FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Entwicklung eines semantisch angereicherten Software Repository

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Sobotka
Matrikelnummer 0826407

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Roman Popp

Wien, 20.08.2014

(Unterschrift Michael Sobotka) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Michael Sobotka
Czermakstraße 18/6/4, 2000 Stockerau

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Michael Sobotka)

Danksagung

Zuallererst möchte ich mich bei meinen Eltern, Hilde und Wolfgang Sobotka, für ihre grenzenlose Unterstützung und den uneingeschränkten Rückhalt über die Jahre hinweg bedanken. Ohne euch beide hätte ich es nie so weit geschafft und der gesamte Platz in dieser Arbeit würde nicht reichen um auszudrücken, wie dankbar ich euch für alles bin, was ihr jemals – nicht nur während meiner Studienzeit – für mich getan habt.
An dieser Stelle deshalb ein ehrliches und aus tiefstem Herzen empfundenes

Danke.

Dank und Anerkennung gebührt auch meinen Betreuern; Prof. Hermann Kaindl, der nicht davor zurückschreckte, mir auch noch nach Mitternacht meine Fragen per Email zu beantworten und der stets einen zeitnahen Termin für mich und meine Anliegen übrig hatte; Dr. Roman Popp und Ralph Hoch für ihr Feedback und die Beantwortung meiner Fragen und nicht zuletzt Dr. Dominik Ertl, der sich tapfer durch meinen Code gewühlt und selbigen am Instituts-Server zum Leben erweckt hat. Ich habe mich stets rundum gut betreut gefühlt.
Abschließend danke ich meinen Studienkollegen, allen voran Alex, Jürgen, Katrin, Stefan, Lukas und Thomas, mit denen ich so manche Lehrveranstaltung gemeinsam absolviert habe und in deren Gegenwart stets so etwas wie „Studenten-Feeling“ bei mir aufgekommen ist. Ihr habt mich allesamt auf die ein oder andere Art inspiriert, sei es durch euer hohes Maß an Motivation und Ehrgeiz, eure unterschiedlichen Blickwinkel und Denkansätze, eure Liebe zum japanischen Film aus den 50ern oder schlichtweg durch euren Sachverstand und eure Programmierkünste. Es war mir eine Ehre, euch kennengelernt zu haben und ich hoffe, man läuft sich in Zukunft hin und wieder mal über den Weg.

Abstract

Due to the ongoing progress of technology, not only are technical aspects of software development, such as programming languages or paradigms, constantly changing but also the structure of software itself. The current trend is shifting towards component- or service-oriented architecture, which understand software as an interaction of individual components and/or services. Ideally, more complex services can be created through the composition of existing basic services.
With regard to software reusability, the efficient and precise discovery of services is becoming increasingly important as the number of existing software components keeps growing. Traditional solutions such as the division of services into predefined categories or purely text-based search techniques are often imprecise, inexpressive, require extensive manual assessment of results on the part of the user and thus generally show a high discrepancy between the returned results and the actual needs of the user.
In this work, the *Semantically Enhanced Software Repository* (SEnSoR) has been developed. SEnSoR is a repository capable of managing both traditional software components in various programming languages, as well as (Semantic) Web services. In contrast to existing solutions such as UDDI, SEnSoR allows for the discovery of existing services based also on their semantic description through a combination of text-based search methods with ontology-based techniques from the field of Semantic Web.
The evaluation of SEnSoR and a comparison with a number of current Semantic Web service matchmakers show that SEnSoR achieves very good results in terms of precision (i.e., the quality of the returned services in relation to a given search query) and query performance. Even with large numbers of services search requests are answered within milliseconds. In combination with a user friendly search API similar to modern (Web) search engines, SEnSoR can thus be used as a useful development tool within existing software environments that aids its users in discovering and selecting suitable services, for instance in the context of business process modeling.

Kurzfassung

Bedingt durch den permanenten technologischen Fortschritt verändern sich im Laufe der Jahre nicht nur technische Aspekte der Softwareentwicklung wie eingesetzte Programmiersprachen oder -paradigmen, sondern auch die Struktur von Software selbst. Ein aktueller Trend geht hin zu komponenten- oder serviceorientierten Architekturen, welche Software als Zusammenspiel einzelner Komponenten bzw. Services auffassen. Komplexere Services lassen sich im Idealfall durch die Komposition bestehender Basis-Services kreieren.
In Bezug auf die Wiederverwendbarkeit kommt damit der effizienten und präzisen Auffindung bestehender Services mit zunehmender Fülle an existierenden Softwarekomponenten immer größere Bedeutung zu. Klassische Lösungen wie die Einteilung von Services in vordefinierte Kategorien oder rein textbasierte Suchverfahren sind oft unpräzise, ausdrucksschwach, erfordern eine umfassende manuelle Ergebnis-Beurteilung aufseiten der Nutzerin und weisen damit allgemein eine hohe Diskrepanz zwischen den zurückgelieferten Ergebnissen und dem eigentlichen Bedarf der Nutzerin auf.
Im Rahmen dieser Arbeit wurde das *Semantically Enhanced Software Repository* (SEnSoR) entwickelt. Dabei handelt es sich um ein Repository, welches sowohl traditionelle Softwarekomponenten in unterschiedlichen Programmiersprachen als auch (Semantic) Web Services in einheitlicher Weise verwaltet. Im Unterschied zu existierenden Lösungen wie beispielsweise UDDI ermöglicht SEnSoR durch die Kombination von textbasierten Suchverfahren mit ontologiegestützten Techniken aus dem Gebiet des Semantic Web die Auffindung vorhandener Services bzw. Methoden auch auf Basis deren semantischer Beschreibung.
Die Evaluierung und der Vergleich mit einer Reihe von aktuellen Semantic Web Service Matchmakern zeigen, dass SEnSoR sehr gute Resultate hinsichtlich der Präzision – also der Qualität von zurückgelieferten Services in Bezug auf eine Suchanfrage – liefert und auch bei großer Anzahl von Services Anfragen im Millisekundenbereich beantwortet. In Verbindung mit einer einfach zu benutzenden Such-API, welche sich an modernen (Web-)Suchmaschinen orientiert, lässt sich SEnSoR damit als hilfreiches Entwicklerwerkzeug innerhalb bestehender Softwarelandschaften einsetzen, welches seine Nutzerinnen bei der Auswahl von passenden Services, etwa auf dem Anwendungsgebiet der Modellierung von Geschäftsprozessen, unterstützt.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Die Entwicklung von Software unterliegt mit dem konstanten technologischen Fortschritt einer beständigen Evolution. Über die Jahrzehnte hinweg veränderten sich nicht nur technische Aspekte wie verwendete Plattformen, Protokolle oder Programmiersprachen, sondern auch die Struktur von Software an sich.

Waren Software-Systeme früher oft als monolithische Anwendungen konzipiert, begreifen moderne komponenten- oder serviceorientierte Architekturen ein Gesamtsystem vielmehr als Zusammenspiel von lose gekoppelten und austauschbaren Einzelmodulen. Das Paradigma des *Service Oriented Computing* (SOC) betrachtet ganze Geschäftsprozesse als Zusammensetzung von Services [52]. Ausgehend von Basis-Services, die eine gewisse Grundfunktionalität implementieren, lassen sich durch deren Komposition komplexere Aufgabenstellungen lösen.

Im Rahmen der Wiederverwendbarkeit von Software kommt dem Gebiet des *Service Discovery*, also der Auffindung passender Services, mit der zunehmenden Fülle an existierenden Softwarekomponenten und (Web) Services immer größere Bedeutung zu. Trotz der regen, aktiven Forschung auf diesem Gebiet besteht immer noch das Grundproblem der Diskrepanz zwischen dem Bedarf der Nutzerin in Form ihrer formulierten Suchanfrage und der von einem Repository zurückgelieferten Services [49].

Mit dem Aufschwung des *Semantic Web* ergeben sich unter der Einbindung von Ontologien neue Möglichkeiten, (Web) Services nicht nur auf syntaktischer, sondern darüber hinaus auch auf semantischer Ebene zu beschreiben. Dies ermöglicht eine Entwicklung weg von klassischen, kategoriebasierten Discovery-Mechanismen hin zu solchen, die auch das spezifizierte Verhalten von Services miteinbeziehen.

Diese Arbeit widmet sich dem Design und der Entwicklung eines Software-Repository, welches sowohl traditionelle Softwarekomponenten als auch (Semantic) Web Services unterstützt und die Auffindung von Services (auch) auf Basis ihrer semantischen Beschreibung ermöglicht.

1.1 Motivation

Ein populäres Anwendungsgebiet für den Einsatz eines solchen Software-Repository ist die Modellierung und Exekution von Geschäftsprozessen – ein aktives Forschungsgebiet, das in den letzten Jahren großes wissenschaftliches Interesse erfahren hat.

Für die Modellierung von Geschäftsprozessen existieren eine Reihe von Modellierungssprachen wie *Business Process Model and Notation* (BPMN). In solchen Sprachen werden Prozesse jeweils über einzelne Aktivitäten oder sogenannte *Tasks* modelliert, die mittels Kontrollflusselementen wie Schleifen oder Verzweigungen zu komplexen Abläufen zusammengesetzt werden können und letztendlich eine bestimmte Geschäftslogik implementieren.

Für die Modellierer solcher Prozesse existiert mit *Activiti*,[1] *jBPM*,[2] oder *ActiveVOS*[3] bereits vielfältige Tool-Unterstützung. Besagte Programme bieten jeweils eine graphische Benutzeroberfläche mit einem „What you see is what you get"-Editor, welcher die Modellierung auch für Nicht-Programmierer ermöglichen soll und darüber hinaus eine jeweilige *process engine*, auf welcher zuvor modellierte Prozesse ausgeführt werden können [31].

Damit ein Prozess jedoch exekutiert werden kann, ist es nötig, den einzelnen abstrakten Aktivitäten wie beispielsweise „sende Email an den Kunden" jeweils eine konkrete Implementierung („SendEmailService") zuzuweisen. Eine Modelliererin steht nun also vor dem Problem, eine passende Softwarekomponente finden und auswählen zu müssen, die ihre gewünschte Funktionalität abdeckt und noch dazu in unterschiedlichen Beschreibungssprachen spezifiziert sein kann.

Existierende Lösungen wie beispielsweise *Universal Description, Discovery and Integration* (UDDI) für Web Services oder die in aktuellen Modellierungswerkzeugen enthaltenen Repositories kategorisieren Softwarekomponenten bzw. Services oft in vorgegebenen Kategorien und bieten bestenfalls eine auf Schlüsselwörtern basierende Suchfunktion. Wie in Abbildung 1.1 zu erkennen ist, erfordert dieser Ansatz jedoch das nachträgliche, manuelle Durchstöbern und Bewerten der gefundenen Ergebnisse durch den Benutzer, da die Zugehörigkeit zu einer bestimmten Kategorie alleine nur eine grobe Einschätzung über die tatsächliche Funktionalität eines Service zulässt [50]. Da Ersteller und Nutzer eines Service zudem nicht notwendigerweise über das selbe Vokabular verfügen, sollte ein Software-Repository Komponenten nicht bloß aufgrund einer reinen textuellen Übereinstimmung auffindbar machen.

Mit steigender Anzahl an Komponenten wird es zudem zunehmend wichtiger, nicht einfach nur eine Auswahl an Ergebnissen anzubieten, sondern diese in Anlehnung an moderne Suchmaschinen in einem *Ranking*-Schritt vorzusortieren, um die Nutzerin bei der Auswahl der am besten passenden Ergebnisse zu unterstützen.

Die Suche nach Methoden auf Basis deren Input- bzw. Output-Parameter(typen) ist ebenfalls problematisch, da Methoden je nach Technologie an unterschiedliche Typsysteme gebunden sind (etwa jenes einer objektorientierten Programmiersprache oder XML). Zudem können Methoden nicht immer aufgerufen werden, auch wenn diese die benötigte Funktionalität vollständig abde-

[1] Activiti: http://activiti.org
[2] jBPM: http://jbpm.jboss.org
[3] ActiveVOS: http://www.activevos.com

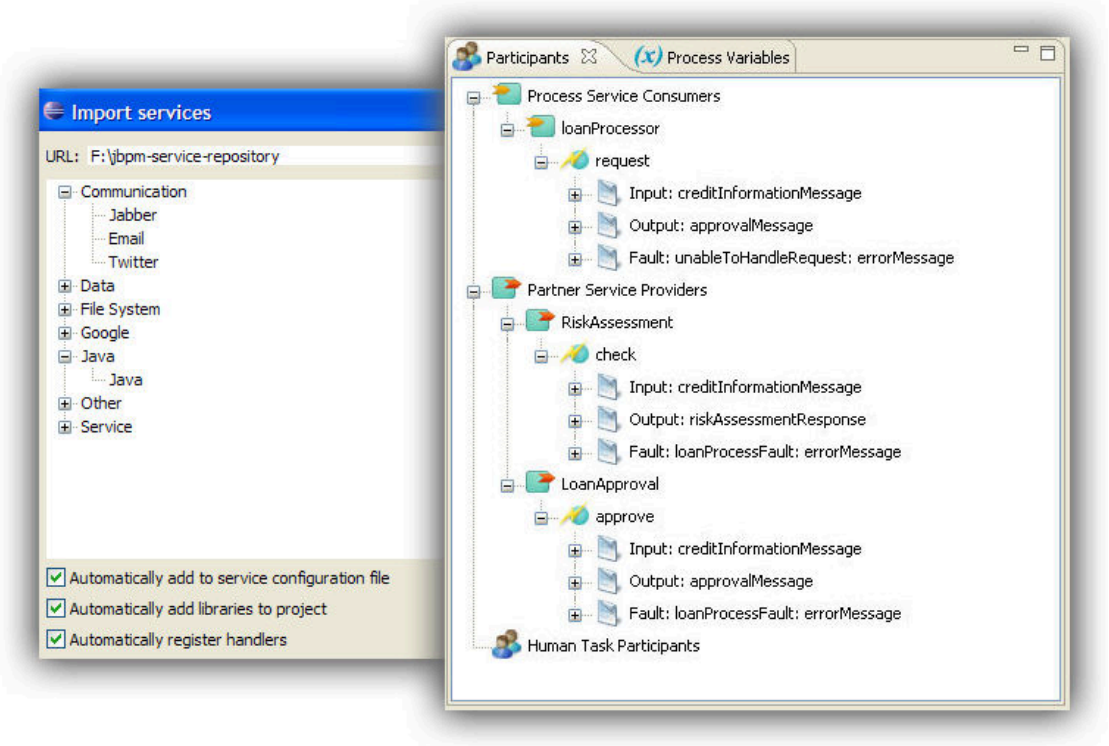

Abbildung 1.1: Service Discovery in jBPM 6.1.0 und ActiveVOS 9.2.

cken. Eventuelle Vor- oder Nachbedingungen können den erfolgreichen Aufruf verhindern, etwa wenn der Versand einer Email einen gültigen Login voraussetzt.

Um seine Nutzerinnen also bestmöglich bei der Auswahl passender Software zu unterstützten, sollte ein modernes Software-Repository die genannten Umstände berücksichtigen.

1.2 Problemstellung

Im Rahmen dieser Arbeit sollten die beiden folgenden Fragen beantwortet werden:

1. Wie können sowohl traditionelle Softwarekomponenten in unterschiedlichen Programmiersprachen als auch (Semantic) Web Services in einem Software Repository einheitlich verwaltet werden?

2. Wie kann die semantische Annotation von Services in Verbindung mit Methoden aus dem Bereich des Semantic Web dabei helfen, die Auswahl passender Services zu präzisieren?

1.3 Zielsetzung & Methodisches Vorgehen

Das Ziel dieser Arbeit sind das Design und die Entwicklung eines Software-Repository zur einheitlichen Speicherung und Verwaltung von traditionellen Softwarekomponenten sowie (Semantic) Web Services.
Das Repository sollte im Hinblick auf die *Service Discovery* klassische textbasierte Lösungen zur Auffindung von Komponenten bzw. Services mit Ansätzen aus dem Bereich des Semantic Web erweitern, um die Suche nach Services und der darin gekapselten Methoden auch auf Basis ihrer semantischen Beschreibung zu ermöglichen.
Der Fokus dieser Arbeit lag dabei auf funktionalen Anforderung von Services bzw. Softwarekomponenten. Das Repository ist explizit für die nicht-globale Verwendung innerhalb einer vertrauenswürdigen Domäne – wie beispielsweise einer Firma – und für die Interaktion mit einem Menschen ausgelegt. Ein beispielhaftes Anwendungsgebiet ist die in Abschnitt 1.1 besprochene Unterstützung bei der Auswahl passender Services im Rahmen der Modellierung von Geschäftsprozessen.
Ausgehend von einer Literaturrecherche wurden zunächst die Anforderungen an das Repository ermittelt. Basierend darauf wurde dessen Architektur erarbeitet und in Form einer prototypischen Implementierung namens SEnSoR (*Semantically Enhanced Software Repository*) umgesetzt, welche das Hauptergebnis dieser Arbeit darstellt. Die Evaluierung von SEnSoR erfolgte unter besonderem Fokus auf Performanz und Präzision der integrierten Suchfunktionalität mittels existierender Tools zur Evaluierung von Semantic Web Service Matchmakern auf Basis von Standard-Metriken aus dem Umfeld des Information Retrieval, was eine Gegenüberstellung mit vergleichbaren Lösungen auf diesem Gebiet ermöglichte.

1.4 Aufbau der Arbeit

Kapitel 2 beschäftigt sich mit den Grundlagen rund um die Themen Komponenten, (Web) Services sowie Semantic Web und geht insbesondere auf Konzepte bzw. Technologien ein, deren Kenntnis im weiteren Verlauf dieser Arbeit erforderlich ist.
In Kapitel 3 werden bestehende Software Repositories und der aktuelle Forschungsstand auf dem Gebiet des Service Discovery (oder auch Service Matchmakings) besprochen.
Aufbauend auf diese beiden Kapitel wird in Kapitel 4 das im Rahmen dieser Arbeit als Prototyp entwickelte Software Repository namens SEnSoR präsentiert, dessen Design und Funktionsweise im Detail erläutert werden.
Kapitel 5 beschäftigt sich mit der Evaluierung des entwickelten Prototyps in Bezug auf dessen Performanz sowie der Präzision der integrierten Suchfunktionalität.
Abschließend werden in Kapitel 6 die Ergebnisse der Arbeit sowie einige weiterführende Punkte besprochen.

KAPITEL 2

Grundlagen

Dieses Kapitel beschäftigt sich mit grundlegenden Konzepten rund um den Themenkomplex Komponenten & Services. Anschließend werden die zugrundeliegenden Technologien bezüglich Web Services sowie Semantic Web Services, welche jeweils eine konkrete Möglichkeit zur Realisierung eines Service darstellen, erläutert.

2.1 Von Komponenten zu Services

Der Bedarf nach Wiederverwendbarkeit von Softwarekomponenten geht einher mit der Evolution von Softwareentwicklung als Ingenieursdisziplin. Bereits 1968 äußerte Douglas McIlroy in seinem Artikel „Mass Produced Software Components“ [59] die Idee, Prinzipien aus der Massenfertigung in die Softwareentwicklung mit einfließen zu lassen, um die Produktivität zu steigern. McIlroy argumentiert, dass es in der industriellen Fertigung üblich wäre, auf vorhandene Standardbauteile wie Schrauben, Formteile und dergleichen zurückzugreifen. Bei der Entwicklung von Software hingegen werde das Rad jedes Mal neu erfunden. Entwickler stellen eher die Frage „Wie bauen wir die Software?“ und nicht „Aus welchen vorhandenen Teilen bauen wir die Software?“.
Mit der Verbesserung der Sprachunterstützung zur Strukturierung von Programmcode (Module, Pakete, etc.) und nicht zuletzt dem Aufschwung der objektorientierten Programmierung bildeten sich neue Softwareentwicklungstechniken wie die komponentenbasierte Entwicklung (*Component-Based Software Engineering*, CBSE) und, aktueller, die serviceorientierte Entwicklung (*Service-Oriented Software Engineering*, SOSE).
Der Begriff einer Software-Komponente ist dabei häufig überladen. Brown und Wallnau charakterisieren das Wesen einer Komponente wie folgt:

> „A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture.“ [9]

Das architektonische Prinzip hinter CBSE ist es, Software als ein Zusammenspiel von Komponenten zu begreifen. Jede Komponente kapselt dabei einen bestimmten Teil der Gesamtfunktionalität der Software, wodurch diese modularisiert wird. Die Abhängigkeit zwischen Komponenten ist im Idealfall gering und die Kommunikation zwischen den Komponenten erfolgt über definierte Schnittstellen.
In der Praxis werden Komponenten durch eine Menge von Objekten realisiert, welche im Zusammenspiel die Funktionalität der Komponente implementieren. Objekte sind dabei als ein Abstraktionsmuster und nicht notwendigerweise im Sinne der objektorientierten Programmierung zu verstehen. Bei einem Objekt handelt es sich lediglich um eine Funktionseinheit oder eine Ressource, die eine bestimmte Teilaufgabe erfüllen kann.
Unter einer serviceorientierten Architektur (SOA) versteht man ebenfalls ein Software-Architektur-Entwurfsmuster für verteilte Anwendungen, bei dessen Grundbausteinen es sich nicht um Komponenten, sondern um Services handelt. Die Grenze zwischen Services und Komponenten ist im Sprachgebrauch oft unscharf und wird zudem manchmal mit einer konkreten Implementierung – wie etwa einem Web Service – gleichgesetzt.
Analog zur wirtschaftlichen Auffassung, bei einem Service im Sinne einer Dienstleistung handle es sich um die Bereitstellung einer Leistung innerhalb einer bestimmten Domäne, definiert Preist ein Service wie folgt:

> „[A service is a] software entity able to provide something of value.“ [70]

Gemäß dieser Definition würde es sich bei Komponenten ebenfalls um Services handeln. Der Unterschied besteht eher im Grad der Abstraktion. Ein Service wird oft als Teil eines (Geschäfts-) Prozesses betrachtet, während es sich bei Komponenten eher um deren softwaretechnische Ausprägung handelt. In der Tat ist es so, dass die Funktionalität eines Service letztendlich über Software-Komponenten realisiert wird:

> „Services are self-describing, open components [...]“ [66].

Erl definiert die Eigenschaften von Services als Bausteine für eine SOA wie folgt [22]:

- **Autonomie:** Services kontrollieren die Logik, welche sie bereitstellen, selbst. Sie können dadurch eigenständig aufgerufen werden.
- **Lose Kopplung:** Services haben keine oder nur ein Mindestmaß an Abhängigkeiten zueinander.
- **Vertrag:** Der Aufruf und die Interaktion von Services untereinander basiert auf einer Einigung über die verwendeten Protokolle und Nachrichtenaustauschformate, welche in der Schnittstellenbeschreibung eines Service, dem Service-Vertrag, definiert sind.
- **Transparenz:** Services verstecken ihre Implementierungsdetails vor dem Aufrufer. Sie sind plattformunabhängig.
- **Wiederverwendbarkeit:** Ein Service implementiert die Funktionalität zur Erfüllung einer spezifischen Aufgabe, wodurch die Wiederverwendbarkeit gefördert wird.

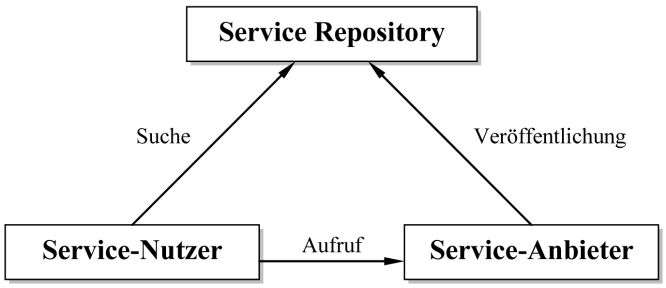

Abbildung 2.1: Das SOA-Dreieck.

- **Komposition:** Neue Services können durch die Komposition vorhandener Services erzeugt werden.
- **Zustandslosigkeit:** Im Idealfall lässt sich ein Service ohne die Speicherung von zusätzlicher Zustandsinformation aufrufen, was den Verwaltungsaufwand gering hält. Im Falle einer zustandsbehafteten Interaktion mit einem Service speichert dieses nur ein Mindestmaß an Zustandsinformation oder delegiert diese Aufgabe ganz an ein darunterliegendes Framework oder die Applikationslogik.
- **Auffindbarkeit:** Es existiert ein definierter Auffindungsmechanismus für Services, welcher Services für Benutzer zugänglich macht.

Abbildung 2.1 zeigt das Grundprinzip einer serviceorientierten Architektur, in welcher drei verschiedene Rollen existieren:

- **Service-Anbieter:** Der Anbieter eines Service möchte selbiges innerhalb der SOA zugänglich machen. Zu diesem Zweck veröffentlicht er dessen Schnittstellenbeschreibung über das Repository.
- **Service-Repository:** Das Repository kann selbst als Service betrachtet werden und stellt einen Verzeichnisdienst bereit, der innerhalb der Architektur zum Auffinden von anderen Services genutzt wird. Typischerweise bietet es Mechanismen zur Veröffentlichung und Verwaltung von existierenden Services über eine API.
- **Service-Nutzer:** Benötigt ein Nutzer ein bestimmtes Service, kontaktiert dieser das Repository über eine entsprechende Suchanfrage. Dieses liefert, sofern vorhanden, die Referenz auf ein passendes Service, welches von einem Anbieter innerhalb der SOA bereitgestellt wird. Die übrige Kommunikation zwischen Anbieter und Nutzer erfolgt ohne Beteiligung des Repository.

Damit das Service Repository Suchanfragen von Clients erfüllen kann, speichert es Informationen über indizierte Services. Dabei unterscheidet man grundsätzlich zwischen den folgenden beiden Kategorien:

1. **Funktionale Eigenschaften:** Die funktionale Beschreibung eines Service beinhaltet die formale Spezifikation dessen Funktionalität. Dazu zählen technische Informationen über aufrufbare Methoden, deren Parameter, Datentypen, Rückgabewerte, Exceptions, Vor- und Nachbedingungen, aber auch verwendete Übertragungsprotokolle, Nachrichtenaustauschformate und dergleichen [81].

   Anhand der funktionalen Eigenschaften eines Service weiß ein Client, was ein Service kann und wie selbiges aufzurufen ist, um dessen Funktionalität nutzen zu können.

2. **Nicht-funktionale Eigenschaften:** Während das Wissen über die funktionalen Eigenschaften eines Service elementar ist, um selbiges aufrufen zu können, handelt es sich bei nicht-funktionalen Eigenschaften um zusätzliche (teils nicht technische) Aspekte, welche bei der Nutzung eines Service ebenfalls eine Rolle spielen und diese eventuell einschränken können.

   Verfügt ein Service beispielsweise über eine geringe Verfügbarkeit, so ist die Chance hoch, dass es im Bedarfsfall nicht erreichbar ist. Kennwerte wie Ausführungszeit, Durchsatz oder Latenz geben darüber hinaus Auskunft über die Performanz eines Service. Sicherheitsrelevante Kriterien wie Authentifizierung, Vertraulichkeit oder Verschlüsselung zählen ebenfalls zu den nicht-funktionalen Anforderungen. Einige Aspekte, wie beispielsweise die Kosten eines Service-Aufrufs, sind dabei eher für die Nutzung *fremder* Services innerhalb der eigenen SOA relevant.

   Finden sich mehrere Service-Kandidaten, die eine gewünschte Funktionalität anbieten, so entscheiden oft nicht-funktionale Eigenschaften innerhalb eines Selektionsprozesses darüber, welches nun tatsächlich ausgewählt wird. Beispielsweise könnte ein Service ganz ausscheiden, weil es ein gefordertes Maß an Sicherheit nicht erfüllen kann, oder man selektiert ein Service nach einer bestimmten Metrik, etwa jenes mit den geringsten Kosten pro Aufruf.

2.2 Web Services

Bei Web Services handelt es sich um *eine* konkrete Möglichkeit, eine serviceorientierte Architektur zu implementieren. Der wesentliche Unterschied zum reinen Service besteht darin, dass die Kommunikation mit einem Web Service über Web-Standards erfolgt. Ein Web Service läuft, ähnlich einer Webseite, auf einem bestimmten Server und wird über einen *Uniform Resource Identifier* (URI) identifiziert. Es ist entweder global über das Internet oder (Organisations-)intern innerhalb eines Intranets verfügbar.
Im Gegensatz zu einer Webseite ist die Funktionalität eines Web Service jedoch nicht für den direkten Aufruf durch einen Menschen, sondern für Client-Programme beziehungsweise andere Web Services, bestimmt.
Das UDDI Consortium definiert Web Services demnach wie folgt:

> „Web services are self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces. Web services communicate directly with other Web services via standards-based technologies.“ [83]

Obige Definition erfasst das grundlegende Wesen eines Web Service, ohne sich im Detail auf konkrete Standards zu beschränken. Die *W3C Web Services Architecture Working Group* geht einen Schritt weiter und definiert Web Services auf einer technisch spezifischeren Ebene:

> „A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.“ [32]

Diese Definition beinhaltet bereits konkrete Web Service-Technologien, auf welche in den folgenden Abschnitten eingegangen wird. Es soll dabei jedoch nicht unerwähnt bleiben, dass auch andere Arten von Web Services, wie beispielsweise RESTful Web Services, existieren.
Aus Sicht des Service Repository, auf welchem der Fokus dieser Arbeit liegt, ist es jedoch unerheblich, welchem architektonischen Prinzip ein Web Service folgt. Um ein Web Service für den Anwender auffindbar zu machen, ist es vielmehr relevant, dass sich dessen Funktionalität, wie in obiger Definition erwähnt, durch ein maschinenlesbares Format beschreiben lässt. Auch im Falle von RESTful Web Services existiert mit der *Web Application Description Language* (WADL) eine auf XML basierende Beschreibungssprache.
Die Verwendung von existierenden Web-Standards wie XML und HTTP bietet zudem gleichermaßen Vorteile wie auch Nachteile. Zum einen liegen die Hauptvorteile von Web Services in deren Plattformunabhängigkeit und Interoperabilität. Web Services ermöglichen beispielsweise die Zusammenarbeit von Applikationen über das Internet, die in unterschiedlichen Technologien entwickelt sind und möglicherweise nicht dafür entworfen wurden.
Andererseits handelt es sich bei XML um ein recht ausschweifendes Datenformat, was sich bei der Datenübertragung im negativen Sinne bemerkbar macht. Zudem eignen sich Web Services tendenziell schlecht für langanhaltende, zustandsbehaftete Transaktionen, da bereits das am weiten verbreitetste Übertragungsprotokoll HTTP selbst ein zustandsloses Protokoll ist.

2.2.1 Architektur

Die klassische „High-Level“-Architektur von Web Services ist in mehrere Schichten unterteilt, wobei jede Ebene auf die darunterliegenden Ebenen aufbaut (Abbildung 2.2).

- **Transport:** Die Transportschicht übernimmt die Übertragung von Nachrichten zwischen einem Web Service und dessen Aufrufer über das Netzwerk. Das gebräuchlichste Protokoll für diese Aufgabe ist das bekannte *Hypertext Transfer Protocol* (HTTP), wobei jedoch auch Alternativen wie das *Simple Mail Transfer Protocol* (SMTP) zum Einsatz gelangen können.

- **Nachrichten:** Die Interaktion mit einem Web Service lässt sich als Austausch von Nachrichten auffassen. Zu diesem Zweck definiert die Nachrichtenschicht das Format, in welchem selbige ausgetauscht werden. Ein weit verbreitetes Protokoll dazu ist SOAP (siehe 2.2.3.2), welches Nachrichten im XML-Format repräsentiert.

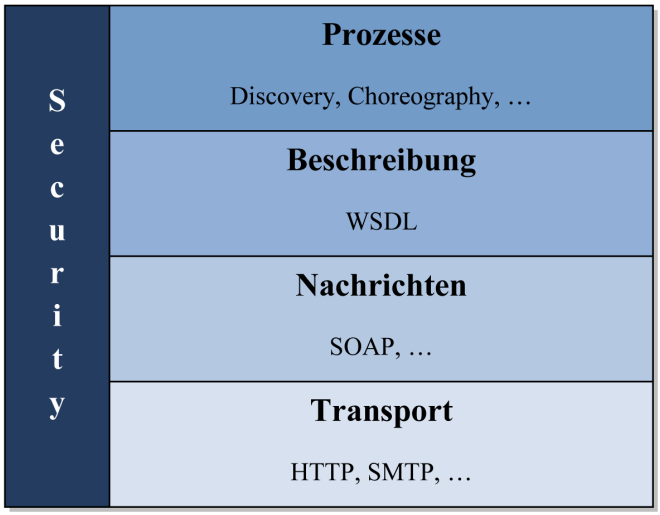

Abbildung 2.2: Web Service Architektur (in Anlehnung an [6]).

- **Beschreibung:** Die Schnittstellenbeschreibung eines Service ermöglicht dessen Aufruf. Typischerweise enthalten sind Informationen wie der Name, die Kategorie und die angebotenen Methoden eines Service. Zudem enthält die Schnittstellenbeschreibung auch die physische Adresse, unter welcher eine Service-Methode über das Netzwerk aufgerufen werden kann.
- **Prozesse:** Aufbauend auf Service-Beschreibungen lassen sich Mechanismen für das (automatisierte) Auffinden von Services oder deren Komposition zur Erfüllung komplexerer Aufgabenstellungen realisieren.

Im Falle von **Security** handelt es sich um einen sogenannten „Cross-Cutting Concern“, welcher sich durch sämtliche Ebenen zieht und sicherheitsrelevante Aspekte wie die Signatur oder Verschlüsselung von Nachrichten behandelt.

2.2.2 Funktionsweise

Abbildung 2.3 zeigt den Vorgang der Auffindung und des Aufrufs eines Web Service, welcher aus mehreren Schritten besteht:

1. Der Client benötigt zunächst eine gewisse Funktionalität, um sein eigenes Programm ausführen zu können. Zu diesem Zweck sendet dieser eine Suchanfrage an ein Service Repository. Das Repository durchsucht seine indizierten Services und antwortet idealerweise mit Referenzen auf existierende Web Services, welche jeweils die Anforderungen des Clients abdecken. Dieser Vorgang wird im Englischen als *Service Discovery* bezeichnet.
2. Der Client kennt nun die Adresse eines passenden Web Service, welches von einem Server bereitgestellt wird und unter einer bestimmten URL erreichbar ist. Er weiß allerdings

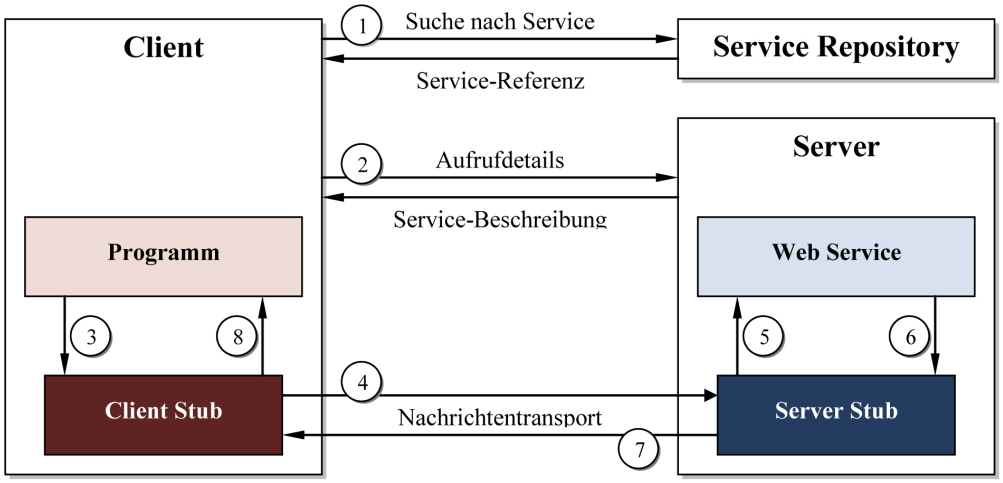

Abbildung 2.3: Die Suche nach einem Web Service und dessen Aufruf.

noch nicht, wie dieses aufzurufen ist und schickt aus diesem Grund eine Anfrage nach den Aufrufdetails direkt an das Web Service. Dieses antwortet mit seiner Schnittstellenbeschreibung, typischerweise einem WSDL-Dokument (siehe 2.2.3).

