötigten Effekte voneinander abhängen.
- **Iteration.** In der Anfrage ist durch eine Iterationsdirektive festgelegt, dass nicht nur eine Instanz einer Effektmenge zu erstellen ist, sondern mehrere, beispielsweise die besten 3, 5 beliebige oder alle. Um dies erreichen zu können, wird in der Regel die Nutzung mehrerer Vergleichsergebnisse nötig.

Die Möglichkeit zur Kombination von Diensten kann erreicht werden, indem ein *Planungsagent* in die Middleware integriert wird. Dieser ist dem Anfrageagenten unterstellt und tritt dann in Aktion, wenn dieser für eine Anfrage feststellt, dass kein angebotener Dienst die gewünschten Effekte alleine erzielen kann. Der Planungsagent versucht dann, durch geschicktes Kombinieren von Teildiensten den Gesamtdienst nachzubilden. Hierzu muss der Vergleicher um eine Operation erweitert werden, welche auch möglicherweise passende Teildienste liefert. Erste Überlegungen hierzu finden sich in [138].

Insgesamt kann die Nützlichkeit einer dienstorientierten Architektur erheblich gesteigert werden, wenn die Möglichkeit besteht, benötigte Dienste auch durch Kombination mehrerer angebotener Dienste bereitzustellen. Die Kombination stellt jedoch hohe Anforderungen an die Beschreibungssprache, da die Funktionalität eines Dienstes so exakt erfasst sein muss, dass daraus automatisch geeignete Teildienste abgeleitet werden können. Im Folgenden werden die oben vorgestellten Fälle genauer betrachtet und

[1] Auch der Dienstgeber selbst kann zur Erbringung seiner Effekte mehrere weitere Subdienstgeber in Anspruch nehmen. Diese *Orchestrierung* wird in Abschnitt 11.2.2 betrachtet. Sie bezieht sich sowohl auf die Suche nach geeigneten Teildienstgebern etwa mit einer der vorgestellten Kombinationstechniken sowie auf die korrekte Koordination des Datenflusses zwischen den gefundenen Teildienstgebern während der Ausführung.

erste Lösungsvorschläge präsentiert, die zeigen, dass DSD als Ausgangspunkt für eine solche Erweiterung geeignet ist.

11.1.1. Verkettung von Diensten

Eine Möglichkeit zur Dienstkombination stellt die *Verkettung von Diensten* dar. Ausgangspunkt ist eine Situation, in der ein Dienst zwar die gewünschten Effekte einer Anfrage erzielen kann, die dazu nötigen Vorbedingungen jedoch nicht gegeben sind. Sie sollen deshalb von einem oder mehreren weiteren Dienstgebern zunächst erfüllt werden. Es existiert eine Reihe von Techniken, um solche Verkettungen effizient berechnen zu können, wie etwa in [122, 161]. Ein Hauptproblem ist jedoch, dass nicht alles, was technisch verbindbar ist, auch semantisch gewünscht ist. Beispielsweise könnte zur Nutzung eines Dienstes der Besitz eines Accounts erforderlich sein, der durch einen anderen Dienst angelegt werden kann, was der Dienstnehmer aber nicht möchte. Weitere Arbeiten auf diesem Gebiet sollten daher zunächst Erweiterungen der Anfragebeschreibung umfassen, um auch solche Präferenzen integrieren zu können.

Einen wichtigen Fall der Verkettung stellt die Voranstellung von Wissensdiensten dar, die zur Ausführung des Hauptdienstes benötigte Instanzen bereitstellen. Wie bereits in Abschnitt 9.1.1 gesehen, können so aus S2-Mengen die wesentlich besser verarbeitbaren S1-Mengen erstellt werden.

11.1.2. Mehrere Effekte in der Anfrage

Der zweite Fall, in dem eine Kombination mehrerer angebotener Dienste nötig werden kann, betrifft Anfragen, in denen mehr als ein Effektoperator angegeben ist, etwa die Beauftragung eines Telefonanschlusses (PhoneExtension) zusammen mit einer Bestellung eines Endgerätes. Gewünscht ist hier eine Semantik der Atomarität: Der Dienstnehmer möchte, dass entweder *alle* angegebenen Effekte erwirkt werden oder *keiner*. Die Ausführung ist daher transaktional zu sichern, was durch die Unumkehrbarkeit einiger Effekte erschwert wird. Wichtige Ansätze zur Unterstützung von Anfragen mit mehreren Effekten finden sich in [14, 100, 101, 156].

Formal enthält die Anfrage also $n > 1$ Effektmengen E_1 bis E_n, für die der Dienstnehmer jeweils die Erzeugung eines Elements wünscht. Für die Beziehung der Mengen untereinander existieren zwei Fälle:

- **Unverknüpft.** Die einzelnen Effekte sind orthogonal, d.h. der Effekt, der tatsächlich in E_1 erwirkt wurde, hat keinen Einfluss auf die erwirkbaren Elemente in E_2 bis E_n usw. Die Effekte können daher getrennt erzielt werden, auch mit unterschiedlichen Dienstgebern, was die Komposition vereinfacht.

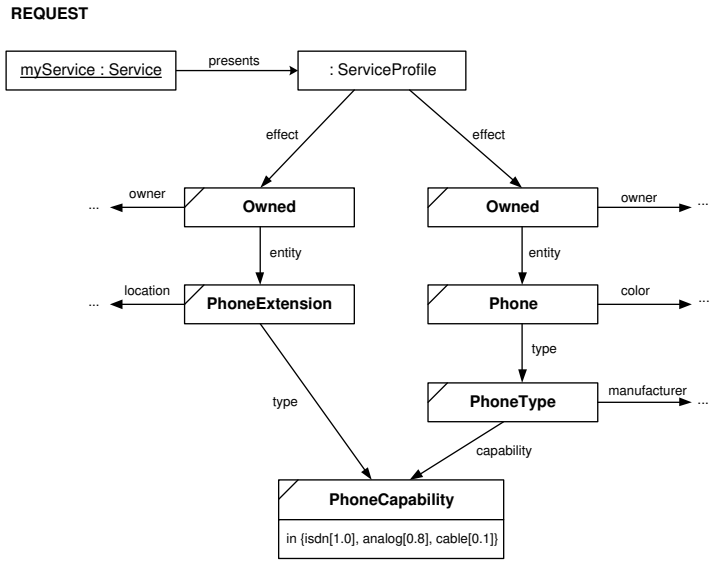


Abbildung 11.1.: Verknüpfte Effekte mit JOIN-Semantik.

- **Verknüpft.** Die Effekte sind voneinander abhängig, d.h. nicht alle Kombinationen $(e_1, \ldots, e_n)$ aus $E_1 \times \ldots \times E_n$ sind gültige, vom Dienstnehmer gewünschte Effekttupel. Beispielsweise könnte der Dienstnehmer ein Endgerät wünschen, welches technisch zum bestellten Telefonanschluss passt. Die Komposition gestaltet sich in diesem Fall weitaus schwieriger, da die Effekte nicht mehr völlig getrennt bereitgestellt werden können.

In der Praxis tritt der Fall verknüpfter Effekte häufiger auf, da verschiedene Ziele meist genau aus diesem Grund in einer Anfrage zusammengefasst werden.

Im Folgenden wird gezeigt, wie die Verknüpfung mehrerer Effekte in DSD realisiert werden kann. Zur besseren Übersicht wird nur von zwei Effekten ausgegangen. Die Notation aller Beschreibungen sollen nur die jeweiligen Ideen vermitteln. Sie sind daher nur vorläufig und beispielhaft. Bei einer Fortführung der Konzepte müssen sowohl Syntax als auch Semantik genau definiert werden. Für Notation und Semantik von verknüpften Effekten sind zwei Vorgehensweisen denkbar:

- **JOIN-Semantik.** Die Effekte sind über eine *gemeinsame Menge* verknüpft. Ein Beispiel zeigt Abbildung 11.1. Der Typ des Telefonanschlusses und die Fähigkeit des Telefons zeigen hier auf die selbe Menge. Als Semantik für gemeinsame Mengen muss festgelegt werden, dass Effekttupel nur dann zur Anfrage passen, wenn die Einzeleffekte *das gleiche* Element in der gemeinsamen Menge besitzen. Alle beteiligten Effekte sind demnach gleichberechtigt. Das Finden geeigneter Dienstgeber und ihrer Konfiguration gestaltet sich in diesem Fall als sehr schwierig, da der gesamte Tupelraum $E_1 \times E_2$ betrachtet werden muss.

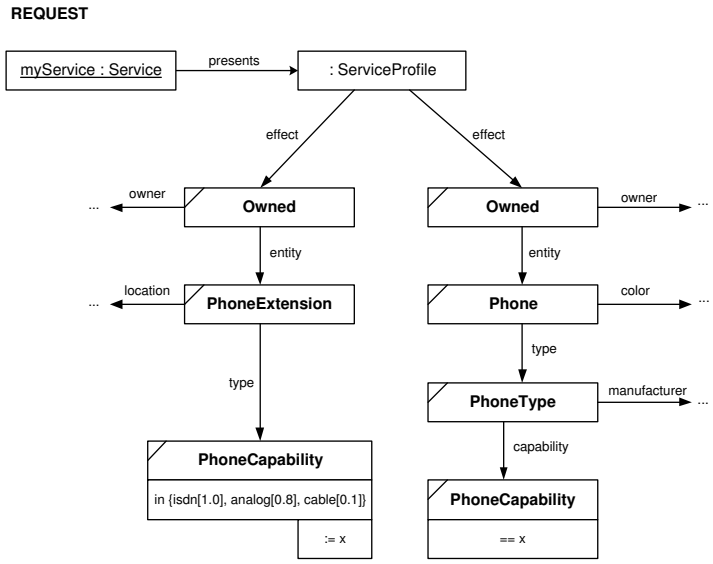


Abbildung 11.2.: Verknüpfte Effekte mit Wertübergabesemantik.

- **Wertübergabesemantik.** In der Anfragebeschreibung existieren spezielle Variablen (nicht zu verwechseln mit DE-Variablen), welche in einer Menge definiert und in einer anderen typischerweise innerhalb einer direkten Bedingung verwendet werden können. Ein Beispiel zeigt Abbildung 11.2. *Zuerst* wird ein bestimmter Telefonanschluss gewünscht und damit ein Typ festgelegt (`:= x`), *anschließend* ein Telefon verlangt, dessen Fähigkeiten diesem Typ entsprechen (`== x`). Durch die getrennte Definition und Verwendung von Variablen wird also eine Reihenfolge unter den Effekten erstellt, welche diese in Haupt- und Nebeneffekte unterteilt. Zu beachten ist, dass eine solche topologische Sortierung nur dann erstellt werden kann, wenn bei Definition und Verwendung keine Zyklen auftreten. Anfragen mit Wertübergabesemantik haben den Vorteil, weitaus effizienter gegen Angebote vergleichbar zu sein, da der Tupelraum nicht insgesamt, sondern die Effekte einzeln, in der gegebenen Reihenfolge verglichen werden können. Zudem ist die Semantik solcher Beschreibungen näher an der Intuition des Dienstnehmers, da häufig ein Primäreffekt (hier der Telefonanschluss) und davon abhängige Sekundäreffekte (hier das Endgerät) unterschieden werden können.

Auch in angebotenen Diensten können mehrere verknüpfte Effekte vorkommen, wenn auch wesentlich seltener als in benötigten Diensten. Ein typisches Beispiel ist ein „Bundle-Angebot“, bei dem der Dienstgeber zwei zueinander passende Effekte im Paket anbietet, etwa eine Hotelreservierung und einen zeitlich passenden Flug an den Ort des Hotels.

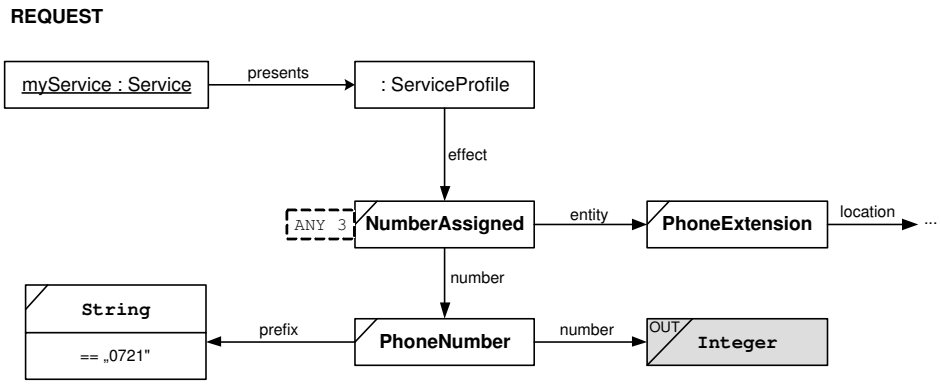


Abbildung 11.3.: Beispiel für die Iterationsdirektive `ANY 3`.

11.1.3. Iterationsdirektiven

Der dritte Fall, der zu einer Kombination angebotener Dienste führen kann, wird in der Literatur bisher quasi nicht betrachtet: die Veränderung der Anzahl der zu erzeugenden Effektinstanzen in der Anfrage. Dies kann durch die Angabe einer *Iterationsdirektive* notiert werden. Wünscht der Dienstnehmer, dass aus einer Effektmenge E nicht nur eine, sondern mehrere Instanzen erzeugt werden, so kann er das durch Angabe einer der folgenden Iterationsdirektiven für eine E kennzeichnen:

- `ALL`. Alle Effekte aus E, die durch Dienste erwirkbar wären, sollen auch erwirkt werden.
- `BEST` n. Aus allen Effekten aus E, die durch Dienste erwirkbar wären, sollen diejenigen n erwirkt werden, die die höchsten Zugehörigkeitswerte zu E besitzen.
- `ANY` n. Aus allen Effekten aus E, die durch Dienste erwirkbar wären, sollen beliebige n erwirkt werden.

Ohne Angabe eine Direktive gilt standardmäßig, dass pro Effektmenge *genau ein* Effekt (nämlich der beste) erwirkt werden soll (siehe Regel 8.104), d.h. implizit ist die *Iterationsdirektive* `BEST 1` spezifiziert.

Ein Beispiel zeigt Abbildung 11.3. Durch die Iterationsdirektive `ANY 3` an NumberAssigned drückt der Dienstnehmer aus, dass er drei Effekte dieser Menge erbracht wissen möchte, d.h. einem Anschluss drei beliebige, aber unterschiedliche Telefonnummern zugewiesen werden. Hierdurch entstehen auch an der ReqOUT-Variable drei Rückgabewerte.

Iterationsdirektiven können als *Quantoren* aufgefasst werden, die festlegen, wie viele Elemente einer Menge betroffen sind. Dabei können die Direktiven `BEST 1` und `ANY`

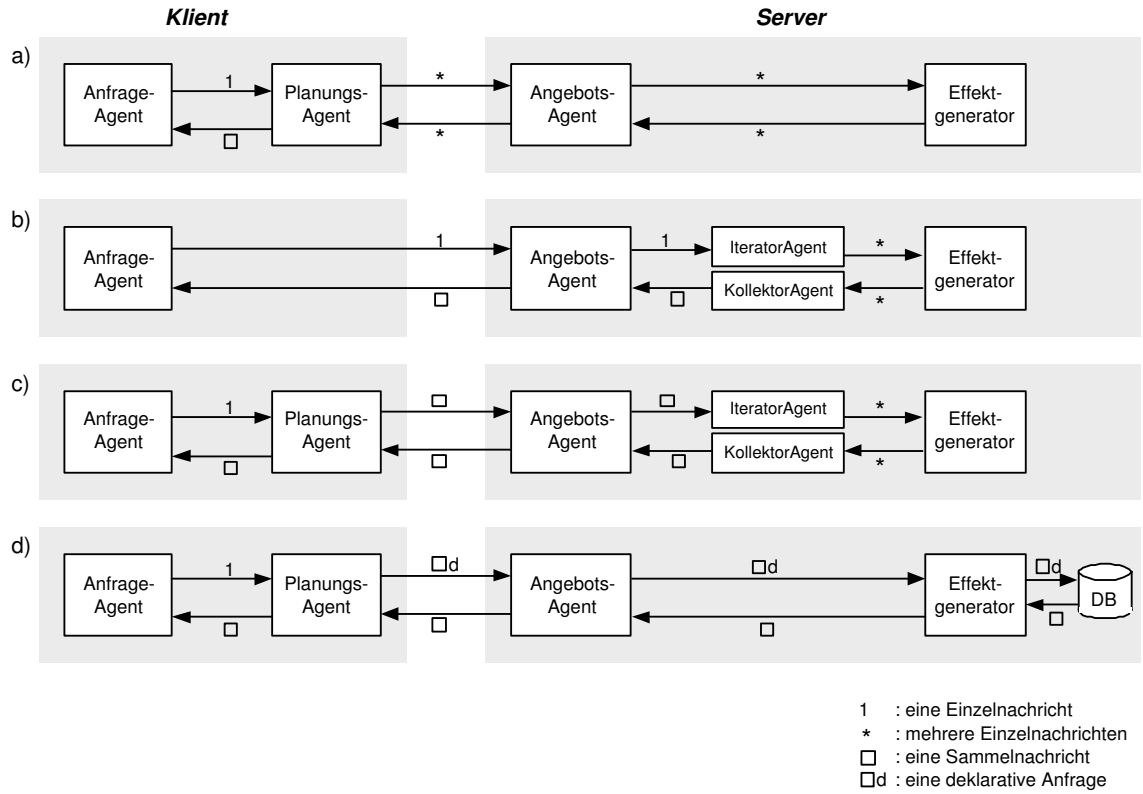


Abbildung 11.4.: Verschiedene Arten zur Erfüllung von Iterationsdirektiven: (a) durch Nutzung von Einzeleffekten, (b) durch serverbestimmte Effektbündel, (c) durch Effektkollektionen mittels enumerierter Variablenbindung oder (d) durch Effektmengen mittels deklarativer Variablenbindung.

1 als Existenzquantoren aufgefasst werden, da genau ein Element gewählt ist; alle anderen Direktiven zählen zu den Allquantoren, da hier mehrere Elemente der Menge betroffen sind.

Die Kommunikation zwischen Dienstnehmer und Dienstgeber wird durch den Einsatz von Iterationsdirektiven komplexer. Im einfachsten, aber auch ineffizientesten Fall führt jeder zu erbringende Einzeleffekt zu einem Aufruf des Dienstgebers. Die Effizienz kann gesteigert werden, wenn der Dienstgeber Dienstnehmern entgegenkommt und angebotene Dienste serverseitig zusammenfasst (vgl. [32]).

Einen Überblick über die einzelnen Ansätze und das daraus resultierende Zusammenspiel zwischen Klient und Server zeigt Abbildung 11.4. Dabei bedeutet eine „1“, dass einmal eine einzelne Nachricht bzw. ein einzelner Aufruf entsteht, ein „$\star$“, dass mehrmals eine Einzelnachricht bzw. ein Einzelaufruf entsteht, ein „$\square$“, dass mehrere Nachrichten bzw. Aufrufe zu einer Einheit zusammengefasst und gemeinsam verarbeitet werden, und „$\square d$“, dass eine deklarative Menge von Nachrichten (d.h. die

Elemente sind nicht explizit aufgelistet) zusammengefasst sind.

- **Einzeleffekte durch eindeutige Variablenbindung.** Der Server fasst keine Effekte zusammen, sondern erlaubt eine eindeutige Auswahl über die Variablen. Um eine Anfrage mit Iterationsdirektiven erfüllen zu können, ist der Anfrageagent auf einen Planungsagenten angewiesen, der den Angebotsagenten mehrfach kontaktiert und die einzelnen Ergebnisse sammelt (siehe Abbildung 11.4a). Der Ansatz hat den Vorteil, dass Dienstanfragen, für die der Planer eine Zerlegung in Einzeldienste kennt, stets durch eine Reihe von einzelnen Dienstnutzungen verarbeitet werden können. Die Iteration findet so auf Seiten des Klienten statt. Durch die hohe Anzahl von Einzelnachrichten wird jedoch das Netz zwischen Klient und Server stark belastet.

- **Serverbestimmte Effektbündel.** Zur Effizienzsteigerung können auch in Angebotsbeschreibungen Iterationsdirektiven zugelassen werden, sodass bei einem Aufruf in einer bestimmten Konfiguration eine Reihe von Effekten entsteht, so genannte Effektbündel. Der Ablauf ist in Abbildung 11.4b dargestellt. Der Angebotsagent erhält nur einen Aufruf und veranlasst daraufhin einen Iteratoragenten, den Effektgenerator mehrmals aufzurufen. Die entstehenden Ergebnisse werden von einem Kollektoragenten aufgesammelt und gebündelt zurückgesandt. Die Iteration findet also auf Seiten des Servers statt. Dies stellt jedoch eine künstliche Vergröberung der angebotenen Dienste dar, da der Dienstnehmer nicht mehr durch entsprechende Konfigurierung der OffIN-Variablen gezielt einzelne Effekte auswählen und daraus selbst die Menge der gewünschten Effekte zusammenstellen kann, wodurch der Planungsagent nicht benötigt wird. Die Wahrscheinlichkeit einer Passung zwischen einer Anfrage- und einer Angebotsbeschreibung mit Iterationsdirektiven wäre demnach sehr gering.

- **Effektkollektionen durch enumerierte Variablenbindung.** Die Vorteile der ersten beiden Verfahren können kombiniert werden, indem Iteration auf Serverseite und feingranulare Konfigurierbarkeit erreicht werden. Dies wird möglich, wenn eine Eingabevariable in einem Angebot für einen Dienstaufruf gleich mit mehreren Werten belegbar ist. Der Dienstgeber muss eine solche *enumerierte Variablenbindung* durch Kennzeichnung der entsprechenden OffIN-Variablen explizit erlauben. Die Füllung einer solchen Variablen kann dann eine DE-Menge sein, die eine direkte Bedingung mit dem Operator `in` enthält. Der Dienstgeber sichert zu, bei einem Aufruf den Effektgenerator mehrfach mit allen enthaltenen Elementen anzustoßen und die Einzelergebnisse gesammelt zurückzusenden. Hierzu verwendet er wieder einen Iterator- und Kollektoragenten (siehe Abbildung 11.4c). Der Klient kann jetzt nach wie vor feingranular Effekte auswählen, zusätzlich wird jedoch das Netz aufgrund der wenigen Nachrichten entlastet.

- **Effektmengen durch deklarative Variablenbindung.** Häufig arbeitet der Effektgenerator mit einer Datenbank, wodurch weitere Optimierungsmöglichkeiten bestehen. Er kann dann für einige seiner Eingabevariablen eine *deklarative Variablenbindung* zulassen, d.h. OffIN-Mengen können mit beliebigen scharfen DE-Mengen gefüllt werden, die selbst wieder eine Iterationsdirektive enthalten können. Wie Abbildung 11.4d zeigt, leitet der Anfrageagent bei einem Aufruf diese Menge direkt an den Effektgenerator weiter. Dieser generiert daraus eine deklarative Anfrage (etwa in SQL), sendet sie and die zugrunde liegende Datenbank und empfängt eine Kollektion von Ergebnissen (etwa ein `ResultSet`). Je nach übermittelter Iterationsdirektive wird dann für einige oder alle Elemente ein Effekt erbracht und das gesammelte Ergebnis an den Klienten gesendet. Die Iteration findet somit sehr effizient im Inneren des Effektgenerators statt. Zudem muss der Klient nicht in der Lage sein, die einzelnen Instanzen auflisten zu können.

11.1.4. Zusammenfassung

Die transparente Kombination angebotener Dienste stellt die wichtigste Erweiterung einer dienstorientierten Architektur dar. Durch sie wird die Anzahl der erfüllbaren Dienstanfragen erheblich gesteigert, insbesondere in Fällen, in denen die Granularität zwischen angebotenen und benötigten Diensten unterschiedlich ist. Für die Kombination von Diensten ist die Beschreibung von zentraler Bedeutung. Sie muss genügend detailliert sein, damit ein Rechner in der Lage ist, daraus selbstständig geeignete Teildienste ableiten zu können. Die Kombination kann dabei entlang verschiedener Aspekte erfolgen: In der Literatur wird häufig nur die *Verkettung* von Diensten betrachtet, bei der jeweils ein Dienstgeber die Voraussetzungen zur Ausführungen eines anderen Dienstes schafft. Im Rahmen automatischer Dienstnutzung hat die Dienstbeschreibung zu erfassen, welche Voraussetzungen der Dienstnehmer bereit ist, durch weitere Dienste erfüllen zu lassen und welche nicht. Wesentlich wichtiger ist die Angabe *mehrerer Effekte* in der Anfragebeschreibung, die unabhängig voneinander oder miteinander verknüpft sein können. Für verknüpfte Effekte lassen sich durch Verwendung einer Wertübergabesemantik im Gegensatz zur Join-Semantik Probleme beim Vergleich lösen. Eine dritte Art, in der Dienstkombination nötig wird, ist die Angabe von Quantoren in Form von *Iterationsdirektiven* in Anfragebeschreibungen, wodurch der Dienstnehmer mehr als einen Effekt pro Effektoperator erwirkt haben möchte. Auf Seiten des Dienstgebers ist die Verwendung solcher Quantoren wenig sinnvoll, da diese Effektbündelung die Granularität künstlich erhöht und positive Vergleichsergebnisse erschwert. Effzienzsteigernd kann hier die Einführung erweiterter Bindungsmöglichkeiten für OffIN-Variablen sein. In solchen Fällen findet die wiederholte Ausführung auf Seiten des Dienstgebers statt, während der Dienstnehmer das aggregierte Gesamtergebnis erhält. In der Praxis kann die Komplexität der Dienstkombination noch höher ausfallen, da die drei Arten der Dienstkombination zusammen auftreten können.

11.2. Erweiterte Ausführung von Diensten

Die Ausführung von Diensten kann an zwei Stellen erweitert werden (siehe Abbildung 11.5): einerseits zwischen Klient und Server (*Choreographie*), andererseits zwischen dem Effektgenerator und möglicherweise zur Bereitstellung seiner Effekte benötigter untergeordneter Dienste (*Orchestrierung*). Im Folgenden werden beide Punkte untersucht und mögliche Erweiterungen für DSD vorgeschlagen.

11.2.1. Choreographie

Der Kommunikation zwischen Klient und Server liegt in einer Architektur basierend auf DSD ein Modell zugrunde, welches von zwei Phasen ausgeht. In der ersten Phase, der Schätzphase, kann der Klient durch synchrone Aufrufe beim Server konkrete Werte beschaffen, um so die endgültige Auswahl eine geeigneten Dienstgebers zu verbessern. In der anschließenden zweiten Phase stößt der Klient einen ausgewählten Dienst an und wartet auf eventuelle Rückgaben. Die gesamte Ausführung geschieht daher mehr oder weniger synchron. Besonders im betrieblichen Umfeld sind jedoch auch eine Reihe von Diensten zu finden, die eine *asynchrone Kommunikation* erwarten und somit auf andere Weise in Anwendungen, Workflows und weitere Dienste integriert werden müssen.

Ein Ansatz, um diesem Umstand gerecht zu werden, ist ein *Warteschlangenmodell*. Hierbei wird die Architektur so erweitert, dass jeder Teilnehmer des Systems eine Warteschlange besitzt, in der Nachrichten anderer Dienstnehmer oder -geber eintreffen und zwischengespeichert werden können. Sie werden dann im Laufe der Zeit sukzessive abgearbeitet oder zu Optimierungszwecken innerhalb der Warteschlange umgeordnet. Eine Skizze der Idee zeigt Abbildung 11.6. Dienste können so über einen längeren Zeitraum und auch verzahnt ablaufen. Die Bedeutung der Nachrichten sowie ihre korrekte Abfolge müssen dann in der Dienstbeschreibung festgehalten werden.

In der in der Arbeit vorgestellten Fassung von DSD ist die Abfolge der auszutauschenden Nachrichten mehr oder weniger „fest verdrahtet“, indem die Choreographie über

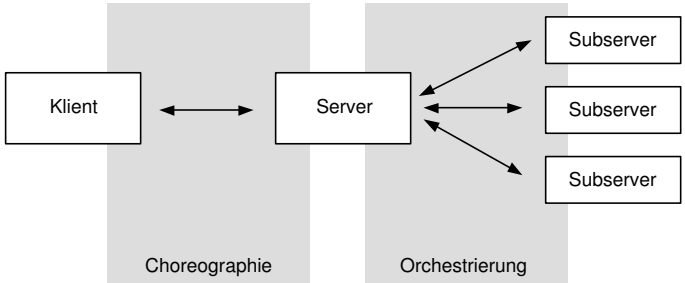


Abbildung 11.5.: Unterscheidung zwischen Choreographie und Orchestrierung.

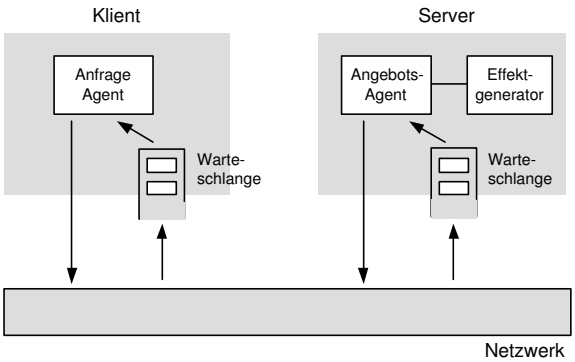


Abbildung 11.6.: Erweiterung der Teilnehmer um Warteschlangen für Nachrichten zur Unterstützung erweiterter Choreographien.

die Semantik der Variablenkategorien ($\mathsf{IN}_{e/x}$, $\mathsf{OUT}_{e/x}$) in die Beschreibung der Funktionalität integriert ist. Um ein komplexeres asynchrones Verhalten (wie etwa für das Warteschlangenmodell) darstellen zu können, müsste diese Information herausgelöst und in einem speziellen Teil der Beschreibung erfasst werden, welcher genauer klärt, wann und von wem bestimmte Nachrichten zu senden und zu verarbeiten sind.

Im Falle einer asynchronen Dienstnutzung sind für die Zeitpunkte, zu denen Nachrichten gesendet bzw. erwartet werden, weitere Beschreibungselemente nötig. Denkbar wären Elemente zur Erfassung eines periodischen Sendens, eines dauerhaften Nachrichtenstroms, eines bedingten Sendens (*immer wenn <Bedingung>, dann <Nachricht>*), eines getriggerten Sendens (*sobald <Bedingung>, dann <Nachricht>*) usw. Es entstehen so neben dem klassischen Kooperationsmuster „Funktionsaufrufer/Funktionsgeber“ weitere Muster wie etwa „Veröffentlicher/Abonnent“ (engl. Publish/Subscribe). Sie sind insbesondere für Befähigungsdienste und langlaufende Dienste von großer Bedeutung.

In der Literatur, insbesondere bei der Prozessbeschreibung in OWL-S, werden komplexe Choreographien häufig durch die Verwendung einer vollständigen Programmiersprache erfasst, die den Ablauf des Nachrichtenaustausches in einer Art Programmcode erfassen. Dieses Vorgehen ist problematisch, da einerseits die hohe Beschreibungsmächtigkeit eine effiziente Vergleichbarkeit gefährdet, andererseits die Integration mit der funktionalen Beschreibung im ServiceProfile erschwert wird.

11.2.2. Orchestrierung

Zur Erbringung seiner angebotenen Effekte kann ein Effektgenerator auf die Unterstützung von Subdienstgebern angewiesen sein, d.h. der Effektgenerator tritt selbst als Dienstnehmer auf. Durch Bereitstellung weiterer Logik soll so ein Mehrwert aus

den bereits existierenden Diensten erstellt werden. Unter *Orchestrierung* versteht man die benötigte Koordination der Kommunikation zwischen Effektgenerator und Subdiensten.

In der Literatur wird die Problematik der Orchestrierung häufig dadurch angegangen, dass Erweiterungen an den eigentlichen Dienstbeschreibungen vorgenommen werden, etwa durch Einführung zusätzlicher programmiersprachlicher Konstrukte oder spezieller Prozessbeschreibungssprachen wie die *Business Process Execution Language for Web Services*[2] *(BPEL4WS)* oder die *Process Specification Language (PSL)* [128]. Für das Auffinden und Nutzen von Diensten ist dies jedoch wenig sinnvoll, im Gegenteil, hierdurch wird das Kapselungsprinzip der Implementierung weitgehend ausgehebelt, indem die Probleme vom Effektgenerator in die Dienstbeschreibung verlagert werden. Geeigneter wäre eine direkte Verwendung der dienstorientierten Middleware aus der Implementierung des Effektgenerators. Hierzu müssten allerdings einige Erweiterungen an deren Funktionalität vorgenommen werden, etwa die Möglichkeit, mehrere Dienstanfragen zusammen transaktional ausführen zu lassen, die Möglichkeit zur Unterscheidung, ob Dienste sequentiell oder parallel ablaufen, sowie die Möglichkeit zur Einstellung von Dienstgütequalitäten bei den Subdienstgebern, auf welche der Effektgenerator zur Erbringung seiner Effekte angewiesen ist.

Erst wenn diese Fragen geklärt sind, kann untersucht werden, ob durch die Verwendung von Subdienstgebern auch die Dienstbeschreibung des eigentlichen angebotenen Diensten erweitert werden muss. In der Tat kann eine dynamische Anpassung der Beschreibung nötig sein, um die aktuelle Verfügbarkeit der Subdienstgeber widerzuspiegeln.

11.2.3. Zusammenfassung

Die Ausführung von Diensten ist gegenüber einer Architektur basierend auf DSD an zwei Positionen erweiterbar. Eine erweiterte *Choreographie* zwischen Klient und Server ist insbesondere für asynchrone Dienste wichtig. Eine Möglichkeit zur Umsetzung bietet ein Warteschlangenmodell. Hierfür müsste die Dienstbeschreibung um eine mächtigere Beschreibung der Nachrichtenfolge ergänzt werden, welche nicht direkt in den Variablen verankert sein sollte. Eine erweiterte *Orchestrierung* zwischen Effektgenerator und Subdienstgebern erfordert kaum Änderungen an der Dienstbeschreibung. Hier muss die von der Middleware angebotene Funktionalität erweitert werden, die dann vom Effektgenerator direkt, d.h. ohne den Umweg einer zusätzlichen Beschreibung genutzt werden kann.

[2] `http://www.ibm.com/developerworks/webservices/library/ws-bpel`

11.3. Dienstnutzung in mobilen Umgebungen

Das zweite Szenario dieser Arbeit (siehe Abschnitt 1.2.2 auf Seite 8) verwendet Dienstorientierung innerhalb einer *mobilen Umgebung*. Als Vision steht ein mobiles Gerät, dessen Leistungsfähigkeit sich nicht von der eines stationären Rechners unterscheidet. Mobile Geräte zeichnen sich jedoch dadurch aus, dass sie in ihrer Leistungsfähigkeit beschränkt sind. PDAs beispielsweise besitzen nur geringe Speicherkapazitäten und auch ihre Prozessorleistung reicht nicht an die von Desktop-PCs heran. Mobiltelefone sind in ihrer Leistung noch weiter eingeschränkt. Um solche Geräte sinnvoll einsetzen zu können, ist es daher nötig, neben der selbst erbringbaren Funktionalität auch auf externe Dienste zurückzugreifen.

Mobile Geräte können jedoch nicht nur Nutzer, sondern auch Erbringer von Diensten sein. Sie lassen sich entlang zweier Dimensionen klassifizieren (siehe Abbildung 11.7):

- *Mobilität.* Es ist zu unterscheiden, ob der angebotene Dienst von einem beweglichen oder einem statischen Gerät bereitgestellt wird.
- *Lokationsabhängigkeit.* Es ist zu unterscheiden, ob die Funktionalität des angebotenen Dienstes einen Bezug zu einem bestimmten Ort besitzt oder nicht.

Die einfachsten Dienste sind nicht-mobile, nicht-lokationsabhängige Dienste wie ein Buchkaufdienst im Internet. Er wird von einem statischen Dienstgeber bereitgestellt

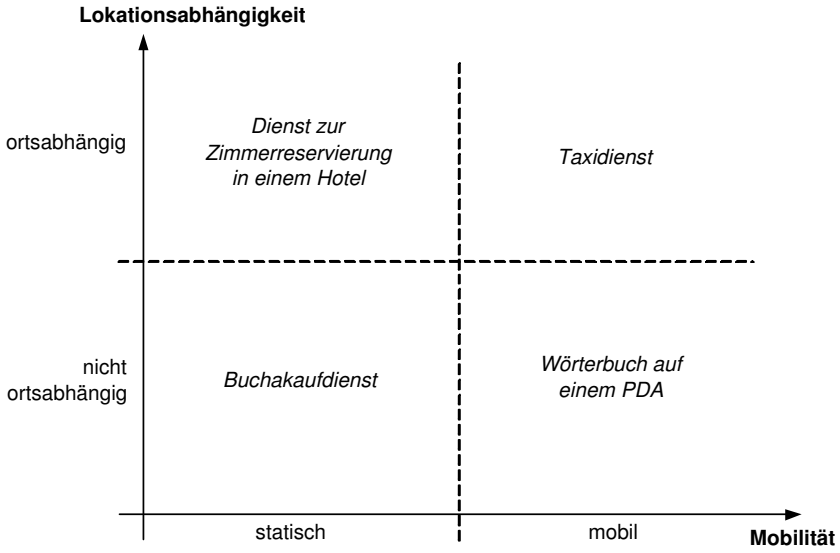


Abbildung 11.7.: Klassifikation angebotener Dienste in mobilen Umgebungen mit Beispieldiensten.

und offeriert eine lokationsunabhängige Funktionalität. Im Gegensatz kann ein Dienst zur Reservierung von Hotelzimmern auf eine bestimmte Stadt beschränkt und damit lokationsabhängig sein. Beispiele für mobile angebotene Dienste könnten ein Wörterbuch auf einem PDA (lokationsunabhängig) oder ein Taxidienst (lokationsabhängig) sein.

In jedem Fall ist jedoch eine dynamische und automatische Bindung von Dienstgebern unerlässlich, da aufgrund der Bewegung der mobilen Geräte, ein Dienstgeber für einen Dienstnehmer nicht mehr erreichbar sein kann und nur eine kontextabhängige Verwendung von Funktionalität nötig ist. Die statische Einbindung von Dienstanbietern verbietet sich, da der Aufenthaltsorts und damit das aktuelle Umfeld der Geräte variabel sein kann. Erforderlich ist eine fallbezogene Auswahl geeigneter Funktionalität zur Laufzeit. Gerade im Falle mobiler Anwendungen ist daher eine vollständig automatisierte Dienstnutzung von großer Bedeutung.

Neben den Vorteilen der Dienstorientierung in mobilen Umgebungen treten hier besondere Schwierigkeiten auf:

- **Kontextbezug.** Bei einem Vergleich von Dienstbeschreibungen muss neben der inhaltlichen Übereinstimmung auch berücksichtigt werden, in welchem Kontext sich Dienstnehmer und Dienstgeber befinden bzw. ob die aktuelle Topologie des Netzes eine sinnvolle Nutzung des Dienstes erlaubt.
- **Dienstvermittlung.** Mobile Netze zeichnen sich durch bewegliche Teilnehmer aus. Es stellt sich daher die Frage, wie Dienstanfragen und Dienstangebot in solchen Umgebungen effizient zueinander finden. Die Verwendung eines zentralen Verzeichnisses, wie dies in den vorangegangenen Kapiteln vereinfacht angenommen wurde, ist für viele mobile Umgebungen nicht geeignet.
- **Dienstausführung.** Bei der Ausführung von Diensten in mobilen Umgebungen ist zu beachten, dass Dienstgeber weitaus häufiger als in statischen Netzen ausfallen oder unerreichbar werden können. Es sind daher Techniken nötig, die (insbesondere bei kombinierten oder langlaufenden Diensten) bereits im Voraus eine möglichst zuverlässige Dienstgeber auswählen und diese bei Fehlerfällen während der Ausführung transparent austauschen.

Die ersten beiden Aspekte werden im Folgenden genauer betrachtet und einige Ansätze für Lösungen vorgestellt.

11.3.1. Kontextbeachtende Dienstnutzung

Durch die Bewegung der Teilnehmer in mobilen Netzen ändert sich deren Kontext ständig, was eine fallweise Dienstbindung zur Laufzeit nötig macht. Um für den je-

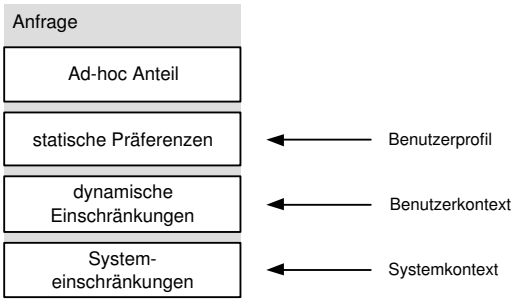


Abbildung 11.8.: Generierung einer Anfragebeschreibung aus einem spontanen Teil und verschiedenen Arten von Kontextinformationen.

weiligen Kontext geeignete Dienstangebote zu finden, muss der Kontext bei der Beschreibung für benötigte Dienste berücksichtigt werden (vgl. [76]). In der Literatur wird die Beschreibung des Kontextes häufig spezifisch auf die jeweilige Anwendung abgestimmt. Ansätze, die generische Lösungen basierend auf ontologischen Beschreibungen bieten, finden sich in [31, 61, 62, 106].

Generell kann der Kontext als Satz zusätzlicher, impliziter Bedingungen innerhalb der Anfrage aufgefasst werden. Es ist daher sinnvoll, aus dem Kontext resultierende Bedingungen nach dem Absenden einer Anfrage automatisch in die Beschreibung einzubetten. Hierdurch wird aus einer rein funktionalen Anfrage eine kontextbeachtende. Der Vergleichsmechanismus muss hierfür nicht angepasst werden.

Abbildung 11.8 zeigt, welche Arten von Kontextinformationen zur Generierung der Anfrage beitragen:

- Das *Benutzerprofil* enthält den statischen Kontext, d.h. die generellen Präferenzen des Dienstnehmers, wie etwa seine bevorzugten Telefonhersteller oder Zahlungsbedingungen. Es kann manuell erstellt oder automatisch aus längerfristigen Beobachtungen des Benutzerverhaltens abgeleitet werden.
- Der *Benutzerkontext* enthält schneller veränderliche Informationen zum Dienstnehmer, etwa seine aktuelle Position oder seine aktuelle Tätigkeit. Aufgrund der hohen Variabilität sollten die Daten automatisch über Sensoren vom System erfasst werden.
- Der *Systemkontext* enthält Informationen zu den aktuellen Eigenschaften der Umgebung, etwa die momentane Auslastung des Netzwerks, zur Verfügung stehende Ein- und Ausgabemöglichkeiten usw. Auch diese Informationen sollten automatisch erfasst werden.

Beginnend mit einer spontanen Anfragebeschreibung, die explizit vom Dienstnehmer erstellt wird und die von ihm gewünschte Funktionalität enthält, werden automatisch

Kontextinformationen aus den drei Gruppen entnommen und als zusätzliche Bedingungen in die Anfrage integriert. Es sollte gelten, dass implizite Bedingungen von expliziten überschrieben werden. Die so entstandene kontextbeachtende Anfragebeschreibung kann dann mit den vorgestellten Verfahren weiter verarbeitet werden.

Auch angebotene Dienste können von ihrem aktuellen Kontext abhängig sein. Dies trifft insbesondere auf mobile Dienste zu. Ihre Beschreibung ist dann jedoch problematisch, wenn die Änderungen in der Umgebungen zu häufigen Anpassungen der Dienstbeschreibung führen. Dies ist insbesondere dann kritisch, wenn die Beschreibung an einen verteilten Dienstvermittlungmechanismus übergeben werden muss. Eine Lösung des Problems könnte die Zerlegung der Beschreibung in zwei Teile sein: Ein statischer Teil enthält die reguläre Beschreibung, während ein dynamischer Teil den aktuellen Kontext des Dienstgebers und den seines Gerätes (seine Position, seine Verfügbarkeit etc.) erfasst. Die beiden Teile könnten dann in getrennten, dafür angepassten Verzeichnissen verwaltet werden. Speziell für DSD könnte der dynamische Teil der Beschreibung auch so hinterlegt werden, dass in einer Schätzphase vor der eigentlichen Dienstnutzung direkt vom Dienstgeber abgefragt und entsprechend berücksichtigt werden kann.

11.3.2. Dienstvermittlung in mobilen Netzen

Gerade in mobilen Umgebungen ist es nicht immer trivial, einen Vermittlungsmechanismus zu finden, der effizient und zuverlässig arbeitet. Der Aufwand hängt im Wesentlichen davon ab, auf welches Maß an Netzinfrastruktur die Vermittlung zurückgreifen kann [75]:

- In *infrastrukturbasierten drahtlosen Netzen* wie WLAN im Infrastrukturmodus, GSM oder UMTS stehen alle Geräte in Kontakt zu einem nicht-mobilen Zugangspunkt, der im Falle der Mobilfunknetze variabel ist. Sinnvollerweise wird die Dienstvermittlung hier von einer nicht-mobilen Komponente übernommen, die vom Zugangspunkt aus erreichbar sind.
- In *infrastrukturlosen drahtlosen Netzen* (so genannten Ad-hoc-Netzen) sorgen keine zusätzlichen Knoten für eine hilfreiche Infrastruktur. Zur Durchführung ihrer Aufgaben sind die Geräte selbst für die Errichtung geeigneter Strukturen verantwortlich. Da sich die Teilnehmer bewegen und jederzeit das Netz durch Abschalten des Gerätes verlassen können, ist grundsätzlich kein Gerät in der Lage, die Aufgaben einer zentralen Dienstvermittlung zu übernehmen. In solchen Netzen muss daher diese Arbeit geschickt auf die Teilnehmer verteilt werden.

Die Vermittlung in infrastrukturbasierten Netzen ist zwar mit einigen Problemen verbunden, insgesamt jedoch eher einfach zu realisieren. Die effiziente, zuverlässige und

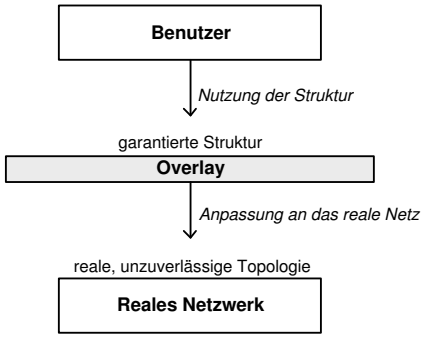


Abbildung 11.9.: Overlay als Vermittler zwischen realem Netzwerk und Benutzer.

semantische Dienstvermittlung in Ad-hoc-Netzen stellt hingegen ein komplexes Problem dar. Als praktikabler Lösungsansatz haben sich *Overlays*, welches eine Zwischenschicht zwischen physikalischem Netzwerk und Benutzer bilden, erwiesen. Generell muss ein solches Overlay so beschaffen sein, dass es einerseits die Dienstvermittlung (also Ankündigung und Suche nach Dienstbeschreibungen) durch Bereitstellung garantierter Strukturen optimal unterstützt und sich andererseits effizient an die ständig verändernde Topologie des physischen Netzes anpasst (siehe Abbildung 11.9). Der Name Overlay rührt daher, dass eine weitere Struktur auf das bereits vorhandene Netz *aufgelegt* wird.

Eine Gruppe stark strukturierter Overlays stellen *semantische Overlays* dar. Bei diesen wird einerseits Wissen über die Inhalte der angebotenen Dienste zur Festlegung der Overlaystruktur verwendet, andererseits Dienstanfragen abhängig von ihrem Inhalt gezielt durch die Struktur geleitet. Vorschläge solcher Strukturen sind *mehrschichtige Cluster* [77] und *Dienstringe* [84]. Bei semantischen Overlays ergibt sich jedoch das Problem, dass sich die starken Strukturbedingungen nur schwer mit der Topologie des realen Netzes in Einklang bringen lassen. Dies trifft insbesondere auf eine Vielzahl von Umgebungen zu, in denen ähnliche Dienste nicht zwangsläufig auch in räumlicher Nähe zueinander zu finden sind. Sinnvoller sind daher schwächer strukturierte Ansätze in Form *halbsemantischer Overlays*. Ihr struktureller Aufbau ist unabhängig von den im Netz angebotenen Diensten, Nachrichten werden jedoch nach wie vor abhängig von ihrem Typ durch das Overlay geleitet. Ein möglicher Vorschlag hierfür ist *Lanes* [83], eine schlanke und gut anpassbare Overlaystruktur, die Bahnen von Knoten bildet.

11.3.3. Zusammenfassung

Dieser Abschnitt hat gezeigt, dass ein dienstorientierter Ansatz für mobile Umgebungen besonders lohnend ist, da die typischerweise leistungsschwachen Geräte so fehlende Funktionalität durch Nutzung externer Dienstgeber komplettieren können.

Besonders wichtig hierbei ist allerdings eine dynamische, fallweise und automatische Auswahl der Dienstgeber, um Ergebnisse zu erzielen, die an den sich häufig ändernden Kontext angepasst sind. Um eine solche Dienstnutzung in mobilen Umgebungen zu verwirklichen, sind eine Reihe weitergehender Fragen zu klären:

- *Kontextbezug.* Wie wird der Kontext des Dienstnehmers und des Dienstgebers bzw. die aktuelle Situation seiner Umgebung beim Vergleich von Diensten berücksichtigt? Als Lösungsvorschlag kann die rein funktionale Anfrage um zusätzliche Bedingungen ergänzt werden, die sich aus verschiedenen Kontextinformationen ableiten lassen. Aus der Beschreibung angebotener Dienste sollten dynamische Teile herausgelöst und getrennt vom Vermittlungsmechanismus zugreifbar gemacht werden.
- *Vermittlung in mobilen Umgebungen.* Wie können Dienstnehmer und Dienstgeber in mobilen Umgebungen effizient, robust und semantisch korrekt zusammengebracht werden? In infrastrukturbasierten Netzen ist dies relativ einfach durch eine auf der Infrastruktur aufsetzende Komponente möglich, etwa ein zentrales Dienstverzeichnis. In Umgebungen ohne Infrastruktur bietet sich die Verwendung von Overlays an, die zwischen den Anforderungen der Teilnehmer und den Gegebenheiten des Netzes vermitteln. Insbesondere halbsemantische Overlays wie Lanes bieten genügend Flexibilität, um ständig an die veränderliche Netzwerktopologie angepasst werden zu können.

11.4. Infrastrukturelle Erweiterungen

Neben Erweiterungen an der eigentlichen Dienstnutzung (also der Kombination, Vermittlung und Ausführung von Diensten) sind auch weitere Arbeiten an der zugrunde liegenden Infrastruktur denkbar, Arbeiten also, die die Verwendung einer dienstorientierten Architektur unterstützen. Im Folgenden sollen weiterführende Arbeiten in drei Gebieten vorgestellt werden:

- **Ontologieverwaltung.** Die Grundlage semantischer Dienstbeschreibungssprachen wie DSD sind Ontologien bestehend aus einheitlichen Schema- und Instanzelementen. Aufgrund der Vielzahl können diese in der Praxis nur verteilt gewartet und gespeichert werden. Eine geeignete Ontologieverwaltung muss sicherstellen, dass alle Teilnehmer des Systems effizient mit allen benötigten Elementen versorgt werden und auch Änderungen daran möglich sind.
- **Vertrauenswürdige Dienstbeschreibungen.** Bei der Erstellung von Dienstangebotsbeschreibungen kann der Dienstgeber bewusst Falschangaben machen, etwa um sich einen Vorteil gegenüber der Konkurrenz zu beschaffen oder um

Dienstnehmer zu betrügen. In der Praxis muss eine dienstorientierte Architektur daher um eine Komponente erweitert werden, die über die Vertrauenswürdigkeit der Teilnehmer wacht. Dies ist insbesondere dann schwierig, wenn diese Aufgabe infrastrukturbedingt nicht von einer zentralen, vertrauenswürdigen Einheit übernommen werden kann.

- **Erstellung von Dienstbeschreibungen.** Die Erstellung von Dienstbeschreibungen ist zeitaufwändig und fehleranfällig. Insbesondere die genaue Wirkung von DSD-Anfragebeschreibungen kann vom Dienstnehmer aufgrund der über die Beschreibung verteilten Verbindungs-, Fehl- und Typvergleichsstrategien nur schwer abgeschätzt werden. Hier sind Verfahren nötig, die Benutzer bei der Erstellung aktiv unterstützen, etwa indem sie eine vereinfachte Sicht auf die Beschreibung bereitstellen.

11.4.1. Ontologieverwaltung

In dieser Arbeit wurde davon ausgegangen, dass die Ontologieinformationen zentral gespeichert sind und jedem Teilnehmer jederzeit zur Verfügung stehen. Allein die Größe verbietet schon eine solche Zentralisierung, zudem werden die Elemente typischerweise von unterschiedlichen, verteilten Expertengruppen oder Einzelpersonen gewartet. In der Praxis wird demnach eine *Ontologieverwaltung* benötigt, die die Ontologieelemente dezentral speichert, die Teilnehmer effizient mit benötigten Elementen versorgt und auch (unter Beachtung der Einigung unter den Teilnehmern) Änderungen daran ermöglicht. Zwei bekannte Ansätze der Literatur, die dieses Ziel ebenfalls verfolgen, sind *KAON* [110] und *SWAP* [35].

Die Verwaltung von Ontologieelementen muss für die verschiedenen Arten unterschiedlich erfolgen:

- *Schemaelemente.* Das Schema stellt eine Einigung der Anwendergruppe dar und ist typischerweise von allgemeinem Interesse. Es wird bei der Definition neuer Instanzen, beim Erstellen von Dienstbeschreibungen sowie beim Vergleich benötigt. Änderungen daran sollten daher nicht von Einzelnen durchführbar sein, sondern nur gemeinschaftlich oder durch Expertengruppen. Hier bietet sich etwa eine internetbasierte Diskussionsplattform in Form eines Wikis[3] an, bei der alle Teilnehmer an der Gestaltung der Schemaelemente mitwirken können (vgl. [129]). Die Speicherung erfolgt dabei verteilt, wobei wichtige und selten veränderte Ontologie (wie etwa `upper`, `category`, `top` und Grundlagenontologien aus `domain`) zum schnelleren Zugriff auf jedem Teilnehmer repliziert werden

[3]Wiki = Sammlung von Internetseiten, die im Gegensatz zu gewöhnlichen Webseiten von allen Benutzern online geändert werden können (siehe `http://www.wiki.org`).

sollten. Generell gilt für die Verwendung von Schemaelementen folgende Regel: Verwendet ein Teilnehmer ein Schemaelement s zur Definition eines anderen Elementes, etwa einer Dienstbeschreibung, so muss er s auf Anfrage bereitstellen oder einen Verweis auf einen externen Speicher liefern können. Dieses reaktive Vorgehen garantiert, dass beim Fehlen einer Schemainformation ein erster Ansprechpartner festgelegt ist. Werden zum Aufstellen einer Dienstbeschreibung zuvor unbekannte Schemaelemente etwa aus anderen Anwendungsgebieten benötigt, sollten diese von der Ontologieverwaltung über gewöhnliche Stichwortsuchen gefunden werden können.

- *Instanzen.* Auch öffentliche Instanzen stellen mit ihrer zentralen Kopie eine Einigung der Anwendergruppe dar. Sie sind teilweise von allgemeinem, d.h. globalem Interesse (wie etwa Maßeinheiten oder große Unternehmen), teilweise nur lokal interessant (wie etwa Telefonanschlüsse). Beschränkt sich das Interesse auf eine Einzelperson bzw. ein Einzelunternehmen, sollten sie als private Instanz deklariert werden und keine zentrale Kopie hinterlegen. Die Verwaltung findet daher je nach Interessensgebiet durch die gesamte Anwendergruppe, eine Expertengruppe oder Einzelpersonen statt. Bei Instanzen gilt der gleiche Grundsatz wie bei Schemaelementen: Die Verwendung einer nicht global verfügbaren Instanz i innerhalb der Definition eines anderen Elements verpflichtet zur Bereitstellung von i. Zuvor unbekannte Instanzen sollten auch hier über eine Stichwortsuche bzw. deklarativ durch Angabe einer DE-Menge aufgesucht werden können.

- *Dienstbeschreibungen.* Beschreibungen von Diensten stellen einmalige, temporäre Elemente dar, die für einen speziellen Anwendungszweck definiert wurden und daher nicht gemeinschaftlich von der Anwendergruppe gewartet werden müssen.[4] Die Verbreitung von Dienstbeschreibungen wird daher nicht von der Ontologieverwaltung übernommen, sondern vom (evtl. verteilten) Dienstverzeichnis im Rahmen der Dienstvermittlung. Der Grundsatz für Schema- und Instanzelemente garantiert, dass eventuell fehlende Beschreibungselemente direkt oder indirekt beim Ersteller nachgeladen werden können.

Die verteilte Speicherung von Ontologieelementen kann durch eine eigenständige Komponente auf jedem teilnehmenden Gerät ermöglicht werden (siehe Abbildung 11.10). Diese operiert in drei „Richtungen“: nach oben stellt sie einem Ontologiebrowser Methoden zur Verfügung, um per Stichwortsuche an Auswahllisten von Ontologieelementen zu gelangen. Dieser Browser kann dann direkt von einem menschlichen Benutzer verwendet werden. Für die Middleware stehen Methoden zur Verfügung, um direkt anhand des Namens oder der Menge bestimmte Elemente zu beziehen. Nach unten legt die Ontologieverwaltung die lokal verfügbaren Elemente in einem

[4] Es ist natürlich denkbar, eine Sammlung beispielhafter Beschreibungen verfügbar zu machen, um so neuen Benutzern die Erstellung von Dienstbeschreibungen zu erleichtern.