3. Das Programm des Clients kommuniziert nicht direkt mit dem Web Service. Vielmehr erzeugt der Client anhand der Schnittstellenbeschreibung einen sogenannten *Stub* (dt. Platzhalter), an welchen die technischen Aspekte des Nachrichtenaustauschs mit dem Web Service delegiert werden. Somit gestaltet sich der Aufruf für die Programmiererin transparent und unterscheidet sich auf Programmierebene grundsätzlich nicht von herkömmlichen Methodenaufrufen.

   Kommt es an einer bestimmten Stelle im Client-Programm zu einem Aufruf des Web Service, konvertiert der Client-Stub selbigen inklusive Parameter in eine Nachricht an das Web Service, welche über das Netzwerk transportiert wird. Da Client und Web Service nicht in derselben Technologie implementiert sein müssen, wird für den Nachrichtenaustausch ein plattformunabhängiges Format (typischerweise SOAP, siehe 2.2.3.2) gemäß der Schnittstellenbeschreibung verwendet. Der Vorgang der Konvertierung in dieses Format wird als *Marshalling* bezeichnet.

4. Die vom Client-Stub generierte Nachricht wird über das gewählte Transportprotokoll (etwa HTTP) an das Web Service übermittelt.

5. Auf Seiten des Servers übernimmt der Server-Stub das *Unmarshalling*, also die Rück-Konvertierung der erhaltenen Nachricht in ein für das Web Service verständliches Format.

6. Die eigentliche Implementierung des Web Service führt die gewünschte Methode mit den übermittelten Parametern aus und generiert einen Rückgabewert, etwa ein Objekt, welches vom Server-Stub analog zu Punkt 3 in eine Antwort-Nachricht konvertiert wird.

7. Die Antwort-Nachricht des Web Service wird zurück an den Client übertragen.

8. Der Client-Stub konvertiert die erhaltene Nachricht, sodass das Client-Programm diese versteht. Im Falle eines synchronen Aufrufs hat das Client-Programm bis zu diesem Zeitpunkt blockiert. Im Falle eines asynchronen Aufrufs erhält das Programm eine Notifikation über das Eintreffen der Antwort-Nachricht, worauf dieses je nach zugrundeliegender Technologie beispielsweise über eine *callback*-Methode reagiert.

2.2.3 Beschreibung von Web Services

2.2.3.1 Web Service Description Language (WSDL)

Die *Web Service Description Language* ist eine Beschreibungssprache für Web Services, die auf der *Extensible Markup Language* (XML) basiert. Die aktuelle Version der WSDL-Spezifikation ist WSDL 2.0 aus dem Jahr 2007 [14]. In der Praxis sind jedoch noch viele Web Services in der Vorgängerversion WSDL 1.1 aus 2001 [15] notiert, sodass aktuell beide Versionen nebeneinander existieren.

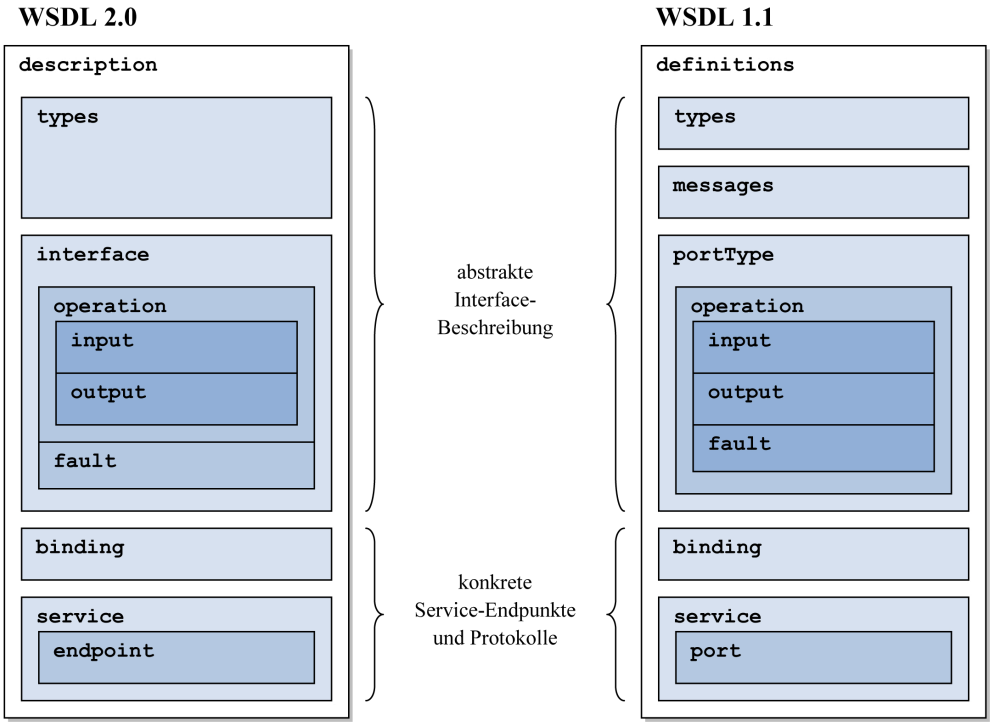

Abbildung 2.4: Der schematische Aufbau eines WSDL-Dokuments.

Abbildung 2.4 zeigt den Aufbau eines WSDL-Dokuments, welcher sich in einen abstrakten sowie einen konkreten Abschnitt gliedert und aus den folgenden Hauptelementen besteht:

- **Types/Messages:** Typen können über *XML Schema* definiert werden und beschreiben auf abstrakter Ebene Daten, die zwischen Web Service und Aufrufer in Form von Nachrichten ausgetauscht werden. In WSDL 1.1 existieren diese Nachrichten (*messages*) noch explizit in der Service-Beschreibung. In Version 2.0 hingegen referenzieren *input*-, *output*- und *fault*-Elemente jeweils direkt einen XML-Schema-Datentyp, der die Struktur der Input-Nachricht beschreibt.

  XML-Schema bietet sogenannte „simple“ und „komplexe“ Typen. Erstere repräsentieren primitive Datentypen wie Strings oder Zahlenbereiche. Über komplexe Typen lassen sich darüber hinaus zusammengesetzte Typen definieren, etwa für den Rückgabewert einer Methode, wenn dieser aus mehreren Werten besteht.

- **Interface:** Das Interface-Element (*porttype* in WSDL 1.1) bildet die Schnittstellenbeschreibung eines Service auf abstrakter Ebene ab und dient zur Gruppierung mehrerer *operation*-Elemente. Diese wiederum beinhalten die Signaturen aller aufrufbaren Methoden des Service.

  Das Ausführen einer Methode wird als Austausch von eingehenden bzw. ausgehenden Nachrichten aufgefasst und folgt einem *message exchange pattern* (MEP), welches für jede Methode definiert wird. Ein gebräuchliches MEP ist beispielsweise „In-Out“, welches einen klassischen Methodenaufruf repräsentiert: eine *input*-Nachricht an das Web Service wird entweder mit einer *output*- oder im Fehlerfall mit einer *fault*-Nachricht quittiert. Neben weiteren vorgefertigten MEPs wie „In-Only“, „Out-Only“ „In-Optional-Out“ besteht auch die Möglichkeit zur Erstellung eigener MEPs.

- **Binding:** Damit das bislang abstrakt spezifizierte Service aufgerufen werden kann, muss der Aufrufer wissen, in welchem konkreten Format Nachrichten über das Netzwerk versendet werden und über welches Protokoll. Die Serialisierung von Nachrichten wird über ein sogenanntes *Binding* innerhalb des gleichnamigen Elements spezifiziert, welches der Struktur des *interface*-Elements folgt. Ein Service kann mehrere Bindings anbieten.

  Die WSDL-Spezifikation bietet vorgefertigte Bindings für reines HTTP und SOAP (über HTTP). Diese spezifizieren jeweils, wie eine abstrakte XML-Nachricht in eine HTTP- bzw. SOAP-Nachricht umgewandelt wird. Im Falle von WSDL 2.0 wird im Binding außerdem definiert, wie das gewählte MEP einer Methode im Protokoll umgesetzt wird.

  Im Normalfall ist es nicht notwendig, dass ein Binding spezifische Informationen über ein physisches Service beinhaltet. Somit ist es möglich, dass sich mehrere Service-Anbieter nicht nur ein gemeinsames Interface, sondern auch das Binding teilen, sodass nur mehr ein konkreter Endpunkt zur Verfügung gestellt werden muss.

- **Service:** Als letzte Information fehlt noch die Zuordnung eines Interfaces zu einem einzigen physischen Service, welches dieses implementiert. Diese Zuordnung geschieht im *service*-Element.

  Ein Service kann mehrere Endpunkte (WSDL 1.1: *ports*) in Form unterschiedlicher Adressen anbieten, die jeweils einem Binding zugeordnet werden. Somit weiß der Aufrufer einer

Methode schlussendlich, an welche Adresse welche Nachricht in welchem Format über welches Protokoll geschickt werden kann.

Zum besseren Verständnis zeigt Abbildung 2.5 ein Beispieldokument in WSDL 2.0 für das fiktive *BookService*, das genutzt werden kann, um Informationen über Bücher anhand der ISBN zu suchen. Das Service bietet eine einzige Methode *getBookByISBN* an, welche dem MEP „In-Out“ folgt. Der Input für das Service ist über den komplexen Datentyp *getBookByISBN-RequestType* definiert und beinhaltet als einzigen Parameter die ISBN eines Buches als String. Die Methode gibt entweder das gewünschte Buch als Output *response* zurück (im Beispiel nicht dargestellt), oder generiert im Falle einer ungültigen ISBN eine *InvalidISBNException*. Das Service verfügt über ein einziges Binding über SOAP und ist unter der (fiktiven) Adresse *http://example.ac.at/BookService* erreichbar.

2.2.3.2 Simple Object Access Protocol (SOAP)

SOAP ist ein Protokoll zum Austausch von strukturierten Nachrichten in verteilten Anwendungen. Es ist ein zustandsloses Protokoll und nutzt XML als Nachrichtenaustausch-Format.
Ursprünglich wurde SOAP bereits 1998 bei Microsoft als Protokoll zum Zugriff auf Objekte entwickelt. Mittlerweile liegt SOAP in Version 1.2 von 2007 [60] vor und hat – wie auch WSDL 2.0 – den Status einer *W3C Recommendation*.
Das Akronym SOAP für *Simple Object Access Protocol* wurde mit Version 1.2 fallen gelassen – vermutlich, da die Begriffe „simpel“ und „Objektzugriff“ in Bezug auf die aktuelle Spezifikation nicht mehr zutreffend sind.
SOAP definiert die folgenden Aspekte der Nachrichtenübermittlung:

- **Message Structure:** Eine SOAP-Nachricht wird im Wurzelelement *envelope* (dt. Umschlag, Hülle) gekapselt, was zugleich den Anfang und das Ende einer Nachricht definiert. *Envelope* verfügt über zwei Kindelemente, den optionalen *header* und den verpflichtenden *body* der Nachricht. Im *header* kann einer Nachricht applikationsspezifische Information, wie beispielsweise Authentifizierungsdaten mitgegeben werden. Der *body* beinhaltet hingegen den eigentlichen Inhalt der Nachricht, oder – im Falle eines Fehlers – ein *fault*-Element.

  Zur Verdeutlichung zeigen Abbildung 2.6 und 2.7 eine Beispiel-Anfrage an das *BookService* aus Kapitel 2.2.3.1, sowie die dazugehörige Antwortnachricht des Web Service. Die Anfrage enthält eine gültige ISBN, woraufhin das Web Service den entsprechenden Buchtitel sowie dessen Autor liefert.

- **Processing Model:** Das SOAP-*Processing Model* definiert Regeln für die Verarbeitung von SOAP-Nachrichten. In der Verarbeitungskette gibt es einen initialen Absender der Nachricht sowie einen ultimativen Empfänger. Dazwischen können weitere Knoten existieren. Im SOAP-Header kann angegeben werden, welche Knoten welche Teile der Nachricht verarbeiten sollen beziehungsweise dürfen.

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsdl"
             xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
             xmlns:tns="http://example.ac.at/"
             targetNamespace=" http://example.ac.at/">

  <!-- Abstract types -->
  <types>
    <xs:schema xmlns=" http://example.ac.at/"
               xmlns:xs="http://www.w3.org/2001/XMLSchema"
               targetNamespace=" http://example.ac.at/">
      <xs:element name="InvalidISBNException" type="tns:InvalidISBNException"/>
      <xs:element name="response" type="tns:getBookByISBNResponseType"/>
      <xs:element name="request"  type="tns:getBookByISBNRequestType"/>
      <xs:complexType name="getBookByISBNRequestType">
        <xs:sequence>
          <xs:element name="isbn" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      ...
    </xs:schema>
  </types>

  <!-- Abstract Service Interface -->
  <interface name="BookService">
    <fault name="InvalidISBNException" element="tns:InvalidISBNException"/>
    <operation name="getBookByISBN" pattern="http://www.w3.org/ns/wsdl/in-out">
      <input  messageLabel="In"  element="tns:request"/>
      <output messageLabel="Out" element="tns:response"/>
      <outfault ref="tns:InvalidISBNException"/>
    </operation>
  </interface>

  <!-- SOAP Binding -->
  <binding name="SoapBinding"
           interface="tns:BookService"
           type="http://www.w3.org/ns/wsdl/soap"
         wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
         wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
    <fault ref="tns:InvalidISBNException"/>
    <operation ref="tns:getBookByISBN"/>
  </binding>
  <!-- Service Endpoints -->
  <service name="BookService" interface="tns:BookService">
    <endpoint name="SoapEndpoint"
              binding="tns:SoapBinding"
              address="http://example.ac.at/BookService"/>
  </service>
</description>
```

Abbildung 2.5: Ein Service-Dokument in WSDL 2.0.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  <soap:Body>
    <b:request xmlns:b="http://example.ac.at/">
      <b:isbn>978-3-15-000001-4</b:isbn>
    </b:request>
  </soap:Body>
</soap:Envelope>
```

Abbildung 2.6: SOAP-Anfrage an das *BookService*.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  <soap:Body>
    <b:response xmlns:b="http://example.ac.at/">
      <b:book>
        <b:author>Goethe</b:author>
        <b:title>Faust I</b:title>
      </b:book>
    </b:response>
  </soap:Body>
</soap:Envelope>
```

Abbildung 2.7: SOAP-Antwortnachricht des *BookService*.

- **Protocol Bindings:** Im Kontext von Web Services nutzt SOAP hauptsächlich HTTP(S) als zugrundeliegendes Transport-Protokoll, erlaubt jedoch auch Alternativen wie beispielsweise das *Simple Mail Transfer Protocol* (SMTP) oder *Message Queues*. Das *Protocol Binding* spezifiziert detaillierte Aspekte des Nachrichtentransports, wie die Adressierung („Wie sieht eine Endpunkt-Adresse aus?“), Serialisierung („Wie sieht die Byte-Repräsentation von XML-Nachrichten aus, um sie über das Netzwerk zu übertragen?“) sowie die Verbindung („Wie werden die Bytes an einen Endpunkt gesendet?“).

2.2.3.3 Semantic Annotation for WSDL (SAWSDL)

Die Web Service Description Language beschreibt ein Web Service auf einer rein syntaktischen Ebene. Der Aufrufer weiß beispielsweise nach dem Parsen eines WSDL-Dokuments, welche Methoden ein Web Service bereitstellt und aus welchen Attributen sich eine Antwortnachricht zusammensetzt. Der WSDL-Standard selbst bietet jedoch keine Möglichkeit, die Ausführungssemantik eines Web Service näher zu spezifizieren.
Betrachtet man zum Beispiel das *BookService* aus Abbildung 2.5, so ist es für einen Menschen verständlich, dass es sich bei der *isbn* eines Buches um die internationale Standardbuchnummer handelt und dadurch ein gewisses Format haben muss. Für einen Parser, dem dieses Zusatzwissen fehlt, handelt es sich dabei jedoch einfach um einen beliebigen String, dessen Bezeichner *isbn* lediglich zur Unterscheidung von anderen Strings dient und keinerlei nähere Bedeutung hat. Mit steigender Anzahl an verfügbaren Web Services wird das automatisierte Auffinden und letztendlich auch die Ausführung eines passenden Service immer schwieriger. Aus diesem Grund wurde SAWSDL [47] im Jahr 2007 als Erweiterung zu WSDL mit dem Ziel eingeführt, die

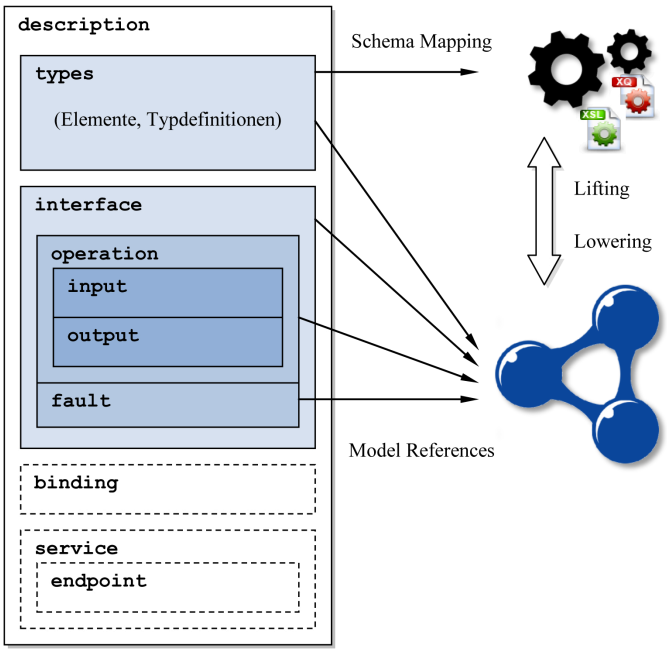

Abbildung 2.8: Die SAWSDL Erweiterung eines WSDL 2.0-Dokuments (in Anlehnung an [47]).

bestehende WSDL-Spezifikation um einen Mechanismus zu erweitern, der es ermöglicht, Komponenten eines WSDL-Dokuments mit Informationen über deren semantische Bedeutung zu versehen. SAWDSL hat ebenfalls den Status einer *W3C Recommendation* [23].
Die beiden wichtigsten Konzepte von SAWSDL sind, wie in Abbildung 2.8 dargestellt, *model references* sowie *schema mapping*. Über eine *model reference* lässt sich ein Verweis von einem Element innerhalb des WSDL-Dokuments auf ein Konzept in einem semantischen Modell herstellen. Wie genau ein semantisches Konzept ausgedrückt wird, wird von SAWSDL nicht eingeschränkt. Eine gebräuchliche Methode ist die Verwendung von Ontologien, welche beispielsweise über die *Web Ontology Language* (OWL) repräsentiert werden können. In diesem Fall werden Konzepte als OWL-Klassen modelliert (siehe 2.3).
Über das *schema mapping* wird zudem definiert, wie eine WSDL-Nachricht in das semantische Modell und wieder zurück transformiert wird. Zu diesem Zweck führt SAWSDL insgesamt drei zusätzliche XML-Attribute sowie ein zusätzliches Element ein:

- **modelReference:** Das Attribut *modelReference* wird dafür verwendet, einem WSDL-Element eine Referenz auf ein oder mehrere semantische Konzepte in Form von URIs bestimmter OWL-Klassen zuzuweisen. Das Attribut kann in WSDL- und XML-Schema-Elementen innerhalb des *types*- und *interface*-Elements verwendet werden (siehe Abbildung 2.8).

  Über das `modelReference`-Attribut lässt sich somit beispielsweise das XML-Schema-Element *isbn* mit einer entsprechenden OWL-Klasse *ISBN* aus der Ontologie *Books* wie

folgt verlinken:

```
<xs:element name="isbn" type="xs:string"
    sawsdl:modelReference="http://example.ac.at/Books.owl#ISBN"/>
```

- **liftingSchemaMapping / loweringSchemaMapping:** Das sogenannte *Lifting* bezeichnet die Transformation einer XML-Nachricht eines Web Service in ein konkretes semantisches Modell, wie etwa der RDF/XML-Repräsentation eines Konzeptes in Form einer OWL-Klasse. *Lowering* steht für die Transformation in die andere Richtung, also von einem semantischen Konzept in eine für das Web Service verständliche XML-Nachricht.

  Da diese beiden unterschiedlichen Ebenen nicht notwendigerweise die selbe Struktur haben, ist es über die beiden Attribute *liftingSchemaMapping* und *loweringSchemaMapping* möglich, Transformationsregeln für die Konvertierung zwischen den Ebenen zu spezifizieren. Somit kann beispielsweise ein Aufrufer des *BookService*, der das Konzept *ISBN* aus der *Books*-Ontologie kennt, eine ISBN über das Lowering in eine konkrete XML-Nachricht transformieren. Diese Nachricht wird an das Web Service geschickt, welches als Rückgabewert wiederum eine XML-Nachricht generiert, die beispielsweise ein Buch beschreibt. Ist dieser Rückgabetyp ebenfalls mit einer *model reference* versehen, wendet der Aufrufer die Transformation für das Lifting an und kann dadurch wieder auf semantischer Ebene weiterarbeiten.

  Für die Transformation von XML-Dokumenten in andere XML-basierte Dokumente bieten sich die beiden Sprachen *Extensible Stylesheet Language Transformations* (XSLT) und *XQuery* an. Der Inhalt eines *liftingSchemaMapping*- oder *loweringSchemaMapping*-Attributs besteht somit einfach in einem Link zu einem entsprechenden XSLT- oder auch XQuery-Programm.

- **attrExtensions:** SAWSDL wurde ursprünglich für WSDL 2.0 entwickelt, unterstützt jedoch die noch weit verbreitete Version 1.1. Die oben erwähnten Attribute sind in beiden Versionen gleichermaßen auf den entsprechenden Elementen anwendbar, mit einer Ausnahme: Das Schema für WSDL 1.1 erlaubt ausschließlich Elemente als Erweiterung für das Element *operation*, was das Hinzufügen des *modelReference*-Attributs verhindert.

  Für diesen Sonderfall wurde das Element *attrExtensions* eingeführt, das als Kindelement von *operation* verwendet werden kann und über diesen Umweg eine *model reference* auch für Methoden in WSDL 1.1 möglich macht.

Wie man erkennen kann, geschieht die Anreicherung eines bestehenden WSDL-Service mit Information über dessen Semantik im Wesentlichen durch das Hinzufügen von *modelReference*-Attributen.
Was zunächst simpel erscheint, offenbart jedoch Probleme im Detail: Die genaue Bedeutung des *modelReference*-Attributs auf den verschiedenen anwendbaren Elementen *interface*, *operation*, *fault*, *element*, *complexType*, *simpleType* und *attribute* hat gemäß der SAWSDL-Spezifikation eher den Charakter einer Empfehlung denn einer präzisen Definition [57].

Beispielsweise besagt die SAWSDL-Spezifikation, dass das Annotieren eines *operation*-Elements (also einer Methode) zur Kategorisierung, aber auch zur Beschreibung von Vor- und Nachbedingungen verwendet werden kann. Da es dafür jedoch nur ein einziges Attribut gibt (*modelReference*), das eine Liste von URIs enthält, ist es bei mehreren Werten nicht mehr nachvollziehbar, welche URI nun auf welches dieser Konzepte verweist. Ein Aufrufer erkennt zwar, dass einer Methode mehrere verschiedene Konzepte zugewiesen wurden, kann diese jedoch nicht auseinanderhalten, was die automatisierte Auffindung eines Service – beispielsweise nach dessen Vorbedingungen – verhindern kann.
Yet Another Semantic Annotation for WSDL (YASA4WSDL), eine Erweiterung zu SAWSDL, führt zur Behebung dieses Problems das zusätzliche Attribut *serviceConcept* ein [13]. Die grundlegende Idee ist es, die technischen Aspekte eines Web Service von den domänenspezifischen zu trennen. Zu diesem Zweck arbeitet YASA4WSDL mit einer *technical ontology* und einer separaten *domain ontology*.
Im folgenden Beispiel sind in der technischen Ontologie *ServiceOntology* unter anderem die beiden Konzepte einer Vorbedingung und eines Rückgabewerts definiert. Diese lassen sich für weitere Services wiederverwenden.
In der domänenspezifischen Ontologie *Books* hingegen sind zwei konkrete Ausprägungen dieser Konzepte definiert. Durch die Reihenfolge ist nun erkenntlich, dass es sich bei *ValidISBN* um eine Vorbedingung und bei *Book* um den Rückgabewert (bzw. Rückgabetyp) des Service handelt. Leider lässt die Spezifikation offen, wie mit etwaigen Argumenten von Vor- oder Nachbedingungen zu verfahren ist.

```
<operation name="getBookByISBN"
           modelReference="Books.owl#ValidISBN
                           Books.owl#Book"
           serviceConcept="ServiceOntology.owl#Precondition
                           ServiceOntology.owl#Result">
  ...
</operation>
```

2.3 Semantic Web Services

Die ursprüngliche Vision des Semantic Web geht auf den gleichnamigen Artikel von Tim Berners-Lee zurück [4]. Darin beschreibt er das Semantic Web als eine Erweiterung des aktuell bestehenden World Wide Web. Die grundlegende Idee ist es, im Internet veröffentlichte Informationen mit einer wohldefinierten Bedeutung zu versehen, um diese nicht nur für Menschen, sondern insbesondere für die maschinelle Verarbeitung zugänglich zu machen.
Im Gegensatz zu beispielsweise einem Web-Browser, der die auf Webseiten verfügbare Information einfach anzeigt ohne deren genauen Inhalt zu verstehen, soll ein sogenannter semantischer Agent dazu in der Lage sein, Informationen selbständig aufzufinden, zu extrahieren und zu interpretieren. Anders als im Falle eines herkömmlichen Web Service bedeutet das für ein Semantic Web Service, dass dessen Funktionalität nicht nur auf syntaktischer, sondern auch auf semantischer Ebene definiert wird. Ein Aufrufer mit einem bestimmten Ziel kann somit über

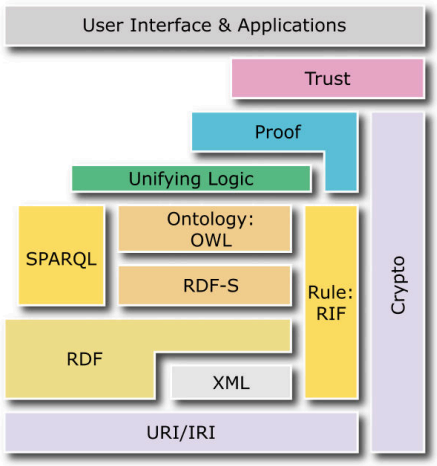

Abbildung 2.9: Semantic Web Architektur (Kopie aus [8]).

Reasoning-Mechanismen feststellen, ob ein bestimmtes Service die passende Funktionalität zur Erfüllung seiner Aufgabe bereitstellt.
Weiterführende Anwendungsgebiete, wie die Komposition einzelner Services zur Erfüllung einer komplexeren Aufgabenstellung oder die automatische Verifikation einer solchen Komposition werden ebenso durch die formale, semantische Spezifikation von Services ermöglicht.
Das Semantic Web stützt sich auf eine Reihe von Technologien, welche unter dem Begriff des *Semantic Web Layer Cake* zusammengefasst werden und dessen Architektur beschreiben (Abbildung 2.9).
Über *Uniform Resource Identifiers* (URIs) lassen sich Ressourcen im Semantic Web eindeutig identifizieren und sind durch die Verwendung von XML zur Repräsentation von Datenstrukturen sowohl von Menschen als auch Maschinen lesbar. Darauf aufbauend existiert mit dem *Resource Description Framework* (RDF) [18] eine simple Beschreibungssprache für Web-Ressourcen. Ein Statement in RDF lässt sich immer als Tripel bestehend aus Subjekt, Prädikat und Objekt lesen, wodurch sich Verknüpfungen zwischen vorhandenen Ressourcen, wie etwa *Wien* (Subjekt) *istHauptstadtVon* (Prädikat) *Österreich* (Objekt), erzeugen lassen. Über die *Simple Protocol and RDF Query Language* (SPARQL) lassen sich RDF-Daten zudem auch abfragen.
Das wichtigste Konzept zur Spezifikation der Bedeutung einer Ressource innerhalb des Semantic Web ist die Verwendung von Ontologien. Gruber definiert eine Ontologie wie folgt:

> „An *ontology* is an explicit specification of a conceptualization.“ [30]

Eine Ontologie ist somit eine konkrete Möglichkeit, Konzepte aus einer bestimmten Domäne sowie deren Beziehungen untereinander zu modellieren. In der Semantic Web Architektur sind dafür *RDF Schema* (RDF-S) sowie die etwas ausdrucksstärkere *Web Ontology Language* (OWL) [29] vorgesehen, welche den de-facto Standard zur Repräsentation von Ontologien im Internet bildet. Dabei handelt es sich um eine Familie von mehreren Sprachen zur Wissensrepräsentation (OWL Lite, OWL DL sowie OWL Full), die sich jeweils in ihrer Ausdrucksstärke, aber

auch dem Grad ihrer Entscheidbarkeit unterscheiden. Im Folgenden werden kurz die wichtigsten Sprachmittel von OWL vorgestellt:

- **Class:** OWL-Klassen bilden Konzepte aus einer bestimmten Domäne ab. Beispielsweise könnte es in einer Bücher-Domäne die Klassen „Buch“, „Autor“, „ISBN“ etc. geben. Klassen können untereinander in Beziehung stehen. In folgendem Beispiel etwa stellt die Klasse *Person* eine Oberklasse von *Author* dar, womit ausgedrückt wird, dass das Konzept eines Autors vom (allgemeineren) Konzept einer Person *subsumiert* wird.

```
<owl:Class rdf:about="#Author">
  <rdfs:subClassOf rdf:resource="#Person"/>
</owl:Class>
```

- **Individual:** Bei *Individuals* handelt es sich um konkrete Ausprägungen von Klassen – beispielsweise könnte es sich bei *George Orwell* um ein bestimmtes Individual der Klasse *Author* handeln. Zu einer Klasse können beliebig viele Individuals existieren, jedoch muss jedes existierende Individual mindestens einer Klasse angehören.

- **Property:** *Properties*, also Eigenschaften oder auch Relationen, werden benutzt um Beziehungen zwischen Klassen bzw. deren Individuals abzubilden. Die Relation *istAutorVon* könnte zum Beispiel die beiden Individuals *George Orwell* der Klasse *Author* und *Farm der Tiere* der Klasse *Book* miteinander verknüpfen.

  Relationen wie *istAutorVon*, welche Beziehungen zwischen je zwei Klassen ausdrücken, bezeichnet man als sogenannte *ObjectProperties*. Im Gegenzug dazu können Klassen aber auch mit Literalen wie Zahlen oder Strings verknüpft werden, um Attribute wie beispielsweise Name oder Alter einer Person abzubilden. In diesem Fall spricht man von *DataProperties*.

  In beiden Fällen können Relationen sowohl in ihrer Definitionsmenge als auch ihrer Wertemenge eingeschränkt werden. So lässt sich sicherstellen, dass nur Individuals mit bestimmten Klassenzugehörigkeiten miteinander verknüpft werden dürfen.

Da das Semantic Web ein verhältnismäßig junges Forschungsgebiet ist und einer natürlichen Evolution unterliegt, gibt es immer wieder Erweiterungen oder Ergänzungen zum bestehenden Technologie-Stack [76]. Semantic Web Services sind in der Architektur nicht explizit aufgeführt, verwenden jedoch dieselben zugrundeliegenden Technologien [38].

2.3.1 Ontology Web Language for Services (OWL-S)

Die *Ontology Web Language for Services* (OWL-S) stellt eine Möglichkeit zur Spezifikation von Semantic Web Services dar. OWL-S wird von der *OWL-S Coalition* entwickelt und liegt aktuell in Version 1.2 vor.[1] Im Gegensatz zur WSDL handelt es sich bei OWL-S nicht um eine Sprache,

[1] OWL-S 1.2: http://www.ai.sri.com/daml/services/owl-s/1.2

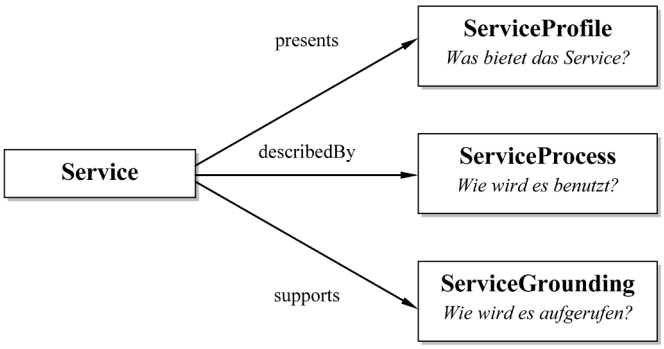

Abbildung 2.10: Die OWL-S Service-Ontologie.

sondern um eine Ontologie, die Konzepte bereitstellt, mit welchen die Funktionalität von Web Services modelliert werden kann [56]. Ein spezifisches Semantic Web Service kann somit als konkrete Ausprägung der OWL-S Ontologie aufgefasst werden.
Konzeptionell wird bezüglich der Funktionalität eines Service zwischen den beiden folgenden Teilaspekten unterschieden:

1. **Information transformation:** Durch die Exekution eines Service wird aus einem übergebenen *Input* ein bestimmter *Output* generiert, welcher das Resultat des Aufrufs darstellt.

2. **State change:** Darüber hinaus kann ein Service-Aufruf eine reale Zustandsänderung wie etwa den Versand einer Ware bewirken (*Effect*) und nur unter bestimmten Bedingungen, etwa dem Vorhandensein einer gültigen Kreditkarte, möglich sein (*Precondition*).

Im Wesentlichen besteht OWL-S aus drei miteinander in Verbindung stehenden Sub-Ontologien, welche unter der Service-Ontologie zusammengefasst werden (Abbildung 2.10). Jede dieser drei Ontologien spezifiziert einen unterschiedlichen Aspekt eines Service.

- **Service Profile:** Im *Profile* eines Service ist spezifiziert, welche Funktionalität es bereitstellt. Neben dem Namen, einer textuellen Beschreibung und der Möglichkeit zur Kategorisierung des Service gibt OWL-S dazu hauptsächlich die vier Relationen *hasInput*, *hasOutput*, *hasPrecondition*, sowie *hasResult* vor, welche jeweils beliebig oft mit einem Service verknüpft werden können. Die Wertemenge für *hasInput* und *hasOutput* ist dabei auf Unterklassen der ebenfalls vorgegebenen *Parameter*-Klasse beschränkt, sodass dafür entweder neue selbstdefinierte Klassen erstellt werden müssen oder auf bereits vorhandene Ontologien zurückgegriffen werden kann.

  Anders als ein WSDL-basiertes Web Service, welches nach außen hin mehrere individuell aufrufbare Methoden bereitstellt, ist in OWL-S bereits das Service selbst die einzige Funktionseinheit. Konzeptionell wird ein Service deshalb als die Sammlung seiner *Inputs*, *Outputs*, *Preconditions* & *Effects* (IOPE) verstanden.

  Erwähnenswert ist, dass in der gesamten Service-Ontologie keine explizite Relation für Exceptions vorgesehen ist. Stattdessen gibt es in OWL-S den sogenannten *conditional*

effect, welcher dazu benutzt werden kann, eine Exception zu modellieren [57]. Ein *conditional effect* drückt aus, welchen Effekt eine bestimmte Bedingung hervorruft. Diese Bedingung kann nun direkt auf eine Klasse referenzieren, die eine Exception repräsentiert, wie beispielsweise „InvalidISBN“.

- **Process Model:** Das *Process Model* definiert, wie ein Aufrufer mit dem zugehörigen Service interagieren kann. Im einfachsten Fall lässt sich ein Service als *atomic process* beschreiben. Ein solcher atomarer Prozess kommt einem gewöhnlichen Methodenaufruf gleich und besteht aus einer einzelnen, abgeschlossenen Interaktion zwischen Service und Aufrufer.

  Komplexere Interaktionen mit dem Service sind ebenfalls möglich und können über *composite processes* modelliert werden. Solcherlei Prozesse verfügen über einen Zustand und können aus mehreren zusammengehörigen Interaktionen zwischen Aufrufer und Service bestehen.

- **Service Grounding:** Über das *Grounding* erfolgt die Zuordnung einer abstrakten Beschreibung eines Semantic Web Service zu dessen konkreter Realisierung. Typischerweise handelt es sich dabei um ein Web Service in WSDL. Im *Grounding* muss daher spezifiziert werden, welche WSDL-Operationen einen Prozess aus dem *Process Model* implementieren. Im Falle eines *atomic process* lässt sich selbiger direkt einer einzigen WSDL-Operation zuordnen.

  Des Weiteren wird im Grounding definiert, wie Inputs und Outputs (die ja als Klassen einer bestimmten Ontologie repräsentiert sind) in konkrete WSDL-Nachrichten umgewandelt werden. Die Transformation für komplexe Datentypen kann dabei, ähnlich dem *schemaMapping* in SAWDSL, über ein XSLT-Stylesheet erfolgen.

Zum besseren Verständnis zeigt Abbildung 2.11 einen Auszug aus dem *EBookOrderService* zum Bestellen von digitalen Büchern (EBooks). Aus dessen Profil (*EBookOrderProfile*) geht hervor, dass das Service die beiden Input-Parameter *EBookRequest* sowie *UserAccount* entgegennimmt und als Output-Parameter ein *EBook* zurückliefert.
Jeder der drei Parameter ist im Process Model (*EBookOrderProcess*) über das *parameterType*-Element mit seinem jeweiligen Typ versehen. Beispielsweise verweist der Input *UserAccount* auf die Klasse *User* innerhalb der Ontologie *Books*. Anders als bei einem simplen String-Parameter weiß ein Aufrufer somit explizit, dass es sich bei besagtem Input-Parameter um das Konzept eines Nutzer-Accounts handelt.
Des Weiteren spezifiziert das Profil jeweils eine Precondition (*Authorization*) und einen Effect (*BookOrdered*). Anders als bei Inputs oder Outputs handelt es sich dabei um Ausdrücke (Expressions), welche im Process Model des Service näher spezifiziert werden und in diesem Beispiel in der *Semantic Web Rule Language* (SWRL) [34] notiert sind.
Die Precondition *Authorization* besteht aus einer Liste aus konjunktiv verknüpften Atomen (*AtomList*). Bei Atomen kann es sich um OWL-Konstrukte wie Klassen, ObjectProperties, DataProperties, Individuals, sameAs und differentFrom (Gleichheit bzw. Ungleichheit zweier Individuals) sowie zusätzlich speziellen SWRL Built-Ins handeln, die unter anderem für Vergleichsoperationen genutzt werden können.