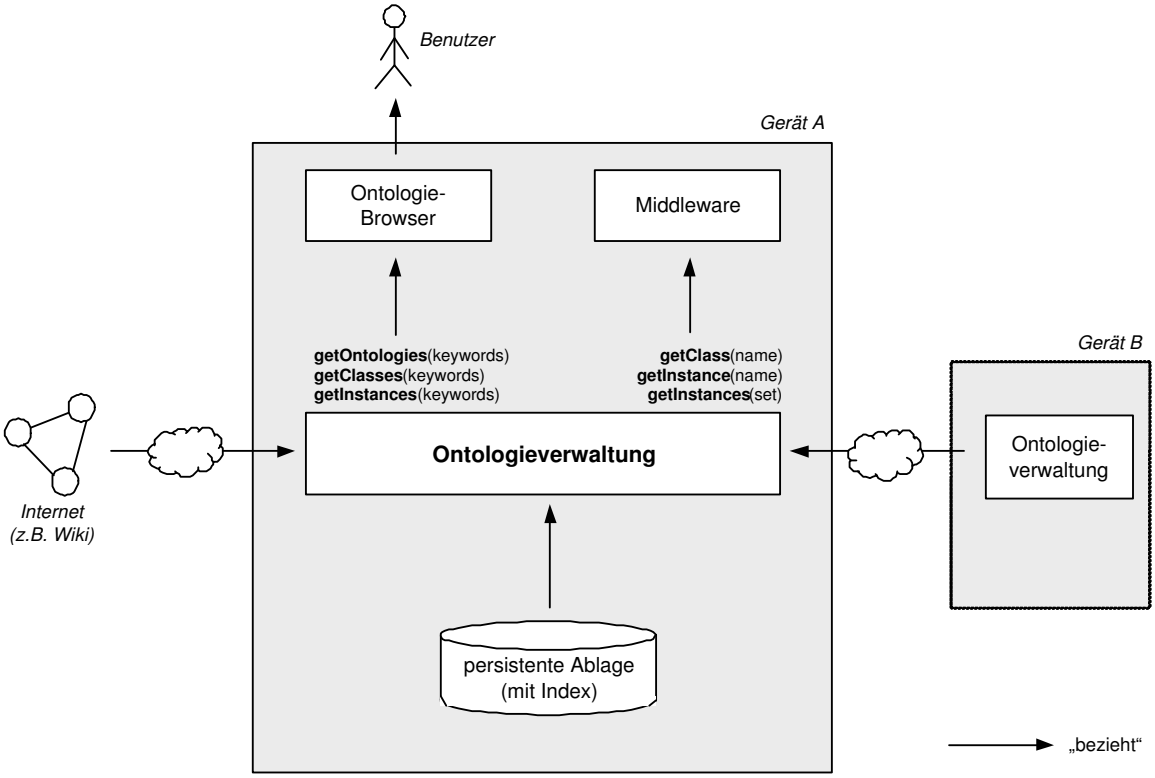


Abbildung 11.10.: Versorgung mit Schema- und Instanzinformationen durch die Ontologieverwaltung.

persistenten Speicher ab, der zur Beschleunigung des Zugriffs typischerweise mit einem Index versehen ist. Horizontal bezieht die Ontologieverwaltung Elemente, die lokal nicht verfügbar sind, einerseits aus gemeinschaftlich verwalteten Ablagen im Internet, andererseits durch Austausch mit Instanzen der Ontologieverwaltung auf anderen Geräten über das jeweilige Netzwerk.

Ein weiteres Problem wird sichtbar: Ist ein Gerät – etwa weil es sich in einem mobilen Netz befindet – nur intermittierend mit dem Internet bzw. anderen Geräten verbunden, so ist ein rein reaktives Vorgehen bei der Beschaffung fehlender Ontologieelemente nicht mehr möglich. Es werden dann Verfahren nötig, die das zukünftige Nutzungsverhalten abschätzen und daraus ableiten, welche Elemente bei noch bestehender Netzverbindung proaktiv einzulagern sind. Im Gegensatz zum reaktiven Caching wird dieser Vorgang als *Hoarding* oder *Prefetching* bezeichnet. Es ist insbesondere dann effektiv, wenn Geräte bei fehlender Verbindung zum Internet auch proaktiv eingelagerte Elemente untereinander austauschen (*kollektives Hoarding*).

11.4.2. Vertrauenswürdige Dienstbeschreibungen

Grundannahme der Arbeit war, dass der Inhalt einer Dienstangebotsbeschreibung die Wirkung des zugehörigen Effektgenerators vollständig und fehlerfrei erfasst, d.h. insbesondere keine bewussten Falschangaben gemacht wurden. In der Praxis kann dies jedoch nicht ausgeschlossen werden. Beispiele für einen solchen Betrug könnten sein, dass bei einem Kaufdienst die Kreditkarte mit einem höheren als dem angegebenen Betrag belastet wird oder bei einem Informationsdienst zwar eine Datei geliefert wird, deren Inhalt jedoch falsch oder mangelhaft ist. Solche Probleme können zu einem Erliegen der dienstorientierten Architektur führen, da die Teilnehmer das gegenseitige Vertrauen verlieren. Nehmen mehrere konkurrierende Unternehmen an einem dienstorientierten System teil, können solche Falschangaben schnell zur Regel werden, wenn sie nicht entsprechend geahndet werden. Eine dienstorientierte Architektur muss also um einen Kontrollmechanismus erweitert werden, der auf einen kooperativen Umgang der Teilnehmer untereinander achtet [112].

Die Grundlage eines solchen Kontrollmechanismus fällt je nach möglicher Schadenshöhe unterschiedlich aus:

- Dienste, bei denen eine bewusst fehlerhafte Ausführung (auf Seiten des Dienstgebers wie auf Seiten des Dienstnehmers) zu *hohen Schäden* führen kann, sollten einer außerhalb des Informationssystems stehenden Gerichtsbarkeit unterworfen sein. Ein Beispiel hierfür ist der oben vorgestellte Kaufdienst, bei dem durch einen Kreditkartenbetrug erhebliche Schäden entstehen können.

- Dienste, deren Falschausführung nur zu *geringen Schäden* führen kann, können nicht sinnvoll einer externen Gerichtsbarkeit unterworfen werden. Die meisten Informations- und Wissensdienste fallen in diese Kategorie. Betrugsfälle müssen innerhalb des Informationssystems ausgehandelt werden, etwa mittels eines Reputationssystems.

Der zweite Fall ist besonders dann kritisch, wenn keine zentrale, dauerhafte und vertrauenswürdige Instanz zur Verfügung steht, die mit der Betreuung dieses Reputationssystems beauftragt werden kann. In einem solchen Fall muss ein verteiltes Reputationssystem verwendet werden. In [111] wurde gezeigt, dass ein solches System tatsächlich möglich ist, wenn es zur Bestimmung der Vertrauenswürdigkeit einzelner Teilnehmer Beweismittel (d.h. nicht-abstreitbare Marken) über getroffene Absprachen und durchgeführte Transaktionen sammelt und auswertet.

Neben der Dienstnutzung muss auch die Vertrauenswürdigkeit bei der Dienstvermittlung sichergestellt werden. Dies ist kritisch, da hier einem Dienstnehmer bewusst Dienstangebote vorenthalten werden können. Eine besondere Problematik entsteht

dann, wenn der Vermittlungsmechanismus verteilt ist und somit eine klare Schuldzuweisung schwierig ist. Eine Möglichkeit, wie etwa das Dienstvermittlungsprotokoll Lanes durch Einführung von nicht-abstreitbaren Marken in seiner Vertrauenswürdigkeit erhöht werden kann, zeigt [113].

11.4.3. Erstellung von Dienstbeschreibungen

Die Erstellung von semantischen Dienstbeschreibungen wie DSD ist aufgrund der Komplexität der ineinander greifenden Sprachkonstrukte auch für Experten nicht immer einfach. Gerade bei einer vollständig automatisierten Verwendung der Beschreibung sowie beim Einsatz von Kontrollmechanismen sind vollständige und fehlerfreie Dienstbeschreibungen jedoch von entscheidender Bedeutung. Der menschliche Nutzer muss daher beim Erstellungsprozess aktiv unterstützt werden. Einen ersten Schritt stellt die Einführung der graphischen Repräsentationsform g-dsd dar, in der ein Benutzer Beschreibungen wesentlich schneller erfassen und bearbeiten kann als in text- oder programmcodebasierter Form. Auch aktive Schablonen (siehe [64]), die dem Nutzer häufig benötigte Aktivitäten (wie etwa das Umwandeln einer Menge in eine Variable, das Hinzufügen von Attributbedingungen zu einer Menge usw.) abnehmen oder teilweise durch den Prozess leiten, können die Erstellung erleichtern.

Insbesondere schwierig ist jedoch das Aufstellen von Anfragebeschreibungen in DSD, da hier das Zusammenwirken der verschiedenen Bedingungs- und Strategietypen nicht immer im Voraus im Detail abzuschätzen sind. Der Vergleicher für DSD verlangt mit präferenzbeinhaltenden Anfragen jedoch genau nach dieser Vollständigkeit. Hier ist eine weitergehende Unterstützung nötig, etwa indem zur Bearbeitung eine vereinfachte Sicht auf die Anfragebeschreibung geboten wird, welche für den Benutzer besser erfassbar ist. Ein Beispiel hierfür könnte die Verwendung beschreibungsweiter, einfacher Prioritäten sein, die vom Benutzer direkt editiert und dann automatisch in Form von Bedingungs- und Strategieangaben auf die einzelnen Mengen der Beschreibung verteilt werden. Eine weitere Möglichkeit stellt das testweise Ausführen von Anfragen bereits zur Programmierzeit dar. Die dabei entstehenden Ergebnisse sollten dann vom Nutzer klassifiziert werden und so zu einer automatischen Adaption der Anfrage in Richtung dieser Auswahl führen (vgl. [158]).

Auch bei der Aufstellung von Angebotsbeschreibungen sind Erweiterungen in Richtung eines höheren Automatisierungsgrads denkbar. Als Vision steht die Idee, dass alle Ressourcen eines Systems (Dateien, Geräte, Geschäftsprozesse, Anwendungen etc.) auf Knopfdruck des Dienstgebers als Dienst öffentlich verfügbar sind, ohne dass vorher manuell eine Dienstbeschreibung erstellt werden muss. Ideen zur Verwirklichung umfassen die Verwendung von Schablonenbeschreibungen, die das System vor der Veröffentlichung geschickt adaptiert, die automatische Bereitstellung von vorlagerbaren Wissensdiensten, die automatische Einbindung von Datenbanken in den

lokalen Instanzenpool sowie die automatische Erstellung des Groundings etwa aus dem Quellcode des Effektgenerators.

11.4.4. Zusammenfassung

In diesem Abschnitt wurden weiterführende Arbeiten im Bereich der Infrastruktur einer dienstorientierten Architektur aufgezeigt, die wichtig für den Betrieb eines solchen Systems sind. Einerseits ist dies eine Komponente zur *Ontologieverwaltung*, die dafür sorgt, dass alle Teilnehmer stets mit von ihnen benötigten Ontologieelementen versorgt werden, obwohl diese dezentral gewartet und gespeichert werden, Komponenten zur Gewährleistung *vertrauenswürdiger Dienstbeschreibungen* auf Basis eines Reputationssystems sowie unterstützende Werkzeuge für menschliche Nutzer zur *Erstellung von Dienstbeschreibungen*, um so einen höheren Grad an Automatisierung und Fehlerfreiheit zu erlangen.

11.5. Fazit

Dieses Kapitel beschäftigte sich mit Arbeiten, die DSD als Beschreibungssprache zum Aufbau einer automatisch und semantisch korrekt arbeitenden Middleware erweitern. Hierbei wurden vier wesentliche Bereiche ausgemacht:

1. Die *Kombination von Diensten*, die durch die Nichterfüllung von Vorbedingungen, durch die Verknüpfung von Effekten sowie durch Verwendung von Iterationsdirektiven nötig werden kann.
2. Die *erweiterte Ausführung von Diensten* in den Bereichen Choreographie und Orchestration.
3. Die Verwendung der Middleware in *mobilen Umgebungen* und die dabei auftretenden Probleme im Bereich der Kontextbeachtung, der Vermittlung und Ausführung.
4. *Infrastrukturelle Erweiterungen*, die wichtig für einen erfolgreichen Betrieb der Architektur sind, etwa die Bereitstellung einer Ontologieverwaltung, einer Komponente zur Gewährleistung vertrauenswürdiger Dienstbeschreibungen sowie Werkzeuge zur Unterstützung bei der Erstellung von Beschreibungen.

Die teilweise aufgezeigten ersten Lösungsansätze haben deutlich gemacht, dass sich DSD aus Ausgangspunkt für die nötige erweiterte Dienstbeschreibungssprache eignet.

12. Zusammenfassung und Ausblick

In diesem Kapitel wird die Arbeit zusammengefasst und ein Ausblick auf mögliche Verbesserungen und Erweiterungen des Ansatzes gegeben.

12.1. Zusammenfassung

Motivation

Dienstorientiertes Rechnen gewinnt als Programmierparadigma zunehmend an Bedeutung. Die Vision ist hierbei ein dynamisches System, in welchem die Teilnehmer in loser Kopplung Funktionalität in Form von Diensten anbieten und nutzen. Diese Dienste werden zur Laufzeit des Systems abhängig von den aktuellen Gegebenheiten gefunden, konfiguriert und aufgerufen. So entstehen große Geschäftsprozesse und mobile Anwendungen, die sich gegenüber anderen Ansätzen durch eine erhöhte Robustheit, Effizienz und Kontextbeachtung auszeichnen, da ungeeignete Dienstgeber dynamisch ausgetauscht werden können. Diese Vision solcher Semantischen Webdienste kann jedoch nur dann Wirklichkeit werden, wenn die Dienstnutzung (bestehend aus Dienstfindung, -konfiguration, und -ausführung) vollständig automatisch und wunschgemäß abläuft. Grundlage hierfür ist eine geeignete Beschreibungssprache, mit welcher die Funktionalität eines Diensts semantisch so präzise ausgedrückt werden kann, dass es einem Rechner ermöglicht wird, passende Angebote und Anfragen ohne menschliche Unterstützung zu erkennen und zu verbinden.

Ziele, Anforderungen und Herausforderungen der Arbeit

Ziel der Arbeit war die Schaffung einer solchen Dienstbeschreibungssprache sowie zugehöriger Verarbeitungswerkzeuge, mit deren Hilfe die Dienstnutzung vollständig automatisiert werden kann (siehe Kapitel 1). Dabei waren folgende Anforderungen zu beachten:

- Die Sprache sollte universal sein, d.h. prinzipiell Dienste aller Domänen erfassen können,

- die Nutzung sollte ohne menschliche Hilfe, also vollständig automatisch geschehen,
- die Nutzung sollte semantisch korrekt ablaufen,
- der initiale Einigungsaufwand sollte in einem akzeptablen Rahmen bleiben,
- die Sprache sollte auch in dynamischen Umgebungen mit einer variablen Dienstlandschaft verwendbar sein.

Die Erreichung der Ziele wurde vor allem durch zwei Paare gegenläufiger Anforderungen erschwert: Ausdrucksmächtigkeit (zur Erfassung aller Dienstarten) gegenüber Verarbeitbarkeit (insbesondere des Vergleichs) sowie Flexibilität (zur Erfassung aller Domänen) gegenüber Strukturiertheit (zur Erreichung der unabhängigen Erstellbarkeit von Beschreibungen).

Stand der Forschung

Zunächst wurden in Kapitel 2 die Grundlagen der beiden Ausgangsgebiete vorgestellt. *Webdienste* bieten mit der Möglichkeit, öffentliche Funktionalität durch Nachrichtenaustausch über das Internet zu nutzen, die Grundlage dienstorientierter Architekturen. Das *Semantische Web* strebt eine semantische Beschreibung von Inhalten im Internet an, um diese auch Rechnern verfügbar zu machen. Grundlage dieser Beschreibungen sind Ontologien, also formale und geteilte Vokabularien für Sachverhalte in der Welt, die durch verschiedene Ontologiesprachen ausgedrückt werden können.

In Kapitel 3 wurden dann eine Reihe von konkreten Ansätzen der Literatur betrachtet, die ähnliche, meist wesentlich erweiterte Ziele wie diese Arbeit verfolgen. Um sie besser miteinander vergleichen zu können und gezielt ihre Stärken und Schwächen darzulegen, wurden zunächst 13 Anforderungen hergeleitet, die für eine Sprache erfüllt sein müssen, damit sie die gesteckten Ziele erreichen kann. Hierunter fallen unter anderem die Trennung in eine eindeutige Angebots- und deterministische Anfragebeschreibung, eine zugrunde liegende formale Semantik, ein effektiver und effizienter Vergleich, die unabhängige Erstellbarkeit von Beschreibungen sowie eine Evaluation, die die Tauglichkeit für die Praxis belegt. Es stellte sich heraus, dass sowohl die beiden größten und einflussreichsten Ansätze OWL-S und WSMO als auch kleinere oder neuere Ansätze wie SWSF, METEOR-S, IRS-III und WSDL-S viele der Anforderungen nicht erfüllen. Gründe hierfür können nicht nur in den vorgeschlagenen Ontologien zu suchen sein, sondern sind tiefer gehend: Die zugrunde liegenden Ontologiesprachen sind aus mehreren Gründen für die Beschreibung von Diensten nicht geeignet: Erstens stellen sie keine geeigneten und effizienten Schlussfolgerungsoperationen zur Verfügung, wie sie für einen sinnvollen Vergleich benötigt würden, zweitens

beschränkt sich ihre formale Semantik meist nur auf die allgemeinen, dienstunspezifischen Teile, drittens ist eine Modellierung in ihnen sehr komplex und unintuitiv und viertens eröffnet ihre hohe Expressivität zusammen mit den oft sehr generischen Dienstontologien einen gewaltigen Spielraum bei der Beschreibung von Diensten, wodurch die unabhängige Erstellbarkeit unmöglich wird.

Ansatz der Arbeit

Der Ansatz dieser Arbeit (überblicksartig vorgestellt in Kapitel 4) stellte daher nicht eine weitere Ontologie zur Beschreibung von Diensten dar, sondern passte auch die zugrunde liegende Ontologiesprache speziell auf die Charakteristika von Dienstbeschreibungen an. Die Arbeit operierte daher auf drei Ebenen:

- Die Schaffung einer grundlegenden Ontologiesprache (*DIANE Elements I*), mit der ein Vokabular zur Beschreibung der Welt erstellt werden kann. Um für die Verwendung innerhalb von Dienstbeschreibungen verwendbar zu sein, wurde sie um Modellierungsrichtlinien und Metaeigenschaften erweitert.
- Die Erweiterung der Ontologiesprache um neue Sprachelemente (*DIANE Elements II*), die aus den Besonderheiten von Diensten abgeleitet sind und exklusiv in Dienstbeschreibungen verwendet werden.
- Die Schaffung von Ontologien zur Erfassung von Diensten (*DIANE Service Description*). Diese basieren auf DE-I und den neuen Sprachelementen aus DE-II.

Die Details zur grundlegenden Ontologiesprache **DIANE Elements I** wurden in Kapitel 5 vorgestellt. *Modellierungsgrundlage* ist im Gegensatz zu den logiksprachenbasierten Ansätzen die klassische Objektorientierung, mit der ein einheitliches, modulares und öffentliches Schema erstellt werden soll. Dabei sind eine Reihe von *Modellierungsrichtlinien* zu beachten, durch welche Ontologien entwickelt werden, die einerseits einen moderaten Einigungsaufwand haben, andererseits ordnend auf die unabhängige Erstellung von Beschreibung wirken. Beispiele für solche Modellierungsrichtlinien sind: Zentrale Ontologien sind anwendungsneutral zu erstellen, ihre Klassen stellen rigide Konzepte dar und enthalten nur intrinsische Eigenschaften. Zusätzlich sind die erfassten Konzepte mit *Metaeigenschaften* zu annotieren. Neu eingeführt wurde die Unterscheidung in wertbestimmte und Entitätsklassen, öffentliche und private Klassen (und damit zusammenhängend die Unterscheidung in lokale und zentrale Kopien von Instanzen) sowie die Unterscheidung in definierende, ableitbare und orthogonale Attribute. Als Syntax wurden drei Repräsentationsformen vorgestellt, die je nach Einsatzgebiet verwendet werden können.

Die neuen Sprachelemente der **DIANE Elements II** bilden das Kernstück der Arbeit und wurden in Kapitel 6 eingeführt. Sie wurden aus den charakteristischen Eigenschaften von Diensten abgeleitet und in operationale, aggregierende, selektierende und bewertende Elemente unterschieden:

- Dienste erbringen eine Wirkung in der Welt. Aus diesem Grund wurden operationale Sprachelemente in Form zweier Operatoren eingeführt, mit deren Hilfe die benötigten Vorzustände sowie die erreichbaren Nachzustände eines Dienstes erfasst werden können.
- Dienste können mehr als einen Effekte erbringen bzw. mit mehr als einem zufrieden sein. Aus diesem Grund wurden aggregierende Sprachelemente in Form deklarativer Mengen eingeführt. Ihre Definition erfolgt strukturiert über drei Bedingungstypen sowie drei Strategien, welche insgesamt die Berechenbarkeit der Operationen auf den so erstellten Mengen garantieren.
- Dienste erlauben die Auswahl oder Einschränkung der von ihnen zu erbringenden Effekte über den Austausch von Nachrichten. Aus diesem Grund wurden selektierende Elemente in Form von Variablen eingeführt. Variablen müssen – in Abhängigkeit von ihrer Kategorie – im Laufe der Dienstnutzung gefüllt werden, was den Informationsfluss zwischen Dienstnehmer und Dienstgeber beschreibt.
- Dienste und die von ihnen erbrachten Effekte können unterschiedlich gut zur Bewältigung einer Aufgabe geeignet sein. Aus diesem Grund wurden bewertende Elemente in Form unscharfer Mengen eingeführt, die in Dienstanfragebeschreibungen integriert werden können, um die Präferenzen des Dienstnehmers genau festzulegen.

Alle Elemente aus DE-II besitzen sowohl eine eigenständige generische Semantik als auch eine erweiterte dienstspezifische Semantik, wenn sie innerhalb einer Dienstbeschreibung eingesetzt werden. Durch die Kombination der Elemente innerhalb einer Dienstbeschreibung entstehen somit neuartige Dienstbeschreibungen:

- Durch die Mischung von Mengen mit Variablen entstehen konfigurierbare Mengen. Werden diese in Dienstbeschreibungen zur Erfassung der Vor- und Nachzustände bei operationalen Elementen verwenden, ergeben sich *rein zustandsorientierte Dienstbeschreibungen*. Im Gegensatz zu existierenden Ansätzen betrachten sie das Ein-/Ausgabeverhalten eines Dienstes nicht getrennt von seiner erwirkbaren Zustandsänderung, sondern sehen die Ein- und Ausgaben vielmehr als Schnittstelle, über die die gewünschte Wirkung eingestellt und abgelesen werden kann. Daher kann auf eine separate Darstellung der Ein- und Ausgaben verzichtet und ein Dienst ausschließlich über seine benötigten und erwirkbaren

Zustände beschrieben werden. Eine Dienstangebotsbeschreibung besteht daher aus deklarativ beschriebenen, konfigurierbaren Mengen von benötigten Vor- und erwirkbaren Nachzuständen.

- Werden die benötigten Effekte einer Dienstanfragebeschreibung mittels bewertender Elemente über unscharfe Mengen beschrieben, entstehen *präferenzbeinhaltende Anfragebeschreibungen*. Im Gegensatz dazu beschreiben existierende Ansätze die benötigte Funktionalität als perfekten Wunschdienst. Da dieser selten exakt zu finden ist, überlassen sie einem auf Heuristiken basierenden Allzweckvergleicher die Entscheidung, welches Dienstangebot geeignet ist. Ein solches beeinflusstes Vergleichsergebnis wird jedoch der Dienstnehmer ohne spätere Einflussmöglichkeit nicht akzeptieren. Können die Präferenzen des Anfragers bezüglich verschiedener Effekte jedoch vollständig in die Anfrage integriert werden, kann der generische Vergleicher durch einen persönlichen Vergleicher ersetzt werden, der exakt nach den Vorstellungen des Anfragers handelt. Eine Dienstanfragebeschreibung besteht daher aus deklarativ beschriebenen, unscharfen (und evtl. konfigurierbaren) Mengen von gewünschten Nachzuständen.

Die einzelnen Sprachelemente aus DE-I und DE-II werden in der **DIANE Service Description** (vorgestellt in Kapitel 7) so kombiniert, dass daraus gültige und semantische eindeutige Dienstbeschreibungen entstehen. Dazu werden in DSD verschiedene Ontologien geschichtet: Eine obere Dienstontologie legt den generellen Aufbau sowie die zu verwendenden Elemente einer Dienstbeschreibung fest, Kategorieontologien erfassen systematisch den Raum der Zustände, die durch Dienste verändert werden können, und in eine obere Ontologie eingebettete Domänenontologien stellen das Vokabular für die betrachteten Anwendungsgebiete zur Verfügung.

Die Semantik einer Dienstbeschreibung leitet sich hauptsächlich aus den darin verwendeten Sprachelemente aus DE-II ab, die eine zusätzliche dienstspezifische Semantik erlangen. Insgesamt können Angebotsbeschreibungen so als Verträge aufgefasst werden, die einen Schätzungsteil, einen Bedingungsteil (bestehend aus Informations- und Zustandsbedingungen) und einen Leistungsteil (bestehend aus Informations- und Zustandsleistung) umfassen. Dienstanfragebeschreibungen hingegen werden als Wunsch aufgefasst, der in einen Informationswunsch und einen Zustandswunsch zerfällt. Insgesamt durchlaufen Dienstbeschreibungen im Laufe ihrer Verwendung im Rahmen einer Dienstnutzung einen Lebenszyklus, in dem die einzelnen Teile zum Einsatz kommen und Schritt für Schritt konfiguriert werden.

Formal wurde die Semantik von DE-I, DE-II und DSD in Kapitel 8 axiomatisch eingeführt. Dazu wurden alle Elemente auf die modale, temporale Prädikatenlogik erster Ordnung mit Identität abgebildet, da für sie bereits eine formale Semantik existiert. Für DE-I wurden insbesondere die Metaeigenschaften, für DE-II die Zugehörigkeit zu einer konfigurierbaren Menge und für DSD die Auffassung von Dienstbeschreibungen als Verträge und Wünsche erfasst.

Mit dem **Vergleich** solcher Dienstbeschreibungen befasste sich Kapitel 9. Er unterschied sich grundlegend von Ansätzen in der Literatur, wo der Vergleich von Dienstbeschreibung auf die vorhandenen Schlussfolgerungsoperationen der zugrunde liegenden Sprache zurückgeführt wird, was zu unintuitiven oder ungenauen Ergebnissen führen kann. Die eigentliche Grundfrage des Vergleichers gerät dabei in den Hintergrund: Welcher angebotene Dienst kann so konfiguriert werden, dass er in jedem Fall einen Effekt erbringt, der vom Anfrager möglichst stark präferiert wird? In DSD hingegen ist die Bewertung der möglichen Effekte bereits in der Anfrage enthalten, sodass keine Ähnlichkeitsberechnung zwischen Anfrage- und Angebotsbeschreibung durchgeführt werden muss, sondern die Zugehörigkeit eines Angebots zu den Präferenzen des Dienstnehmers ermittelt werden kann. Dieser Vergleichswert ist allein durch die Anfrage bestimmt und daher unverzerrt, da keine Heuristiken verwendet werden, die dem Dienstnehmer unbekannt sind. Auf diese Weise entsteht für jede Anfrage ein *persönlicher Vergleicher*. Auch die kombinierte Überprüfung von Signatur und Spezifikation unterscheidet sich von Vergleichsverfahren der Literatur. Neben der Berechnung des Vergleichswertes wird also explizit auch die dazu nötige Konfiguration der Variablen bestimmt. Durch die gemeinsame Betrachtung können die Signaturen zweier Dienste differieren und dennoch zueinander passen. Der Vergleicher hat die Möglichkeit sich weitere Informationen über ein Dienstangebot durch Nutzung einer vorangestellten Schätzphase zu beschaffen. Dem Vergleicher kann zudem ein verteilt arbeitender Vorvergleicher vorgeschaltet werden, der die zu einer Anfrage definitiv unpassenden Angebotsbeschreibungen herausfiltert.

Die Arbeitsweise des Vergleichers unterteilt sich in mehrere Schritte:

1. Zunächst werden die Eingabevariablen der Angebotsbeschreibung optimal belegt. Sie stellen die Informationsbedingungen des Dienstgebers dar. Dabei ist zu beachten, welche Variablen isoliert betrachtet werden können (Standardkontext) und welche gemeinsam mit anderen optimiert werden müssen (erweiterter Kontext).

2. Anschließend werden die Vergleichswerte für die besten Belegungen berechnet. Dazu werden die in der Angebotsbeschreibung auftretenden Mengen in einer Beschreibung in drei Stufen unterteilt und entsprechend behandelt. Die Stufen geben an, ob die Instanzen der Menge implizit bekannt sind, vollzählig aufgelistet werden können oder nicht bekannt sind.

3. Im Anschluss daran werden die Ausgabevariablen der Angebotsbeschreibung belegt. Sie stellen die Informationswünsche des Dienstnehmers dar.

4. Abschließend wird überprüft, ob die Zustandsvorbedingungen der Angebotsbeschreibung erfüllt sind.

Der Vergleicher ist die zentrale Komponente einer dienstorientierten Architektur. Seine Existenz für DSD zeigt, dass die Expressivität von DE und DSD trotz der neuen Sprachelemente beherrschbar bleibt. Der Vergleicher liefert wichtige Informationen für die nachfolgenden Komponenten, welche auf dieser Grundlage die Dienstnutzung automatisieren können.

Evaluation

In der Evaluation in Kapitel 10 wurde überprüft, ob die Ziele und Anforderungen der Arbeit mit Dienstbeschreibungen auf Basis von DSD erreicht werden können. Dabei wurde in zwei Schritten vorgegangen: In der *inneren Evaluation* sollte gezeigt werden, dass die Dienstnutzung tatsächlich automatisiert werden kann, wenn als Vorbedingung semantisch korrekte DSD-Beschreibungen vorliegen; die *äußere Evaluation* sollte dann zeigen, dass solche Beschreibungen für realistische Dienste tatsächlich und unabhängig voneinander erstellbar sind. Für die innere Evaluation wurden zwei Tests durchgeführt: Test 1 untersuchte die Effektivität des Vergleichers und zeigte, dass ein Vergleichsvorgang genügend Informationen liefert, um darauf aufbauend die Nutzung des Dienstes zu automatisieren. Eine entsprechende Middleware basierend auf einer indirekten Kommunikation zwischen Dienstnehmer und Dienstgeber wurde vorgestellt, die insbesondere die vom Vergleicher berechneten Variablenbelegungen umsetzt. Test 2 überprüfte die Effizienz des Vergleichers und zeigte, dass dieser in einer prototypischen Implementierung auf einem Standardrechner für 1.000 realistische Vergleiche im Mittel nur 2 bis 3 s benötigt und einen einzelnen positiven Vergleich in durchschnittlich 80 ms berechnet. Auch für die äußere Evaluation wurden zwei Tests durchgeführt: Test 3 überprüfte die Flexibilität der Sprache, indem realistische Dienste von Experten in semantisch korrekten DSD-Beschreibungen erfasst werden mussten. Der dazu benötigte initiale Aufwand zur Definition des Vokabulars war aufgrund der schlanken objektorientierten Grundlage mäßig. Für die wichtigen Applikationsdienste konnten quasi alle (98%) der vorgegebenen Dienste direkt oder indirekt umgesetzt werden. Test 4 überprüfte abschließend die unabhängige Erstellbarkeit von Beschreibungen. Experimente mit sechs Nicht-Experten, welche zuvor in DSD geschult und dann in zwei isoliert arbeitende Gruppen eingeteilt wurden, ergaben eine Präzision von 1 und einen Recall von 0.7 für Angebots- und Anfragebeschreibungen desselben Dienstes. Diese Werte sind für die Praxis sicher bereits ausreichend, dennoch wäre eine weitere Verbesserung des Recalls wünschenswert. Auch sollten Experimente mit größeren Teilnehmerzahlen die Werte bestätigen.

Zusammenfassend zeigten die vier Tests, dass mit DSD-Beschreibung die Ziele der Arbeit erreicht sind: Realistische Dienste (unter den Einschränkungen aus Abschnitt 1.5) können unter akzeptablen Aufwand in DSD erfasst werden (unter Abstrichen auch unabhängig voneinander). Ein Vergleicher kann dann daraus effizient Vergleichsergebnisse berechnen, die zur vollständig automatischen Nutzung der ausgewählten

Dienste verwendet werden können. Die 13 Anforderungen, die auch zum Vergleich der Ansätze der Literatur verwendet wurden, waren für DSD weitestgehend erfüllt. DSD stellt somit eine geeignete Grundlage für Automatisierung der Dienstnutzung dar.

12.2. Ausblick

Mit Dienstbeschreibungen in DSD können bestimmte Dienste semantisch korrekt beschrieben und in dynamischen Umgebungen automatisch genutzt werden. DSD kann daher als geeignete Grundlage für weitere Forschungen auf diesem Gebiet verwendet werden. Einige weiterführende Arbeiten wurden bereits in Kapitel 11 vorgestellt. Sie umfassten die Komposition mehrerer Dienstangebote, um die Wünsche einer Dienstanfrage erfüllen zu können, die Betrachtung der erweiterten Ausführung von Diensten in den Bereichen Choreographie und Orchestration, die Verwendung des Ansatzes in mobilen Umgebungen, in denen der Kontext eines Dienstes eine wichtige Rolle spielt, sowie infrastrukturelle Erweiterungen, die für den Betrieb einer dienstorientierten Architekturen wichtig sind, etwa eine Ontologieverwaltung, Systeme, die vertrauenswürdige Dienstbeschreibungen garantieren, und Werkzeuge, die menschliche Benutzer bei der Erstellung von DSD-Beschreibungen unterstützen sollen. Im Folgenden werden noch einige weiterreichende Ideen und Visionen vorgestellt, wie DSD direkt verbessert oder für zukünftige Anwendungen ausgebaut werden könnte.

Zunächst sind Erweiterungen an der **Expressivität** von DSD denkbar. Hierzu muss jedoch genau untersucht werden, welche zusätzlichen Konstrukte die Berechenbarkeit des Vergleichsvorgangs von DSD gefährden oder unverhältnismäßig verlangsamen könnten. Dies könnte durch eine Abbildung von DE/DSD auf eine existierende Logiksprache (wie etwa auf eine Variante von OWL oder WSML) geschehen, für die die Grenzen der Berechenbarkeit recht gut bekannt sind. Zu bedenken ist jedoch, dass für DE nicht zwingend generische Schlussfolgerungsoperationen zur Verfügung stehen müssen, sondern spezialisierte Implementierungen für das vom Vergleich benötigte `best-subset` ausreichen. Mögliche Erweiterungen an DE/DSD sollten insbesondere im Bereich der Typvergleichsstrategie (etwa durch Zulassung von mehrerer Typen, die nicht auf einem Pfad in der Vererbungshierarchie stehen, in einer Menge), im Bereich der Verbindungsstrategie (etwa durch Einführung eines Bedingungsoperators `if-then-else`), im Bereich der Ausdrückbarkeit von Abhängigkeiten zwischen Instanzen verschiedener Mengen und im Bereich der Wertübergaben zwischen Mengen angesetzt werden. Bei den Erweiterungen ist stets auch die Relevanz für die Praxis im Auge zu behalten. Die Ergänzung um Konstrukte, die nur für seltene Spezialfälle nötig sind, durch welche die Algorithmen jedoch erheblich komplexer werden, sollten kritisch betrachtet werden.

Neben der Erhöhung der Expressivität einer einzelnen Beschreibung sollte auch untersucht werden, ob nicht eine Zusammenfassung mehrerer Beschreibungen zu einer Gesamtbeschreibung zu verbesserten Resultaten führen könnte. In solchen **mehrdimensionalen Beschreibungen** wäre ein einzelner (benötigter oder angebotener) Dienst dann durch mehrere, unterschiedliche Beschreibungen erfasst. Die wichtigste Dimension stellt sicherlich der Abstraktionsgrad der Beschreibung dar. Auf höchstem Abstraktionsniveau könnte ein Verkaufsdienst dann zum Beispiel sehr generisch dargestellt werden, etwa ohne seine konkreten Ein- und Ausgaben zu nennen. Auf mittlerer Ebene könnten die Beschreibung konkreter werden und etwa auch die benötigten Eingaben zusammengefasst aufgeführt werden (etwa, dass als Eingabe ein Buch ausgewählt werden muss). Erst auf niedrigster Ebene erfolgt dann eine sehr detaillierte Beschreibung, die auch konkret Bezug auf auszutauschende primitive Datentypen nimmt (etwa, dass entweder Autor und Titel des Buches als Strings oder seine ISBN als Integer erwartet werden). Die einzelnen Stufen könnten durch Transformationsvorschriften miteinander verbunden sein (etwa eine Vorschrift, dass man von einem allgemeinen Buch zu dessen Autor und Titel kommt, indem man in einer bestimmten Datenbank nachschlägt), sodass auch eine Passung zweier Dienstbeschreibungen auf höherer Ebene in bestimmten Fällen und unter Zuhilfenahme der Transformationsvorschriften zu einer Abbildung zwischen den konkreten Diensten auf unterster Ebene führen kann, was eine automatische Ausführung ermöglichen würde (z.B. könnte eine Passung zwischen allgemeinen Telefonkaufdiensten vermittelt werden, auch wenn diese verschiedene Währungen oder Zahlungsmodalitäten besitzen/erwarten). Eine weitere Dimension könnte für Anfragebeschreibungen eingeführt werden. Für einen konkreten Effektwunsch könnte eine Reihe von Alternativen möglich sein, die weit vom ursprünglichen Effektwunsch entfernt sind und daher nicht mehr ohne Weiteres in einer gemeinsamen DSD-Abfragebeschreibung erfasst werden können (z.B. soll beim Scheitern eines Bahnticketkaufs versucht werden, eine Mitfahrgelegenheit zu finden). Solche Alternativbeschreibungen können dann zur vollständigen Abbildung eines Wunsches zusammengefasst werden. Beim Scheitern der ersten enthaltenen Anfragebeschreibungen wird auf die zweite zurückgegriffen usw.

Weiterhin könnte überprüft werden, ob die relativ strenge Anforderung nach einer **gemeinsamen Ontologie** fallen gelassen oder zumindest abgeschwächt werden könnte. Besonders in spontan gebildeten Umgebungen wie Ad-hoc-Netzen wäre das von großer Wichtigkeit. Das Mediatorkonzept aus WSMO stellt hier einen möglichen Lösungsweg dar, jedoch müsste darauf geachtet werden, dass der Einigungsaufwand nicht in viel größerem Maße bei der Erstellung der Mediatoren anfällt. Problematisch ist der Ansatz, bereits zur Entwurfszeit einen Mediator für die Vermittlung zwischen je zwei Ontologien anzugeben, da so für n Ontologien $\binom{n}{2} \in O(n^2)$ Mediatoren zu entwickeln sind. Andererseits verschiebt eine Vermittlung über eine zentrale abstrakte Sprache das Problem nur, da dieses Sprache in einem Einigungsprozess entstehen muss. Sinnvoll sind daher nur Mediatoren, die zur Laufzeit und selbstständig (oder zumindest unter sehr geringem Aufwand für einen menschlichen Benutzer) eine Abbildung

zwischen Ontologien vornehmen können, d.h. die Mediatorspezifikation zumindest semi-automatisch generieren können (vgl. [87]). Möglicherweise können Mediatoren bei ihrer Arbeit unterstützt werden, indem die Schemata mit zusätzlichen Metaeigenschaften annotiert werden, was erste Anhaltspunkte oder Ausschlusskriterien für einen Mediation liefern kann.

Zukünftig könnten auch Überlegungen bezüglich der **Kategorieontologien** angestellt werden. Die Semantik ihrer Zustandsklassen beziehen sie in der vorliegenden Arbeit allein aus dem geteilten Verständnis innerhalb der Anwendergruppe. Gerade bei Test 4 der Evaluation nach unabhängiger Erstellbarkeit von Beschreibungen führten Unklarheiten bezüglich dieser Semantik von Zuständen zu einem nicht optimalen Wert für den Recall. Es stellt sich die Frage, ob nicht auch die Zustandsklassen selbst (evtl. zusammen mit den von ihnen beschriebenen Entitäten) über eine formal definierte Semantik verfügen müssten, in welcher hinterlegt ist, was genau durch den Zustand gemeint ist. Für die verschiedenen Unterklassen von Possession wie Owned oder Rented könnte etwa formal hinterlegt werden, dass diese zum Teil zeitlich begrenzt sind, das Nutzungsrecht weitergegeben werden darf usw. Zusätzlich könnten Implikationen zwischen Zuständen angegeben werden (etwa, dass der Besitz eines Telefons und die Verfügbarkeit eines Anschlusses in die Lage versetzt, telefonieren zu können oder dass der Besitz einer Kinokarte gleichzusetzen ist mit der Reservierung eines Platzes), sodass Dienstanfragen erweitert und auch semantisch weiter entfernte Dienste gefunden werden könnten. Beispielsweise könnten bei der Suche nach einem Dienst, der einen Platz in einem Kino reserviert, auch Dienste gefunden werden, bei denen eine Kinokarte gekauft werden kann. Als Fernziel wäre es damit sogar denkbar, dass Agenten eigenständig auf Basis ihrer Bedürfnisse Dienstangebots- und -anfragebeschreibungen generierten. Eine solche Fundierung der Zustände auf eine formale Basis ist jedoch vermutlich aufgrund des direkten Realweltbezugs äußerst schwierig. Besonders beachtet werden muss, dass die Einigungsproblematik nicht einfach auf die Ebene der formalen Semantik verschoben wird, sondern dass diese Semantik einem Rechner helfen kann, Rückschlüsse über erbringbare und zu erbringende Zustände zu ziehen.

Die Reihe der möglichen Erweiterungen und weiteren Arbeiten zeigt, dass das Gebiet der Semantischen Webdienste auch in Zukunft ein spannendes Forschungsthema bleiben wird. Es bleibt zu hoffen, dass die in dieser Arbeit entwickelten Konzepte eine geeignete Grundlage für diese Weiterentwicklungen bilden.

Teil IV.

Anhang

A. Grounding in DSD

Ein Teil einer Dienstbeschreibung in DSD stellt das *Grounding* dar, welches den Zusammenhang zwischen der abstrakten, formellen Beschreibung des Dienstes im Profile und der konkreten, ausführbaren Funktion des Effektgenerators beschreibt. In DSD besitzen sowohl Beschreibungen für Angebots- als auch für eine Anfragebeschreibungen ein Grounding.

A.1. Schema des Groundings

Das Schema des Groundings ist in Abbildung A.1 dargestellt [39]. Ausgangspunkt ist die Klasse ServiceGrounding, die von der Klasse Service in `upper` über das Attribut supports referenziert wird. ServiceGrounding muss als abstrakte Klasse angesehen werden, da keine direkten Instanzen hiervon existieren. Vielmehr stehen für die verschiedenen Techniken der Effektgeneratoren unterschiedliche spezialisierte Groundingklassen zur Verfügung. Beispielhaft ist das für ein Grounding auf einen Effektgenerator als Javamethode dargestellt.

Das JavaServiceGrounding beschreibt zunächst mittels des Attributs javaVersion welche Version der Java Virtual Machine vorhanden sein muss, damit die zugrunde liegende

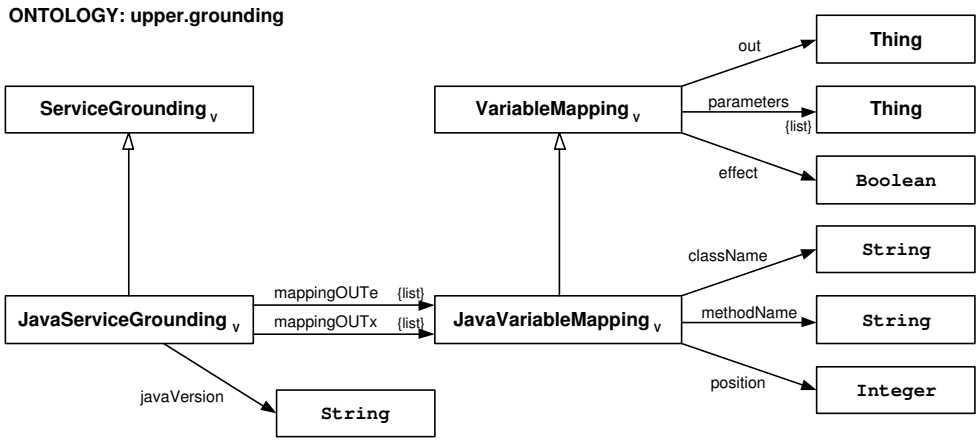


Abbildung A.1.: Ontologie `upper.grounding` für Groundings in DSD.

Funktion verwendet werden kann. Wichtiger sind jedoch die listenwertigen Attribute[1] mappingOUTe und mappingOUTx. Für jede in der Dienstbeschreibung auftretende OUT-Variable muss die Klasse JavaVariableMapping einmal instanziiert und in das entsprechende listenwertige Attribut eingefüllt werden. Weitere Instanzen sind für Methoden nötig, die keine Ausgaben in Form von Rückgabewerte besitzen. JavaVariableMapping beschreibt, wie der Wert einer OUT-Variablen durch die reale Funktion berechnet wird. Dazu verweist das geerbte Attribut out auf die entsprechenden OUT-Variable im Profile. className und methodName geben an, welche Javamethode aufgerufen wird. Als Parameter werden ihr die in parameters angegebenen Werte von IN-Variablen übergeben. Mithilfe des Attributs effect wird festgelegt, ob der Aufruf dieser Methode die Effekte des Profils erbringt.

Das Attribut position hat für Groundings von Angebots- und Anfragebeschreibungen eine unterschiedliche Bedeutung. In Angebotsbeschreibungen legt es fest, in welcher Reihenfolge die Methoden aufgerufen sollen, wenn darüber Unklarheit herrscht (etwa bei mehreren Mappings ohne OUT-Variable). In einer Anfragebeschreibungen legt das Attribut fest, in welcher Reihenfolge die Werte der OUT-Variablen im Rückgabetyp der gewünschten Funktion auftreten sollen.

A.2. Beispiel für ein Grounding

Für ein beispielhaftes Grounding einer Angebotsbeschreibung soll der Dienst aus Abschnitt 7.4.3 wieder aufgegriffen werden (siehe Abbildung 7.18, Seite 190). Wir nehmen an, die Effektgenerator dieses Verkaufsdiensts sei in Java programmiert und stehe über die Klasse `PhoneShop` mit den Methoden `login`, `priceOfBook` und `buyBook` zur Verfügung:

```
import dsd.schema.domain.money.Price;
import dsd.schema.domain.literature.Author;

public class PhoneShop {
   public static void login(String accountName, String passwd);
   public static String possiblePhone(Company comp, PhoneCap pc);
   public static Price priceOfPhone(string phoneType);
   public static Price chargedPrice(string phoneType);
   public static void buyPhone(String phoneType, Color color,
            int ccnumber, GregorianCalendar ccvalid, String company);
}
```

[1] Listenwertige Attribute können nur innerhalb eines Groundings verwendet werden und wurden daher in Kapitel 5 nicht explizit eingeführt. Sie erlauben die Füllung eines Attributs mit einer geordneten Liste von Instanzen oder Literalen.

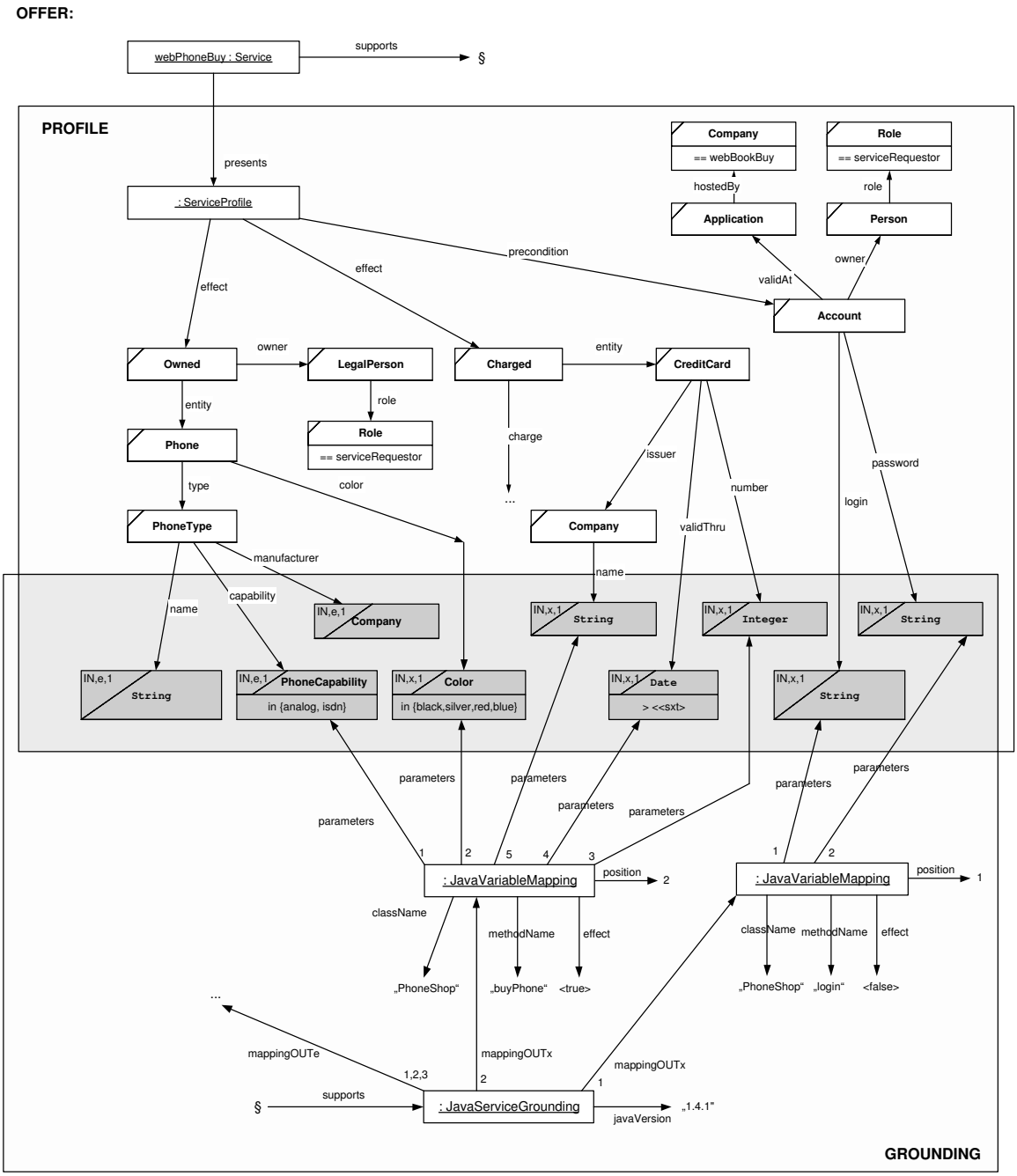


Abbildung A.2.: Beispielhafte Angebotsbeschreibung mit Grounding auf Javamethoden. Die mappingOUTe-Attribute sind aus Gründen der Übersichtlichkeit weggelassen.

Abbildung A.2 zeigt das zugehörige Grounding auf die Javamethoden in g-dsd. Nötig sind fünf Instanzen von JavaServiceGrounding: Zwei als Füllwerte von mappingOUTx, die die Abbildung auf die `login`- und `buyPhone`-Methode vornehmen, und drei als Füllwerte von mappingOUTe, die die Abbildung auf die Methoden `priceOfPhone`, `chargedPrice` und `possiblePhone` vornehmen und hier aus Gründen der Übersichtlichkeit nicht abgebildet sind. Nur die Methode `buyBook` erbringt die beschriebenen Effekte, nämlich dass ein Buch gekauft und die Kreditkarte belastet wird. Im Mapping ist daher das Attribut effect auf <`true`> gesetzt.

A.3. Mapping zwischen Datentypen

Bei der Abbildung zwischen einer realen Funktion und DSD treffen Datentypen aus unterschiedlichen Umgebungen aufeinander: Typen aus DSD und Typen aus der Programmierumgebung der Funktion. Der Zusammenhang zwischen diesen wird nicht in jeder Dienstbeschreibung erneut angegeben, sondern ist allgemein für jede Unterklasse von ServiceGrounding festgelegt. Generell gilt, dass die primitiven Datentypen von DSD direkt auf eingebaute Datentypen der Programmierumgebung der beschriebenen Funktion abgebildet werden sollten. Jedoch sind auch Abbildungen auf selbstdefinierte Typen denkbar. Klassentypen aus DSD können im Normalfall nicht auf das Typsystem der Programmierumgebung abgebildet werden.

Für JavaServiceGroundings gilt die folgende besondere Regelung zur Typabbildung:

- Primitive Datentypen von DSD werden direkt auf korrespondierende primitive Typen oder Javaklassen abgebildet. Die genaue Umsetzung liefert die Tabelle in Abbildung A.3.
- Auch DE-Klassen können in Java verwendet werden. Sie werden dazu auf die entsprechenden Javaklassen der j-dsd-Repräsentierung abgebildet. Die Java-Methoden der beschriebenen Funktionen müssen in solchen Fällen direkt mit den j-dsd-Typen arbeiten. Im Beispiel von oben verwendet beispielsweise die Javamethode `priceOfPhone` als Rückgabewert den DSD-Typ `dsd.schema.domain.money.Price`. Eine weiter gehende Abbildung von dsd-Klassen auf Javaklassen ist im Grounding nicht vorgesehen, sondern muss durch externe Wrapper durchgeführt werden.

Primitiver Typ in DSD	Typ in Java
`Integer`	`int`
`Double`	`double`
`String`	`java.lang.String`
`Boolean`	`boolean`
`Date`	`java.util.GregorianCalendar`
`Time`	`java.util.GregorianCalendar`
`DateTime`	`java.util.GregorianCalendar`
`Duration`	`long`

Abbildung A.3.: Umsetzung der primitiven Datentypen von DSD auf Typen in Java.

B. Repräsentationsformen f-dsd und j-dsd

Dieser Teil des Anhang stellt zwei weitere Repräsentationsformen für DSD vor, die neben der graphischen Form, welche durchgängig in dieser Arbeit verwendet wird, verwendet werden können. Dies sind

- eine *formale Repräsentation* (kurz f-dsd), welche durch eine formale Grammatik definiert ist (siehe Anhang C). Ihr Ziel es ist, möglichst kompakte, textbasierte Beschreibungen zu erhalten. Die Syntax von f-dsd ist angelehnt an F-Logic und übernimmt die Darstellung primitiver Werte aus XML Schema.
- eine *Java-basierte Nebenrepräsentation* (kurz j-dsd). In ihr wird die Ontologie als Satz von Java-Klassen erfasst. Als Vorteile können durch Kompilierung des Quellcodes automatisch semantische Fehler entdeckt werden, deren Überprüfung in f-dsd komplexe Zusatzprogramme erfordert hätte, und die übersetzten Klassen sind direkt in Java-Programmen verwendbar.

Die Syntaxen von j-dsd und f-dsd ähneln sich in vielen Punkten. Um Wiederholungen zu vermeiden, werden die beiden Repräsentationsformen im Folgenden daher zusammen vorgestellt. Dabei wird auf die Notation von Schemata, Instanzen, scharfen und unscharfen Mengen, Variablen sowie Operatoren eingegangen.

B.1. Schemata

B.1.1. Primitive Datentypen

In f-dsd erfolgt die Notation durch den Namen des primitiven Typs:

- `Integer`. Repräsentiert die Menge der ganzen Zahlen.
- `Double`. Repräsentiert die Menge der Fließkommazahlen.

- `String`. Repräsentiert die Menge der Zeichenketten, d.h. endliche Ketten von Einzelzeichen.
- `Boolean`. Repräsentiert die Menge der Wahrheitswerte, d.h. die Menge bestehend aus `wahr` und `falsch`.
- `Date`. Repräsentiert die Menge der Datumswerte. Ein Datum steht für einen vergangenen, jetzigen oder zukünftigen Tag in der Zeit, also ein nicht-periodisches Ereignis. Ein Tag beginnt um 0:00 Uhr und endet mit dem Beginn des folgenden Tages.
- `Time`. Repräsentiert die Menge der Zeitpunkte an einem Tag. Eine solcher Zeitpunkt wiederholt sich an jedem Tag. Seine Dauer ist null.
- `DateTime`. Repräsentiert die Menge der einzelnen Punkte in der Zeit. Die Dauer eines Zeitpunkts ist null. `DateTime` kann als Kombination von `Date` und `Time` angesehen werden.
- `Duration`. Repräsentiert die Menge der Zeitdauern. Eine Zeitdauer hat keinen bestimmten Anfangs- und Endzeitpunkt, sondern steht stellvertretend für jedes Zeitintervall dieser Länge.