```
<service:Service rdf:ID="EBookOrderService">
  <service:presents rdf:resource="#EBookOrderProfile" />
  <service:describedBy rdf:resource="#EBookOrderProcess" />
  <service:supports rdf:resource="#EBookOrderGrounding" />
</service:Service>

<!-- Profile -->
<profile:Profile rdf:ID="EBookOrderProfile">
  <service:presentedBy rdf:resource="#EBookOrder1Service" />
  <profile:serviceName xml:lang="en">EBookOrder</profile:serviceName>
  <profile:textDescription xml:lang="en">
     An e-book order web service, [...]
  <profile:textDescription>
  <profile:hasInput  rdf:resource="#EBookRequest" />
  <profile:hasInput  rdf:resource="#UserAccount" />
  <profile:hasOutput rdf:resource="#EBook" />
  <profile:hasPrecondition rdf:resource="#Authorization" />
  <profile:hasResult rdf:resource="#BookOrdered" />
</profile:Profile>

<!-- Process Model -->
<process:AtomicProcess rdf:ID="EBookOrderProcess">
  <service:describes rdf:resource="#EBookOrder1Service" />
  <process:hasInput  rdf:resource="#EBookRequest" />
  <process:hasInput  rdf:resource="#UserAccount" />
  <process:hasOutput rdf:resource="#EBook" />

  <!-- Precondition in SWRL-Syntax -->
  <process:hasPrecondition>
    <expr:SWRL-Condition rdf:ID="Authorization">
      <expr:expressionLanguage rdf:resource=
        "http://www.daml.org/services/owl-s/1.2/generic/Expression.owl#SWRL"/>
      <expr:expressionObject>
        <swrl:AtomList>
          <rdf:first>
            <swrl:ClassAtom>
              <swrl:classPredicate
                rdf:resource="http://example.ac.at/Book.owl#Authorized"/>
              <swrl:argument1 rdf:resource="#UserAccount"/>
            </swrl:ClassAtom>
          </rdf:first>
          <rdf:rest
            rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        </swrl:AtomList>
      </expr:expressionObject>
    </expr:SWRL-Condition>
  </process:hasPrecondition>
  ...
</process:AtomicProcess>

<!-- Input -->
<process:Input rdf:ID="UserAccount">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://example.ac.at/Book.owl#User
  </process:parameterType>
</process:Input>

<!-- Output -->
<process:Output rdf:ID="EBook">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://example.ac.at/Book.owl#Book
  </process:parameterType>
</process:Output>

<!-- Grounding -->
<grounding:WsdlGrounding rdf:ID="EBookOrderGrounding">
  ...
</grounding:WsdlGrounding>
```

Abbildung 2.11: Auszug aus einer OWL-S Service-Beschreibung.

Im Beispiel besteht die Liste an Atomen aus dem einzigen Klassen-Atom *Authorized*, welches als Argument auf das Input-Individual *UserAccount* verweist. Damit soll ausgedrückt werden, dass es sich beim an das Service übergebenen *UserAccount* um eine autorisierte Nutzerin handeln muss. Ist dies in einem gegebenen Aufruf-Kontext der Fall, wertet die Vorbedingung insgesamt zu wahr aus und gilt damit als erfüllt.
Andernfalls lässt sich das Service zwar ebenso aufrufen, jedoch ist die Ausführungssemantik in diesem Fall unbestimmt und ein Aufrufer muss mit einer Fehlfunktion rechnen.

KAPITEL 3

Related Work

In diesem Kapitel werden einige existierende Software- bzw. Service-Repositories vorgestellt und hinsichtlich ihrer Möglichkeiten zur Auffindung von Services (*Service Discovery*) untersucht. Danach werden unterschiedliche aktuelle Ansätze aus dem Gebiet des *(Semantic) Web Service Matchmakings* besprochen.

3.1 Repositories

3.1.1 Universal Description, Discovery & Integration (UDDI)

UDDI ist ein globales Register für Web Services. Es definiert ein standardisiertes, auf Kategorien basiertes Datenmodell. Sowohl Unternehmen, welche ihre Services veröffentlichen, als auch Services selbst können in vordefinierte Kategorien, beispielsweise über das *North American Industry Classification System* (NAICS) eingeteilt werden. Die Suche nach Services ist stark limitiert und basiert im Wesentlichen auf einer String-Suche. Suchanfragen nach der Funktionalität von Services im technischen Sinne sind nicht möglich.
Paolucci et al. [64] sowie Luo et al. [51] stellen jeweils experimentelle Ansätze zur Integration von OWL-S in UDDI vor, um Services auf Basis ihrer semantischen Beschreibung auffindbar zu machen. Beide Ansätze stellen jeweils ein Mapping zwischen OWL-S und dem UDDI-Datenmodell vor und erweitern UDDI um eine Komponente, welche die Suche nach Services anhand von Input- bzw. Output-Parametern erlaubt. Die Suche nach Vor- und Nachbedingungen ist nicht möglich und beide Ansätze bieten, wohl da im Hintergrund immer noch die limitierte Suchfunktionalität von UDDI genutzt wird, kein abgestuftes Ranking von gefundenen Services. Sinnvollerweise sollte dieser Schritt jedoch bereits von einem Repository erledigt werden.
Mittlerweile gibt es keine großen öffentlich zugänglichen UDDI-Registries mehr. Die drei größten an der Entwicklung beteiligten Unternehmen – IBM, Microsoft und SAP – haben ihre Dienste allesamt eingestellt. Unternehmensintern ist es mit UDDI-Implementierungen wie beispielsweise *Apache jUDDI* jedoch möglich, eine eigene UDDI-Registry zu betreiben.[1]

[1] jUDDI: https://juddi.apache.org

3.1.2 Vienna Runtime Environment for Service-oriented Computing (VRESCo)

Bei *VRESCo* [35] handelt es sich um eine an der TU Wien entwickelte Laufzeitumgebung, welche eine Infrastruktur für das Paradigma des *Service Oriented Computing* (SOC) bereitstellt. Als solches liegt der Fokus von *VRESCo* eher auf Aufgaben zur Laufzeit wie die dynamische Selektion, das dynamische Binding und der Aufruf von Web Services. Inkludiert in *VRESCo* ist jedoch auch eine eigenes implementierte Service Registry.
Das Datenmodell der *VRESCo Service Registry* unterscheidet zwischen abstrakten Features und konkreten Implementierungen von Services, was die Gruppierung von Services auf Basis deren Funktionalität ermöglicht. Über eine eigene, SQL-ähnliche Abfragesprache namens VQL können Suchanfragen nach Services anhand von Kriterien, die erfüllt werden sollen, formuliert werden. Ein Kriterium kann beispielsweise die Kategorie oder der Name eines Service, aber auch der Input oder Output einer Methode sein. Zudem können Kriterien als exakt oder optional deklariert werden und durch logische Operatoren zu komplexen Suchanfragen verbunden werden.
Die Benutzung der Such-API erfordert demnach eine gemäß der VQL exakt formulierte Suchanfrage und stellt somit den Gegenentwurf zu einer einfachen Suchmaske dar. Eine Nutzerin muss im Vorfeld sehr genau wissen, nach welchen Kriterien sie eine Anfrage formuliert. Dementsprechend werden auch nur exakt mit der Anfrage übereinstimmende Ergebnisse anstelle eines Rankings von Services zurückgeliefert.

3.1.3 Brechó Repository

Die Autoren von [55] beschäftigen sich insbesondere mit der Wiederverwendbarkeit von traditionellen Softwarekomponenten und verfolgen mit dem *Brechó Repository* den Ansatz, Sofwarekomponenten in unterschiedlichen Programmiersprachen automatisiert in (WSDL-)Services zu konvertieren.
Der Benutzer kann über eine graphische Benutzeroberfläche darüber entscheiden, welche Methoden (beispielsweise einer Java-Klasse) zu einem Service konvertiert werden sollen. Danach wird die Komponente, abhängig von der verwendeten Programmiersprache, in ein Web Service transformiert, welches fortan vom Repository bereitgestellt wird und dessen Aufruf über selbiges verwaltet wird. Externe Web Services werden ebenso unterstützt.
Neben der automatisierten *Service Generation* bietet *Brechó* zudem Module für die Zugriffskontrolle sowie zur Kostenabrechnung von Services zur Laufzeit. In diesem Sinne lässt sich *Brechó* nicht mehr als klassisches Repository zur reinen Verwaltung von Services klassifizieren – ähnlich zu *VRESCo* erstreckt sich dessen Anwendungsgebiet auch auf die Laufzeit.

3.1.4 Maven Central Repository

Central ist ein globales Repository für das bekannte Java Build-Tool *Maven* zur Speicherung von Java-Bibliotheken. Jedes Maven-Artefakt wird in einem standardisierten Format, bestehend aus *groupId*, *artifactId* sowie *version* gespeichert und indiziert. Laut eigener Statistik umfasst das Repository mittlerweile über 700.000 indizierte Komponenten mit über einem Terabyte Speicherbedarf (Stand: Juli 2014).

Eine zugehörige Suchmaschine erlaubt es, lediglich nach besagten drei Attributen sowie dem Namen einer Klasse zu suchen.[2] Die Suche nach Komponenten anhand ihrer Funktionalität ist nicht möglich, auch nicht anhand ihrer Dokumentation in Form von JavaDoc.

3.1.5 Online Web Service Repositories

Sabou et al. [74] untersuchen neben UDDI sechs verschiedene Web Service Repositories wie *XMethods* oder *WebserviceX.NET* mit besonderem Fokus auf der Auffindbarkeit von Services.[3] Die untersuchten Repositories bieten jeweils eine Reihe von frei verfügbaren, meist WSDL-basierten Web Services zur freien Nutzung, welche entweder selbst zur Verfügung gestellt oder aus anderen Quellen aggregiert werden. Die verfügbaren Services sind jeweils über ein Web Portal zugänglich, welches das Browsen sowie die Suche nach Services ermöglicht.
Alle untersuchten Repositories leiden unter einer sehr eingeschränkten Suchfunktionalität von (exakter) Schlagwort- über Substring- bis hin zu Kategorien-Suche, welche jeweils auf der Dokumentation der Services beruht. Keines der Repositories bezieht Information über die Funktionalität bzw. die Semantik von Services mit ein.

3.2 Service Matchmaking

Im Unterschied zur reinen *Service Discovery*, welche den Vorgang der Auffindung eines Service auf Basis dessen Spezifikation in Form von funktionalen bzw. nicht-funktionalen Eigenschaften bezeichnet, beinhaltet das *Service Matchmaking* darüber hinaus auch eine qualitative Bewertung bzw. Reihung der gefundenen Ergebnisse. Finden sich mehrere passende Services, ist es die Aufgabe eines Matchmaking-Algorithmus, aus einer Menge von Kandidaten eine Auswahl zu treffen und möglichst jene Services zurückzuliefern, welche eine gegebene Anfrage bestmöglich zu erfüllen vermögen.
Mittlerweile existiert eine Vielzahl an unterschiedlichen (Semantic) Service-Matchmakern, welche sich gemäß dem Diagramm aus Abbildung 3.1 kategorisieren lassen. Die Achsen des Diagramms repräsentieren dabei die beiden Hauptunterscheidungsmerkmale von Service Matchmakern [39] [40]:

1. Merkmale (Features) der Service-Spezifikation, die für das Matchmaking herangezogen werden. Dazu zählen Service-Signatur (Inputs/Outputs, IO), Service-Spezifikation (Preconditions/Effects, PE), monolithische Service-Beschreibung auf Grundlage von Logik, Text oder sonstigen Tags, nichtfunktionale Service-Parameter (NF) und Kombinationen aus letzterem mit jedem der vorigen Merkmale.

2. Art des Matching-Vorgangs. Dieser kann entweder logikbasiert, nicht logikbasiert oder auch aus einem hybriden Ansatz bestehen.

Der folgende Abschnitt gibt einen Überblick über die Bandbreite verschiedener Ansätze für die drei Arten des Matchings und stellt jeweils einige ausgewählte Vertreter jeder Gruppe vor.

[2] Maven Central Repository Search: http://search.maven.org
[3] XMethods: http://xmethods.com, WebserviceX.NET: http://www.webservicex.net

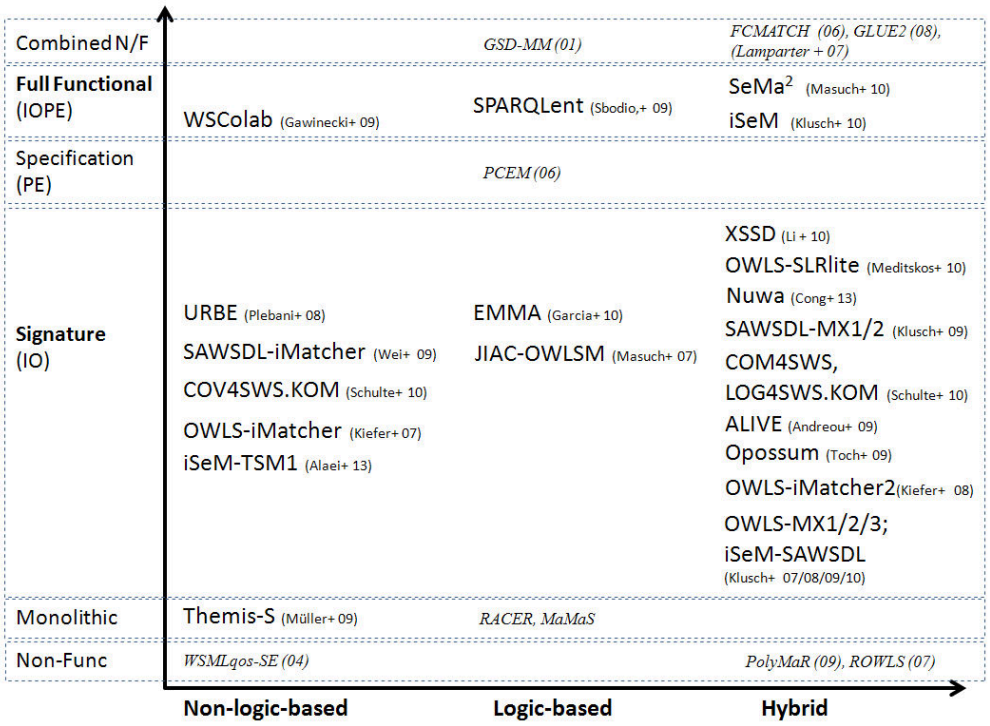

Abbildung 3.1: Klassifizierung von Semantic Service-Matchmakern (Kopie aus [46]).

3.2.1 Logikbasiert

Logikbasierte Matchmaker nutzen die verfügbare semantische Information von Services wie Inputs, Outputs, Preconditions und Effects (IOPE), welche formal in einer Beschreibungslogik wie der Web Ontology Language (OWL) spezifiziert sind. Beim logikbasierten Matching werden von einem *Reasoner* in einem Inferenz-Schritt Beziehungen zwischen logischen Konzepten ermittelt und anschließend dazu genutzt, den Grad der semantischen Übereinstimmung einer Suchanfrage mit einem Service zu bestimmen.
Rein logikbasierte Matchmaker stellen die verhältnismäßig kleinste Gruppe dar. Ein aktueller Vertreter aus dieser Kategorie ist *EMMA* [26], welcher in einer Filterphase bereits jene Services aussortiert, die eine Übereinstimmung mit einer Suchanfrage nicht erfüllen können. Dies geschieht durch die Transformation der Anfrage in die *SPARQL Protocol And RDF Query Language* (SPARQL), einer SQL-ähnlichen Abfragesprache für RDF. Die so gefilterten Services werden anschließend an einen integrierten Matchmaker (OWLS-MX3) weitergereicht, der das eigentliche Matchmaking übernimmt.
OWLSM [36] operiert auf der I/O-Signatur von OWL-S Services und ermittelt logische Beziehungen zwischen Inputs und Outputs der Suchanfrage mit einem Service (I/O-Subsumption).
SPARQLent [75] berücksichtigt zudem auch Preconditions und Effects, die in SPARQL ausgedrückt sind und deren Erfüllung bzw. Nichterfüllung in einem gegebenen Kontext ebenfalls über eine SPARQL-Query ausgewertet wird.

PCEM [7] arbeitet ausschließlich mit Preconditions und Effects von Services, welche jeweils über die *Planning Domain Definition Language* (PDDL) spezifiziert sein können, und transformiert diese in ein Prolog-Programm. Anschließend wird ein Prolog-Reasoner für das eigentliche Matchmaking benutzt. Dieser überprüft, ob Preconditions und Effects einer gegebenen Service-Anfrage von einem vorhandenen Service entweder exakt oder zumindest teilweise erfüllt werden können.

3.2.2 Nicht-logikbasiert

Nicht-logikbasierte Matchmaker führen keinerlei Reasoning durch, um den Übereinstimmungsgrad zwischen einer Suchanfrage und einem Service zu bestimmen und arbeiten damit nicht auf semantischer, sondern auf rein syntaktischer Ebene. Dies reduziert zwar einerseits den Aufwand der semantischen Annotation von Services, typischerweise jedoch auch die Ausdrucksstärke der Suche [62].
Geläufige nicht-logikbasierte Ansätze sind:

- Auffassung eines Service als Menge von strukturiertem oder unstrukturiertem Text. Auf Basis einer geeigneten Metrik wird die textuelle Ähnlichkeit zu einer Suchanfrage ermittelt. Dieser Ansatz findet ebenso in modernen (Web-)Suchmaschinen Verwendung.
- Transformation eines Service in einen Graphen und anschließendes Matching basierend auf der Ähnlichkeit zweier Graphstrukturen. Ein konkreter Ansatz hierzu findet sich in [16].
- Ermittlung der Distanz (Pfadlänge) zwischen Service-Konzepten. Beispielsweise organisiert die lexikalische Datenbank *WordNet* Hauptwörter und Verben in einer „is-a“ Hierarchie, sodass die Distanz zwischen zwei Konzepten innerhalb dieser Hierarchie über deren jeweilige Vor- und Nachfahren bestimmt werden kann [67]. Je kürzer der Pfad, desto höher die Ähnlichkeit.

Die ursprüngliche Version des *OWLS-iMatcher* interpretiert Service-Signaturen als reinen Text und ermittelt die Ähnlichkeit zu einer Suchanfrage über diverse *text-similarity* Metriken. Neuere Versionen des *OWLS-iMatcher* [37] sowie des *SAWSDL-iMatcher* [84] beziehen jeweils auch Beziehungen zwischen I/O-Konzepten innerhalb einer Ontologie mit ein.
Themis-S [1] zählt zu der kleinen Gruppe der monolithischen Matchmaker. Er kombiniert die klassische textbasierte Suche über das *Vector Space Model* (VSM), in welchem Dokumente bzw. deren Terme als Vektoren in einem Vektorraum angesehen werden, mit WordNet. Durch die Ermittlung von Synonymen, welche sich über die WordNet-Datenbank abfragen lassen, werden auch Services mit Suchbegriffen gefunden, die zwar nicht wörtlich, aber aus semantischer Sicht in Service-Beschreibungen vorhanden sind. Dadurch wird das sogenannte *vocabulary problem* eingedämmt, welches aus der unterschiedlichen Bezeichnung derselben Konzepte mit verschiedenem Vokabular resultiert [25].
WSColab [27] führt diesen Ansatz noch einen Schritt weiter: Ähnlich zu modernen Web 2.0 Plattformen können Services gemeinschaftlich mit Schlüsselwörtern, sogenannten *tags*, versehen werden. Daraus ergibt sich eine erweiterte Service-Beschreibung, die nicht mehr statisch

durch das Service-Dokument alleine vorgegeben ist, sondern von den Nutzern mitgestaltet werden kann. Der Matchmaker bezieht die so entstandenen *tag clouds* bei der Berechnung der textuellen Ähnlichkeit zu einer gegebenen Suchanfrage mit ein.

3.2.3 Hybrid

Hybride Matchmaker kombinieren die Resultate aus logikbasiertem und nicht logikbasiertem Matching zu einer einzigen Metrik, auf Basis derer sie Services bewerten, reihen und gemäß dieser Reihung zurückliefern.
Durch die Verwendung mehrere Metriken wird allgemein eine größere Flexibilität erzielt, weshalb hybride Matchmaker zahlenmäßig die größte Gruppe darstellen. Wenn beispielsweise einzelne Informationen eines Service wie dessen Kategorie oder die Typen von Input- bzw. Output-Parametern fehlen, kann das Service dennoch als Ergebnis aufscheinen, etwa wenn es eine hohe textuelle Ähnlichkeit mit der Suchanfrage aufweist. Eine Schwierigkeit liegt hauptsächlich darin, die unterschiedlichen Einzel-Metriken am Ende zu einer Gesamt-Metrik zu verbinden [62].
Als repräsentativer aktueller Vertreter dieser Gruppe kann der hybride Matchmaker *SeMa*^{2} [58], der Nachfolger von *JIAC-OWLSM*, angesehen werden. Dieser kombiniert die folgenden Aspekte zu einer (gewichteten) Gesamtmetrik: Namensgleichheit von Services, textuelle Ähnlichkeit, logikbasiertes I/O-Matching sowie die logikbasierte Übereinstimmung der Prädikate und Argumente von Preconditions & Effects (in SWRL-Syntax).
OWLS-MX in den Versionen 1 bis 3 [42] [43] kombiniert logikbasiertes I/O-Matching mit textueller Ähnlichkeit. Dessen Nachfolger *iSeM* [44] bezieht zusätzlich Preconditions und Effects in SWRL-Syntax sowie die strukturelle Ähnlichkeit von Services in den Matchmaking-Prozess mit ein. Die optimale Gewichtung der verschiedenen Metriken wird in einem Vorbereitungsschritt mittels Techniken aus dem *Machine Learning* auf Basis eines Trainings-Datensets ermittelt und justiert (was jedoch das Vorhandenseins eines solchen Trainingsdatensets voraussetzt).
Mit *SAWSDL-MX1/2* [45] und *iSeM-SAWSDL* [44] existiert auch jeweils eine Version der beiden letzteren Matchmaker für SAWSDL ohne P/E-Matching.

KAPITEL 4

SEnSoR – Semantically Enhanced Software Repository

Bei SEnSoR handelt es sich um ein Software Repository, das sowohl traditionelle Softwarekomponenten als auch (Semantic) Web Services unterstützt. Die Anwendung ist für die Nutzung innerhalb einer geschlossenen Umgebung ausgelegt und kann auf einem beliebigen Application Server innerhalb eines Netzwerks installiert werden. Die Funktionalität von SEnSoR ist über eine Web-Oberfläche, eine API sowie ein Plugin für die Entwicklungsumgebung *Eclipse* nutzbar. In diesem Kapitel werden, ausgehend von den Anforderungen, die zugrundeliegende Architektur und die in SEnSoR verwendeten Konzepte vorgestellt.

4.1 Anforderungen

SEnSoR ist in erster Linie als Entwicklerwerkzeug konzipiert, das Nutzerinnen bei der Auswahl von passenden Services unterstützen soll. Die konzeptionelle Idee hinter SEnSoR besteht darin, vorhandene Softwarekomponenten und Web Services einheitlich in einem zentralen Repository zu verwalten, sodass diese in einem größeren Kontext – wie etwa der Modellierung und Exekution von Geschäftsprozessen – (wieder-)verwendet werden können.
Konkret werden die folgenden Anforderungen von SEnSoR abgedeckt:

- **Einheitliches Datenmodell:** Services sollen unabhängig von ihrer ursprünglichen Beschreibungssprache in ein einheitliches Datenmodell konvertiert werden. Diese Aufgabe wird bei der Aufnahme in das Repository sprachabhängig von einem Parser übernommen, welcher Informationen über das Service aus dessen Beschreibung extrahiert. Die extrahierten Daten werden anschließend in einem Konvertierungsschritt auf das Datenmodell gemappt. Die Implementierung von SEnSoR soll als *proof-of-concept* mit Java, WSDL und OWL-S je einen Vertreter von traditionellen Softwarekomponenten, Web Services, sowie Semantic Web Services unterstützen.

- **Service-Management:** Das Repository soll Funktionalität zum Hinzufügen und Löschen von Services bereitstellen. Des weiteren soll es möglich sein, semantische Information in Form von Referenzen auf eine Domänen-Ontologie nachträglich hinzuzufügen bzw. zu editieren.

- **Suche nach Services & Methoden:** Die wichtigste Aufgabe aus Nutzersicht ist das Auffinden von passenden Services bzw. Methoden. Die Suchfunktionalität in SEnSoR soll einen hybriden Ansatz verfolgen, der die klassische textbasierte Suche mit einer logikbasierten semantischen Suche nach *Inputs*, *Outputs*, *Preconditions* & *Effects* (IOPE) verbindet.

- **Browsing:** Neben der gezielten Suche soll SEnSoR auch das Browsing von Services und Methoden nach Kategorie und Typ sowie deren Filterung bzw. Sortierung nach Parametern wie Name, Qualifier etc. unterstützen.

- **Performanz & Skalierbarkeit:** Ein Nachteil von insbesondere logikbasierten Service-Matchmakern ist oft deren Performanz. Besonders bei einer hohen Anzahl an indizierten Services kann sich die durchschnittliche Antwortzeit einer Suchanfrage auf einen Wert im zweistelligen Sekundenbereich belaufen [46]. Im Sinne der Benutzerfreundlichkeit soll sich SEnSoR eher an modernen Web-Suchmaschinen orientieren, welche üblicherweise bereits nach einem Bruchteil einer Sekunde Ergebnisse liefern.

- **Administration:** Eingriffe in das System wie die Konfiguration oder die Nutzerverwaltung sollen über eine Web-Oberfläche zugänglich sein. Diese kann innerhalb eines Netzwerks unkompliziert über den Browser verwendet werden und bedarf somit keiner vorigen Installation auf Client-Seite.

- **Integration:** Die Nutzung von SEnSoR soll sich in bestehende Softwarelandschaften integrieren lassen und über eine einfache API sowie über ein Plugin für die Entwicklungsumgebung *Eclipse* möglich sein. Für den plattformübergreifenden Zugriff auf die API soll diese als Web Service realisiert sein.

- **Security:** Im Unterschied zu UDDI ist SEnSoR nicht als globales Repository, sondern für die Nutzung innerhalb einer geschlossenen Umgebung wie beispielsweise einer Firma konzipiert. Der Zugriff auf die Funktionalität von SEnSoR soll dennoch nur autorisierten Nutzern mit unterschiedlichen Rechten gestattet sein. SEnSoR soll zu diesem Zweck ein auf Rollen basiertes Zugriffsmodell unterstützten.

4.2 Architektur

Die Architektur von SEnSoR ist in Abbildung 4.1 dargestellt und folgt dem klassischen 3-Schichten-Architekturmuster, bestehend aus Präsentation (*user interface*, UI), Anwendungslogik sowie Persistenz- bzw. Datenschicht.
Die Anwendungslogik ist in folgende Module unterteilt:

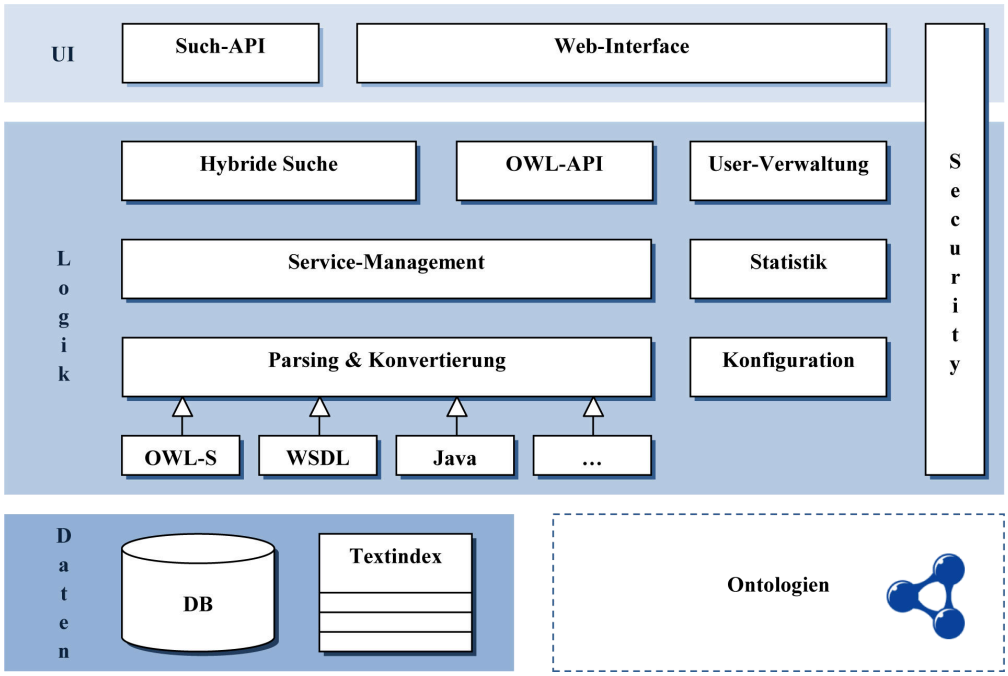

Abbildung 4.1: Die Architektur von SEnSoR .

- **Parsing:** Das Parsing-Modul liest Services in Form von hochgeladenen Dateien oder URIs ein und ermittelt zunächst den Typ eines Service aus dessen Dateiendung. Je nach Typ – momentan unterstützt sind OWL-S, WSDL sowie Java (Interfaces) – extrahiert eine entsprechende Parser-Implementierung sämtliche vorhandenen Informationen eines Service sowie der darin enthaltenen Methoden und konvertiert diese in ein gemeinsames Datenmodell (siehe 4.3). Nach einem erfolgreichen Parsing-Vorgang wird das Service in der Datenbank persistiert und gleichzeitig in den Textindex aufgenommen.

- **Service-Management:** Dieses Modul bildet die Schnittstelle zur Persistenz-Schicht und bietet Funktionen zum Speichern und Löschen indizierter Services. Weiters ermöglicht es das nachträgliche Annotieren von Inputs, Outputs sowie Preconditions und Effects von Methoden über das Web-Interface und bietet eine Reihe von Abfragemöglichkeiten nach Services und Methoden, welche sowohl für das Browsen als auch für die gezielte Suche benötigt werden.

- **Hybride Suche:** Das Suchmodul deckt die Kernfunktionalität der Suche nach Services bzw. Methoden sowohl auf textueller, als auch auf semantischer Ebene ab. Intern ist hierfür ein Matchmaking-Algorithmus implementiert, der sich gemäß 3.2 als hybrid klassifizieren lässt und logikbasiertes IOPE-Matching mit textbasiertem Matching basierend auf der vollen Service-Spezifikation kombiniert (siehe 4.5).

  Die Suchfunktionalität ist sowohl über das Web-Interface als auch über die Such-API verfügbar, welche als Web Service angeboten wird (siehe 4.6.2).

- **OWL-API:** Die OWL-API bietet Methoden für die Suche nach Klassen, Object- oder DataProperties innerhalb von Ontologien sowie zur Ermittlung deren Beziehungen untereinander und wird hauptsächlich im Rahmen der semantischen Suche nach Methoden verwendet. Intern stützt sich das Modul auf den OWL-DL Reasoner *Pellet* [78].[1]

- **User-Verwaltung:** Die User-Verwaltung dient zum Editieren und Löschen vorhandener Benutzerkonten sowie zur Erstellung neuer Benutzereinträge.

- **Konfiguration:** Das Konfigurationsmodul bietet hauptsächlich die Möglichkeit, diverse Parameter der (textuellen) Suche zu justieren. Dazu zählen unter anderem die Anzahl an Suchergebnissen oder das (De-)Aktivieren der Abfrageerweiterung (*query expansion*, QE), welche die von der Nutzerin eingetippten Begriffe automatisch durch Synonyme aus einem Thesaurus erweitert, um die Anzahl an Suchergebnissen zu erhöhen [63].

- **Statistik:** Das Statistik-Modul aggregiert statistische Daten aus anderen Modulen. Es gibt Auskunft über die Anzahl indizierter Services und diverse Kenngrößen des Textindex wie die durchschnittliche Anzahl an Termen oder die am häufigsten auftretenden Terme innerhalb aller Service-Dokumente. Des Weiteren bietet es eine detaillierte Aufschlüsselung des durchschnittlichen Antwortzeitverhaltens aller bisherigen Suchanfragen sowie eine Übersicht über alle von Services referenzierten Ontologien.

Technologisch gesehen ist SEnSoR eine Java Enterprise-Anwendung mit Web-Oberfläche. Die Module der Anwendungslogik sind demnach als *Enterprise Java Beans* (EJBs) implementiert. Die Präsentationsschicht ist über *Java Server Faces* (JSF) realisiert. Für das objekt-relationale Mapping zwischen Logik- und Datenschicht gelangt die *Java Persistence API* (JPA) zum Einsatz.
SEnSoR wird als einzelnes Java Web-Archiv gepackt und läuft auf jedem zu Java EE 6 kompatiblen Application Server wie *Glassfish*.[2] An Abhängigkeiten benötigt SEnSoR eine laufende Instanz einer relationalen Datenbank wie *PostgreSQL*,[3] sowie eine Instanz der NoSQL-Datenbank *MongoDB* für den Textindex. Diese können jeweils auf demselben oder unterschiedlichen Servern innerhalb eines Netzwerks laufen.[4]
Der Zugriff auf die Funktionalität von SEnSoR über das Web-Interface oder die Such-API ist über eine rollenbasierte Zugriffskontrolle (*role based access control*, RBAC) [24] geregelt. Benutzer sind in die beiden Kategorien *Administrator* sowie *User* unterteilt und müssen sich zuerst über eine Kombination aus Benutzername und Passwort am System authentisieren, um Zugriff zu erhalten.
Der aktive Eingriff in das System über das Konfigurationsmodul und die Erstellung neuer Nutzerkonten ist dabei alleinig Administratoren vorbehalten. Die Implementierung des Security-Moduls stützt sich auf das populäre Java-Framework *Spring Security*.[5]

[1] Pellet: http://clarkparsia.com/pellet
[2] Glassfish: https://glassfish.java.net
[3] PostgreSQL: http://www.postgresql.org
[4] MongoDB: http://www.mongodb.org
[5] Spring Security: http://projects.spring.io/spring-security

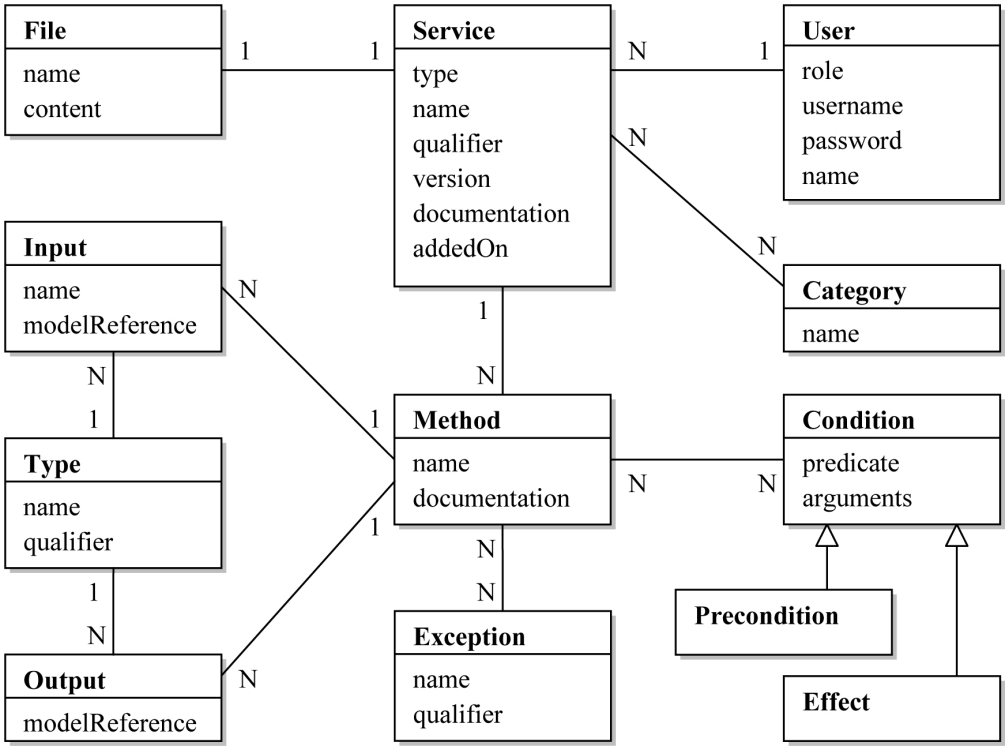

Abbildung 4.2: Das Datenmodell von SEnSoR in UML-Notation.

4.3 Datenmodell

Abbildung 4.2 zeigt das Datenmodell von SEnSoR, welches die funktionalen Eigenschaften von Services abdeckt. Die zentrale Entität in SEnSoR ist das Service. Services können unterschiedlichen Typs sein und kapseln ihre Funktionalität in jeweils mindestens einer Methode. Sowohl Services als auch Methoden haben jeweils einen Namen und einen Qualifier, welcher sich im Falle von Java aus dem Paketnamen und im Falle von Web Services aus der URI ergibt. Jedes Service ist genau einer Quelldatei zugeordnet, deren Inhalt unverändert in der Datenbank mit abgespeichert wird.

Services können von Benutzern, deren Konten (verschlüsselt) in der User-Tabelle hinterlegt sind, zum Repository hinzugefügt werden. Zudem kann jedes Service beliebig vielen Kategorien angehören, bei deren Namen es sich entweder um reinen Text oder eine URI auf ein Konzept aus einer Ontologie handeln kann.

Das *version*-Attribut dient der (simplen) Versionierung von Services. Wird eine neue Version eines Services zum Repository hinzugefügt, kann dieses intern unter einer höheren Versionsnummer abgelegt werden. Bei mehreren Versionen stellt das Service mit der höchsten Versionsnummer implizit das „aktive“ Service dar.

Des weiteren speichert SEnSoR die Signatur von Methoden in Form von Input- und Output-Parametern mit deren jeweiligen Typen. In Anlehnung an SAWSDL können Inputs und Outputs

über das Attribut *modelReference* mit einer Referenz auf eine Klasse innerhalb einer bestehenden Ontologie versehen werden, was die Basis für das logikbasierte I/O-Matching bildet. Darüber hinaus speichert SEnSoR auch Exceptions (sofern vorhanden).
Zusätzlich werden Preconditions und Effects von Methoden gespeichert. In beiden Fällen handelt es sich dabei aus Datensicht um eine Menge von Prädikaten mit zugehörigen Argumenten in Form von OWL-URIs, welche wiederum die Basis für das logikbasierte P/E-Matching bilden.

4.4 Parsing & Konvertierung

Das Konvertieren des Quellcodes eines Services in das gemeinsame Datenmodell erfolgt sprachabhängig und in zwei Schritten. Zunächst extrahiert ein Parser Informationen über das Service aus dessen Schnittstellenbeschreibung. Anschließend werden die vom Parser extrahierten Daten von einer Konvertierungskomponente in das einheitliche Datenmodell transformiert. Für den Parsing-Vorgang definiert SEnSoR das Interface `IServiceParser` und stellt selbst Implementierungen für WSDL 1.1, WSDL 2.0, OWL-S 1.2 sowie Java auf Basis vorhandener Bibliotheken bereit. Weitere Sprachen wie etwa C# können hinzugefügt werden, indem eine Klasse bereitgestellt wird, welches besagtes Interface implementiert.

4.4.1 Parsen von WSDL-basierten Services

Bei WSDL-Dokumenten handelt es sich im Grunde, wie in 2.2.3.1 beschrieben um XML-Dokumente. Diese lassen sich prinzipiell mit jedem beliebigen XML-Parser einlesen. Für WSDL existieren jedoch bereits mehrere Java-Bibliotheken, die jeweils eine einfachere API für das Lesen, Erstellen und Manipulieren von WSDL-Dokumenten bieten und deren Vor- bzw. Nachteile nachfolgend kurz vorgestellt werden.

- **Web Services Description Language for Java Toolkit (WSDL4J):** WSDL4J ist eine von IBM entwickelte Open-Source-Bibliothek und gleichzeitig die Referenzimplementierung für den *Java Specification Request* JSR 110, dessen Ziel es ist, eine einheitliche Java-API für WSDL bereitzustellen.[6] Die Bibliothek unterstützt lediglich WSDL 1.1, was einen großen Nachteil hinsichtlich der Kompatibilität darstellt.

  Semantische Annotationen über SAWDSL werden im Standard nicht unterstützt. Es existiert jedoch eine Erweiterung namens *SAWSDL4J*, welche die API zum Manipulieren der wichtigsten Elemente wie etwa „Operation“ und „Message“ dahingehend erweitert.[7] Da WSDL4J jedoch XML-Schema naiv als bloßes DOM-Objekt handhabt, erfolgt der Zugriff auf Schema-Elemente wie zum Beispiel Input-Typen eines Service in nicht einheitlicher Weise (und recht umständlich) über XPath-Abfragen.

- **Apache Woden:** Woden ist Teil des *Apache Web Services Project*, verfolgt dasselbe Ziel wie WSDL4J und ist ebenfalls open-source.[8] Der Hauptunterschied ist, dass Woden expli-

[6] WSDL4J: http://sourceforge.net/projects/wsdl4j/files/WSDL4J
[7] SAWSDL4J: http://sawsdl4j.sourceforge.net
[8] Woden: http://ws.apache.org/woden

zit nur WSDL 2.0 unterstützt. Zu dem Projekt gehört zwar ein Werkzeug für die Konvertierung von WSDL 1.1 nach 2.0, doch zum reinen Parsen von Services ist dieser zusätzliche Schritt unnötig und zudem potentiell verlustbehaftet.

SAWSDL wird auch in Woden nicht standardmäßig unterstützt. Wie bei WSDL4J existiert auch hier eine Erweiterung namens *Woden4SAWSDL*, welche die komplette SAWSDL-Spezifikation abdeckt.[9]

- **EasyWSDL:** Die *Easy WSDL Toolbox* unterstützt im Gegenzug zu den oben genannten Bibliotheken sowohl WSDL 1.1 als auch WSDL 2.0 in einer einheitlichen API [5], die sich bezüglich der Namensgebung an WSDL 2.0 orientiert (etwa *getInterface()* statt *getPortType()*).[10]

  SAWSDL ist sehr einfach über ein Plugin integrierbar. Zwar sind theoretisch auch Schema-Elemente direkt über die API zugänglich, was das Parsen von Input- bzw. Output-Typen von Methoden erleichtern soll, doch die praktische Anwendung im Rahmen der Entwicklung von SEnSoR zeigt, dass diese Funktionalität in Kombination mit dem SAWSDL-Plugin in allen veröffentlichten Versionen bis hin zur aktuellsten Version 2.3 nicht immer funktioniert und öfters Exceptions auslösen kann.

Für das Parsen von WSDL-Dokumenten in SEnSoR kommt *EasyWSDL* zum Einsatz, hauptsächlich deshalb, da es sowohl WSDL 1.1 als auch 2.0 unterstützt. Das oben erwähnte Problem im Zusammenhang mit Schema-Typen wird durch die Verwendung von eigenem Code (im Wesentlichen XPath-Abfragen) umgangen. Abbildung 4.3 zeigt anhand eines WSDL 1.1-Dokuments, welche Service-Attribute während des Parsens extrahiert werden.
Wie in 2.2.3.3 erwähnt, gibt es weder in WSDL noch der Erweiterung SAWSDL ein direktes Sprachmittel zur Spezifikation von Preconditions und Effects. Über das *modelReference*-Attribut ist es jedoch möglich, einen entsprechenden Verweis auf eine SWRL-Regel zu setzen.[11]
Service-Kategorien sind nicht direkt innerhalb eines WSDL-Dokuments spezifizierbar und können später bei Bedarf über das Web-Interface nachgetragen werden. WSDL-Faults, in Abbildung 4.3 zwar nicht vorhanden, werden ebenfalls mit eingelesen.

4.4.2 Parsen von OWL-S-basierten Services

Für das Parsen von Services in OWL-S existiert mit der *OWL-S Java API* [77] eine Bibliothek zum Erstellen, Einlesen und Editieren von OWL-S Dokumenten.[12] Darüber hinaus ist es sogar möglich, ein OWL-S Service zu exekutieren, sofern dieses über ein Grounding, beispielsweise in Form eines ausführbaren WSDL-Dokuments (siehe 2.3.1), verfügt. Die Bibliothek unterstützt OWL-S in den Versionen 1.1 und 1.2 sowie SWRL. Ähnlich zu den im vorigen Abschnitt vorgestellten WSDL-Bibliotheken bietet die OWL-S API einfache *get*-Methoden zum Zugriff auf sämtliche in OWL-S spezifizierten Konzepte eines Service.

[9] Woden4SAWSDL: http://lsdis.cs.uga.edu/projects/meteor-s/opensource/woden4sawsdl
[10] EasyWSDL: http://easywsdl.ow2.org
[11] Representing Conditions in SAWSDL: http://www.w3.org/2002/ws/sawsdl/spec/examples/#conditions
[12] OWL-S API: http://on.cs.unibas.ch/owls-api/