In j-dsd werden die primitiven Typen nicht direkt durch die in Java eingebauten Datentypen `int`, `double` etc. abgebildet, sondern durch spezielle Klassen nachgebildet. Diese Klassen haben den Namen des primitiven Datentyps mit einem vorangestellten `XSD_`, um sie von gleich lautenden Java-Klassen unterscheiden zu können. Alle diese Klassen erben von der abstrakten Oberklasse `XSD_Type`, was sie als primitive Typen kennzeichnet. `XSD_Type` erbt von der allgemeinen Oberklasse `Thing` (siehe Abschnitt 5.2.2). Die Klassen befinden sich im Javapaket `dsd.elements.xsd`. Intern werden die Werte entweder durch die eingebauten Javatypen repräsentiert (`int, double, boolean`), durch analoge Javaklassen abgebildet (`String`) oder in anderen Datenstrukturen abgespeichert (`java.util.GregorianCalendar` oder `long` für die Datumstypen). Zur Erzeugung von primitiven Werten steht daher ein entsprechender Konstruktor zur Verfügung. Eine beispielhafte Definition von `XSD_Integer` sieht wie folgt aus:

```
package dsd.elements.xsd;

public abstract class XSD_Type extends Thing {}

public class XSD_Integer extends XSD_Type
{
   private int value;
```

```
    XSD_Integer(int value)
    {
        this.value = value;
    }
}
```

Zur Implementierung der **Ordnungsrelation** auf primitiven Datentypen steht für j-dsd in `XSD_Type` die abstrakte Methode `int compareTo(XSD_Type value)` zur Verfügung, welche in den konkreten Untertypen implementiert wird. Sie liefert -1, 0 oder 1, je nachdem ob der Referenzwert kleiner, gleich oder größer als der Vergleichswert `value` ist. Für unscharfe Vergleiche steht zusätzlich die `double fCompareTo(XSD_Type value)`[1] zur Verfügung, die einen Fließkommawert aus [-1,1] zurück gibt. Als Parameter kann die maximale Abweichung angegeben werden.

B.1.2. Klassen

Klassen werden in f-dsd durch den Klassennamen gefolgt von der Attributdefinition (siehe Abschnitt 5.2.2) in eckigen Klammer notiert. Klassennamen beginnen stets mit einem Großbuchstaben:

```
PhoneType[]
Price[]
```

In j-dsd wird das Konzept der Klasse auf gewöhnliche öffentliche Javaklassen abgebildet. Hier würden die Klassen PhoneType und Price als

```
public class PhoneType extends Thing {}
public class Price extends Thing {}
```

dargestellt und wie in Java üblich in einzelnen Dateien `Device.java` und `Price.java` abgelegt. Jede Klasse erbt von der allgemeinen Oberklasse `Thing` aus dem Paket `dsd.elements` (falls nicht anders angegeben, siehe Abschnitt 5.2.2).

In f-dsd erfolgt die Definition der **Attribute** in den eckigen Klammern nach der Klassendefinition, jeweils als <`Attributname`> : <`Zieltyp`> durch Komma getrennt. Ein Beispiel wäre:

[1] Das `f` steht für fuzzy, deutsch also unscharf.

```
PhoneType
[
   name: String,
   availableSince: Date,
   manufacturer: Company
]
```

In j-dsd werden Attribute durch gewöhnliche öffentliche Javaattribute der zugehörigen Javaklasse dargestellt. Ihr Typ entspricht dem Zieltyp. Im Beispiel wäre das:

```
public class PhoneType extends Thing
{
   public XSD_String name;
   public XSD_Date availableSince;
   public Company manufacturer;
}
```

In f-dsd werden **definierende Attribute** mit dem vorangestellen Schlüsselwort `defprop` markiert, **ableitbare Attribute** bleiben auch hier ohne Markierung:

```
PhoneType
 [
     defprop name: String,
     availableSince: Date,
     manufacturer: Company,
 ]
```

In j-dsd ist die Markierung schwieriger und geschieht durch den statischen Vektor `defprops`. In ihm wird jedes definierende Attribut mit seinem vollständigen Namen eingetragen. Dies geschieht in einem statischen Initialisierer der Klasse. Im Beispiel:

```
public class PhoneType extends Thing
{
   public XSD_String name;
   static {defprops.add("domain.telecommunication.
                         PhoneType.name");}

   public XSD_Date availableSince;
   public Company manufacturer;
}
```

In f-dsd wird die **Vererbung** durch `extends <Name der Oberklasse>` nach dem Namen der Klasse notiert:

```
Company extends LegalPerson []
```

In j-dsd wird der Mechanismus aus Java genutzt, um eine Vererbung darzustellen. Ein Erben von `Thing` muss hier explizit angegeben werden:

```
public class LegalPerson extends Thing {}

public class Company extends LegalPerson {}
```

In f-dsd werden **orthogonale Attribute** durch das Schlüsselwort `orthprop` gekennzeichnet:

```
PhoneType
 [
     defprop name: String,
     availableSince: Date,
     manufacturer: Company,
     orthprop isPriced: Priced
 ]
```

In j-dsd erfolgt die Kennzeichnung analog zu definierenden Attributen über durch den statischen Vektor `orthprops` in `Thing`:

```
public class PhoneType extends Thing
{
   ...
   public Priced isPriced;
   static {orthprops.add("domain.telecommunication.
                          PhoneType.isPriced");}
}
```

Die formale Semantik orthogonaler Attribute findet sich in Regel 8.19.

Die Sammlung der Klassen und Instanzen einer zu definierenden **Ontologie** werden in f-dsd in einer Datei zusammengefasst. Diese beginnt mit dem Markierer `{ontology <Name>}`, mit dem die Neudefinition der Ontologie mit dem angegebenen Namen eingeleitet wird. In der Datei selbst können eine beliebige Anzahl von Klassen und

```
{ontology domain.telecommunications}

PhoneType
[
   defprop name: String,
   availableSince: Date,
   manufacturer: Company
]

Telco extends Company
[]
```

Abbildung B.1.: Datei `domain.telecommunication.dsd` zur Definition der Ontology `domain.telecommunication` in f-dsd.

Instanzen definiert werden, welche alle in der angegebenen Ontologie abgelegt werden. Abbildung B.1 zeigt einen Ausschnitt aus der Datei `domain.telecommunications.dsd`, welche das Anwendungsgebiet der Telekommunikation repräsentiert.

Für die Umsetzung des Ontologiekonzeptes in j-dsd wird das vorhandene Java-Package-Konzept verwendet. Dabei gilt, dass eine Klasse der Ontologie <name> im Javapaket `dsd.schema.`<name> zum Liegen kommt. Dies führt dazu, dass die Pakete der primitiven Datentypen sowie die der speziellen Konstrukte wie `Thing` oder `ThingList` explizit über `import` eingebunden werden müssen. Klassen derselben Ontologie liegen im gleichen Javapaket und können somit direkt verwendet werden. Folgender Code zeigt die Definition von PhoneType in j-dsd:

```
package dsd.schema.domain.telecommunication;

import dsd.elements.*;
import dsd.elements.xsd.*;

public class PhoneType
{
   public XSD_String name;
   static {defprops.add("domain.telecommunication.PhoneType.name");}

   public XSD_Date availableSince;
   public Company manufacturer;
}
```

In f-dsd geschieht die Markierung von Klassen einer **externen Ontologie** durch die Angabe von `at <Ontologiename>` nach der externen Klasse. Eine beispielhafte Definition von Company könnte daher wie folgt aussehen:

```
{ontology domain.economy}

Company extends LegalPerson at domain.legal
[
   name: String,
   domicile: City at domain.location
]
```

Da j-dsd den gewöhnlichen Package-Mechanismus von Java nutzt, um Ontologien abzubilden, kann die Referenzierung von Klassen aus anderen Ontologien durch zwei bekannten Mechanismen erfolgen: (1) Einbinden der benötigten Klassen über `import`-Anweisungen oder (2) vollständige Benamung der Klassen mittels der Punkt-Notation. Folgendes Beispiel nutzt beide Möglichkeiten:

```
package dsd.schema.domain.economy;
import dsd.schema.domain.location.City;

public class Company extends dsd.schema.domain.legal.LegalPerson
{
   public XSD_String name;
   public City domicile;
}
```

In f-dsd wird die Markierung einer **Entitätsklasse** durch Voranstellen des Schlüsselworts `entityclass` vor der Klassendefinition erreicht. **Wertbestimmte Klassen** erhalten die Markierung `valueclass`. Bei fehlender Markierung wird als Standard eine Markierung mit `entityclass` angenommen.

```
entityclass Person
[
   name: String
]

valueclass Price
[
   amount: Double,
   currency: Currency
```

```
]

entityclass Currency []
```

In j-dsd erfolgt die Markierung im statischen Attribut `entityclasses` vom Typ `Vector` in `Thing`. Hier werden alle Namen von Entitätsklassen abgelegt. Die Namen sind dabei vollständig mit Ontologie anzugeben. Nicht abgelegte Klassen gelten als wertbestimmte Klassen. Zum Eintrag in den `Vector` eignet sich ein statischer Initialisierer. Die Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
public class Person extends Thing
{
   static {entityclasses.add("domain.person.Person");}

   public XSD_String name;
}

public class Price extends Thing
{
   public XSD_Double amount;
   public Currency currency;
}

public class Currency extends Thing
{
   static {entityclasses.add("domain.money.Currency");}
}
```

Öffentliche Entitätsklassen werden in f-dsd durch voranstellen des Schlüsselworts `public` markiert:

```
public entityclass Actor [...]

entityclass Person [...]
```

In j-dsd erfolgt die Markierung im statischen Attribut `publicclasses` vom Typ `Vector` in `Thing`. Hier werden alle Namen von öffentlichen Entitätsklassen abgelegt. Die Namen sind dabei vollständig mit Ontologie anzugeben. Nicht abgelegte Klassen gelten als teilöffentliche Klassen. Zum Eintrag in den `Vector` eignet sich ein statischer Initialisierer. Die Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
public class Currency extends Thing
{
   static {entityclasses.add("domain.money.Currency");}
   static {publicclasses.add("domain.money.Currency");}
   ...
}

public class Person extends Thing
{
   static {entityclasses.add("domain.person.Person");}
   ...
}
```

B.2. Instanzen

B.2.1. Werte primitiver Typen

In f-dsd werden Werte primitiver Typen in der vorgestellten Standardrepräsentation notiert. In j-dsd werden primitive Werte durch Javaobjekte der speziellen XSD-Typen dargestellt. Zur Erzeugung hat jeder XSD-Typ einen Konstruktor, der als Parameter den Wert in der vorgestellten Standardrepräsentation erwartet. Für jeden Typ existiert zudem eine geeignete `toString`-Methode, die den Wert in der Standardrepräsentation ausgibt.

B.2.2. Instanzen von Klassen

In f-dsd können **benamte Instanzen** der speziellsten Klasse k nur innerhalb der Ontologie definiert werden, welche auch die Klasse k enthält. Die Syntax lautet: `<Name der Instanz> as <Klassenname>`. Dabei muss der Name der Instanz *innerhalb der Klasse* eindeutig sein. Global wird er durch Anfügen des Ontologie- und Klassennamens vor dem Instanznamen eindeutig. Instanznamen beginnen immer mit einem Kleinbuchstaben. Im Beispiel werden in der Telekommunikationsontologie Instanzen für bestimmte Telefonmodelle der realen Welt angelegt:

```
{ontology domain.telecommunication}

siemensGigasetS445 as PhoneType
sinus421 as PhoneType
```

In j-dsd gestaltet sich die Definition benamter Instanzen schwieriger. Zwar werden Instanzen von DSD in j-dsd auf gewöhnliche Javaobjekte abgebildet, jedoch wird dadurch alleine keine Benamung und keine Dauerhaftigkeit der Objekte erreicht. Zur Behebung des Problems wird in j-dsd für jede benamte Instanz eine *Erzeugungsklasse* angelegt. Diese verfügt über eine statische Methode `create`, welche bei Bedarf das Javaobjekt zurückliefern kann. Der Name der Instanz wird durch das Attribut `instanceName` (geerbt von `Thing`) bestimmt, welches bei der Erzeugung gesetzt wird. Hierbei kommt der oben erwähnte, global eindeutige Name bestehend aus Ontologie-, Klassen- und Instanzname zum Einsatz. Die Erzeugungsklasse trägt den Namen der Instanz (mit beginnendem Kleinbuchstaben) und befindet sich im Javapackage, das sich aus Ontologie und Klassennamen ergibt und mit der Prefix `dsd.instance` beginnt. Die DSD-Klasse selbst muss über eine `import`-Anweisung eingebunden werden, da sie sich in einem anderen Package (`dsd.schema...`) befindet.

Eine Beispieldefinition einer Instanz sieht in j-dsd wie folgt aus:

```
package dsd.instance.domain.telecommunication;

import dsd.schema.domain.telecommunication.PhoneType;

public class siemensGigasetS445
{
   public static PhoneType create()
   {
       PhoneType instance = new PhoneType();
       instance.instanceName
         = "dsd.instance.domain.telecommunication.
            PhoneType.siemensGigasetS445";
       return instance;
   }
}
```

Die Gleichheit benamter Instanzen wird in j-dsd nicht auf die Gleichheit von Javaobjekten (mittel `==`-Operator) zurückgeführt, sondern muss über die `equals`-Methode erfolgen. Diese testet die Gleichheit dann über einen Stringvergleich der zugehörigen `instanceName`s. Dies hat zwei Vorteile: Erstens muss die `create`-Methode nicht die in Umlauf gebrachten Javaobjekte kontrollieren, zweitens funktioniert der Ansatz auch in verteilten Umgebungen, in denen unterschiedliche Java Virtual Machines autonom ihre eigenen Javazeiger verwalten.

In f-dsd erfolgt die Definition **anonymer Klassen** durch `anonymous <Klassenname>`, wobei häufig die Angabe der Ontologie durch ein anschließendes `at <Ontologiename` nötig ist. Beispiel:

```
anonymous Price at domain.money
anonymous WeightMeasure at domain.measure
```

Die vorgestellten Definition machen so keinen Sinn, da sie erstens nicht über einen Namen aufgegriffen werden können und zweitens keine ausgefüllten Attribute besitzen (siehe nächster Abschnitt).

Da die Probleme von Benamung und Dauerhaftigkeit bei anonymen Instanzen entfallen, können diese in j-dsd sehr einfach durch Anlegen normaler Javaobjekte über den Konstruktor erstellt werden. Im Beispiel:

```
new dsd.schema.domain.money.Price();
new dsd.schema.domain.measure.WeightMeasure();
```

In f-dsd muss die **Füllung der Attribute** einer Instanz direkt nach der Definition der Instanz angegeben werden. Diese geschieht im *Instanzattributblock* , welcher durch eckige Klammern eingeschlossen wird. Darin erfolgt die Füllung in der Syntax <`Attributname`> = <`Füllwert`>. Mehrere Füllungen werden durch Komma getrennt. Als Füllwerte können (je nach Typ) primitive Werte, existierende benamte Instanzen oder an dieser Stelle definierte anonyme Instanzen dienen. Beispiel:

```
{ontology domain.telecommunication}

siemensGigasetS445 as PhoneType
[
   name = "Siemens Gigaset S445",
   manufacturer = siemens,
   weight = anonymous WeightMeasure
            [
               val = 252,
               unit = gram
            ],
   availableSince = <2003-05-01>,
]
```

Im Beispiel ist zu sehen, dass zwar benamte Instanzen und Klassen aus anderen Ontologien verwendet wurden (wie siemens aus `domain.economy` oder WeightMeasure aus `domain.measure`), deren Ontologienamen aber nicht explizit notiert wurden. Generell gilt, dass eine spezifizierende Typangabe weggelassen werden kann, wenn die Instanz/Klasse genau dem Zieltyp des Attributs entspricht. Sie kann in diesem Falle ja aus dem Schema abgeleitet werden. Bei Untertypen muss bei benamten Instanzen per

`as <Klassenname>` (`at <Ontologiename>`) die eigentliche Klasse (mit Ontologie) markiert werden.

In j-dsd werden die Attribute einer Instanz gefüllt, in dem die Attribute des jeweiligen Javaobjekts gesetzt werden. Dabei werden primitive Füllwerte über den entsprechenden Konstruktor des XSD-Types erzeugt, benamte Instanzen durch Aufruf der zugehörigen `create`-Methode referenziert und anonyme Instanzen in einem Vorschritt erzeugt. Die Füllung erfolgt für benamte Instanzen in deren `create`-Methode, für anonyme Instanzen direkt nach deren Erzeugung durch den Konstruktor. Das Beispiel von oben in j-dsd zeigt Abbildung B.2.

B.3. Mengen

B.3.1. Typbedingungen

In f-dsd werden Mengen durch `set of <Typbedingung>` notiert, wobei die Typbedingung den Namen der Klasse darstellt. Stammt die Klasse aus einer fremden Ontologie, kann dies durch ein anschließendes `at <Ontologiename>` gekennzeichnet werden. Mengen werden immer in runden Klammern notiert:

```
(set of Company)
(set of Double)
(set of Person at domain.person)
```

In j-dsd wird ein Trick angewandt, um Mengen abbilden zu können. Dies ist nötig, da später Mengen an die Position von Einzelinstanzen eingesetzt werden können müssen (siehe dazu Abschnitt 6.4). Daher erbt die allgemeine Oberklasse `Thing` von der Javaklasse `Set`. In `Set` gibt es ein Java-Attribut `boolean isSet`, welches festlegt, ob ein Javaobjekt eine DSD-Instanz oder eine DSD-Menge repräsentiert. DSD-Mengen mit der Typbedingung k werden daher als Javaobjekte der Klasse k dargestellt, wobei `isSet` auf `true` gesetzt wird. Die Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
Company companySet = new Company();
companySet.isSet = true;

XSD_Double doubleSet = new XSD_Double();
doubleSet.isSet = true;

import dsd.schema.domain.person.Person;
Person personSet = new Person();
personSet.isSet = true;
```

B.3.2. Direkte Bedingungen

In f-dsd werden direkte Bedingungen durch `with elements <Operator> <Vergleichsinstanz/-wert>` notiert:

```
(set of Company
 with elements == webBuy)

(set of Double
 with elements >= 0.00
 with elements <= 8.00)

(set of PhoneType
 with elements in {sinus421, siemensGigasetS445})
```

j-dsd verwendet ein spezielles Javaattribut `Vector directConditions` in `Set` zur Aufnahme der direkten Bedingungen. Für jede Bedingung muss hier ein Javaobjekt vom Typ `DirectCondition` eingefügt werden. `DirectCondition` hat zwei Attribute: `String operator` und `Object value`. Als `operator` kommen die oben erwähnten Operatoren in Frage, `value` ist entweder ein `Thing` oder ein `Vector`, je nachdem ob der Operator `in` verwendet wurde. `DirectCondition` verfügt auch über eine Methode `test`, welche testet, ob eine Instanz bzw. ein Wert die Bedingung erfüllt. Zwei der Beispiele von oben sehen damit in j-dsd wie folgt aus:

```
Company companySet = new Company();
companySet.isSet = true;
companySet.directConditions = new Vector();
DirectCondition company_dc01 = new DirectCondition();
company_dc01.operator = "==";
company_dc01.value = dsd.instance.domain.economy.
                     Company.webBuy.create();
company.directConditions.add(company_dc01);

PhoneType phtSet = new PhoneType();
phtSet.isSet = true;
phtSet.directConditions = new Vector();
DirectCondition pht_dc01 = new DirectCondition();
pht_dc01.operator = "in";
pht_dc01.value = new Vector();
pht_dc01.value.add(dsd.instance.domain.telecommunication.
  PhoneType.siemensGigasetS445.create());
```

```
pht_dc01.value.add(dsd.instance.domain.telecommunication.
  PhoneType.sinus421.create());
phtSet.directConditions.add(actor_dc01);
```

B.3.3. Attributbedingungen

In f-dsd steht für Attributbedingungen das Konstrukt `where <Attributname> in <Zielmenge>` zur Verfügung. Das Beispiel sieht also wie folgt aus:

```
(
  set of PhoneType
  where manufacturer in
  (
    set of Company
    with elements == siemens
  )
  where availableSince in
  (
    set of Date
    with elements >= <2003-05-01>
  )
)
```

Als Abkürzung können einelementige Zielmengen durch das Konstrukt `where <Attributname> == <einziges Element der Zielmenge>` beschrieben werden. Das Beispiel kann also wie folgt verkürzt werden:

```
(
  set of PhoneType
  where manufacturer == siemens
  where availableSince in
  (
    set of Date
    with elements >= <2003-05-01>
  )
)
```

In j-dsd werden die gewöhnlichen Attribute der Klasse genutzt, um Attributbedingungen auszudrücken. Es gilt, dass Attribute einer Menge, die nicht `null` sind, als Attributbedingungen gelten. Das Beispiel von oben sieht also in j-dsd wie folgt aus, wenn man annimmt, dass die Company- und `Date`-Menge zuvor bereits in `companySet` und `dateSet` definiert sind:

```
PhoneType phtSet = new PhoneType();
phtSet.isSet = true;
phtSet.manufacturer = companySet;
phtSet.availableSince = dateSet;
```

B.3.4. Fehlstrategien

In f-dsd wird die Fehlstrategie direkt nach dem `where` der Attributbedingung in geschweifte Klammern gesetzt. Die Syntax lautet `when missing <Fehlstrategiename>`, wobei im Namen der Strategie Unterstriche durch Leerzeichen ersetzt werden. Das Beispiel von oben sieht also in f-dsd wie folgt aus:

```
(
  set of PhoneType
  where {when missing ignore} availableSince in
  (
    set of Date
    with elements >= <2003-05-01>
  )
)
```

In j-dsd steht zur Markierung der Fehlstrategie eine nicht-statische Hashtabelle `missingStrategy` in `Set` zur Verfügung. Diese wird mit Tupeln der Form `(<Attributname>, <Fehlstrategiename>)` gefüllt. Fehlende Attribute haben implizit die Standardstrategie `assume_failed`. Das Beispiel von oben sieht in j-dsd also wie folgt aus (sofern `dateSet` bereits definiert ist):

```
PhoneType phtSet = new PhoneType();
phtSet.isSet = true;
phtSet.missingStrategy = new Hashtable();
phtSet.missingStrategy.put("availableSince", "ignore");
phtSet.availableSince = dateSet;
```

B.3.5. Verbindungsstrategien

In f-dsd wird eine Verbindungsstrategie am Ende der Mengendefinition durch `combine by <Ver- bindungsausdruck>` notiert. Dieser muss vollständig geklammert sein, d.h. je zwei Operanden werden in runden Klammern umschlossen. Das Beispiel von oben sieht also wie folgt aus:

```
(
  set of PhoneType
  where manufacturer == siemens
  where isRated in
  (
    set of Rated
    where rating in
    (
      set of Rating
      with elements in {good,veryGood}
    )
  )
  combine by (manufacturer or isRated)
)
```

Die Notation der Verbindungsstrategie in j-dsd wird im Rahmen der unscharfen Verbindungsstrategien vorgestellt.

B.3.6. Typvergleichsstrategien

In f-dsd wird die Typvergleichsstrategie durch `or supertype` nach dem Klassennamen der Menge notiert. Eine evtl. maximale Länge von Vererbungsbeziehungen folgt in eckigen Klammern. Das Beispiel sieht also wie folgt aus:

```
(set of Company or supertype[1,1])
```

In j-dsd kann die Typvergleichsstrategie durch das Attribut `String typeCheckStrategy` in `Set` eingestellt werden. Werte sind die oben vorgestellten Kurzbezeichnungen. Im Beispiel:

```
Company companySet = new Company();
companySet.isSet = true;
companySet.typeCheckStrategy = "super[1,1]";
```

B.3.7. Test auf Mengenzugehörigkeit

In j-dsd existiert in `Set` eine Methode zur Bestimmung der Mengenzugehörigkeit: `double contains (Thing t)`. Diese testet, ob eine gegebene Instanz `t` in der Menge liegt. Wenn ja, liefert sie 1.0, wenn nicht 0.0. `contains` stützt sich auf drei private Hilfsmethoden `checkType`, `checkDirectConditions` und die rekursive Methode `checkPropertyConditions`, um das Ergebnis zu bestimmen.

B.4. Unscharfe Mengen

B.4.1. Unscharfe direkte Bedingungen

Ein Beispiel in f-dsd könnte wie folgt aussehen:

```
(
  set of Double
  with elements ~>=[6.00] 7.00
  with elements ~<=[9.00] 8.00
)
```

In f-dsd sieht das beispielsweise wie folgt aus:

```
(
  set of Company
  with elements in {siemens[1.0], nokia[0.8], samsung[0.2]}
)

(
  set of Date
  with elements in {<2004-01-01>[0.75], <2005-01-01>[0.5]}
)
```

Im zweiten Fall wird eine domänenspezifische Funktion eingesetzt, die für diese Klasse definiert wurde. Eine beispielhafte Nutzung einer solchen Methode sieht dann in f-dsd wie folgt aus:

```
(
  set of City
  with elements _near(this,karlsruhe)
)
```

In dieser Menge sind beispielsweise Städte, die sich in der Nähe von Karlsruhe befinden. Die Ähnlichkeit wurde dabei von einem Domänenexperten festgelegt und könnte etwa die Entfernung per Luftlinie in Betracht ziehen.

B.4.2. Unscharfe Fehlstrategien

In f-dsd notiert man diese Strategie wie gewohnt. Im Beispiel:

```
(
  set of PhoneType
  where {when missing assume value[0.8]} availableSince in
  (
    set of Date
    with elements >= <2003-05-01>
  )
)
```

In j-dsd erfolgt der Eintrag wie gehabt in der Hashtable `missingStrategy`.

B.4.3. Unscharfe Verbindungsstrategien

Ein Beispiel in f-dsd könnte (abgekürzt) wie folgt aussehen:

```
(
  set of PhoneType
  where manufacturer in ...
  where availableSince in ...
  where isRated in ...
  combine by (0.8 * (exp(manufacturer,2) and availableSince)
              + 0.2 * isRated)
)
```

In j-dsd wird eine alternative Verbindungsstrategie im Attribut `connectingStrategy` von `Set` abgelegt. Als Typ dient die abstrakte Javaklasse `Operation`, von der alle konkreten Operationen in Form von Klassen `Mul`, `Add`, `Min`, `Max`, `Exp` und `WS` (für weighted sum) erben. Jede dieser Operationen hat eine gewisse Anzahl von Operanden, die selbst wieder von Typ Operation sind, sowie evtl. zusätzliche Parameter. Hierdurch wird es möglich, einen rekursiven Operatorbaum aufzubauen. Ein besondere Operation stellt `PropertyConditionValue` dar: sie repräsentiert ein Blatt im Operatorbaum als Name einer einzelnen Attributbedingung. Zur Auswertung einer Operation existiert in `Operation` die abstrakte Methode `evaluate`, die von den konkreten Operationen implementiert wird. Innere Knoten verrechnen dabei ihre Operanden entsprechend ihrer Semantik, während Blätter das Ergebnis der entsprechenden Attributbedingung zurückliefern.

B.4.4. Unscharfe Typvergleichsstrategien

In j-dsd erfolgt ein Eintrag im Javaattribut `typeCheckStrategy` von `Set`. f-dsd verwendet die Schreibweise `or supertype[n, f]`.

B.4.5. Test auf unscharfe Mengenzugehörigkeit

In j-dsd übernimmt die bereits in Abschnitt 6.1.7 vorgestellte Methode `contains` die Berechnung der Zugehörigkeit. Ihr Rückgabewert ist `double`.

B.5. Variablen

In f-dsd werden Variablen durch `var from` <`Grundmenge`> definiert. <`Grundmenge`> bezeichnet den Wertebereich der Variable; der Variable kann nur ein Wert bzw. eine Instanz aus dieser Menge zugewiesen werden. Die Grundmenge kann durch eine Klasse oder eine deklarative Menge angegeben werden. Beispiele für Variablendefinitionen in f-dsd könnten sein:

```
var from Date

var from
(
   set of PhoneType
   where manufacturer in
   (
       set of Company
       with elements in {siemens,nokia}
   )
)
```

Im ersten Fall hat die Variable die gesamte Klasse der Datumswerte als Grundmenge, im zweiten Fall kann die Variable mit der Instanz eines Telefonmodells von Siemens oder Nokia gefüllt werden.

Da Variablen stets eine Grundmenge besitzen, existiert in j-dsd eine zusätzliche Java-Klasse `Variable`, die von der Klasse `Set` erbt. Variablen werden dann wie Sets durch Instanzen dargestellt, in denen einerseits wie gehabt das Attribut `isSet` auf `true`, andererseits das neue Attribut `bindingStatus` auf `OPEN` gesetzt wird. Standardmäßig

steht dieser `bindingStatus` auf `NO_VARIABLE`, was angibt, dass es sich um eine normale Instanz bzw. eine normale Menge handelt. Mehr zum Bindungszustand einer Variable findet sich im nächsten Abschnitt.

Die zwei Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
XSD_Date dateVar = new XSD_Date();
dateVar.isSet = true;
dateVar.setBindingStatus(Variable.OPEN);

PhoneType phtVar = new PhoneType();
phtVar.isSet = true;
phtVar.setBindingStatus(Variable.OPEN);
phtVar.manufacturer = companySet;
```

B.5.1. Bindungszustand

In f-dsd und g-dsd können Bindungszustände nicht angegeben werden. Hier gelten alle Variablen als ungebunden. In j-dsd wird der Bindungszustand durch das Javaattribut `bindingStatus` in `Variable` beschrieben. Hierzu stehen die vier vorgestellten Zustände als Konstanten zur Verfügung. Ein Beispiel könnte sein:

```
XSD_Date dateVar = new XSD_Date();
dateVar.isSet = true;
dateVar.setBindingStatus(Variable.FILLED);
dateVar.fillWith(new XSD_Date("2004-08-31"));
```

B.5.2. Kategorien

In f-dsd werden Variablen in Dienstbeschreibungen durch `var (IN|OUT, e|x,` i`)+ from <Grundmenge>` definiert. Durch die drei Parameter wird also die Kategorie der Variable festgelegt. Dabei wird i auf 1 gesetzt werden, wenn eine Variable der Ausführungsphase definiert wird. Solche Kategorieangaben können beliebig oft wiederholt werden (gekennzeichnet durch das +).

In j-dsd werden die Kategorien einer Variable im Attribut `cats` erfasst. Es nimmt einen Vector vom Typ `VariableCategory` auf. Diese Klasse und hat drei Attribute: `where`, welches mit `IN` oder `OUT` belegt werden muss, `when`, welches mit `E` oder `X` belegt werden muss, und `step`, welches mit der Schrittnummer i belegt wird.

B.6. Operatoren

In f-dsd sieht das Beispiel wie folgt aus. Da mehrere Effekte und Vorbedingungen an x möglich sind, werden diese mit `+=` anstatt `=` zugewiesen:

```
x as X
[
  precondition +=
  (
    set of Account
    where validFor = ...
  ),
  effect +=
  (
    set of Owned
    where entity = ...
  )
]
```

Die Umsetzung in j-dsd erfolgt analog.

```
package dsd.instance.domain.telecommunication;

import dsd.elements.xsd.*;
import dsd.schema.domain.telecommunication.PhoneType;
import dsd.schema.domain.measure.WeightMeasure;
import dsd.instance.domain.economy.Company.siemens;

public class siemensGigasetS445
{
   public static PhoneType create()
   {
       PhoneType instance = new PhoneType();
       instance.instanceName
          = "dsd.instance.domain.telecommunication.
             PhoneType.siemensGigasetS445";

       //Anonyme Instanz
       WeightMeasure wm01 = new WeightMeasure()
       wm01.val = 252;
       wm01.unit = dsd.instance.domain.measure.
                   WeightMeasure.gram.create();

       instance.name
           = new XSD_String("Siemens Gigaset S445");
       instance.manufacturer = siemens.create();
       instance.weight = wm01;
       instance.availableSince = new XSD_Date("<2003-05-01>");

       return instance;
   }
}
```

Abbildung B.2.: Definition der Instanz `siemensGigasetS445` und Füllen ihrer Attribute in j-dsd.

C. Formale Grammatik von DIANE Elements

Dieser Teil des Anhang stellt die formale Grammatik von DIANE Elements in f-dsd vor. Die Notation ist im ANTLR-Format[1] gegeben.

```
start:
       GKLAMMERAUF
       ( "ontology" (DOPPELPUNKT)? paket
       | "offer"
       | "request"
       )
       GKLAMMERZU
       (klassendefinition | instanzdefinition)*;

klassendefinition:
       ("public")? ("entityclass" | "valueclass")?
       GROSSBEZEICHNER (vererbung)?
       klassendefblock;

vererbung:
       "extends" klasse;

klassendefblock:
       KLAMMERAUF
        (attributdefinition (KOMMA attributdefinition)*)?
       KLAMMERZU;

attributdefinition:
       ("defprop" | "orthprop")?
       KLEINBEZEICHNER DOPPELPUNKT klasse;

paket:
       KLEINBEZEICHNER  (PUNKT KLEINBEZEICHNER)*;
```

[1] http://www.antlr.org

```
klasse:
       GROSSBEZEICHNER ("at" paket)?;

instanz:
       KLEINBEZEICHNER  ("as" klasse)?;

instanzdefinition:
       KLEINBEZEICHNER  "as"
       ((klasse (instanzattributblock)?) |  declsetdef);

instanzattributblock:
      KLAMMERAUF zuweisung (KOMMA zuweisung)* KLAMMERZU;

zuweisung:
       KLEINBEZEICHNER GLEICH
       ( primitiverwert | instanz | LABEL
       | anonidef | vardef | setdef );

anonidef:
       "anonymous" (l:LABEL)? klasse (instanzattributblock)?;

primitiverwert:
       ( STRING | ZAHL | BOOLEAN
       | DATE | TIME | DATETIME | DURATION);

vardef:
       "var"
       (l:LABEL)?
       (
         RKLAMMERAUF
           who KOMMA
           when KOMMA
           ZAHL
           (KOMMA ("enum" | "decl"))?
         RKLAMMERZU
       )+
       "from" setdef
       ("default" (primitiverwert | instanz | anonidef))?;

who:
       ("in" | "IN" | "out" | "OUT");

when:
       ("e" | "E" | "x" | "X");
```

```
setdef:
       ( klasse | enumsetdef | declsetdef );

enumsetdef:
       GKLAMMERAUF
       ( instanz (KLAMMERAUF ZAHL KLAMMERZU)?
         (KOMMA instanz (KLAMMERAUF ZAHL KLAMMERZU?)*
       | primitiverwert (KLAMMERAUF ZAHL KLAMMERZU)?
         (KOMMA primitiverwert (KLAMMERAUF ZAHL KLAMMERZU)?)*
       )
       GKLAMMERZU;

declsetdef:
       RKLAMMERAUF
         (l:LABEL)?
         "set" KLAMMERAUF ZAHL KLAMMERZU "of" klasse
         (typecheckstrategy)?
         (directcondition)*
         (propertycondition)*
         (connectingstrategy)?
       RKLAMMERZU;

typecheckstrategy:
       "or" "supertype" (KLAMMERAUF ZAHL (KOMMA ZAHL)? KLAMMERZU)?;

directcondition:
       "with" "elements"
       (primitivecondition | instancecondition);

primitivecondition:
       (WAVE)?
       ( TAGAUF (KLAMMERAUF primitiverwert KLAMMERZU)?
       | KLEINERGLEICH (KLAMMERAUF primitiverwert KLAMMERZU)?
       | GLEICHGLEICH (KLAMMERAUF primitiverwert
                       KOMMA primitiverwert KLAMMERZU)?
       | GROESSERGLEICH (KLAMMERAUF primitiverwert KLAMMERZU)?
       | TAGZU (KLAMMERAUF primitiverwert KLAMMERZU)?
       | UNGLEICH (KLAMMERAUF primitiverwert
                   KOMMA primitiverwert KLAMMERZU)?
       )
       primitiverwert;

instancecondition:
       ( (GLEICHGLEICH  | UNGLEICH) instanz
       | "sim" RKLAMMERAUF instanz RKLAMMERZU
```

```
       | "in" enumsetdef
       );

propertycondition:
       ("where" | "and")
       (ms=missingstrategy)?
       KLEINBEZEICHNER
       ( GLEICHGLEICH primitiverwert
       | GLEICHGLEICH instanz
       | GLEICHGLEICH vardef
       | "in" setdef
       | "in LABEL
       );

missingstrategy:
       GKLAMMERAUF "when" "missing"
       ( "assume" "failed"
       | "ignore"
       | "assume" "fulfilled"
       | "assume" "value" KLAMMERAUF ZAHL KLAMMERZU
       )
       GKLAMMERZU;

connectingstrategy:
       "combine" "by" cexpression;

cexpression:
       ( KLEINBEZEICHNER
       | ("exp" RKLAMMERAUF cexpression KOMMA ZAHL RKLAMMERZU)
       | RKLAMMERAUF
          cexpression
          ("and"  | "or"  | "mul"  | "add")
          cexpression
         RKLAMMERZU
       | ("min"  | "max") RKLAMMERAUF cexpression
         (KOMMA cexpression)+ RKLAMMERZU
       | RKLAMMERAUF ZAHL STERN cexpression
         (PLUS ZAHL STERN cexpression)+ RKLAMMERZU
       );


// LEXER

ZIFFER              : '0'..'9';
US                  : '_';
```

```
KLAMMERAUF          : '{';
KLAMMERZU           : '}';
RKLAMMERAUF         : '(';
RKLAMMERZU          : ')';
GKLAMMERAUF         : '{';
GKLAMMERZU          : '}';
GLEICH              : '=';
GLEICHGLEICH        : "==";
KOMMA               : ',';
PUNKT               : '.';
DOPPELPUNKT         : ':';
STERN               : '';
SEMI                : ';';
HOCH                : '^';
TAGAUF              : '<';
TAGZU               : '>';
KLEINERGLEICH       : "<=";
GROESSERGLEICH      : ">=";
UNGLEICH            : "!=";
PLUS                : '+';
WAVE                : '~';
DOLLAR              : '$';


//Primitive Werte

ZAHL        : ('-')? (ZIFFER)+ ('.' (ZIFFER)+)?;

BOOLEAN     : TAGAUF ("true" | "false") TAGZU;

STRING      : '"' (~('"'|'\\'|'\n'|'\r')) '"';

DATE        : TAGAUF
                ZIFFER ZIFFER ZIFFER ZIFFER
                '-' ZIFFER ZIFFER
                '-' ZIFFER ZIFFER
              TAGZU;

TIME        : TAGAUF
                ZIFFER ZIFFER DOPPELPUNKT ZIFFER ZIFFER
                (DOPPELPUNKT ZIFFER ZIFFER
                (PUNKT ZIFFER ZIFFER ZIFFER)?)?
              TAGZU;

DATETIME    : TAGAUF
```

```
                ZIFFER ZIFFER ZIFFER ZIFFER
                '-' ZIFFER ZIFFER '-' ZIFFER ZIFFER
                'T'
                ZIFFER ZIFFER DOPPELPUNKT ZIFFER ZIFFER
                (DOPPELPUNKT ZIFFER ZIFFER (PUNKT ZIFFER ZIFFER ZIFFER)?)?
              TAGZU;

DURATION    : TAGAUF
                'P' (KZAHL ('Y'|'M'|'D'))*
                ('T' (KZAHL ('.' KZAHL)? ('H'|'M'|'S'))*)?
              TAGZU;

KZAHL       : ZIFFER ((ZIFFER) (ZIFFER)?)?;


//Bezeichner

GROSSBEZEICHNER:
              GROSSBUCHSTABE (US | GROSSBUCHSTABE |
              KLEINBUCHSTABE | ZIFFER)*;

KLEINBEZEICHNER:
              KLEINBUCHSTABE (US | GROSSBUCHSTABE |
              KLEINBUCHSTABE | ZIFFER)*;
```

D. Verwendete Ontologien

Dieser Teil des Anhang präsentiert einige Ontologien, die im Laufe der Arbeit verwendet wurden oder für weitere Beschreibungen benötigt werden.

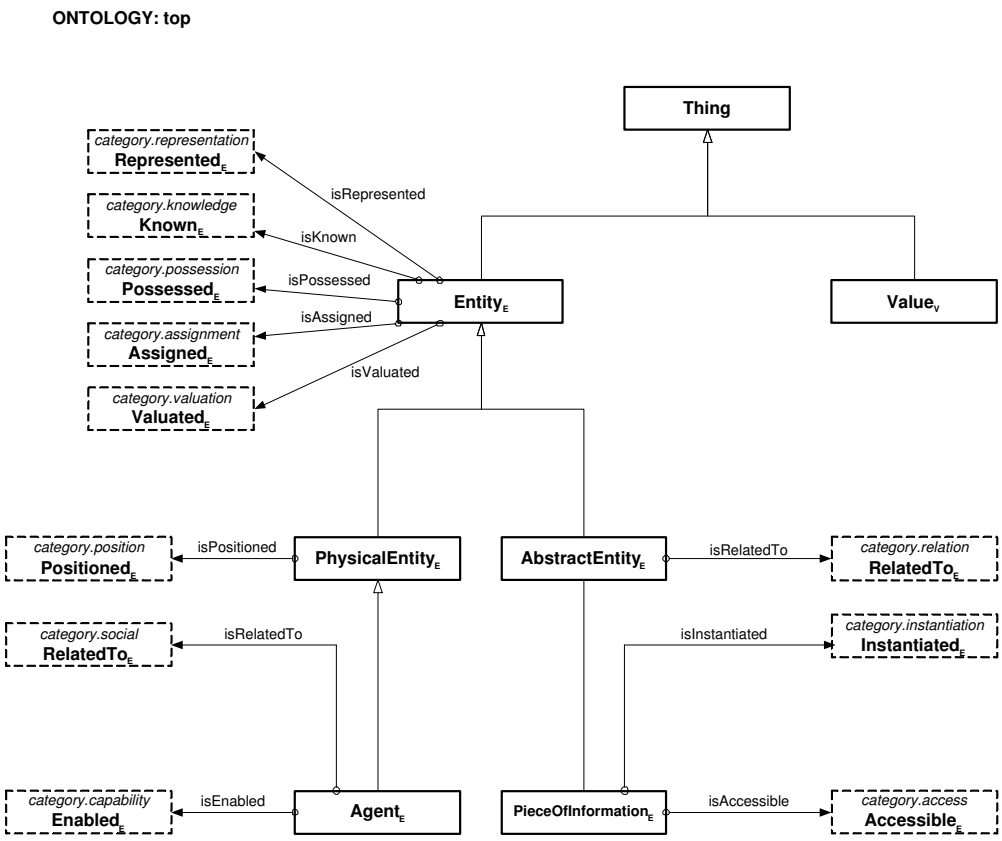


Abbildung D.1.: Die obere Ontologie `top`

D.1. Die obere Ontologie

D.1.1. top

Die obere Ontologie `top` zerlegt die generische Klasse Thing in grundlegende Konzepte (siehe Abbildung D.1):

- Wertbestimmte Klassen (Value) haben eine einfache Identität, Entitätsklassen (Entity) haben eine komplexe Identität.
- Physische Entitäten (PhysicalEntity) bestehen aus Materie, „werfen einen Schatten“ und haben eine Lokation, während abstrakte Entitäten (AbstractEntity) nicht-materielle, gedankliche Konzepte darstellen.
- Eine besondere Art physischer Entitäten sind Agenten (Agent). Sie zeichnen sich dadurch aus, dass sie selbstständig handeln können.
- Eine besondere Art abstrakter Entitäten sind Informationseinheiten (PieceOfInformation), d.h. Entitäten die Informationen repräsentieren.

Häufige orthogonale Attribute sind ausfaktorisiert und an den entsprechenden Konzepten der oberen Ontologie definiert.

D.2. Die obere Dienstontologie

D.2.1. upper

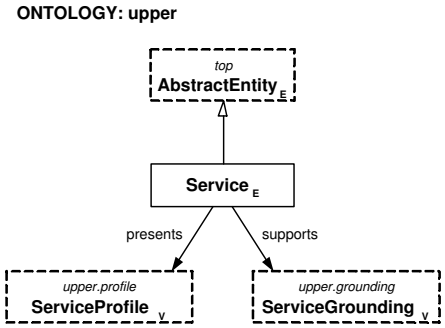


Abbildung D.2.: Die obere Dienstontologie von DSD.

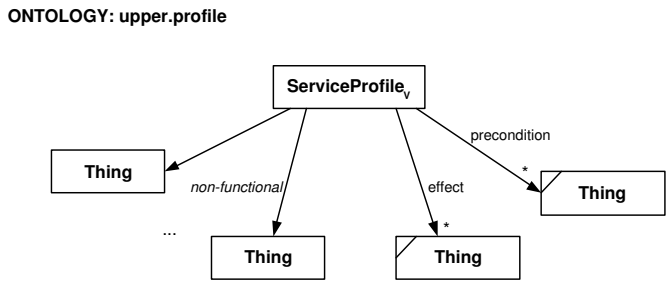


Abbildung D.3.: Die Ontologie für das Dienstprofil von DSD.

D.2.2. upper.profile

D.2.3. upper.grounding

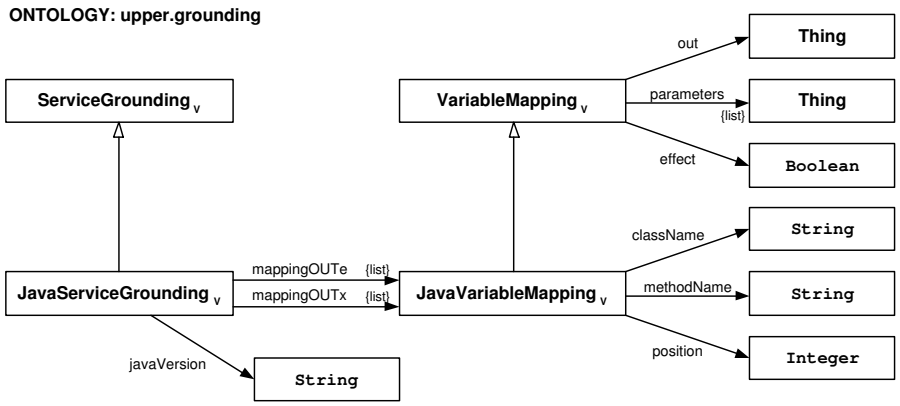


Abbildung D.4.: Die Ontologie für das Dienstfundament von DSD.

D.3. Kategorieontologien

D.3.1. category

Zustände stellen verdinglichte extrinsische Eigenschaften dar. Wie Sie werden durch die Entitätsklasse State in der Ontologie `category` repräsentiert. State hat ein Attribut entity, das auf die beschriebene Entität zeigt, sowie die Attribute startsAt und endsAt, die angeben, in welchem Zeitraum der Zustand gilt. Durch Anlegen konkreter Instanzen von State und deren Unterklassen werden orthogonale Attribute implizit gefüllt, wie in Regel 8.19 festgehalten.

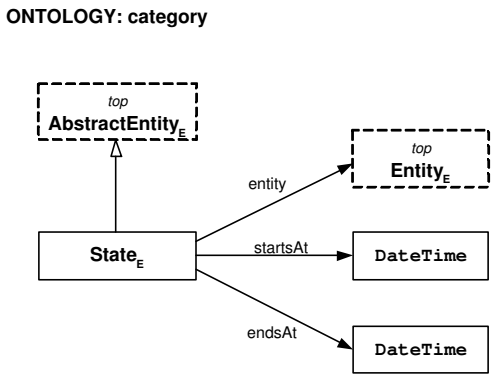


Abbildung D.5.: Die Ontologie `category`, in der die Oberklasse State definiert ist.

D.3.2. category.represenation

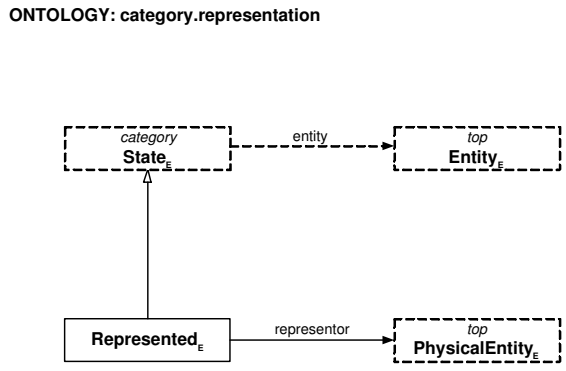


Abbildung D.6.: Die Ontologie `category.representation` zur Darstellung von Zuständen, die etwas über die Representation einer Entität aussagen.

Die Ontologie `category.representation` (siehe Abbildung D.6) drückt aus, wie eine Entität repräsentiert wird. Oberklasse ist Represented, die aussagt, das einer Entität entity allgemein ein representor als physikalische Entität zugewiesen ist.

D.3.3. category.knowledge

Die Ontologie `category.knowledge` (siehe Abbildung D.7) enthält nur einen Zustand: Known. Dieser beschreibt, dass der Agent von der Instanz als entity bestimmte Füllwerte kennt. Known bezieht sich stets auf abgeleitete Attribute. Wissen über orthogonale Attribute muss über die entsprechende verdinglichte Zustandsklasse erlangt werden.

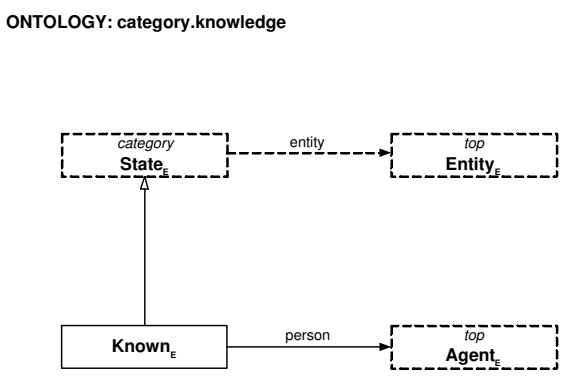


Abbildung D.7.: Die Ontologie `category.knowledge` zur Darstellung von Zuständen, die etwas über das Wissen über die Füllwerte einer Instanz aussagen.

D.3.4. category.possession

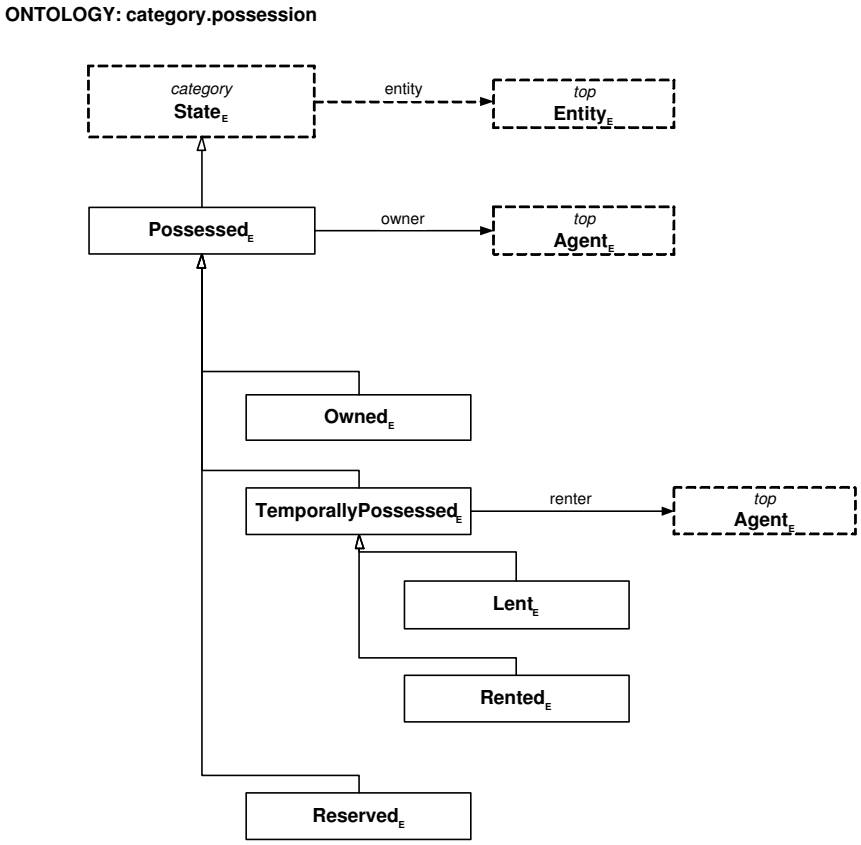


Abbildung D.8.: Die Ontologie `category.possession` zur Darstellung von Zuständen, die etwas über den Besitzzustand einer Entität aussagen.

Die Ontologie `category.possession` (siehe Abbildung D.8) drückt den Besitzzustand

einer Entität aus. Oberklasse ist Possessed, die aussagt, dass einer Entität entity ein Agent (also z.B. eine Person oder ein Unternehmen) zugewiesen ist (der owner), der ein Nutzungsrecht besitzt. Unterschieden werden folgende Abstufungen:

- Owned als echtes Eigentumsverhältnis. Dem Besitzer gehört die Entität. Er darf dieses Recht auch an Dritte weitergeben.
- Lent als Leihverhältnis. Der „Besitzer“ darf die Entität zeitweise und unentgeltlich nutzen. Dieses Recht darf er nicht an Dritte weitergeben.
- Rented als Leihverhältnis. Der „Besitzer“ darf die Entität gegen ein Entgelt zeitweise nutzen. Dieses Recht darf er nicht an Dritte weitergeben. Lent und Rented werden zu TemporallyOwned zusammengefasst. Dabei drückt das Attribut renter den Verleiher aus.
- Reserved als geplantes Nutzungsrecht. Dem „Besitzer“ wird zugesichert, die Entität im angegebenen Zeitraum nutzen zu dürfen.

D.3.5. category.assignment

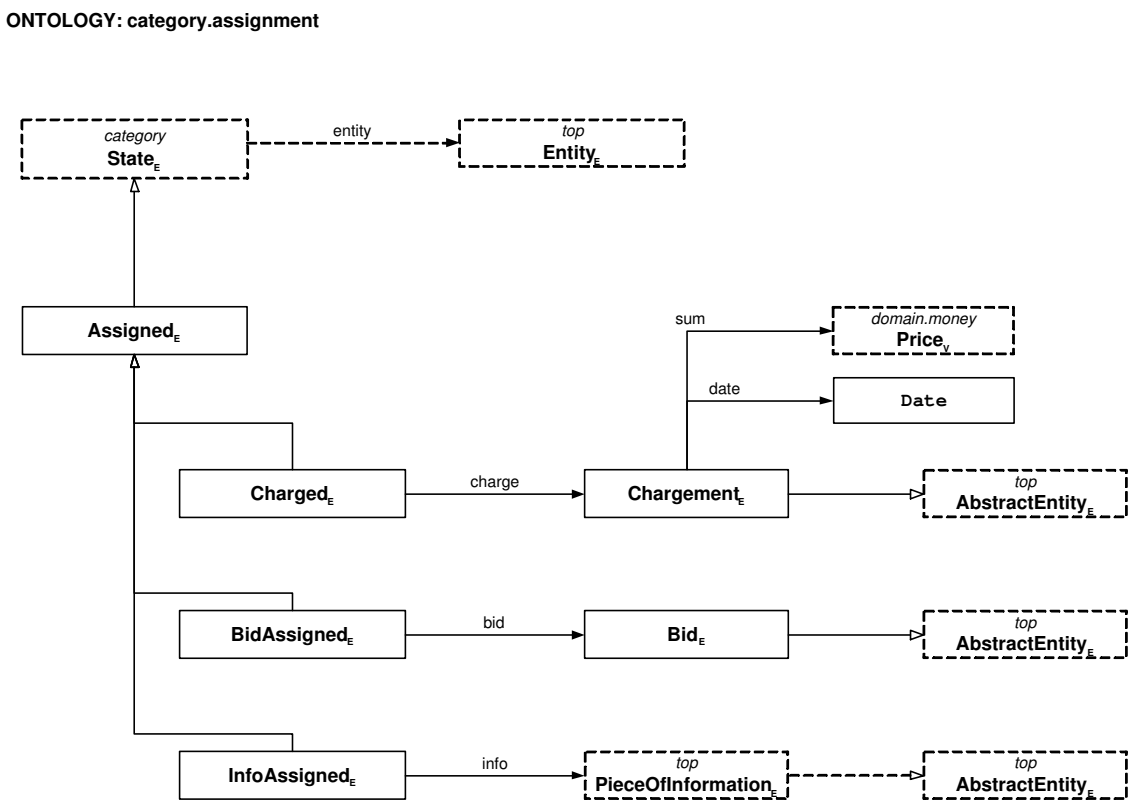


Abbildung D.9.: Die Ontologie `category.assignment` zur Darstellung von Zuständen, die etwas über Dinge aussagt, die einer Entität abhängig zugeordnet sind.

Die Ontologie `category.assignment` (siehe Abbildung D.9) enthält Zustände, die abhängige Zuordnungen von abstrakten Entitäten zu einer Entität beschreiben. Abhängig bedeutet, dass mit dem Verschwinden der Entität auch alle Zuordnungen und selbst das Zugeordnete nichtig sind. Unterschieden werden eine monetäre Belastung einer Entität ausgedrückt durch Charged, die Zuweisung eines Gebots ausgedrückt durch BidAssigned sowie die Zuordnung einer Informationseinheit durch InfoAssigned.