```
<wsdl:definitions name="Ebookorder1"                                          Service.qualifier, uri
                  xmlns="http://example.ac.at/Ebookorder1"...>

<wsdl:types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
    <xsd:simpleType name="TitleType"
                    sawsdl:modelReference="http://example.ac.at/Book.owl#Title">   Input.modelReference
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
    <xsd:simpleType name="UserType"
                    sawsdl:modelReference="http://example.ac.at/Book.owl#User">    Input.modelReference
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
    <xsd:complexType name="BookType"
                     sawsdl:modelReference="http://example.ac.at/Book.owl#Book">   Output.modelReference
      ...
    </xsd:complexType>
  </xsd:schema>
</wsdl:types>

<wsdl:message name="getEBookRequest">
  <wsdl:part name="EBookRequest" type="TitleType" />                           Input.name, type
  <wsdl:part name="UserAccount" type="UserType" />                             Input.name, type
</wsdl:message>
<wsdl:message name="getEBookResponse">
  <wsdl:part name="EBook" type="BookType" />                                   Output.type
</wsdl:message>

<wsdl:portType name="Ebookorder1Soap">
  <wsdl:operation name="getEBook">                                             Method.name
    <wsdl:documentation> Retrieves EBooks [...] </wsdl:documentation>          Method.documentation
    <wsdl:input message="getEBookRequest"
        sawsdl:modelReference="http://example.ac.at/Book.owl#Authorization"/>  Precondition (Link)
    <wsdl:output message="getEBookResponse"
        sawsdl:modelReference="http://example.ac.at/Book.owl#BookOrdered" />    Effect (Link)
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="Ebookorder1SoapBinding" type="Ebookorder1Soap">
  ...
</wsdl:binding>

<wsdl:service name="Ebookorder1Service">                                       Service.name
  <wsdl:documentation>
    An e-book order web service, where an ebook request is given [...]         Service.documentation
  </wsdl:documentation>
  <wsdl:port name="Ebookorder1Soap" binding="Ebookorder1SoapBinding">
    <wsdlsoap:address location="http://example.ac.at/Ebookorder1"/>
  </wsdl:port>
</wsdl:service>

</wsdl:definitions>
```

Abbildung 4.3: Extrahierte Information aus einem WSDL-Dokument.

Abbildung 4.4 zeigt an einem Beispieldokument, welche Informationen aus einem Service extrahiert werden. Da die OWL-S Terminologie im Gegensatz zu WSDL keine Methoden kennt, wird ein OWL-S Service in SEnSoR als Service mit einer einzigen, gleichnamigen Methode aufgefasst.
Informationen wie Name, textuelle Beschreibung sowie Inputs, Outputs, Preconditions und Effects werden aus dem *Service Profile* entnommen, deren Konzepte jeweils auf das *Process Model* verweisen (in Abb. 4.4 farblich dargestellt). Anders als bei WSDL können Prädikate und Argumente von Preconditions bzw. Effects explizit aus dem Dokument heraus geparst werden. Das *Grounding* (im Beispiel nicht dargestellt) ist für das Parsen irrelevant.

4.4.3 Parsen von traditionellen Softwarekomponenten am Beispiel Java

Die meisten Programmiersprachen bieten standardmäßig kein dezidiertes Sprachmittel, um Konzepte wie Preconditions bzw. Effects auszudrücken. Eine rühmliche Ausnahme bietet die objek-

```
<rdf:RDF xml:base = "http://example.ac.at/EBookOrder1.owls ...>                  Service.qualifier, uri

<profile:Profile rdf:ID="EBookOrderProfile">
  <profile:serviceName xml:lang="en">EBookOrder</profile:serviceName>            Service & Method.name
  <profile:textDescription xml:lang="en">
      An e-book order web service, where an ebook request is given [...]        Service & Method
  </profile:textDescription>                                                           .documentation
  <profile:serviceCategory>E-Commerce</profile:serviceCategory/>
  <profile:hasInput        rdf:resource="#EBookRequest" />
  <profile:hasInput        rdf:resource="#UserAccount" />
  <profile:hasOutput       rdf:resource="#EBook" />
  <profile:hasPrecondition rdf:resource="#Authorization" />
  <profile:hasResult       rdf:resource="#BookOrdered" />
</profile:Profile>
...
<process:hasPrecondition>
  <expr:SWRL-Condition rdf:ID="Authorization">
  ...
  <swrl:classPredicate rdf:resource="http://example.ac.at/Book.owl#Accepted"/>  Precondition.predicate
  <swrl:argument1 rdf:resource="#EBookRequest"/>                                Precondition.argument[]
  ...
  <swrl:classPredicate rdf:resource="http://example.ac.at/Book.owl#Authorized"/> Precondition.predicate
  <swrl:argument1 rdf:resource="#UserAccount"/>                                 Precondition.argument[]
  </expr:SWRL-Condition>
</process:hasPrecondition>
<process:hasResult>
  <expr:SWRL-Condition rdf:ID="BookOrdered">
  ...
  <swrl:classPredicate rdf:resource="http://example.ac.at/Book.owl#Ordered"/>   Effect.predicate
    <swrl:argument1 rdf:resource="#EBook"/>                                     Effect.argument[]
  </process:hasResult>
  ...
  </expr:SWRL-Condition>
</process:AtomicProcess>

<process:Input rdf:ID="EBookRequest">                                           Input.type
  <rdfs:label>Title</rdfs:label>                                                Input.name
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
      http://example.ac.at/Book.owl#Title                                       Input.modelReference
  </process:parameterType>
</process:Input>
<process:Input rdf:ID="UserAccount">
  <rdfs:label>UserAccount</rdfs:label>                                          Input.type
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"> Input.name
    http://example.ac.at/Book.owl#User                                          Input.modelReference
  </process:parameterType>
</process:Input>
<process:Output rdf:ID="EBook">                                                 Output.type
  <rdfs:label>EBook</rdfs:label>
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://example.ac.at/Book.owl#Book                                          Output.modelReference
  </process:parameterType>
</process:Output>
```

Abbildung 4.4: Extrahierte Information aus einem OWL-S-Dokument.

torientierte Programmiersprache Eiffel, welche 1985 von dem französischen Informatiker Bertrand Meyer entworfen wurde und das Prinzip *Design by Contract* verfolgt. Dabei handelt es sich um einen Ansatz des Software-Designs, die Schnittstellenbeschreibung eines Programmmoduls um einen sogenannten „Vertrag“ zu erweitern, welcher die einzuhaltenden Bedingungen sowie die erwarteten Ergebnisse für den Aufruf einer Methode in Form von Preconditions, Effects (sowie im Falle von Eiffel auch Invarianten) beinhaltet. Der Aufrufer ist für die Einhaltung sämtlicher Preconditions vor einem Methodenaufruf verantwortlich, während die aufgerufene Methode die Einhaltung der Effects garantieren muss.

Abbildung 4.5 zeigt beispielhaft, wie solch ein Vertrag konkret aussieht. Die Methode *withdraw* zum Abheben eines Geldbetrags *amount* von einem Konto verfügt über zwei benannte Vorbedingungen und eine Nachbedingung. Die *require*-Klausel stellt einerseits sicher, dass der abzuhebende Betrag tatsächlich positiv ist (Zeile 4) und andererseits, dass nicht mehr Geld abgehoben wird als derzeit auf dem Konto verfügbar, ausgedrückt durch die Instanzvariable *balance*

```
 1. withdraw (amount: INTEGER)
 2.   -- Withdraws a given amount of cash from the account.
 3.   require
 4.     valid_amount:    amount > 0
 5.     wont_overdraw:   amount >= balance
 6.   do
 7.     balance := balance - amount
 8.   ensure
 9.     balance_correct: balance = (old balance) - amount
10.   end
```

Abbildung 4.5: Eine Funktion mit Vor- und Nachbedingungen in Eiffel.

(Zeile 5). Die eigentliche Funktionalität der Methode findet sich innerhalb der *do*-Klausel (Zeile 7). Über die *ensure*-Klausel wird nach Ausführung kontrolliert, ob der Betrag auch wirklich korrekt abgebucht wurde. Verletzt ein Funktionsaufruf eine der drei Bedingungen, wird zur Laufzeit eine Exception ausgelöst.
Im Sinne der Wiederverwendbarkeit und Wiederauffindbarkeit von Softwarekomponenten geht die Überprüfung von Bedingungen zur Laufzeit bereits einen Schritt zu weit. Vielmehr reicht es aus, der Schnittstellenbeschreibung Informationen über Vor- und Nachbedingungen in Form von Metadaten hinzuzufügen.
Informell lässt sich diese Aufgabe über Kommentare lösen, die in so gut wie allen Programmiersprachen verfügbar sind. In Java bietet sich dazu jedoch das Sprachmittel der *Annotation* an, welches mit Java 5 eingeführt wurde. Annotationen beginnen mit einem @-Symbol gefolgt von einem Namen und können optional über Attribute verfügen. Beispielsweise sagt die bekannte Annotation *@Deprecated* auf einer Methode aus, dass diese veraltet ist und zukünftig nicht mehr benutzt werden sollte.
Da in der Java-Klassenbibliothek keine passenden Annotationen zur Spezifikation der Semantik von Methoden vorhanden sind, bietet SEnSoR die folgenden vier eigenen Annotationen an: *@Input*, *@Output*, *@Precondition* sowie *@Effect*. Zudem existiert mit *@Category* zusätzlich eine Annotation zur Einteilung eines Service in eine oder mehrere Kategorien. Abbildung 4.6 zeigt zwei der besagten Annotationen.
Die Attribute *modelReference* (angelehnt an SAWDSL), *predicate* sowie das Array *arguments* werden jeweils dazu benutzt, auf ein Konzept einer bestehenden Ontologie zu referenzieren. Die Anwendung der Annotationen sind jeweils auf die entsprechenden Sprachelemente wie auf Input-Parameter (für *@Input*), Methoden (für *@Output*, *@Precondition* *@Effect*) sowie Klassen (für *@Category* beschränkt (Zeilen 1 & 6). Des weiteren hat keine der Annotationen Einflüsse auf die Laufzeit des Programms, da diese vom Übersetzer nach dem Kompilieren – ähnlich wie Kommentare – entfernt werden (Zeilen 2 & 7).
Für das eigentliche Parsen von Java-Klassen wird die Bibliothek *QDox* verwendet.[13] QDox liest Java-Klassen ein und konvertiert diese in ein eigenes Datenmodell, welches anschließend über eine Fülle von API-Methoden wie *getMethods()* oder *getAnnotations()* abgefragt werden kann. Ebenso lassen sich auch die oben vorgestellten Annotationen parsen.

[13] QDox: http://qdox.codehaus.org

```
1.  @Target(ElementType.PARAMETER)
2.  @Retention(RetentionPolicy.SOURCE)
3.  public @interface Input {
4.    String modelReference();
5.  }
6.  @Target(ElementType.METHOD)
7.  @Retention(RetentionPolicy.SOURCE)
8.  public @interface Precondition {
9.    String predicate();
10.   String[] arguments();
11. }
```

Abbildung 4.6: Java-Annotationen für Input-Parameter und Vorbedingungen von Methoden.

Abbildung 4.7 zeigt an einem beispielhaften, annotierten Java-Interface, welche Informationen (fettgedruckt) aus einer Klasse extrahiert und anschließend in das Datenmodell von SEnSoR konvertiert werden:

```
package at.ac.example;                                                    Service.qualifier

/**
 * This service provides methods to order books.                          Service.documentation
 */
@Categories({@Category("E-Commerce"),@Category("Book")})                  Service.category[]
public interface IBookService {                                          Service.name