D.3.6. category.valuation

ONTOLOGY: category.valuation

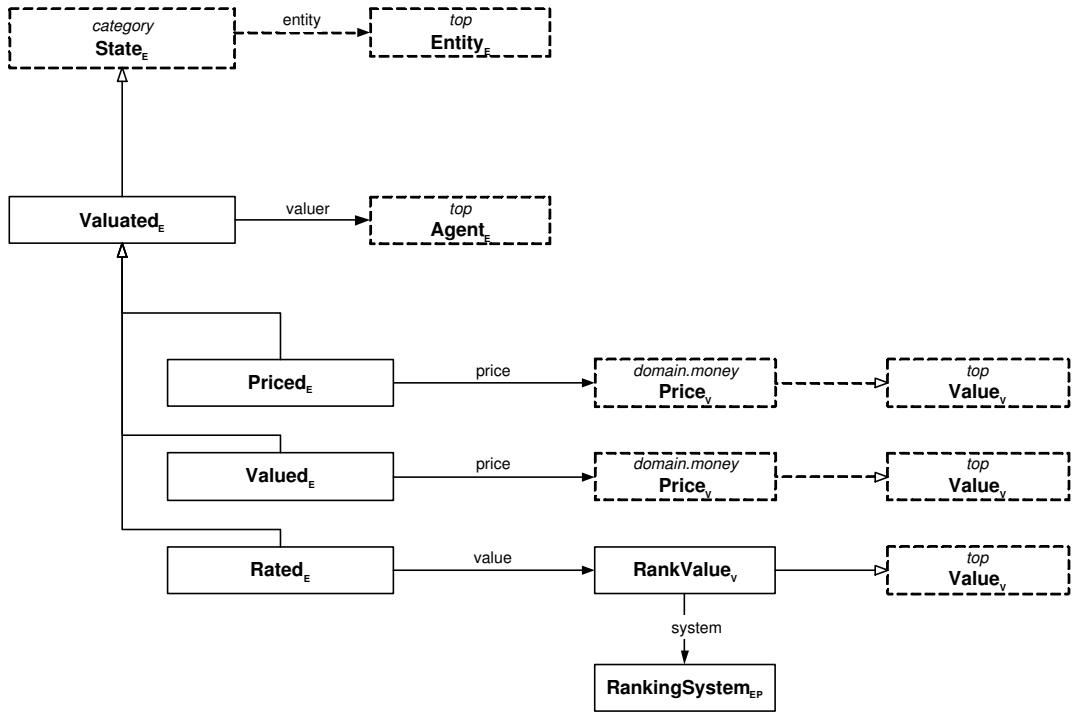


Abbildung D.10.: Die Ontologie `category.valuation` zur Darstellung von Zuständen, die etwas über die Bewertung einer Entität aussagen.

Die Ontologie `category.valuation` (siehe Abbildung D.10) sagt etwas über die Bewertung einer Entität durch einen Value aus. Die Oberklasse Valuated enthält dabei in valuer denjenigen Agenten, der die Bewertung vornimmt. Unterschieden werden

- Priced als Festsetzung eines Verkaufspreises für eine Entität.
- Valued als Festsetzung eines abstrakten Wertes für eine Entität, der nicht zwangsläufig als Verkaufspreis verwendet werden kann.
- Ranked als Abgabe einer Bewertung in einem domänenspezifischen Bewertungssystem. Dieses soll durch eine Unterklasse von RankValue ausgedrückt werden.

D.3.7. category.position

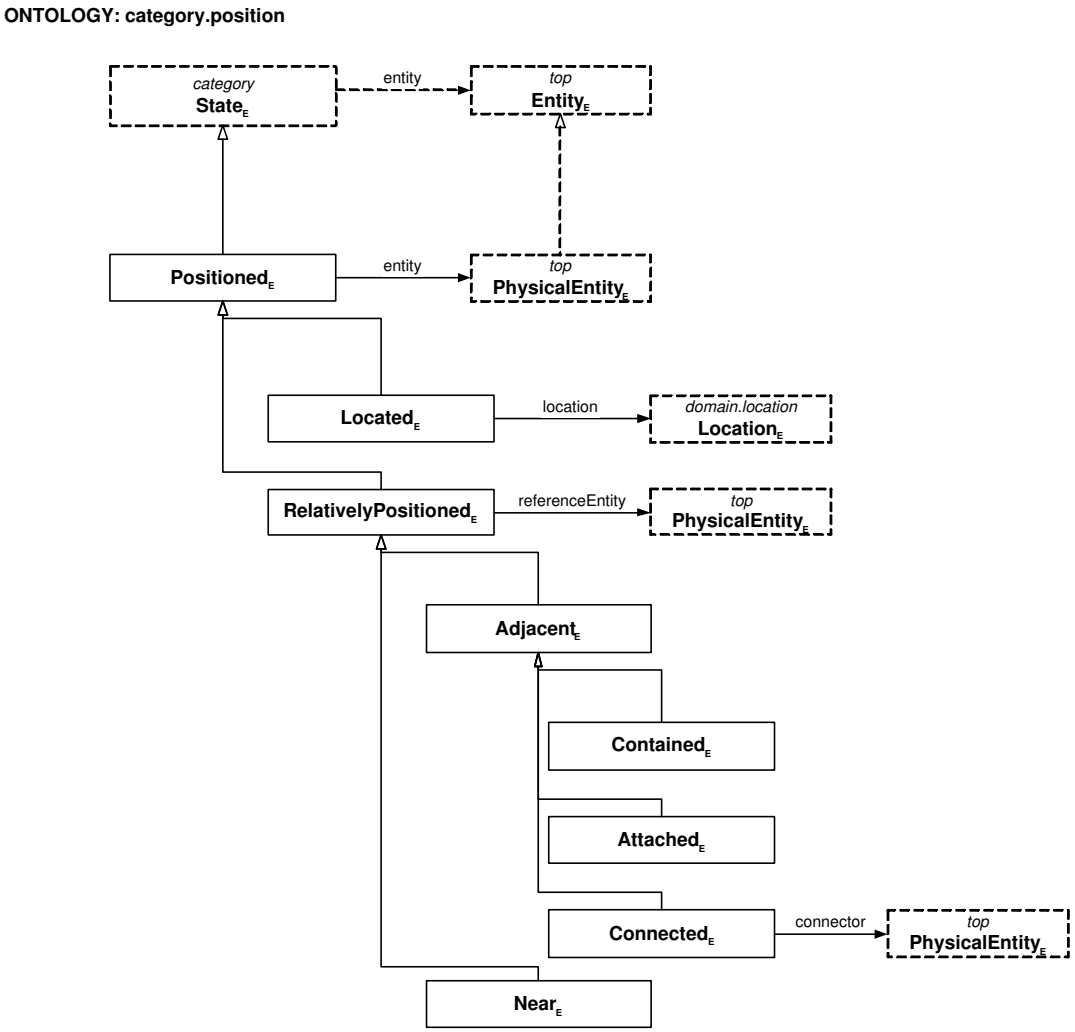


Abbildung D.11.: Die Ontologie `category.position` zur Darstellung von Zuständen, die etwas über die räumliche Position physikalischer Entitäten aussagen.

Die Ontologie `category.position` (siehe Abbildung D.11) enthält Zustände, die die räumliche Position von physikalischen Entitäten beschreiben. Unterschieden werden absolute Positionsangaben mittels Located sowie relative Positionsangaben mittels RelativelyPositioned in Bezug zu einer referenceEntity. Relative Positionsangaben können sein:

- Contained. Die Entität befindet sich innerhalb der Referenzentität.
- Attached. Die Entität ist direkt mit der Referenzentität verbunden.

- Connected. Die Entität ist indirekt über einen connector mit der Referenzentität verbunden.
- Near. Die Entität ist nicht mit der Referenzentität verbunden, befindet sich jedoch in deren Nähe.

D.3.8. category.relationship

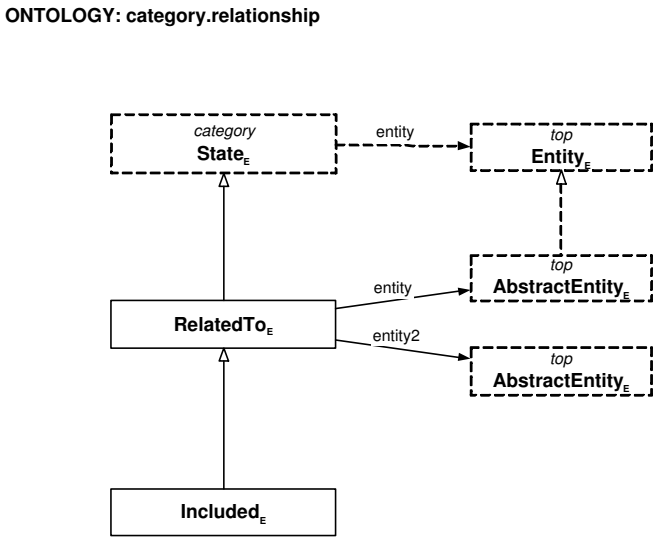


Abbildung D.12.: Die Ontologie `category.relationship` zur Darstellung von Zuständen, die etwas über die semantische Beziehung zwischen zwei abstrakten Entitäten aussagen.

Die Ontologie `category.relationsship` (siehe Abbildung D.12) enthält Zustände, um die semantische Beziehung zwischen zwei abstrakten Entitäten ausdrücken zu können. Hierzu dient der allgemeine Zustand RelatedTo, der die Entität in entity mit einer weiteren abstrakten Entität in entity2 vergleicht. Als eine vorgeschlagene Beziehung existiert Included, die besagt, dass die erste Entität die zweite semantisch enthält.

D.3.9. category.instantiation

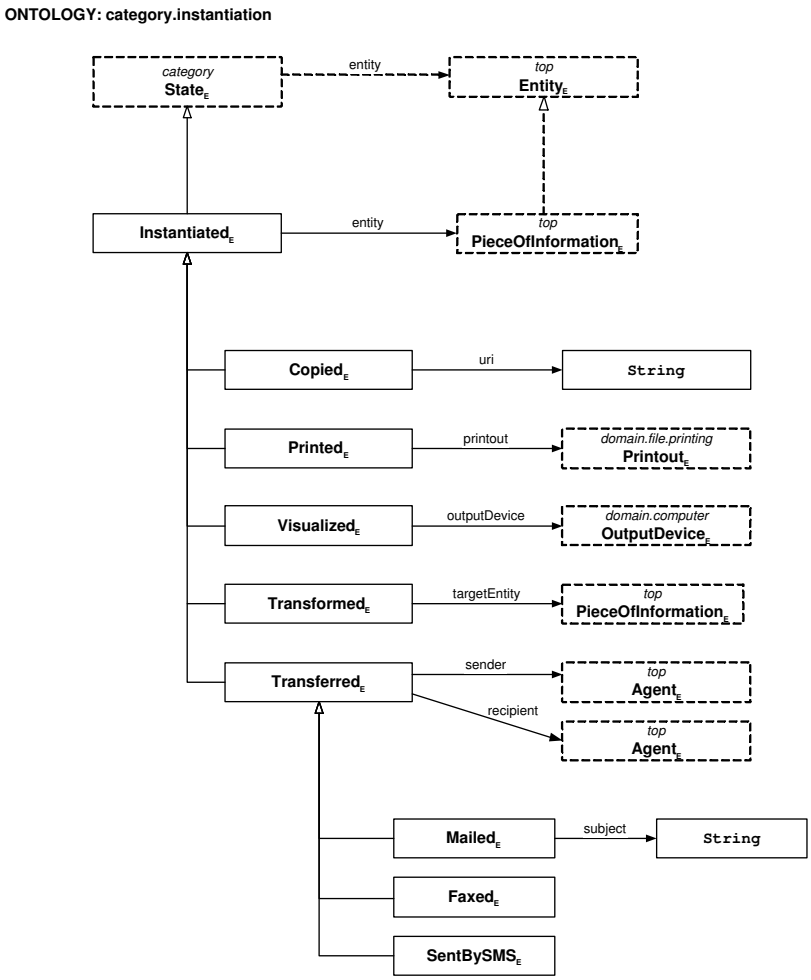


Abbildung D.13.: Die Ontologie `category.instantiation` zur Darstellung von Zuständen, die etwas über die Instanziierung von Informationseinheiten aussagen.

Die Ontologie `category.instantiation` (siehe Abbildung D.13) enthält Zustände, um die Instanziierung (d.h. die Darstellung oder Materialisierung) von Informationseinheiten (wie z.B. Dateien oder Datenbankeinträge) als entity ausdrücken zu können. Unterschieden werden:

- Copied, was besagt, dass eine Kopie der Informationseinheit an der angegebenen uri vorliegt.
- Printed, was besagt, dass ein Ausdruck der Informationseinheit vorliegt.
- Visualized, was besagt, dass die Informationseinheit für menschliche Betrachter sichtbar gemacht ist.

- Transformed, was besagt, dass die Informationseinheit in eine andere inhaltsgleiche Informationseinheit (etwa mit anderem Format) umgewandelt wurde.

- Transferred, was besagt, dass die Informationseinheit an eine andere Entität übermittelt wurde. Die Unterklassen Mailed, Faxed und SentBySMS bestimmen die Art der Übermittlung.

D.3.10. category.capability

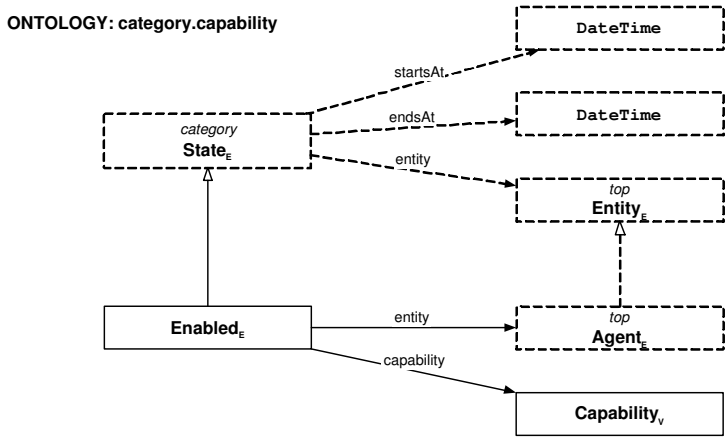


Abbildung D.14.: Die Ontologie `category.capability` zur Darstellung von Zuständen, die etwas über die Befähigung zum Durchführen einer Handlung aussagen.

Die Ontologie `category.capability` (siehe Abbildung D.14) beschreibt Zustände, welche die Befähigung zur Durchführung einer zeitlich ausgedehnten Handlung ausdrücken. Als Zustand dient hier Enabled, der besagt, dass die Person entity die Fähigkeit capability erlangt. Der Zeitraum der Fähigkeit wird durch die aus State geerbten Attribute startsAt und endsAt ausgedrückt.

D.3.11. category.access

Die Ontologie `category.access` (siehe Abbildung D.15) beschreibt Zustände, die die Zugriffsmöglichkeiten auf Informationseinheiten darstellen. Als allgemeiner Zustand existiert hier Accessible, der besagt, dass eine Person Zugriff auf eine Informationseinheit hat, diese also direkt verwenden kann. Genauere Unterklassen sind hier denkbar.

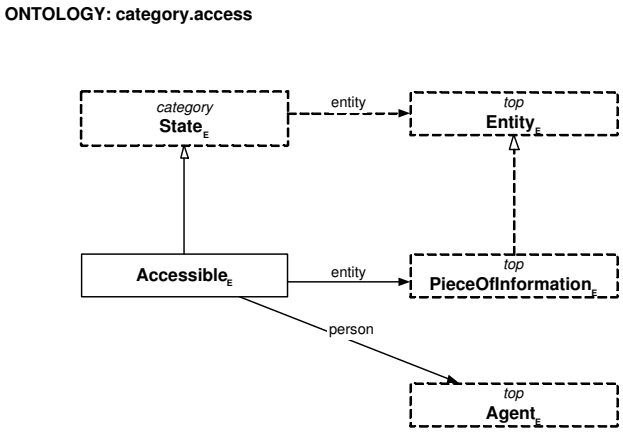


Abbildung D.15.: Die Ontologie `category.access` zur Darstellung von Zuständen, die etwas über die Zugriffsmöglichkeiten auf Informationseinheiten aussagen.

D.4. Wichtige Domänenontologien

D.4.1. domain.telecommunication

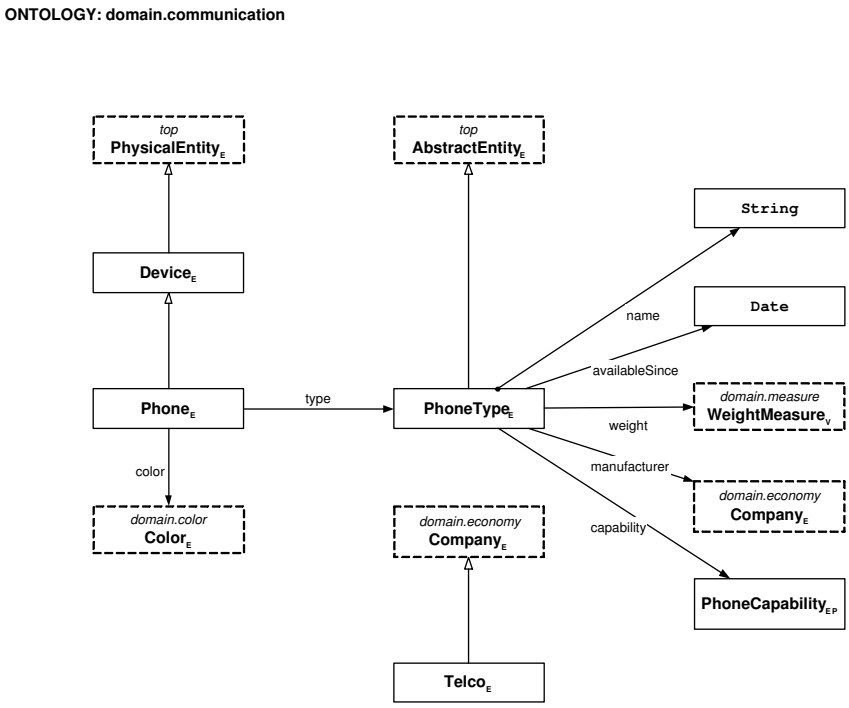


Abbildung D.16.: Ausschnitt aus der Ontologie `domain.telecommunication`.

Abbildung D.16 stellt eine kleinen Ausschnitt aus der Welt der Telekommunikation

vor. Im Mittelpunkt stehen ein konkretes Telefonendgerät (Phone) sowie ein abstraktes Telefonmodell (PhoneType).

D.4.2. domain.information

Abbildung D.17 zeigt einen Ausschnitt aus der äußerst wichtigen Domänenontologie `domain.information`. Diese dient der Beschreibung von Informationen in Form von Dateien. Zu sehen sind die verschiedenen Ebenen der Information: Auf unterster Ebene stellt eine Informationseinheit einen einfachen Bytestrom dar. Innerhalb einer Datei erhält er zusätzliche Merkmale wie Format oder Ersteller. Jede Datei repräsentiert ein Dokument, etwa einen Text, ein Bild etc. Dieses Dokument hat einen bestimmten Inhalt, der die Absicht des Dokumentes festlegt, etwa eine Erklärung, eine Zusammenfassung etc. An der Spitze steht die Entität, auf die sich der Inhalt bezieht.

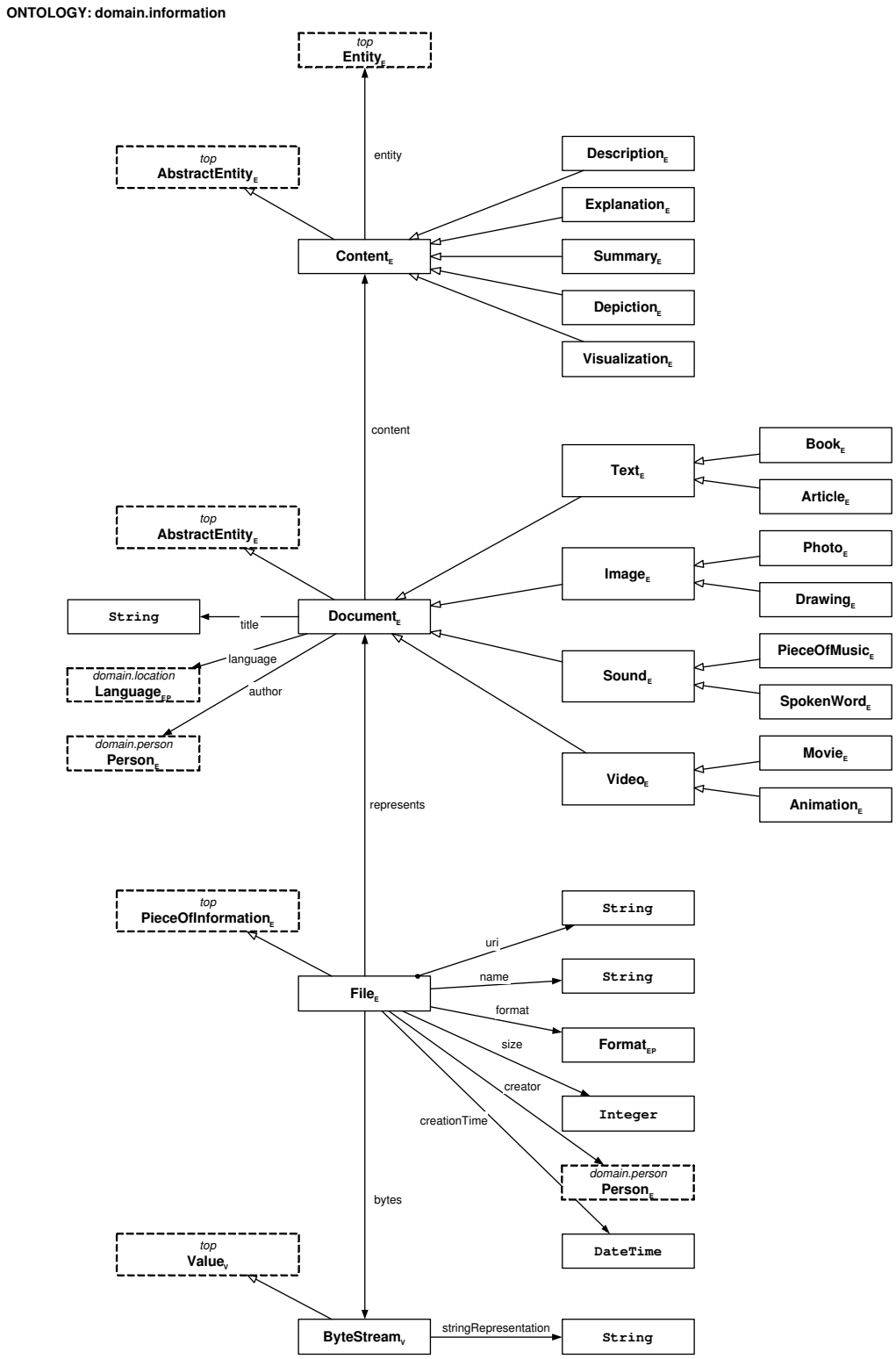


Abbildung D.17.: Ausschnitt aus der Ontologie `domain.information`.

Literaturverzeichnis

[1] Agarwal, S., S. Handschuh und S. Staab: *Annotation, Composition and Invocation of Semantic Web Services*. Journal on Web Semantics, 2(1):1–24, 2005.

[2] Akkiraju, R., J. Farrell, J. Miller, M. Nagarajan, , M.-T. Schmidt, A. Sheth und K. Verma: *Web Service Semantics - WSDL-S*. W3C Member Submission, November 2005. http://www.w3.org/Submission/WSDL-S/.

[3] Ambroszkiewicz, S.: *enTish: An Approach to Service Description and Composition*. In: *In Proceedings of the First European Workshop on Object Orientation and Web Services*, S. 49–53, Darmstadt, Juli 2003.

[4] Angele, J. und G. Lausen: *Ontologies in F-Logic*, Kap. 2, S. 29–50. International Handbooks on Information Systems. Springer, 2004.

[5] Ankolenkar, A., M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne und K. Sycara: *DAML-S: Web Service Description for the Semantic Web*. In: *Proceedings Of the First International Semantic Web Conference (ISWC)*, Sardinien, Italien, Juni 2002.

[6] Austin, J. L.: *How to Do Things with Words*. Harvard University Press, Cambridge, MA, USA, 1962.

[7] Avancha, S., A. Joshi und T. Finin: *Enhancing the Bluetooth Service Discovery Protocol*. Techn. Ber. TR-CS-01-08, Computer Science and Electrical Engineering, University of Maryland Baltimore County, August 2001.

[8] Baader, F., D. Calvanese, D. McGuinness, D. Nardi und P. Patel-Schneider: *Description Logic Handbook – Theory, Implementation and Applications*. Cambridge University Press, 2002.

[9] Balzer, S., T. Liebig und M. Wagner: *Pitfalls of OWL-S: A Practical Semantic Web Use Case*. In: *Proceedings of the Second International Conference on Service Oriented Computing*, S. 289–298, New York, NY, USA, Dezember 2004.

[10] Bansal, S. und J. M. Vidal: *Matchmaking of Web Services Based on the DAML-S Service Model*. In: *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, S. 926–927, Melbourne, Australien, 2003. ACM.

[11] Battle, S., A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su und S. Tabet: *Semantic Web Services Framework (SWSF) Overview*. SWSF Whitepaper, Mai 2005.

[12] Bechhofer, S., F. V. Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider und L. A. Stein: *OWL Web Ontology Language Reference*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/owl-ref/.

[13] Beckett, D.: *RDF/XML Syntax Specification*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

[14] Berardi, D., D. Calvanese, G. D. Giacomo, M. Lenzerini und M. Mecella: *Automatic Composition of E-services That Export Their Behavior*. In: *Proceedings of First International Conference On Service Oriented Computing (ICSOC-03)*, Trento, Italien, Dezember 2003.

[15] Bernrs-Lee, T.: *Notation 3*. W3C Design Note, 1998. http://www.w3.org/DesignIssues/Notation3.

[16] Bernrs-Lee, T., J. Hendler und O. Lassila: *The Semantic Web*. Scientific American, Mai 2001.

[17] Biron, P. V. und A. Malhotra: *XML Schema Part 2: Datatypes*. W3C Recommendation, Oktober 2004. http://www.w3.org/TR/xmlschema-2/.

[18] Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris und D. Orchard: *Web Service Architecture*. W3C Working Group Note, Februar 2004. http://www.w3.org/TR/ws-arch.

[19] Bosc, P., E. Damiani und M. Fugini: *Fuzzy Service Selection in a Distributed Object-Oriented Environment*. IEEE Transactions on Fuzzy Systems, 9(5):682–698, Oktober 2001.

[20] Box, D.: *A Guide to Developing and Running Connected Systems with Indigo*. MSDN Magazine, Januar 2004.

[21] Brickley, D. und R. V. Guha: *RDF Vocabulary Description Language: RDF Schema*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/rdf-schema/.

[22] Bronstein, Semendjajew, Musiol und Mühlig: *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, 3. Aufl., 1997.

[23] Bruijn, J. de, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer und D. Fensel: *The Web Service Modeling Language WSML*. WSML Final Draft D16.1v0.21, Oktober 2005. http://www.wsmo.org/TR/d16/d16.1/v0.21/.

[24] Bryson, J. J., D. Martin, S. A. McIlraith und L. A. Stein: *Semantic Web Services as Behavior-Oriented Agents*. IEEE Computer, 35(11):48–54, 2002.

[25] Buhler, P. und J. M. Vidal: *Semantic Web Services as Agent Behaviors*. In: Burg, B. (Hrsg.): *Agentcities: Challenges in Open Agent Environments*, S. 25–31. Springer-Verlag, 2003.

[26] Chakraborty, D., F. Perich, S. Avancha und A. Joshi: *DReggie: Semantic Service Discovery for M-Commerce Applications*. In: *Proceedings of the Workshop on Reliable and Secure Applications in Mobile Environment, 20th Symposiom on Reliable Distributed Systems*, New Orleans, LA, USA, Oktober 2001.

[27] Chinnica, R., M. Gudgin, J.-J. Moreau, J. Schlimmer und S. Weerawarana: *Web Services Descriptions Language (WSDL) Version 2.0*. W3C Working Draft, August 2004. http://www.w3.org/TR/wsdl20/.

[28] Ciaccia, P., D. Montesi, W. Penzo und A. Trombetta: *Imprecision and User Preference in Multimedia Queries*. In: *In Proceedings of the First International Symposium on Foundations of Information and Knowledge Systems (FoIKS2000)*, S. 50–71, Burg, Februar 2000.

[29] Constantinescu, I. und B. Faltings: *Efficient Matchmaking and Directory Services*. Techn. Ber. IC/2002/77, Swiss Federal Institute of Technology, Lausanne, Schweiz, November 2002. http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200277.pdf.

[30] Czajkowski, K., D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke und W. Vambenepe: *The WS-Resource Framework*. Globus Whitepaper, Mai 2004.

[31] Dolog, P. und W. Nejdl: *Challenges and Benefits of the Semantic Web for User Modelling*. In: *Proceedings of the Workshop on Adaptive Hypermedia and Adaptive Web-Based Systems (AH2003) at the Twelfth International World Wide Web Conference*, Budapest, Ungarn, Mai 2003.

[32] Domagalski, R. und B. König-Ries: *Möglichkeiten der Anfragebearbeitung in Mobilen Ad-Hoc-Netzwerken*. In: *Grundlagen und Anwendungen mobiler Informationstechnologie, Workshop des GI-Arbeitskreises Mobile Datenbanken und Informationssysteme*, Heidelberg, März 2004.

[33] Domingue, J., L. Cabral, F. Hakimpour, D. Sell und E. Motta: *IRS-III: A Platform and Insfrastructure for Creating WSMO-Based Semantic Web Services*. In: *First WSMO Implementation Workshop (WIW)*, Frankfurt, September 2004.

[34] Dumas, M., J. O'Sullivan, M. Heravizadeh, D. Edmond und A. Hofstede: *Towards a Semantic Framework for Service Description*. In: *9th International Conference on Database Semantics*, Hong-Kong, China, April 2001. Kluwer Academic Publishers.

[35] Ehrig, M., C. Tempich, S. Staab, F. V. Harmelen, R. Siebes, M. Sabou, J. Broekstra und H. Stuckenschmidt: *SWAP: Ontology-Based Knowledge Management with Peer-to-Peer*. In: *4th European Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS'03)*, S. 557–562, London, Grobritannien, April 2003.

[36] Fan, J., K. Barker, B. Porter und P. Clark: *Representing Roles and Purpose*. In: *In Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001)*, S. 38–43, Victoria, Kanada, 2001.

[37] Fensel, D. und C. Bussler: *The Web Service Modeling Framework WSMF*. Electronic Commerce Research and Applications, 1(2), 2002.

[38] Fensel, D., E. Motta, V. R. Benjamins, M. Crubezy, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, F. V. Harmelen, M. Musen, E. Plaza, G. Schreiber, R. Studer und B. Wielinga: *The Unified Problem-Solving Method Development Language UPML*. Knowledge and Information Systems, 5(1), 2002.

[39] Fischer, T.: *Entwicklung eines Kontainermanagers zur Unterstützung der Konfigurierung und Ausführung semantisch beschriebener Dienste*, März 2004. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[40] Fischer, T.: *Entwicklung und Durchführung eines Seminars zur Evaluation von DSD*, August 2005. Diplomarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[41] Fischer, T.: *Ergebnisse der Anwendung des Benchmarks auf DSD*. http://hnsp.inf-bb.uni-jena.de/DIANE/benchmark/, August 2005.

[42] Foster, I. und C. Kesselman: *Globus: A Toolkit-Based Grid Architecture*. In: Foster, I. und C. Kesselman (Hrsg.): *The Grid: Blueprint for a New Computing Infrastructure*, S. 259–278. Morgan Kaufmann, 1999.

[43] Foster, I. und C. Kesselman: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[44] Foster, I., C. Kesselman, J. Nick und S. Tuecke: *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. OGSA Whitepaper for the Open Grid Service Infrastructure WG, Global Grid Forum, Juni 2002.

[45] Foster, I. und C. Liu: *A Constraint Language Approach to Grid Resource Selection*. Techn. Ber. TR-2003-07, University of Chicago, IL, USA, 2003.

[46] Foster, I. T.: *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. In: *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, S. 1–4, London, Grobritannien, 2001. Springer-Verlag.

[47] Fowler, J., B. Perry, M. Nodine und B. Bargmeyer: *Agent-Based Semantic Interoperability in InfoSleuth*. SIGMOD Record, 28(1):60–67, März 1999.

[48] Gangemi, A., N. Guarino, C. Masolo, A. Oltramari und L. Schneider: *Sweetening Ontologies with DOLCE*. In: *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, S. 166–181, London, Grobritannien, 2002. Springer-Verlag.

[49] Gibbins, N., S. Harris und N. Shadbolt: *Agent-Based Semantic Web Services*. In: *World Wide Web Conference (WWW2003)*, Budapest, Ungarn, Mai 2003.

[50] Goble, C. und D. D. Roure: *The Grid: an application of the semantic web*. SIGMOD Record, 31(4):65–70, 2002.

[51] Grau, B. C.: *A Possible Simplification of the Semantic Web Architecture*. In: *Proceedings of the International WWW Conference*, New York, NY, USA, 2004.

[52] Gruber, T.: *A Translation Approach to Portable Ontology Specifications*. Knowledge Acquisition, 5:199–220, 1993.

[53] Gruninger, M.: *Process Specification Language: Principles and Applications*. AI Magazine, 24:63–74, 2003.

[54] Guarino, N. und C. Welty: *A Formal Ontology of Properties*. In: Dieng, R. und O. Corby (Hrsg.): *Proceedings of the 12th International Conference On Knowledge Engineering and Knowledge Management (EKAW-2000)*, Menlo Park, CA, USA, Oktober 2000. AAAI Press.

[55] Guarino, N. und C. A. Welty: *An Overview of OntoClean*. In: Staab, S. und R. Studer (Hrsg.): *Handbook on Ontologies*, Kap. 8, S. 151–159. Springer Verlag, 2004.

[56] Guo, R., J. Le und X. Xia: *Capability Matching of Web Services Based on OWL-S*. In: *International Workshop on Database and Expert Systems Applications (DEXA)*, S. 653–657, Kopenhagen, Dänemark, August 2005.

[57] Haas, H. und A. Brown: *Web Services Glossary*. W3C Working Group Note, Februar 2004. http://www.w3.org/TR/ws-gloss/.

[58] Haller, A., E. Cimpian, A. Mocan, E. Oren und C. Bussler: *WSMX - A Semantic Service-Oriented Architecture*. In: *Proceedings of the International Conference on Web Services (ICWS 2005)*, Orlando, FL, USA, Juli 2005.

[59] Hausmann, J. H., R. Heckel und M. Lohmann: *Model-Based Discovery of Web Services*. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, S. 324–331, San Diego, CA, USA, Juni 2004.

[60] Hayes, P.: *RDF Semantics*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/rdf-mt/.

[61] Heckmann, D.: *Introducing 'Situational Statements' as an Integrating Data Structure for User Modeling, Context-Awareness and Resource-Adaptive Computing*. In: *Proceedings of Workshop on Adaptivity and User Modeling in Interactive Software Systems*, Karlsruhe, Oktober 2003.

[62] Heckmann, D.: *A Specialized Representation for Ubiquitous Computing and User Modeling*. In: *Proceedings of the First Workshop on User Modelling for Ubiquitous Computing*, Johnstown, USA, Juni 2003.

[63] Heflin, J., R. Volz und J. Dale: *Requirements for a Web Ontology Language*. W3C Working Draft, März 2002. http://www.w3.org/TR/2002/WD-webont-req-20020307/.

[64] Herzog, T.: *Aktive VISIO-Schablonen zur grafischen Erstellung von DIANE-Dienstbeschreibungen*, Oktober 2004. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[65] Höllrigl, T., C. Schaa und F. Schell: *Entwicklung einer formalen Repräsentation für die DIANE-Dienstbeschreibung*, Mai 2004. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[66] Horrocks, I., P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof und M. Dean: *SWRL: A Semantic Web Rule Language – Combining OWL and RuleML.* W3C Member Submission, April 2004. http://www.w3.org/Submission/SWRL/.

[67] Jaeger, M. C., G. Rojec-Goldmann, G. Mühl, C. Liebetruth und K. Geihs: *Ranked Matching for Service Descriptions using OWL-S.* In: Müller, P., R. Gotzhein und J. B. Schmitt (Hrsg.): *Kommunikation in verteilten Systemen (KiVS 2005)*, Informatik Aktuell, S. 91–102, Kaiserslautern, Februar 2005. Springer.

[68] Keller, U., R. Lara, H. Lausen, A. Polleres und D. Fensel: *Automatic Location of Services.* In: *In Proceedings of the Second European Semantic Web Conference (ESWC 2005)*, Heraklion, Griechenland, Mai-Juni 2005.

[69] Keller, U., R. Lara, A. Polleres, I. Toma, M. Kifer und D. Fensel: *WSMO Web Service Discovery.* WSML Working Draft, November 2004. http://www.wsmo.org/2004/d5/d5.1/v0.1/.

[70] Keller, U., M. Stollberg und D. Fensel: *WOOGLE Meets Semantic Web Fred.* In: *In Proceedings of the Workshop on WSMO Implementations (WIW 2004)*, Frankfurt a.M., September 2004.

[71] Kifer, M., R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen und D. Fensel: *A Logical Framework for Web Service Discovery.* In: *Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications (SWS 2004)*, Hiroshima, Japan, November 2004.

[72] Kifer, M., G. Lausen und J. Wu: *Logical Foundations of Object-Oriented and Frame-Based Languages.* Journal of the ACM, 42(4):741–843, Juli 1995.

[73] Klein, M.: *Handbuch zur DIANE Service Description.* Techn. Ber. 2004-17, Universität Karlsruhe, Fakultät für Informatik, Dezember 2004.

[74] Klein, M.: *DSD-Ontologien.* http://hnsp.inf-bb.uni-jena.de/DIANE/ontology/, Juli 2005.

[75] Klein, M.: *Vermittlung von Diensten.* In: Höpfner, H., C. Türker und B. König-Ries : *Mobile Datenbanken und Informationssysteme*, Kap. 4, S. 59–83. dpunkt Verlag, Heidelberg, 2005.

[76] Klein, M. und J. Grünbauer: *Service Offer and Request Descriptions in Mobile Environments – A Position Paper.* In: *8. Workshop Des GI-Arbeitskreises 'Mobile Datenbanken und Informationssysteme' im Rahmen der BTW 2005*, Karlsruhe, März 2005.

[77] Klein, M. und B. König-Ries: *Multi-Layer Clusters in Ad-Hoc Networks - An Approach to Service Discovery*. In: *Proceedings of the First International Workshop on Peer-to-Peer Computing (Co-Located with Networking 2002)*, S. 187–201, Pisa, Italien, Mai 2002.

[78] Klein, M. und B. König-Ries: *A Process and a Tool for Creating Service Descriptions Based on DAML-S*. In: *Proceedings of the 4th VLDB Workshop on Technologies for E-Services (TES'03)*, S. 143–154, Berlin, September 2003.

[79] Klein, M. und B. König-Ries: *Combining Query and Preference - An Approach to Fully Automatize Dynamic Service Binding*. In: *Short Paper at IEEE International Conference on Web Services*, San Diego, CA, USA, Juli 2004.

[80] Klein, M. und B. König-Ries: *Coupled Signature and Specification Matching for Automatic Service Binding*. In: *Proceedings of the European Conference on Web Services (ECOWS 2004)*, Erfurt, September 2004.

[81] Klein, M. und B. König-Ries: *Integrating Preferences into Service Requests to Automate Service Usage*. In: *First AKT Workshop on Semantic Web Services*, Milton Keynes, Grobritannien, Dezember 2004.

[82] Klein, M., B. König-Ries und M. Müssig: *What is Needed for Semantic Service Descriptions - A Proposal for Suitable Language Constructs*. International Journal on Web and Grid Services (IJWGS), 1(2), 2005.

[83] Klein, M., B. König-Ries und P. Obreiter: *Lanes – A Lightweight Overlay for Service Discovery in Mobile Ad Hoc Network*. In: *Proceedings of the Third Workshop on Applications and Services in Wireless Networks (ASWN2003)*, Bern, Schweiz, Juli 2003.

[84] Klein, M., B. König-Ries und P. Obreiter: *Service Rings – A Semantical Overlay for Service Discovery in Ad Hoc Networks*. In: *Proceedings of the Sixth International Workshop on Network-Based Information Systems (NBIS2003), Workshop at DEXA 2003, Prag, Tschechien*, September 2003.

[85] Klein, M., B. König-Ries und P. Obreiter: *Stepwise Refinable Service Descriptions: Adapting DAML-S to Staged Service Trading*. In: *Proceedings of the First International Conference on Service Oriented Computing*, S. 178–193, Trento, Italien, Dezember 2003.

[86] Klusch, M., B. Fries, M. Khalid und S. Katia: *OWLS-MX: Hybrid Semantic Web Service Retrieval*. In: *Proceedings of the First International AAAI Fall Symposium on Agents and the Semantic Web*, Arlington, VA, USA, November 2005.

[87] König-Ries, B.: *Ein Verfahren zur Semi-Automatischen Generierung von Mediatorspezifikationen*. Doktorarbeit, Institute for Program Structures and Data Organization, Universität Karlsruhe, 1999. DISDBIS 55.

[88] Kröger, F.: *Temporal Logic of Programs*, Bd. 8 d. Reihe *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.

[89] Küster, U., M. Stern und B. König-Ries: *A Classification of Issues and Approaches in Service Composition*. In: *International Workshop on Engineering Service Compositions (WESC 2005) at the International Conference on Service Oriented Computing (ICSOC 2005)*, Amsterdam, Niederlande, Dezember 2005.

[90] Labrou, Y. und T. Finin: *A Proposal for a New KQML Specification*. Techn. Ber. TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, USA, Februar 1997.

[91] Lara, R., H. Lausen, S. Arroyo, J. de Bruijn und D. Fensel: *Semantic Web Services: Description Requirements and Current Technologies*. In: *International Workshop on Electronic Commerce, Agents, and Semantic Web Services, In Conjunction with the Fifth International Conference on Electronic Commerce (ICEC 2003)*, Pittsburgh, PA, USA, September 2003.

[92] Lara, R., D. Roman, A. Polleres und D. Fensel: *A Conceptual Comparison of WSMO and OWL-S*. In: *European Conference on Web Services (ECOWS 2004)*, Erfurt, September 2004.

[93] Li, J., M. Ozsu und D. Szafron: *Query languages in multimedia database systems*. Techn. Ber., Technischer Bericht, Department of Computing Science, The University of Alberta, Alberta, Kanada, 1995.

[94] Li, L. und I. Horrocks: *Matchmaking Using an Instance Store: Some Preliminary Results*. In: *Proceedings of the 2003 International Workshop on Description Logics (DL'2003), Poster*, Rom, Italien, 2003.

[95] Li, L. und I. Horrocks: *A Software Framework for Matchmaking Based on Semantic Web Technology*. In: *Proceedings of the Twelfth International World Wide Web Conference (WWW 2003)*, Budapest, Ungarn, Mai 2003.

[96] Ludwig, S. A. und P. V. Santen: *A Grid Service Discovery Matchmaker Based on Ontology Description*. In: *EuroWeb 2002 Conference*, Oxford, Grobritannien, Dezember 2002.

[97] Manola, F. und E. Miller: *Resource Description Framework (RDF)*. W3C Recommendation, Februar 2004. http://www.w3.org/RDF/.

[98] Martin, D. L., A. J. Cheyer und D. B. Moran: *The Open Agent Architecture: A Framework for Building Distributed Software Systems*. Applied Artificial Intelligence, 13(1-2):91–128, Januar-März 1999.

[99] McIlraith, S., T.C. Son und H. Zeng: *Semantic Web Services*. IEEE Intelligent Systems, Special Issue on the Semantic Web, 16(2):46–53, März/April 2001.

[100] McIlraith, S. und T. Son: *Adapting Golog for composition of semantic Web services*. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, Frankreich, April 2002.

[101] Medjahed, B., A. Bouguettaya und A. K. Elmagarmid: *Composing Web services on the Semantic Web*. The VLDB Journal, 12(4):333–351, 2003.

[102] Mika, P., M. Sabou, A. Gangemi und D. Oberle: *Foundations for DAML-S: Aligning DAML-S to DOLCE*. In: *In Proceedings of the AAAI Spring Symposium on Semantic Web Services*, Standford, CA, USA, März 2004.

[103] Missikoff, M.: *Harmonise: An Ontology-Based Approach for Semantic Interoperability*. ERCIM News, Special Theme: Semantic Web, 1(51), Oktober 2002.

[104] Mitra, N.: *SOAP*. W3C Recommendation, Juni 2003. http://www.w3.org/TR/soap/.

[105] Müssig, M.: *Schaffung des Vergleichers für DIANE Service Descriptions*, Februar 2005. Diplomarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[106] Nébel, I., B. Smith und R. Paschke: *A User Profiling Component with the Aid of User Ontologies*. In: *Proceedings of the Workshop 'Learning – Teaching – Knowledge – Adaptivity' (LLWA 03)*, Karlsruhe, 2003.

[107] Nepal, S., M. Ramakrishna und J. Thom: *A Fuzzy Object Query Language (FOQL) for Image Databases*. In: *Proceedings of the 6th International Conference on Database Systems for Advanced Applications (DASFAA'99)*, S. 117–124, Hsinchu, Taiwan, April 1999.

[108] Niles, I. und A. Pease: *Towards a Standard Upper Ontology*. In: Welty, C. und B. Smith (Hrsg.): *In Proceedings of the Second International Conference On Formal Ontology in Information Systems (FOIS-2001)*, Ogunquit, ME, USA, Oktober 2001.

[109] Noia, T. D., E. D. Sciascio, F. M. Donini und M. Mongiello: *A System for Principled Matchmaking in an Electronic Marketplace*. In: *Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Ungarn, Mai 2003.

[110] Oberle, D., R. Volz, B. Motik und S. Staab: *An Extensible Ontology Software Environment*. In: Staab, S. und R. Studer (Hrsg.): *Handbook on Ontologies*, Kap. 3, S. 311–333. Springer, 2004.

[111] Obreiter, P.: *A Case for Evidence-Aware Distributed Reputation Systems*. In: *Second International Conference on Trust Management (iTrust'04)*, S. 33–47, Oxford, Grobritannien, 2004. Springer LNCS 2995.

[112] Obreiter, P., B. König-Ries und M. Klein: *Stimulating Cooperative Behavior of Autonomous Devices - An Analysis of Requirements and Existing Approaches*. In: *Proceedings of the Second International Workshop on Wireless Information Systems (WIS2003)*, S. 71–82, Angers, Frankreich, 2003.

[113] Obreiter, P., B. König-Ries und G. Papadopoulos: *Engineering Incentive Schemes for Ad Hoc Networks - A Case Study for the Lanes Overlay*. In: *First EDBT-Workshop on Pervasive Information Management*, Heraklion, Griechenland, März 2004.

[114] Oltramari, A., A. Gangemi, N. Guarino und C. Masolo: *Restructuring WordNet's Top-Level: The OntoClean approach*. In: Simov, K. (Hrsg.): *In Proceedings of the Workshop on Ontologies and Lexical Knowledge Bases (OntoLex'2)*, S. 17–26, Las Palmas, Spanien, Mai 2002.

[115] O'Sullivan, J., D. Edmond und A. t. Hofstede: *Service Description: A Survery of the General Nature of Services*. Techn. Ber. FIT-TR-2003-02, Queensland University of Technology, 2002.

[116] O'Sullivan, J., D. Edmond und A. t. Hofstede: *What's in a Service? Towards Accurate Description of Non-Functional Service Properties*. Distributed and Parallel Databases, 12:117–133, 2002.

[117] Pan, J. und I. Horrocks: *Metamodeling Architecture of Web Ontology Languages*. In: *Proceedings of the Semantic Web Working Symposium*, S. 131–149, Standford, CA, USA, Juli 2001.

[118] Paolucci, M., T. Kawmura, T. Payne und K. Sycara: *Semantic Matching Of Web Services Capabilities*. In: *Proceedings of the First International Semantic Web Conference*, Sardinien, Italien, 2002.

[119] Paolucci, M., J. Soudry, N. Srinivasan und K. Sycara: *Untangling the Broker Paradox in OWL-S*. In: *In Proceedings Of the AAAI 2004 Spring Symposium on Semantic Web Services*, Stanford, CA, USA, März 2004.

[120] Patel-Schneider, P. F., P. Hayes und I. Horrocks: *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/owl-semantics/.

[121] Payne, T., M. Paolucci und K. Sycara: *Advertising and Matching DAML-S Service Descriptions*. In: *Semantic Web Working Symposium (SWWS)*, Stanford University, CA, USA, Juli 2001.

[122] Ponnekanti, S. R. und A. Fox: *SWORD: A Developer Toolkit for Web Service Composition*. In: *Proceedings Of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, Mai 2002.

[123] Preist, C.: *A Conceptual Architecture for Semantic Web Services*. In: *Third International Semantic Web Conference (ISWC)*, S. 395–409, Hiroshima, Japan, November 2004.

[124] Roman, D., H. Lausen und U. Keller: *Web Service Modeling Ontology (WSMO)*. WSMO Final Draft, April 2005. http://www.wsmo.org/2004/d2/v1.2/.

[125] Roure, D. de, N. Jennings und N. Shadbolt: *The Semantic Grid: A Future e-Science Infrastructure*. In: Berman, F., G. Fox und A. J. G. Hey (Hrsg.): *Grid Computing - Making the Global Infrastructure a Reality*, S. 437–470. John Wiley and Sons Ltd., 2003.

[126] Sabou, M., D. Richards und S. V. Splunter: *An Experience Report on Using DAML-S*. In: *Proceedings Of the Twelfth International World Wide Web Conference Workshop on E-Services and the Semantic Web (ESSW)*, Budapest, Ungarn, 2003.

[127] Santini, S. und R. Jain: *Similarity Queries in Image Databases*. In: *Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*, S. 646, Washington, DC, USA, 1996. IEEE Computer Society.

[128] Schlenoff, C., M. Gruninger, F. Tissot, J. Valois, J. Lubell und J. Lee: *The Process Specification Language (PSL): Overview and Version 1.0 Specification*. Techn. Ber. NISTIR 6459, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2000.

[129] Schmitt, H.: *Ein System zur verteilten Ontologieverwaltung für das DIANE-Projekt*, August 2005. Diplomarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[130] Scicluna, J., A. Polleres und D. Roman: *Ontology-Based Choreography and Orchestration of WSMO*. WSMO Working Draft, Oktober 2005. http://www.wsmo.org/TR/d14/v0.2.

[131] Sessions, R.: *Fuzzy boundaries: objects, components, and web services*. Queue, 2(9):40–47, 2005.

[132] Singh, M. P., A. S. Rao und M. P. Georgeff: *Formal Methods in DAI: Logic-Based Representation and Reasoning*. In: Weiss, G. (Hrsg.): *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, Kap. 8, S. 331–376. MIT Press, Cambridge, MA, USA, 1999.

[133] Sirin, E., J. Hendler und B. Parsia: *Semi-Automatic Composition of Web Services Using Semantic Descriptions*. In: *Proceedings of Web Services: Modeling, Architecture and Infrastructure. Workshop in Conjunction with ICEIS2003*, Angers, Frankreich, 2003.

[134] Sivashanmugam, K., K. Verma, A. Sheth und J. Miller: *Adding Semantics to Web Services Standards*. In: *Proceedings of the First International Conference on Web Services (ICWS03)*, S. 395–401, Las Vegas, NV, USA, Juni 2003.

[135] Smith, I. A. und P. R. Cohen: *Toward a Semantics for an Agent Communication Language Based on Speech Acts*. In: Shrobe, H. und T. Senator (Hrsg.): *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, S. 24–31, Menlo Park, California, 1996. AAAI Press.

[136] Smith, M. K., C. Welty und D. L. McGuinness: *OWL Web Ontology Language Guide*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/owl-guide/.

[137] Staab, S. und R. Studer: *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, Berlin, November 2003.

[138] Stern, M.: *Generierung von Anwendungsbeispielen für den Vergleicher von DIANE Service Descriptions*, Januar 2005. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[139] Stern, M.: *Konzeption und Realisierung von Dienstkomposition für DIANE Service Descriptions*, Dezember 2005. Diplomarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[140] Stollberg, M., D. Roman, R. Herzog und P. Zugmann: *SWF Goal and Service Description Language*. SWF-Deliverable D3, Juni 2004.

[141] Studer, R., V. R. Benjamins und D. Fensel: *Knowledge engineering: principles and methods*. Data Knowledge Engineering, 25(1-2):161–197, 1998.

[142] Sycara, K., M. Paolucci, M. V. Velsen und J. Giampapa: *The RETSINA MAS Infrastructure*. Autonomous Agents and Multi-Agent Systems, 7(1-2):29–48, 2003.

[143] Sycara, K., S. Widoff, M. Klusch und J. Lu: *LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace*. Autonomous Agents and Multi-Agent Systems, 5:173–203, 2002.

[144] Sycara, K. P., M. Klusch, S. Widoff und J. Lu: *Dynamic Service Matchmaking Among Agents in Open Information Environments*. SIGMOD Record, 28(1):47–53, 1999.

[145] Tangmunarunkit, H., S. Decker und C. Kesselman: *Ontology-Based Resource Matching in the Grid – the Grid Meets the Semantic Web*. IEEE Intelligent Systems, 16(2), 2001.

[146] Toma, I., D. Roman und K. Iqbal: *WSMO and Grid*. WSMO Working Draft, D25.1, November 2004.

[147] Trastour, D., C. Bartolini und J. Gonzalez-Castillo: *A Semantic Web Approach to Service Description for Matchmaking of Services*. In: *Proceedings of the International Semantic Web Working Symposium (SWWS)*, Stanford, CA, USA, 2001.

[148] Tuecke, S., K. Czajkowski, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm und D. Snelling: *Open Grid Services Infrastructure (OGSI) Version 1.0*. OGSA Whitepaper, Juni 2003.

[149] Verma, K., K. Gomadam, A. P. Sheth, J. A. Miller und Z. Wu: *The METEOR-S Approach for Configuring and Executing Dynamic Web Processes*. Techn. Ber., University of Georgia, Computer Science Department, Juni 2005.

[150] Verma, K., A. Mocan, M. Zaremba, A. Sheth, J. A. Miller und C. Bussler: *Linking Semantic Web Service Efforts – Integrating WSMX and METEOR-S*. In: *Proceedings of the Second International Workshop on Semantic and Dynamic Web Processes at International Conference on Web Services (ICWS 2005)*, Orlando, FL, USA, Juli 2005.

[151] Verma, K., K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar und J. Miller: *METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services*. Journal of Information Technology and Management, Special Issue on Universal Global Integration, 6(1):17–39, 2005.

[152] Welty, C. A. und D. A. Ferrucci: *What's in an Instance?*. Techn. Ber. No. 94-18, RPI Computer Science Department, Troy, NY, USA, 1994. http://www.cs.vassar.edu/faculty/welty/papers/instances/instances.pdf.

[153] Wileden, J. C., A. L. Wolf, W. R. Rosenblatt und P. L. Tarr: *Specification-level interoperability*. Communications of the ACM, 34(5):72–87, 1991.

[154] Wooldridge, M. und N. R. Jennings: *Agent Theories, Architectures, and Languages: A Survey*, Kap. 1, S. 1–22. Springer Verlag, Berlin, 1995.

[155] Wroe, C., R. Stevens, C. Goble, A. Roberts und M. Greenwood: *A Suite of DAML+OIL Ontologies to Describe Bioinformatics Web Services and Data*. International Journal of Cooperative Information Systems, 1(12):597–624, 2003.

[156] Wu, D., B. Parsia, E. Sirin, J. A. Hendler und D. S. Nau: *Automating DAML-S Web Services Composition Using SHOP2.*. In: *Proceedings of the Second International Semantic Web Conference(ISWC2003)*, Sanibel Island, FL, USA, Oktober 2003.

[157] Younas, K.-M. C., M. C.-C. Lo und T.-H. Tan: *Fuzzy Matchmaking for Web Services*. In: *19th International Conference on Advanced Information Networking and Applications (AINA2005)*, Tamkang University, Taiwan, März 2005.

[158] Yussupova, N., B. König-Ries, D. V. Popov und I. A. Vaynerman: *Personalizing the Usage of Complex Services*. In: *In Proceedings of the 7th International Workshop on Computer Science and Information Technologies (CSIT2005)*, Ufa, Russland, September 2005.

[159] Zaremski, A. M.: *Signature and Specification Matching*. Doktorarbeit, Carnegie Mellon University, 1996.

[160] Zaremski, A. M. und J. M. Wing: *Specification matching of software components*. In: *Proceedings Of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, S. 6–17. ACM Press, 1995.

[161] Zhang, R., I. B. Arpinar und B. Aleman-Meza: *Automatic Composition of Semantic Web Services*. In: *Proceedings Of the 2003 International Conference On Web Services (ICWS'03)*, Las Vegas, NV, USA, Juni 2003.

Dienstorientierte Architekturen werden nur dann ihr volles Potenzial entwickeln können, wenn es gelingt, Dienste fallweise und zur Laufzeit einzubinden. In der Praxis ist dafür eine automatische und korrekte Dienstnutzung nötig. Ziel dieser Dissertation war daher die Schaffung einer dafür geeigneten semantischen Dienstbeschreibungssprache und ihrer Verarbeitungsmechanismen. Ihr Ansatz ist es, die Basis der Sprache in zwei Teile zu zerlegen: einen leichtgewichtigen Teil zur Erfassung des Domänenwissens sowie zusätzliche spezialisierte Elemente zur Erfassung der Charakteristika von Diensten. Es entstehen so Beschreibungen, die die erreichbaren Zustände von Diensten in den Mittelpunkt stellen, alle weiteren Aspekte darin integrieren und sich so effizient vergleichen lassen.

ISBN 3-86644-013-8

www.uvka.de