  /**
   * Orders a book by the given ISBN.
   *
   * @param isbn The International Standard Book Number of the book.
   * @return The corresponding "Book"-entity or null if not found.       Method.documentation
   * @throws InvalidISBNException If the format of the ISBN is not valid.
   */
  @Output(modelReference = "http://example.ac.at/Book.owl#Book")          Output.modelReference
  @Precondition(predicate = "http://example.ac.at/Book.owl#ValidISBN",    Precondition.predicate
                arguments = {"ISBN"})                                     Precondition.argument[]
  @Effect(predicate = "http://example.ac.at/Book.owl#Ordered",            Effect.predicate
          arguments = {"Book"})                                           Effect.argument[]
  public Book                                                             Output.type
      orderBookByISBN (                                                   Method.name
      @Input(modelReference = "http://example.ac.at/Book.owl#ISBN")       Input.modelReference
             ISBN isbn)                                                   Input.type, name
      throws InvalidISBNException;                                        Exception.name
}
```

Abbildung 4.7: Extrahierte Information aus einem Java-Interface.

4.5 Suche

Die Suche nach Services und deren Funktionalität in Form ihrer Methoden ist eine Kernanforderung an das System und folgt einem hybriden Ansatz. Services und Methoden, die über keine semantische Annotation verfügen, können über eine textbasierte Suche gemäß ihrer Dokumen-

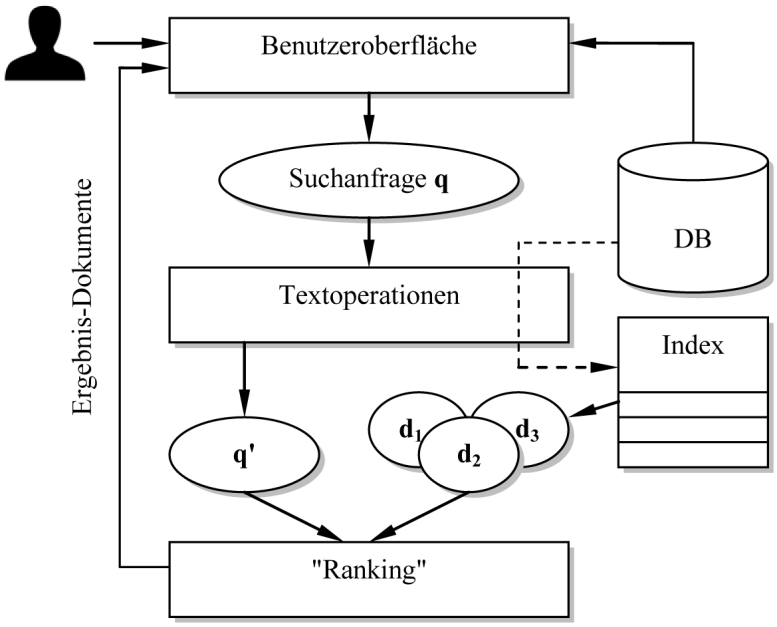

Abbildung 4.8: Der Suchprozess einer textbasierten Suche. Angelehnt an [28, Kap. 1] und [12].

tation, Methodennamen etc. gesucht werden. Für semantisch annotierte Services ist zusätzlich eine semantische Suche anhand ihrer *inputs, outputs, preconditions & effects* (IOPE) möglich.

4.5.1 Textbasierte Suche

Die Hauptaufgabe einer textbasierten Suche ist es, der Nutzerin zu einer gegebenen Suchanfrage möglichst relevante Ergebnisdokumente zu präsentieren. Als Grundlage dafür dienen indizierte Dokumente und deren textuelle Features wie enthaltene Wörter, deren Häufigkeiten, etc.
Im Idealfall erfolgt die Suche für die Nutzerin über eine einfach zu bedienende Benutzerschnittstelle, die von erfolgreichen Suchmaschinen wie *Google* oder *Yahoo* maßgeblich geprägt wurde. Typischerweise tippt die Anwenderin einen oder mehrere Suchbegriffe in ein Textfeld und bekommt nach einem Klick die Ergebnisse präsentiert.
Der Suchprozess, wie in Abbildung 4.8 dargestellt, besteht dabei aus Sicht einer Suchmaschine aus mehreren Teilen:

1. **Parsen der Suchanfrage:** Über die Benutzeroberfläche formuliert die Anwenderin ihren sogenannten *information need* q in Form von Stichwörtern.

2. **Textoperationen:** Die eingegebene Suchanfrage q wird in ihre einzelnen Wörter zerlegt. Zudem finden an dieser Stelle diverse Aufbereitungsschritte statt, wie zum Beispiel die Reduktion eines Suchbegriffs auf dessen Wortstamm, sodass die ursprüngliche Anfrage q in die Zwischenrepräsentation q' transformiert wird. Näheres hierzu findet sich im Folgekapitel.

3. **Ranking:** Auf Basis der Suchbegriffe werden unter den indizierten Dokumenten relevante Dokumente d_1, d_2, ... d_n herausgefiltert. Der Begriff der Relevanz variiert hierbei und hängt von einem zugrundeliegenden Modell ab. Je nach Modell wird gemäß einer Ranking-Funktion für jedes Paar (q', d_1), (q', d_2), ... (q', d_n) ein Relevanz-Wert, üblicherweise in Form einer Zahl, berechnet. In 4.5.1.3 und 4.5.1.4 werden hierzu zwei konkrete Modelle vorgestellt.

4. **Präsentation der Ergebnisse:** Zuletzt werden die gefundenen Dokumente nach absteigender Relevanz gereiht, aus einer Datenbank geladen und über die Benutzeroberfläche präsentiert. Idealerweise umfasst die Präsentation eine kurze Zusammenfassung des Ergebnis-Dokuments, damit die Benutzerin interessante Ergebnisse schnell erkennen oder ihre Anfrage überarbeiten kann, woraufhin der Kreislauf von Neuem beginnen kann.

Die Aufnahme von Dokumenten in den Index, dargestellt durch den gestrichelten Pfeil, geschieht „offline“, ohne Nutzer-Beteiligung. Wichtig hierbei ist, dass die zu indizierenden Dokumente die gleichen Textoperationen durchlaufen wie später die Suchanfragen. Eine Suchanfrage q kann nämlich ebenfalls als (kurzes) Dokument d_q aufgefasst werden und ist dadurch mit den indizierten Dokumenten „kompatibel“.
Hinsichtlich der Antwortzeit des Systems ist mit zunehmender Anzahl von indizierten Dokumenten insbesondere Punkt 3 am kritischsten. Ein gut strukturierter Textindex sorgt als Herzstück des Systems für die schnelle Auffindbarkeit von Dokumenten anhand von enthaltenen Begriffen und ein damit einhergehendes Berechnungsverfahren ermöglicht das effiziente Ranking der gefundenen Dokumente.
Im Fall von Web-Suchmaschinen, die eine gewaltige Anzahl an Dokumenten indizieren, sind zudem weitere Optimierungsmaßnahmen nötig, um den Suchraum nicht explodieren zu lassen und die häufige, aufwendige Neuberechnung der Relevanz von Dokumenten einzuschränken. Da SEnSoR jedoch nicht wie etwa UDDI (ursprünglich) als globales Repository, sondern für eine abgegrenzte Domäne mit einer damit einhergehenden, überschaubaren Mengen an Nutzern und Services konzipiert ist, ermöglicht die in den folgenden Kapiteln vorgestellte Infrastruktur auch ohne solcherlei Optimierungen selbst bei mehreren Tausend indizierten Services eine Beantwortung von Suchanfragen im Millisekundenbereich (vgl. 5.2).

4.5.1.1 Invertierter Index mit MongoDB

Der Textindex folgt der aus dem Gebiet des *Text Retrieval* bekannten Struktur eines *inverted index* [54, Kap. 1-2]. Wie in 4.9 dargestellt, handelt es sich dabei im Wesentlichen um eine (sehr) lange Liste von Termen. Jeder Term verfügt zudem über eine Referenz auf jene Dokumente, in welchen er enthalten ist.
In SEnSoR handelt es sich bei indizierten Dokumenten um Services bzw. Methoden, welche dem Datenmodell aus Kapitel 4.3 entsprechen. Beim Hinzufügen eines neuen Service in das Repository wird dieses parallel dazu in den Service-Index aufgenommen. Gleichsam werden die Methoden des Service im Methoden-Index erfasst. Diese Trennung ermöglicht die separate Text-Suche nach Services bzw. Methoden, die zugrundeliegende Implementierung ist jedoch dieselbe.

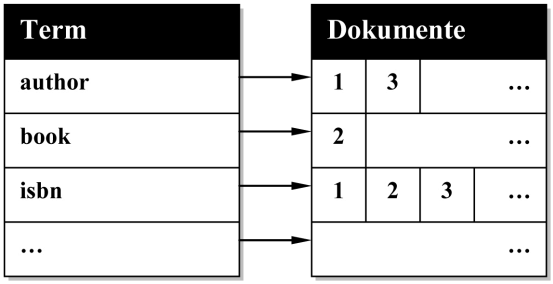

Abbildung 4.9: Schematische Darstellung eines invertierten Index.

Aus Sicht des Index handelt es sich also bei jedem Service und jeder Methode um ein eindeutig identifizierbares Dokument, welches aus einer Menge von Termen besteht. In Abbildung 4.9 ist die ID eines Service-Dokuments vereinfacht als Zahl dargestellt. Tatsächlich wird hierfür der voll-qualifizierte Name eines Service bzw. einer Methode herangezogen.[14]
Bei der Wahl, welche Terme ein Dokument beschreiben, besteht ein gewisser Spielraum. Für Services empfehlen sich dessen Name sowie dessen Dokumentation. Für Methoden gibt es darüber hinaus die Möglichkeit, auch die Namen der Input-Parameter, Input-Typen, Exceptions bis hin zu Kommentaren im Code miteinzubeziehen. In SEnSoR wird standardmäßig sämtliche genannte textuelle Information außer Kommentaren verwendet.[15] Der daraus entstandene, unstrukturierte Text (engl. *bag of words*) repräsentiert letztendlich ein Service bzw. eine Methode. Um den Index möglichst klein (und sauber) zu halten, wird jeder Term vor seiner Aufnahme bereinigt. Dies geschieht wie folgt:

1. **Trimmen:** Oft existieren in Service-Beschreibungen aus Formatierungsgründen unnötige Leerzeichen und Sonderzeichen am Ende von Wörtern wie der Punkt am Ende dieses Satzes. Diese werden vor dem Indizieren jeweils von den einzelnen Termen entfernt.

2. **Schwarze Liste:** Sogenannte Stoppwörter (Wörter, die keine für die Suche relevante Information beinhalten) wie Artikel, Konjunktionen etc. werden ebenso entfernt wie beispielsweise HTML-Markup, das zur Formatierung, etwa in JavaDoc, verwendet werden kann.

3. **Trennung von Termen:** Da in Programmiersprachen Methodennamen gewöhnlich keine Leerzeichen enthalten dürfen, ist es gängige Konvention, diese zugunsten der besseren Lesbarkeit entweder nach dem *camelCasing*-Verfahren (dt. „Binnenmajuskel“) wie in $getBookByAuthor$ oder mittels Unterstrichen wie in $get\_book\_by\_author$ zu benennen. Hauptwörter stehen dabei oft für Parameter, während Verben die Funktionalität eines Service oder einer Methode beschreiben [74]. Solcherlei Terme beinhalten damit wichtige Information für die textbasierte Suche und werden deshalb vor dem Indizieren getrennt.

[14] Genaugenommen müssen zur eindeutigen Identifikation von Methoden auch deren Input-Typen beachtet werden. Der Grund dafür sind überladene Methoden, welche denselben Namen haben können.

[15] Ein Argument für Weglassung von Code-Kommentaren ist, dass diese nicht denselben Stellenwert wie etwa Methodennamen besitzen und damit ein Suchergebnis potentiell verfälschen können.

4. **Stemming:** Stemming ist ein Verfahren aus der Linguistik, welches eine morphologische Ausprägung eines Wortes auf dessen Wortstamm zurückführt. Beispielsweise wird aus dem Genitiv eines Wortes, etwa *Buches*, der Stamm *Buch*. Mittlerweile existieren viele verschiedene Stemming-Algorithmen für unterschiedliche Sprachen. In SEnSoR kommt der klassische Algorithmus nach Martin Porter [69] für die englische Sprache zum Einsatz, welcher auf dessen Website[16] für eine Vielzahl an modernen Programmiersprachen verfügbar ist.

Abbildung 4.10 zeigt an einem kleinen Beispiel, wie Services indiziert werden. Service-Dokumente sind hierbei in *JavaScript Object Notation* (JSON) notiert. Zunächst wird ein Service-Datensatz aus der Datenbank in ein Service-Dokument mit einer id (s_1, s_2) und einer Liste der enthaltenen Terme ($terms$) transformiert. Anschließend wird über die Term-Liste jedes Dokuments iteriert. Ist ein Term noch nicht im Index vertreten, wird dieser neu angelegt. In jedem Fall wird die id des aktuellen Dokuments in der $docs$-Liste (vgl. Abbildung 4.9) gespeichert. Dies ermöglicht es nun, Dokumente anhand eines eingegebenen (Such-)Begriffes zu finden.

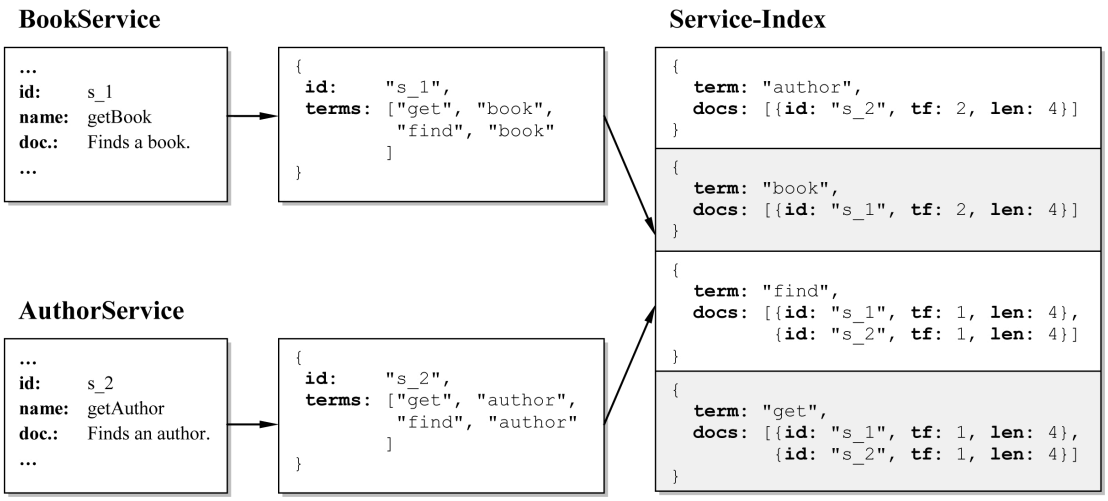

Abbildung 4.10: Aufnahme von Services in den Textindex.

Der Zusatz „invertiert“ im Begriff „invertierter Index“ rührt übrigens daher, dass der beschriebene Vorgang die Zuordnung „Dokument → Terme“ zu „Term → Dokumente“ transformiert hat. Zu einem gegebenen Term sind nun also alle Dokumente gespeichert, welche diesen beinhalten. Dadurch ist es bei einem Suchvorgang möglich, recht schnell ausschließlich jene Dokumente herauszufiltern, die einen gegebenen Suchbegriff auch tatsächlich enthalten, ohne dabei den Inhalt sämtlicher Dokumente immer wieder erneut betrachten zu müssen, wodurch die Performanz des Suchvorganges wesentlich erhöht wird.
Zusätzlich werden innerhalb der Index-Datenstruktur Kennzahlen gespeichert, welche später dem Ranking als Grundbausteine für die Berechnung der Relevanz von Dokumenten dienen:

[16] http://tartarus.org/~martin/PorterStemmer/

- tf (*term frequency*): Anzahl der Vorkommen eines Terms innerhalb eines Dokuments. Im Beispiel aus Abbildung 4.10 beträgt die tf für den Term *book* und das Dokument s_1 etwa den Wert 2, da der Term innerhalb des Dokuments zweimal vorkommt.

- len (*length*): Länge, also die Anzahl der Terme eines Dokuments. Zwar ist es nicht notwendig, diesen Wert per Tupel <Term, Dokument> zu speichern, da er für ein Dokument immer gleich ist. Allerdings ist es aus Performance-Sicht für spätere Berechnungen dennoch sinnvoll, etwas zusätzlichen Speicherplatz gegen die Notwendigkeit von Unterabfragen (*joins*) einzutauschen.

Für die Implementierung der Index-Datenstruktur gibt es prinzipiell mehrere geeignete Möglichkeiten. Für kleinere Anwendungen reicht unter Umständen bereits eine Hashmap oder eine dünnbesetzte (*sparse*) Matrix aus, die im Hauptspeicher gehalten wird.
In SEnSoR wird stattdessen die Dokument-orientierte NoSQL-Datenbank *MongoDB* verwendet, welche folgende Vorteile bietet:[17]

- **Persistenz:** In MongoDB werden Datensätze im JSON-Format exakt wie in Abbildung 4.10 gespeichert. Die Datenbank stellt alle benötigten Operationen zum Einfügen, Löschen oder Editieren eines Datensatzes bereit und speichert diese in sogenannten *collections*, vergleichbar mit Tabellen in relationalen Datenbanken, auf der Festplatte. Es ist somit möglich, je einen invertierten Index für Services und Methoden in verschiedenen Collections zu speichern. Da MongoDB im Gegensatz zu relationalen Datenbanken Schemafrei ist, können auch Datensätze unterschiedlicher Form, etwa mit unterschiedlich langen $docs$-Listen, problemlos persistiert werden.

- **Skalierbarkeit:** Ein Kern-Feature von MongoDB ist das sogenannte *sharding* (*shard*, dt. „Bruchstück“), was der horizontalen Skalierbarkeit, also der Aufteilung von Daten auf mehrere Datenbank-Instanzen entspricht. Somit kann der Index bei Bedarf in mehrere Bereiche, etwa alphabetisch für Terme von A bis E, F bis J usw. aufgeteilt werden. Dieses Feature ist weniger aus Sicht des Speicherplatzes, als aus Sicht der Datenverarbeitung interessant. Das Ranking von Dokumenten zu einer gegebenen Suchanfrage, wie im Folgekapitel vorgestellt, kann dadurch von mehreren Instanzen parallel durchgeführt werden.

Als großer Nachteil von NoSQL-Datenbanken wird oft das Fehlen von Transaktionen und einem damit verbundenen strengen Konsistenzmodell genannt [80]. Theoretisch könnte es ohne Anpassung der Applikationslogik somit passieren, dass, während gerade ein neues Service indiziert wird, eine gleichzeitig bearbeitete Suchanfrage ein teilweise inkorrektes Ergebnis liefert, da möglicherweise noch nicht alle Terme im Index aktualisiert worden sind. Zugunsten der Vorteile von MongoDB als Infrastruktur für die Textsuche wird diese mögliche Lese-Inkonsistenz toleriert, zumal diese von der Benutzerin nicht wahrnehmbar ist.

[17] http://www.mongodb.org/

4.5.1.2 Parallelisiertes Ranking mit MapReduce

MapReduce ist ein ursprünglich bei Google entwickeltes Verfahren für verteilte Berechnungen auf großen Datenmengen [20]. Mittlerweile gibt es davon verschiedene Implementierungen wie etwa *Hadoop* von Apache.[18] Unter anderem ist das Verfahren auch fester Bestandteil von MongoDB und kommt in SEnSoR für das Ranking von Dokumenten zum Einsatz.
MapReduce ist laut den Autoren von der funktionalen Programmierung inspiriert. In funktionalen Sprachen wie Haskell operieren die beiden Funktionen *map* und *reduce* (auch bekannt als *fold*) jeweils auf Listen. Die Funktionsweise lautet wie folgt:

- **map:** Die map-Funktion nimmt eine Liste sowie eine Funktion als Argument, welche auf jedes Element der Liste angewendet wird. Der Rückgabewert der Funktion ist die durch die Funktionsanwendung modifizierte Liste.
- **reduce:** Die reduce-Funktion nimmt ebenfalls eine Liste sowie eine Aggregat-Funktion als Argument. Ausgehend vom ersten Listenelement wird die Aggregat-Funktion iterativ auf alle weiteren Elemente angewendet. Der Rückgabewert hängt von der Natur der Aggregat-Funktion ab, in vielen Fällen handelt es sich jedoch dabei um einen einzigen Wert vom selben Typ wie die übergebene Liste.

Zum besseren Verständnis zeigen die folgenden beiden Aufrufe das Funktionsprinzip von map und reduce an einem kleinen Beispiel:

```
1. map    (*10) [1,2,3,4,5]       => [10,20,30,40,50]
2. reduce (+)   [10,20,30,40,50] => 150
```

In obigem Beispiel wird im ersten Aufruf die Funktion „Multiplikation mit 10“ auf eine Liste mit den Zahlen von eins bis fünf angewendet. Der zweite Aufruf summiert die Elemente der Liste auf.[19]
Nicht jede Art der Berechnung kann sinnvoll in einen MapReduce-Job transformiert werden. Es ist damit jedoch effizient möglich, die paarweise Ähnlichkeit von Dokumenten zu berechnen [19] [21] und somit ist MapReduce gut für die textbasierte Suche geeignet [33].
Übertragen auf JSON-Datensätze, aus denen der Index besteht, lauten die Signaturen der beiden Methoden wie folgt:

```
map    (k1, v1)        => list(k2, v2)
reduce (k2, list(v2)) => list(v2)
```

Die Variablen k und v stehen dabei jeweils für key und $value$ des Datensatzes. Im Falle des Index lassen sich k und v eins zu eins wie in Abbildung 4.10 dargestellt auf den $term$ und seine Dokumente $docs$ übertragen.

[18] Hadoop: http://hadoop.apache.org/

[19] Bei 1. handelt es sich um ausführbaren Code in der funktionalen Programmiersprache *Haskell*, bei 2. der Einfachheit wegen nicht. Der funktionierende Code lautet wie folgt: `foldl (+) 0 [10,20,30,40,50]`. Der Unterschied besteht darin, dass die fold-Funktion als zusätzliches Argument einen Startwert verlangt – im Falle der Addition also das neutrale Element 0.

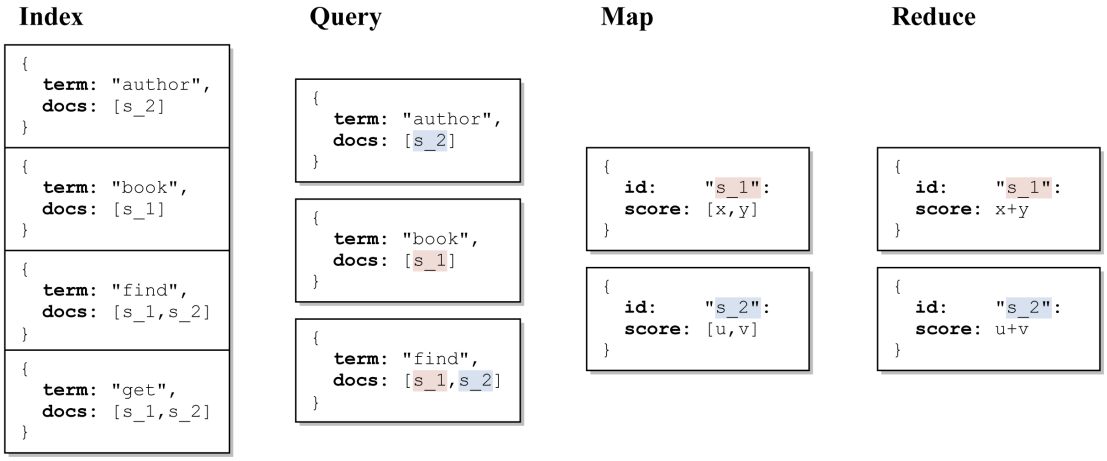

Abbildung 4.11: Die Phasen eines MapReduce-Jobs.

Abbildung 4.11 zeigt den Ablauf eines MapReduce-Jobs anhand der Suchanfrage *find author book*. Das gewünschte Endergebnis ist eine Menge an Dokumenten zusammen mit deren Gewichtung (*score*), die angibt, wie relevant ein gefundenes Dokument hinsichtlich der Suchanfrage ist. Dies geschieht in drei Phasen wie folgt:

1. **Query:** Zunächst wird über die „Query"-Phase eine Vorselektion bezüglich der relevanten Dokumente getroffen. In obigem Beispiel bleiben nur jene Dokumente übrig, welche einen oder mehrere Terme der Suchanfrage beinhalten. Anders als in relationalen Datenbanken, wo dies über eine $SELECT$-Abfrage geschehen würde, passiert dies in MongoDB nach „*query by example*". Man übergibt der Datenbank ein JSON-Objekt und setzt Werte für jene Attribute, nach denen gefiltert werden soll – in obigem Beispiel nach dem Attribut *term* für die Suchbegriffe *find*, *author* und *book*.

2. **Map:** Nun wird die *map*-Funktion für jeden übrig gebliebenen Datensatz aus der Query-Phase aufgerufen. *Map* hat die Möglichkeit, über die vordefinierte Funktion *emit(key, value)* neue Datensätze unterschiedlicher Form in einer temporären Collection zu generieren. Dies ermöglicht es, über die Dokumente $docs$ eines jeden Terms zu iterieren und einen *score* für jedes Dokument zu errechnen. Im Beispiel wurden etwa für das Service s_1 die fiktiven Werte x und y für die beiden Terme *book* und *find* errechnet. Der Output der Map-Phase ist somit eine Menge von Dokumenten mit einer Liste von *scores* für jeden ihrer Terme.

3. **Reduce:** Anschließend wird die *reduce*-Funktion auf die in der Map-Phase generierten Datensätze angewendet. Die Aufgabe der Funktion ist es, für jedes Dokument einen Gesamt-Score aus den Teil-Scores zu berechnen. Für das Beispiel geschieht dies auf Basis einer simplen Metrik, welche die Scores einfach aufsummiert. Wichtig ist es zu verstehen, dass nicht garantiert ist, wie oft das zugrundeliegende Framework (MongoDB) *reduce* für ein und denselben Term aufruft. Insbesondere kann dies mehr als einmal geschehen. Aus

diesem Grund muss die übergebene *reduce*-Funktion idempotent sein – das heißt, es muss gelten:

$$reduce(k, [reduce(k, v)]) == reduce(k, v) \quad \forall k, v.$$

In SEnSoR sind die beiden in den nachfolgenden Abschnitten vorgestellten Ranking-Modelle als MapReduce-Funktionen implementiert. Über eine Konfigurationsseite ist es zudem möglich, weitere Modelle durch die Bereitstellung eigener Funktionen für *map reduce* selbst hinzuzufügen. Da diese im Falle von MongoDB in JavaScript zu implementieren sind, können sie ohne Neustart oder gar Neu-Kompilation des Systems umgehend verwendet werden.

4.5.1.3 MapReduce: Vector Space Model

Im Vector Space Model (VSM) wird jedes Dokument als Vektor, eingebettet in einen hochdimensionalen Euklidschen Vektorraum, aufgefasst. Jede Dimension eines solchen Dokument-Vektors steht dabei für einen enthaltenen Term [54, Kap. 6].
Das *BookService* aus Abbildung 4.10 und die Suchanfrage *get book* lassen sich etwa im dreidimensionalen Raum wie in Abbildung 4.12 visualisieren.

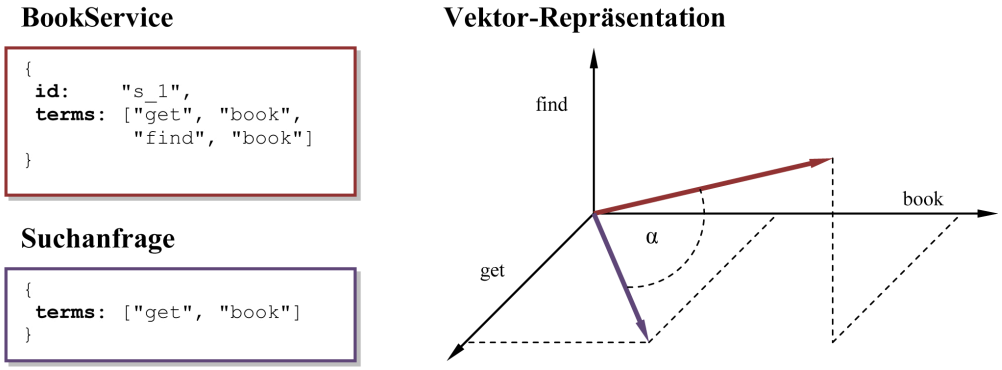

Abbildung 4.12: Dokumente und deren Repräsentation im Vektorraum.

In obigem Beispiel besteht der gesamte Index zur leichteren Veranschaulichung aus lediglich drei Termen – *get*, *book* und *find*. Jeder dieser drei Terme steht wiederum für eine Achse, auf welchen sich die Dokument-Vektoren gemäß ihren entsprechenden *term frequencies* auftragen lassen. Beispielsweise verfügt der rote Vektor – welcher das *BookService* repräsentiert – über zwei Einheiten auf der *book*-Achse, da dieser Term genau zweimal im Dokument enthalten ist und über jeweils eine Einheit auf den beiden Achsen für *get* und *book*.
Wie man erkennen kann, wird auch eine Suchanfrage als Dokument im selben Vektorraum aufgefasst. Im Beispiel bildet die Suchanfrage *get book* einen Vektor mit je einer Einheit auf den beiden entsprechenden Achsen. Der Begriff der Ähnlichkeit zweier Dokumente ergibt sich geometrisch. Eine weit verbreitete Methode, die auch in SEnSoR implementiert ist, ist die Berechnung des eingeschlossenen Winkels α zweier Vektoren – je kleiner dieser ausfällt, umso ähnlicher sind sich die Dokumente.

$$\text{score}_{cos} = cos(\alpha) = cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| \cdot |\vec{d}|} = \frac{\sum_{i=0}^{|V|} q_i d_i}{\sqrt{\sum_{i=0}^{|V|} q_i^2} \sqrt{\sum_{i=0}^{|V|} d_i^2}} \tag{4.1}$$

Besagte Metrik ist als *cosine similarity* bekannt und wird gemäß Formel 4.1 berechnet. Über das Skalarprodukt zweier Vektoren lässt sich der Kosinus von α errechnen, welcher Werte zwischen 0 (keinerlei Übereinstimmung) und 1 (Deckungsgleichheit) annehmen kann. Die Länge der Dokument-Vektoren entspricht der Mächtigkeit des Vektorraums $|V|$, was mit der Anzahl der Terme im Index gleichzusetzen ist.
Die Werte für d_i und q_i stehen für die Gewichtung der einzelnen Dimensionen – also Terme – des Dokument-Vektors d und des Query-Vektors q. Eine gebräuchliche Form der Term-Gewichtung ist das sogenannte tf-idf-Maß und in Formel 4.2 dargestellt.

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \cdot \text{idf}_t \tag{4.2}$$

Für die Gewichtung werden zwei Faktoren herangezogen. Der erste Faktor bezeichnet die *term frequency* $\text{tf}_{t,d}$, also die Häufigkeit des Vorkommens eines Terms t innerhalb eines Dokuments d. Dieser Wert wird wie in 4.5.1.1 beschrieben direkt im Index gespeichert. Die grundlegende Idee dahinter ist, dass ein Dokument, welches einen bestimmten Begriff etwa zehnmal enthält, relevanter in Bezug auf diesen Term ist als ein Dokument, das den Begriff nur ein einziges Mal beinhaltet. Da es jedoch fraglich ist, ob ersteres Dokument wirklich zehnmal so relevant ist, kann der Effekt eines hohen $\text{tf}_{t,d}$-Wertes wie in Formel 4.3 abgeschwächt werden.

$$\widehat{\text{tf}}_{t,d} = \log(1 + \text{tf}_{t,d}) \tag{4.3}$$

Der zweite Faktor steht für die *inverse document frequency* idf_t und wird gemäß Formel 4.4 berechnet. Er stützt sich auf die *document frequency* df_t eines Terms, welche für jeden Term angibt, in wie vielen Dokumenten dieser vorkommt. Der Wert für df_t ist implizit im Index gespeichert und ergibt sich aus der Länge der $docs$-Liste eines jeden Terms. Die idf dient dazu, seltene Terme, die nur in wenigen Dokumenten vorkommen, zu priorisieren. Sind zum Beispiel $N = 1000$ Dokumente indiziert und ein Term kommt nur in einem einzigen Dokument vor, so beläuft sich dessen idf_t-Wert auf 3 – seltene Terme erhalten also einen „Bonus“. Im Gegensatz dazu erhält ein Term, der in allen Dokumenten vorkommt (beispielsweise ein Artikel), den Wert 0 und trägt somit gar nichts zum Gesamtergebnis bei.

$$\text{idf}_t = \log \frac{N}{\text{df}_t} \tag{4.4}$$

Wie man aus obigen Formeln erkennen kann, hängt die Berechnung der Termgewichtung auch von der Anzahl der indizierten Dokumente N ab. Das bedeutet, dass sich diese Werte beim Hinzufügen eines neuen Service (geringfügig) ändern. Da es jedoch bei einem großen Index äußerst aufwändig ist, die tf-idf jedes Mal im Rahmen einer Suchanfrage für alle Dokumente erneut zu berechnen, macht es aus Sicht der Performance Sinn, diese Werte ebenfalls im Index zu speichern und in regelmäßigen Abständen zu aktualisieren, etwa ab einer bestimmten Anzahl neu hinzugekommener Services.

```
 1. function mapCosine() {
 2.   var term = this._id;
 3.   var docs = this.docs;
 4.   for (i = 0; i < docs.length; i++) {
 5.     var doc = docs[i];
 6.     var tf = doc.termFrequency;
 7.     var df = docs.length;
 8.     var d_i = tfIdf(tf, df);
 9.     var q_i = isQueryTerm(term) ? tfIdf(1, df) : 0;
10.     var docVal = new Object();
11.     docVal.dividend = q_i * d_i;
12.     docVal.quadraticDocWeight = d_i * d_i;
13.     emit(doc.id, docVal);
14.   }
15. }
16. function tfIdf(tf, df) {
17.   return Math.log(1 + tf) *
            Math.log(NUMBER_OF_DOCUMENTS / df);
18. }
19. function isQueryTerm(term) {
20.   for (var i = 0; i < QUERY_TERMS.length; i++) {
21.     if (QUERY_TERMS[i] === term) {
22.       return true;
23.     }
24.   }
25.   return false;
26. }
```

Abbildung 4.13: Die *map*-Funktion für das Ranking im VSM.

Abbildung 4.13 zeigt die JavaScript-Implementierung der *map*-Funktion, wie sie in SEnSoR genutzt werden kann. Die Berechnung richtet sich eng an Formel 4.1 mit der kleinen Optimierung, dass die Länge des Query-Vektors $|\vec{d}|$ aus dem Divisor gestrichen wird, da dieser Wert für jedes Dokument gleich ist und somit am Ranking der Dokumente untereinander nichts ändert.

Ausgangspunkt sind die gespeicherten JSON-Datensätze aus dem Index (*term*, *docs*). Diese werden von MongoDB automatisch an *map* übergeben und sind daher implizit aus *map* heraus zugänglich (Zeilen 2-3).

Dividend und Divisor werden für jedes Dokument jeweils gesondert berechnet (Zeilen 8-12), im Objekt $docVal$ gekapselt und an die Reduce-Phase weitergereicht (Zeile 13). Die Funktion *isQueryTerm()* überprüft dabei, ob ein Term Teil der Suchanfrage ist. Das funktioniert deshalb, da einem MapReduce-Job im Vorfeld Konstanten mitgegeben werden können, wie ein Array mit den Termen der Suchanfrage *QUERY_TERMS* oder der Gesamtanzahl aller indizierten Dokumente *NUMBER_OF_DOCUMENTS*, welche über globale Sichtbarkeit verfügen.

In der Reduce-Phase (Abbildung 4.14) werden nun für jedes Dokument die Werte für Dividend und Divisor gemäß Formel 4.1 aufsummiert (Zeilen 5-8). Nun ist es am Ende nur noch nötig, den Bruch auszurechnen. Zu diesem Zweck bietet MongoDB die Möglichkeit, zusätzlich eine *finalize*-Funktion (Abbildung 4.15) zu übergeben, die für jedes Ergebnis-Objekt einmal angewendet wird und für ebensolche „Aufräumarbeiten“ am Ende eines MapReduce-Jobs genutzt

```
1. function reduceCosine(key, values) {
2.   var result = new Object();
3.   result.dividend = 0;
4.   result.quadraticDocWeight = 0;
5.   for (i = 0; i < values.length; i++) {
6.     var docVal = values[i];
7.     result.dividend += docVal.dividend;
8.     result.quadraticDocWeight += docVal.quadraticDocWeight;
9.   }
10.  return result;
11. }
```

Abbildung 4.14: Die *reduce*-Funktion für das Ranking im VSM.

```
1. function finalizeCosine(key, reducedValue) {
2.   var docScore = reducedValue.dividend /
                    Math.sqrt(reducedValue.quadraticDocWeight);
3.   return isNaN(docScore) ? 0 : docScore;
4. }
```

Abbildung 4.15: Die *finalize*-Funktion für das Ranking im VSM.

werden kann.
Am Ende erhält man nun für jedes indizierte Dokument ein Tupel *(docId, docScore)*. Absteigend nach $docScore$ sortiert bilden die am höchsten bewerteten Dokumente (die Zahl der gewünschten Ergebnisdokumente ist konfigurierbar) das Ergebnis der Suchanfrage.

4.5.1.4 MapReduce: Okapi BM25

Anders als beim VSM handelt es sich bei Okapi BM25 nicht um ein geometrisch begründetes, sondern um ein probabilistisches Modell. Es ist rund zwei Jahrzehnte jünger als das VSM und wurde ursprünglich in [72] vorgestellt. Die eigentliche Ranking-Funktion wird als BM25 bezeichnet, Okapi heißt das System, in welchem diese erstmals implementiert worden ist. Untersuchungen von Spärck Jones et al. im Rahmen der TREC-Konferenzen[20] zeigen, dass das Modell für unterschiedliche Suchaufgaben gute Ergebnisse liefert [79], weshalb es in heutigen Anwendungen weit verbreitet ist.
Ein wesentlicher Unterschied von BM25 im Vergleich zum VSM ist, wie in Formel 4.5 ersichtlich, dass nur Terme der Suchanfrage q in der Berechnung vorkommen und nicht alle Terme aus dem Index, was einen großen (positiven) Einfluss auf die Performance hat. Die beiden Konstanten L und L_{avg} bezeichnen die Länge eines Dokuments sowie die durchschnittliche Länge aller indizierten Dokumente.

[20] Die Text REtrieval Conference (TREC, http://trec.nist.gov/) widmet sich jedes Jahr verschiedenen Forschungsgebieten im Umfeld des *Information Retrieval* und bietet die Infrastruktur für die Evaluierung neuartiger (Such-) Algorithmen.

$$\text{score}_{bm25}(q,d) = \sum_{t \in q} \log \frac{N}{\text{df}_t} = \frac{(k_1+1)\text{tf}_{t,d}}{k_1((1-b)+b\cdot(L_d/L_{avg}))+\text{tf}_{t,d}} \tag{4.5}$$

Ähnlichkeiten zur tf-idf-Gewichtung sind jedoch unverkennbar. So entspricht gleich der erste Faktor der idf (vgl. Formel 4.4), allerdings in einem Modell basierend auf Wahrscheinlichkeiten verankert [71]. Kommt beispielsweise der Term t_i in $\text{df}_t = n$ aus insgesamt N verschiedenen Dokumenten vor, so ist die Wahrscheinlichkeit, dass ein zufällig gewähltes Dokument d diesen Term beinhaltet:

$$P(t \in d) = \frac{\text{df}_t}{N} \tag{4.6}$$

Die idf steht mit der Wahrscheinlichkeit $P(t \in d)$ wie folgt in direktem Zusammenhang (zur Erinnerung, es gilt: $-\log x = \log(1/x)$):

$$\text{idf}_t = \log \frac{N}{\text{df}_t} = -\log \frac{\text{df}_t}{N} = -\log P(t \in d) \tag{4.7}$$

Das Konzept der Aufsummierung der idf-Werte für jeden einzeln Term der Suchanfrage lässt sich ebenfalls über Wahrscheinlichkeitsrechnung herleiten. Unter der vereinfachenden Annahme, dass zwei Terme t_1 und t_2 jeweils statistisch unabhängig voneinander in einem Dokument auftreten, gilt nämlich:

$$\text{idf}_{t_1 \wedge t_2} = -\log P(t_1 \wedge t_2) = -\log(P(t_1)P(t_2)) = -(\log P(t_1) + \log P(t_2)) = \text{idf}_{t_1} + \text{idf}_{t_2} \tag{4.8}$$

Der zweite Faktor der Formel 4.5 basiert wie in der tf-idf-Gewichtung ebenfalls auf der *term frequency* $\text{tf}_{t,d}$, allerdings kann diese über die beiden Parameter b und k_1 gesteuert werden. Dabei gibt b im Wertebereich von $[0,1]$ an, wie stark sich die Länge des Dokumentes L_d auf die Term-Gewichtung auswirkt, während der positive Parameter k_1 die Auftrittshäufigkeit des Terms $\text{tf}_{t,d}$ gewichtet. Bei $k_1 = 0$ evaluiert der gesamte zweite Faktor zu 1 (d.h. der Wert für $\text{tf}_{t,d}$ ist irrelevant) und je höher k_1, desto stärker die Gewichtung.
Die Wahl der beiden Parameter b und k_1 bietet einen gewissen Freiheitsgrad bei der Optimierung der Ranking-Funktion und hängt teilweise von der Art der indizierten Dokumente ab. Experimente haben gezeigt, dass die Wertebereiche $0.5 < b < 0.8$ und $1.2 < k_1 < 2$ vernünftige Ergebnisse liefern [73]. In SEnSoR sind beide Parameter über eine Konfigurationsdatei anpassbar.
Abbildung 4.16 zeigt die Implementierung der *map*-Funktion für BM25, welche auf Formel 4.5 basiert. Da die Term-Gewichtung anders als beim Kosinus-Pendant im Vector Space Model für jeden Term direkt erfolgen kann (Zeile 8), reicht es aus in der Map-Phase einfach das Tupel *(docId, docScore)* zu generieren (Zeile 9). Sämtliche Konstanten (in Großbuchstaben) werden vor der Ausführung des MapReduce-Jobs gesetzt und sind global verwendbar.
In der Reduce-Phase (Abbildung 4.17) werden die einzelnen *scores* pro Dokument einfach aufsummiert und repräsentieren direkt das Ranking-Ergebnis. Eine *finalize*-Funktion ist daher nicht nötig.

```
 1. function mapBM25() {
 2.   var docs = this.docs;
 3.   for (i = 0; i < docs.length; i++) {
 4.     var doc = docs[i];
 5.     var len = doc.length;
 6.     var tf = doc.termFrequency;
 7.     var df = docs.length;
 8.     var score = scoreBM25(len, tf, df);
 9.     emit(doc.id, docScore);
10.   }
11. }
12. function scoreBM25(len, tf, df) {
13.   var dividend = (BM25_K1 + 1) * tf;
14.   var divisor = BM25_K1 * ((1 - BM25_B) + BM25_B *
                    (len / AVG_DOCUMENT_LENGTH)) + tf;
15.   var factor1 = Math.log(NUMBER_OF_DOCUMENTS / df);
16.   var factor2 = dividend / divisor;
17.   return factor1 * factor2;
18. }
```

Abbildung 4.16: Die *map*-Funktion für BM25.

```
1. function reduceBM25(key, values) {
2.   var docScore = 0;
3.   for (i = 0; i < values.length; i++) {
4.     docScore += values[i];
5.   }
6.   return docScore;
7. }
```

Abbildung 4.17: Die *reduce*-Funktion für BM25.

Mittlerweile gibt es verschiedenste Erweiterungen von BM25. Die Variante BM25L [53] etwa verhindert, dass sehr lange Dokumente beim Ranking benachteiligt werden. Im Falle von Service-Dokumenten ist das ein eher geringes Problem, da deren Länge hauptsächlich von der Dokumentation (sofern vorhanden) abhängt. Zum Vergleich: Die durchschnittliche Anzahl an Termen sämtlicher *public*-Methoden der ausführlich dokumentierten Java 7 Standard-Bibliothek beträgt nach dem Indizieren in etwa 75.

4.5.2 Semantische Suche

Anders als bei der textbasierten Suche, welche Service-Dokumente als reinen (uninterpretierten) Text betrachtet, bezieht die semantische Suche auch *Inputs*, *Outputs*, *Preconditions* & *Effects* (IOPE) von Methoden bei der Ermittlung des Übereinstimmungsgrades zu einer gegebenen Suchanfrage mit ein.
Das Ziel ist, nicht nur bloße Empfehlungen für relevante Methoden auf Basis einer Text-Metrik, sondern tatsächlich aufrufkompatible Methoden zurückzuliefern. Im Gegensatz zum Ranking

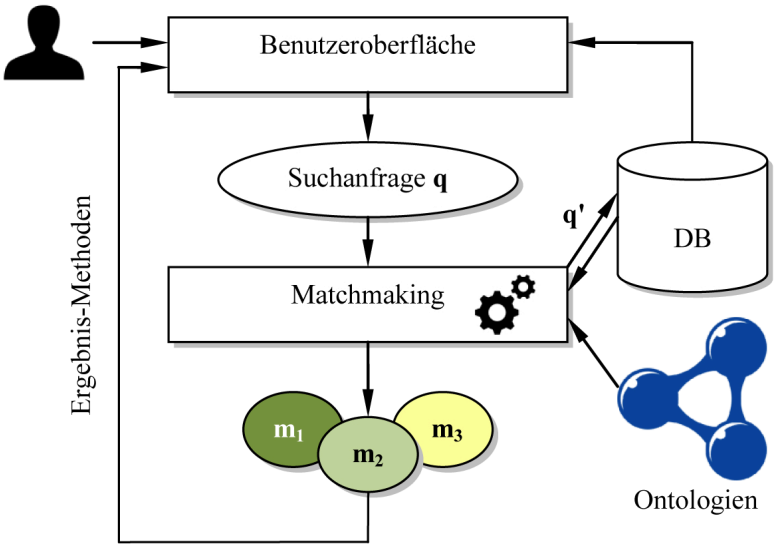

Abbildung 4.18: Der Suchprozess einer semantischen Suche.

der Textsuche bezeichnet man diesen Vorgang als *Matchmaking*.

Für die Nutzerin gestaltet sich der Suchprozess gemäß Abbildung 4.18 nach außen hin im Wesentlichen unverändert. Der Unterschied besteht lediglich darin, die Suchparameter genauer zu spezifizieren, um zwischen den vier genannten Konzepten (IOPE) zu unterscheiden.

In SEnSoR gibt es analog dafür die vier Präfixe *i*=, *o*=, *p*= und *e*=, sodass die Suchanfrage *o*=*Book* etwa Methoden findet, die ein Buch zurückliefern. *Book* referenziert dabei eine bestimmte Klasse einer bestehenden Ontologie und muss im Falle von gleichnamigen Klassen aus unterschiedlichen Ontologien voll qualifiziert als URI angegeben werden.

Intern unterscheidet sich der Matchmaking-Vorgang deutlich von der textbasierten Suche. Typischerweise arbeiten Matchmaker für Semantic Web Services sequenziell. Trifft eine Suchanfrage q ein, um beispielsweise Methoden zu finden, die einen bestimmten Output zurückliefern, so wird für jedes indizierte Service mit Hilfe eines Reasoners überprüft, ob es eine kompatible Methode bereitstellt.

Dieser Ansatz ist aus Sicht des Antwortzeitverhaltens für eine große Anzahl an Services bzw. Methoden suboptimal, da sich die Zeit für das Reasoning dadurch aufsummiert. In SEnSoR werden deshalb alle Ontologien, die beim Parsen von Services referenziert wurden, über die gesamte Laufzeit hinweg im Hauptspeicher gehalten. Der intern verwendete Reasoner Pellet wird dabei nur jedes Mal neu angestoßen, wenn sich etwas am aktuellen Stand geändert hat (etwa wenn eine neue Ontologie hinzugekommen ist).

Statt eine Suchanfrage q nun einzeln gegen jede indizierte Methode zu „matchen“, berechnet der Matchmaker in SEnSoR mit Hilfe von Pellet einmalig ein (potentiell großes) Set aller möglichen Kandidatenlösungen, transformiert dieses Set in mehrere SQL-Abfragen q' und lässt vom Datenbankmanagementsystem ausschließlich jene Methoden zurückliefern, die einer solchen Kandidatenlösung entsprechen.

```
<owl:Class rdf:about="#Person"/>

<owl:Class rdf:about="#Author">
  <rdfs:subClassOf rdf:resource="#Person"/>
</owl:Class>

<owl:Class rdf:about="#SciFiAuthor">
  <rdfs:subClassOf rdf:resource="#Author"/>
</owl:Class>


<owl:Class rdf:about="#Book"/>

<owl:Class rdf:about="#Novel">
  <rdfs:subClassOf rdf:resource="#Book"/>
</owl:Class>

<owl:Class rdf:about="#SciFiNovel">
  <rdfs:subClassOf rdf:resource="#Novel"/>
</owl:Class>
```

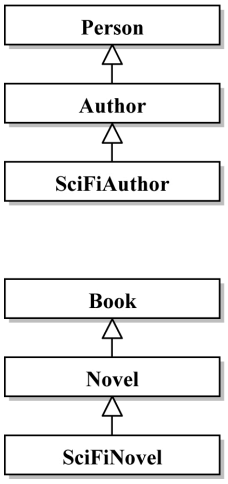

Abbildung 4.19: Beispiel-Klassenhierarchie in OWL/RDFS und UML-Notation.

4.5.2.1 Logikbasiertes Input/Output-Matching

Benötigt man zum Beispiel eine Methode, welche zu einem bestimmten Roman (engl. *novel*) dessen Autor zurückliefert und klammert zunächst eventuelle Vor- oder Nachbedingungen aus, so könnte eine entsprechende Suchanfrage etwa *„i=Novel o=Author“* lauten. Die beiden zugehörigen Konzepte *Novel* sowie *Author* verweisen dabei jeweils auf Klassen einer entsprechenden Ontologie, die in Abbildung 4.19 als Bestandteil einer einfachen OWL-Klassenhierarchie dargestellt sind.
Existiert beispielsweise eine Methode m_1, die exakt die beiden Konzepte der Suchanfrage (*Novel* und *Author*) abdeckt, so ist diese offensichtlich relevant und verfügt darüber hinaus auch über den maximalen Grad an Übereinstimmung zur Suchanfrage – allerdings wäre besagte Methode aufgrund der Namensgleichheit auch von der Textsuche gefunden worden.
Der Vorteil der semantischen Annotation von Methoden zeigt sich erst darin, dass die vorhandenen logischen Beziehungen von Konzepten innerhalb der Ontologie ausgenutzt werden. Befindet sich etwa eine weitere Methode m_2 im Repository, die ebenfalls einen Autor zurückliefert, aber statt eines Romans das weniger spezifische Konzept eines Buches (*Book*) erwartet, so ist diese Methode ebenfalls kompatibel zur Suchanfrage. In der Beispiel-Ontologie ist explizit festgehalten, dass das Konzept *Novel* spezifischer ist als das Konzept *Book* – das allgemeinere Konzept *Book subsumiert* die jeweils spezifischeren Konzepte *Novel* und *SciFiNovel* weiter unten in der Klassenhierarchie.
Hinsichtlich der Kompatibilität zweier Methoden auf Basis ihrer Input- und Output-Parameter ist dabei jeweils die Richtung der Subsumption-Relation $\sqsupseteq$ relevant [36]:

- **Input-Parameter** verhalten sich *kontravariant*. Eine Methode m_1 ist zu einer weiteren Methode m_2 Input-kompatibel, wenn m_1 – wie oben beschrieben – entweder dasselbe oder ein *allgemeineres* Konzept als Input verlangt.

- **Output-Parameter** hingegen verhalten sich *kovariant*. Damit eine Methode m_1 Output-kompatibel zu einer weiteren Methode m_2 ist, muss m_1 entweder dasselbe oder ein *spezifischeres* Konzept als Output zurückliefern. Im laufenden Beispiel käme deshalb auch eine Methode infrage, die statt des gewünschten *Authors* die spezifischere Ausprägung eines *SciFiAuthors* zurückliefert, da es sich dabei ebenfalls um einen *Author* handelt.

Ausgehend vom Konzept der Subsumption ergeben sich formal vier verschiedene Grade der semantischen Übereinstimmung (engl. *degree of match*, DoM) für logikbasierte I/O–Matchmaker [42] [65] [82]:

- **EXACT:** Exakte Übereinstimmung zwischen einer Methode M und einer Suchanfrage Q liegt genau dann vor, wenn gilt:

$$\begin{gathered}\forall\, m_i \in Input(M)\ \exists\, q_i \in Input(Q) : m_i \equiv q_i \\ \wedge \\ \forall\, q_o \in Output(Q)\ \exists\, m_o \in Output(M) : m_o \equiv q_o\end{gathered}$$

  Die Konzepte der I/O-Signatur einer Methode M sind logisch äquivalent zu jenen der Suchanfrage Q. Zusätzlich darf M nicht mehr Input-Parameter verlangen als Q, wohl aber zusätzliche (nicht benötigte) Output-Parameter zurückliefern.

- **PLUGIN:** Dieser Übereinstimmungsgrad kommt ursprünglich aus dem Software Engineering [85] und beschreibt die Kompatibilität zwischen Methoden von Software-Komponenten bzw. deren I/O-Signaturen anhand von kontravarianten Eingangs- sowie kovarianten Ausgangsparametern wie folgt:

$$\begin{gathered}\forall\, m_i \in Input(M)\ \exists\, q_i \in Input(Q) : m_i \sqsupseteq q_i \\ \wedge \\ \forall\, q_o \in Output(Q)\ \exists\, m_o \in Output(M) : m_o \in LSC(q_o)\end{gathered}$$

  $LSC(x)$ steht dabei für *least specific concepts* und beschreibt die Menge der direkten Subklassen von x innerhalb einer Ontologie. Diese zusätzliche Einschränkung auf den Output von M ist in der originalen Definition nicht vorhanden, soll jedoch sicherstellen, dass sich der erwartete Rückgabewert nicht zu weit vom spezifizierten entfernt, was sonst bei sehr tiefen Klassenhierarchien leicht auftreten könnte.

- **SUBSUMES:** Ein SUBSUMES-Match lockert die oben beschriebene Einschränkung von PLUGIN und erlaubt Output-Subsumption entlang der ganzen Klassenhierarchie.

$$\begin{gathered}\forall\, m_i \in Input(M)\ \exists\, q_i \in Input(Q) : m_i \sqsupseteq q_i \\ \wedge \\ \forall\, q_o \in Output(Q)\ \exists\, m_o \in Output(M) : m_o \sqsubseteq q_o\end{gathered}$$

Dieser Übereinstimmungsgrad ist folglich schwächer als PLUGIN, garantiert jedoch weiterhin Aufrufkompatibilität.

- **SUBSUMED–BY:** Der schwächste Grad der logikbasierten I/O-Übereinstimmung erlaubt auch Methoden, die einen geringfügig allgemeineren Output zurückliefern. In diesem Fall wird die Richtung der Subsumption bei Outputs umgedreht:

$$\begin{gathered}\forall\, m_i \in Input(M)\; \exists\, q_i \in Input(Q) : m_i \sqsupseteq q_i \\ \wedge \\ \forall\, q_o \in Output(Q)\; \exists\, m_o \in Output(M) : m_o \in LGC(q_o)\end{gathered}$$

  Die Funktion $LGC(x)$ steht dabei für *least generic concept* und liefert die Menge der direkten Superklassen von x. Anders als die drei zuvor genannten Übereinstimmungsgrade verletzt dieser die unmittelbare Aufrufkompatibilität, da nun auch kontravariante Ausgangsparameter erlaubt sind. Seine Daseinsberechtigung gründet sich jedoch auf der Annahme, dass ein etwas allgemeinerer Rückgabetyp für einen Aufrufer ebenfalls noch von Nutzen sein kann.

Die Hierarchie der oben angeführten logikbasierten Übereinstimmungsgrade ist absteigend und zugunsten der besseren Veranschaulichung in Abbildung 4.20 jeweils an einem Beispiel illustriert.
Je nach Übereinstimmungsgrad liefert der Reasoner unterschiedliche Mengen für passende I/O-Konzepte. In Pellet gibt es zu diesem Zweck eine einfache API, um (direkte) Super- bzw. Subklassen zu einer gegebenen Klasse zu finden. Bei einem EXACT-Match beinhaltet sowohl die Input- als auch die Output-Menge klarerweise jeweils nur ein einziges Konzept. Für den SUBSUMES-Match aus Abbildung 4.20 hingegen lautet die entsprechende Input-Menge *{Novel, Book}* und die Menge an möglichen Outputs *{Author, Person}*.

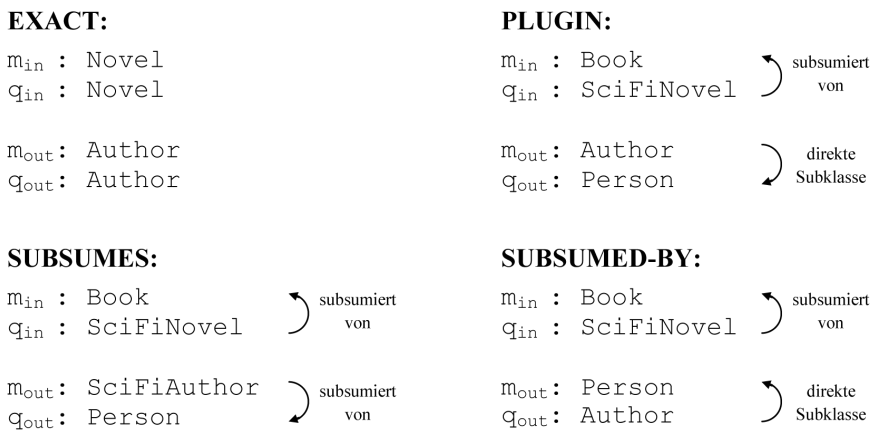

Abbildung 4.20: Grade der logikbasierten I/O-Übereinstimmung.

In SEnSoR werden nun, beginnend bei EXACT, für jeden der vier Übereinstimmungsgrade besagte Mengen in eine entsprechende SQL-Abfrage transformiert. Für SUBSUMES lautet diese wie folgt:[21]

```
SELECT m
FROM Method AS m WHERE EXISTS (
  FROM m.inputParameters AS i
  WHERE i.modelReference IN ('Novel', 'Book'))
AND WHERE EXISTS (
  FROM m.outputParameters AS o
  WHERE o.modelReference IN ('Author', 'Person'))
```

Dieser Prozess wird entlang der degree-of-match-Hierarchie so lange wiederholt, bis die konfigurierte Anzahl an Suchergebnissen gefunden wurde oder keine weiteren Ergebnisse vorliegen.

4.5.2.2 Logikbasiertes Precondition/Effect-Matching

Zusätzlich zu Input- bzw. Output-Konzepten lassen sich Methoden auch anhand ihrer Vor- und Nachbedingungen suchen. Dieser Vorgang gestaltet sich dabei ähnlich zum I/O-Matching, ist jedoch etwas aufwändiger.
Möchte man beispielsweise die Suche nach Methoden auf solche einschränken, die eine gültige Kreditkarte – etwa eine Visa-Karte – voraussetzen, könnte eine entsprechende Precondition wie folgt ausgedrückt werden:

$$p_1\text{: } \textit{hasCreditCard(Customer, VisaCard)}$$

In diesem Fall handelt es sich bei *hasCreditCard* um ein OWL *ObjectProperty*, also ein Prädikat mit den beiden Argumenten *Customer* und *VisaCard*, welche wiederum für OWL-Klassen stehen.
Im Idealfall erfüllt eine Methode exakt die geforderte Vor- oder Nachbedingung, das heißt, sowohl das Prädikat selbst als auch dessen Argumente stimmen exakt überein. Wie beim I/O-Matching gibt es jedoch Abstufungen im Grad der Übereinstimmung. Betrachtet man zum Beispiel folgende Vorbedingung unter der Annahme, dass innerhalb der Ontologie die Klasse *Customer* von der Klasse *Person* und *VisaCard* von *CreditCard* subsumiert wird, so erkennt man, dass diese eine verallgemeinerte Version von p_1 darstellt:

$$p_2\text{: } \textit{hasCreditCard(Person, CreditCard)}$$

Eine Methode mit der Vorbedingung p_2 kann somit p_1 erfüllen, da es sich bei den Argumenten *Person* und *CreditCard* jeweils um allgemeinere Konzepte handelt. Generell gelten die folgenden Beziehungen für die Kompatibilität von Vor- bzw. Nachbedingungen [2] [3]:

[21] SEnSoR nutzt die *Java Persistence API* (JPA) für das objektrelationale Mapping. Die Abfrage ist deshalb nicht in Standard-SQL-Syntax, sondern in der JPA-eigenen *Java Persistence Query Language* (JPQL) gehalten, die unter anderem auch Punkt-Notation an Stelle von JOIN erlaubt.

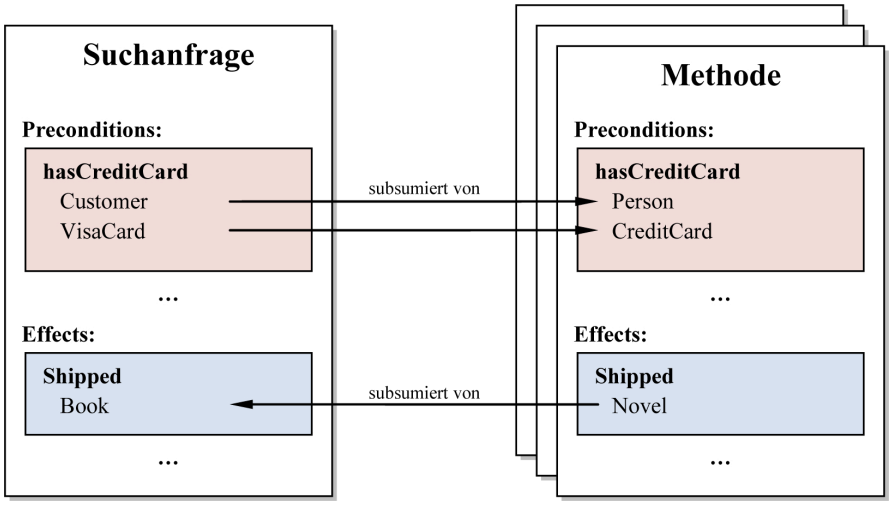

Abbildung 4.21: Separates Precondition/Effect-Matching für jede vorselektierte Methode.

- **Vorbedingungen:** Die Vorbedingung P_q einer Suchanfrage ist zur Vorbedingung P_m kompatibel, wenn erstere letztere impliziert, also gilt: $P_q \Rightarrow P_m$. Die Menge an Variablenbelegungen, für die P_q zu wahr auswertet, ist entweder gleich oder eine Teilmenge derer P_m's.

- **Nachbedingungen:** Die Nachbedingung E_q einer Suchanfrage ist zur Nachbedingung E_m kompatibel, wenn erstere von letzterer impliziert wird, also gilt $E_m \Rightarrow E_q$. Die Menge an Variablenbelegungen, für die P_q zu wahr auswertet, ist entweder gleich oder eine Obermenge derer P_m's.

Das Matching von Vor- und Nachbedingungen in SEnSoR erfolgt dementsprechend wie folgt:

1. Zunächst werden über eine SQL-Abfrage jene Methoden vorselektiert, welche über die entsprechenden Prädikate aus der Suchanfrage verfügen.

2. Anschließend wird über sämtliche Vor- und Nachbedingungen der Suchanfrage iteriert und jeweils für jede Methode geprüft, ob diese über eine kompatible Vor- bzw. Nachbedingung verfügt. Für OWL-Klassen und ObjectProperties wird dabei jeweils die Richtung der Subsumption kontrolliert, wie in Abbildung 4.21 illustriert.

   Für DataProperties werden simple Datentypen wie beispielsweise Zahlen schlicht auf Gleichheit überprüft, etwa in *hasValue(Year, 2014)*. Weiters unterstützt SEnSoR die in SWRL vorhandenen Vergleichsoperatoren („Built-Ins“) *lessThan*, *lessThanOrEqual*, *greaterThan* sowie *greaterThanOrEqual*. Dadurch können auch Zahlenbereiche für Input- bzw. Output-Parameter von Methoden über Vor- und Nachbedingungen spezifiziert werden. Beispielsweise könnte die Vorbedingung *greaterThan(Year, 1999)* einer Methode ausdrücken, dass diese eine Jahreszahl ab 2000 als Input erwartet.

3. Anders als beim I/O-Matching gibt es in SEnSoR keine abgestuften Grade der Übereinstimmung von Vor- oder Nachbedingungen. Eine Bedingung ist entweder erfüllt oder nicht. Enthält eine Suchanfrage allerdings auch Input- bzw. Outputs, so ergibt sich daraus eine Hierarchie aus insgesamt acht verschiedenen Einstufungen für eine Methode, wobei der Zusatz PE für gültige Vor- und Nachbedingungen steht:

$$EXACT\_PE > PLUGIN\_PE > SUBSUMES\_PE > SUBSUMED\text{–}BY\_PE >$$
$$EXACT > PLUGIN > SUBSUMES > SUBSUMED\text{–}BY$$

   Die so klassifizierten Methoden werden am Ende in einem Ranking-Schritt sortiert und gemäß obiger Hierarchie gereiht. Wurden mehrere Methoden gefunden, die über den selben logikbasierten Übereinstimmungsgrad verfügen, werden diese innerhalb der Hierarchie absteigend nach ihrer textuellen Übereinstimmung sortiert.

   Liefert das logikbasierte Matching nicht genug Ergebnisse, wird die ursprüngliche Suchanfrage zudem als reine Text-Query interpretiert und an die textbasierte Suche, wie in 4.5.1 beschrieben, weitergeleitet. Die durch diesen Schritt (möglicherweise) zusätzlich gefundenen Methoden werden an die vorhandene Ergebnisliste angereiht, sodass die semantische Suche stets Priorität gegenüber der textbasierten Suche hat.

4.6 Integration in bestehende Softwarelandschaften

SEnSoR läuft als eigenständige Anwendung auf einem Application Server innerhalb eines Netzwerks. Die Funktionalität des Systems kann auf mehrere Arten genutzt werden.

4.6.1 Administration

Die Administration in SEnSoR erfolgt über eine graphische Benutzeroberfläche in Form einer Webanwendung und ist bequem über den Webbrowser erreichbar. Nach dem erfolgreichen Login stehen der Nutzerin – abhängig von ihren Rechten – jeweils eigene Menüpunkte für die Suche, das Browsen, sowie das Hinzufügen von Services anhand einer URL oder durch das Hochladen des Quellcodes zur Verfügung.
Sowohl Services als auch Methoden verfügen jeweils über eine Detailansicht (Abbildung 4.22 rechts), welche jeweils einen Überblick über das indizierte Service bzw. die indizierte Methode bietet und im Falle von Methoden das (nachträgliche) Editieren von Inputs, Outputs, Preconditions und Effects ermöglicht.
Administrationen haben zudem Zugriff auf die Benutzerverwaltung und können über eine Konfigurationsseite (Abbildung 4.22 links) zudem Eingriffe in das System vornehmen, wie etwa der Änderung des JavaScript-Codes für die MapReduce-Implementierung der textbasierten Suche (siehe 4.5.1) zur Laufzeit.

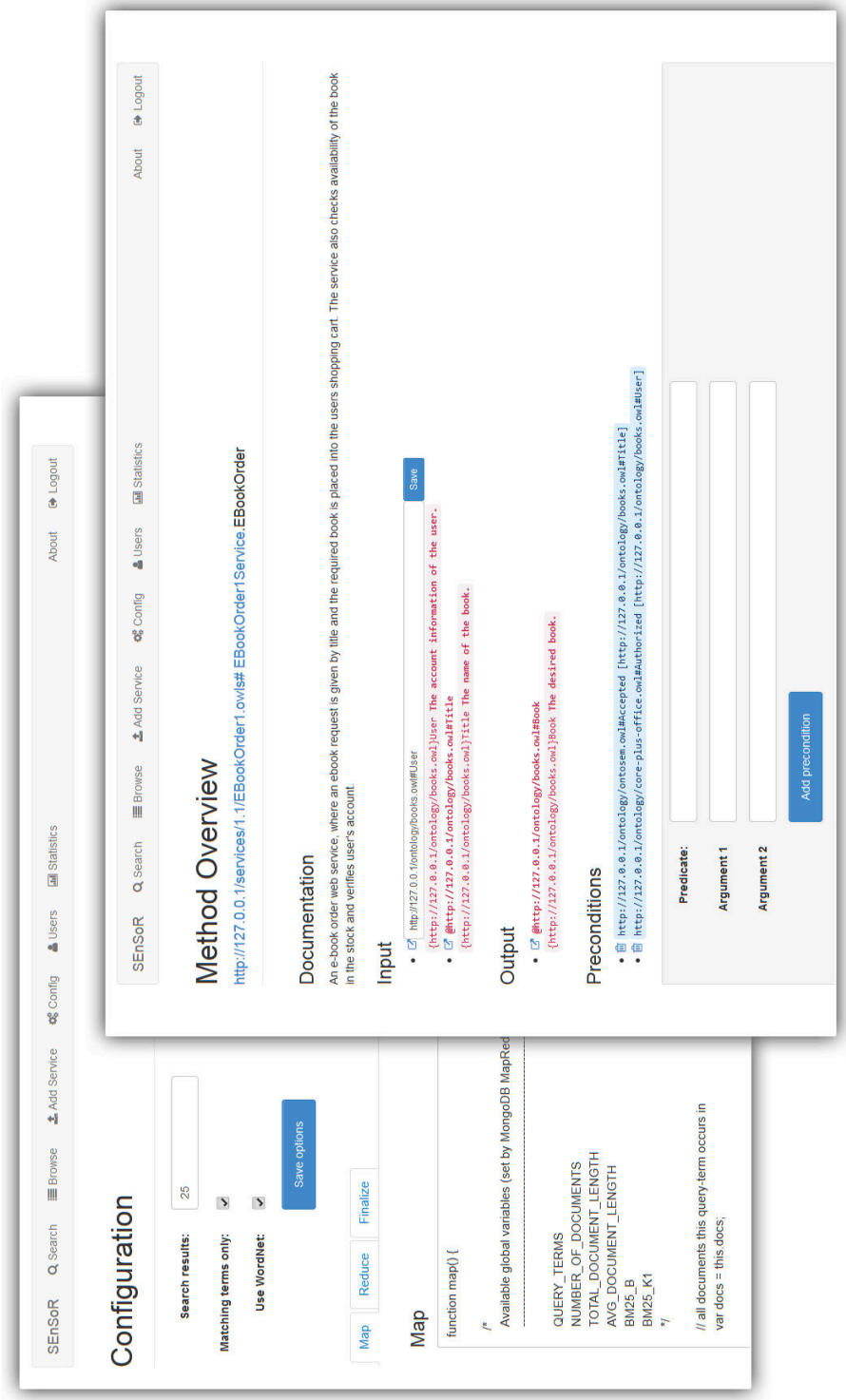

Abbildung 4.22: Die Web-Oberfläche von SEnSoR .

4.6.2 Query-API

Die Suche nach Services bzw. Methoden ist sowohl über die SEnSoR -Webanwendung als auch plattformunabhängig über ein WSDL-basiertes Web Service zugänglich, welches gemeinsam mit dem Repository auf dem Application-Server läuft. Die Query-API bietet die folgenden aufrufbaren Methoden:

- **`List<Service> findServices(String query)`**
- **`List<Method> findMethods(String query)`**

  Bei einer Suchanfrage (*query*) handelt es sich in beiden Fällen – wie bei modernen Suchmaschinen üblich – um durch Leerzeichen getrennte Begriffe, welche durch ein logisches UND verknüpft sind. Diese Begriffe werden für die textbasierte Suche herangezogen.

  Zusätzlich gibt es die Möglichkeit, über eine Reihe von Modifikatoren Filter zu setzen, welche den Suchraum einschränken und damit die Suche präzisieren. Modifikatoren bestehen jeweils aus einem Buchstaben, gefolgt von einem „ist-gleich"-Zeichen und lauten wie folgt:

  - **c (category):** Kategorie eines Service, interpretiert als einfacher String.
  - **t (type):** Typ eines Service. Unterstützte Typen sind: JAVA, OWLS und WSDL.

  Für die Suche nach Methoden sind zudem die folgenden vier Modifikatoren verfügbar, welche jeweils die semantische Suche triggern:

  - **i (input):** Referenz auf eine OWL-Klasse für einen Input-Parameter.
    Beispiel: *i=Book*.
  - **o (output):** Referenz auf eine OWL-Klasse für einen Output-Parameter.
    Beispiel: *o=Author*.
  - **p (precondition):** Referenz auf eine Vorbedingung als OWL-Klasse, ObjectProperty, DataProperty oder ein unterstütztes SWRL Built-In. Es kann entweder (vereinfacht) das Prädikat alleine oder vollständig mit den Typen seiner jeweiligen Argumente angegeben werden.
    Beispiel: *p=hasCreditCard*, *p=hasCreditCard(Customer,VisaCard)*
  - **e (effect):** Referenz auf eine Nachbedingung als OWL-Klasse, ObjectProperty, DataProperty oder ein unterstütztes SWRL Built-In, entweder mit oder ohne Argumenttypen.
    Beispiel: *e=Authorized*, *e=Authorized(User)*

  Bei den von SEnSoR unterstützten SWRL Built-Ins handelt es sich um die Vergleichsoperatoren: *lessThan*, *lessThanOrEqual*, *greaterThan* sowie *greaterThanOrEqual*.

Ist der Name einer OWL-Klasse innerhalb der von SEnSoR indizierten Ontologien nicht eindeutig, wird eine Exception geworfen, welche die Nutzerin darüber informiert und zugleich als Hilfestellung sämtliche Klassen gleichen Namens mitsamt URI listet. In diesem Fall muss die Klasse vollständig über deren URI qualifiziert werden, zum Beispiel *i=http://example.ac.at/Books.owl#Book*. Andernfalls reicht es jeweils aus, einfach den Namen der Klasse anzugeben.

Das Browsen von Services wird innerhalb der Webanwendung in einer eigenen Ansicht unterstützt. Die Query-API bietet dafür zusätzlich folgende Methoden:

- **`List<String> getCategories()`**

  Liefert eine Liste aller vorhandenen Service-Kategorien.

- **`List<Service> getServicesByCategory(String category)`**

  Liefert alle Services zu einer gegebenen Kategorie.

- **`List<Service> getServicesByType(String type)`**

  Liefert alle Services eines bestimmten Typs (JAVA, OWLS, WSDL).

- **`List<Service> findServices(int n)`**

  Liefert n Services, absteigend sortiert nach ihrem Datum der Indizierung. Für $n < 0$ werden *alle* Services retourniert.

- **`List<Method> findMethods(int n)`**

  Liefert n Methoden, absteigend sortiert nach ihrem Datum der Indizierung. Für $n < 0$ werden *alle* Methoden retourniert.

4.6.3 Das SEnSoR Plugin für Eclipse

Eclipse ist eine populäre Entwicklungsumgebung (IDE) für Java-Projekte.[22] Darin integriert ist das *Plug-in Development Environment* (PDE), welches es Entwicklern erlaubt, eigene Plugins zu veröffentlichen und der IDE hinzuzufügen.[23]
Das Plugin für Eclipse bietet die Möglichkeit, die Suchfunktionalität von SEnSoR direkt aus der IDE heraus zu nutzen. Fügt man das Plugin zur IDE hinzu, erhält man eine zusätzliche *Eclipse View*, welche sich beliebig anordnen lässt und eine simple Suchmaske bereitstellt (Abbildung 4.23).
Zur Kommunikation mit dem Repository nutzt das Plugin intern die SEnSoR Query API (4.6.2) über das bereitgestellte Web Service. Damit kommt das Plugin mit einem Mindestmaß an Abhängigkeiten in Form eines einzigen Java-Pakets aus, welches den Java-Code zum Aufruf des Web Service auf Client-Seite beinhaltet. Dieser wird mittels des integrierten Tools `wsimport` aus der Standard-Distribution des *Java Development Kit* (JDK) automatisch aus dem WSDL-Dokuments des Web Service erstellt.

[22] Eclipse IDE: http://www.eclipse.org
[23] Eclipse Plug-in Development Environment (PDE): http://www.eclipse.org/pde

Zurückgelieferte Suchergebnisse werden direkt in Eclipse angezeigt. Zusätzlich kann der Quellcode eines Service (etwa ein WSDL-Dokument) mittels Drag&Drop im Eclipse-Editor betrachtet werden.

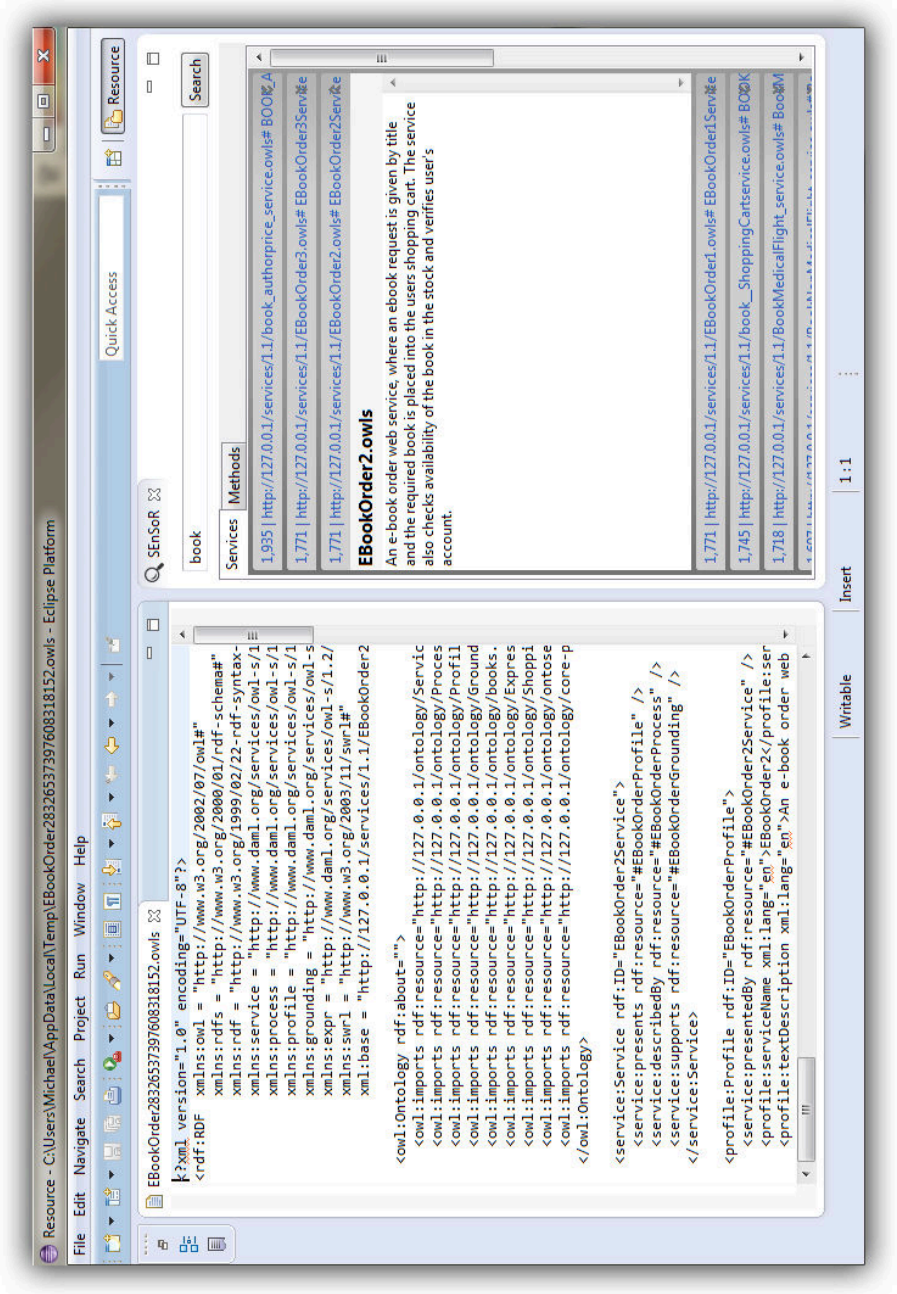

Abbildung 4.23: Das SEnSoR Plugin für die Entwicklungsumgebung Eclipse.

KAPITEL 5

Evaluierung

Eine Hauptaufgabe von SEnSoR ist die Suche nach passenden Services bzw. Methoden. Die beiden wichtigsten und in diesem Kapitel behandelten Kriterien für die Evaluierung eines jeden Matchmaking-Algorithmus sind zum einen dessen Performanz und zum anderen dessen Präzision, also die Qualität der zurückgelieferten Ergebnisse [82].

5.1 Aufbau

5.1.1 Testkorpora

Anders als auf dem Gebiet des *Information Retrieval* existiert bis dato kein echter Standard-Testkorpus für das Matchmaking von Services. Einerseits sollte ein guter Testkorpus eine gewisse Größe hinsichtlich der Anzahl an verfügbaren Test-Services aufweisen, um die Performanz im Sinne des Antwortzeitverhaltens sowie die Skalierung eines Matchmaking-Algorithmus bewerten zu können. Um darüber hinaus die Qualität bzw. die Genauigkeit des Algorithmus zu evaluieren, benötigt man zusätzlich eine Menge an vorgegebenen Suchanfragen mit zugehörigen *relevance sets*, also einer Einstufung zurückgeliferter Suchergebnisse im Hinblick auf deren Relevanz bezüglich der Suchanfrage.
Für die Evaluierung wurden deshalb zwei Testkorpora mit zusätzlich je einer Abstufung gewählt:

- **OWLS-TC 4:** Die *OWL-S service retrieval test collection* ist ein Korpus bestehend aus insgesamt 1.083 Semantic Web Services in OWL-S 1.1, der für die Evaluierung von SWS-Matchmaking-Algorithmen entwickelt wurde und mittlerweile in Version 4 vorliegt.[1] Derzeit handelt sich bei OWLS-TC 4 um den umfassendsten Testkorpus für SWS-Matchmaking [48].

  Der Korpus besteht hauptsächlich aus Services aus (ehemals) öffentlichen UDDI-Registries, welche semi-automatisch von WSDL nach OWL-S transformiert wurden. Sowohl

[1] OWLS-TC 4: http://projects.semwebcentral.org/projects/owls-tc

Inputs als auch Outputs aller inkludierten Services sind von Hand annotiert worden und verweisen jeweils auf Konzepte aus ca. 40 verschiedenen Ontologien, sodass diese für I/O-Matching geeignet sind. Des Weiteren verfügen 166, also etwa 15% aller Services zusätzlich über Preconditions und/oder Effects in SWRL-Syntax, sodass diese Untergruppe auch für IOPE-Matching genutzt werden kann.

OWLS-TC 4 beinhaltet darüber hinaus 42 Suchanfragen (18 davon mit Preconditions und/oder Effects) in Form von Services aus dem Korpus selbst. Bei der Evaluierung ist es nun die Aufgabe, möglichst kompatible bzw. ähnliche Services zu den gegebenen 42 Suchanfragen-Services zu finden. Für jede Suchanfrage existiert zudem ein zugehöriges *relevance set* in Form von passenden Services, die jeweils händisch als relevant im Hinblick auf das Service der Suchanfrage eingestuft wurden. Der Grad der Relevanz liegt sowohl binär (1 = relevant, 0 = irrelevant), als auch in den folgenden Abstufungen vor:

  - 3 (*äußerst relevant*): Services, die exakt das oder sogar mehr erfüllen, was in der Suchanfrage verlangt wurde.
  - 2 (*relevant*): Services, die eine Suchanfrage vollständig oder zumindest teilweise erfüllen *könnten*.
  - 1 (*potentiell relevant*): Services, die noch teilweise mit der Suchanfrage zu tun haben und eventuell hilfreich sein könnten.
  - 0 (*irrelevant*): Services, die mit der Suchanfrage überhaupt nichts zu tun haben.

  Die Einstufung der Suchergebnisse ist nicht vollständig, es existiert also nicht für jede mögliche Kombination aus Suchanfrage und Ergebnis auch eine Bewertung.

- **OWLS-TC 4 (PE):** Jene Untergruppe von 166 Services aus OWLS-TC 4, welche mit Preconditions und/oder Effects annotiert sind (siehe oben).

- **java:** Alle Klassen der Java-Standard-Bibliothek aus dem Paket `java` und dessen Unterpaketen, gefiltert nach statischen Methoden und solchen mit der Sichtbarkeit `public`.

  Bei den meisten Klassen handelt es sich zwar nicht um Services im Sinne dieser Arbeit, jedoch sind diese im Gegensatz zu OWLS-TC4 äußerst sorgfältig dokumentiert (Javadoc) und ergeben dadurch einen wesentlich größeren Textindex, um den Aspekt der Performanz der textbasierten Suche bewerten zu können.

- **java.util:** Die Untergruppe aller Klassen der Java-Standard-Bibliothek aus dem Paket `java.util` samt Unterpaketen.

5.1.2 Werkzeuge

Die Evaluierung der Genauigkeit der hybriden Suche nach Services bzw. Methoden in SEnSoR erfolgt mittels des Java-Programms *Semantic Web Service Matchmaker Evaluation Environment* (SME^2), welches am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI)

Saarbrücken entwickelt wurde und der Evaluierung von Semantic Web Service Matchmaking-Algorithmen dient.[2]
SME^2 verfügt über eine graphische Benutzeroberfläche und inkludiert bereits den Testkorpus OWLS-TC 4, welcher für die Evaluierung der Suche in SEnSoR genutzt wird. Zu diesem Zweck ist es nötig, dem Programm ein Plugin in Form eines Java-Pakets bereitzustellen, welches über eine Klasse verfügt, die das Interface `de.dfki.sme2.IMatchmakerPlugin` implementiert. SME^2 ruft nacheinander für jede gegebene Suchanfrage in Form einer Service-URL aus dem OWLS-TC 4 Testkorpus eine darin spezifizierte Methode auf, welche die vom Matchmaker gefundenen Ergebnisse zurückliefert.
Anschließend präsentiert das Programm eine Gegenüberstellung der Ergebnisse (Abbildung 5.1) – farblich markiert nach dem Grad der Relevanz (siehe 5.1) – und errechnet auf Basis der im Testkorpus inkludierten *relevance sets* eine Reihe an Kennzahlen für die Evaluierung [41].

5.1.3 Testsystem

Als Testsystem fungiert ein Desktop-PC mit Windows 7 64bit, Intel Core i5-4670 Vierkern-Prozessor zu je 3,4GHz, 8GB DDR3 Arbeitsspeicher und einer SATA3-Festplatte mit 7.200 U/min. Lokal laufende Prozesse beinhalten jeweils eine Instanz von MongoDB 2.4.9 für den Textindex, Postgres 9.3 als Datenbank sowie Glassfish 4.0 als Application-Server.

5.2 Performanz

Für die Evaluierung des Antwortzeitverhaltens der Methoden-Suche in SEnSoR dienen die folgenden vier Anfragetypen:

- **Q1 (Text):** Suchanfrage 1 repräsentiert jeweils den Standardfall einer rein textuellen Suche, wie sie von einer Nutzerin eingegeben werden könnte. Die Suchbegriffe sind jeweils an eine bestimmte Methode aus dem jeweiligen Testkorpus angelehnt.

- **Q2 (Text):** Suchanfrage 2 ist ebenfalls rein textbasiert. Anstelle beliebiger Suchbegriffe wurden jedoch jeweils fünf Begriffe aus den „Top-10“ jener Terme gewählt, welche in den meisten indizierten Dokumenten vorkommen, also die höchste *document frequency* aufweisen. Diese Terme provozieren den höchsten Berechnungsaufwand für die textbasierte Suche.

- **Q3 (IO):** Q3 testet zudem die semantische Suche und besteht aus einem Query-Service aus dem OWLS-TC 4 Testkorpus, welches über I/O-Annotationen, jedoch nicht über Preconditions oder Effects verfügt.

  Um mit der SEnSoR Such-API kompatibel zu sein, wird das besagte Service zuerst vom OWL-S-Parser geparst und anschließend eine entsprechende Suchanfrage extrahiert, welche aus der gesamten textuellen Information des Service sowie dessen Inputs und Outputs besteht.

[2] SME^2 : http://projects.semwebcentral.org/projects/sme2

Abbildung 5.1: Übersicht der Suchergebnisse in SME^2 .

- **Q4 (IOPE):** Q4 testet ebenfalls die semantische Suche und besteht auch aus einem Query-Service aus dem OWLS-TC 4 Korpus, welches jedoch im Gegensatz zu Q3 zusätzlich sowohl über Preconditions als auch Effects verfügt.

Die Berechnungen entlang des Suchprozesses gliedern sich in mehrere Teilschritte, welche sequentiell durchgeführt werden. Ist das Ende eines Schrittes erreicht, wird jeweils vom System ein Zeitstempel gesetzt. Anschließend schlüsselt das in SEnSoR eingebaute Statistik-Modul die Gesamtausführungszeit einer Suchanfrage in die folgenden Phasen auf:

1. **Text (Index):** Errechnen und Sortieren der textuellen Ähnlichkeiten aller indizierten Methoden mittels der MapReduce-Implementierung des Vector Space Models (VSM) und *cosine similarity* bzw. BM25.

2. **Text (Datenbank):** Abfrage der in Phase 1 gefundenen Methoden aus der Datenbank.

Im Falle der rein textbasierten Suche ist diese damit abgeschlossen. Für die semantische Suche kommen die folgenden vier Schritte hinzu:

3. **Inferenz:** Abfrage der Subsumption-Beziehungen aller Input- bzw. Output-Konzepte über den OWL-Reasoner Pellet.

4. **IOPE Matching:** Berechnung des Übereinstimmungsgrades indizierter Methoden anhand von IOPE. Dieser Schritt entfällt, wenn die Suchanfrage nur IO beinhaltet.

5. **IO Matching:** Berechnung des Übereinstimmungsgrades indizierter Methoden anhand von IO. Dieser Schritt entfällt, wenn die Suchanfrage nur PE beinhaltet.

6. **Ranking:** Reihung der gefundenen Methoden anhand ihres semantischen und textuellen Übereinstimmungsgrades, wie in 4.5.2.1 und 4.5.2.2 beschrieben.

Tabelle 5.1 gibt einen Überblick über Größe der verwendeten Testkorpora, der benötigten Dauer zur vollständigen Indizierung und einer Gegenüberstellung der beiden textbasierten Anfragetypen Q1 und Q2 für 25 Suchergebnisse. Sowohl Q1 als auch Q2 wurden jeweils fünf Mal ausgeführt, um Schwankungen bei den Ausführungszeiten auszugleichen. Die gemessenen Werte verstehen sich somit jeweils als arithmetisches Mittel innerhalb der fünf Testläufe.
Es lässt sich erkennen, dass die textuelle Suche mit BM25 durchweg bessere Ausführungszeiten als mit dem VSM liefert und darüber hinaus wesentlich besser mit der Größe des Testkorpus skaliert. Dieser Unterschied zeigt sich weniger in der Anzahl der Services bzw. Methoden, als in der Anzahl der indizierten Terme, welche sich hauptsächlich aus dem Umfang der Dokumentation von Methoden ergibt.
Für gewöhnliche Suchanfragen (Q1) liegt die Antwortzeit mit BM25 durchgehend bei etwa 30ms, während sich diese für das VSM und `java.util` bereits auf über eine Sekunde und für den termreichsten Korpus (`java`) auf über acht Sekunden beläuft, was aus Nutzersicht nicht mehr tolerierbar ist. BM25 hingegen liefert selbst bei „Worst-Case“-Suchanfragen (Q2) noch durchwegs gute Antwortzeiten. Alle gemessenen Werte liegen unterhalb von 200ms.

	OWLS-TC 4 (PE)	OWLS-TC 4	java.util	java
Services	**166**	**1.083**	**256**	**1.635**
Terme	616	1.658	5.706	14.866
Dok. Länge (ges.)	4.954	23.220	73.083	267.211
Dok. Länge (avg.)	29,8	21,4	285,5	163,4
Methoden	**166**	**1.083**	**2.767**	**13.753**
Terme	938	1.954	5.236	19.297
Dok. Länge (ges.)	6.768	28.374	190.650	1.025.547
Dok. Länge (avg.)	40,8	26,2	68,9	74,6
Indizierdauer [s]	**7,2**	**27,3**	**36,6**	**437,2**
Services / Sek.	**23,1**	**39,7**	**7,0**	**3,7**
Methoden / Sek.	**23,1**	**39,7**	**75,6**	**31,5**

	BM25	VSM	BM25	VSM	BM25	VSM
Q1 (Text)	*return zipcode of US city*		*get info author who wrote given novel*		*find or create logger*	
Text (Index) [ms]	5	47	16	206	12	1.142
Text (DB) [ms]	20	16	16	15	18	18
gesamt [ms]	**25**	**63**	**33**	**222**	**30**	**1.159**

	service given return city provide		*service return given price inform*		*return param throw specific method*		*return param method null get*	
Q2 (Text)								
Text (Index) [ms]	6	50	20	208	34	1.130	159	8.756
Text (DB) [ms]	20	20	17	16	20	20	19	22
gesamt [ms]	**26**	**70**	**37**	**224**	**54**	**1.150**	**178**	**8.778**

	BM25	VSM	BM25	VSM
Q1 (Text)	*correctly rounded positive square root*			
Text (Index) [ms]	9	8.087		
Text (DB) [ms]	22	19		
gesamt [ms]	**31**	**8.106**		

Tabelle 5.1: Benchmark-Ergebnisse: Indizierung und textbasierte Methoden-Suche.

	OWLS-TC 4 (PE)		OWLS-TC 4	
	BM25	**VSM**	**BM25**	**VSM**
Q3 (IO)	*5 OWLS-Services mit IO*		*5 OWLS-Services mit IO*	
Text (Index) [ms]	13	50	36	214
Text (DB) [ms]	19	25	16	18
Inferenz [ms]	0	3	0	0
IOPE Matching [ms]	0	0	0	0
IO Matching [ms]	3	0	14	10
Ranking [ms]	0	0	0	0
gesamt [ms]	**35**	**78**	**66**	**242**
Q4 (IOPE)	*5 OWLS-Services mit IOPE*		*5 OWLS-Services mit IO*	
Text (Index) [ms]	13	50	46	216
Text (DB) [ms]	25	19	16	20
Inferenz [ms]	0	0	0	0
IOPE Matching [ms]	16	18	22	22
IO Matching [ms]	3	3	12	12
Ranking [ms]	0	0	0	0
gesamt [ms]	**56**	**91**	**96**	**270**

Tabelle 5.2: Benchmark-Ergebnisse: Semantische Methoden-Suche.

Tabelle 5.2 zeigt die Ergebnisse der semantischen Suche anhand der Anfragetypen Q3 und Q4, welche jeweils fünf Mal mit unterschiedlichen Query-Services aus dem OWLS-TC 4 Korpus durchgeführt und anschließend gemittelt wurden.
Zur besseren Veranschaulichung stellen die beiden Diagramme aus Abbildung 5.2 auf der Folgeseite die Ausführungszeiten sämtlicher Suchanfragen gegenüber und zeigen den jeweiligen Anteil der verschiedenen Phasen an der Gesamtausführungszeit:

1. **Text (Index):** Wie aus den beiden Diagrammen zu erkennen, hat die Suche im Textindex speziell für größere Korpora den größten Anteil an der Gesamtzeit. Für jede Suchanfrage wird ein MapReduce-Job gestartet, dessen Ergebnisse anschließend in einem *SortedSet* (das Java-Äquivalent einer sortierten Liste) nachsortiert werden müssen. Dieser Vorgang skaliert stark mit der Anzahl an indizierten Termen.

2. **Text (Datenbank):** Die darauffolgende Abfrage der Ergebnisse aus der Datenbank ist konstant, was daran liegt, dass es sich dabei um eine Datenbankoperation handelt, die lediglich minimal mit der Zahl an gewünschten Suchergebnissen skaliert.

3. **Inferenz:** Die Zeit für die Inferenz ist verschwindend gering, da sämtliche verwendeten Ontologien durchweg im Hauptspeicher gehalten werden und der verwendete Reasoner Pellet ausschließlich (erneut) angestoßen werden muss, wenn eine zusätzliche Ontologie hinzukommt.

4./5. **IOPE Matching / IO Matching:** Sowohl das IO-, als auch das IOPE-Matching sind in Java-Logik implementiert und machen bei semantischen Abfragen den zweiten großen

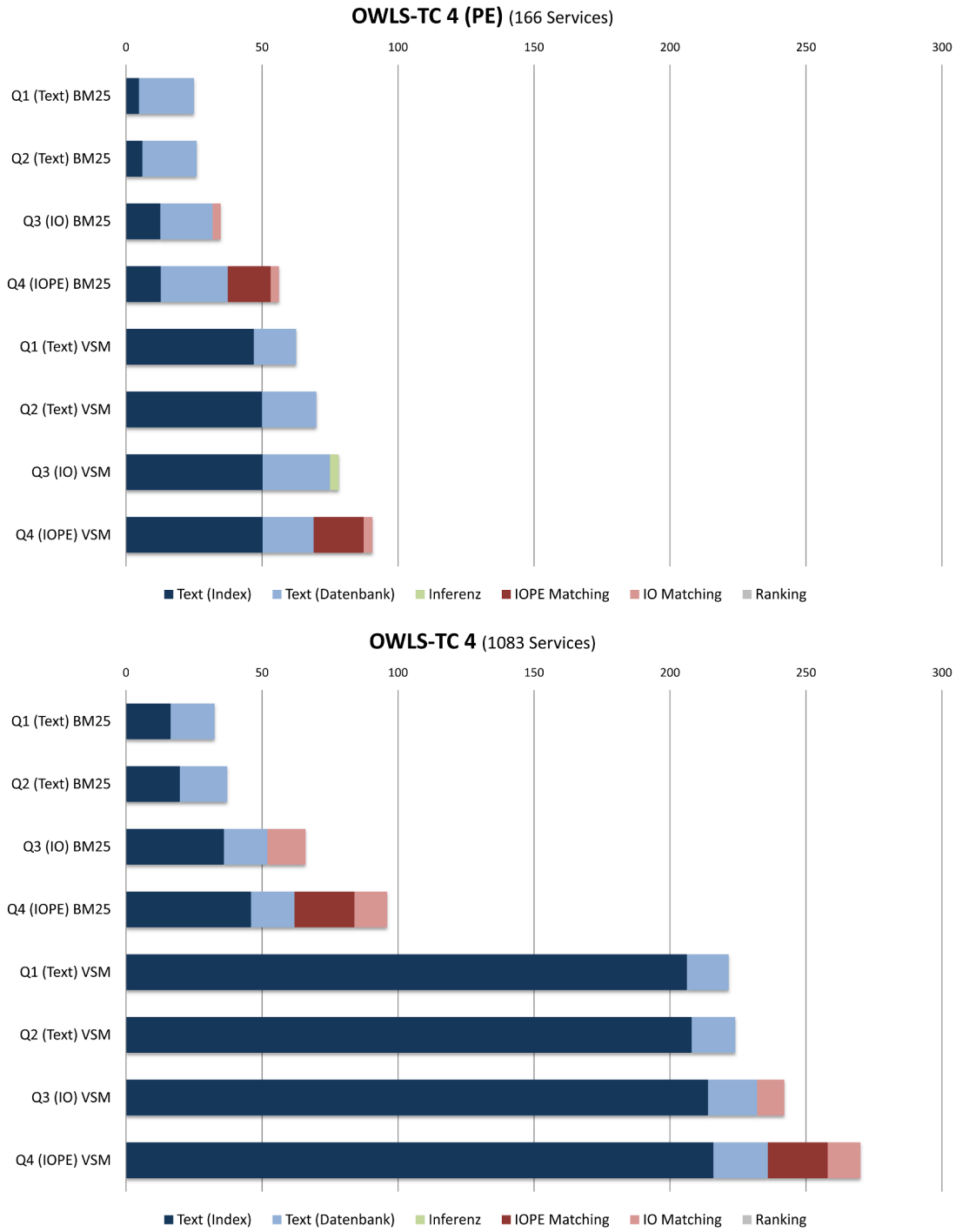

Abbildung 5.2: Vergleich Antwortzeitverhalten in [ms].

Anteil an der Gesamtausführungszeit aus. Beide Vorgänge skalieren jeweils mit der Anzahl der spezifizierten IOPE-Parameter, welche im Normalfall gering ausfällt, sowie – weniger stark – mit der Anzahl der geforderten Suchergebnisse, da der Vorgang in diesem Fall für mehrere Services durchgeführt werden muss.

6. **Ranking:** Die letzte Phase, das Ranking der Ergebnisse, liegt wieder unter dem Millisekunden-Bereich. Hierbei werden sämtliche Ergebnis-Listen zusammengeführt und nach Übereinstimmungsgrad sortiert, sodass diese Operation mit der Anzahl der gefundenen Ergebnisse aus den vorangegangenen Phasen skaliert.

5.3 Präzision

5.3.1 Precision/Recall

Precision und *Recall* gehören zu den wichtigsten Basiskenngrößen auf dem Gebiet des Information Retrieval und sind wie folgt definiert [11]:

- **Precision** (dt. „Genauigkeit“): Anteil an zurückgelieferten Ergebnissen, die relevant im Hinblick auf die Suchanfrage sind.

$$\text{precision} = \frac{|\{\text{relevant documents}\}| \cap |\{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|} \tag{5.1}$$

- **Recall** (dt. „Vollständigkeit“) Anteil der zurückgelieferten, relevanten Dokumente an der Gesamtheit aller relevanten Dokumente.

$$\text{recall} = \frac{|\{\text{relevant documents}\}| \cap |\{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|} \tag{5.2}$$

Abbildung 5.3 zeigt einen Precision/Recall-Plot für den OWLS-TC 4 Testkorpus, der das Verhältnis der beiden Kenngrößen darstellt. Sind die ersten zurückgelieferten Suchergebnisse noch meist relevant (hohe Precision bei geringem Recall), so sinkt die Precision zusehends mit steigender Anzahl an gefundenen Ergebnissen (geringe Precision bei hohem Recall), da weit nach hinten gereihte Ergebnisse meist nicht mehr relevant für die Suchanfrage sind. Daraus ergibt sich der charakteristische, sinkende Verlauf einer Precision/Recall-Kurve.
Der Zusatz *macro-averaged* bezeichnet zudem den Durchschnitt, also das statistische Mittel aus allen 42 Suchanfragen. Die drei Kurven repräsentieren die rein textbasierte Suche mit BM25 (rot) sowie die semantische Suche mit Input/Output-Subsumption (grün) und zusätzlichem Precondition/Effect-Matching (rosa). Wie man aus dem Diagramm erkennen kann, schneidet die rein textbasierte Suche (erwartungsgemäß) am schlechtesten ab, gefolgt von der hybriden Variante mit Input/Output-Matching. Das zusätzliche Precondition/Effect-Matching verbessert die Präzision bei geringem Recall ein weiteres Mal, wenn auch geringfügig.

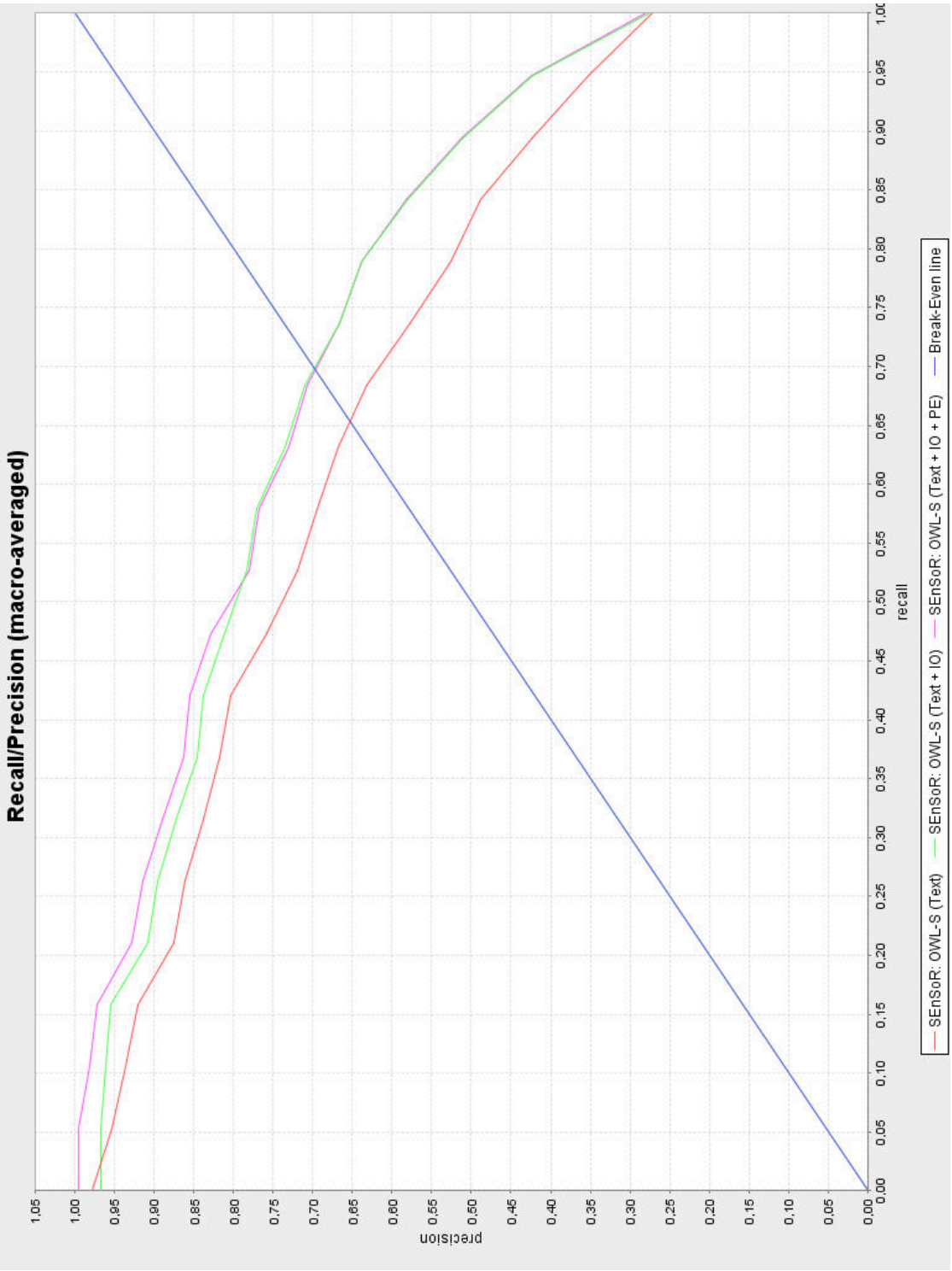

Abbildung 5.3: Recall/Precision-Diagramm (OWLS-TC 4).

5.3.2 Precision „at k“ & R-precision

Für die Nutzerin ist im Rahmen der Suche nach Services (und für Suchmaschinen im Allgemeinen) weniger das Gesamtergebnis der Suche, sondern insbesondere die Qualität der obersten „top-k“ Ergebnisse interessant.
Die Metrik *Precision at k* führt aus diesem Grund eine Grenze nach k zurückgelieferten Resultaten ein und gibt die Präzision auf Basis dieser k Ergebnisse an. Befinden sich etwa 8 relevante Ergebnisse unter den ersten zehn, so bedeutet das für $k = 10$ eine *Precision at 10* von 0,8. Existieren in einem Korpus weniger als k relevante Ergebnisdokumente für eine Suchanfrage, kann eine Genauigkeit von 1 nicht erreicht werden.
Die verwandte Metrik *R-precision* oder auch *Precision at R* [17] berücksichtigt diesen Umstand und gibt die Genauigkeit an der R-ten Stelle der zurückgelieferten Suchergebnisse an, wobei R die Gesamtanzahl aller relevanten Dokumente zu einer Suchanfrage bezeichnet. Sind somit r relevante Ergebnisse unter den top-R, so wird das Verhältnis daraus einfach wie folgt berechnet:

$$\text{R-precision} = \frac{r}{R} \tag{5.3}$$

Die beiden Diagramme aus Abbildung 5.4 und 5.5 auf den Folgeseiten zeigen jeweils die gemessenen Werte für *Precision at k=10* sowie *R-precision* für alle 42 Suchanfragen aus dem OWLS-TC 4 Korpus.
Wie man daraus erkennen kann, liefert die hybride Suche für 26 von 42 Anfragen ausschließlich relevante Ergebnisse innerhalb der ersten zehn. Im Schnitt beläuft sich die *R-precision* auf ca. 0,7 für die hybride und 0,64 für die rein textbasierte Suche.
Für einige der Ausreißer wie Suchanfrag 1 oder 5 existieren lediglich zwei relevante Ergebnisse, sodass die *Precision at 10* wie weiter oben beschrieben nicht über 0,2 hinausgehen kann und der optische Eindruck damit etwas verzerrt wird. Vergleicht man diese Anfragen mit dem Diagramm für R-Precision (0.5), stellt man fest, dass innerhalb der ersten zwei Ergebnisse jeweils nur eines der beiden relevanten gefunden wurde.
Dieses Problem ergibt sich in beiden Fällen aus der Struktur des Anfrage-Service, welches jeweils speziell so konstruiert ist, dass die Preconditions/Effects der möglichen Kandidaten jeweils nur teilweise erfüllt werden. In diesem Fall reiht der Matchmaker die Ergebnisse nach ihrem textbasierten Ranking, welches in beiden Fällen das „falsche“ Service nach vorne reiht.
Beispielsweise versetzen die vier Services *open_door.owls*, *close_door.owls*, *unlock_door.owls* sowie *lock_door.owls* aus dem OWLS-TC 4 Korpus eine (fiktive) Tür jeweils in einen der entsprechenden vier Zustände (*Open*, *Closed*, *Locked*, *Unlocked*), welche gleichzeitig die jeweiligen Nachbedingungen darstellen. Der Ursprungszustand der Tür kann hingegen jeweils eine Kombination aus {*Closed*, *Open*} sowie {*Locked*, *Unlocked*} sein. So ist für das Service *open_door.owls* der Anfangszustand beispielsweise *Closed* und *Unlocked*, was über eine entsprechende Vorbedingung modelliert ist.
Da jedoch keine explizite Beziehung zwischen dem Schließ- und dem Öffnungszustand einer Tür innerhalb der verwendeten Ontologie existiert und alle vier Services jeweils unterschiedliche Kombinationen aus besagten Vor- und Nachbedingungen aufweisen, liefert das PE-Matching für keine Paarung der vier Services eine semantische Übereinstimmung untereinander. Während ein menschlicher Betrachter die Funktionalität der Services allein anhand deren Namen erkennen

Abbildung 5.4: Precision at k=10 (OWLS-TC 4).

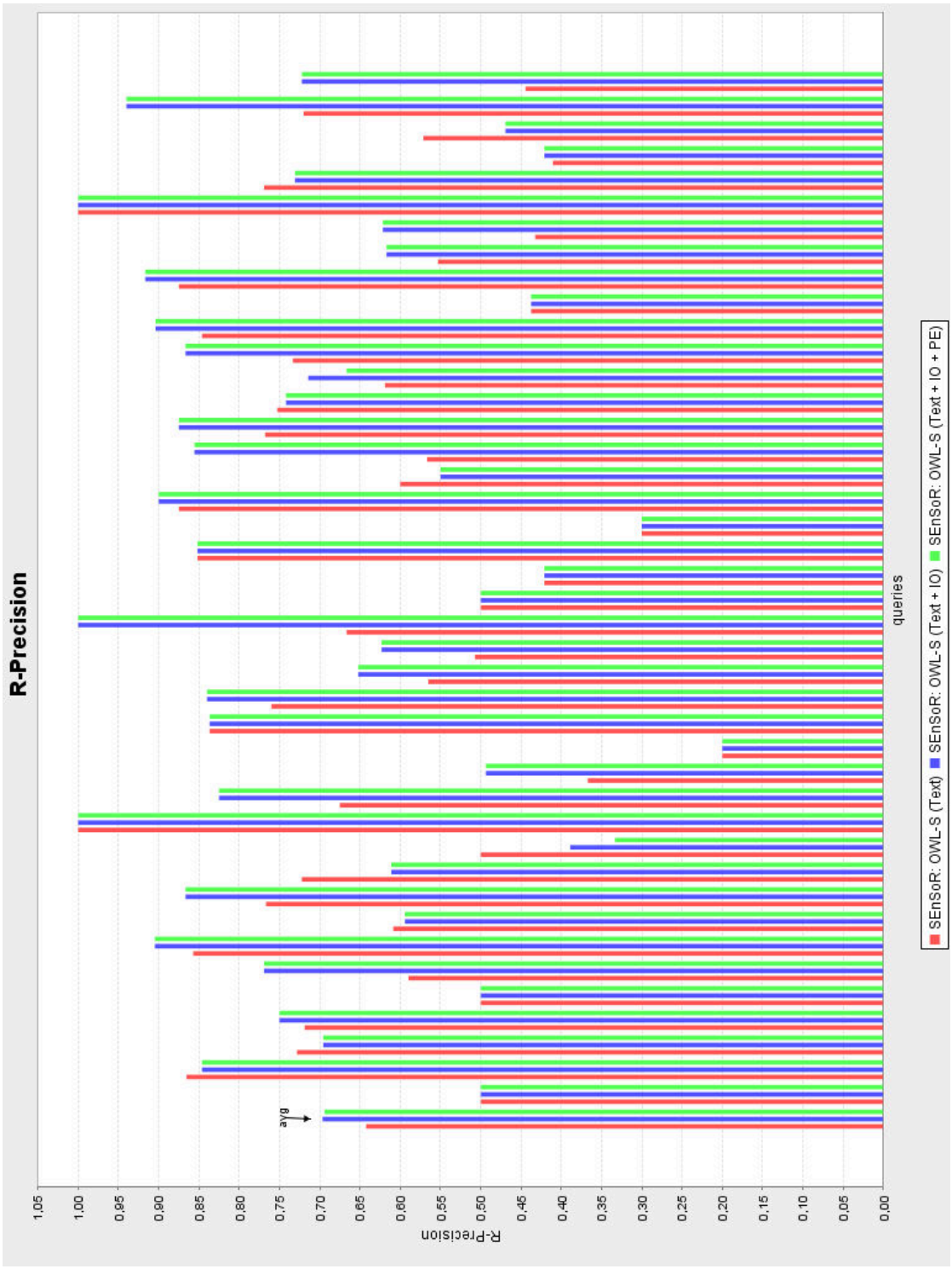

Abbildung 5.5: R-precision (OWLS-TC 4).

kann (beispielsweise assoziiert man das Öffnen einer Tür eher mit dem Aufschließen als dem Zumachen), reiht das semantische Matching aufgrund von fehlender Information ein als nicht relevant eingestuftes Ergebnis nach vorne.

5.3.3 Normalized Distributed Cumulative Gain (nDCG)

Die in 5.3.2 beschriebenen Metriken arbeiten jeweils auf einem binären *relevance set* mit den beiden Ergebnis-Einstufungen 1 (relevant) oder 0 (nicht relevant). Hierbei macht es keinen Unterschied, an welcher Position relevante Ergebnisse zurückgeliefert werden, solange sie innerhalb der ersten k Ergebnisse liegen.
Im Falle des OWLS-TC 4 Korpus liegt zusätzlich für jede Suchanfrage eine Ergebnis-Einschätzung mit den abgestuften Werten von 0 bis 3 (siehe 5.1.1) vor. Dadurch ist es möglich, bei der Evaluierung auch die Reihung der Ergebnisse miteinzubeziehen, um qualitativ gute Ergebnisse mit einer höheren Einstufung an den vorderen Positionen zu belohnen.
Die Metrik *Discounted Cumulative Gain* (DCG) erfüllt diese Anforderung und ist wie folgt definiert [10]:

$$\mathrm{DCG}_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{5.4}$$

Die Variable k steht jeweils für die Position eines zurückgelieferten Ergebnisses, während rel dessen Relevanz-Wert angibt. Je weiter hinten die Position eines Ergebnis-Service, umso mehr wird dessen Anteil am Gesamtergebnis durch den Logarithmus im Nenner abgeschwächt.
Ordnet man alle vorhandenen Ergebniseinstufungen innerhalb des *relevance sets* absteigend nach Grad der Relevanz, erhält man den höchsten maximal möglichen Wert für DCG, den *Ideal DCG* (IDCG). Setzt man diesen nun ins Verhältnis zum DCG, erhält man den sogenannten *normalized DCG* (nDCG), welcher die Qualität der Ergebnisse in Bezug auf das ideale Ergebnis auf das Intervall $\{0, 1\}$ projiziert:

$$\mathrm{nDCG}_k = \frac{\mathrm{DCG}_k}{\mathrm{IDCG}_k} \tag{5.5}$$

Das Diagramm aus Abbildung 5.6 zeigt den nDCG für den OWLS-TC 4 Testkorpus, jeweils für die ersten $k = 10$ zurückgelieferten Ergebnisse. Im Durchschnitt beträgt dieser etwa 0,78 für die rein textbasierte Suche und 0,86 - 0,87 für die hybride Suche. Auch hier zeigt sich, dass das zusätzliche P/E-Matching nur einen geringen Zuwachs an Präzision bringt.
Der hohe nDCG-Wert zeigt, dass die hybride Suche in SEnSoR nicht nur prinzipiell relevante Ergebnisse liefert, sondern qualitativ gute Ergebnisse auch früh nach vorne reiht, was ein mühsames Durchblättern durch die Suchergebnisse reduziert.

5.4 Vergleich mit verwandten Systemen

Tabelle 5.3 zeigt einen Vergleich der hybriden Suche nach Services in SEnSoR mit zehn OWL-S Matchmakern, die an dem zuletzt im Jahr 2012 abgehaltenen Wettbewerb *Semantic Service Se-*

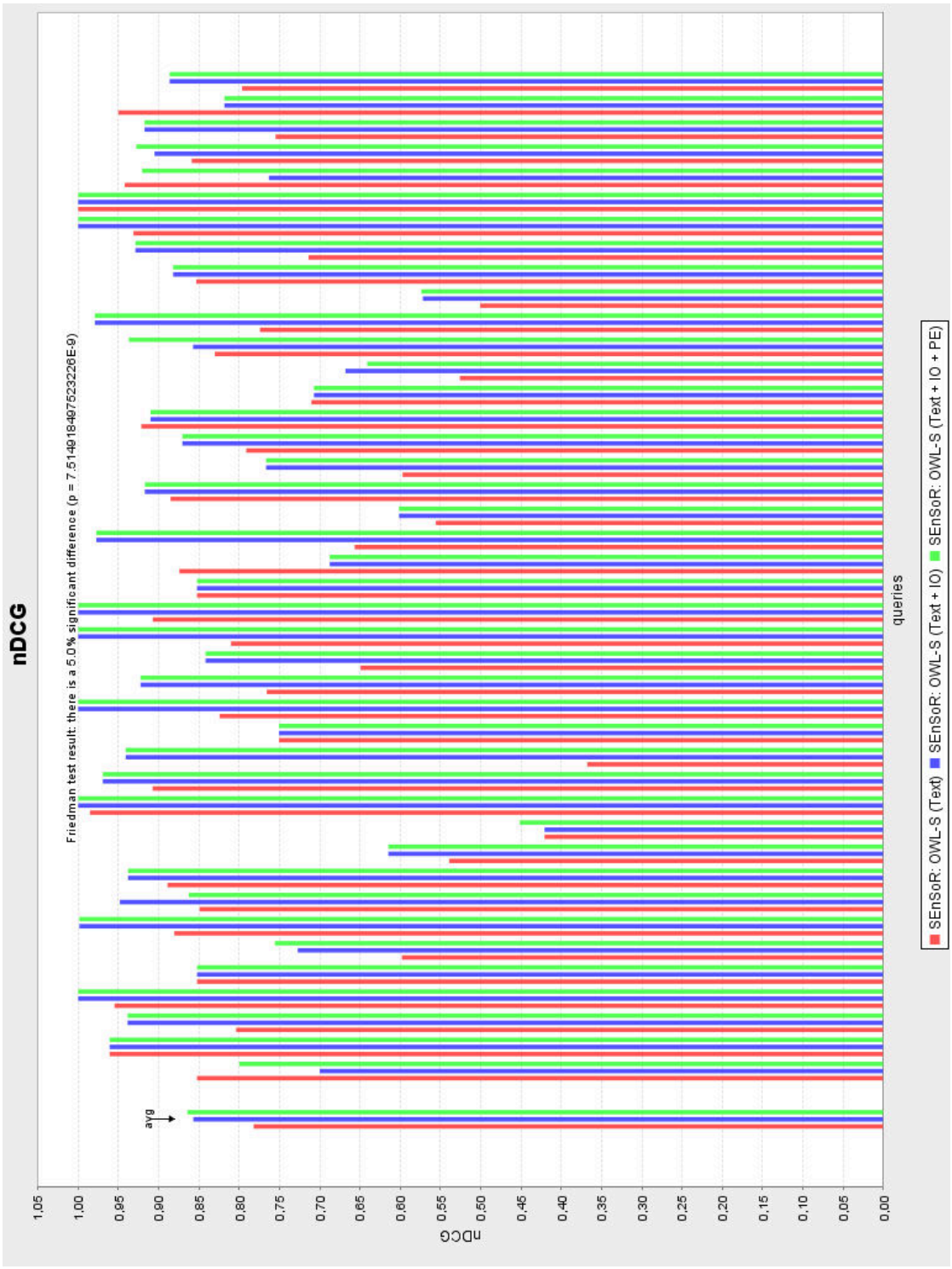

Abbildung 5.6: Normalized Discounted Cumulative Gain für k=10 (OWLS-TC 4).

		nDCG (k=100)	AQRT [ms]	Avg. Precision
1.	SeMa² v2	0,927	5.084	0,877
2.	iSeM-TSM1	0,916	4.447	0,861
3.	Nuwa-OWLS	0,911	18.356	0,853
4.	OWLS-MX3	0,899	5.369	0,831
5.	XSSD	0,881	125	0,795
6.	**SEnSoR / IOPE**	**0,877**	**240**	**0,780**
7.	EMMA	0,870	9.644	0,762
8.	iSeM 1.1	0,841	2.340	0,922
9.	SPARQLent	0,728	576	0,612
10.	OWLS-SLR (lite)	0,723	460	0,609
11.	OWLS-iMatcher	0,719	2.152	0,672

Tabelle 5.3: Vergleich von SEnSoR mit zehn OWL-S Matchmakern aus dem S3 Contest 2012 (OWLS-TC 4).

lection Contest (S3) teilgenommen haben [40].[3]

Als wichtigste Metrik wurde der nDCG ($k = 100$) gewählt, da dieser für die Suche nach Services in Bezug auf die Qualität der zuvorderst zurückgelieferten Ergebnisse die größte Aussagekraft besitzt.

Zudem ist die durchschnittliche Antwortzeit (*average query response time*, AQRT) sowie die durchschnittliche Präzision (*average precision*) für jeweils alle 42 Suchanfragen und 1.083 zurückgelieferten Ergebnisse aufgelistet.

Wie man aus der Tabelle erkennen kann, positioniert sich SEnSoR bezüglich nDCG genau in der Mitte des Kandidatenfelds, wobei der Abstand zum ersten Platz mit 5% weitaus geringer als mit 15,8% zum letzten Platz ausfällt. Ähnlich verhält es sich mit der *average precision* über alle 1.083 Ergebnisse.

Bezüglich der *average query response time* ist SEnSoR den höher platzierten Matchmakern mit Ausnahme von XSSD jedoch deutlich überlegen. Der gemessene Wert von 240ms wird wie in 5.2 gezeigt für praxisorientierte Sucheinstellungen wie lediglich 25 Ergebnissen sogar noch deutlich unterschritten und ergibt sich hier aus dem Umstand, dass für den OWLS-TC 4 Benchmark immer alle 1.083 Ergebnisse erwartet werden und diese aus der Datenbank geladen werden müssen.[4]

Dies stellt zugleich einen wesentlichen Unterschied zwischen SEnSoR und anderen Matchmakern dar, da letztere zumeist alle Daten (Services, Ontologien) ausschließlich im Hauptspeicher halten. Beispielsweise verwendet der hybride Matchmaker XSSD intern sogenannte Service-Feature-Matrizen als Datenstruktur zur Speicherung von Services, wodurch der Kommunikationsaufwand mit einer Datenbank zwar gänzlich entfällt, jedoch auch keinerlei Persistenz gewährleistet ist.

[3] S3 Contest: http://www-ags.dfki.uni-sb.de/klusch/s3

[4] Leider ist der Wert für die AQRT von SEnSoR nicht direkt mit jenen aus dem S3 Contest vergleichbar, da auf unterschiedlichen Testsystemen gemessen wurde und einige Matchmaker nicht (mehr?) lauffähig sind. Der Wert von etwa 240ms ergibt sich aus der Interpolation basierend auf den gemessenen Laufzeiten von *OWLS-MX3* und *XSSD*, welche mit einer Abweichung von etwa 3% jedoch ohnehin sehr ähnlich zu den Werten aus dem S3 Contest sind.

Nur etwa 15% aller Services aus OWLS-TC 4 verfügen über Preconditions und/oder Effects, sodass das zusätzliche P/E-Matching in SEnSoR gegenüber dem reinen Input/Output-Matching entgegen der eigentlichen Erwartung nicht wesentlich schlechter abschneidet. Dieser Umstand zeigt sich speziell an den dicht beieinanderliegenden Werten für SEnSoR und XSSD. Obwohl XSSD kein P/E-Matching durchführt, liefert dieser trotzdem leicht bessere Gesamtergebnisse. Auch die drei bestplatzierten Matchmaker unterstützen diese Beobachtung. Zwar bezieht der erstplatzierte Matchmaker SeMa^2 v2 Preconditions/Effects mit ein, doch die beiden zweit- und drittplatzierten Matchmaker iSeM-TSM1 und Nuwa-OWLS (ohne P/E-Matching) zeigen, dass dies nicht zwingend notwendig für sehr gute Resultate zu sein scheint.
Die Unterschiede in der Präzision ergeben sich neben verschiedenen Ansätzen (siehe 3.2.3) auch aus Implementierungsdetails. So hat bereits das Parsen der textuellen Features von Services großen Einfluss auf die Genauigkeit der Suche. Insbesondere finden sich in OWLS-TC 4 viele Terme ähnlich zu *_diagnosticprocessorganization_service* innerhalb der Service-Beschreibung, was eine Trennung in einzelne Terme unnötig erschwert. Weitere Unstimmigkeiten innerhalb des Korpus inkludieren manch fehlerhaften Verweis innerhalb der OWL-S Servicedokumente (falsche Benennung von Variablen) sowie die teils sehr geringe Anzahl an als relevant eingestuften Ergebnis-Services.
Des weiteren handelt es sich bei Suchanfragen im OWLS-TC 4 Testkorpus selbst um OWL-S Services. Es besteht demnach ein großer Freiheitsgrad an der Formulierung der eigentlichen Suchanfrage bezüglich der dafür verwendeten Terme und IOPE-Parameter, was den direkten Vergleich von Matchmakern zusätzlich erschwert.
Empirische Tests haben zudem gezeigt, dass die Veränderung einzelner Parameter innerhalb der hybriden Suche vergleichsweise große Auswirkungen auf die Messwerte haben. So verschlechtert sich der nDCG bei Verwendung der *cosine similarity* mit *tf-idf*-Gewichtung (siehe 4.5.1.3) um 1,6% gegenüber BM25 bei zugleich schlechterer Performanz. Auch die Reihung der Ergebnisse hat erheblichen Einfluss auf die Präzision. Die in 4.5.2 beschriebene Reihung ergibt jedoch im Mittel das bestmögliche Resultat.

KAPITEL 6

Conclusio

Abschließend werden die Ergebnisse dieser Arbeit zusammengefasst und einige offene Punkte vorgestellt.

6.1 Zusammenfassung

Das Ziel dieser Arbeit bestand im Design und der Entwicklung eines Software-Repository, in welchem sowohl traditionelle Softwarekomponenten als auch (Semantic) Web Services verwaltet werden können. Das Hauptanwendungsgebiet des Repository liegt in der Wiederverwendbarkeit existierender Services und soll Nutzerinnen bei der Auswahl passender Services oder Methoden, beispielsweise im Kontext der Modellierung von Geschäftsprozessen, unterstützen.
Konkret wurden folgende Problemstellungen behandelt.

1. *Wie können sowohl traditionelle Softwarekomponenten in unterschiedlichen Programmiersprachen als auch (Semantic) Web Services in einem Software Repository einheitlich verwaltet werden?*

In SEnSoR werden Services bei der Aufnahme in das Repository unabhängig von ihrer konkreten technologischen Ausprägung in ein einheitliches Datenmodell überführt, welches die funktionalen Eigenschaften von Services abdeckt. Diese Aufgabe wird – je nach Typ des Service – von einer entsprechenden Parser-Implementierung erledigt, welche Informationen gemäß dem Datenmodell aus der Service-Beschreibung extrahiert.
Anhand einer jeweiligen Implementierung für Java, WSDL und OWL-S wurde gezeigt, dass dieser Ansatz nicht nur für klassische Softwarekomponenten, sondern auch für (Semantic) Web Services funktioniert. Durch die Bereitstellung weiterer Parser-Implementierungen wäre es zudem möglich, weitere Technologien wie beispielsweise *RESTful Web Services* zu unterstützen.
Die Administration des Repositories selbst sowie die Verwaltung der darin gespeicherten Services erfolgt über eine Webanwendung.

2. *Wie kann die semantische Annotation von Services in Verbindung mit Methoden aus dem Bereich des Semantic Web dabei helfen, die Auswahl passender Services zu präzisieren?*

In SEnSoR lassen sich sowohl Input- und Output-Parameter als auch Vor- und Nachbedingungen von Methoden mit Referenzen auf entsprechende Konzepte einer Domänen-Ontologie versehen. Diese werden, sofern vorhanden, bereits bei der Aufnahme des Service in das Repository mit eingelesen oder können nachträglich über die Webanwendung hinzugefügt werden.
Dies ermöglicht einen hybriden Ansatz für die Suche nach Services bzw. Methoden, welcher die klassische textbasierte Suche mit logikbasiertem IOPE-Matching verbindet. Nutzerinnen können dementsprechend nicht nur nach bloßen Stichwörtern, sondern auch gezielt nach Methoden mit entsprechenden Inputs, Outputs, Vor- oder Nachbedingungen suchen.
Ein in SEnSoR integrierter Matchmaking-Algorithmus ermittelt den Grad der semantischen bzw. der textuellen Ähnlichkeit indizierter Services mit einer Suchanfrage und liefert nach einem Ranking-Schritt jene Services oder Methoden mit der höchsten Übereinstimmung zurück.
Die Evaluierung hat zudem ergeben, dass SEnSoR hinsichtlich der Qualität zurückgelieferter Ergebnisse mit aktuellen Semantic Web Service Matchmakern mithalten kann und viele davon sogar übertrifft. Gleichzeitig weist SEnSoR ein sehr gutes Antwortzeitverhalten auf und beantwortet Suchanfragen selbst bei einer großen Anzahl an indizierten Services innerhalb eines Bruchteils einer Sekunde, was hauptsächlich auf die effiziente Implementierung der textbasierten Suche mittels des MapReduce-Paradigmas zurückgeführt werden kann.

6.2 Weiterführende Punkte

Im Rahmen der Konzeption und Entwicklung von SEnSoR ergaben sich einige weiterführende Punkte, die über den Umfang dieser Arbeit hinausgehen, für die weitere Entwicklung jedoch berücksichtigt werden könnten:

- **Präzision:** Die Evaluierung der Genauigkeit der in SEnSoR implementierten Suche ergab unter anderem, dass die Nutzung von BM25 anstatt des Vector Space Models mit *cosine similarity* und *tf-idf*-Gewichtung ein Plus an Präzision von etwa 1,6% (nDCG) bringt. Es wäre zu evaluieren, ob andere Verfahren weitere Verbesserungen erzielen.

  Eine interessantere Erweiterung zu BM25 in diesem Kontext ist etwa BM25F [68], ein Modell, das die Struktur eines Dokumentes beim Ranking miteinbezieht. Die grundlegende Idee ist, dass Terme, die zum Beispiel im Namen eines Service oder einer Methode vorkommen, höher gewichtet werden als beispielsweise Terme aus der Dokumentation.

  Da der Textindex in SEnSoR wie in 4.5.1.1 beschrieben auf der Schema-freien Datenbank MongoDB basiert, ließe sich beim Indizieren solcher Terme ein zusätzlicher Parameter im Index abspeichern, der dessen strukturelle Zugehörigkeit angibt. Diese Parameter könnten dann innerhalb eines MapReduce-Jobs verwendet werden, um ein ähnliches Termgewichtungsverfahren wie in BM25F zu implementieren.

- **Monitoring:** Die Suchfunktionalität in SEnSoR ist konzeptionell auf funktionale Anforderungen von Services ausgelegt. Diese müssen erfüllt sein, damit ein Service aufrufbar

ist. Nicht-funktionale Anforderungen können jedoch ebenso die Auswahl eines Service beeinflussen, etwa wenn dieses eine sehr schlechte Verfügbarkeit aufweist.

Durch das Hinzufügen eines Monitoring-Moduls und einer entsprechenden Erweiterung des Datenmodells könnten auch nicht-funktionale Anforderungen in SEnSoR integriert werden. Eine Client-Anwendung, die ein Service aus SEnSoR nutzt, könnte nach dessen Aufruf beispielsweise über das *Interceptor*-Pattern [61] eine Nachricht an das Monitoring-Modul schicken, welche Informationen über die (erfolgreiche) Ausführung wie bspw. Ausführungszeit beinhaltet. Auf Basis dieser Log-Nachrichten kann das Monitoring-Modul *Quality of Service* (QoS) Attribute von Services ermitteln.

- **Versionierung:** Bei der Aufnahme eines neuen Service prüft das Service-Management Modul, ob bereits eine frühere Version desselben Service vorhanden ist. Falls ja, kann dieses unter einer neuen Versionsnummer in das Repository aufgenommen werden.

  Verfügt eine frühere Service-Version jedoch über manuell hinzugefügte Informationen wie etwa Vor- oder Nachbedingungen von Methoden, werden diese im Moment nicht auf das neue Service (sofern möglich) übertragen. Hierzu wäre ein zusätzlicher Matching-Schritt erforderlich, der die Signaturen von Methoden prüft und bereits vorhandene Information gegebenenfalls auf die neue Version überträgt.

- **Notifikation:** Im Laufe der Zeit können Änderungen sowohl an bestehenden Services als auch an referenzierten Ontologien auftreten. Langfristig gesehen ist es deshalb sinnvoll, diese Änderungen aktiv von einem weiteren Modul überwachen zu lassen und Nutzerinnen gegebenenfalls zu informieren. Zu diesem Zweck kann das ohnehin bestehende Benutzerverwaltungsmodul um ein Notifikationsmodul erweitert werden, welches über ein *Publish/Subscribe*-Modell besagte Änderungen an subskribierte Nutzerinnen kommuniziert.

Literaturverzeichnis

[1] Jörg Becker, Oliver Müller, and Manuel Woditsch. An Ontology-Based Natural Language Service Discovery Engine - Design and Experimental Evaluation. In *Proceedings of the 18th European Conference on Information Systems (ECIS '10)*, 2010.

[2] Umesh Bellur and Harin Vadodaria. On Extending Semantic Matchmaking to Include Preconditions and Effects. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2008)*, pages 120–128. IEEE Computer Society, 2008.

[3] Ayse Basar Bener, Volkan Ozadali, and Erdem Savas Ilhan. Semantic matchmaker with precondition and effect matching using SWRL. *Expert Systems with Applications: An International Journal*, 36(5):9371–9377, July 2009.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[5] Nicolas Boissel-Dallier, Jean-Pierre Lorré, and Frédérick Bénaben. Management Tool for Semantic Annotations in WSDL. In *Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009 (OTM '09)*, pages 898–906. Springer, 2009.

[6] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture (2004): http://www.w3.org/TR/2004/NOTE-ws-arch-2004021.

[7] Luis Botelho, Alberto Fernández, Benedikt Fires, Matthias Klusch, Lino Pereira, Tiago Santos, Pedro Pais, and Matteo Vasirani. *Service Discovery*, chapter 10, pages 205–234. Birkhäuser Basel, 2008.

[8] Steve Bratt. Semantic Web, and Other Technologies to Watch (W3C Vortrag) (2006): http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb.

[9] Alan W. Brown and Kurt C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–46, September 1998.

[10] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to Rank Using Gradient Descent. In *Proceedings of the 22nd International Conference on Machine Learning (ICML '05)*, pages 89–96. ACM, 2005.

[11] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval – Implementing and Evaluating Search Engines*. MIT Press, 2010.

[12] Stefano Ceri, Alessandro Bozzon, Marco Brambilla, Emanuele Valle, Piero Fraternali, and Silvia Quarteroni. The Information Retrieval Process. In *Web Information Retrieval*, pages 13–26. Springer Berlin Heidelberg, 2013.

[13] Yassin Chabeb and Samir Tata. Yet Another Semantic Annotation for WSDL. In *Proceedings of the IADIS International Conference on WWW/Internet (WWW/I-08)*, pages 437–441. IADIS Press, 2008.

[14] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 2.0 (2007): http://www.w3.org/TR/2007/REC-wsdl20-20070626.

[15] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1 (2001): http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[16] Zijie Cong and Alberto Fernández. Behavioral Matchmaking of Semantic Web Services. In *4th International Joint Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (SMR2)*, volume 667, pages 131–140. CEUR Workshop Proceedings, November 2010.

[17] Nick Craswell. R-Precision. In *Encyclopedia of Database Systems*, page 2453. Springer, 2009.

[18] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax (2014): http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225.

[19] Adam Czyszczoń and Aleksander Zgrzywa. The MapReduce Approach to Web Service Retrieval. In *Computational Collective Intelligence. Technologies and Applications*, volume 8083 of *Lecture Notes in Computer Science*, pages 517–526. Springer Berlin Heidelberg, 2013.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commununications of the ACM*, 51(1):107–113, January 2008.

[21] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. Pairwise Document Similarity in Large Collections with MapReduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers (HLT-Short '08)*, pages 265–268. Association for Computational Linguistics, 2008.

[22] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[23] Joel Farrell and Holger Lausen. Semantic Annotations for WSDL and XML Schema (2007): http://www.w3.org/TR/2007/REC-sawsdl-20070828.

[24] David Ferraiolo and Richard Kuhn. Role-Based Access Control. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[25] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The Vocabulary Problem in Human-system Communication. *Communications of the ACM*, 30(11):964–971, November 1987.

[26] José María García, David Ruiz, and Antonio Ruiz-Cortés. Improving Semantic Web Services Discovery Using SPARQL-Based Repository Filtering. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:12–24, December 2012.

[27] Maciej Gawinecki, Giacomo Cabri, Marcin Paprzycki, and Maria Ganzha. WSColab: Structured Collaborative Tagging for Web Service Matchmaking. In *Proceedings of the 6th International Conference on Web Information Systems and Technologies, Volume 1 (WEBIST '10)*, pages 70–77. INSTICC Press, 2010.

[28] Ayse Goker, John Davies, and Margaret Graham. *Information Retrieval: Searching in the 21st Century*. John Wiley & Sons, 2007.

[29] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition) (2012): http://www.w3.org/TR/2012/REC-owl2-overview-20121211.

[30] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993.

[31] Christian Gutschier. How to execute parts of BPMN. Master's thesis, Technische Universität Wien, 2014.

[32] David Haas and Allen Brown. Web Services Glossary (2004): http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211.

[33] Djoerd Hiemstra and Claudia Hauff. MapReduce for Experimental Search. In *Proceedings of the 19th Text REtrieval Conference (TREC 2010)*. National Institute of Standards and Technology (NIST), 2010.

[34] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML (2004): http://www.w3.org/Submission/2004/SUBM-SWRL-20040521.

[35] Waldemar Hummer, Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. VRESCo — Vienna Runtime Environment for Service-oriented Computing. In *Service Engineering*, pages 299–324. Springer Vienna, 2011.

[36] Michael C. Jaeger, Gregor Rojec-Goldmann, Christoph Liebetruth, Gero Mühl, and Kurt Geihs. Ranked Matching for Service Descriptions Using OWL-S. In *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pages 91–102. Springer Berlin Heidelberg, 2005.

[37] Christoph Kiefer and Abraham Bernstein. The Creation and Evaluation of iSPARQL Strategies for Matchmaking. In *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 463–477. Springer Berlin Heidelberg, 2008.

[38] Michael Kifer, Jos de Bruijn, Harold Boley, and Dieter Fensel. A Realistic Architecture for the Semantic Web. In *Proceedings of the First International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML '05)*, pages 17–29. Springer Berlin Heidelberg, 2005.

[39] Matthias Klusch. Semantic Web Service Coordination. In *CASCOM: Intelligent Service Coordination in the Semantic Web*, chapter 4, pages 59–104. Springer, 2008.

[40] Matthias Klusch. *The S3 Contest: Performance Evaluation of Semantic Service Matchmakers*, chapter 1. Springer, 2012.

[41] Matthias Klusch, Minko Dudev, Jozef Mišutka, Patrick Kapahnke, and Martin Vasileski. Semantic Web Service Matchmaker Evaluation Environment (SME^2) User Manual v2.2 (2010): http://projects.semwebcentral.org/projects/sme2.

[42] Matthias Klusch, Benedikt Fries, and Katia Sycara. OWLS-MX: A Hybrid Semantic Web Service Matchmaker for OWL-S Services. *Web Semantics: Science, Servies and Agents on the World Wide Web*, 7(2):121–133, April 2009.

[43] Matthias Klusch and Patrick Kapahnke. Adaptive Signature-Based Semantic Selection of Services with OWLS-MX3. *Multiagent and Grid Systems*, 8(1):69–82, January 2012.

[44] Matthias Klusch and Patrick Kapahnke. The iSeM matchmaker: A flexible approach for adaptive hybrid semantic service selection. *Web Semantics: Science, Services and Agents on the World Wide Web*, 15:1–14, September 2012.

[45] Matthias Klusch, Patrick Kapahnke, and Ingo Zinnikus. Adaptive Hybrid Semantic Selection of SAWSDL Services with SAWSDL-MX2. *International Journal on Semantic Web and Information Systems*, 6(4):1–26, 2010.

[46] Matthias Klusch, Birgitta König-Ries, David Martin, Massimo Paolucci, Abraham Bernstein, Ulrich Lampe, Stefan Schulte, and Terry Payne. 5th International Semantic Service Selection Contest: Performance Evaluation of Semantic Service Matchmakers, Summary Report for 2012 Edition: http://www-ags.dfki.uni-sb.de/klusch/s3/html/2012.html.

[47] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6):60–67, November 2007.

[48] Ulrich Küster and Birgitta König-Ries. On the Empirical Evaluation of Semantic Web Service Approaches: Towards Common SWS Test Collections. In *Proceedings of the 2th IEEE International Conference on Semantic Computing (ICSC '08)*, pages 339–346. IEEE Computer Society, 2008.

[49] Chengpu Li, Xiaodong Liu, and Jessie Kennedy. Semantics-Based Component Repository: Current State of Arts and a Calculation Rating Factor-Based Framework. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC '08)*, pages 751–756. IEEE Computer Society Press, 2008.

[50] Min Liu, Weiming Shen, Qi Hao, and Junwei Yan. An Weighted Ontology-based Semantic Similarity Algorithm for Web Service. *Expert Systems with Applications*, 36(10):12480–12490, December 2009.

[51] Jim Luo, Bruce Montrose, Anya Kim, Amitabh Khashnobish, and Myong Kang. Adding OWL-S Support to the Existing UDDI Infrastructure. In *Proceedings of the IEEE International Conference on Web Services (ICWS '06)*, pages 153–162. IEEE Computer Society, 2006.

[52] Haresh Luthria and Fethi A. Rabhi. Service Oriented Computing in Practice - An Agenda for Research into the Factors Influencing the Organizational Adoption of Service Oriented Architectures. *Journal of Theoretical and Applied Electronic Commerce Research*, 4(1):39–56, April 2009.

[53] Yuanhua Lv and ChengXiang Zhai. When Documents Are Very Long, BM25 Fails! In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '11)*, pages 1103–1104. ACM, 2011.

[54] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[55] Anderson Marinho, Leonardo Murta, and Cláudia Werner. Extending a Software Component Repository to Provide Services. In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering (ICSR '09)*, pages 258–268. Springer Berlin Heidelberg, 2009.

[56] David Martin, Mark Burstein, Drew McDermott, Sheila McIlraith, Massimo Paolucci, Katia Sycara, Deborah L. McGuinness, Evren Sirin, and Naveen Srinivasan. Bringing Semantics to Web Services with OWL-S. *World Wide Web*, 10(3):243–277, September 2007.

[57] David Martin, Massimo Paolucci, and Matthias Wagner. Bringing Semantic Annotations to Web Services: OWL-S from the SAWSDL Perspective. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Conference on Semantic Web (ISWC '07/ASWC '07)*, pages 340–352. Springer Berlin Heidelberg, 2007.

[58] Nils Masuch, Benjamin Hirsch, Michael Burkhardt, Axel Heßler, and Sahin Albayrak. SeMa2: A Hybrid Semantic Service Matching Approach. In *Semantic Web Services*, pages 35–47. Springer Berlin Heidelberg, 2012.

[59] Douglas McIlroy. Mass Produced Software Components. In *Proceedings of the NATO Software Engineering Conference*, pages 138–155, 1968.

[60] Nilo Mitra and Yves Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition) (2007): http://www.w3.org/TR/2007/REC-soap12-part0-20070427.

[61] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Patterns for measuring performance-related QoS properties in service-oriented systems. In *Proceedings of the 17th Conference on Pattern Languages of Programs (PLoP '10)*, page 20. ACM, 2010.

[62] Soodeh Pakari, Esmaeel Kheirkhah, and Mehrdad Jalali. A Novel Approach: A Hybrid Semantic Matchmaker For Service Discovery In Service Oriented Architecture. *International Journal of Network Security & Its Applications (IJNSA)*, 6(1):37–48, January 2014.

[63] Dipasree Pal, Mandar Mitra, and Kalyankumar Datta. Improving Query Expansion Using WordNet. *Journal of the Association for Information Science and Technology*, 2014.

[64] Massimo Paolucci, Takahiro Kawamura, Terry Payne, and Katia P. Sycara. Importing the Semantic Web in UDDI. In *Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web (CAiSE '02/WES '02)*, pages 225–236. Springer London, 2002.

[65] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the First International Semantic Web Conference (ISWC 2002)*, pages 333–347. Springer, 2002.

[66] Michael P. Papazoglou and Dimitrios Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):24–28, October 2003.

[67] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. WordNet::Similarity – Measuring the Relatedness of Concepts. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI '04)*, pages 1024–1025. AAAI Press, 2004.

[68] José R. Pérez-Agüera, Javier Arroyo, Jane Greenberg, Joaquin Perez Iglesias, and Victor Fresno. Using BM25F for Semantic Search. In *Proceedings of the 3rd International Semantic Search Workshop (SEMSEARCH '10)*, pages 2:1–2:8. ACM, 2010.

[69] Martin Porter. An algorithm for suffix stripping. *Program: Electronic Library and Information Systems*, 14(3):130–137, 1980.

[70] Chris Preist. A Conceptual Architecture for Semantic Web Services. In *The Semantic Web — ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 395–409. Springer Berlin Heidelberg, 2004.

[71] Stephen Robertson. Understanding Inverse Document Frequency: On theoretical arguments for IDF. *Journal of Documentation*, 60(5):503–520, 2004.

[72] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *Overview of the Third Text REtrieval Conference (TREC-3)*, pages 109–126. National Institute of Standards and Technology (NIST), 1995.

[73] Stephen E. Robertson and Hugo Zaragoza. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, April 2009.

[74] Marta Sabou and Jeff Pan. Towards semantically enhanced Web service repositories. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):142–150, June 2007.

[75] Marco Luca Sbodio, David Martin, and Claude Moulin. Discovering Semantic Web Services Using SPARQL and Intelligent Agents. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):310–328, November 2010.

[76] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[77] Evren Sirin and Bijan Parsia. The OWL-S Java API. In *Proceedings of the Third International Semantic Web Conference (ISWC '04)*, 2004.

[78] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.

[79] Karen Spärck-Jones, Stephen Walker, and Stephen E. Robertson. A Probabilistic Model of Information Retrieval: Development and Comparative Experiments. *Information Processing and Management: an International Journal*, 36(6):779–808, November 2000.

[80] Michael Stonebraker. Stonebraker on NoSQL and enterprises. *Communications of the ACM*, 54(8):10–11, 2011.

[81] Ioan Toma, Flavio De Paoli, and Dieter Fensel. On Modelling Non-Functional Properties of Semantic Web Services. In *Handbook of Research on Service-Oriented Systems and Non-Functional Properties: Future Directions*, pages 61–85. IGI Global, 2012.

[82] Vassileios Tsetsos, Christos Anagnostopoulos, and Stathes Hadjiefthymiades. On the Evaluation of Semantic Web Service Matchmaking Systems. In *Proceedings of the Fourth IEEE European Conference on Web Services (ECOWS '06)*, pages 255–264. IEEE Computer Society, 2006.

[83] UDDI.org. UDDI Executive White Paper (2001): http://uddi.org/pubs/UDDI_Executive_White_Paper.pdf.

[84] Dengping Wei, Ting Wang, Ji Wang, and Abraham Bernstein. SAWSDL-iMatcher: A Customizable and Effective Semantic Web Service Matchmaker. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(4):402–417, December 2011.

[85] Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. *Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, October 1997.